Cereon

CDS 1.0
Assembler reference

Copyright © 2007-2008, Cybernetic Intelligence GmbH
All Rights Reserved

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Authors:
Andrei Kapustin

Revision history

Date	Comment
05 Dec 2007	Initial draft.
08 Jan 2008	Allowed `includebin` directive to be labelled.
13 Feb 2008	Allowed groups to be renamed.
15 Feb 2008	Allowed using ‘`*`’ as an operand to refer to the current location.
18 Feb 2008	All name/value properties will now consistently use ‘`:=`’.
28 July 2008	Added the `use` directive. Renamed include path environment variables `INCLUDE` and `INCLUDEBIN` to `CDS_1_0_INCLUDE` and `CDS_1_0_INCLUDEBIN` correspondingly.

1 Contents

2 Introduction

This document is a definitive guide into the Cereon Macro Assembler (henceforth referred to simply as an “assembler”).

The Cereon Macro Assembler is a typical member of its class, allowing the translation of programs written in Cereon Assembler Language into an object form to be linked and executed. However, it also provides some features not normally found in a common assembler, such as:

- Full support for ISO-10646, specifically including assembler sources.
- Emission of the NGOFF (Next Generation Object File Format) object modules. While allowing the user to choose from several different output formats is by no means new for an assembler, the Cereon Macro Assembler is geared towards emission of NGOFF object modules. Other formats (such as ELF) are also supported for legacy reasons.

3 The structure of the source program

The source program offered to the assembler is a text file, where each line can either be empty or contain an instruction, an assembler directive or a comment. Assembler always assumes the source to be composed from ISO-10646 characters; when this is not the case (for example, when the source file is an ISO-8859-1 – encoded text file, which is the most common form for all source programs), the source program is converted to ISO-10646 before being translated.

3.1 Elements of an assembler program

This chapter defines common elements that can appear in the assembler instructions and directives.

3.1.1 Source files

As far as the assembler is concerned, a source file is a text file containing the assembler program or part thereof. Such file can be created and edited by any text editor available on the platform where development takes place.

By default, assembler assumes that all source files use the character encoding that is "native" for the host platform (by far the most common character encoding used for writing programs in all languages is iso-8859-1). However, it is possible, when invoking the assembler, to specify that an alternative encoding (such as UTF-8 or even EBCDIC) shall be used to interpret the contents of the source files.

3.1.2 Source lines

An assembler program is a text file consisting of a sequence of logical source lines, where each logical source line can:

- Contain an assembler instruction or directive (possibly with comment).
- Contain a comment.
- Be empty.

In the simplest case, logical source lines correspond to physical lines contained in a text file; the platform-specific line separator (ASCII 10 on Unix, ASCII 13 10 on Windows, etc.) is used to determine physical source line boundaries; however, a single logical source line can also span several physical source lines if line continuations are used.

3.1.2.1 Line continuations

If the rightmost non-space character in a physical source line is a backslash '\', then the immediately following physical source line is considered to be a continuation of the previous physical source lines, both contributing to the same logical source line. The continuation line can itself end with a backslash, thus providing further line continuation. There is no limit on how long a sequence of line continuations can be.

When there are two consecutive physical source lines L_1 and L_2 such that L_2 immediately follows L_1 and L_1 ends with a backslash, the following rules are used to combine these two physical source lines into a logical source line:

- The tailing backslash in L_1 is discarded as well as all spaces that (optionally) follow it.
- All leading spaces in L_2 are discarded.
- L_2 is then appended to the end of L_1.

Although spaces are not usually significant in assembler programs, there is a context where spaces matter: within character and string constants. Therefore, be careful when breaking a long string constant into several continued lines.

3.1.2.2 TAB handling

Although it is recommended not to use the ASCII TAB character within assembler sources, it is still possible that some people will do so. The assembler processes TABs in the assembler sources using the following rules:

- TABs are processed independently for each physical (*not* logical) source line in left-to-right order.
- When a TAB character is discovered, it is replaced with a sufficient number of space characters to advance the position to the next tab stop. Tab stops are pre-designated positions at fixed intervals within the physical source line (that interval is also known as tab width). At least one space is always inserted.
- The default tab width is 8 characters (for historical reasons); however, it is possible, when invoking an assembler, to specify that a different tab width shall be used.

TABs are replaced with spaces before logical source lines are formed.

3.1.3 Comments

In any source line within an assembler program all characters starting from the leftmost semicolon character '`;`' until the end of that line are considered a comment. These characters appear in the source listing (if one is generated by the assembler), but have no other significance.

Comments are identified after the logical source lines have been prepared. In particular, if a physical source line where the comment starts has a continuation, that entire continuation will be part of the comment.

3.1.4 Identifiers

To an assembler, an identifier is a symbolic name that can be used to refer to an entity within the source program (e.g. a symbol, a section, an instruction mnemonic, a macro name, etc.)

In its simplest form, an identifier is written as a sequence of letters and/or digits that starts with a letter. Since the assembler is ISO-10646-aware:

- Any character belonging to an ISO-10646 category `Lu`, `Ll`, `Lt`, `Lm` or `Lo` is considered a "letter".
- Any character belonging to an ISO-10646 category `Nd`, `Nl` or `No` is considered a "digit".

In addition, the following characters are allowed anywhere within identifiers (each of these characters is treated as a special "letter"; in particular, a valid identifier can start with one of those):

- An underscore ('`_`', ISO-10646 code 95).
- A dollar sign ('`$`', ISO-10646 code 36).
- An "at" sign ('`@`', ISO-10646 code 64).
- A dot character ('`.`', ISO-10646 code 46).
- A sequence of two colons ('`::`', ISO-10646 code 58).

Therefore, all of the following identifiers are valid:

```
x
_x
$x
.text
x@y
a.b.c
x::y::z
::record.field
```

The assembler also allows the so-called "quoted" identifiers, which are written as an sequence of arbitrary characters and/or escape sequences enclosed in vertical bars:

```
|x|
|this is a valid identifier|
|ab\ncd|
|\||
```

Such "quoted" identifiers are parsed similarly to string constants; in particular, escape sequences can be used within "quoted" identifiers to denote special or nonprintable characters.

3.1.5 Keywords

A keyword is an identifier that, when used in a certain context, has a special meaning. Examples of keywords are instruction and directive mnemonics (such as `li.l` or `byte`), register names (such as `r0` or `$ip`) and operators (such as `shl`). Keywords are generally case-insensitive (i.e. instruction mnemonic `li.l` can also be written as `LI.L`, similarly, a register name such as `$ip` can also be written as `$IP`).

For a detailed list of instruction mnemonics and register names consult the "Cereon Architecture Reference Manual".

3.1.6 Constants

A constant represents its own value. Depending on the type of the value being represented, assembler allows integer, real and string constants to be written.

3.1.6.1 Integer constants

An integer constant represents an unsigned integer value. The maximum possible range for an integer constants is from 0 to 2^{64}-1; however, in some context the permitted range of an integer constant will be further reduced (for example, when used as an operand in a `byte` data definition directive, the permitted range of an integer constant is from 0 to 255).

There are several ways to specify an integer constant, described in the following chapters.

3.1.6.1.1 Decimal integer constants

A decimal integer constant is written as a sequence of decimal digits `0`..`9`. The following are all valid decimal integer constants:

```
0
1
18446744073709551615
```

Note that the latter is the largest permitted integer constant (2^{64}-1).

3.1.6.1.2 Hexadecimal integer constants

A hexadecimal integer constant is written as a sequence of hexadecimal digits `0`..`9`, `a`..`f` and `A`..`F` (where letters `A`..`F` represent hexadecimal digits 10..15 respectively) immediately preceded by a prefix `0x` or `0X`. The following are all valid hexadecimal integer constants:

```
0x0
0x1
0XFFFFFFFFFFFFFFFF
```

Note that the latter is the largest permitted integer constant (2^{64}-1).

3.1.6.1.3 Octal integer constants

An octal integer constant is written as a sequence of octal digits `0`..`7` immediately preceded by a prefix `0`. The following are all valid decimal octal constants:

```
00
01
01777777777777777777777
```

Note that the latter is the largest permitted integer constant (2^{64}-1). Note also that a plain “`0`”, when used as a constant, it treated as a decimal constant, hence the need to write an octal “zero” as “`00`”; in practice the distinction is negligible, as both “`0`” and “`00`” represent the same value.

3.1.6.1.4 Binary integer constants

A binary integer constant is written as a sequence of binary digits `0`..`1` immediately preceded by a prefix `0b` or `0B`. The following are all valid binary integer constants:

```
0b0
0b1
0B11111111111111111111111111111111111111111111111111
11111111111111
```

Note that the latter is the largest permitted integer constant (2^{64}-1).

3.1.6.1.5 Character integer constants

A character integer constant is written as an arbitrary printable character (except a single quote `'` or backslash `\`, see below) enclosed in single quotes. The value of such a constant is the ISO-10646 code of the specified character. The following are all valid character integer constants:

```
' ' (space, ISO-10646 code 32)
'A' (letter 'A', ISO-10646 code 65)
'_' (underscore, ISO-10646 code 95)
```

Alternatively, an escape sequence can be written within quotes instead of a single character. See the “Escape sequences” chapter below for more details.

3.1.6.2 Real constants

A real constant represents a 64-bit IEEE-854 floating point value; however, in some context this value would be converted to a shorter format (e.g to a 32-bit IEEE-754 floating point value when a real constant is used as an operand in a `float` directive, or to a 21-bit reduced-precision floating point value when a real constant is used as an operand in a `li.d` instruction).

There are several ways to specify a real constant, described in the following chapters.

3.1.6.2.1 Decimal real constants

A decimal real constant is written as a sequence of decimal digits that contains a decimal point, an exponent or both:

`0.0`	(value 0)
`12.34`	(value 12.34)
`1.2e-10`	(value 1.2×10^{-10})
`1e10`	(value 1×10^{10})

Note that if a real constant contains an exponent portion, then decimal point is optional. Note also that the “+” sign is optional for positive exponents, so “`1e2`” and “`1e+2`” both represent the same value.

Decimal real constants can only be used to represent finite IEEE-754 values (i.e. normalized and denormalized values and zeroes), but not special values, such as NaNs or infinities.

3.1.6.2.2 Hexadecimal real constants

A hexadecimal real constant is written as a sequence of hexadecimal digits `0`..`9`, `a`..`f` and `A`..`F` (where letters `A`..`F` represent hexadecimal digits 10..15 respectively) immediately preceded by a prefix `0x` or `0X` that contains a decimal point, an exponent or both. Unlike decimal real constants, the exponent base for hexadecimal real constants is 16.

```
0x0.0        (value 0)
0x1.8        (value 1 + 8×16^-1 = 1.5)
0xFF.FF      (value 255 + 255×16^-2 = 255.99609375)
0x1e2        (value 1×16^2 = 256)
```

Note that if a real constant contains an exponent portion, then decimal point is optional. Note also that the “+” sign is optional for positive exponents, so “`1e2`” and “`1e+2`” both represent the same value.

Unlike decimal real constants, hexadecimal real constants can be used to represent all possible IEEE-754 encodings, including special values, such as NaNs or infinities. For example, to represent a positive infinity one would write `0x0e+7FF`.

3.1.6.2.3 Octal real constants

An octal real constant is written as a sequence of octal digits `0..7` immediately preceded by a prefix `0` that contains a decimal point, an exponent or both. Unlike decimal real constants, the exponent base for octal real constants is 8.

```
00.0         (value 0)
01.4         (value 1 + 4×8^-1 = 1.5)
077.77       (value 63 + 63×8^-2 = 63.984375)
01e2         (value 1×8^2 = 64)
```

Note that if a real constant contains an exponent portion, then decimal point is optional. Note also that the “+” sign is optional for positive exponents, so “`1e2`” and “`1e+2`” both represent the same value.

Unlike decimal real constants, octal real constants can be used to represent all possible IEEE-754 encodings, including special values, such as NaNs or infinities. For example, to represent a positive infinity one would write `00e+3777`.

3.1.6.2.4 Binary real constants

A binary real constant is written as a sequence of a binary digits `0..1` immediately preceded by a prefix `0b` or `0B` that contains a decimal point, an exponent or both. Unlike decimal real constants, the exponent base for binary real constants is 2.

```
0b0.0        (value 0)
0b1.1        (value 1 + 1×2^-1 = 1.5)
0b11.11      (value 3 + 3×2^-2 = 3.75)
0b1e2        (value 1×2^2 = 4)
```

Note that if a real constant contains an exponent portion, then decimal point is optional. Note also that the “+” sign is optional for positive exponents, so “`1e2`” and “`1e+2`” both represent the same value.

Unlike decimal real constants, binary real constants can be used to represent all possible IEEE-754 encodings, including special values, such as NaNs or infinities. For example, to represent a positive infinity one would write `0b0e+11111111111`.

3.1.6.3 String constants

A string constant is written as an arbitrary sequence of printable characters (except for a double quote ‶ or backslash `\`) enclosed in double quotes:

`‶‶`	(empty string)
`‶a‶`	(a string containing one character)
`‶abcde‶`	(a string containing five characters)

Normally, a string can contain any ISO-10646 characters; however, in certain contexts the permitted range for string's characters may be further reduced (for example, when a string constant is used as an operand of an `ascii` directive, all string's characters must be value ASCII characters).

In addition to “normal” characters a string constant can include the so-called escape sequences, which represent special characters (such as nonprintable or control characters).

3.1.6.3.1 String constant concatenation

When several string constants appear immediately one after another, assembler treats them as a single string constant by concatenating their values. For example, all of the following forms represent the same string constant:

`‶a‶ ‶b‶ ‶c‶ ‶d‶ ‶e‶`	(5 fragments concatenated)
`‶ab‶ ‶cde‶`	(2 fragments concatenated)
`‶abcde‶`	(a string constant is not fragmented)

3.1.6.4 Escape sequences

An escape sequence is a sequence of characters hat starts with a backslash `\`. Escape sequences can occur within character integer constants, string constants and quoted identifiers; in all of these cases escape sequences are interpreted in the same manner. Note that, although an escape sequence is written as several characters, it always represents a single ISO-10646 character.

Depending on the form used, there are mnemonic, numeric and literal escape sequences.

3.1.6.4.1 Mnemonic escape sequences

Mnemonic escape sequences are used to represent the most frequently used nonprintable characters. The following table summarizes mnemonic escape sequences supported by the assembler:

Escape sequence	Special character	Value
`\a`	Alarm	07_{16}
`\b`	Backspace	08_{16}

`\e`	Escape	`1B`$_{16}$
`\f`	Form feed	`0C`$_{16}$
`\n`	New line	`0A`$_{16}$
`\r`	Carriage return	`0D`$_{16}$
`\t`	Horizontal tabulation	`09`$_{16}$
`\v`	Vertical tabulation	`0B`$_{16}$

3.1.6.4.2 Numeric escape sequences

A numeric escape sequence can be used top specify an arbitrary ISO-10646 code point. It is written in either octal or hexadecimal form (note that, unlike integer constants, numeric escape sequences cannot be written in decimal or binary notation).

An octal escape sequence is written as a backslash `\` followed by up to 11 octal digits:

`\12`	(value `0A`$_{16}$, same as `\n`)
`\012`	(value `0A`$_{16}$, same as `\n`)
`\37777777777`	(value `FFFFFFFF`$_{16}$)

A hexadecimal escape sequence is written as a prefix `\x` or `\X` followed by up to 8 hexadecimal digits `0..9`, `a..f` and `A..F` (where letters `A..F` represent hexadecimal digits 10..15 respectively):

`\xA`	(value `0A`$_{16}$, same as `\n`)
`\x0A`	(value `0A`$_{16}$, same as `\n`)
`\xFFFFFFFF`	(value `FFFFFFFF`$_{16}$)

3.1.6.4.3 Literal escape sequences

A literal escape sequence is written as a backslash `\` followed by an arbitrary printable character. Such an escape sequence represents the character that follows the backslash Among other things, this allows embedding the double quote and backslash characters into string constants:

`"a\\b\"c"`	(value `a\b"c`)

3.2 Expressions

In an assembler program an expression is a specification of how an integer, real or string value shall be calculated. An expression, when evaluated, always yields a value of a specific type (i.e. integer, real or string) and, in general, can be used in any place within an assembler instruction or directive where a value of the corresponding type is expected.

The simplest form of an expression is just a single constant – such expression produces the value represented by that constant. However, assembler also allows you to construct arbitrarily complex expressions if the value you want cannot be represented by a constant but requires some calculation instead.

In a general form, an expression is a mixture of:

- Operands, which specify the values used in calculating the expression's result.
- Operators, which specify the rules used for result calculation.
- Function calls, which provide a shorthand notation for commonly used calculations.

The following sections outline all of these categories.

3.2.1 Operands

An operand is a single value participating in an expression evaluation. Assembler allows three types of operands – constant operands, named operands and properties.

3.2.1.1 Constant operands

A constant operand is an integer, real or string constant. The type and value of such operand is the same as the type and value of the corresponding constant.

3.2.1.2 Named operands

A named operand is an identifier that is a segment, section or symbol name. The type of such operand is always a 64-bit integer; the value of such operand is the address of the corresponding segment, section or symbol.

3.2.1.3 Properties

A property operand is written as `<name>``` ` ```<property>`, where:

- `<name>` is an identifier that is a segment, section or symbol name.
- The back quote (ISO 10646 code 96) is used as a name/property separator. Any number of spaces is permitted both before and after the separator; however, it is recommended that no spaces are used there.
- `<property>` is an identifier referring to the property of the `<name>` that is to be used. Property names are case-insensitive, so ``x`size`` can also be written as ``x`Size`` or ``x`SIZE``.

The following table summarizes available properties:

Property	Applies to	Type	Value
`size`	segments sections symbols	integer	The size of the corresponding segment, section or symbol.
`start`	segments sections symbols	integer	The address of the corresponding segment, section or symbol. When used in expression, ``X`start`` yields the same value as `X`.
`end`	segments sections symbols	integer	The end address of the corresponding segment, section or symbol. ``X`end`` is always the same as ``X`start + X`size``.
`defined`	all names	integer	For any name `X`, ``X`defined`` is 1 iff X is a name of a section, segment, symbol, macroprocessor variable, macroprocessor function or macro known to an assembler at

			the point where the property is being used, 0 otherwise.

3.2.1.4 Current location

When a single asterisk '`*`' is used as an operand, it refers to the offset of the current item within its section.

3.2.2 Operators

Operators use infix or prefix syntax to specify calculations that shall be performed. Most operators are written using special characters (such as + or `*`); however, there are also operators that are written using dedicated keywords (such as `shl`).

It shall be noted that many operators are overloaded. For example, the + operator can be used to:

- Add two integer values, yielding an integer result.
- Add two real values, yielding a real result.
- Concatenate two string values, yielding a string result.

3.2.2.1 Integer operators

The following integer operators are provided:

Operator	Signature	Description

`A + B`	`A : integer`
	`B : integer`
	`result : integer`
		Adds two signed integer values, yielding an integer result.

`A - B`	`A : integer`
	`B : integer`
	`result : integer`
		Subtracts two signed integer values, yielding an integer result.

`A * B`	`A : integer`
	`B : integer`
	`result : integer`
		Multiplies two signed integer values, yielding an integer result.

`A / B`	`A : integer`
	`B : integer`
	`result : integer`
		Divides a signed integer value by a signed integer value, yielding an integer result. If `B = 0`, an error results.

`A % B`	`A : integer`
	`B : integer`
	`result : integer`
		Divides a signed integer value by a signed integer value, yielding an integer remainder. If `B = 0`, an error results.

`A shl B`	`A : integer`
	`B : integer`
	`result : integer`
		Shifts an unsigned integer value `A` left, using `B` as an unsigned integer shift counter, yielding an integer result.

`A shr B`	`A : integer`	Shifts an unsigned integer

	`B : integer`
	`result : integer`
		value `A` right, using `B` as an unsigned integer shift counter, yielding an integer result.

`A asl B`	`A : integer`
	`B : integer`
	`result : integer`
		Shifts a signed integer value `A` left, using `B` as an unsigned integer shift counter, yielding an integer result.

`A asr B`	`A : integer`
	`B : integer`
	`result : integer`
		Shifts a signed integer value `A` right, using `B` as an unsigned integer shift counter, yielding an integer result.

`A and B`	`A : integer`
	`B : integer`
	`result : integer`
		Performs the bitwise `AND` of two integer values, yielding an integer result.

`A or B`	`A : integer`
	`B : integer`
	`result : integer`
		Performs the bitwise `OR` of two integer values, yielding an integer result.

`A xor B`	`A : integer`
	`B : integer`
	`result : integer`
		Performs the bitwise `Exclusive OR` of two integer values, yielding an integer result.

`A implies B`	`A : integer`
	`B : integer`
	`result : integer`
		Performs the bitwise `implication` of two integer values, yielding an integer result.

`A and then B`	`A : integer`
	`B : integer`
	`result : integer`
		If `A = 0`, then yields 0. Otherwise evaluates `B`; if `B = 0` then yields 0; otherwise yields 1.

`A or else B`	`A : integer`
	`B : integer`
	`result : integer`
		If `A <> 0`, then yields 1. Otherwise evaluates `B`; if `B <> 0` then yields 1; otherwise yields 0.

`A implies then B`	`A : integer`
	`B : integer`
	`result : integer`
		If `A = 0`, then yields 1. Otherwise evaluates `B`; if `B <> 0` then yields 1; otherwise yields 0.

`-A`	`A : integer`
	`B : integer`
	`result : integer`
		Yields the 2's complement of `A`.

`not A`	`A : integer`
	`B : integer`
	`result : integer`
		Yields the 1's complement of `A`.

`A = B`	`A : integer`
	`B : integer`
		Yields 1 if `A = B`; otherwise yields 0.

	`result : integer`	

`A <> B`	`A : integer`
	`B : integer`
	`result : integer`
		Yields 0 if `A = B`; otherwise yields 1.

`A < B`	`A : integer`
	`B : integer`
	`result : integer`
		Yields 1 if `A < B`; otherwise yields 0. Operands are compared as signed integer values.

`A <= B`	`A : integer`
	`B : integer`
	`result : integer`
		Yields 1 if `A <= B`; otherwise yields 0. Operands are compared as signed integer values.

`A > B`	`A : integer`
	`B : integer`
	`result : integer`
		Yields 1 if `A > B`; otherwise yields 0. Operands are compared as signed integer values.

`A >= B`	`A : integer`
	`B : integer`
	`result : integer`
		Yields 1 if `A >= B`; otherwise yields 0. Operands are compared as signed integer values.

3.2.2.2 Real operators

The following real operators are provided:

Operator	Signature	Description

`A + B`	`A : real`
	`B : real`
	`result : real`
		Adds two real values, yielding a real result.

`A - B`	`A : real`
	`B : real`
	`result : real`
		Subtracts two real values, yielding a real result.

`A * B`	`A : real`
	`B : real`
	`result : real`
		Multiplies two real values, yielding a real result.

`A / B`	`A : real`
	`B : real`
	`result : real`
		Divides a real value by a real value, yielding a real result.

`-A`	`A : real`
	`B : real`
	`result : real`
		Yields the `A` negated.

`A = B`	`A : real`
	`B : real`
	`result : integer`
		Yields 1 if `A = B`; otherwise yields 0.

`A <> B`	`A : real`
	`B : real`
	`result : integer`
		Yields 0 if `A = B`; otherwise yields 1.

`A < B`	`A : real`
	`B : real`
		Yields 1 if `A < B`; otherwise yields 0.

	`result : integer`	

`A <= B`	`A : real`
	`B : real`
	`result : integer`
		Yields 1 if `A <= B`; otherwise yields 0.

`A > B`	`A : real`
	`B : real`
	`result : integer`
		Yields 1 if `A > B`; otherwise yields 0.

`A >= B`	`A : real`
	`B : real`
	`result : integer`
		Yields 1 if `A >= B`; otherwise yields 0.

3.2.2.3 String operators

The following real operators are provided:

Operator	Signature	Description

`A + B`	`A : string`
	`B : string`
	`result : string`
		Concatenates two string values, yielding a string result.

`A = B`	`A : string`
	`B : string`
	`result : integer`
		Yields 1 if `A = B`; otherwise yields 0.

`A <> B`	`A : string`
	`B : string`
	`result : integer`
		Yields 0 if `A = B`; otherwise yields 1.

`A < B`	`A : string`
	`B : string`
	`result : integer`
		Yields 1 if `A < B`; otherwise yields 0. Strings are compared lexicographically; character case is significant.

`A <= B`	`A : string`
	`B : string`
	`result : integer`
		Yields 1 if `A <= B`; otherwise yields 0. Strings are compared lexicographically; character case is significant.

`A > B`	`A : string`
	`B : string`
	`result : integer`
		Yields 1 if `A > B`; otherwise yields 0. Strings are compared lexicographically; character case is significant.

`A >= B`	`A : string`
	`B : string`
	`result : integer`
		Yields 1 if `A >= B`; otherwise yields 0. Strings are compared lexicographically; character case is significant.

3.2.3 Functions

A function call is written as:

```
<function name>(<operand1>, … ,<operandn>)
```

where:

- `<function name>` is an identifier referring to the function that is being called.
- Each `<operandi>` is an expression representing one of the function's arguments.

Note that when a function without arguments is called, the brackets must still follow the function name, only there will be no operands within (e.g. `now()`).

The following table summarizes all functions provided by the assembler (note that all function names are case-sensitive):

Function	Signature	Description

`length(A)`	`A : string`
	`result : integer`
		The number of characters in the string `A` (i.e. the length of the string).

`left(A, B)`	`A : string`
	`B : integer`
	`result : string`
		The leftmost `B` characters of `A` or, if `B > length(A)`, the entire `A`. If `B < 0` then an empty string.

`right(A, B)`	`A : string`
	`B : integer`
	`result : string`
		The rightmost `B` characters of `A` or, if `B > length(A)`, the entire `A`. If `B < 0` then an empty string.

`mid(A, B)`	`A : string`
	`B : integer`
	`result : string`
		The substring of `A` starting with a 0-based position `B` and extending until the end of the string. If `B < 0`, then the entire `A`. If `B > length(A)`, then an empty string.

`mid(A, B, C)`	`A : string`
	`B : integer`
	`C : integer`
	`result : string`
		The substring of `A` starting with a 0-based position `B` and extending until the 0-based position `C` (not including the character at position `C` itself). If `B < 0`, then behaves as is `B = 0`. If `B > length(A)`, then an empty string. If `C < B`, then an empty string. If `C > length(A)`, then behaves as if `C = length(A)`.

`int(A)`	`A : real`
	`result : integer`
		Returns a signed 64-bit integer value that is the closest representable value to `A`.

`int(A)`	`A : string`
	`result :`
		Extracts the longest left-aligned string representation of an integer value

	`integer`	from the string `A`, converts it to an integer and returns the result. Permitted representations are those allowed by the assembler syntax for integer constants.
`real(A)`	`A : integer` `result : real`	Returns a 64-bit real value that is the closest representable value to A.
`real(A)`	`A : string` `result : real`	Extracts the longest left-aligned string representation of a real value from the string `A`, converts it to a real and returns the result. Permitted representations are those allowed by the assembler syntax for real constants.
`string(A)`	`A : integer` `result : string`	Creates and returns a decimal integer string representation of the specified value.
`string(A)`	`A : real` `result : string`	Creates and returns a decimal integer string representation of the specified value.
`code(A)`	`A : string` `result : integer`	Returns the ISO-10646 code of the first (leftmost) character of the specified string. If the string is empty, returns 0.
`char(A)`	`A : integer` `result : string`	Returns a string of length 1 whose one and only character has ISO-10646 code `A`. If `A` is outside the range [0..23^{2}), only its lower 32 bits are used.
`lower(A)`	`A : string` `result : string`	Returns the argument string where all characters have been converted to lower case.
`upper(A)`	`A : string` `result : string`	Returns the argument string where all characters have been converted to upper case.

3.2.4 Special expressions

In addition to operators, operands and function calls, there are several other syntactic constructs that can also be used in expressions.

3.2.4.1 Using brackets to change evaluation order

Normally, the order in which expression is evaluated is determined by the precedence and associativity of operators used therein. However, brackets can be used in mathematical sense to change the evaluation order; any sub-expression within brackets is evaluated before its value is used as an operand (e.g. `2 + 2 * 2 = 6`, but `(2 + 2) * 2 = 8`).

3.2.4.2 The conditional expression

The conditional expression has the form:

```
A ? B : C
```

Where:

- `A` is an expression that yields an integer.
- `B` and `C` are expressions that can yield integer, real or string value; however, both must yield the value of the same type.

When a conditional expression is evaluated:

- `A` is evaluated first.
- If `A <> 0`, then `B` is evaluated and the value yielded by `B` is also yielded by the entire conditional expression.
- If `A = 0`, then `C` is evaluated and the value yielded by `C` is also yielded by the entire conditional expression.

Note that only one of `B` or `C` will be evaluated (depending on the value yielded by `A`). This allows writing conditional expressions like `(Y = 0) ? 0 : (X / Y)`, which guards against division-by-zero.

3.2.5 Operator precedence and evaluation order

The following table summarizes the precedence and evaluation order of operators. Operators are listed in order of decreasing precedence.

Operators	**Evaluation order**
`f(`e_1`, …, `e_n`)`	Inside out

`+ (unary)`
`- (unary)`
`not (unary)`
	Inside out

`* / %`	Left to right

`+ (binary)`
`- (binary)`
	Left to right

`shl`
`shr`
`asl`
`asr`
	Left to right

`=`
`<>`
`<`
`<=`
`>`
`>=`
	Left to right

`and`
`and then`
`or`
`or else`
`xor`
	Left to right

`implies`
`implies then`	

`?:`	Inside out

3.2.6 Constant expressions

A constant expression is an expression whose value can be determined at compile time. Any expression where all operands are constants is a constant expression; in addition, a constant expression may:

- Use named operands if these operands have constant values (for example, a constant value may be assigned to a symbol by an `equ` directive, which will make that symbol a compile-time constant and allow its use in constant expressions).
- Use properties which can be determined at compile time (for example, if `F` is a name of the procedure defined in the current compilation unit, then ``F`size`` can be used in a constant expression as an operand, because the size of the function is known to the assembler at compile time. However, if `S` is a name of a section or segment, then ``S`size`` is not a compile-time constant, as sections and segments are constructed by linker, yet ``S`start`` may be a compile-time constant is the segment or section in question is assigned an explicit address in an assembler source).

All expressions that yield real or string results must be constant expressions.

3.2.7 Deferred expressions

A deferred expression is an expression which cannot be fully evaluated by the assembler at compile time. Typical examples are expressions whose operands are segment or section sizes, addresses and/or sizes of external symbols, etc.

Only expressions that yield integer results may be deferred. In addition, all sub-expressions of these deferred expressions that yield real or string results must be constant expressions. This is just another way of saying that:

- Deferred expressions are evaluated by linker (whether static linker or dynamic linker is used depends on the linkage model) instead of the assembler.
- The linker (either static or dynamic) can only evaluate integer expressions.

There are few contexts where deferred expressions are not permitted and constant expressions must be used instead (such as specifying a segment start address); these contexts mainly occur in assembler directives. All such contexts are explicitly specified as requiring constant expressions in the chapters of this document describing corresponding directives. Assembler instructions, on the other hand, allow deferred expressions wherever a value is needed (i.e. as immediate operands, memory address offsets, bit field sizes and shift counts, etc.)

3.3 The memory model

Any assembler program written for a Cereon target (indeed, any program written in any language for a Cereon target) is always written for a flat 64-bit memory model.

Whether physical or virtual 64-bit address space is used is determined by the properties of the platform where that program is run.

However, when writing a program for a Cereon platform, the programmer operates in terms of segments and sections.

3.3.1 Segments

In its basics, a segment is a continuous block of memory within the 64-bit address space, which can contain program code, data, heap, stack, etc. The key characteristic of a segment is in that the entire address range of a segment has the same access privileges and is managed by the operating system as one entity. For example, if some word within a segment has an “execute” permission (i.e. can be executed as an instruction), then the whole segment has an “execute” permission.

When writing an assembler program, you can specify what segments exist, as well as access permissions assigned to these segments as well as their other properties (for example, if you want a segment to start at a specific address within a 64-bit address space, you can specify that too, etc.)

Note, however, that not all output formats permit segments to be defined in an object file. For example, when compiling an assembler language program into an ELF object module, you cannot define any segments in the assembler source, only sections. In this case the linker is responsible for combining sections into segments (based on section names, section access permissions, a separate memory map definition file fed to the linker, etc.)

3.3.2 Sections

To a programmer, a section is a block of code, data or uninitialized memory that is guaranteed to remain continuous when the program is being executed. Sections are assigned to segments based either on segment definitions provided in the assembler source or on whatever assignments the linker makes. Note that, although some object formats (notably NGOFF) permit assigning sections to segments directly in the assembler source, it does not mean that all sections must be so assigned. Indeed, it is still possible to write an assembly language program where some (or even all) sections are not assigned to segments and let the linker do the work. This, in particular, allows writing assembler language programs that can be translated into any supported object file format – just do not define any segments in the assembler source.

The most important characteristic of a section is its access permissions, which specifies whether the entire contents of a section can be read as data, written to and/or executed. Other characteristics of a section may include section name, section address, maximum size the section is permitted to grow to, etc.

3.3.3 Naming and joining

Both sections and segments can be assigned optional names.

When the linker processes object files to create executable image, all sections defined in all input object modules that have the same name are assumed to contribute to the same section (which is produced by either concatenation or overlaying of all contributing sections). Similarly, all segments defined in all input object modules that have the same name contribute to the same segment. Naturally, when sections (or segments) are joined, their attributes must not conflict.

The special case are sections and segments that do not have a name. Such sections and segments are referred to as “anonymous” sections and segments and are never joined with each other or any other named section or segment.

3.3.4 Groups

A group is a set of sections that must be treated as a unit. Specifically, when one of the sections in a group is included into the linked image, all group members must be included.

Some object formats (such as NGOFF) allow the same section to be a member of more than one group. In this case, if the section is included into the linked image, then all sections belonging to all groups of which the original section is a member are included as well.

3.3.5 Supported memory models

Cereon applications (including those written in assembler language) can utilize one of the several memory models. Depending on what memory model is chosen, the program may be limited in what areas of a 64-bit address space it can access.

The following sections describe memory models supported by the assembler.

3.3.5.1 ILP64

The name of this memory model is an abbreviation from “Integer, Long integer and Pointer are all 64-bit”. In effect, this means that:

- The program can access the entire 64-bit address space (because 64-bit pointers are used).
- The program can operate with 64-bit integer data (because 64-bit long integers are used).
- When operating with integer data, the program uses 64-bit integers for preference.

This memory model offers maximum flexibility, as it uses the entire range of Cereon features.

3.3.5.2 LP64

The name of this memory model is an abbreviation from “Integer is 32-bit, Long integer and Pointer are both 64-bit”. In effect, this means that:

- The program can access the entire 64-bit address space (because 64-bit pointers are used).
- The program can operate with 64-bit integer data (because 64-bit long integers are used).

- When operating with integer data, the program uses 32-bit integers for preference. In memory, these integer values will be stored as 32-bit words, which will then be sign- or zero-extended when loaded into a register.

This memory model may result in smaller programs, as most integer values will happily fit within 32 bits. The efficiency is not sacrificed, as the Cereon architecture has been specifically designed to be as efficient working with 32-bit data as with 64-bit data. The downside is in development and maintenance cost, as the programmer must then explicitly decide which variables may or may not fit into 32-bits; if the range of one of those 32-bit variables increases later on, significant pieces of code may need to be rewritten.

3.3.5.3 IP32

The name of this memory model is an abbreviation from "Integer and Pointer are both 32-bit, but 64-bit Long integer type is also provided". In effect, this means that:

- The program can access only the lower 4GB of the entire 64-bit address space (because 32-bit pointers are used). In memory, pointers will be stored as 32-bit words, which will then be zero-extended when loaded into a register.
- The program can operate with 64-bit integer data (because 64-bit long integers are used).
- When operating with integer data, the program uses 32-bit integers for preference. In memory, these integer values will be stored as 32-bit words, which will then be sign- or zero-extended when loaded into a register.

This memory model is best suited for porting 32-bit applications to Cereon platforms. Many of these applications are written with implicit assumptions that:

- Integer values are 32 bits long, and
- Conversion between pointers and integers is lossless both ways.

3.3.5.4 Compatibility between memory models

In general, it is dangerous to link together object modules produced for different memory models, as the resulting image will, most likely, not work correctly.

However, in case you absolutely must do this, remember the following compatibility chain:

```
IP32 → LP64 → ILP64
```

Where the arrow $M_1 \rightarrow M_2$ means that:

- It may be safe for an object module that uses memory model M_1 to call a procedure that has been compiled for memory model M_2.
- It may be safe for an object module that uses memory model M_1 to use a variable defined in an object module that has been compiled for memory model M_2.

Note that the reverse (i.e. `ILP64` procedure calling an `IP32` procedure) is guaranteed to be unsafe. Therefore, an `IP32` procedure may not call an `ILP64` procedure if the latter requires a callback (as the callback will also be an `IP32` procedure, which should not be called from `IPL64` code).

Therefore, to avoid complications – don't mix memory models. The assembler allows you to specify what memory model a particular assembler source is written for (default is `ILP64`, as this is the Cereon native memory model), and the linker will normally complain when you try to mix several memory models into the same executable (unless you silence it with a dedicated command line option).

4 Directives

This section describes directives implemented by the Cereon assembler. Unlike instructions (where each instruction translates directly into a single 32-bit Cereon instruction), directives are used to:

- Define data (both initialized and uninitialized).
- Specify the program's memory model (segmentation, sections, alignment, etc.)
- Control the translation process (by performing macro expansion, conditional compilation, issuing messages, etc.)

The following sections describe assembler directives by their functionality areas.

4.1 Memory model directives

These directives specify the memory model that an assembler language program will use, as well as defining program's sections and segments.

4.1.1 segment…end segment

This directive has the following form:

```
[<name>:] segment <properties>
   .   .   .
[<name>:] end segment
```

In the simplest case, everything between the `segment` and `end segment` directives belongs to the specified segment (situation may be more complicated if segment stacking is used, see below). Depending on whether the segment is named or not, one of two choices is possible:

- If the segment is named, then the `<name>` used in the `end segment` directive must match the `<name>` used in the `segment` directive. The corresponding named segment will appear in an object module and may be joined with other segments with the same name by linker.
- If the segment is not named, then both `segment` and `end segment` directives must not specify the `<name>` portion. In this case a new, anonymous, segment is created and placed into an object module; this anonymous segment will not be joined with any other segment by the linker.

4.1.1.1 Segment stacking

A segment stacking occurs when the `segment … end segment` directives appear within another pair of `segment … end segment` directives, as in:

```
.text:  segment
   .   .   .
.data:  segment
   .   .   .
.data:  end segment
   .   .   .
```

```
.text:  end segment
```

In this case, the inner `segment` directive temporarily suspends the enclosing segment and starts a new one. When the inner `end segment` directive is processed, the inner segment is closed and the outer segment is resumed. The behaviour is as if the inner `segment` directive saved the "current" segment on a stack and the inner `end segment` directive restored it (hence the test "segment stacking"). `Segment` … `end segment` directives can be nested to any depth; however, it is important to realize that all segments are equal regardless of how or where they are defined.

It is important to ensure that the proper nesting is observed; for example, the following is an error:

```
.text:  segment
   .   .   .
.data:  segment
   .   .   .
.text:  end segment
   .   .   .
.data:  end segment
```

4.1.1.2 Specifying segment properties

By default, segments declared in an assembler program are assigned the minimal set of properties. More often than not this is not enough. Typically, as a programmer you need to specify at least some of the following properties of a segment, such as:

- The access permissions (i.e. whether the contents of the segment can be read as data, written to or executed as instructions).
- The address of the segment (which is frequently known at compile-time if the segment contains ROMable code).
- The required segment alignment.
- The maximum size to which the segment can grow (this can ensure that linker will not create joined segments that are too large to fit into required memory areas).
- The memory model used by the segment.

To do this, the `segment` directive allows you to specify segment properties. Each segment property is specified by a single operand of a `segment` directive; use as many operands as you need and separate them by commas, as in:

```
.text:  segment code, align := 4, model := ip32
```

The following sections describe permitted operands of the `segment` directive in more details. Note that all keywords referring to segment properties (such as `read`, `align`, etc.) are case-insensitive.

4.1.1.2.1 read, write and execute

These properties set individual access permissions for the segment in question. They are written as one of:

```
read
write
execute
```

When used as an operand in a `segment` directive, these properties set segment's read, write and execute permissions correspondingly.

4.1.1.2.2 code, data and const

These properties set combined access permissions for the segment in question, based on the most commonly needed access permissions for different types of contents. They are written as one of:

```
code
data
const
```

When used as an operand in a `segment` directive, these properties set segment's permissions as follows:

- `code` sets read and execute permission.
- `data` sets read and write permissions.
- `const` sets read permission.

4.1.1.2.3 align

This property specifies the alignment requirement of a segment. It is written as:

```
align := <alignment>
```

where `<alignment>` is a constant expression yielding an integer value representing the alignment boundary, which must be a power of 2. For example, the directive

```
.text:  segment code, align := 4
```

specifies that `.text` is a code segment (i.e. contains executable instructions) and is aligned at a 4-byte boundary (since all instructions must be naturally aligned).

Any number of spaces is allowed before and after the '=' operator. If the alignment of a segment is not explicitly specified, the segment does not have alignment requirement (i.e. it can start at any address).

4.1.1.2.4 address

This property specifies the constant address of a segment. It is written as:

```
address := <address>
```

where `<address>` is a constant expression yielding an integer value representing the segment's address. For example, the directive

```
.text:  segment code, address := 0xFFFFFFFFFFF00000
```

specifies that `.text` is a code segment (i.e. contains executable instructions) and is located at address `0xFFFFFFFFFFF00000` (which is, probably, the ROM).

Any number of spaces is allowed before and after the '`:=`' operator. If the address of a segment is not explicitly specified, the segment can start at any address (as determined by linker and/or loader).

4.1.1.2.5 max

This property specifies the maximum size the segment can grow to when the linker performs segment joining. It is written as:

```
        max := <size>
```

where `<size>` is a constant expression yielding an integer value representing the segment's maximum permitted size. For example, the directive

```
.text:  segment code, max := 65536
```

specifies that `.text` is a code segment (i.e. contains executable instructions) and is guaranteed not to exceed 64K in size.

Any number of spaces is allowed before and after the '`:=`' operator. If the maximum size of a segment is not explicitly specified, the segment can grow to any size.

4.1.1.2.6 private, shared and thread

These properties specify how the memory is allocated for the segment. They are mutually exclusive and are written as one of:

```
        private
        shared
        thread
```

Depending on which property was used, the following memory allocation strategy will be used for the segment:

- `private` segments are allocated per process. If several instances of the same application are running, each instance will have its own, private, copy of the segment. This is the default allocation mode for global static data.
- `shared` segments are allocated once per application. If several instances of the same application are running, they will all have shared access to the same copy of a `shared` segment. In effect, shared segments provide statically allocated shared memory. Code segments will be frequently marked as `shared`, since there is no need to keep an identical copy of a program's code replicated in each instance of that program that is running.
- `thread` segments represent the other extreme – in a multithreaded application each thread will have its own, private, copy of a `thread` segment.

This permits using `thread` segments as statically allocated thread-local storage.

If the memory allocation mode for a given segment is not specified explicitly, `private` is assumed.

4.1.1.2.7 preload, loadondemand and noload

These properties specify how the OS loader brings segments into the memory image of a process. They are mutually exclusive and are written as one of:

```
preload
loadondemand
noload
```

Depending on which property was used, the following strategy will be used for the segment:

- `preload` segments are brought into memory when an application is started. By the time an application begins execution, all its `preload` segments are in memory.
- `loadondemand` segments are not loaded when an application starts. However, the OS keeps track of them; any attempt to use one of these segments is intercepted by OS, which then loads the segment. These segments are best suited for information that is not normally needed for the program execution, but may still be required in some circumstances (such as exception handlers or debug information).
- `noload` segments are never automatically loaded into the process' memory image. If an application needs to use one of these segments, it must explicitly ask OS to load it. These segments are best suited for information that is not needed during program execution, but may still be required in some unusual circumstances.

If the segment loading mode for a given segment is not specified explicitly, `preload` is assumed.

Note that when an assembler-language program is written to run on a bare hardware (i.e. without OS support), only `preload` segments will be part of that program's memory image.

4.1.1.2.8 model

This property specifies what memory model the segment conforms to. It is written as:

```
model := {ilp64|lp64|ip32}
```

When a segment conforms to a specific memory model, all sections and symbols within that segment must conform to the same memory model.

Any number of spaces is allowed before and after the '`:=`' operator. If the memory model of a segment is not explicitly specified, the current default memory model is assumed for that segment (as specified by the `model` directive, see below). Memory model names (`ilp64`, `lp64` and `ip32`) are case-insensitive.

4.1.2 section…end section

This directive has the following form:

```
[<name>:] section <properties>
   .   .   .
[<name>:] end section
```

In the simplest case, everything between the `section` and `end section` directives belongs to the specified section (situation may be more complicated if section stacking is used, see below). Depending on whether the section is named or not, one of two choices is possible:

- If the section is named, then the `<name>` used in the `end section` directive must match the `<name>` used in the `section` directive. The corresponding named section will appear in an object module and may be joined with other sections with the same name by linker.
- If the section is not named, then both `section` and `end section` directives must not specify the `<name>` portion. In this case a new, anonymous, section is created and placed into an object module; this anonymous section will not be joined with any other section by the linker.

4.1.2.1 Assigning sections to segments

Depending on whether the `section … end section` directive is nested within a `segment … end segment` directive, two choices are possible.

If this is the case, then the section is assigned to the specified segment (note that assignment of sections to segments is incompatible with some object file formats produces by the assembler, such as ELF). If segment stacking is in effect, the section is assigned to the innermost enclosing segment. Such sections are called bound sections, because it is always known which segment they will end up assigned to.

If the `section … end section` directive is not nested within any `segment … end segment` directives, then the section is not assigned to any segment. Such sections are called roaming sections, as it is up to linker (whether static or dynamic) to decide which segment these sections will ultimately end up in.

4.1.2.2 Section stacking

A section stacking occurs when the `section … end section` directives appear within another pair of `section … end section` directives, as in:

```
.CODE:  section
   .   .   .
.CONST: section
   .   .   .
.CONST: end section
   .   .   .
```

```
.CODE:  end section
```

In this case, the inner `section` directive temporarily suspends the enclosing section and starts a new one. When the inner `end section` directive is processed, the inner section is closed and the outer section is resumed. The behaviour is as if the inner `section` directive saved the “current” section on a stack and the inner `end section` directive restored it (hence the test “section stacking”). `Section … end section` directives can be nested to any depth; however, it is important to realize that all sections are equal regardless of how or where they are defined.

It is important to ensure that the proper nesting is observed; for example, the following is an error:

```
.CODE:  section
   .   .   .
.CONST: section
   .   .   .
.CODE:  end section
   .   .   .
.CONST: end section
```

In addition, the following restrictions must be observed:

- It is not permitted to stack a roaming section inside a bound section.
- It is not permitted to stack a bound section inside a roaming section.
- It is not permitted to stack a bound section inside another bound section if the two are bound to different segments.

While these rules may seem somewhat complicated, in reality there is a simple interpretation of them – section stacking is not permitted to affect segment stacking.

4.1.2.3 Specifying section properties

By default, sections declared in an assembler program are assigned the minimal set of properties. More often than not this is not enough. Typically, as a programmer you need to specify at least some of the following properties of a section, such as:

- The access permissions (i.e. whether the contents of the section can be read as data, written to or executed as instructions).
- The address of the section (which is frequently known at compile-time if the section contains ROMable code).
- The required section alignment.
- The maximum size to which the section can grow (this can ensure that linker will not create joined section that are too large to fit into required memory areas).
- The memory model used by the section.
- The combination mode for the section if several object modules contain definitions of the same section.

While some of these properties can be safely inferred (for example, sections inherit access permissions and memory model from the segment to which they are assigned), there are still situations where:

- Properties inherited from the containing segment are not what the programmer wants.
- A section is a roaming section, which cannot inherit anything from its segment because it is not assigned to any.

To help in these situations, the `section` directive allows you to specify section properties. Each section property is specified by a single operand of a `section` directive; use as many operands as you need and separate them by commas, as in:

```
.CODE:  section code, align := 4, model := ip32
```

The following sections describe permitted operands of the `section` directive in more details. Note that all keywords referring to section properties (such as `read`, `align`, etc.) are case-insensitive.

4.1.2.3.1 read, write and execute

These properties set individual access permissions for the section in question. They are written as one of:

```
        read
        write
        execute
```

When used as an operand in a `section` directive, these properties set section's read, write and execute permissions correspondingly.

4.1.2.3.2 code, data and const

These properties set combined access permissions for the section in question, based on the most commonly needed access permissions for different types of contents. They are written as one of:

```
        code
        data
        const
```

When used as an operand in a `section` directive, these properties set section's permissions as follows:

- `code` sets read and execute permission.
- `data` sets read and write permissions.
- `const` sets read permission.

4.1.2.3.3 align

This property specifies the alignment requirement of a section. It is written as:

```
        align := <alignment>
```

where `<alignment>` is a constant expression yielding an integer value representing the alignment boundary, which must be a power of 2. For example, the directive

```
.CODE:  section code, align := 4
```

specifies that `.CODE` is a code section (i.e. contains executable instructions) and is aligned at a 4-byte boundary (since all instructions must be naturally aligned).

Any number of spaces is allowed before and after the '`:=`' operator. If the alignment of a section is not explicitly specified, the section does not have alignment requirement (i.e. it can start at any address).

It is not safe for an alignment constraint of a section to exceed an alignment constraint of its containing segment. Therefore, if an attempt is made to define a bound section with an alignment constraint exceeding that of its containing segment, a warning is issued and the segment's alignment constraint is increased to match that of the bound section in question. Needless to say, this is not a recommended programming practice.

4.1.2.3.4 address

This property specifies the constant address of a section. It is written as:

```
        address := <address>
```

where `<address>` is a constant expression yielding an integer value representing the section's address. For example, the directive

```
.CODE:  section code, address := 0xFFFFFFFFFFF00000
```

specifies that `.CODE` is a code section (i.e. contains executable instructions) and is located at address `0xFFFFFFFFFFF00000` (which is, probably, the ROM).

Any number of spaces is allowed before and after the '`:=`' operator. If the address of a segment is not explicitly specified, the segment can start at any address (as determined by linker and/or loader).

When assigning addresses to bound sections and their containing segments, one shall remember that:

- A bound section must be entirely within an address range of its containing segment.
- Different segments cannot intersect.
- Different sections cannot intersect.

4.1.2.3.5 max

This property specifies the maximum size the section can grow to when the linker performs section joining. It is written as:

```
        max := <size>
```

where `<size>` is a constant expression yielding an integer value representing the section's maximum permitted size. For example, the directive

```
.CODE:  section code, max := 65536
```

specifies that `.CODE` is a code segment (i.e. contains executable instructions) and is guaranteed not to exceed 64K in size.

Any number of spaces is allowed before and after the '`:=`' operator. If the maximum size of a segment is not explicitly specified, the segment can grow to any size.

4.1.2.3.6 private, shared and thread

These properties specify how the memory is allocated for the section. They are mutually exclusive and are written as one of:

```
        private
        shared
        thread
```

Depending on which property was used, the following memory allocation strategy will be used for the section:

- `private` sections are allocated per process. If several instances of the same application are running, each instance will have its own, private, copy of the section. This is the default allocation mode for global static data.
- `shared` sections are allocated once per application. If several instances of the same application are running, they will all have shared access to the same copy of a `shared` section. In effect, shared sections provide statically allocated shared memory. Code sections will be frequently marked as `shared`, since there is no need to keep an identical copy of a program's code replicated in each instance of that program that is running.
- `thread` sections represent the other extreme – in a multithreaded application each thread will have its own, private, copy of a `thread` section. This permits using `thread` sections as statically allocated thread-local storage.

If the memory allocation mode for a given roaming section is not specified explicitly, `private` is assumed. If the memory allocation mode for a given bound section is not specified explicitly, it is inherited from the bound section's containing segment.

A section is not permitted to reside in a segment if the two have different memory allocation modes.

4.1.2.3.7 preload, loadondemand and noload

These properties specify how the OS loader brings sections into the memory image of a process. They are mutually exclusive and are written as one of:

```
        preload
        loadondemand
        noload
```

Depending on which property was used, the following strategy will be used for the section:

- `preload` sections are brought into memory when an application is started. By the time an application begins execution, all its `preload` sections are in memory.
- `loadondemand` sections are not loaded when an application starts. However, the OS keeps track of them; any attempt to use one of these sections is intercepted by OS, which then loads the section. These sections are best suited for information that is not normally needed for the program execution, but may still be required in some circumstances (such as exception handlers or debug information).
- `noload` sections are never automatically loaded into the process' memory image. If an application needs to use one of these sections, it must explicitly ask OS to load it. These sections are best suited for information that is not needed during program execution, but may still be required in some unusual circumstances.

If the section loading mode for a given roaming section is not specified explicitly, `preload` is assumed. If the section loading mode for a given bound section is not specified explicitly, it is inherited from the bound section's containing segment.

A section is not permitted to reside in a segment if the two have different memory loading mode.

Note that when an assembler-language program is written to run on a bare hardware (i.e. without OS support), only `preload` sections will be part of that program's memory image.

4.1.2.3.8 model

This property specifies what memory model the section conforms to. It is written as:

```
        model := {ilp64|lp64|ip32}
```

When a section conforms to a specific memory model, all symbols within that segment must conform to the same memory model.

Any number of spaces is allowed before and after the '`:=`' operator. If the memory model of a roaming section is not explicitly specified, the current default memory model is assumed for that section (as specified by the `model` directive, see below). If the memory model of a bound section is not explicitly specified, it is inherited from the bound section's containing segment. Memory model names (`ilp64`, `lp64` and `ip32`) are case-insensitive.

4.1.2.3.9 combine and common

These properties specify how the linker combines several definitions of the same section that originate in several different object modules. They are mutually exclusive and are written as one of:

```
        combine
        common
```

Depending on which property was used, the following strategy will be used by the linker:

- `combine` means that all definitions of the section coming from all object modules will be concatenated together, forming a larger section. A typical example where this behaviour is necessary is when linking code sections – each object module will have its own `.CODE` section containing that object module's code, and all these `.CODE` sections must be concatenated to produce a large `.CODE` section containing all of the application's code.
- `common` means that all definitions of the section coming from all object modules will be overlaid. The size of the resulting section is the same as the size of the largest overlay. In addition, there must be no conflict as to what the initial contents of each byte of the resulting section is. A typical example of when this is useful is when compiling C++ templates. The compiler will normally produce a single code section for each template and place it in every object module where it is used. During linking, all definitions of each template are overlaid, resulting in the executable image containing only one copy of each template.

If a combination mode of a section is not specified explicitly, `combine` is assumed.

4.1.3 model

This directive has the following form:

```
        model   {ilp64|lp64|ip32}
```

The directive sets the default memory model to the specified value. All subsequent `section` and `segment` directives that do not have the `model` property explicitly specified will be assigned that default memory model.

At the beginning of the compilation unit, the default memory model is as specified from the assembler command line. If the default memory model is not specified from the assembler command line, the default memory model at the beginning of the compilation unit is `ilp64`.

4.1.4 group … end group

This directive has the following form:

```
[<name>:] group
   <section name>
   .    .    .
   <section name>
 [<name>:] end group
```

This directive specifies that all sections whose names appear between the `group` and `end group` directives belong to the specified group.

Depending on whether the group is named or not, one of two choices is possible:

- If the group is named, then the `<name>` used in the `end group` directive must match the `<name>` used in the `group` directive. The corresponding named group will appear in an object module and may be joined with other groups with the same name by linker.
- If the group is not named, then both `group` and `end group` directives must not specify the `<name>` portion. In this case a new, anonymous, group is created and placed into an object module; this anonymous group will not be joined with any other group by the linker.

Group members are specified by writing one member section name per line within the `group … end group` directives. For example, the following anonymous group contains 2 member sections `proc1::code` and `proc1::const`:

```
        group
            proc1::code
            proc1::const
        end group
```

No other assembler constructs (such as other directives or instructions) are allowed between the `group` and `end group` directives.

4.2 Symbol definition directives

Symbol definition directives are used to specify properties of symbols used in the assembler program.

4.2.1 proc … end proc

This directive has the following form:

```
<name>: proc
   .   .   .
<name>: end proc
```

where `<name>` is a name of the procedure being defined and must be the same in both `proc` and `end proc` directives.

This directive has two purposes:

- Defining the boundaries of a procedure, and
- Specifying the symbol that shall be used to refer to the procedure's address.

Typically, the `proc … end proc` directive would encompass instructions (and literal pools, if necessary) of a single procedure, as in:

```
foo:    proc
        jr      $ra ; just return at once
foo:    end proc
```

Therefore, each `proc … end proc` directive must appear within some section.

In addition to defining the address of the procedure's naming symbol to be the address of the procedure's first instruction, the `proc … end proc` directive also defines the "size" of the symbol. This is done automatically (as the assembler always knows how many bytes of code and/or data were written between `proc` and `end proc` directives) and is called *implicit symbol size specification* (as opposed to *explicit* symbol size specification, see `size` directive).

4.2.2 var … end var

This directive has the following form:

```
<name>: var
    .    .    .
<name>: end var
```

where `<name>` is a name of the variable being defined and must be the same in both `var` and `end var` directives.

This directive has two purposes:

- Defining the boundaries of a variable, and
- Specifying the symbol that shall be used to refer to the variable's address.

Typically, the `var … end var` directive would encompass data definition directives, as in:

```
foo:    var
        dd      L(10)
        dd      [4]B(0)
        dd      A("string")
foo:    end var
```

Therefore, each `var … end var` directive must appear within some section.

In addition to defining the address of the variable's naming symbol to be the address of the variable's first data byte, the `var … end var` directive also defines the "size" of the symbol. This is done automatically (as the assembler always knows how many bytes of data were written between `var` and `end var` directives) and is called *implicit symbol size specification* (as opposed to *explicit* symbol size specification, see `size` directive).

4.2.3 equ

This directive has the following form:

```
<name>: equ <expression>
```

where each `<name>` is a name of a symbol not otherwise defined in this compilation unit and `<expression>` is an expression yielding an integer value.

The directive tells the assembler that the address of the symbol with given `<name>` is the value of the `<expression>`.

Regardless of what object file format is used as assembler's output, any constant expression is permitted as the symbol's value, so the following directives are always valid:

```
X:      equ     1
Y:      equ     X * 10
```

(Note that the definition of `Y` refers to `X`; however, this is valid as the value of `X` is a compile-time constant).

Depending on the object file format produced, assembler may also allow deferred expressions for specification of symbol values, as in:

```
.text:  section
   .    .    .
.text:  end section

X:      equ     .text + 1 ;  address of the 2nd byte of the section
```

Currently, the only object file format that permits deferred expressions in `equ` directives is NGOFF.

4.2.4 public

This directive has the following form:

```
        public <name1>, …, <namen>
```

where each `<name`$_i$`>` is a name of a symbol defined in this compilation unit.

The directive specifies public visibility for the listed symbols. Normally, symbols defined in a compilation unit are not visible outside the compilation unit's boundaries. However, when made public, symbols become visible to other compilation units (where they must be declared with an `extern` directive, see below, in order to be used).

4.2.5 export

This directive has the following form:

```
        export <name1>, …, <namen>
```

where each `<namei>` is a name of a symbol defined in this compilation unit.

The directive specifies export visibility for the listed symbols. This is quite similar to public visibility (as declared by `public` directive), but there is also a difference. When several object modules are linked together to produce an executable module, they can use each other's public symbols; however, none of these symbols will be visible outside the linked executable module. These object modules can also use each other's export symbols; however, all these export symbols will then also be visible outside of the executable module. This is important for dynamic linking, where dynamic-link libraries must provide global symbols that their clients can use.

4.2.6 import

This directive has the following form:

```
        import <name1>, …, <namen>
```

where each `<namei>` is a name of a symbol not defined or declared as external (see next chapter) in this compilation unit.

The directive specifies import visibility for the listed symbols. What this means is that these symbols are not defined explicitly in this compilation unit or any other compilation units that will be linked together to form an executable module. Instead, the linker sets up the executable module for dynamic linking, to bring in the dynamic-link library exporting the actual definition of these symbols at load time.

In order to have enough information to prepare the executable module for dynamic linking, the linker must know which dynamic-link libraries are to be used. This is achieved by giving the linker one or more import libraries along with all necessary object modules. Each import library is, basically, a set of rules such as "imported symbol S_1 can be found in dynamic-link library L_1, imported symbol S_2 can be found in dynamic-link library L_2, and so on.

There is also a qualified form of the `import` directive:

```
        import <name1> from <library1>, …, <namen> from
<libraryn>
```

When this form is used, it tells the assembler that at load time the actual definition of each `<namei>` will be exported by the dynamic-link library `<libraryi>`. The linker has all the information it needs to prepare such symbols for dynamic linking, so no import libraries are needed.

Naturally, a mixed form is also possible, as in:

```
        import X, Y from "mydll", Z
```

It is permitted for several compilation units to `import` the same symbol, provided that there is no disagreement on what library that symbol is imported from. For example, if one compilation unit contains the directive:

```
import X
```

and another compilation unit contains the directive:

```
import X from "mydll"
```

then, when the two corresponding object modules are linked together, the two imported symbols will be recognized as a double importing of the same symbol from a known dynamic-link library, so no import library will be needed.

4.2.7 extern

This directive has the following form:

```
extern <name₁>, …, <nameₙ>
```

where each `<nameᵢ>` is a name of a symbol not defined or imported in this compilation unit.

The directive specifies external visibility for the listed symbols. This means that, as far as the current compilation unit is concerned, the actual definitions of these symbols exist in some other compilation unit and it's up to the linker to find one.

Note that external symbols used by a compilation unit can be resolved to any of the following:

- A public symbol defined by some other compilation unit (resolution is performed by the linker when an executable module is made).
- An exported symbol defined by some other compilation unit (resolution is performed by the linker when an executable module is made).
- An imported symbol defined by some other compilation unit (partial resolution is performed by the linker when an executable module is made, but final resolution will be made by dynamic linker when the executable module is loaded into memory).

4.2.8 keep

This directive has the following form:

```
keep <name₁>, …, <nameₙ>
```

where each `<nameᵢ>` is a name of a local symbol (i.e. a symbol defined in this compilation unit but not made publicly visible by `public` or `export` directives).

Normally, local symbols do not make it into an object module or, if they need to be present in an object module (for example, if used as relocation targets), their names are stripped. This helps to reduce the size of an object module at the cost of making some tasks more difficult (for example, listing the contents of an object module will not tell you where the boundaries between private functions are, etc.)

The keep directive instructs the assembler to make sure that specific local symbols *must* make their way into the produced object module (even if removing them would have been safe otherwise) and that these symbols will retain their names.

4.2.9 weak

This directive has the following form:

weak <name$_1$>, …, <name$_n$>

where each <name$_i$> is a name of a global symbol (i.e. a symbol mentioned in a public, export, import or extern directive).

The weak directive instructs the assembler that all global symbols mentioned therein are weak globals.

There are two cases, depending on whether the weak symbol is defined in the current compilation unit or not:

- If the symbol appears in a public or export directive, it is defined in this compilation unit. Marking such symbol as “weak” results in a *weak definition*. Such definition is ignored by linker (both static and dynamic) if another definition of the same symbol exists. If there are *only* weak definitions of a symbol, one of them is chosen arbitrarily.
- If the symbol appears in an import or extern directive, it is defined in some other compilation unit. Marking such symbol as “weak” results in a *weak reference*. When the linker (whether static or dynamic) fails to resolve a weak reference (because, typically, such definition does not exist), it silently assumes that the symbol’s value is 0 and does not search available libraries for a potential definition.

4.2.10 size

This directive has the following form:

size <name$_1$> := <value$_1$>, …, <name$_n$> := <value$_n$>

where each <name$_i$> is a name of a symbol and <value$_i$> is a constant expression yielding an integer result.

The size directive assigns explicit sizes to the symbols as specified. For some symbols this may be necessary (for example, there is no other way to assign size to a symbol defined by an equ directive), otherwise the default means of assigning sizes to symbols must be overridden (for example, a procedure name symbol will be implicitly assigned size according to how many bytes of code and data occur between proc and end proc directives, but the programmer may have different intentions).

4.2.11 alias

This directive has the following form:

```
        alias <name_1> := <alias_1>, …, <name_n> := <alias_n>
```

where each <name$_i$> is a name of a symbol and <alias$_i$> is a constant expression yielding a string result.

The `alias` directive assigns aliases to symbols as specified. An *alias* of a symbol is an arbitrary string that is used instead of the symbol's name in all messages about the symbol. For example, if a compilation unit contains directives:

```
    extern  _SPL___package_c_cfunction
    alias _SPL___package_c_cfunction:=”package::function”
```

and, at link time, the definition of the symbol `_SPL___package_c_cfunction` is not found, then the actual error message issued by the linker would be along the lines of “the symbol `package::function` is not found”, which is far more informative to the programmer than if the symbol name was used.

The main purpose of aliases, as the above example suggests, is to free both static and dynamic linker from having to perform name unmangling in case an error message must be issued. However, it is also possible to utilize symbol alias information for other purposes (for example, when disassembling an object module, it makes it possible to replace mangled symbol names with their aliases which are familiar to the programmer, etc.)

Note that any symbol known within a compilation unit (including external and imported symbols) can be explicitly assigned an alias. If an external or imported symbol is assigned an alias that is different from an alias of the actual definition of that symbol, it is unspecified whether an error or warning will be reported or the situation will be handled silently.

Note, also, that some object file formats (such as ELF) do not permit symbol aliasing.

4.2.12 signature

This directive has the following form:

```
        signature <name_1> := <sig_1>, …, <name_n> := <sig_n>
```

where each <name$_i$> is a name of a symbol and <sig$_i$> is a constant expression yielding a string result.

A signature is an arbitrary string that can be assigned to a symbol. When several object modules assign signatures to the same symbol, all these signatures must match, otherwise a “signature mismatch” error is reported by the linker. The primary purpose of symbol signatures is to encode the information about symbol's type, thus to allow for type-safe linking of separately compiled program fragments.

4.2.13 rename

This directive has the following form:

```
        rename <name1> := <global1>, …, <namen> := <globaln>
```

where each `<namei>` is a name of a symbol, segment, section or group and `<globali>` is a global name of the corresponding symbol, segment, section or group.

The `rename` directive allows symbols, segments and sections to be named in the source program differently than in the resulting object module. Consider:

```
.text1: section
   .   .   .
.text1: end section

.text2: section
   .   .   .
.text2: end section

        global .text1 := .text, .text2 := .text
```

In the source program, there are two different sections `.text1` and `.text2`. However, both are renamed, so the generated object module will contain two separate sections named `.text`.

4.3 Data definition directives

Data definition directives are used to reserve (and, possibly, initialize) data memory.

4.3.1 dd

This directive has the following form:

```
 [<name>:] dd <declarator1>, …, <declaratorn>
```

where:

- `<name>`, if specified, is a symbol that, when used as an operand in an expression, will yield the address of the first data byte defined by the directive.
- Each of the operands `<declaratori>` is a specification of some initialized or uninitialized data area that must be inserted into the object program.

Since the directive defines actual contents, it must appear within section boundaries (i.e. between `section` and `end section` directives).

The following sections describe various forms in which declarators can be written.

4.3.1.1 Primitive declarators

A primitive declarator specifies a data area containing a single value that can be either initialized or uninitialized. It has one of the forms:

```
        <data type>
        <data type><initializer>
```

where:

- `<data type>` is a (case-insensitive) mnemonic specification of the type and alignment of a value, and
- An optional `<initializer>`, if present, specifies the actual value. If there is no initializer, the data area is left uninitialized.

The following `<data type>` mnemonics are supported:

- `B` (byte) – the primitive declarator defines one byte of data memory. If the initializer is specified, it must be an expression yielding an integer result in range $-2^7..2^8-1$.
- `H` (half-word) – the primitive declarator defines one half-word of data memory. If the initializer is specified, it must be an expression yielding an integer result in range $-2^{15}..2^{16}-1$. The data item must be naturally aligned.
- `HU` (half-word unaligned) – the primitive declarator defines one half-word of data memory. If the initializer is specified, it must be an expression yielding an integer result in range $-2^{15}..2^{16}-1$. The data item does not have to be naturally aligned.
- `W` (word) – the primitive declarator defines one word of data memory. If the initializer is specified, it must be an expression yielding an integer result in range $-2^{31}..2^{32}-1$. The data item must be naturally aligned.
- `WU` (word unaligned) – the primitive declarator defines one word of data memory. If the initializer is specified, it must be an expression yielding an integer result in range $-2^{31}..2^{32}-1$. The data item does not have to be naturally aligned.
- `L` (long word) – the primitive declarator defines one long word of data memory. If the initializer is specified, it must be an expression yielding an integer result in range $-2^{63}..2^{64}-1$. The data item must be naturally aligned.
- `LU` (long word unaligned) – the primitive declarator defines one long word of data memory. If the initializer is specified, it must be an expression yielding an integer result in range $-2^{63}..2^{64}-1$. The data item does not have to be naturally aligned.
- `F` (float) – the primitive declarator defines one word of data memory. If the initializer is specified, it must be a constant expression yielding a real result, which is stored in 32-bit real format. The data item must be naturally aligned.
- `FU` (float unaligned) – the primitive declarator defines one word of data memory. If the initializer is specified, it must be a constant expression yielding a real result, which is stored in 32-bit real format. The data item does not have to be naturally aligned.

- `D` (double) – the primitive declarator defines one long word of data memory. If the initializer is specified, it must be a constant expression yielding a real result, which is stored in 64-bit real format. The data item must be naturally aligned.
- `DU` (double unaligned) – the primitive declarator defines one long word of data memory. If the initializer is specified, it must be a constant expression yielding a real result, which is stored in 64-bit real format. The data item does not have to be naturally aligned.
- `A` (ASCII) – the primitive declarator defines a string of ASCII characters. If the initializer is specified, it must be a constant expression yielding a string result, which is stored using 1 byte per character. If no initializer is specified, the declarator results in 1 uninitialized ASCII character.
- `U` (Unicode) – the primitive declarator defines a string of Unicode BMP characters. If the initializer is specified, it must be a constant expression yielding a string result, which is stored using 2 bytes per character. If no initializer is specified, the declarator results in 1 uninitialized Unicode BMP character. The string must be naturally aligned.
- `UU` (Unicode unaligned) – the primitive declarator defines a string of Unicode BMP characters. If the initializer is specified, it must be a constant expression yielding a string result, which is stored using 2 bytes per character. If no initializer is specified, the declarator results in 1 uninitialized Unicode BMP character. The string does not have to be naturally aligned.
- `I` (ISO-10646) – the primitive declarator defines a string of ISO-10646 characters. If the initializer is specified, it must be a constant expression yielding a string result, which is stored using 4 bytes per character. If no initializer is specified, the declarator results in 1 uninitialized ISO-10646 character. The string must be naturally aligned.
- `IU` (ISO-10646 unaligned) – the primitive declarator defines a string of ISO-10646 characters. If the initializer is specified, it must be a constant expression yielding a string result, which is stored using 4 bytes per character. If no initializer is specified, the declarator results in 1 uninitialized ISO-10646 character. The string does not have to be naturally aligned.

4.3.1.2 Declarator groups

A declarator group is a comma-separated sequence of declarators enclosed in brackets which syntactically behaves as a single declarator. For example, the declarator group:

```
        (L0, B1, B2, H3, W)
```

Is, semantically, a single declarator that defines 16 bytes of data memory:

- A long word with the value 0.
- Two consecutive bytes with values 1 and 2, respectively.
- A half-word with the value 3.
- An uninitialized word.

4.3.1.3 Repeat counters

A repeat counter is an instruction to an assembler to repeatedly process a declarator some given number of times. It is written as:

```
[<repeat counter>]<declarator>
```

where:

- `<repeat counter>` is a constant expression yielding an integer result, and
- `<declarator>` is the declarator to be repeated.

The value `N` yielded by the `<repeat counter>` expression is treated as an unsigned integer repeat counter, and the net effect is as if `N` consecutive `<declarator>`s have been specified. For example, the declarator

```
[5] L0
```

is the same as

```
(L0, L0, L0, L0, L0)
```

Note that any declarator, including a declarator group, can be repeated; similarly, repeating is not confined to only one declarator level; therefore, the following declarator:

```
[2](B0,[3]B1)
```

is the same as

```
(B0, B1, B1, B1, B0, B1, B1, B1)
```

4.4 Assembly control directives

Directives described in this section affect the assembler's behaviour and compilation sequence.

4.4.1 include

This directive has one of the following forms:

```
include <header>
include <header> from <context>
```

where:

- `<header>` is a constant expression yielding a string result, which is interpreted as a name of the header file to include, and
- `<context>` is an identifier representing the name of an include context to include the header from.

Both directives cause the contents of the textual header file to be processed in place of the `include` directive. The `<header>` string gives the name of the header file; the difference between the two forms is in how the header file is located.

The first form (without explicit include context specification) is called an *unqualified* include and allows either absolute or relative `<header>` file name. If the `<header>` file name is absolute, the corresponding header file is included. If, however, the `<header>` file name is relative, the following locations are searched for the header file, in order, until the header file with the required name is found:

1. The directory containing the source file where the `include` directive occurs.
2. If the source file was itself `include`d by another source, the directory where that another source resides, and so on up the inclusion stack.
3. The list of designated include directories. This list consists of all directories specified by the `CDS_1_0_INCLUDE` environment variable and all directories explicitly designated for the purpose with a dedicated command line option.
4. The current working directory.

The second form (with explicit include context specification) is called a *qualified* include and allows only relative `<header>` file names. The header file is searched only in the directories associated with the include context having the specified `<context>` name. The initial set of known include contexts is defined by environment variables (each environment variable whose value is a directory or a list of directories is treated as a default include context) and can further be extended or modified with dedicated assembler command line option.

4.4.2 includebin

This directive has one of the following forms:

```
[<name>:] includebin <filename>
[<name>:] includebin <filename> from <context>
```

where:

- `<name>`, if specified, is a symbol that, when used as an operand in an expression, will yield the address of the first data byte included by the directive.
- `<filename>` is a constant expression yielding a string result, which is interpreted as a name of the header file to include, and
- `<context>` is an identifier representing the name of an include context to include the file from.

Both directives cause the contents of the binary file to be included in place of the `includebin` directive. Since this, effectively, amounts to defining out-of-line data at the current location, `includebin` directives must be nested within `section` … `end section` directives. The `<filename>` string gives the name of the file; the difference between the two forms is in how the file is located.

The first form (without explicit include context specification) is called an *unqualified* include and allows either absolute or relative `<filename>`. If the `<filename>` is absolute, the contents of the corresponding file is included. If, however, the

<filename> is relative, the following locations are searched for the file, in order, until the file with the required name is found:

5. The directory containing the source file where the includebin directive occurs.
6. If the source file was itself included by another source, the directory where that another source resides, and so on up the inclusion stack.
7. The list of designated include directories. This list consists of all directories specified by the CDS_1_0_INCLUDEBIN environment variable and all directories explicitly designated for the purpose with a dedicated command line option.
8. The current working directory.

The second form (with explicit include context specification) is called a *qualified* include and allows only relative <filename>s. The file is searched only in the directories associated with the include context having the specified <context> name. The initial set of known include contexts is defined by environment variables (each environment variable whose value is a directory or a list of directories is treated as a default include context) and can further be extended or modified with dedicated assembler command line option.

4.4.3 if … end if

This directive has the following form:

```
if <expression₁>
   <statements₁>
else if <expression₂>
   <statements₂>
.   .   .
else
   <statementsₙ>
end if
```

where each $<expression_i>$ is a constant expression yielding an integer result and each $<statements_i>$ is an arbitrary sequence of directives and instructions.

The if… end if directive controls conditional compilation of the fragments of the program. All $<expression_i>$ expressions are evaluated in sequence until one of them yields a non-zero result, the corresponding sequence of $<statements_i>$ is then compiled. If all $<expression_i>$ yield a zero result, then the $<statements_n>$ following the else delimiter are compiled instead. An if … end if directive can have an arbitrary number of else if clauses. If there are no statements in the else clause, the else delimiter can be omitted as well.

It is permitted to nest if … end if directives to an arbitrary depth.

4.4.4 :=

This directive has one of the following forms:

```
<identifier> := <value>
<identifier>(<param1>, …, <paramn>) := <value>
```

where:

- `<identifier>` is a name of the macroprocessor variable or function being defined,
- Each `<parami>` is a name of one parameter of a macroprocessor function being defined, and
- `<value>` is an arbitrary sequence of characters.

In its first form, the directive defines a macroprocessor variable with the specified name and value, as in:

```
LIMIT := 10
```

Each occurrence of the macroprocessor variable in subsequent source lines will be textually replaced with its value, so the following two instructions occurring after the above definition are the same:

```
        li.l $t1, 10
        li.l $t1, LIMIT
```

In its second form, the directive defines a macroprocessor function with the specified name, number of parameters and body, as in:

```
SQUARE(x) := (x)*(x)
```

Each call of the macroprocessor function in subsequent source lines will be textually replaced with its body, so the following two instructions occurring after the above definition are the same:

```
        li.l $t1, (10)*(10)
        li.l $t1, SQUARE(10)
```

The following additional restrictions are imposed by the assembler:

- All macroprocessor variables and functions live in the same namespace, also shared with macros. Therefore, it is not possible to have both a macroprocessor variable and a macroprocessor function with the same name to coexist (or a macroprocessor variable and macro with the same name).
- It is possible to redefine a macroprocessor variable or function with another `:=` directive; the old definition is then lost.
- Both macroprocessor variables and macroprocessor functions constitute a textual replacement of their occurrences in directives and statements with their values, any actual calculation specified in the body of a macroprocessor variable or function is performed after all replacements have been made (this is the reason why, in the definition of the "`SQUARE(x) := (x)*(x)`" macroprocessor function the parameter `x` in the function body is enclosed in

brackets – to allow calls like `SQUARE(a+b)` to correctly expand to `(a+b)*(a+b)` and not, incorrectly, to `a+b*a+b`).

4.4.5 undef

This directive has the following form:

```
        undef <identifier1>, …, <identifiern>
```

where each `<identifieri>` is a name of a macroprocessor variable, macroprocessor function or macro.

The directive causes the assembler to discard the definition of each of the listed macroprocessor variables, macroprocessor functions and/or macros. If some (or all) of the identifier do not refer to an existing definition of any of these entities, no error is reported.

4.4.6 macro

This directive has the following form:

```
<name>: macro <parameter1>, …, <parametern>
   <macro body>
<name>: end macro
```

where:

- `<name>` is the name assigned to the macro. The `<name>` mentioned in the `end macro` directive must match the `<name>` in the `macro` directive.
- Each `<parameteri>` is one macro parameter of the form:
  `<name>[:=<value>]`
  where `<name>` is a name of the macro parameter and `<value>`, if specified, is the default value of the macro parameter.

This directive defines a new macro with the specified name, parameters and body (macro body can be an arbitrary sequence of directives and/or instructions). Consider:

```
move.l: macro x,y
        l.l     $t0, x
        s.l     $t0, y
move.l: end macro
```

Once a macro is defined, it can be used in a manner similar to an instruction; for example, the following statement later in the program:

```
        move.l 8[$s1], 16[$s1]
```

will be macro-expanded to:

```
        l.l     $t0, 8[$s1]
        s.l     $t0, 16[$s1]
```

Note that all occurrences of each macro parameter within the macro body are replaced with corresponding macro call arguments when macro expansion is performed.

4.4.6.1 Specifying parameters in macro calls

When calling a macro, its argument list consists of zero or more comma-separated macro arguments. Each macro argument specifies the value for one parameter of the macro being called and can be either a positional or named macro argument.

A *positional* macro argument is just an arbitrary string, as in the above example. This string is used as a value of the macro parameter at the corresponding position in the macro parameters list (hence the name “positional”).

A `named` macro argument has a form `<name>:=<value>`, where `<name>` is the name of the macro parameter whose value is specified and `<value>` is the arbitrary string to assign to that macro parameter as value.

Unlike positional macro arguments, named macro arguments can occur in any order, so the following three macro calls are equivalent:

```
        move.l 8[$s1], 16[$s1]
        move.l x:=8[$s1], y:=16[$s1]
        move.l y:=16[$s1], x:=8[$s1]
```

The following restrictions are imposed on macro arguments lists:

- In a macro argument list, all positional macro arguments must occur before any named macro arguments.
- The value of each macro parameter can be specified at most once.

4.4.6.2 Specifying default macro parameter values

A default value of a macro parameter is the value assigned to that parameter if the macro call does not have a corresponding macro argument. Consider the “increment” macro:

```
inc.l:  macro reg, by:=1
        addi.l  reg, by
inc.l:  end macro
```

The following macro call:

```
        inc.l   $t0, 2
```

then expands to

```
        addi.l  $t0, 2
```

thus incrementing `$t0` by 2. However, it is also possible to call the same macro without specifying an explicit value for the `by` macro parameter:

```
        inc.l   $t0
```

in which case the default value is used, resulting in

```
        addi.l  $t0, 1
```

All macro parameters that do not have default values *must* have their values explicitly specified in each macro call; otherwise an error is reported.

4.4.6.3 Labels in macro calls

If a macro call is labelled, the label is assigned to the first statement within the macro body; therefore a macro call:

```
lab:    inc.l   $t0, 2
```

expands to

```
lab:
        addi.l  $t0, 2
```

4.4.6.4 Local symbols in macros

It is sometimes necessary to use local symbols (instruction and directive labels) within a macro body. The straightforward approach, as in:

```
sort.l: macro r1, r2
        bge.l   r1, r2, done
        mov.l   $t0, r1
        mov.l   r1, r2
        mov.l   r2, $t0
done:
sort.l: end macro
```

does not work, because each macro call would result in the expanded macro body defining the same symbol `done` over and over again.

To help in situations such as the one above, the exclamation sign `!` can be used within macro bodies to assist in generation of unique local names. The assembler assigns a unique ID to each macro call (subsequent calls of the same macro will have different call IDs). Any occurrence of an exclamation sign `!` within a macro body is replaced with the macro call ID when a macro call is made and macro expansion is performed. Syntactically, a macro call ID is a valid identifier, which allows concatenating it with further identifier fragments to generate local symbol names unique for a given macro call; therefore, the example above becomes:

```
sort.l: macro r1, r2
        bge.l   r1, r2, !done
        mov.l   $t0, r1
        mov.l   r1, r2
        mov.l   r2, $t0
!done:
sort.l: end macro
```

4.5 Reporting directives

Directives described in this section are used to issue messages during program assembly.

4.5.1 error

This directive has the following form:

```
        error <message>
```

where `<message>` is a constant expression yielding a string result.

This directive causes the assembler to issue the user-defined error `<message>`.

4.5.2 warning

This directive has the following form:

```
        warning <message>
```

where `<message>` is a constant expression yielding a string result.

This directive causes the assembler to issue the user-defined warning `<message>`.

4.5.3 assert

This directive has the following form:

```
        assert <assertion>
```

where `<assertion>` is a constant expression yielding an integer.

This directive is used for checking compile-time conditions. It first evaluates the `<assertion>`. If the `<assertion>` evaluates to a non-zero value, no further action is taken. If, however, it evaluates to zero, an error is reported.

4.6 Miscellaneous directives

Directives described in this section do not belong to any specific group.

4.6.1 end

This directive has the following form:

```
        end
```

It signifies the end of the compilation unit. Note that the `end` directive cannot be labelled or have operands.

In an absence of the `end` directive, the entire contents of the assembler source file (along with all included headers) is considered to belong to the compilation unit. If, however, the `end` directive is present, the assembler stops to process the source right after the logical source line where the `end` directive occurred; the format and contents of the subsequent logical source lines is irrelevant.

4.6.2 entry

This directive has the following form:

```
        entry <symbol name>
```

When encountered in a translation unit, this directive specifies that the `<symbol name>` is a symbol designated as the program's entry point (note that a section, segment or group name cannot be used here). Several object modules may specify the program's entry point, provided they do not disagree on what symbol is designated as one.

Note that there is no requirement that the symbol designated as an entry point is local to a compilation unit – it may be public, exported, external or even imported.

4.6.3 align

This directive has the following form:

```
[<name>:] align <expression>
```

where:

- `<name>`, if specified, is a symbol that, when used as an operand in an expression, will yield the address of the first data byte reserved by the directive.
- The `<expression>` is a constant expression that specifies the required alignment, which must be a constant expression yielding an integer result that is a power of 2.

The directive reserves zero or more uninitialized bytes, as necessary to ensure that the following instruction or directive starts at an `<expression>`-byte boundary (for example, "`align 8`" means "align at a 8-byte boundary, etc.)

Normally, it is not safe for a section to contain `align` directives with higher alignment requirements than section's own alignment. If this happens, the assembler will issue a warning and increase the section's alignment requirement to match that of the `align` directive. Needless to say, this is not a recommended programming practice.

Since the directive may introduce actual padding bytes, it must appear within section boundaries (i.e. between `section` and `end section` directives).

4.6.4 target

This directive has the following form:

```
        target <name>
```

where `<name>` is an identifier of the target for which the compilation unit shall be translated. A dedicated assembler command line option “`-Help:Targets`” can be used to list all supported targets.

The `target` directive specifies that the following instructions and directives shall be translated under the assumption that the specified CPU will be the target platform. Among other things, this affects the availability of various assembly program elements depending on what optional features the selected target offers. For example, if the selected target does not have a Performance Monitoring feature, then instructions that access performance monitoring registers will not be permitted.

It is allowed for a compilation unit to contain any number of `target` directives; the scope of each `target` directive extends from the directive itself to the closest following `target` directive or the end of compilation unit, whatever occurs first.

4.6.5 use

This directive has the following form:

```
        use <library>, … , <library>
```

where each `<library>` is a string constant expression specifying the name of the library that shall be search by the linker for external symbols.

4.7 Predefined macroprocessor variables

When an assembler starts processing of a translation unit, a number of macroprocessor variables is already defined. These include:

- Standard macroprocessor variables (unless some or all of then were un-defined using a dedicated command line option), and
- Macroprocessor variables explicitly defined with a dedicated command line option.

The following standard macroprocessor variables are currently defined:

Variable	Description
`TARGET_ISA_VERSION`	The value of this variable is a numeric string representing the Cereon ISA version implemented by the target CPU, currently `1`.
`TARGET_BYTE_ORDER`	The value of this variable is a string constant `"big"` or `"little"`, depending on whether the translation is performed for a big-endian or little-endian target.
`TARGET_HAS_MONITORING`	The value of this variable is a numeric string `1` or `0`, depending on whether the target CPU has a Performance Monitoring feature or not.

`TARGET_HAS_VM`	The value of this variable is a numeric string `1` or `0`, depending on whether the target CPU has a Virtual Memory feature or not.
`TARGET_HAS_PM`	The value of this variable is a numeric string `1` or `0`, depending on whether the target CPU has a Protected Memory feature or not.
`TARGET_HAS_UNALIGNED`	The value of this variable is a numeric string `1` or `0`, depending on whether the target CPU has an Unaligned Operand feature or not.
`TARGET_HAS_DEBUG`	The value of this variable is a numeric string `1` or `0`, depending on whether the target CPU has a Debug feature or not.
`TARGET_HAS_FP`	The value of this variable is a numeric string `1` or `0`, depending on whether the target CPU has a Floating Point feature or not.
`TARGET_HAS_BASE`	The value of this variable is a numeric string `1` or `0`, depending on whether the target CPU has a Base feature or not.
`BUILD_MODE`	The value of this variable is a string constant “`debug`” or “`release`”, depending on whether the translation is performed in debug or release mode.

5 Instructions

In a typical assembler program, most of the logical source lines will contain instructions, where each such source line translates into exactly one Cereon instruction.

The general form of an instruction line is:

```
[<label>:] <opcode> [<operands>] [;<comment>]
```

where:

- `<label>` is an optional identifier that labels the instruction. When used as an operand in an expression, the value of `<label>` is the address of the corresponding instruction.
- `<opcode>` is a mnemonic operation code identifying the instruction.
- `<operands>`, if present are instruction operands. If the instruction has more than one operand, individual operands are separated by commas '`,`'.
- `<comment>`, if present, is a comment that starts at the leftmost semicolon '`;`' and continues until the end of the logical source line.

Any number of spaces (and/or TABs) is permitted:

- Before and after the `<label>`.
- Between the colon '`:`' and `<opcode>`.
- Between the last operand (or `<opcode>`, if the instruction has no operands) and the semicolon '`;`' that starts the comment.

For the complete list of Cereon instructions and their operands, consult the "Cereon Architecture Reference Manual".

A special case is a source line that contains no instruction, but only label (and, optionally, a comment). This label refers to the closest following instruction. For example, in the following code snippet `x`, `y` and `z` will all have the same value – the address of the `li.l` instruction:

```
x:
y:
z:      li.l    $t0, 1
```

5.1 Implicit operands

Each Cereon instruction has a specific number of operands. However, it is commonly the case that the destination register of an instruction is the same as the 1st operand register, as in:

```
        add.l   r0, r0, r1
        addi.l  $a0, $a0, 10
        not.l   $t1, $t1
```

To make writing an assembler language program easier, the following syntactic shortcuts are permitted:

- If, in a 3-operand instruction, the destination register is the same as the 1st operand register, the destination register can be omitted.
- If, in a 2-operand instruction, the destination register is the same as the operand register, the destination register can be omitted.

Therefore, the three instructions above can be rewritten as:

```
add.l   r0, r1
addi.l  $a0, 10
not.l   $t1
```

Note that, although conditional branch instructions have two register operands, the above rule does not apply to them, as neither of the register operands is a destination register.

5.2 Using expressions

In general, an expression yielding a value of an integer or real type can be used wherever an instruction requires, correspondingly, an integer or real value as part of its operand:

```
addi.l  $a0, 2+3*4
li.l    $t0, x+3[$s0]
li.f    $fs1, 2.0 / 4
```

5.3 Specifying memory addresses

Cereon instructions that use memory address as one of their operand fall into two categories:

- Load and store instructions, where memory address is specified as a base register plus constant offset.
- Jump and branch instructions, where memory address is specified as a constant offset to the instruction's `$ip` register.

In an assembler language program, there are two forms a memory operand can take:

- An explicit specification of base register and offset, written as `<offset>[<base register>]`, where `<offset>` is an arbitrary expression yielding an integer result (deferred expressions are allowed here as well as constant expressions), and `<base register>` is an integer register (e.g. "`4[$t0]`" or "`x+y[$ip]`". If the offset is 0, it can be omitted entirely (e.g. "`[$s0]`" is the same as "`0[$s0]`").
- An implicit memory address is represented by an arbitrary expression yielding an integer result. The assembler decides what base register to use.

The combination of two addressing modes (register-based or `$ip`-relative) and two address formats (explicit base or implicit base) gives four choices:

- A load or store instruction which uses explicit base specification for the memory address, as in “`l.l $t0, 4[$t1]`”. The `<base register>` and `<offset>` portions of the address specification are translated directly into values that occupy corresponding instruction fields. Note that the `<offset>` is limited in range to [-2^{15}..2^{15}-1], as the corresponding instruction field is only 16 bits long.
- A jump or branch instruction which uses explicit base specification for the memory address, as in “`j 100[$ip]`”. Since these instructions implicitly use `$ip` as the base register, the use of any other register as a base is not allowed. Note also that Cereon architecture requires that any instruction reading from the `$ip` register sees the value of the immediately following instruction there, so “`j [$ip]`” means “jump to the next instruction”, which is just a more expensive way of saying “`nop`”. The `<offset>` portion of the memory address is stored into the offset portion of the jump or branch instruction. Note that the offset range is also limited by what can be represented in an instruction.
- A load or store instruction that uses an address with an implicit base, such as “`l.l $t0, x`” to load a 64-bit value from the memory area labelled by `x` into `$t0`. Cereon assembler currently always assumes that `$ip` is used as an implicit base, so the data area labelled by `x` must reside in the same section as the load instruction and be within [-2^{15}..2^{15}-1] bytes range of it.
- A jump or branch instruction that uses an address with an implicit base, such as “`jal x`” to call a procedure labelled by `x`. Normally, the jump or branch target must reside in the same segment as the jump or branch instruction that uses it and be reachable (for example, if a branch target is more than 2^{15} instructions away from the branch instruction, there is no possible way to perform the branch, as branch instructions are limited to [-2^{15}..2^{15}-1] instructions branch range). However, the linker is smart enough to generate long branch veneers (described in the corresponding section of this document) to effectively lift this restriction; as far as an assembler language programmer is concerned, a jump or branch instruction can transfer control to anywhere in the 64-bit address space.

5.4 Using typed registers

As specified in the “Cereon Architecture Reference Manual”, each general-purpose register has three names:

- A numeric name (such as `r0`, `r15`, etc.), which specifies the register number (in range [0..31]) but does not tell how the contents of the register is interpreted.
- An integer name (such as `$a0`, `$t0` or `$ip`), which not only refers to a specific register, but also tells the assembler that the register is treated as containing an integer value.
- A real name (such as `$fa0` or `$ft0`), which not only refers to a specific register, but also tells the assembler that the register is treated as containing a

real value. Note that some registers (such as `$sp` or `$ip`) do not have corresponding real names, as these registers are never supposed to contain real values.

When writing assembler instructions, remember that:

- An instruction that expects one of its operands to be an integer register will allow you to use either numeric or integer register name for that operand, but not a real register name.
- An instruction that expects one of its operands to be a real register will allow you to use either numeric or real register name for that operand, but not an integer register name.

Therefore, the following instructions are valid:

```
li.l    r0, 100
li.l    $rv, 100
add.l   r0, $sp, r1
li.d    r0, 100.0
li.d    $frv, 100.0
add.d   r0, $fs1, r1
```

whereas the following instructions are not:

```
li.l    $frv, 100     ; $frv is not an integer register name
add.l   r0, $fa1, r1  ; $fa1 is not an integer register name
li.d    $rv, 100.0    ; $rv is not a real register name
add.d   r0, $s1, r1   ; $s1 is not a real register name
```

5.5 Writing channel programs

To simplify writing channel programs for targets equipped with DMA channels, the assembler allows writing channel programs mnemonically, as described in the following sections.

5.5.1 Register names

In DMA instructions, registers of a DMA channel are referred to by their numeric names `r0 .. r3`.

5.5.2 Transfer sources and destinations

Many DMA instructions require the source and/or destination of a data transfer to be specified. There are 4 different forms a source or destination of a DMA data transfer can take; consequently, there are 4 syntactic forms that can be used in an assembler program:

- `<register>`, where `<register>` is a DMA register name. The source (or destination) of a DMA data transfer is an I/O port whose number resides in the lower 16 bit of the corresponding DMA register.

- `[<register>]`, where `<register>` is a DMA register name. The source (or destination) of a DMA data transfer is a memory address which resides in the corresponding DMA register.
- `[<register>++]`, where `<register>` is a DMA register name. The source (or destination) of a DMA data transfer is a post-incremented memory address which resides in the corresponding DMA register.
- `[--<register>]`, where `<register>` is a DMA register name. The source (or destination) of a DMA data transfer is a pre-decremented memory address which resides in the corresponding DMA register.

For example, the following instruction reads 100 bytes from the I/O port whose number is in DMA register `r0` and stores these bytes to 100 consecutive memory addresses starting at address in DMA register `r1`:

```
        dma.t.b [r1++], r0, 100
```

5.5.3 Writing DMA instructions

Each DMA instruction occupies a single line and has a form similar to a “normal” CPU instruction, i.e.:

```
[<label>:] <DMA opcode> [<operands>] [;<comment>]
```

where:

- `<label>` is an optional identifier that labels the DMA instruction. When used as an operand in an expression, the value of `<label>` is the address of the corresponding DMA instruction.
- `<opcode>` is a mnemonic operation code identifying the DMA instruction.
- `<operands>`, if present are instruction operands. If the instruction has more than one operand, individual operands are separated by commas ‘`,`’.
- `<comment>`, if present, is a comment that starts at the leftmost semicolon ‘`;`’ and continues until the end of the logical source line.

As a general rule, there must be one operand for each individual field of a DMA instruction encoding. The form of this operand depends on the contents of the corresponding DMA instruction field:

- If the field is a 2-bit DMA register designator, the corresponding operand must be a DMA register name.
- If the field is a 4-bit specification of a source or destination of a DMA data transfer, the corresponding operand must be a data source/destination specifier in one of the forms described above.
- If the field is an immediate field, the corresponding operand must be an expression yielding an integer result that lies in the range representable within the field in question. Both constant and deferred expressions are allowed.

6 Invoking the assembler

The general form of assembler invocation is:

```
cerasm [ -<option> ...] [--] <filename>...
```

where:

- `cerasm` is the name of the assembler executable. Depending on the host OS the actual name of the assembler executable file may be different (for example, on Windows it is `cerasm.exe`).
- Each `-<option>` specifies a command line option that adjusts some facet of the assembler behaviour.
- Each `<filename>` must be a name of an assembler source file (the recommended filename extension for assembler source files is `.asm`). Note that assembler does not presently support integrated linking.
- An option terminator (two consecutive dashes `--`), if present in the command line as an independent option, marks the end of the options list; all subsequent parameters are treated as assembler source file names even if they start with a dash.

6.1 Environment variables

During operation, assembler uses a number of OS environment variables, all of which are described below.

6.1.1 CDS_1_0_ASSEMBLER_OPTIONS

If this environment variable is defined, its value must be a list of assembler options in the same format as used in the assembler command line during assembler invocation.

When an assembler processes the command line, all options specified by the `CDS_1_0_ASSEMBLER_OPTIONS` environment variable are processed as if they were explicitly specified in the assembler's command line; the only exception to that rule is in that if some option defined by the `CDS_1_0_ASSEMBLER_OPTIONS` environment variable conflicts with another option explicitly specified in the assembler command line, the former is ignored. This behaviour allows using the `CDS_1_0_ASSEMBLER_OPTIONS` environment variable to set the most commonly needed assembler options, which can be overridden in each particular case if necessary.

For example, if the value of the `CDS_1_0_ASSEMBLER_OPTIONS` environment variable is "`-Endian:Big`", assembler will always generate code for a big-endian target unless invoked with an option "`-Endian:Little`".

6.1.2 CDS_1_0_ASSEMBLER_DEBUG_OPTIONS

This environment variable, if defined, is used in a manner similar to `CDS_1_0_ASSEMBLER_OPTIONS`, except assembler options specified therein are only effective when performing a debug build. If some options specified in

`CDS_1_0_ASSEMBLER_OPTIONS` and
`CDS_1_0_ASSEMBLER_DEBUG_OPTIONS` are in conflict, those specified by
`CDS_1_0_ASSEMBLER_DEBUG_OPTIONS` take precedence; however, both have lower priority than options explicitly specified in the assembler's command line.

6.1.3 CDS_1_0_ASSEMBLER_RELEASE_OPTIONS

This environment variable, if defined, is used in a manner similar to `CDS_1_0_ASSEMBLER_OPTIONS`, except assembler options specified therein are only effective when performing a release build. If some options specified in `CDS_1_0_ASSEMBLER_OPTIONS` and `CDS_1_0_ASSEMBLER_RELEASE_OPTIONS` are in conflict, those specified by `CDS_1_0_ASSEMBLER_RELEASE_OPTIONS` take precedence; however, both have lower priority than options explicitly specified in the assembler's command line.

6.1.4 CDS_1_0_INCLUDE

This environment variable, if defined, must have as its value a list of directories where header files included by unqualified `include` directives are looked for. Note that these directories have lower priority than those designated for unqualified `include` headers lookup with corresponding command line options.

6.1.5 CDS_1_0_INCLUDEBIN

This environment variable, if defined, must have as its value a list of directories where files included by unqualified `includebin` directives are looked for. Note that these directories have lower priority than those designated for unqualified `includebin` lookup with corresponding command line options.

6.1.6 Custom include contexts

It is possible to define other environment variables to designate non-standard include contexts. Each of these environment variables must have as its value the list of directories comprising the corresponding include context and is made available to the assembler as an include context with the same name as an environment variable in question.

6.2 Options

An assembler option has one of the following forms:

```
-<option keyword>
-<option keyword>:<option parameters>
```

Where `<option keyword>` is a keyword identifying the option and `<option parameters>`, if present, specifies additional information for the option.

Both option keywords and option parameters are generally case-insensitive except where character case matters (for example, when specifying file or directory names as option parameters on a Unix host, where file system is case-sensitive).

For some options both forms (i.e. with and without option parameters) are permitted; for example the "`-Listing:<listing file name>`" option, which tells the assembler to produce a source listing into the specified file, can also be written as simply "`-Listing`", in which case the listing file name is derived from the assembler source file name by replacing its `.asm` filename extension with `.lst`.

6.3 Option files

Quite frequently a need arises to specify the same set of options for a large number of assembler invocations. If these options are specific to a given assembler project, the facilities provided by the `CDS_1_0_ASSEMBLER_OPTIONS` environment variable and related variables are insufficient (as they affect all assembler invocations); instead, option files can be set up and used.

An option file is a text file that contains one or more assembler options. To improve readability, line breaks and extra spaces are allowed between options in an arbitrary manner.

When invoking the assembler, a dedicated `–Via:<option file name>` option is used to tell the assembler to read an option file, treating all assembler options specified therein as if they occurred in the assembler command line. For example, if an option file is named `options` and contains the following lines:

```
-Endian:Big
-Debug
```

then invoking the assembler with the following command line:

```
cerasm –Via:options source.asm
```

has the same effect as:

```
cerasm -Endian:Big -Debug source.asm
```

Any number of option files can be specified; the maximum length of an option file is not limited except by available memory.

A special case is an option file including other option files (the `–Via:<option file name>` command line option can, like any other option, occur within an option file). Such inclusion is permitted as long as option file dependencies do not cause an inclusion cycle.

6.4 Option conflicts

Unlike most existing assemblers, the Cereon assembler is very strict about its command line. In particular, it is not permitted to specify two conflicting options in the same assembler invocation, so the following will cause a command-line error:

```
cerasm –Endian:Big –Endian:Little source.asm
```

Most existing assemblers will allow similar invocations (i.e. with conflicting options), automatically resolving these conflicts in an arbitrary manner (e.g. some assemblers may choose the 1st definition of a conflicting option, while other assemblers will use the last definition), sometimes with a warning. It is, however, a firm belief of the Cereon assembler authors that conflicting command line options shall not be permitted at all, as any strategy used for automatic conflict resolution makes the invocation result dependent upon the order in which assembler options are specified.

The only situation where option conflicts are resolved automatically is when options explicitly specified in the assembler command line (or in option files included by a `–via` option that occurs in the assembler command line) are in conflict with options specified by the `CDS_1_0_ASSEMBLER_OPTIONS`, `CDS_1_0_ASSEMBLER_DEBUG_OPTIONS` or `CDS_1_0_ASSEMBLER_RELEASE_OPTIONS` environment variables (or in option files included by a `–Via` option that occurs in one of these environment variables). In this case, options specified in the assembler command line are always chosen; this behaviour permits explicit overriding of any implicit assembler options.

6.5 Character encoding

Normally, assembler expects all textual input (i.e. source files, headers and option files) to use the “native” character encoding of the host OS and produces all textual output (source listing, dependency files, error and warning messages, etc.) using the same encoding. The definition of a “native” encoding is specific to a given OS (for example, on Linux hosts the “native” encoding is selected when a host is configured).

It is, however, possible to specify that a different encoding shall be used on a per-text-type basis (i.e. “an encoding for source files and headers”, “an encoding for option files”, “an encoding for error and warning messages”, etc.)

6.6 Message files

All messages issued by the assembler (i.e. banners, statistics, warnings, errors, etc.) are stored in a textual message file. In the minimum configuration the assembler can operate entirely without message files, in which case built-in English Neutral messages are issued. Additional message files can also be provided in a particular assembler installation, containing the assembler messages localized for a specific culture.

When invoking an assembler, a dedicated command line option `–Locale:<locale ID>` can be used to specify the culture for which assembler messages shall be tailored. For example, invoking assembler with an option `–Locale:de` will cause all assembler messages, including diagnostics, to be issued in German (provided, of course, that the corresponding message file is available).

Message files always use UTF-8 encoding.

7 Assembler options

This section contains a complete description of each assembler option.

The following notation is used when describing assembler options:

`[ x ]`	– The element `x` is optional.
`[ x …]`	– The element `x` can be repeated 0 or more times.
`{ x	y }`	– Either `x` or `y` can be chosen.

7.1 -Banner

This option has the following format:

```
    -Banner[:{On|Off}]
```

When used, this option turns on (`-Banner:On`) or off (`-Banner:Off`) the printing of the version and copyright banner when the assembler is invoked; a default form `-Banner` is equivalent to `-Banner:On`, which is also the default assembler behaviour.

7.2 -Debug

This option has the following format:

```
    -Debug[:{On|Off}]
```

When used, this option specifies whether the translation shall be performed in debug (`-Debug:On`) or release (`-Debug:Off`) mode; a default form `-Debug` is equivalent to `-Debug:On`. The default assembler behaviour (when this option is not specified) is to perform compilation in release mode.

When compiling in debug mode:

- The macroprocessor variable `BUILD_MODE` is defined as the string `"debug"`.
- Debug information is generated and placed into an object file.

When compiling in release mode:

- The macroprocessor variable `BUILD_MODE` is defined as the string `"release"`.
- Debug information is not generated.

Note that object modules created in debug and release modes are binary compatible (i.e. can be linked together, whether statically or dynamically).

7.3 -Define

This option has the following format:

```
-Define:<name>[=<value>]
```

When used, this option causes a macroprocessor variable to be defined with the specified value; if a value is omitted the macroprocessor variable is defined with an empty string as its value.

It is explicitly permitted to define standard macroprocessor variables (such as `TARGET_ISA_VERSION`) with this option; these definitions take precedence over default definitions that would otherwise be provided by the assembler.

7.4 -Dependencies

This option has the following format:

```
-Dependencies[:<destination>]
```

When used, this option causes a textual file containing all dependencies of a compilation unit to be written. This file lists the main source file of a compilation unit as well as all headers directly or indirectly `include`d and all files directly or indirectly `includebin`ed, with one fully qualified (i.e. with drive and full path) file name per line.

Dependency files are emitted in the "native" character set for the host OS unless another character set is selected with `–DependencyEncoding` option.

Depending on the exact form of the option's parameter, the dependencies file may be written to several different locations:

- If the `<destination>` parameter is not specified, the name of the dependencies file is derived from the name of the main assembler source file by replacing the filename extension (typically `.asm`) with `.dep`. The dependencies file is written into the same directory where the corresponding assembler source file resides.
- If the `<destination>` parameter is specified and ends with a path component separator, it is assumed to refer to a directory where the dependencies file shall be written; the name of the dependency file is derived from the name of the main assembler source file by replacing the filename extension (typically `.asm`) with `.dep`. The destination directory is created automatically if it does not already exist. Non-absolute directory names are considered to be relative to the current working directory, not to the assembler source file's location.
- If the `<destination>` parameter is specified and does not end with a path component separator, it is assumed to refer to a file where the dependencies shall be written. Note that when assembler is invoked to compile more than one compilation unit, this feature cannot be used, as it will cause dependency file conflict. Non-absolute file names are considered to be relative to the current working directory, not to the assembler source file's location.
- If the `<destination>` parameter is a single dash '`–`', the list of dependencies is written to the standard output; in this case the character set

used for the dependencies output is that selected for the standard output with the `-StdoutEncoding` option.

7.5 -DependenciesEncoding

This option has the following format:

```
-DependenciesEncoding:<encoding>
```

When used, this option causes dependency files to be written using the specified encoding (character set). The `<encoding>` parameter must be one of the encoding names supported by the assembler, use the “`cerasm -Help:Encodings`” command to list supported encodings.

If this option is not specified, dependency files are written using the “native” character set of the host OS. The same effect can be achieved by using this option with a special `native` parameter, as in

```
-DependenciesEncoding:Native
```

7.6 -DisableWarning

This option has the following format:

```
-DisableWarning:<warning ID>[,<warning ID> …]
```

When used, this option causes warnings with the specified ID not to be issued. Any number of comma-separated warning IDs can be specified in the same `-DisableWarning` option; similarly, it is permitted to have more than one `-DisableWarning` option in the same command line.

7.7 -EnableWarning

This option has the following format:

```
-EnableWarning:<warning ID>[,<warning ID> …]
```

When used, this option causes warnings with the specified ID to be issued. Any number of comma-separated warning IDs can be specified in the same `-EnableWarning` option; similarly, it is permitted to have more than one `-EnableWarning` option in the same command line.

7.8 -Endian

This option has the following format:

```
-Endian:{Big|Little}
```

When used, this option causes the assembler to generate code suitable for a big-endian (`-Endian:Big`) or little-endian (`-Endian:Little`) target, correspondingly. If the target endianness is not specified explicitly and the target only permits one endianness, that endianness is used. If the target endianness is not specified explicitly and the target can operate in both big-endian and little-endian modes, big-endian is assumed.

7.9 -Format

This option has the following format:

```
-Format:<output format>
```

When used, this option instructs the assembler to emit the output (i.e. the object module) in the specified format. Use the “`cerasm –Help:Formats`” command to list supported formats.

7.10 -FullMessagePaths

This option has the following format:

```
-FullMessagePaths[:{On|Off}]
```

When used, this option specifies whether source file names where error and warning messages are anchored shall be printed in fully qualified (`–FullMessagePaths:On`) or relative (`–FullMessagePaths:Off`) form; a default form `–FullMessagePaths` is equivalent to `–FullMessagePaths:On`. The default assembler behaviour (when this option is not specified) is to print anchor file names in a short (relative) form.

7.11 -Help

This option has the following format:

```
-Help[:<subject>]
```

When used, it causes the help on the specific subject to be printed. Possible `<subject>` choices are:

- `–Help` (no subject) – prints the assembler command line & options reference.
- `–Help:Targets` – prints the list of supported target platforms.
- `–Help:Formats` – prints the list of supported output formats.
- `–Help:Encodings` – prints the list of supported character sets.
- `–Help:Locales` – prints the list of supported message locales.

7.12 -IncludeContext

This option has the following format:

```
-IncludeContext:<name>=<directory list>
```

When used, it defined the include context with the specified name to be defined and associated with the specified list of directories where header files are located. The `<directory list>` is a list of full paths of one or more directories separated by a host OS – specific path list separator (`:` on Unix, `;` on Windows, etc.)

An include context defined from the assembler command line overrides a definition made by an environment variable (if one exists); this allows any include context to be redefined for one compilation only.

7.13 -IncludePath

This option has the following format:

```
    -IncludePath:<directory list>
```

When used, it adds the specified directories to the list of directories where unqualified `include` headers are searched. The `<directory list>` is a list of full paths of one or more directories separated by a host OS – specific path list separator (`:` on Unix, `;` on Windows, etc.)

Include directories specified with this option have higher priority than those specified with the `CDS_1_0_INCLUDE` environment variable (if one is defined).

7.14 -IncludebinPath

This option has the following format:

```
    -IncludebinPath:<directory list>
```

When used, it adds the specified directories to the list of directories where unqualified `includebin` files are searched. The `<directory list>` is a list of full paths of one or more directories separated by a host OS – specific path list separator (`:` on Unix, `;` on Windows, etc.)

Include directories specified with this option have higher priority than those specified with the `CDS_1_0_INCLUDEBIN` environment variable (if one is defined).

7.15 -Listing

This option has the following format:

```
    -Listing[:<destination>]
```

When used, this option causes a textual file containing the source listing of a compilation unit to be written.

Source listing files are emitted in the “native” character set for the host OS unless another character set is selected with `–ListingEncoding` option.

Depending on the exact form of the option’s parameter, the source listing file may be written to several different locations:

- If the `<destination>` parameter is not specified, the name of the source listing file is derived from the name of the main assembler file by replacing the filename extension (typically `.asm`) with `.lst`. The source listing file is written into the same directory where the corresponding assembler source file resides.

- If the `<destination>` parameter is specified and ends with a path component separator, it is assumed to refer to a directory where the source listing file shall be written; the name of the source listing file is derived from the name of the main assembler source file by replacing the filename extension (typically `.asm`) with `.lst`. The destination directory is created automatically if it does not already exist. Non-absolute directory names are considered to be relative to the current working directory, not to the assembler source file's location.
- If the `<destination>` parameter is specified and does not end with a path component separator, it is assumed to refer to a file where the source listing shall be written. Note that when assembler is invoked to compile more than one compilation unit, this feature cannot be used, as it will cause source listing file conflict. Non-absolute file names are considered to be relative to the current working directory, not to the assembler source file's location.
- If the `<destination>` parameter is a single dash '`-`', the source listing is written to the standard output; in this case the character set used for the listing is that selected for the standard output with the `–StdoutEncoding` option.

7.16 -ListingEncoding

This option has the following format:

```
    -ListingEncoding:<encoding>
```

When used, this option causes source listing files to be written using the specified encoding (character set). The `<encoding>` parameter must be one of the encoding names supported by the assembler, use the "`cerasm –Help:Encodings`" command to list supported encodings.

If this option is not specified, source listing files are written using the "native" character set of the host OS. The same effect can be achieved by using this option with a special `native` parameter, as in

```
    -ListingEncoding:Native
```

7.17 -Locale

This option has the following format:

```
    -Locale:<locale ID>
```

When used, this option causes all assembler messages and diagnostics to be written using the specified locale instead of the default locale. The `<locale ID>` can have one of the following forms:

- `-Locale:<language>`, where `<language>` is a 2-letter ISO-639 language code – causes assembler messages to be written using the neutral locale for the specified language.

- `-Locale:<language>_<country>`, where `<language>` is a 2-letter ISO-639 language code and `<country>` is a 2-letter ISO-3166 country code – causes assembler messages to be written using the culture locale for the specified language and country.
- `-Locale:<language>_<country>_<variant>`, where `<language>` is a 2-letter ISO-639 language code, `<country>` is a 2-letter ISO-3166 country code and `<variant>` is an arbitrary locale variant string – causes assembler messages to be written using the full locale for the specified language, country and variant.
- `-Locale:Native` – causes assembler messages to be written using the "native" locale of the host OS.
- `-Locale:Invariant` – causes assembler messages to be written using an invariant (culture-independent) locale.

If this option is not specified, the `-Locale:Invariant` is assumed. An attempt to instruct the assembler to produce output and messages using a locale for which the assembler message file is not available causes the locale's parent to be used; if there is no message file for that parent locale, then its parent is used in turn, etc; if all else fails, an invariant locale is used for assembler output and messages.

7.18 -Model

This option has the following format:

```
-Model:{ilp64|lp64|ip32}
```

When used, this option sets the default memory model to be used for all compilation units. If the option is not specified, ILP64 memory model is used as a default.

7.19 -Output

This option has the following format:

```
-Output:<destination>
```

When used, this option causes the result of the translation (typically an object file) to be written to the specified destination.

Depending on the exact form of the option's parameter, the output file may be written to several different locations:

- If the `<destination>` parameter is not specified, the name of the output file is derived from the name of the main assembler source file by replacing the filename extension (typically `.asm`) with `.rel`.
- If the `<destination>` parameter is specified and ends with a path component separator, it is assumed to refer to a directory where the output file shall be written; the name of the output file is derived from the name of the main assembler source file by replacing the filename extension (typically `.asm`) with `.rel`. Non-absolute directory names are considered to be relative to the current working directory, not to the assembler source file's location.

- If the `<destination>` parameter is specified and does not end with a path component separator, it is assumed to refer to an output file itself. Non-absolute file names are considered to be relative to the current working directory, not to the assembler source file’s location.
- If the `<destination>` parameter is a single dash ‘`-`‘, then an output file is not written; this mode can be used to perform syntactic & semantic checks of assembler sources without producing an output file.

7.20 -PreprocessorOutput

This option has the following format:

```
-PreprocessorOutput[:<destination>]
```

When used, this option causes a textual file containing the preprocessed form of each compilation unit to be written.

Preprocessor output files are emitted in the “native” character set for the host OS unless another character set is selected with `–PreprocessorOutputEncoding` option.

Depending on the exact form of the option’s parameter, the preprocessor output may be written to several different locations:

- If the `<destination>` parameter is not specified, the name of the preprocessor output file is derived from the name of the main assembler source file by replacing the filename extension (typically `.asm`) with `.pre`. The preprocessor output file is written into the same directory where the corresponding assembler source file resides.
- If the `<destination>` parameter is specified and ends with a path component separator, it is assumed to refer to a directory where the preprocessor output file shall be written; the name of the preprocessor output file is derived from the name of the main assembler source file by replacing the filename extension (typically `.asm`) with `.pre`. The destination directory is created automatically if it does not already exist. Non-absolute directory names are considered to be relative to the current working directory, not to the assembler source file’s location.
- If the `<destination>` parameter is specified and does not end with a path component separator, it is assumed to refer to a file where the preprocessor output shall be written. Note that when the assembler is invoked to compile more than one compilation unit, this feature cannot be used, as it will cause preprocessor output file conflict. Non-absolute file names are considered to be relative to the current working directory, not to the assembler source file’s location.
- If the `<destination>` parameter is a single dash ‘`-`‘, preprocessor output is written to the standard output; in this case the character set used for the preprocessor output is that selected for the standard output with the `–StdoutEncoding` option.

7.21 -PreprocessorOutputEncoding

This option has the following format:

```
      -PreprocessorOutputEncoding:<encoding>
```

When used, this option causes assembler to write preprocessor output using the specified encoding (character set). The `<encoding>` parameter must be one of the encoding names supported by the assembler, use the "`cerasm -Help:Encodings`" command to list supported encodings.

If this option is not specified, the preprocessor output is written using the "native" character set of the host OS. The same effect can be achieved by using this option with a special `native` parameter, as in

```
      -PreprocessorOutputEncoding:Native
```

7.22 -Progress

This option has the following format:

```
      -Progress[:{On|Off}]
```

When used, this option turns on (`-Progress:On`) or off (`-Progress:Off`) the printing of source file names of all compilation units as they are processed; a default form `-Progress` is equivalent to `-Progress:On`, which is also the default assembler behaviour.

7.23 -Quiet

This option has the following format:

```
      -Quiet[:{On|Off}]
```

The `-Quiet:On` option is a shortcut for the `-Banner:Off -Progress:Off` and `-Statistics:Off` option combination. Similarly, the `-Quiet:Off` option is a shortcut for the `-Banner:On -Progress:On` and `-Statistics:On` option combination.

7.24 -SourceEncoding

This option has the following format:

```
      -SourceEncoding:<encoding>
```

When used, this option causes assembler to assume that all source and header files use the specified encoding (character set). The `<encoding>` parameter must be one of the encoding names supported by the assembler, use the "`cerasm -Help:Encodings`" command to list supported encodings.

If this option is not specified, all source and header files are assumed to be using the “native” character set of the host OS. The same effect can be achieved by using this option with a special `native` parameter, as in

```
    -SourceEncoding:Native
```

7.25 -SourceTabWidth

This option has the following format:

```
    -SourceTabWidth:<value>
```

When used, this option causes assembler to assume that tab characters encountered in source and header files are equivalent to just enough spaces to advance the line position to the next tab stop and that tab stops are `<value>` columns apart. Historically, the most frequently used tab width was 8 columns; however, many recent IDEs and editors use tab width of 4 (or even 2) for source files.

If this option is not specified, all source and header files are assumed to be using the default tab width of 8.

7.26 -Statistics

This option has the following format:

```
    -Statistics[:{On|Off}]
```

When used, this option turns on (`-Statistics:On`) or off (`-Statistics:Off`) the printing of translation statistics at the end of assembler invocation; a default form `-Statistics` is equivalent to `-Statistics:On`, which is also the default assembler behaviour.

7.27 -StderrEncoding

This option has the following format:

```
    -StderrEncoding:<encoding>
```

When used, this option causes assembler to issue all error and warning messages using the specified encoding (character set). The `<encoding>` parameter must be one of the encoding names supported by the assembler, use the “`cerasm -Help:Encodings`” command to list supported encodings.

If this option is not specified, all error and warning messages are written using the “native” character set of the host OS. The same effect can be achieved by using this option with a special `native` parameter, as in

```
    -StderrEncoding:Native
```

7.28 -StdoutEncoding

This option has the following format:

```
    -StdoutEncoding:<encoding>
```

When used, this option causes assembler to issue all standard output using the specified encoding (character set). The `<encoding>` parameter must be one of the encoding names supported by the assembler, use the "`cerasm -Help:Encodings`" command to list supported encodings.

If this option is not specified, all standard output written by the assembler uses the "native" character set of the host OS. The same effect can be achieved by using this option with a special `native` parameter, as in

```
    -StdoutEncoding:Native
```

7.29 -Target

This option has the following format:

```
    -Target:<target platform>
```

When used, this option instructs the assembler to perform compilation for the specified target. Use the "`cerasm -Help:Targets`" command to list supported targets.

If the target is not explicitly specified from the command line, the assembler assumes the most restrictive target (i.e. Base feature only).

7.30 -Undefine

This option has the following format:

```
    -Undefine:<name>
```

When used, this option causes a standard macroprocessor variable to be *not* defined by default.

7.31 -UndefineStandard

This option has the following format:

```
    -UndefineStandard[:{On|Off}]
```

When used, this option specifies whether all standard macroprocessor variables normally defined by the assembler shall be defined (`-UndefineStandard:OFF`) or not defined (`-UndefineStandard:On`) when the translation starts.; a default form `-UndefineStandard` is equivalent to `-UndefineStandard:On`, while the default assembler behaviour is to define all standard macroprocessor variables.

7.32 -Via

This option has the following format:

```
-Via:<option file name>
```

When used, this option instructs the assembler to read the specified option file and process all command line options contained therein as if they were specified directly in the assembler's command line in place of a `–Via` option.

Option files are assumed to be text files that use a "native" character set of the host OS; however, it is possible to instruct the assembler to use a different encoding for option files with the `–ViaEncoding` option.

Option files can contain within them references to other option files (i.e. `–Via` options are permitted within option files) as long as option file references do not form a cycle.

7.33 -ViaEncoding

This option has the following format:

```
-ViaEncoding:<encoding>
```

When used, this option causes assembler to assume that all option files use the specified encoding (character set). The `<encoding>` parameter must be one of the encoding names supported by the assembler, use the "`cerasm –Help:Encodings`" command to list supported encodings.

If this option is not specified, all option files are assumed to be using the "native" character set of the host OS. The same effect can be achieved by using this option with a special `native` parameter, as in

```
-ViaEncoding:Native
```

7.34 -WarningAsError

This option has the following format:

```
-WarningAsError:<warning ID>[,<warning ID> …]
```

When used, this option causes warnings with the specified ID to be treated as errors. Any number of comma-separated warning IDs can be specified in the same `–WarningAsError` option; similarly, it is permitted to have more than one `–WarningAsError` option in the same command line. In particular, a warning issued as an error causes the assembler to exit with a nonzero exit code, signalling a compilation error.

Note that treatment of certain (or all) warnings as errors does not affect warning filtering – a warning message simply becomes an error message *if* it is decided that a warning message must be issued.

7.35 -WarningAsWarning

This option has the following format:

```
    -WarningAsWarning:<warning ID>[,<warning ID> …]
```

When used, this option causes warnings with the specified ID to *not* be treated as errors. Any number of comma-separated warning IDs can be specified in the same −`WarningAsWarning` option; similarly, it is permitted to have more than one −`WarningAsWarning` option in the same command line.

Note that treatment of certain (or all) warnings as errors does not affect warning filtering – a warning message simply becomes an error message *if* it is decided that a warning message must be issued.

7.36 -WarningLevel

This option has the following format:

```
    -WarningLevel:{0|1|2|3|4}
```

When used, this option selects the specified warning level. Generally, the higher the warning level the more warnings are issued; warning level 0 causes all warnings to be suppressed, while warning level 4 causes all warnings to be reported.

If the option is not specified, the default warning level is 2.

7.37 -WarningsAsErrors

This option has the following format:

```
    -WarningsAsErrors[:{On|Off}]
```

When used, this option turns on (`-WarningsAsErrors:On`) or off (`-WarningsAsErrors:Off`) the treatment of all warnings as errors; a default form `-WarningsAsErrors` is equivalent to `-WarningsAsErrors:On`, while `-WarningsAsErrors:Off` is the assembler default. In particular, a warning issued as an error causes the assembler to exit with a nonzero exit code, signalling a compilation error.

Note that treatment of certain (or all) warnings as errors does not affect warning filtering – a warning message simply becomes an error message *if* it is decided that a warning message must be issued.

8 Appendix A: GNU Free Documentation License

Version 1.2, November 2002

```
Copyright (C) 2000,2001,2002  Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in

part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.