



Wind River Diab Compiler Assembler User's Guide 5.9.7

29 January 2020

WHEN IT MATTERS, IT RUNS ON **WIND RIVER**

Copyright Notice

Copyright © 2019 Wind River Systems, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of Wind River Systems, Inc.

Wind River, Simics, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. Helix, Pulsar, Rocket, Titanium Cloud, Titanium Control, Titanium Core, Titanium Edge, Titanium Edge SX, Titanium Server, and the Wind River logo are trademarks of Wind River Systems, Inc. Any third-party trademarks referenced are the property of their respective owners. For further information regarding Wind River trademarks, please see:

www.windriver.com/company/terms/trademark.html

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided for your product on the Wind River download and installation portal, Wind Share:

<http://windshare.windriver.com>

Wind River may refer to third-party documentation by listing publications or providing links to third-party websites for informational purposes. Wind River accepts no responsibility for the information provided in such third-party documentation.

Corporate Headquarters

Wind River
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.
Toll free (U.S.A.): +1-800-545-WIND
Telephone: +1-510-748-4100
Facsimile: +1-510-749-2010

For additional contact information, see the Wind River website:

www.windriver.com

For information on how to contact Customer Support, see:

www.windriver.com/support

Wind River Diab Compiler Assembler User's Guide 5.9.7

29 January 2020

TABLE OF CONTENTS

1. Wind River Diab Assembler.	1
1.1. About the Assembler.	1
1.2. das Command.	1
2. Assembly Syntax Rules.	3
2.1. Format of an Assembly Language Line.	3
2.1.1. Labels.	7
2.1.2. Opcode.	7
2.1.3. Operand Field.	8
2.1.4. Comment.	9
2.2. Symbols.	9
2.3. Reserved Symbols for ColdFire.	10
2.4. Direct Assignment Statements.	10
2.5. External Symbols.	11
2.6. Local Symbols.	14
2.7. Constants.	15
3. Sections and Location Counters.	19
3.1. Program Sections.	19
3.2. Location Counters.	20
4. Assembler Expressions.	22
4.1. Evaluation of Terms and Expressions.	22
4.2. Unary Operators.	23
4.2.1. High Adjust Operator for ARM, ColdFire, MIPS, PowerPC, RH850, and TriCore.	28
4.3. Binary Operators.	29
4.3.1. Operator Precedence.	30
5. Assembler Directives.	32
5.1. About Assembler Directives.	32
5.2. Assembler Directives.	32

6. Assembler Macros.	53
6.1. About Assembler Macros.	53
6.2. Macro Definition.	55
6.2.1. Syntax for ARM, ColdFire, MIPS, PowerPC, RH850, and x86.	55
6.2.2. Syntax for Tricore.	55
6.2.3. Optional Parameter Use.	56
6.2.4. Optional Parameter Examples: Referencing by Parameter Name.	56
6.2.5. Optional Parameter Examples: Referencing by Parameter Number.	58
6.2.6. Special Optional Parameter \0.	60
6.2.7. Separating Parameter Names From Text.	63
6.2.8. Generating Unique Labels.	65
6.2.9. NARG Symbol.	68
6.3. Invoking a Macro.	68
6.4. Macros to Define Structures.	69
7. Assembly Listing.	71
7.1. Assembler Listing Option.	71
7.2. Example Assembly Listing for File Swap.lst.	72
8. Instruction Mnemonics and Operand Addressing Modes.	76
8.1. Instruction Mnemonics.	76
8.2. Operand Addressing Modes.	78
8.2.1. Registers.	78
8.2.2. Expressions.	82


1. WIND RIVER DIAB ASSEMBLER

1.1. About the Assembler

The Wind River Diab assembler is most easily invoked using the **dcc** and **dplus** driver programs (often referred to simply as *drivers*).

For information in this regard, see the *Wind River Diab Compiler User's Guide*.

This guide describes the operation of the Wind River assembler (**das**). It does not describe the instruction set for a given architecture. For in-depth information on the architecture and instructions, please refer to the manufacturer's documentation.

 **Note:** For PowerPC, the **das** assembler understands both Book E instruction mnemonics and “**se_**” and “**e_**” forms. The compiler generates Book E mnemonics in preference to VLE-specific ones, unless it is absolutely necessary to do otherwise. For more on Book E and VLE instructions, see the documentation provided by your PowerPC processor's manufacturer.

The target for the assembler is selected by the same methods as for the compiler. For more information in this regard, see the *Wind River Diab Compiler User's Guide* and the *Wind River Diab Compiler Options Reference*. When using the compiler drivers **dcc**, **dplus**, etc., the target for the assembler is selected automatically by the driver. The ARM assembler supports the ARM Unified Assembler Language (UAL).

1.2. Das Command

Use the **das** command to execute the assembler.

The command is as follows:

```
das [options] [input-files]
```

where:

das

Invokes the assembler.

options

Command-line options. See the *Wind River Diab Compiler Options Reference: Assembler Options* for details. Options must precede the names of input files.

input-files

A list of filenames, paths permitted, separated by white space, naming the file(s) to be assembled; the default suffix is **.s**.

The assembler assembles the input file and generates an object file as determined by the selected target configuration. By default, the output file has the name of the input file with an extension suffix of **.o**. The **-o** option can be used to change the output filename.

The form **-@name** can also be used for either *options* or *input-file*. If found, the name must be either that of an environment variable or file (a path is allowed), the contents of which replace **-@name**.

Example: assemble **test.s** with a symbol named **DEBUG** equal to 2 for use in conditional assembly statements:

```
das -D DEBUG=2 test.s
```

2. ASSEMBLY SYNTAX RULES

2.1. Format of an Assembly Language Line

An assembly language file consists of a series of statements, one per line.

Format

The maximum number of characters in an assembly line is 1024. If -Xsemi-is-newline is enabled, a semicolon (;) can also be used as a statement separator (for ARM, ColdFire, MIPS, PowerPC, and x86).

The format of an assembly language statement is:

```
[label : ]      [opcode ]      [operand
      field ]      [# comment ]
```

The pound sign (#) is used here for the comment delimiter (substitute the appropriate symbol for your architecture if it differs).

Spaces and tabs may be used freely between fields and between operands (except that -Xspace-off option prohibits spaces between operands; see the -Xspace... entry in the Wind River Diab Compiler Options Reference).

A comment starts with "#" as shown above (for ARM, if using the preprocessor, then use "//" at the start of a line as "#" is misinterpreted as a preprocessor directive). See [Comment on page 9](#) for additional comment details.

All fields are optional depending on the circumstances. In particular:

- Blank lines are permitted.
- A statement may contain only a *label*.
- The *opcode* must be preceded by a label or white space (one or more blanks or tabs). A statement may contain only an *opcode*. (Assembler directives may start in column one but only if the -Xlabel-colon option is given.)
- A line may consist of only a *comment* beginning in any column.

For RH850, the dispose and prepare instructions accept register lists as operands. The rules for them are:

- A register list is enclosed in curly brackets {}, and is generally an unsorted list of register specifiers of the form rX, with 19 < X < 32.
- Duplicate entries are silently ignored by the assembler and not considered an error.
- Alias names as shown in the RH850 register list ([Registers on page 78](#)) are supported.

Assembly Code Examples

ARM

```
# mv_word(dest,src,cnt)
# move cnt (r2) 4-byte words from src (r1) to dest (r0)
.globl mv_word
.text mv_word:
    sub     r13,r13,#8
    mov     r3,r2
    cmp     r2,#0
    beq     .L2
```



```

        mov     r2,r2,ls1 #2
        add     r0,r0,r2
        add     r1,r1,r2
        add     r1,r1,#4
        add     r0,r0,#4
.L4:
        sub     r1,r1,#4
        sub     r0,r0,#4
        ldr     r12,[r1,#0]
        str     r12,[r0,#0]
        subs    r3,r3,#1
        bne     .L4
.L2:
        add     r13,r13,#8
        mov     pc,lr
        nop

```

ColdFire

Motorola (Freescale) Embedded Mnemonics (-Xmnem-emb):

```

; mv_word(dest,src,cnt)
; move cnt 4-byte words from src to dest
PSECT XDEF _mv_word
_mv_word:
    link a6,#-0
    move.l 8(a6),a1
    move.l 12(a6),a0
    move.l 16(a6),d0
    bra .L3
.L4:
    move.b 0(a0,d0),0(a1,d0)
.L3:
    subq.l #1,d0
    bne .L4
    unlk a6
    rts

```

MIT Mnemonics (-Xmnem-mit):

```

| mv_word(dest,src,cnt)
| move cnt 4-byte words from src to dest
.text .globl _mv_word
_mv_word:
    link a6,#-0
    movl a6@ (8),a1
    movl a6@ (12),a0
    movl a6@ (16),d0
    jra .L3
.L4:
    movb a0@ (d0:1:4),a1@ (d0:1:4)
.L3:
    subql #1,d0
    bne .L4
    unlk a6
    rts

```

Motorola (Freescale) UNIX Mnemonics (-Xmnem-moto):

```
# mv_word(dest,src,cnt)
# move cnt 4-byte words from src to dest text

global mv_word
mv_word:
    link %a6,&-0
    mov.l (8,%a6),%a1
    mov.l (12,%a6),%a0
    mov.l (16,%a6),%d0
    br L%3
L%4:
    mov.b (%a0)+, (%a1)+
L%3:
    subq.l #1,d0
    bne L%4
    unlk %a6
    rts
```

MIPS

```
# mv_word(dest,src,cnt)
# move cnt ($6) 4-byte words from src ($5) to dest ($4)
.globl mv_word
.text mv_word:
    blez    $6,.L2
    nop
.L3:
    lw      $25,0($5)
    addiu   $5,$5,4
    sw      $25,0($4)
    addiu   $6,$6,-1
    bne     $6,$0,.L3
    addiu   $4,$4,4
.L2:
    jr      $31
    nop
```

PowerPC

```
# mv_word(dest,src,cnt)
# move cnt (r5) 4-byte words from src (r4) to dest (r3)
.text .globl mv_word
mv_word:
    b       .L5
.L4:
    addic.  r5,r5,-1
    lwzx    r12,r4,r5
    stwx    r12,r3,r5
.L5:
    bne     .L4
    blr
```

RH850

```
# mv(word(dest,src,cnt)
# mv cnt (r8) 4-byte words from src (r7) to dest(r6)
.text .glob mv_word
```

```

mv_word:
    br .L5
.L4:
    ld.w 0[r7],r9
    st.w r9,0[r6]
    addi 4,r7,r7
    addi 4,r6,r6
    addi -1,r8,r8
.L5:
    bne .L4
    jmp
    [r31]

```

TriCore

```

##$m1 - ABI mnemonics
# TC - TriCore instructions
# TC12 - Optimized for TriCore ISA 1.2
# ko 0 - Reorder info .file "t.c"
# el
# sz 0 -Size opt info .text

                                # void mv_word(char* dest, char* src, int cnt)
                                # {
    bf
    fs 0
    .align 2
    .export mv_word
mv_word:
    ee                          #      while (cnt--) {
    jeq %d4,0,.L2
.L3:                            #      *dest++ = *src++;
    ld.b %d15,[%a5+]1
    st.b [%a4+]1,%d15
    jned %d4,1,.L3
.L2:
                                #      }
                                # }

    $be
    tl 0x0 ret
    ef .type mv_word,@function
    .size mv_word,.-mv_word # Number of nodes = 12
                                # Allocations for mv_word #
    %a4 dest                    #
    %a5 src                    #
    %d4 cnt                    # Allocations for module

```

x86

```

// mv_word(dest,src,cnt)
// move cnt (%ebx) 4-byte words from src (%edx) to dest (%eax)
.text .export mv_word
mv_word:
    pushl    %ebp
    movl     %esp, %ebp
    pushl    %ebx
    movl     16(%ebp), %ebx
.L4:

```

```

    testl    %ebx, %ebx      # if cnt is zero,
    je       .L2            #   return movl
    movl     %ebx, %edx      # move data ...
    shll     $2, %edx
    addl     12(%ebp), %edx
    movl     (%edx), %edx
    movl     %ebx, %eax
    shll     $2, %eax
    addl     8(%ebp), %eax
    movl     %edx, (%eax)
    decl     %ebx            # decrement cnt
    jmp      .L4            # back to top of loop
.L2:
    popl     %ebx
    popl     %ebp
    ret      return         # return

```

Labels on page 7

A *label* is a user-defined symbol which is assigned the value of the current location counter; both of which are entered into the assembler's symbol table. The value of the label is relocatable.

Opcode on page 7

The opcode of an assembly language statement identifies the statement as either a machine instruction or an assembler directive.

Operand Field on page 8

In general, an operand field consists of operands separated by commas. The number of operands vary by architecture.

Comment on page 9

The comment field consists of all characters in a source line including and following the comment character through the end of the line (the next new-line character). These characters are ignored by the assembler.

2.1.1. Labels

A *label* is a user-defined symbol which is assigned the value of the current location counter; both of which are entered into the assembler's symbol table. The value of the label is relocatable.

A label is a symbolic means of referring to a specific location within a program. The following govern labels:

- A label is a symbol; see [Symbols on page 9](#) for the rules on forming symbols.
- A label must occur first in a statement; but there may be multiple labels on one line.
- A label line must begin in column one (no preceding blank space).
- A label may be optionally terminated with a colon, unless the -Xlabel-colon option is used in which case the colon is required.
Examples:

```

start:
genesis: restart:    # Multiple labels
7$:                 # A local label
4:                  # A local label

```

(See [Local Symbols on page 14](#) for details on local labels.)

Parent topic: [Format of an Assembly Language Line on page 3](#)

2.1.2. Opcode

The opcode of an assembly language statement identifies the statement as either a machine instruction or an assembler directive.

The opcode must be preceded by a label or white space (one or more blanks or tabs). One or more blanks (or tabs) must separate the opcode from the operand field in a statement. No blanks are necessary between a label ending with a colon and an opcode. However, at least one blank is recommended to improve readability.

A machine instruction is indicated by an instruction mnemonic.

An assembler directive (or just "directive"), performs some function during the assembly process. It does not produce any executable code, although it may assign space in a program for data. Assembler directives may start in column one but only if the -Xlabel-colon option is given.

The assembler is case-insensitive regarding opcodes.

Parent topic: [Format of an Assembly Language Line on page 3](#)

2.1.3. Operand Field

In general, an operand field consists of operands separated by commas. The number of operands vary by architecture.

Architecture	Number of Operands
ColdFire, x86	0-2
ARM, MIPS	0-3
PowerPC, TriCore	0-5
RH850	0-14

The format of the operand field for machine instruction statements is the same for all instructions. The format of the operand field for assembler directives depends on the directive itself.

Our assembler accepts binary, octal, decimal and hex:

```
Decimal: addi 123,r8,r8
Hex: addi 0x7B,r8,r8
Octal: addi 0173,r8,r8
Binary: addi 0b1111011,r8,r8
```

The prefix for Hex, Octal and Binary cannot be omitted.

An immediate value has a sign if and only if it will be sign extended.

Example for RH850:

```
ADDI imm16, reg1, reg2
imm16: [-32768, 32767]

ADD imm5, reg2
imm5: [-16, 15]

SETF cccc, reg2
cccc: [0, 15]
```

Parent topic: [Format of an Assembly Language Line on page 3](#)

2.1.4. Comment

The comment field consists of all characters in a source line including and following the comment character through the end of the line (the next new-line character). These characters are ignored by the assembler.

Note

In addition to the architecture-specific characters described below, an asterisk "*" in column 1 is also treated as a comment delimiter.

ARM

For ARM, the comment delimiters are semicolon ";", at sign "@", pound sign "#". (To treat the semicolon as a statement separator, use -Xsemi-is-newline.)

In addition, the C++ comment marker "//" is valid at the start of a line and must be used there if pre-processing to select ARM vs. Thumb instructions (because the standard comment delimiter, "#", will be taken misinterpreted as a preprocessor directive if it begins a line). On the other hand, use "#" to begin a comment which is not the first text on the line (because "//" is invalid in that case).

ColdFire

For Coldfire, the comment delimiters are semicolon ";" (Embedded mnemonics) and vertical bar "|" (MIT mnemonics).

MIPS

For MIPS the comment delimiter is pound sign "#".

PowerPC, RH850

For PowerPC and RH850 the comment delimiters are pound sign "#" and semicolon ";". (To treat the semicolon as a statement separator, use -Xsemi-is-newline.)

TriCore, x86

For TriCore and x86 the comment delimiters are "//" (the C++ comment marker) and "#" (pound sign). Use "//" only at the beginning of a line (in column 1).

Parent topic: [Format of an Assembly Language Line on page 3](#)

2.2. Symbols

A symbol consists of a number of characters.

These characters have the following restrictions:

- Valid characters include A-Z, a-z, 0-9, period ".", dollar sign "\$", and underscore "_".

- The first character must not be a "\$" dollar sign.
- The first character must not be numeric except for local symbols ([Local Symbols on page 14](#)).

The only limit to the length of symbols is the amount of memory available to the assembler. Upper and lower cases are distinct: "Alpha" and "alpha" are separate symbols.

A symbol is said to be declared when the assembler recognizes it as a symbol of the program. A symbol is said to be defined when a value is associated with it. A symbol may not be redefined, unless it was initially defined with the directive `symbol .set expression` (see `symbol[:].set expression` in [Assembler Directives on page 32](#)).

There are several ways to define a symbol:

- As the label of a statement.
- In a direct assignment statement.
- With the `.equ/.set` directives.
- As a local common symbol via the `.lcomm` directive.

The `.comm` directive will declare a symbol as a common symbol. If a common symbol is not defined in any module, it will be allocated by the linker to the end of the `.bss` section. See *COMMON Sections* in the *Wind River Diab Compiler Linker User's Guide* for additional details.

2.3. Reserved Symbols for ColdFire

For ColdFire, in some cases conflicts can occur between processor register names and symbols. In order to avoid this, the assembler reserves these symbol names. Case is not significant, that is, all upper and lower case permutations are reserved.

Symbols	Description
d0 to d7	Data registers.
a0 to a7, sp	Address registers.
pc, zpc	Program counter register.
cc, ccr	Condition code registers.
sr	Status register.

2.4. Direct Assignment Statements

A direct assignment statement assigns the value of an arbitrary expression to a specified symbol.

The format of a direct assignment statement is one of the following:

`symbol[:] = expression`

`symbol[:] =: expression`

The `=:` syntax has the side effect that symbol will be visible outside of the current file. Examples of valid direct assignments are:

```

vect_size = 4
vectora = 0xfffe
vectorb = vectora-vect_size
CRLF: =: 0x0D0A

```

2.5. External Symbols

A program may be assembled in separate modules, and linked together to form a single program. By using external symbols, it is possible to define a label in one file and use it in another. The linker will relocate the reference so that the same address is used.

Forms of External Symbols

There are two forms of external symbols:

- Ordinary external symbols declared with the **.globl**, **.global**, **.xdef**, or **.export** directives.
- Common symbols declared with the **.comm** directive.

For example, the statements below define the array **table** and the routine **two()** to be external symbols.

ARM

```

                .globl  table, two
                .data
table:
                .space  20           # twenty bytes long
                .text
two:
    mov     r2,#2           # return 2
    mov     pc,lr

```

ColdFire

For Coldfire, high level languages such as C and C++ prefix external symbols with the underscore character “_” to avoid name clashes.

Embedded mnemonics:

```

XDEF _table, _two
    DSECT
_table:
    DS.B 20 ; twenty bytes long
    PSECT
_two:
    move.l #2,d0 ; return 2
    rts

```

MIT mnemonics:

```

.globl _table, _two
.data

```



```

_table:
    .space 20 | twenty bytes long
    .text
_two:
    movl #2,d0 | return 2
    rts

```

UNIX mnemonics:

```

global _table, _two
    data
_table:
    space 20 # twenty bytes long
    text
_two:
    mov.l &2,%d0 # return 2
    rts

```

MIPS

```

        .globl  table, two
        .data
table:
        .space  20                # twenty bytes long
        .text
two:
        addiu   $2,$0,2           # return 2
        jr      $31
        nop

```

PowerPC

```

table:
        .space  20                # twenty bytes long
        .text
two:
        addi    r3,r0,2           # return 2
        blr

```

RH850

```

table:
        .space  20                # twenty bytes long
        .text
two:
        mov     2,r10             # return 2
        jmp     [r31]

```

TriCore

```

table:
    .space 20          # twenty bytes long
    .text
two:
    mov     %d2,2      # return 2
    ret

```

x86

```

                .export table, two
                .text
table:
    .space 20          # twenty bytes long
    .text
two:
    pushl    %ebp
    movl     $2, %eax   # return 2
    popl     %ebp
    ret

```

External symbols are only declared to the assembler by the **.globl**, **.global**, **.xdef**, or **.export** directives. They must be defined (i.e., given a value) in another statement by one of the methods mentioned above. They need not be defined in the current file; in that case they are flagged as “undefined” in the symbol table. If they are undefined, they are considered to have a value of zero in expressions.

The following statements, which may be located in a different file, use the labels defined above.

ARM

```

b      two
ldr    r0,=table

```

ColdFire

```

jsr _two
move.l d0,_table

```

MIPS

```

jal     two
nop
lui     $5,%hi(table)
sw      $4,%lo(table)($5)

```

PowerPC

```
bl      two
addis   r4,r0,table@ha
stw     r3,r4,table@l
```

RH850

```
jarl     two,r31
movhi    table@ha,r0,r9
st.w     r8,table@l[r9]
```

TriCore

```
call     two
movh.a   %a4,table@ha
st.w     %d2,[%a4]table@l
```

x86

```
call     two
movl     %eax,(table)
pushl    (table)
```

Note that whenever a symbol is used that is not defined in the same file, it is considered to be a global undefined symbol by the assembler.

An external symbol is also declared by the **.comm** directive in one or more modules (see *.comm symbol, size [,alignment]* in [Assembler Directives on page 32](#)). For the rest of the assembly such a symbol, called a common symbol, will be treated as though it is an undefined global symbol. The assembler does not allocate storage for common symbols; this task is left to the linker. The linker computes the maximum size of each common symbol with the same name, allocates storage for it at the end of the final **.bss** section, and resolves linkages to it (unless the **-Xbss-common-off** is used; see the **-Xbss-off**, **-Xbss-common-off** entry in the Wind River Diab Compiler Options Reference).

2.6. Local Symbols

Local symbols provide a convenient way of generating labels for branch instructions. Use of local symbols reduces the possibility of attempting to define a symbol more than once in a program, and separates entry point symbols from local references, such as the top of a loop. Local symbols cannot be referenced by other object modules. The assembler implements two styles of local symbols.

Generic Style Locals

The generic style local symbols are of the form ***n*\$** where *n* is any integer.

Examples of valid local symbols:

```
1$
27$
394$
```

Leading zeroes are significant, e.g., **2\$** and **02\$** are different symbols. A local symbol is defined and referenced only within a single local symbol block. There is no conflict between local symbols with the same name which appear in different local symbol blocks. A new local symbol block is started when either:

- A non-local label is defined.
- A new program section is entered.

GNU-Style Locals

A GNU-style local symbol consists of one to five digits when defined. A GNU-style local symbol is referenced by the digits followed by the character **f** or **b**. When the digits are suffixed by an **f**, the nearest definition going forward (toward the end of the source) is referenced. When suffixed with the character **b**, the nearest definition going backward (toward the beginning of the file) is referenced. Example:

```
15:
        .long    15f      # Reference definition below.
        .long    15b      # Reference definition above.
15:
```

By default the GNU style local symbols are recognized by the assembler. This can be disabled with the option `-Xgnu-locals-off` (see the Wind River Diab Compiler Options Reference).

2.7. Constants

The assembler supports integral, floating point, and string constants. Integral constants may be entered in decimal, octal, binary or hexadecimal form, or they may be entered as character constants. Floating point constants can only be used with the **.float** and **.double** directives.

Integral Constants

Internally, the assembler treats all integer constants as signed 32-bit binary two's complement quantities. Valid constant forms are listed below. The order of the list is significant in that it is scanned from top to bottom, and the first matching form is used.

'c'

character constant

0x*hex-digits*

hexadecimal constant

0*octal-digits*

octal constant

\$*hex-digits*

hexadecimal constant (for ColdFire, PowerPC, RH850, TriCore, x86)

/hex-digits

hexadecimal constant

@octal-digits

octal constant

%binary-digits

binary constant

decimal-digits

decimal constant

octal-digits o

octal constant

octal-digits q

octal constant

binary-digits b

binary constant

Examples:

```

abc = 12          12 decimal
bcd = 012         12 octal (10 decimal)
cde = 0x12        12 hex (18 decimal)

```

To represent special character constants, use the following escape sequences:

Constant	Value	Meaning
'\b'	8	backspace
'\t'	9	horizontal tab
'\n'	10	line feed (newline)
'\v'	11	vertical tab
'\f'	12	form feed
'\r'	13	return
'\''	39	single quote
'\\'	92	backslash

By using a "\nnn" construct, where *nnn* is an octal value, any character can be specified:

```
'\101'    same as 'A' (65 decimal)
'\60'     same as '\0' (48 decimal)
```

Floating Point Constants

Floating point constants have the following format:

```
[+|-].i{i}{e|E}[+|-]i
```

where *i* is an integer. All parts are optional as long as the constant starts with a sign or a digit and contains either a decimal point or an exponent (**e** or **E** and a following digit). Also, + **NAN** and [+/-] **INF** are supported. Examples:

```
float    1.2, -3.14, 0.27172e1
double  -123e-45, .56, 1e23
```

String Constants

The form of a string is:

`"characters"`

where *characters* is one or more printable characters or escape codes.

Characters represented in the source text with internal values less than 128 are stored with the high bit set to zero. Characters with source text values from 128 through 255, and characters represented by the `"\nnn"` construct are stored as is.

A newline character must not appear within the character string. It can be represented by the escape sequence `\n` as described below. The `("`) is a delimiter character and must not appear in the string unless preceded by a backslash `"\"`.

Assigning a string constant to a variable has no effect, as in this example:

```
$ cat d.s
    .text
    .ifeq D=="b"
    nop
    .endif

$ das -L -tPPCES -DD="b" d.s
"d.s", line 2: error: syntax error
"d.s", line 4: error: endif statement without leading if
" 1          .text
                                2          .ifeq D=="b"
00000000 00 6000 0000          3          nop
                                4          .endif
```

Here, the **das** command is for a PowerPC target (substitute the target configuration specification for other architectures).

(See also the `-D` entry in the *Wind River Diab Compiler Options Reference: Assembler Options*.)

The following escape sequences are also valid as single characters:

Constant	Value	Meaning
<code>\b</code>	8	Backspace
<code>\t</code>	9	Horizontal tab
<code>\n</code>	10	Line Feed (New Line)
<code>\v</code>	11	Vertical tab
<code>\f</code>	12	Form feed
<code>\r</code>	13	Enter
<code>\"</code>	34	Double quote <code>""</code>
<code>\\</code>	92	Backslash <code>"\"</code>
<code>\nnn</code>	<i>nnn</i> (octal)	Octal value of <i>nnn</i>

Some examples follow. The final two are equivalent.

Statement	Hex Code Generated
<code>.ascii "hello there"</code>	68 65 6C 6C 6F 20 74 68 65 72 65
<code>.ascii "Warning-\007\007\n"</code>	77 61 72 6E 69 6E 67 2D 07 07 0A
<code>.ascii "Warning-",7,7,"n"</code>	Same as previous line.

Literals as Operands for ColdFire

For Coldfire, a pound sign `"#"` must precede a literal that is an operand.

```
move.l #2,d0 ; return 2
move.l #0x100,d0 ; return 256
```

3. SECTIONS AND LOCATION COUNTERS

3.1. Program Sections

Assembly language programs are usually divided into sections to separate executable code from data, constant data from variable data, initialized data from uninitialized data, etc.

Predefined Sections

Some important predefined sections are described below, with a reference to the assembler directive that switches output to each section.

.text

Instruction space.

.text_vle

For PowerPC. Instruction space for processors using the VLE (Variable Length Encoding) instruction set.

If you want to create sections for VLE instructions that are not named **.text_vle**, you must:

- Specify that the section type for these sections is **TEXT** (see **.section name, [alignment], [type]**, for how to do this in an assembler directive, or see the *Section-Definition* section in the *Wind River Diab Compiler Linker User's Guide*, for how to do this in a linker command file).
- Specify a VLE-based target on the command line (e.g., -tPPCVLE).

.data

Initialized data.

.bss

Uninitialized data.

.sbss [symbol, size [,alignment]]

For ColdFire, MIPS, PowerPC, RH850, and TriCore. Short uninitialized data.

.rdata

For MIPS and RH850. Read-only data.

.rodata

For ARM, ColdFire, PowerPC, TriCore, and x86. Read-only data.

.sdata

For ColdFire, MIPS, PowerPC, RH850, and TriCore. Short initialized data.

.sdata2

For ColdFire, MIPS, PowerPC, and RH850. Constant short initialized data.

By invoking these directives, it is possible to switch among the sections of the assembly language program. New sections can also be defined with the **.section** directive (see **.section name, [alignment], [type]** in [Assembler Directives on page 32](#)).

The assembler maintains a separate location counter for each section. Thus for assembly code such as:

```
.text
instruction-block-1
.data
data-block-1
.text
instruction-block-2
.data
data-block-2
```

In the object file, *instruction-block-2* will immediately follow *instruction-block-1*, and *data-block-2* will immediately follow *data-block-1*.

ELF sections are aligned based on their contents or on a specified alignment in a **.section** directive. ELF sections are not extended to any boundary whether aligned or not.


For COFF, sections are aligned and sized to the first of:

- an alignment specified with a **-.section** directive
- the value given with an **-Xdefault-align** command-line option
- a default value of 8.

(For more information about **-Xdefault-align**, see the Wind River Diab Compiler Options Reference)


Padding introduced into a code section (but not other types of sections) by means of an **.align** or **.alignn** directive (or for COFF, at the end of a section) is as follows:

Architecture	Operation
ColdFire	Filled with the nop instruction (0x4e71).
ARM, MIPS, and TriCore	Zero-filled (the nop instruction).
PowerPC	Filled with the nop instruction (0x60000000).
RH850	Filled with the nop instruction (0x00000000).
x86	Filled with the nop instruction (0x90).

 **Note:** See the **-f** linker option, the *dld Linker Command* chapter of the *Wind River Diab Compiler Linker User's Guide*, for filling of gaps between input sections in an output section.

3.2. Location Counters

The assembly current location counter is represented by the character **"."** or—for PowerPC, MIPS, and RH850 only—the character **"\$"**. In the operand field of any statement or assembly directive it represents the address of the first byte of the statement.

 **Note:** A current location counter appearing as an operand in a **.byte** directive (see *.byte expression*, ... in [Assembler Directives on page 32](#)) always has the value of the address at which the first byte was loaded; it is not updated while evaluating the directive.

The assembler initializes the location counter to zero. Normally, consecutive memory locations are assigned to each byte of the generated code. However, the location where the code is stored may be changed by a direct assignment altering the location counter:

```
. = expression
```

expression must not contain any forward references, must not change from one pass to another, and must not have the effect of reducing the value of `"."`. Note that the assembler supports absolute sections when using ELF, so setting `"."` to an absolute position is equivalent to using the **.org** directive and will produce a section named **.abs.xxxxxxxx**, where **xxxxxxx** is the hexadecimal address of the section, with leading zeros to fill to eight digits. The linker will then place this section at the specified address. For example, this will create a section named **.abs.00ff0000** located at the specified address:

```
. = 0xff0000
```

Storage area may also be reserved by advancing the `"."`. For example, if the current value of `"."` is 0x1000, the following would reserve 100 (hex) bytes of storage:

```
. = . +0x100
```

The next instruction would be stored at address 0x1100 (which is a more readable way of doing the same thing):

```
.skip 0x100
```

4. ASSEMBLER EXPRESSIONS

4.1. Evaluation of Terms and Expressions

Expressions are combinations of terms joined together by unary or binary operators. An expression is always evaluated to a 32-bit value. If the instruction calls for only 8 or 16 bits, the least significant 8 or 16 bits are used.

A term is a component of an expression. A term may be one of the following:

- A constant
- A symbol
- An expression or term enclosed in parentheses (). Any quantity enclosed in parentheses is evaluated before the rest of the expression. This can be utilized to alter the normal precedence of operators, e.g., differentiating between $a*b+c$ and $a*(b+c)$, or to apply a unary operator to an entire expression, e.g., $-(a*b+c)$.

Any expression, when evaluated, is either *absolute* or *relocatable*:

1. An expression is *absolute* if its value is fixed. An expression whose terms are constants, or symbols whose values are constants via a direct assignment directive, is absolute. A relocatable expression minus a relocatable expression, where both items belong to the same program section is also absolute.
2. An expression is *relocatable* if it contains a label whose value will not be defined until link time. In this case the assembler will generate an entry in the relocation table in the object file. This entry will point to the instruction or data reference so that the linker can patch the correct value after memory allocation. The allowed relocatable expressions are defined in F. Object and Executable File Formats together with the relocation type used. The following demonstrates the use of relocatable expressions, where "alpha" and "beta" are symbols:

alpha

relocatable

alpha+5

relocatable

alpha-0xa

relocatable

alpha*2

not relocatable (error)

2-alpha

not relocatable, since the expression cannot be linked by adding alpha's offset to it

alpha-beta

absolute, since the distance between alpha and beta is constant, as long as they are defined in the same section

alpha@l


ColdFire, PowerPC, RH850, Tricore: relocatable (the low 16 bits of alpha)

%lo(alpha)

MIPS: relocatable (the low 16 bits of alpha)

4.2. Unary Operators

The unary operators recognized by the assembler are described below.

 **Note:** In descriptions of the unary operators, phrases like “**expr** evaluates to ... offset from the ... base register” mean that the assembler generates a constant that is adjusted as necessary by the linker so that the final value in memory is an offset from the designated base register. These constructs are used for position-independent code or , or for small data or constant areas (both “position-independence” and “small” are implemented using the same mechanisms and registers) . To execute correctly, the designated base register must be loaded with the base of the data area as appropriate.

ARM

%endof(*section-name*)

Address of the end of the given section. Evaluates to **.endof***section_name*, a symbol created by the linker. (See *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*.)

expr **@data**

expr evaluates to a 32 bit offset from the data base register (**r9**), typically initialized to **__SDA_BASE__**.

expr **@h**

The most significant 16 bits of *expr* are extracted. Provided for use in assembly language, but not generated by the compiler.

expr **@ha**

High adjust: The most significant 16 bits of *expr* are extracted and adjusted for the sign of the least significant 16 bits of *expr*. See [High Adjust Operator for ARM, ColdFire, MIPS, PowerPC, RH850, and TriCore on page 28](#) below. Provided for use in assembly language, but not generated by the compiler.

expr **@l**

The least significant 16 bits of *expr* are extracted.

%sizeof(*section-name*)

Size of the given section. Evaluates to **.sizeof***section_name*, a symbol created by the linker (see *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*).

%startof(*section-name*)

Address of the start of the given section. Evaluates to **.startof***section_name*, a symbol created by the linker (see *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*).

+	unary add
-	negate
~	complement

ColdFire

.ENDOF(*section-name*)

Address of the end of the given section. Evaluates to **.endof.section_name**, a symbol created by the linker. (See the *Wind River Diab Compiler Linker User's Guide*.)

expr **@h**

The most significant 16 bits of *expr* are extracted.

expr **@ha**

High adjust: The most significant 16 bits of *expr* are extracted and adjusted for the sign of the least significant 16 bits of *expr*. See [High Adjust Operator for ARM, ColdFire, MIPS, PowerPC, RH850, and TriCore on page 28](#).

expr **@l**

The least significant 16 bits of *expr* are extracted.

expr **@sda**

The least significant 16 bits of the 32-bit *expr* are extracted, then the 16-bit *expr* is evaluated relative to the "small data area" base register **a5**.

expr **@sdax**

The least significant 16 bits of the 32-bit *expr* are extracted, then the 16-bit *expr* is evaluated relative to the "small data area" base register **a5**.

.SIZEOF(*section-name*)

Size of the given section. Evaluates to **.sizeof.section_name**, a symbol created by the linker (see the *Wind River Diab Compiler Linker User's Guide*).

.STARTOF(*section-name*)

Address of the start of the given section. Evaluates to **.startof.section_name**, a symbol created by the linker (see the *Wind River Diab Compiler Linker User's Guide*).

+	unary ad
-	negate
~	complement

MIPS

%endof(*section-name*)

Address of the end of the given section. Evaluates to **.endof.section_name**, a symbol created by the linker. (See the *Wind River Diab Compiler Linker User's Guide*.)

expr **@h**

The most significant 16 bits of *expr* are extracted.

%hiadj(*expr*) **%hi**(*expr*) *expr* **@ha**

High adjust: The most significant 16 bits of *expr* are extracted and adjusted for the sign of the least significant 16 bits of *expr*. See [High Adjust Operator for ARM, ColdFire, MIPS, PowerPC, RH850, and TriCore on page 28](#).

For users of other architectures: usually, the **%hi** operator is a synonym for **@h**. For MIPS, per the manufacturer's manual, **%hi** "adjusts" as described below; the compiler also provides **@h** for the unadjusted value.

%lo(*expr*) *expr* **@l**

The least significant 16 bits of *expr* are extracted.

%piclo(*expr*)

The least significant 16 bits of the 32 bit *expr* are extracted and adjusted for the sign of the least significant 16 bits.

Then *expr* is evaluated relative to the "small constant area" base register **\$23**.

%scaoff(*expr*) **%picoff**(*expr*)

The least significant 16 bits of the 32 bit *expr* are extracted.

Then *expr* is evaluated relative to the "small constant area" base register **\$23**.

%pichi(*expr*)

The most significant 16 bits of the 32 bit *expr* are extracted and adjusted for the sign of the least significant 16 bits.

Then *expr* is evaluated relative to the "small constant area" base register **\$23**.

%pidhi(*expr*)

The most significant 16 bits of the 32 bit *expr* are extracted, and if the least significant 16 bits are negative, one is added to the upper 16 bits.

Then *expr* is evaluated relative to the "small data area" base register **\$28**.

%pidlo(*expr*)

The least significant 16 bits of the 32 bit *expr* are extracted.

Then *expr* is evaluated relative to the "small data area" base register **\$28**.

%sdaoff(*expr*) **%pidoff**(*expr*)

The least significant 16 bits of the 32 bit *expr* are extracted.

Then 16 bit *expr* is evaluated relative to the "small data area" base register **\$28**.

%sizeof(*section-name*)

Size of the given section. Evaluates to **.sizeof.section_name**, a symbol created by the linker (see *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*).

%startof(*section-name*)

Address of the start of the given section. Evaluates to **.startof.section_name**, a symbol created by the linker (see *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*).

+	unary add
-	negate
~	complement

PowerPC

%endof(*section-name*)

Address of the end of the given section. Evaluates to **.endof***section_name*, a symbol created by the linker. (See *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*.)

expr **@h %hi**(*expr*)

The most significant 16 bits of *expr* are extracted.

expr **@ha %hiadj**(*expr*)

High adjust: the most significant 16 bits of *expr* are extracted and adjusted for the sign of the least significant 16 bits of *expr*. See [High Adjust Operator for ARM, ColdFire, MIPS, PowerPC, RH850, and TriCore on page 28](#) below.

expr **@l %lo**(*expr*)

The least significant 16 bits of *expr* are extracted.

expr **@sdx %sdaoff**(*expr*)

The least significant 16 bits of the 32 bit *expr* are extracted.

Then 16 bit *expr* is evaluated relative to the "small data area" base register **r13**.

expr **@sdax**

The least significant 16 bits of the 32 bit *expr* are extracted.

Then 16 bit *expr* is evaluated relative to the "small constant area" base register **r2**.

Size of the given section. Evaluates to **.sizeof***section_name*, a symbol created by the linker (see *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*).

%sizeof(*section-name*)

Size of the given section. Evaluates to **.sizeof***section_name*, a symbol created by the linker (see *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*).

%startof(*section-name*)

Address of the start of the given section. Evaluates to **.startof***section_name*, a symbol created by the linker (see *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*).

+	unary add
-	negate
~	complement

RH850

%endof(*section-name*)

Address of the end of the given section. Evaluates to **.endof***section_name*, a symbol created by the linker. (See *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*.)

expr **@h %hi**(*expr*)

The most significant 16 bits of *expr* are extracted.

`expr @ha %hiadj(expr)`

High adjust: the most significant 16 bits of `expr` are extracted and adjusted for the sign of the least significant 16 bits of `expr`. See [High Adjust Operator for ARM, ColdFire, MIPS, PowerPC, RH850, and TriCore on page 28](#) below.

`expr @l %lo(expr)`

The least significant 16 bits of `expr` are extracted.

`expr @data`

`expr` is evaluated relative to the "small data area" base register **r4**.

`%sizeof(section-name)`

Size of the given section. Evaluates to `.sizeof.section_name`, a symbol created by the linker (see *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*).

`%startof(section-name)`

Address of the start of the given section. Evaluates to `.startof.section_name`, a symbol created by the linker (see *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*).

+	unary add
-	negate
~	complement

TriCore

`%endof(section-name)`

Address of the end of the given section. Evaluates to `.endof.section_name`, a symbol created by the linker. (See *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*.)

`expr @h %hi(expr)`

The most significant 16 bits of `expr` are extracted.

`expr @ha %hiadj(expr)`

High adjust: the most significant 16 bits of `expr` are extracted and adjusted for the sign of the least significant 16 bits of `expr`. See [High Adjust Operator for ARM, ColdFire, MIPS, PowerPC, RH850, and TriCore on page 28](#) below.

`expr @l %lo(expr)`

The least significant 16 bits of `expr` are extracted.

`expr @sdarx %sdaoff(expr)`

The least significant 16 bits of the 32 bit `expr` are extracted.

Then 16 bit `expr` is evaluated relative to the "small data area" base register **%a0**.

`expr @sdax`

The least significant 16 bits of the 32 bit `expr` are extracted.

Then 16 bit `expr` is evaluated relative to the "small constant area" base register **%a1**.

%sizeof(*section-name*)

Size of the given section. Evaluates to **.sizeof.section_name**, a symbol created by the linker (see *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*).

%startof(*section-name*)

Address of the start of the given section. Evaluates to **.startof.section_name**, a symbol created by the linker (see *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*).

+	unary add
-	negate
~	complement

x86**.ENDOF**(*section-name*)

Address of the end of the given section. Evaluates to **.endof.section_name**, a symbol created by the linker. (See *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*.)

.SIZEOF(*section-name*)

Size of the given section. Evaluates to **.sizeof.section_name**, a symbol created by the linker (see *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*).

.STARTOF(*section-name*)

Address of the start of the given section. Evaluates to **.startof.section_name**, a symbol created by the linker (see *Symbols Created By the Linker* in the *Wind River Diab Compiler Linker User's Guide*).

+	unary add
-	negate
~	complement

High Adjust Operator for ARM, ColdFire, MIPS, PowerPC, RH850, and TriCore on page 28

Sometimes the compiler (or a hand-coded assembly language program, if not the compiler) uses two instructions to copy an address to or from a location in memory. Each instruction can include 16 bits of the address as an immediate value, and the two 16-bit parts of the address are added to form the full address.

4.2.1. High Adjust Operator for ARM, ColdFire, MIPS, PowerPC, RH850, and TriCore

Sometimes the compiler (or a hand-coded assembly language program, if not the compiler) uses two instructions to copy an address to or from a location in memory. Each instruction can include 16 bits of the address as an immediate value, and the two 16-bit parts of the address are added to form the full address.

For the purposes of this discussion:

- The first instruction has the higher 16 bits of the address.
- The second instruction has the lower 16 bits of the address.

In some cases, the second instruction sign extends the low 16 bits (for example, 0x8000 is sign-extended to 0xffff8000). If so, the first instruction must compensate so that the correct address is calculated when the two parts of the address are added.

How the first instruction compensates depends on the most significant bit of the lower 16 bits of the address:

- If it is zero, no adjustment is made.
- If it is 1, the first instruction adds 1 to the higher 16 bits of the address. The second instruction adds 0xffff, which is equivalent to -1. Thus, the two additions negate each other.

Parent topic: [Unary Operators on page 23](#)

4.3. Binary Operators

The *binary* operators recognized by the assembler are:

Binary Operator	Description
+	add
-	subtract
*	multiply
/	divide
	bitwise OR (ARM, MIPS, PowerPC, RH850, TriCore, x86)
!	bitwise OR (ColdFire)
%	modulo
&	bitwise AND
^	bitwise exclusive OR
<<	shift left
>>	shift right
==	equal to
!=	not equal to
<=	less than or equal to
<	less than
>=	greater than or equal to
>	greater than

Operator Precedence on page 30

Expressions are evaluated with the following precedence in order from highest to lowest. All operators in each row have the same precedence.

4.3.1. Operator Precedence

Expressions are evaluated with the following precedence in order from highest to lowest. All operators in each row have the same precedence.

Table 1. Assembler Operator Precedence and Associativity

Architecture	Operator	Associativity
All	unary + - ~	right to left
ARM	@code @data @h @ha @l %startof %endof %sizeof	left to right
ColdFire	@h @ha @l %sda %sdax .startof. .endof. .sizeof	left to right
MIPS	@h @ha %hi %hiadj @l %lo %pichi %piclo %picoff %pidhi %pidlo %pidoff %scaoff %sdaoff %startof %endof %sizeof	left to right
PowerPC	@h @ha %hi %hiadj @l %lo %sdaoff %sdarx %sdax %startof %endof %sizeof	left to right
RH850	@h @ha %hi %hiadj @l %lo %code %data %tidata %startof %endof %sizeof	left to right
TRICORE	@h @ha %hi %hiadj @l %lo %sdaoff %sdarx %sdax %startof %endof %sizeof	left to right
x86	.startof. .endof. .sizeof.	left to right
All	* / % (modulo)	left to right
	binary + -	left to right
	<< >>	left to right
	< <= > >=	left to right
	== !=	left to right
	&	left to right

Architecture	Operator	Associativity
	\wedge	left to right
		left to right

Parent topic: [Binary Operators on page 29](#)

5. ASSEMBLER DIRECTIVES

5.1. About Assembler Directives

All the assembler directives (or just "directives") described here that are prefixed with a period "." are also available without the period. Most are shown with a "." except for those traditionally written without it.

The -Xlabel-colon option controls whether labels must be terminated with colons (see the *Wind River Diab Compiler Options Reference*). If -Xlabel-colon is set, then directives that cannot take a label may start in column 1. A directive that can take a label—that is, can produce data in the current section—may not start in column 1.

Spaces are optional between the operands of directives unless the -Xspace-off option is in force (see the *Wind River Diab Compiler Options Reference*).

In addition to the directives documented in this chapter, the assembler recognizes the following directives generated by some compilers for symbolic debugging:

.d1_line_end	.d1_line_start	.d1file
.d1line	.d2_cfa_offset	.d2_cfa_offset_list
.d2_cfa_register	.d2_cfa_same_value	.d2_cfa_same_value_list
.d2_line_end	.d2_line_start	.d2file
.d2line	.d2string	.def
.dim	.endef	.line
.ln	.scl	.size
.sleb128	.tag	.type
.uleb128	.val	

5.2. Assembler Directives

The assembler supports a set of directives.

symbol[:] = expression

See [symbol\[:\] =: expression on page 33](#). See the -Xlabel-colon... entry in the *Wind River Diab Compiler Options Reference* regarding the initial colon.

symbol[:] =: expression


Equivalent to `symbol = expression` except that `symbol` will be made a global symbol. See the `-Xlabel-colon...` entry in the *Wind River Diab Compiler Options Reference* regarding the initial colon.

symbol[:] .equ expression

The statement must be labeled with a symbol and sets the symbol to be equal to `expression`. See `-Xlabel-colon...` in the *Wind River Diab Compiler Options Reference*, regarding the initial colon. Example:

```
nine:      .equ      9
```

If the expression contains a single symbol, the newly-defined symbol will have the same size and type as the symbol in the expression. This can be used to create an alias of a function or data symbol.

 **Note:** Symbols defined with `.equ` may not be redefined. Use the second form of the `.set` directive in [.set symbol, expression on page 48](#), instead of `.equ` if redefinition is required.

.0byte

Used to tell the linker that this section depends on another. This is needed so the linker option `-Xremove-unused-sections` doesn't delete anything important.

.2byte

This is a synonym for `.short` ([.short expression ,... on page 48](#)) except that there are no alignment restrictions and an unaligned relocation type will be generated if required by the target.

.4byte

This is a synonym for `.long` ([.long expression ,... on page 43](#)) except that there are no alignment restrictions and an unaligned relocation type will be generated if required by the target.

.align expression

Aligns the current location counter to the value given by `expression` (which must be absolute). When the option `-Xalign-value` is set, `expression` is used as the alignment value, and must be a power of 2. When the option `-Xalign-power2` is set, the alignment value is 2 to the power of `expression`.

For ColdFire and x86, the default is `-Xalign-value`.

For ARM, MIPS, PowerPC, RH850, and TriCore, the default is `-Xalign-power2`.

There is no effect if the current location is already aligned as required.

In a section of type **TEXT**, if a "hole" is created, it will be handled as follows:

Architecture	Operation
ColdFire	Filled with the nop instruction (0x4e71) unless a different value is specified with -Xalign-fill-text.
ARM, MIPS, TriCore	Zero-filled (the nop instruction) unless a different value is specified with -Xalign-fill-text.
PowerPC	filled with the nop instruction (0x60000000) unless a different value is specified with -Xalign-fill-text.
RH850	Filled with the nop instruction (0x00000000) unless a different value is specified with -Xalign-fill-text.
x86	Filled with the nop instruction (0x90) unless a different value is specified with -Xalign-fill-text.

Example:

```
.align 4
```

With -Xalign-value, aligns on a 4-byte boundary; with -Xalign-power2, aligns on a $2^4 = 16$ -byte boundary.

.alignn expression

Aligns the current location counter to the value given by *expression* (which must be absolute).

There is no effect if the current location is already aligned as required.

In a section of type **TEXT**, if a "hole" is created, it will be handled as follows:

Architecture	Operation
ColdFire	Filled with the nop instruction (0x4e71) unless a different value is specified with -Xalign-fill-text.
ARM, MIPS, TriCore	Zero-filled (the nop instruction) unless a different value is specified with -Xalign-fill-text.
PowerPC	filled with the nop instruction (0x60000000) unless a different value is specified with -Xalign-fill-text.
RH850	Filled with the nop instruction (0x00000000) unless a different value is specified with -Xalign-fill-text.
x86	Filled with the nop instruction (0x90) unless a different value is specified with -Xalign-fill-text.

Example:

```
.alignn 4
```

Will align on 4 byte boundary.

.ascii "string"

The **.ascii** directive stores the internal representation of each character in the string starting at the current location. See the *String Constants* section in [Constants on page 15](#) for rules for writing the "string".

The **.ascii** directive is actually a synonym of the **.byte** directive — its operands may be a list of expressions including non-strings. See **.byte** for details ([.byte expression ,... on page 35](#)).

.asciz "string"

The **.asciz** directive is equivalent to the **.ascii** directive with a zero (null) byte automatically appended as the final character of the string. In the C language, strings are null terminated. See the *String Constants* section in [Constants on page 15](#) for rules for writing the "string".

.balign expression

Equivalent to [.align expression on page 33](#).

.blkb expression

Equivalent to [.skip size on page 49](#).

.blkl

Defines a block of 32 bit words.

.blkw

Defines a block of 16 bit words.

.bss

Switches output to the **.bss** section. Note that **.bss** contains uninitialized data only, which means that the **.skip**, **.space**, and **ds.b** directives are the only useful directives inside the **.bss** section.

.bsect

Equivalent to [.bss on page 35](#).

.byte expression ,...

Reserves one byte for each expression in the operand field and initializes the value of the byte to be the low-order byte of the corresponding expression. Multiple expressions are separated by commas.

Any expression may be a string containing one or more characters. Each character in the string will be allocated one byte. See the *String Constants* section in [Constants on page 15](#) for the rules for writing a string.

Example:


```
.byte 17,65,0101,0x41      # sets 4 bytes
      .byte 0               # sets a single byte to 0
      .byte 7,7,"Warning",7,7,0 # sets 12 bytes
```

The pound sign (#) is used here for the comment delimiter (substitute the appropriate symbol for your architecture if it differs).

.comm symbol, size [,alignment]

Define *symbol* as the address of a common block with length given by expression *size* bytes and make it global. Contrast with **.lcomm**, (**.lcomm symbol, size [,alignment]** on page 42) which does not make the symbol externally visible.

The *size* and *alignment* expressions must be absolute. If the **-Xlabel-colon** option is set, this directive may begin in the first column.

All common blocks with the same name in different files will refer to the same block. The linker will collect and allocate space for all common blocks, and, by default, place this space at the end of the **.bss** section; see the *Wind River Diab Compiler Linker User's Guide* for details.

The optional *alignment* expression specifies the alignment of the common block. It must be absolute. If not specified, the default value equals the greatest power of 2 which is less than or equal to the minimum of *size* and the value specified by **-Xdefault-align**, which defaults to 8 (see the *Wind River Diab Compiler Options Reference*).

See the **-Xalign-value**, **-Xalign-power2** entry in the *Wind River Diab Compiler Options Reference* for options for giving the alignment by power of 2 or the value specified. The default is as follows:

- ColdFire, x86: Align on the value specified.
- ARM, MIPS, PowerPC, RH850, TriCore: Treat the *alignment* value as a power of 2.

Note that the COFF object file does not support common block alignment. An *alignment* value is ignored for COFF object files. See the **-Xcommon-align** entry in the *Wind River Diab Compiler Options Reference*.

The following examples assume **-Xdefault-align=8**.

ColdFire, x86

```
.comm a1,100      # 100 bytes aligned on an 8-byte boundary
.comm a2,7,4      # 7 bytes aligned on a 4-byte boundary
```

The pound sign (#) is used here for the comment delimiter (substitute the appropriate symbol for your architecture if it differs).

ARM, MIPS, PowerPC, RH850, TriCore

```
.comm a1,100      # 100 bytes aligned on an 8-byte boundary.
.comm a2,7,2      # 7 bytes aligned on a 4-byte boundary.
```

The pound sign (#) is used here for the comment delimiter (substitute the appropriate symbol for your architecture if it differs).

.cross

Turns on cross reference listing if the option **-L** or **-l** is specified. Cross reference listing can be turned off with **.nocross**.

.data

Switches output to the **.data** (initialized data) section.

data.l

Equivalent to [.long expression ,... on page 43](#).

data.w

Equivalent to [.short expression ,... on page 48](#).

dc.b expression

Equivalent to [.byte expression ,... on page 35](#).

dc.l expression

Equivalent to [.long expression ,... on page 43](#).

dc.w expression

Equivalent to [.short expression ,... on page 48](#).

dc.x

Equivalent to [.extended on page 39](#).

ds.b size

Equivalent to [.skip size on page 49](#).

ds.l

Equivalent to [.blkl on page 35](#).

ds.w

Equivalent to [.blkw on page 35](#).

.double float-constant ,...

Reserves space and initializes double 64-bit IEEE floating point values.

Example:

```
double 1.0, -123.45e-56
```

.dsect

Equivalent to [.data on page 36](#).

.eject

Forces a page break if a listing is produced by the -L or -l options. See chapter 7: *Assembly Listing* for an example of an assembly listing.

.else

The **.else** directive is used with the **.ifx** directives to reverse the state of the conditional assembly, i.e., if statements were skipped prior to the **.else** directive, statements following the **.else** directive will be processed, and vice versa. See [.if expression on page 40](#) for an example.

.elseif expression

The **.elseif** directive must follow a **.ifx** or another **.elseif** directive in a conditional assembly block. If all prior conditions (at the same nesting level) have been false, then the *expression* will be tested and if non-zero, the statements following it assembled, else statements will be skipped until the next **.elseif**, **.else**, or **.endif** directive. The *expression* must be absolute. See [.if expression on page 40](#) for an example.

.elsec

Equivalent to [.else on page 38](#).

.end

This directive indicates the end of the source program. All characters after the end directive are ignored.

.endc

Equivalent to [.endif on page 38](#).

.endif

This directive indicates the end of a condition block; each **.endif** directive must be paired with a **.ifx** directive. See [.if expression on page 40](#) for an example.

.endm

This directive indicates the end of a macro body definition. Each **.endm** directive must be paired with a **.macro** directive. See chapter 6: *Assembler Macros* for a detailed description.

.entry symbol ,...


Equivalent to [.global symbol ,... on page 40](#).

.error "string"

Generate an error message showing the given string. See the *String Constants* section in [Constants on page 15](#) for rules for writing the "string". If the -Xlabel-colon option is set, this directive may begin in the first column.

.even

Aligns the location counter on the default alignment value, specified by the `-Xdefault-align` option (see the *Wind River Diab Compiler Options Reference*).

 **Note:** For Coldfire, use the compiler option `-Xalign-off` when compiling to force the compiler to generate **.even** directives, the alignment of which can be controlled by the assembler option `-Xdefault-align` rather than **.align** directives. See the *Wind River Diab Compiler Options Reference* for information about these options.

.exitm

Exit the current macro invocation. If the `-Xlabel-colon` option is set, this directive may begin in the first column.

.export symbol ,...

Equivalent to [.global symbol ,... on page 40](#).

.extended

For x86. Reserves a long double or extended floating point type (80 bits) for each expression in the operand field.

.extern symbol ,...

Declare that each symbol in the symbol list is defined in a separate module. The linker supplies the value from the defining module during linking. Multiple **.extern** directives for the same symbol are permitted. Example:

```
.extern add,sub,mul,div
```

If the `-Xlabel-colon` option is set, this directive may begin in the first column.

.file "file"

Specifies the name of the source file for inclusion in the symbol table of the object file. The default is the name of the file. This directive is used by compilers to pass the name of the original source file to the symbol table. Example:

```
.file "test.c"
```

If the `-Xlabel-colon` option is set, this directive may begin in the first column.

.fill count,[size[,value]]

Reserves a block of data that is `count*size` bytes big and initialized to `count` copies of `value`. The size must be a value between 1 and 4. The default size is 1 and the default `value` is 0.

.float float-constant ,...

Reserves space and initializes single 32-bit IEEE floating point values. Example:

```
.float 3.14159265, .089e4
```

.global symbol ,...

Declares each symbol in the symbol list to be visible outside the current module. This makes each symbol available to the linker for use in resolving **.extern** references to the symbol. Example:

```
.global add,sub,mul,div
```

If the `-Xlabel-colon` option is set, this directive may begin in the first column.

.globl symbol ,...

Equivalent to [.global symbol ,... on page 40](#).

.half

Equivalent to [.short expression ,... on page 48](#).

.ident "string"

Appends the character string to a special section called **.comment** in the object file. See the *String Constants* section in [Constants on page 15](#) for rules for writing the "string". Example:

```
.ident "version 1.1"
```

If the `-Xlabel-colon` option is set, this directive may begin in the first column.

.if expression

The **.if** construct provides for conditional assembly. The *expression* must be absolute. If the *expression* evaluates to non-zero, all subsequent statements until the next **.elseif**, **.else**, or **.endif** directive at the same nesting level are assembled. If the terminating statement was **.elseif** or **.else**, then all statements following it up to the next **.endif** at the same level are skipped.

If the *expression* is zero, all statements up to the next **.elseif**, **.else**, or **.endif** at the same nesting level are skipped. An **.elseif** directive is evaluated and statements following it are skipped or not in the same manner as for the initial **.if** directive. If an **.else** directive is encountered, the statements following it up to the matching **.endif** are assembled.

.if constructs may be nested. Example:

```
.if      long_file_names maxname: .equ    1024
        .elseif medium_file_names maxname: .equ    128 .else maxname: .equ    14
        .endif
```

The following directives are equivalent: **.else** and **.elsec**, and **.endif** and **.endc**.

.ifendian**.ifendian big**

Assemble the following block of code if the mode is big-endian.

.ifendian little

Assemble the following block of code if the mode is -little-endian.

Note that the "endian" mode is set automatically from the target options and may not be directly changed by the user.

.ifeq expression

.ifeq is an alias for **.if expression == 0**. See [.if expression on page 40](#) for more details.

.ifc "string1", "string2"

.ifc is effectively an alias for **.if "string1" = "string2"** (**.if** does not allow string expressions). See [.if expression on page 40](#) for more details. See the *String Constants* section in [Constants on page 15](#) for rules for writing each "string".

For compatibility with other assemblers, either string may be enclosed in single quotes rather than double quotes. Within such a single-quoted string, two single quotes will be replaced by one single quote.

.ifdef symbol

Assemble the following code if the *symbol* is defined. See also [.ifndef symbol on page 42](#). See [.if expression on page 40](#) for more details on **.if** constructs.

.ifge expression

The **.ifge** is an alias for **.if expression >= 0**. See [.if expression on page 40](#) for more details.

.ifgt expression

The **.ifgt** is an alias for **.if expression > 0**. See [.if expression on page 40](#) for more details.

.ifle expression

The **.ifle** is an alias for **.if expression <= 0**. See [.if expression on page 40](#) for more details.

.iflt expression

The **.iflt** is an alias for **.if expression < 0**. See [.if expression on page 40](#) for more details.

.ifnc "string1", "string2"

.ifnc is effectively an alias for **.if "string1" != "string2"** (**.if** does not allow string expressions). See [.if expression on page 40](#) for more details. See the *String Constants* section in [Constants on page 15](#) for rules for writing each "string".

For compatibility with other assemblers, either string may be enclosed in single quotes rather than double quotes. Within such a single-quoted string, two single quotes will be replaced by one single quote.

.ifndef symbol

Assemble the following code if the *symbol* is not defined. See [.ifdef symbol on page 41](#). See also [.if expression on page 40](#) for more details on **.if** constructs.

.ifne expression

.ifne is an alias for **.if expression != 0**. See [.if expression on page 40](#) for more details.

.import symbol ,...

Equivalent to [.extern symbol ,... on page 39](#).

.incbin "file"[,offset[,size]]

Insert the content of a specified file into the assembly output. The assembler searches for the file in the current directory and all paths added using the **-I** option. If *offset* is specified, *offset* bytes are skipped at the beginning of the file. If *size* is specified, only *size* bytes are inserted into the assembly output.

.include "file"

Inserts the contents of the named file after the **.include** directive. May be nested to any level. Example:

```
.include "globals.h"
```

.lcnt expression

Set or change the number of lines on each page of the listing file. The default value is 60. This count may be set initially with the **-Xplen** option (see the Wind River Diab Compiler Options Reference), and it includes any margin set by option **-Xpage-skip** (the Wind River Diab Compiler Options Reference). See chapter 7: *Assembly Listing* for an example of an assembly listing. Example:

```
.lcnt 72
```

If the **-Xlabel-colon** option is set, this directive may begin in the first column.

.lcomm symbol, size [,alignment]

Define a symbol as the address of a local common block of length *size* expression bytes in the **.bss** section. If the **-Xlabel-colon** option is set, this directive may begin in the first column.

Note that the symbol is not made visible outside the current module. Contrast with **.comm**.

The *size* and *alignment* expressions must be absolute. See the description of **.comm** for a description of the *alignment* parameter and its default value. Example:

```
.lcomm local_array,200 # 200 bytes aligned on 8 bytes by
      default
```

.list

Turns on listing of lines following the **.list** directive if the option -L or -l is specified. Listing can be turned off with the **.nolist** directive. See [Example Assembly Listing for File Swap.lst on page 72](#) for an example of an assembly listing. If the -Xlabel-colon option is set, this directive may begin in the first column.

.literals

For ARM and MIPS (MIPS16 only).

Flushes the assembler's accumulated literal pool for the current section. This is used to emit the literal table at the exact point desired.

Whenever the distance to the most remote instruction referencing a literal gets close to the maximum, the assembler automatically emits part of table. It tries to generate the table after a branch if possible; otherwise it will insert a branch around the table.

.llen expression

Set the number of printable character positions per line of the listing file. The default value is 132. A value of 0 means unlimited line length. This count may be set initially by the -Xllen option (see the Wind River Diab Compiler Options Reference). See chapter 7: *Assembly Listing* for an example of an assembly listing. Example:

```
.llen 132
```

If the -Xlabel-colon option is set, this directive may begin in the first column.

.llong expression ,...

Reserves 8 bytes (64 bits) for each expression in the operand field and initializes the value of the word to the corresponding expression. Example:

```
.llong 0xfedcba9876543210,0123456,-75 # 24 bytes
```

The pound sign (#) is used here for the comment delimiter (substitute the appropriate symbol for your architecture if it differs).

.localentry symbol ,...

Define the offset in byte from symbol's global entry point to local entry point.

This directive is valid for PPC64 only. (See also [.global symbol ,... on page 40](#).)

.long expression ,...

Reserves one long word (32 bits) for each expression in the operand field and initializes the value of the word to the corresponding expression. Example:

```
.long 0xfedcba98,0123456,-75 # 12 bytes
```

The pound sign (#) is used here for the comment delimiter (substitute the appropriate symbol for your architecture if it differs).

name .macro [parameter ,...]

Start definition of macro *name*. All lines following the **.macro** directive until the corresponding **.endm** directive are part of the macro body. See the *Assembler Macros* chapter for a detailed description.

For ColdFire, MIPS, and PowerPC, note that the form:

```
.macro name parameter ,...
```

is also permitted for compatibility with other tools but is not recommended.

.mexit

Equivalent to [.exitm on page 39](#).

.name "file"

Equivalent to [.file "file" on page 39](#).

.nolist

Turns off listing of lines following the **.nolist** directive if the option -L or -l is specified. Listing can be turned on with the **.list** directive. See chapter 7: *Assembly Listing* for an example of an assembly listing.

.org expression

Sets the current location counter to the value of expression. The value must either be an absolute value or be relocatable and greater than or equal to the current location. Using the **.org** directive with an absolute value in ELF mode will produce a section named **.abs.xxxxxxxx**, where xxxxxxxx is the hexadecimal address of the section (with leading zeros as required to fill to eight digits). The linker will then place this section at the specified address. Example:

```
.org    0xff0000
```

will produce a section named **.abs.00ff0000** located at that address.

.p2align expression

Aligns the current location counter to 2 to the power of *expression*. The **.p2align** directive is equivalent to **.align** when the -Xalign-power2 option is enabled.

.page

Equivalent to [.eject on page 38](#).

.pagelen expression

Equivalent to [.lcnt expression on page 42](#).

.pcpdata

Equivalent to [.data on page 36](#).

.pcptext

Equivalent to [.text on page 50](#).

.plen expression

Equivalent to [.lcnt expression on page 42](#).

.previous

Assembly output is directed to the program section selected prior to the last **.section**, **.text**, **.data**, etc. directive.

.psect

Equivalent to [.text on page 50](#).

.psize page-length [,line-length]

Set the number of lines per page and number of character positions per line of the listing file. This directive is exactly equivalent to setting *page-length* with the [.lcnt expression on page 42](#) and setting *line-length* with the [.llen expression on page 43](#); see them for additional details. See chapter 7: *Assembly Listing* for an example of an assembly listing. Example:

```
.psize 72,132
```

If the `-Xlabel-colon` option is set, this directive may begin in the first column.

.rdata

Switches output to the **.rodata** (read-only data) section. Equivalent to **.rodata** directive.

.rodata

Switches output to the **.rodata** (read-only data) section. Equivalent to **.rdata** directive.

.sbss [symbol, size [,alignment]]

With no arguments, switch output to the **.sbss** section (short uninitialized data space). If the `-Xlabel-colon` option is set, this directive may begin in the first column.

With arguments, define a symbol as the address of a block of length *size* expression bytes in the **.sbss** section and make it global.

The *size* and *alignment* expressions must be absolute. See the description of **.comm** for a description of the *alignment* parameter and its default value. Examples:

```
.sbss           # switch to .sbss section
               .sbss local_array,200 # reserve space in .sbss section
```

The pound sign (#) is used here for the comment delimiter (substitute the appropriate symbol for your architecture if it differs).

.sbttl "string"

Equivalent to [.subtitle "string" on page 49](#).

.sdata

Switches output to the **.sdata** (short data space) section.

.sdata2

Switches output to the **.sdata2** (constant short data space) section.

.section name, [alignment], [type]

The assembly output is directed into the program section with the given name. The section name may be quoted with the (") character or not quoted. The section is created if it does not exist, with the attributes specified by *type*. *type* is one or more of the following characters, written as either as a quoted "string" or without quotes. If *type* is not specified, the default is **d** (data).

Type Character	Linker Command File Section Type	Description of Section Contents
b	BSS	zero-initialized data
c	TEXT	executable code
d	DATA	data
m	TEXT DATA	mixed code and data
n	COMMENT	not allocatable — the section is not to occupy space in target memory; for example, debugging information sections such as .debug in ELF
o	not applicable	COMDAT section (see <i>COMDAT Sections</i> in the <i>Wind River Diab Compiler Linker User's Guide</i> .)
r	CONST	readable data
w	DATA	writable data
x	TEXT	executable code

- See *Type Specification: ([=]BSS), ([=]COMMENT), ([=]CONST), ([=]DATA), ([=]TEXT), ([=]BTEXT); OVERLAY, NOLOAD, OPTIONAL* in the *Wind River Diab Compiler Linker User's Guide*.

- '**o**', for **COMDAT**, is an additional attribute of a section and is usually used with another type specification character. If "**o**" is used with another section type character, the linker command file section type will be that of the other section type character; if used by itself, the default will be **COMMENT**.

The *alignment* expression must evaluate to an integer and specifies the minimum alignment that must be used for the section.

Note that the COFF object module format is unable to handle the alignment information. Instead, the linker command language can be used to align the section.

The compiler uses the **b** type with the **#pragma section** directive to specify an uninitialized section. Example: direct assembly output to a section named **".rom"**, with four-byte alignment, containing read-only data and executable code:

```
.section ".rom",4,rx
```

.section n

The assembly output is directed into the program section named **"_Sn"**. Example: direct assembly output to a section named **"_S1"**:

```
.section 1
```

.sectionlink section-name


This directive will cause the current section to be linked as if it had the name *section-name*. This directive is available only for ELF object output. If the **-Xlabel-colon** option is set, this directive may begin in the first column.

.set option

The following **.set option** directives are available:

reorder noreorder

When processed by the low level optimizer (**lloptor** on some targets **reorder**) before assembly, enable/disable low level optimizations (thus, the **.set reorder** and **.set noreorder** directives are actually "low level optimizer" directives rather than assembler directives). Optimizations include scheduling (reordering) code, and simplifying or removing redundant instructions. Code generated for modules compiled with optimization includes a **.set reorder** directive. Use **.set noreorder** in **asm** strings and **asm** macros in such code to disable reordering changes to these hand-coded assembly inserts. Follow with **.set reorder** to re-enable reordering optimization. See *Reordering in asm Code*.

 Note: **.set reorder** is not automatically implied by the end of a function. Thus if **.set noreorder** is not explicitly followed by a matching **.set reorder**, other code, including compiler generated code may be unintentionally covered by the **.set noreorder** directive. This can prevent the low-level optimizer from making changes to compiler generated function prolog/epilog code that are actually required for correctness.

For **MIPS**, **SH**, and **SPARC**, when processed by the assembler, **.set reorder** instructs the assembler to add a NOP instruction after a branch to fill its delay slot. **.set noreorder** disables NOP insertion. The default is **.set noreorder** (the code is left unchanged).

mips32 mips16

For MIPS. Change to mips32 mode. Currently only relocations for function calls are supported. Only supported with **-tMIPS16**.

Switch back to mips16 mode after **.set mips32**.

macro nomacro

For MIPS. Allow/disallow special MIPS macro instructions. The default is **.set macro**.

at noat

For MIPS. Allow/disallow use of **\$at** by MIPS macro instructions. The default is **.set at**.


If the -Xlabel-colon option is set, this directive may begin in the first column.

.set symbol, expression

Defines *symbol* to be equal to the value of *expression*. This is an alternative to the **.equ** directive. Example:

```
.set    nine,9
```

If the expression contains a single symbol, the newly-defined symbol will have the same size and type as the symbol in the expression. This can be used to create an alias of a function or data symbol.

 **Note:** Using this form of **.set**, the symbol may not be redefined later. Use the next form of **.set** with the symbol first on the line if redefinition is required

If the -Xlabel-colon option is set, this directive may begin in the first column.

symbol[:].set expression

Defines *symbol* to be equal to the value of *expression*. This form of the **.set** is different from **symbol[:].equ expression** in that it is possible to redefine the value of *symbol* later in the same module. See the -Xlabel-colon-... entry in the *Wind River Diab Compiler Options Reference* regarding the initial colon.

expression may not refer to an external or undefined symbol. Example:

```
number: .set    9 ...
        number: .set    number+1
```

.short expression ,...

Reserves one 16 bit word for each expression in the operand field and initializes the value of the word to the corresponding expression. Example:

```
.short 0xba98, 012345, -75, 17 # reserves 8 bytes.
```

The pound sign (#) is used here for the comment delimiter (substitute the appropriate symbol for your architecture if it differs).

.size symbol, expression

Sets the size information for *symbol* to *expression*. Note that only the ELF object file format uses the size information.

.skip size

The **.skip** directive reserves a block of data initialized to zero. size is an expression giving the length of the block in bytes. Example:

```
name:    .skip    8
```

is the same as:

```
name:    .byte    0,0,0,0,0,0,0,0
```

.space expression

Equivalent to [.skip size on page 49](#).

.string "string"

Equivalent to [.ascii "string" on page 35](#).

.strz "string"

Equivalent to [.asciz "string" on page 35](#).

.struc**.struc**

statements

.ends

The **.struc** construct provides a way of defining structures within an assembly program. All labels defined in the structure are assigned values reflecting their offset from the start. Note that no space is reserved for the structure.

```
.struc op:    .skip 2 # op is set to 0 flags:  .skip 1 #
              flags is set to 2 .skip 1 # pad next:  .skip 4 # next is set to 4 prev:  .skip
4 #          prev is set to 8 value:  .skip 4 # value is set to 12 ssize:  .ends    # size is se
t           to 16 .data head:  .skip size # reserve 16 bytes for head .text
              lda    r25,r0,head    # let r25 point to head ld    r13,r25,value    # load val
ue          at r25+12 .text addis   r25,r0,head@ha # let r25 point to head
              addi    r25,r25,head@l lwz     r13,r25,value    # load value at r25+12 .text
              move.l  head+value,d1
```

.subtitle "string"

Sets the subtitle to the character string. This string replaces the **%nS** format specification in the format the string defined by the **-Xheader-format** option (see the *Wind River Diab Compiler Options Reference*). The subtitle may be set any number of times. The default subtitle is blank. See the *String Constants* section in [Constants on page 15](#) for rules for writing the "string".

```
.subtitle "string search function"
```

If the -Xlabel-colon option is set, this directive may begin in the first column.

.text

Switches output to the **.text** (instruction space) section.

.text_vle

For PowerPC VLE only. Switches output to the **.text_vle** (instruction space) section.

.title "string"

Sets the title to character string. The title may be set any number of times. The default title is blank. See the *String Constants* section in [Constants on page 15](#) for rules for writing the "string". Example:

```
.title "program.s"
```

If the -Xlabel-colon option is set, this directive may begin in the first column.

.ttl "string"

Equivalent to [.title "string" on page 50](#).

.type symbol, type

Marks *symbol* as *type*. The *type* can be one of the following:

#object @object object

symbol names an object

#function @function function

symbol names a function

@tfunc

For ARM. *symbol* names a Thumb function

Note that only the ELF object file format uses type information.

.uhalf

This is a synonym for **.short** ([.short expression ,... on page 48](#)) except that there are no alignment restrictions and an unaligned relocation type will be generated if required by the target.

.ulong

This is a synonym for **.long** ([.long expression ,... on page 43](#)) except that there are no alignment restrictions and an unaligned relocation type will be generated if required by the target.

.ushort

This is a synonym for **.short** ([.short expression ,... on page 48](#)) except that there are no alignment restrictions and an unaligned relocation type will be generated if required by the target.

.uword

For ColdFire and x86, see [.ushort on page 51](#).

For ARM, PowerPC, RH850, and TriCore, see [.ulong on page 50](#).

warning "string"

Generate a warning message showing the given string. See the *String Constants* section in [Constants on page 15](#) for rules for writing the "string". If the -Xlabel-colon option is set, this directive may begin in the first column.

.weak symbol ,...

Declares each *symbol* as a weak external symbol that is visible outside the current file. Global references are resolved by the linker. Note that only the ELF object file format supports weak external symbols. Example:

```
.weak add,sub,mul,div
```

If the -Xlabel-colon option is set, this directive may begin in the first column.

.width expression

Equivalent to [.llen expression on page 43](#).

.word expression, ...

Reserves one word for each expression in the operand field and initializes the value of the word to the corresponding expression. For ColdFire and x86, the word is 16 bits. For ARM, PowerPC, RH850, and TriCore the word is 32 bits.

For example:

ColdFire and x86

```
.word 0xba98,012345,-75 # rees 6 bytes.
```

ARM, PowerPC, RH850, and TriCore

```
.word 0xfedcba98,0123456,-75 # reserves 12 bytes.
```

The pound sign (#) is used here for the comment delimiter (substitute the appropriate symbol for your architecture if it differs).

.xdef symbol ,...

Equivalent to [.global symbol ,... on page 40](#).

.xref symbol ,...

Equivalent to [.extern symbol ,... on page 39](#).

.xopt

Pass **-X** options to the assembler using the format:

```
.xopt  option  
      name [=value ]
```

Example:

```
.xopt align-value
```

has the same effect as using `-Xalign-value` on the command line. In case of a conflict, **.xopt** overrides the command-line option. Also, some `-X` options are only tested before the assembly starts; in that case, the **.xopt** directive will have no effect. This option is primarily for internal use; the command-line options are preferred.

6. ASSEMBLER MACROS

6.1. About Assembler Macros

Assembler macros enable the programmer to encapsulate a sequence of assembly code in a *macro definition*, and then inline that code with a simple parameterized *macro invocation*.

ARM

```
.macro movindirect reg1,reg2 // macro definition
ldr    r0,[reg1,#0]
str    r0,[reg2,#0]
.endm

movindirect  r8,r9           // macro invocation #1
movindirect  r10,r11        // macro invocation #2
```

Produces:

```
ldr    r0,[r11,#0]          // macro expansion #1
str    r0,[r10,#0]
ldr    r0,[r9,#0]           // macro expansion #2
str    r0,[r8,#0]
```

ColdFire

```
mov8: .macro reg1,reg2 ; macro definition
move.l (reg1),(reg2)
move.l 4(reg1),4(reg2)
.endm

mov8 a0,a1 ; macro invocation #1
mov8 a4,a5 ; macro invocation #2
```

Produces:

```
move.l (a0),(a1) ; macro expansion #1
move.l 4(a0),4(a1)
move.l (a4),(a5) ; macro expansion #2
move.l 4(a4),4(a5)
```

MIPS

```
ld32: .macro reg,ident      # macro definition
      lui    reg,%hi(ident)
      lw     reg,%lo(ident)(reg)
      .endm
```

```
ld32    $4,yvar      # macro invocation #1
ld32    $5,xvar      # macro invocation #2
```

Produces:

```
lui     $4,%hi(yvar)  # macro expansion #1
lw      $4,%lo(yvar) ($4)
lui     $5,%hi(xvar)  # macro expansion #2
lw      $5,%lo(xvar) ($5)
```

PowerPC

```
ld32:   .macro    reg,ident      # macro definition
        addis     reg,r0,ident@ha
        lwz       reg,ident@l(reg)
        .endm

ld32    r3,yvar        # macro invocation #1
ld32    r4,xvar        # macro invocation #2
```

Produces:

```
addis   r3,r0,yvar@ha   # macro expansion #1
lwz     r3,yvar@l(r3)
addis   r4,r0,xvar@ha   # macro expansion #2
lwz     r4,xvar@l(r4)
```

RH850

```
ld32: .macro    reg,ident      # macro definition
      movhi     ident@ha,r0,reg
      ld.w      ident@l[reg],reg
      .endm

ld32 r3,yvar        # macro invocation #1
ld32 r4,xvar        # macro invocation #2
```

Produces:

```
movhi   yvar@ha,r0,r3   # macro expansion #1
ld.w    yvar@l[r3],r3
movhi   xvar@ha,r0,r4   # macro expansion #2
ld.w    xvar@l[r4],r4
```

TriCore

```
ld32:   .macro    dreg,areg,ident # macro definition
        movh.a     areg,ident@ha
        ld.w       dreg,[areg]ident@l
        .endm
```

```
ld32      %d2,%a0,yvar    # macro invocation
```

Produces:

```
movh.a    %a0,yvar@ha
ld.w      %d2,[%a0]yvar@l
```

x86

```
move:     .macro    reg1,reg2    # macro definition
          movl      reg1,reg2
          .endm

          move      %eax,%edi     # macro invocation #1
          move      %edi,%ebx     # macro invocation #2
```

Produces:

```
movl      %eax,%edi    # macro expansion #1
movl      %edi,%ebx    # macro expansion #2
```

6.2. Macro Definition

6.2.1. Syntax for ARM, ColdFire, MIPS, PowerPC, RH850, and x86

A macro definition has the form:

<i>label</i> :	.macro	[<i>parameter ,...</i>]
	<i>macro body</i>	
	.endm	

where *label* is the name of the macro, without containing any period.

The syntax used for TriCore is valid but is not recommended.

Related information

[Syntax for Tricore on page 55](#)

[Optional Parameter Use on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Name on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Number on page 58](#)

[Special Optional Parameter \0 on page 60](#)

[Separating Parameter Names From Text on page 63](#)

[Generating Unique Labels on page 65](#)

[NARG Symbol on page 68](#)

6.2.2. Syntax for Tricore

A macro definition has the form:

	.macro <i>name</i>	[<i>parameter</i> ,...]
	<i>macro body</i>	
	.endm	

Related information

[Syntax for ARM, ColdFire, MIPS, PowerPC, RH850, and x86 on page 55](#)

[Optional Parameter Use on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Name on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Number on page 58](#)

[Special Optional Parameter \0 on page 60](#)

[Separating Parameter Names From Text on page 63](#)

[Generating Unique Labels on page 65](#)

[NARG Symbol on page 68](#)

6.2.3. Optional Parameter Use

Optional parameters can be referenced in the macro body.

You can do this using two different methods:

- by parameter name
- by parameter number

The following two examples show a macro that calculates the following:

```
par1 = par2 + par3
```

(where the parameters are assumed to be in registers).

Related information

[Syntax for ARM, ColdFire, MIPS, PowerPC, RH850, and x86 on page 55](#)

[Syntax for Tricore on page 55](#)

[Optional Parameter Examples: Referencing by Parameter Name on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Number on page 58](#)

[Special Optional Parameter \0 on page 60](#)

[Separating Parameter Names From Text on page 63](#)

[Generating Unique Labels on page 65](#)

[NARG Symbol on page 68](#)

6.2.4. Optional Parameter Examples: Referencing by Parameter Name

The following examples reference by parameter name.

ARM

```
add3:  .macro  par1,par2,par3    # definition
        add    par1,par2,par3
        .endm

add3    r9,r10,r11              # invocation
```

Produces:

```
add    r9,r10,r11
```

ColdFire

```
add3l: .macro.size par1,par2,par3 ; definition
        move.size par1,par3
        add.l par2,par3
        .endm

        add3l.l d4,d5,d6 ; invocation
```

Produces:

```
move.l d4,d6
add.l d5,d6
```

MIPS

```
add3:   .macro  par1,par2,par3  # definition

        add    par1,par2,par3
        .endm

        add3    $7,$8,$9        # invocation
```

Produces:

```
add    $7,$8,$9
```

PowerPC

```
add3:   .macro  par1,par2,par3  # definition
        add    par1,par2,par3
        .endm

        add3    r7,r8,r9        # invocation
```

Produces:

```
add    r7,r8,r9
```

RH850

```
add3:   .macro  par1,par2,par3  # definition
        mov    par2,par1
        add    par3,par1
        .endm

        add3    r10,r6,r7        # invocation
```

Produces: :

```
mov    r6,r10
add    r7,r10
```

TriCore

```
add3a:      .macro      par1,par2,par3  # definition
            add         par1,par2,par3
            .endm

            add3a       %d2,%d3,%d4      # invocation
```

Produces:

```
add      %d2,%d3,%d4
```

x86

```
add3:      .macro      par1,par2,par3  # definition
            movl        par2,par1
            addl        par3,par1
            .endm

            add3        %eax,%edi,%ebx  # invocation
```

Produces:

```
movl     %edi,%eax
addl     %ebx,%eax
```

Related information

[Syntax for ARM, ColdFire, MIPS, PowerPC, RH850, and x86 on page 55](#)

[Syntax for TriCore on page 55](#)

[Optional Parameter Use on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Number on page 58](#)

[Special Optional Parameter \0 on page 60](#)

[Separating Parameter Names From Text on page 63](#)

[Generating Unique Labels on page 65](#)

[NARG Symbol on page 68](#)

6.2.5. Optional Parameter Examples: Referencing by Parameter Number

Optional parameters can be referenced by number using `\n` syntax, where `\1`, `\2`, ... `\9`, `\A`, ... `\Z` are the first, second, etc., actual parameters passed to the macro. When the `\n` syntax is used, formal parameters are optional in the macro definition. If present, both the named and numbered form may be freely mixed in the same macro body.

ARM

```
add3:      .macro
            add         \1,\2,\3
            .endm
```

```
add3    r9,r10,r11    # invocation
```

Produces:

```
add     r9,r10,r11
```

ColdFire

```
add3l: .macro ; definition
        move.l \1,\3
        add.l \2,\3
        .endm

        add3l d4,d5,d6 ; invocation
```

Produces:

```
move.l d4,d6
add.l d5,d6
```

MIPS

```
add3:   .macro                # definition
        add    \1,\2,\3
        .endm

        add3   $7,$8,$9      # invocation
```

Produces:

```
add     $7,$8,$9
```

PowerPC

```
add3:   .macro                # definition
        add    \2,\1
        .endm

        add3   r7,r8          # invocation
```

Produces:

```
add     r8,r7
```

RH850

```
add3:   .macro                # definition
        add    \2,\1
```



```

        .endm

add3    r7,r8        # invocation

```

Produces:

```

        add    r8,r7

```

TriCore

```

add3b:  .macro      par1,par2,par3    # definition
        add         \1,\2,\3
        .endm

add3b    %d3,%d4,%d5    # invocation

```

Produces:

```

        add    %d3,%d4,%d5

```

x86

```

add3:  .macro      # definition

        movl    \2,\1
        addl    \3,\1
        .endm

add3    %eax,%edi,%ebx    #invocation

```

Produces:

```

        movl    %edi,%eax
        addl    %ebx,%eax

```

Related information

[Syntax for ARM, ColdFire, MIPS, PowerPC, RH850, and x86 on page 55](#)

[Syntax for Tricore on page 55](#)

[Optional Parameter Use on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Name on page 56](#)

[Special Optional Parameter \0 on page 60](#)

[Separating Parameter Names From Text on page 63](#)

[Generating Unique Labels on page 65](#)

[NARG Symbol on page 68](#)

6.2.6. Special Optional Parameter \0

The special parameter `\0` denotes the actual parameter attached to the macro name with a "." character in an invocation. Usually this is an instruction size.

ARM

```

move:  .macro   dregp,sregp      // definition
        ldr.\0    r1,[sregp,0]
        str.\0    r1,[dregp,0]
        .endm

        move.h    r10,r11        // invocation

```

Produces:

```

        ldr        r1,[r11,0]
        str        r1,[r10,0]

```

ColdFire

```

add3:  .macro src1,src2,dest ; definition
        move.\0 src1,dest
        add.l src2,dest
        .endm

        add3.b d0,d1,d2 ; invocation

```

Produces:

```

        move.b d0,d2
        add.l d1,d2

```

MIPS

```

move:  .macro   dregp,sregp      # definition
        l\0      $1,0(sregp)
        s\0      $1,0(dregp)
        .endm

        move.h    $10,$11        # invocation

```

Produces:

```

        lh        $1,0($11)
        sh        $1,0($10)

```

PowerPC

```

move:  .macro   dregp,sregp      # definition
        l\0z     r0,0(sregp)
        st\0     r0,0(dregp)
        .endm

        move.h    r10,r11        # invocation

```

Produces:

```

    lhz    r0,0(r11)
    sth    r0,0(r10)

```

RH850

```

move:  .macro  dregp,sregp    # definition
        ld.\0    0[sregp],r6
        st,\0    r6,0[dregp]
        .endm

        move.h   r10,r11      # invocation

```

Produces:

```

        ld.h     0[r11],r6
        st.h     r6,0[r10]

```

TriCore

```

move:  .macro          dregp,sregp    # definition
        ld.\0          %d0,[sregp]0
        st.\0          [dregp]0,%d0
        .endm

        move.w         %a1,%a2      # invocation

```

Produces:

```

        ld.w         %d0,[%a2]0
        st.w         [%a1]0,%d0

```

x86

```

move:  .macro  reg1,reg2    # definition
        mov\0    reg1,reg2
        .endm

        move.l   %eax,%ebx    # invocation
        move.w   %ax,%bx
        move.b   %al,%bl


```

Produces:

```

        movl    %eax,%ebx
        movw    %ax,%bx
        movb    %al,%bl

```

 **Note:** For x86, be sure to use a register name appropriate for the specified instruction size.

Related information

[Syntax for ARM, ColdFire, MIPS, PowerPC, RH850, and x86 on page 55](#)

[Syntax for Tricore on page 55](#)

[Optional Parameter Use on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Name on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Number on page 58](#)

[Separating Parameter Names From Text on page 63](#)

[Generating Unique Labels on page 65](#)

[NARG Symbol on page 68](#)

6.2.7. Separating Parameter Names From Text

In the macro body, the characters "&&" can optionally precede or follow a parameter name to concatenate it with other text. This is useful when a parameter is to be part of an identifier:

ARM

```
xadd:  .macro  par1,par2,hcnst      # definition
        add    par1,par2,0x&&hcnst
        .endm

xadd    r4,r5,ff00                # invocation
```

Produces:

```
add    r4,r5,0xff00
```

ColdFire

```
xadd: .macro hcnst,dst ; definition
        add.l 0x&&hcnst,dst
        .endm

xadd ff00,d0 ; invocation
```

Produces:

```
add.l 0xff00,d0
```

MIPS

```
xadd:  .macro  par1,par2,hcnst      # definition
        addi   par1,par2,0x&&hcnst
        .endm

xadd    $4,$5,ff00                # invocation
```

Produces:

```
addi    $4,$5,0xff00
```

PowerPC

```
xadd:    .macro    par1,par2,hcnst        # definition
          addi      par1,par2,0x&&hcnst
          .endm

          xadd      r4,r5,ff00            # invocation
```

Produces:

```
addi      r4,r5,0xff00
```

RH850

```
xadd:    .macro    par1,par2,hcnst        # definition
          addi      0x&&hcnst,par1,par2
          .endm

          xadd      r4,r5,ff00            # invocation
```

Produces:

```
addi      0xff00,r4,r5
```

TriCore

```
xadd:    .macro    par1,par2,hcnst        # invocation
          add        par1,par2,0x&&hcnst
          .endm

          xadd      %d2,%d3,ff            # invocation
```

Produces:

```
add        %d2,%d3,0xff
```

x86

```
xmov:    .macro    hcnst,reg              # definition
          mov        0x&&hcnst,reg
          .endm

          xmov      f,%eax                # invocation
```

Produces:

```
mov        0xf,%eax
```

Related information

[Syntax for ARM, ColdFire, MIPS, PowerPC, RH850, and x86 on page 55](#)

[Syntax for TriCore on page 55](#)

[Optional Parameter Use on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Name on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Number on page 58](#)

[Special Optional Parameter \0 on page 60](#)

[Generating Unique Labels on page 65](#)

[NARG Symbol on page 68](#)

6.2.8. Generating Unique Labels

The special parameter \@ is replaced with a unique string to make it possible to create labels that are different for each macro invocation. The following macro defines a string of up to four bytes in the **.data** (or **.ldata** for TriCore) section at a uniquely generated label (however the length of the string is not checked), and then generates code to load the contents at that label (the string itself) into a register.

ARM

```
lstr:  .macro  string,reg      ; definition
      .data
.Lm\@:
      .byte  string,0
      .previous
      ldr    reg,=.Lm\@
      .endm

      lstr  "abc",d0          ; invocation
```

Produces:

```
.Lm.0001:  .data
           .byte  "abc",0
           .previous
           ldr    r0,=.Lm.0001
```

ColdFire

```
lstr:  .macro string,reg ; definition
      .data
.Lm\@:
      .byte string,0
      .previous
      move.l #.Lm\@,reg
      .endm

      lstr "abc",d0 ; invocation
```

Produces:

```
.Lm.0001:  .data
           .byte  "abc",0
```

```
.previous
move.l #.Lm.0001,d0
```

MIPS

```
lstr:  .macro  reg,string      # definition
        .data
.Lm\@:
        .byte  string,0
        .previous
        lui    reg,%hi(.Lm\@)
        lw     reg,%lo(.Lm\@)(reg)
        .endm

lstr    $2,"abc"              # invocation
```

Produces:

```
.Lm.0001:      .data
                .byte  "abc",0
                .previous
                lui    $2,%hi(r0,.Lm.0001)
                lw     $2,%lo(.Lm.0001)($2)
```

PowerPC

```
lstr:  .macro  reg,string      # definition
        .data
.Lm\@:
        .byte  string,0
        .previous
        addis   reg,r0,.Lm\@@ha
        lwz     reg,.Lm\@@l(reg)
        .endm

lstr    r3,"abc"              # invocation
```

Produces:

```
.Lm.0001:      .data
                .byte  "abc",0
                .previous
                addis   r3,r0,.Lm.0001@ha
                lwz     r3,.Lm.0001@l(r3)
```

RH850

```
lstr:  .macro  reg,string
        .data
```

```

.Lm\@:
    .byte    string,0
    .previous
    movhi    .Lm\@@ha,r0,reg
    ld.w     .Lm\@@l[reg],reg
    .endm
    lstr     r10,"ABC"

```

Produces:

```

    .data
.Lm.0000:
    .byte    "ABC",0
    .previous
    movhi    .Lm.0000@ha,r0,r10
    ld.w     .Lm.0000@l[r10],r10

```

TriCore

```

lstr:      .macro      reg,string      # definition
           .section   .ldata,,r
.Lm\@:
           .byte      string,0
           .previous
           movh.a      %a14,.Lm\@@ha
           ld.w         reg,[%a14].Lm\@@l
           .endm

           lstr        %d4,"abc"      # invocation

```

Produces:

```

           .section   .ldata,,r
.Lm.0005:
           .byte      "abc",0
           .previous
           movh.a      %a14,.Lm.0005@ha
           ld.w         %d4,[%a14].Lm.0005@l

```

x86

```

lstr:      .macro      reg,string      #definition
           .data
.Lm\@:
           .byte      string,0
           .previous
           movl        .Lm\@,reg
           .endm

           lstr        %eax,"abc"      # invocation

```

Produces:


```

        .data
.Lm.0001:
        .byte    "abc",0
        .previous
        movl     .Lm.0001,%eax

```

Related information

[Syntax for ARM, ColdFire, MIPS, PowerPC, RH850, and x86 on page 55](#)

[Syntax for Tricore on page 55](#)

[Optional Parameter Use on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Name on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Number on page 58](#)

[Special Optional Parameter \0 on page 60](#)

[Separating Parameter Names From Text on page 63](#)

[NARG Symbol on page 68](#)

6.2.9. NARG Symbol

The special symbol NARG represents the actual number of non-blank parameters passed to the macro (not including any \0 parameter).

The pound sign (#) is used here for the comment delimiter (substitute the appropriate symbol for your architecture if it differs).

```

init:   .macro   value           # definition
        .if      NARG == 0
        .byte    0
        .else
        .byte    value
        .endc
        .endm

init           # invocation #1
init    10     # invocation #2

```

Produces:

```

        .byte    0           # expansion #1
        .byte    10         # expansion #2

```

Related information

[Syntax for ARM, ColdFire, MIPS, PowerPC, RH850, and x86 on page 55](#)

[Syntax for Tricore on page 55](#)

[Optional Parameter Use on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Name on page 56](#)

[Optional Parameter Examples: Referencing by Parameter Number on page 58](#)

[Special Optional Parameter \0 on page 60](#)

[Separating Parameter Names From Text on page 63](#)

[Generating Unique Labels on page 65](#)

6.3. Invoking a Macro

A macro is invoked by using the macro name anywhere an instruction can be used.

The macro body will be inserted at the place of invocation, and the formal parameters in the macro definition will be replaced with the actual parameters, or operands, given after the macro name.

Actual parameters are separated by commas. To pass an actual parameter that includes special characters, such as blanks, commas and comment symbols, angle brackets "< >" may be used. Everything in between the brackets is regarded as one parameter.

If the option -Xmacro-arg-space-on is given, blanks may be included in an actual parameter without using brackets. For example:

```
init:  .macro  command,list
        .data
        command list
        .previous
        .endm

init    byte,<0,1,2,3>
```

Produces:

```
.data
.byte  0,1,2,3
.previous
```

6.4. Macros to Define Structures

Although **struct** is not part of the assembly language, the macros shown below allow you to assign offsets to symbols so they can refer to structure members.

These macros do not allocate memory; they merely assign values to symbols. The value of a structure "member" is its offset from the beginning of the structure.

The macros use **CURRENT_OFFSET_VALUE** to set the offsets of structure members: the **STRUCT** macro sets **CURRENT_OFFSET_VALUE** to 0; the **MEMBER** macro defines a symbol named for the member and having as its value **CURRENT_OFFSET_VALUE**, then increments **CURRENT_OFFSET_VALUE** by the size of the member.

```
STRUCT                                .macro
CURRENT_OFFSET_VALUE                 .set      0
                                     .endm

MEMBER                                .macro name, size
name = CURRENT_OFFSET_VALUE
CURRENT_OFFSET_VALUE                 .set CURRENT_OFFSET_VALUE + size
                                     .endm
```

CURRENT_OFFSET_VALUE must be incremented with this form of the **.set** directive because it allows the symbol so set to be set again later in the module. See

See [symbol\[:\].set expression](#) in [Assembler Directives on page 32](#) for details.

Also, note that:

- The **MEMBER** macro cannot be labeled.
- These macros cannot be used to define nested structures because there is only one **CURRENT_OFFSET_VALUE** used for all instances.

- A final **MEMBER** can be used to define the size of the structure.

Example

The macros define the symbols **first_name**, **middle_initial**, and **last_name** with values 0, 20, and 21 respectively, and define **name_size** as the total size of the "structure" with a value of 46.

```
STRUCT
MEMBER    first_name,20
MEMBER    middle_initial,1
MEMBER    last_name,25
MEMBER    name_size,0
```

One might use this, for example, as follows:

```
.data
rec1:
    .skip    20      # reserve space for a first name
    .skip    1       # ... middle initial
    .skip    25      # ... and last name
```

Then an expression such as **rec1+last_name** in an instruction would access the **last_name** "member" of the **rec1** "structure".

The pound sign (#) is used here for the comment delimiter (substitute the appropriate symbol for your architecture if it differs).

7. ASSEMBLY LISTING

7.1. Assembler Listing Option

If the `-l` or `-L` option is specified, a listing is produced. The `-l` option produces a listing file with the default extension `.lst` (or the extension specified with `-Xlist-file-extension="string"`). The `-L` option sends the listing to standard output.

The listing contains the following:

Location

Hexadecimal value giving the relative address of the generated code within the current section.

PI

"PI" stands for "Program Location counter number". Maps one-to-one to the section number in the object file (but not necessarily in the same order). When the same section is used at several discontinuous places in the source, the same section number will be used for all instances.

Code

Generated code in hexadecimal.

Line

Source line number.

Source Statement

Source code lines.

To change the format of the assembly line, see the `-Xline-format` entry in the *Wind River Diab Compiler Options Reference*.

If the `-H` option is used, a header containing the source filename and the cumulative number of errors is displayed at the top of each page. To change the format of the header, see the `-Xline-format` entry in the *Wind River Diab Compiler Options Reference*.

Errors are not included in the listing but are always written to **stderr**.

The following shows a listing produced by assembling an extract from file **swap.s** with the following command.

For ARM:

```
dcc -tARMEN -c -Xpreprocess-assembly -Wa,-l -Wa,-H -l swap.s.
```

For ColdFire, MIPS, RH850, TriCore, and x86, the command is similar to the one below for PowerPC (substitute the target configuration specification for other architectures):

```
das -tPPC860EN -l -H swap.s.
```

The file **swap.s** is used with the bubble sort example in the *Wind River Diab Compiler Getting Started* manual.

For ARM, note the use of the **dcc** command and special options passed to the assembler using the `-Wa` options (rather than the usual **das** command). This is because the actual ARM **swap.s** assembly file contains code for both the ARM and Thumb processors, and the **dcc** preprocessor is used to select the desired variant. The listing contains "..." lines where the code not selected or comments were deleted for brevity. Also, note the first line: the actual file assembled, **/var/tmp/d**, was the temporary output file from the preprocessor. See file **swap.c** for details.

7.2. Example Assembly Listing for File Swap.lst

Refer to the following examples of assembly listing.

ARM

			File: /var/tmp/d Errors 0	
Location	Pl	Code	Line	Source Statement
...				
			24	.name "swap.s"
			25	.section .text2,,c
			26	.align 4
			27	.xdef swap
			28	
			29	swap:
...				
00000000	01	e24d d004	33	sub sp,sp,#4
00000004	01	e58d e000	34	str lr,[sp,#0]
...				
00000008	01	e590 1000	37	ldr r1,[r0,#0]
0000000c	01	e590 2004	38	ldr r2,[r0,#4]
00000010	01	e580 2000	39	str r2,[r0,#0]
00000014	01	e580 1004	40	str r1,[r0,#4]
...				
00000018	01	e59d e000	45	ldr lr,[sp,#0]
0000001c	01	e28d d004	46	add sp,sp,#4
00000020	01	e1a0 f00e	47	mov pc,lr

ColdFire

			File: swap.s Errors 0	
Location	Pl	Code	Line	Source Statement
...				
			1	.file "swap.s"
			2	.section .text2,,c
			3	
			4	.align 4
			5	.xdef _swap
			6	
			7	_swap:
00000000	01	4e56 0000	8	link a6,#-0
			9	
00000004	01	206e 0008	10	move.l 8(a6),a0
00000008	01	2010	11	move.l (a0),d0
0000000a	01	20a8 0004	12	move.l 4(a0),(a0)
0000000e	01	2140 0004	13	move.l d0,4(a0)
			14	
00000012	01	4e5e	15	unlk a6
00000014	01	4e75	16	rts

MIPS

			File: swap.s Errors 0	
Location	Pl	Code	Line	Source Statement

```

1      .file      "swap.s"
2      .section   .text2,,c
3      .align     4
4      .xdef      swap
5
6  swap:
00000000 01 8c820000    7      lw        $2,0($4)
00000004 01 8c980004    8      lw        $24,4($4)
00000008 01 ac980000    9      sw        $24,0($4)
0000000c 01 03e00008   10     jr        $31
00000010 01 ac820004   11     sw        $2,4($4)

```

PowerPC

			File: swap.s	Errors	0
Location	Pl	Code	Line	Source	Statement
			1	.file	"swap.s"
			2	.section	.text2,,c
			3	.align	4
			4	.xdef	swap
			5		
			6	swap:	
00000000	01	9421 fff8	7	stwu	r1,-8(r1)
00000004	01	7c08 02a6	8	mfspr	r0,8
00000008	01	9001 000c	9	stw	r0,12(r1)
			10		
0000000c	01	8083 0000	11	lwz	r4,0(r3)
00000010	01	80a3 0004	12	lwz	r5,4(r3)
00000014	01	90a3 0000	13	stw	r5,0(r3)
00000018	01	9083 0004	14	stw	r4,4(r3)
			15		
0000001c	01	8001 000c	16	lwz	r0,12(r1)
00000020	01	7c08 03a6	17	mtspr	8,r0
00000024	01	3821 0008	18	addi	r1,r1,8
00000028	01	4e80 0020	19	blr	

RH850

```

1
// Equivalent C code:
2 //
3 //
void swap (int array[])
4 //
{
5 //
    int temp = array[0];
6 //
    array[0] = array[1];
7 //
    array[1] = temp;
8 //
}
9

```

			10	.align	1
			11	.global	_swap
			12	_swap:	
00000000	00	263f 0100	13	ld.w	0[r6],r7
00000004	00	2647 0500	14	ld.w	4[r6],r8
00000008	00	6647 0100	15	st.w	r8,0[r6]
0000000c	00	663f 0500	16	st.w	r7,4[r6]
00000010	00	7f00	17	jmp	[r31]

TriCore

			File: swap.s	Errors	0
Location	Pl	Code	Line	Source	Statement
			1	# swap.s: assembly "swap" function	
				# for Bubble Sort example: TriCore	
				target.	
			2	#	
			3	# Equivalent C code:	
			4	#	
			5	# void swap (int array[])	
			6	# {	
			7	# int temp = array[0];	
			8	# array[0] = array[1];	
			9	# array[1] = temp;	
			10	# }	
			11		
			12	.name	"swap.s"
			13	.section	.text,,c
			14	.align	4
			15	.globl	swap
			16		
			17	swap:	
00000000	00	5444	18	ld.w	%d4,[%a4]0
00000002	00	1945 0400	19	ld.w	%d5,[%a4]4
00000006	00	7445	20	st.w	[%a4]0,%d5
00000008	00	5944 0400	21	st.w	[%a4]4,%d4
0000000c	00	0090	22	ret	
			23		

x86

			File: swap.s	Errors	0
Location	Pl	Code	Line	Source	Statement
			1	.name	"swap.s"
			2	.section	.text2,,c
			3	.align	4
			4	.xdef	swap
			5		
			6	swap:	
00000000	01	8b4c 2404	7	movl	4(%esp), %ecx
00000004	01	8b11	8	movl	(%ecx), %edx
00000006	01	8b41 04	9	movl	4(%ecx), %eax
00000009	01	8901	10	movl	%eax, (%ecx)

0000000b	01	8951 04	11	movl	%edx, 4(%ecx)
0000000e	01	c3	12	ret	

8. INSTRUCTION MNEMONICS AND OPERAND ADDRESSING MODES

8.1. Instruction Mnemonics

Support for instruction mnemonics is as follows:

Support for Instruction Mnemonics

ARM

The assembler supports all ARM and Thumb instructions as described in the *ARM Architecture Reference Manual*, including the simplified mnemonics described there.

ColdFire


The instruction mnemonics used by the assembler are described in the *MCF52xx Microprocessor User's Manual* with a few variations described below.

MIPS

The assembler supports all MIPS instructions as described in the *MIPS RISC Architectures* manuals.

PowerPC

The assembler supports all PowerPC instructions as described in the *PowerPC Microprocessor Family: The Programming Environments* manuals, including the simplified mnemonics described there.

 **Note:** For PowerPC, the **das** assembler understands both Book E instruction mnemonics and "**se_**" and "**e_**" forms. The compiler generates Book E mnemonics in preference to VLE-specific ones, unless it is absolutely necessary to do otherwise. For more on Book E and VLE instructions, see the documentation above or the documentation provided by your PowerPC processor's manufacturer.

RH850

The assembler supports all RH850 instructions as described in the current architecture guide.

TriCore

The assembler supports instructions and mnemonics described in the *TriCore Architecture Manual*.

x86

The assembler supports instructions and mnemonics described in the *Intel Architecture Software Developer's Manual*.

Special Instructions and Extensions

ARM Special instructions


The assembler also recognizes the following directives:

```
.code16
.code32
```

All code to follow these is either Thumb or ARM code respectively.

ColdFire Special Instructions

Some of the instructions can apply to byte, word, or long operands; thus the normal instruction mnemonic is suffixed with **b**, **w** or **l** to indicate which operand length that was intended. For example, there are three mnemonics for the **tst** instruction: **tst.b**, **tst.w** and **tst.l**.

 **Note:** The "." preceding the operand length field is optional. Thus, the **tst** instructions can be written as **tstb**, **tstw**, and **tstl**.

When using the **bcc**, **bra**, and **bsr** instructions without a size, the assembler will produce the shortest form it can. This may be an 8, 16, or, depending on processor type, a 32 bit address relative to the program counter (PC). If the processor type does not support the 32 bit PC-relative branch instruction an error will be generated.

With the option `-Xbra-is-jra` on (see the *Wind River Diab Compiler Options Reference*), the assembler will generate an absolute jump instead of generating an error. Absolute address is implemented for conditional branches by inverting the sense of the condition and branching around a 32 bit **jmp** or **jsr** instruction. As an alternative, the instructions **bcc**, **bra** and **bsr** can be replaced with **jcc**, **jra** (or **jbra**), and **jsr**, respectively to get this behavior on selected instructions, without specifying the `-Xbra-is-jra` option.

The following table summarizes the additional instruction conventions accepted by the assembler for ColdFire:

Description	Examples
move instruction without ending "e".	<pre>mov.b movl</pre>
address register instructions (e.g. adda) without ending "a".	<pre>add.l d0,a0 cmp.l d0,a0 move.l d0,a0 subl d0,a0</pre>
Immediate instructions (e.g. addi) without ending "i".	<pre>add.l #10,d0 and.l #0x80,d0 cmp.l #10,a0 eor.l #0xff,d0 orl #0xff,d0 subl d0,a0</pre>
No dot "." necessary to indicate instruction size.	<pre>moveb d1,d0 movw d2,d3 cmpl #10,d4</pre>
Branches without a size are optimized to use shortest possible size. The branch will still be PC-relative unless the <code>-Xbra-is-jra</code> option is used.	<pre>bra label bne loop bsr func</pre>
Byte size branches (bra.b) can use the ".s" (short) suffix.	<pre>bra.s label</pre>

Description	Examples
Optimized jump instructions selects the shortest possible PC-relative or absolute branch.	<pre> jra label jbra label jne loop jbsr func </pre>

MIPS Special Instructions

One extra macro instruction has been added:

```
jbal    label
```

This is a function call that is always position-independent. Depending on the distance to the *label*, this will be translated to either a **bal** instruction, or the following:

```

lui     $1,%pichi(func)
addu    $1,$1,$23
addi    $1,$1,%piclo(func)
jalr    $1

```

8.2. Operand Addressing Modes

8.2.1. Registers

Refer to the following lists of register names.

ARM

Registers can be specified in the following ways, in either lower or upper case:

Register	Use/Description
r0 - r15	General purpose registers; can only be used where a general purpose register is expected.
sp	Same as r13 .
lr	Same as r14 .
pc	Same as r15 .

ColdFire

Register names can be given in either lowercase or uppercase. Beyond the usual Motorola (Freescale) register names, the following register names are accepted:

Name	Description
sp	Same as a7
fp	Same as a6

MIPS

Registers can be specified in the following ways, in either lower or upper case:

Register Name	Software Name	Use/Description
\$0	\$zero zero	Always zero.
\$1	\$at at	Reserved for the assembler.
\$2 - \$3	\$v0 - \$v1 v0 - v1	Temporary registers; not preserved by functions. Hold variables whenever possible. Also used for result passing.
\$4 - \$7	\$a0 - \$a3 a0 - a3	Temporary registers; not preserved by functions. Hold variables whenever possible. Also used to pass the first 4 words of actual arguments.
\$8 - \$15	\$t0 - \$t7 t0 - t7	Temporary registers; not preserved by functions. Hold variables whenever possible.
\$16 - \$22	\$s0 - \$s6 s0 - s6	Preserved registers; saved when used by functions. Hold variables which cannot be put in the temporary registers.
\$23	\$s7 s7	Pointer to code section; used for position-independent code (PIC).
\$24 - \$25	\$t8 - \$t9 t8 - t9	Temporary registers; not preserved by functions. Hold variables whenever possible.
\$26 - \$27	\$k0 - \$k1 k0 - k1	Reserved by the operating system.
\$28	\$gp gp	Global Pointer; points to the small data section (SDA) and is also used for position-independent data (PID).
\$29	\$sp sp	Stack pointer; saved when used by function. See information on stack layout on 9.2 Stack Layout, p. 81, for details about this register.

Register Name	Software Name	Use/Description
\$30	\$s8 s8	Preserved register; saved when used by functions. Hold variables which cannot be put in the temporary registers. Used as a frame pointer for functions using the alloca() function.
\$31	\$ra ra	Return address; saved if used by function.
\$f0 - \$f2		Temporary registers; not preserved by functions. Hold variables whenever possible. Also used for result passing.
\$f4 - \$f10		Temporary registers; not preserved by functions. Hold variables whenever possible.
\$f12 - \$f14		Temporary registers; not preserved by functions. Hold variables whenever possible. Also used to pass the first two words of floating point arguments.
\$f16 - \$f18		Temporary registers; not preserved by functions. Hold variables whenever possible.
\$f20 - \$f30		Preserved floating point registers; saved when used by functions. Hold variables which cannot be put in \$f0- \$f18 .

PowerPC

Registers can be specified in the following ways, in either lower or upper case:

Register	Use/Description
r0 - r31 0 - 31	General purpose registers; can only be used where a general purpose register is expected.
f0 - f31 0 - 31	Floating point registers; can only be used where a floating point register is expected.
cr0 - cr7 0 - 7	Condition code registers; can only be used where a condition code register is expected.
sr0 - sr15 0 - 15	Segment registers; can only be used where a segment register is expected.
0 - 1023 spr0 - spr1023 xer (1)	Special purpose registers; can only be used where a special purpose register is expected. Only the most common

Register	Use/Description
ctr (9) lr (8)	register names are shown. The assembler recognizes all special purpose registers for the supported targets.

RH850

Registers can be specified in the following ways, in either lower or upper case:

- r0 - r31: General purpose registers. Can only be used where a general purpose register is expected. The following aliases are supported:
 - r0: zero
 - r3: sp
 - r4: gp
 - r5: tp
 - r30: ep
 - r31: lp
- vr0 - vr31: Vector registers. Can only be used where a vector register is expected.
- Special purpose registers by name: bpam, dbic, fpsr, ...

For information about register definitions, see the RH850 Compiler ABI Specification.

TriCore

Registers can be specified in the following ways, in either lower or upper case:

Register	Use/Description
%d0 - %d7	Scratch registers. Not preserved by functions. They hold variables whenever possible. %d4-%d7 are used for non-pointer arguments. %d2 is used for 32-bit non-pointer return values; %d2/%d3 are used for 64-bit return values.
%d8 - %d15	Preserved Registers. %d15 is used as an implicit data register for many 16-bit instructions.
%a0	Small data area base register.
%a1	Literal Data Pointer.
%a2-%a7	Scratch registers. Not preserved by functions. They hold variables whenever possible. %a2-%a7 are used for pointer arguments. %a2 is used to return pointer values.
%a8-%a9	Reserved base address registers.

Register	Use/Description
%a10	Stack Pointer (SP)
%a11	Return address register (RA)
%a12-%a15	Preserved Registers. %a15 is used as an implicit base address register for many 16-bit load/store instructions.

x86

Registers can be specified in the following ways, in either lower or upper case:

Register Name	Software Name	Description
eax	ax	Accumulator register.
edx	dx	Data register.
ebx	bx	Base register.
ecx	cx	Loop counter.
esi	si	Source index for copying data.
edi	di	Destination index for copying data.
ebp	bp	Frame pointer.
eip	ip	Program counter (instruction pointer).
esp	sp	Stack pointer.

Related information

[Expressions on page 82](#)

8.2.2. Expressions

The following tables shows examples of expressions for common addressing modes.

For information about relocation types, see the *Wind River Diab Compiler User's Guide* for your architecture.

ColdFire

Addressing Mode	Mnemonic Types: Embedded	Mnemonic Types: MIT
Data Register Direct	d0	d0
Address Register Direct	a0	a0
Address Register Indirect (indirect)	(a0)	a0@
Address Register Indirect with Postincrement (autoincrement)	(a0)+	a0@+
Address Register Indirect with Predecrement (autodecrement)	-(a0)	a0@-
Address Register Indirect with displacement (16 bit indexed) See Notes 1 and 2.	(12,a0) 12(a0) (var).w(a5) (var@sda).w(a5)	a0@12 a5@var:w
Address Register Indirect with index (double indexed with 8 bit offset). See Note 3.	(12,a0,d1) (a0,d1) 12(a0,d1) (12,a0,d1.w*4)	a0@(12,d1) a0(d1) a0(12,d1:w:4)
Absolute Short	(label).w	label:w
Absolute Long	label (label).l	label label:l
Immediate Data	#17 #var	#17 #var
Program Counter Indirect with displacement (16 bit indexed)	(12,pc) 12(pc) (var).w(pc)	pc@12 pc@var:w
Program Counter Indirect with index (double indexed with 8 bit offset)	(12,pc,d1) (pc,d1) 12(pc,d1) (12,pc,d1.w*4)	pc@(12,d1) pc(d1) pc(12,d1:w:4)
The following addressing modes are for 68K/CPU32 only.		
Memory Indirect Postindexed	([4,a0],d1*4,8)	a0@(4)@(8,d1:4)
Memory Indirect Preindexed	([4,a0,d1*4],8)	a0@(4,d1:4)@(8)
PC Memory Indirect Postindexed	([4,pc],d1*4,8)	pc@(4)@(8,d1:4)
PC Memory Indirect Preindexed	([4,pc,d1*4],8)	pc@(4,d1:4)@(8)

Notes:

1. **(var).w(a5)**: the size of the expression or the value of the constant used in an addressing mode will determine which format to use. The assembler will use the shortest format possible. If the value of the expression is unknown (i.e. refers to relocatable label), the assembler will use the longest format possible. The size of an expression can be forced by suffixing it with a **.w/l** (Embedded mnemonics) or a **:w/l** (MIT mnemonics). Note that since the period (".") can be used in a label identifier, parentheses must be used, that is, **(label).w**.
2. There are no limits on the complexity of an expression as long as all the operands are constants. When a label is used in the expression, the assembler will generate a relocation entry so that the linker can patch the instruction with the correct address.
3. The size and scale of the index register is optional. The default size is 1 (long) and the default scale is 1.

MIPS

Example	Description
<pre>lui \$2,%hi(var) lw \$2,%lo(var)(\$2)</pre>	Load \$2 with the value pointed to by the 32 bit address of <i>var</i> . %hi(var) will extract the higher 16 bits and adjust them so that when adding the sign extended lower 16 bits, <i>var@l</i> , the resulting value will be the address of <i>var</i> .
<pre>lui \$2,%hi(var+4) lw \$2,%lo(var+4)(\$2)</pre>	Same as above, but use the address of <i>var</i> plus 4.
<pre>lw \$2,%sdaoff(svar)(\$28)</pre>	Load \$2 with the value of <i>svar</i> located in the 64K Small Data Area (SDA). This area is typically located in sections .sdata and .sbss and is pointed to by register \$gp (\$28) .
<pre>lw \$2,svar(\$0)</pre>	Load \$2 with the value of <i>svar</i> located in the 64K data area around address 0.
<pre>lui \$4,%pidhi(var) addu \$4,\$4,\$28 lw \$4,%pidlo(var)(\$4)</pre>	Load \$4 with the value of <i>var</i> by adding a 32 bit offset to the SDA base register described above. This is a way of getting position-independent data for data areas bigger than 64K.
<pre>lui \$4,%pichi(var) addu \$4,\$4,\$23 lw \$4,%piclo(var)(\$4)</pre>	Load \$4 with the value of <i>var</i> by adding a 32 bit offset to the code base register \$23 . This is a way of getting position-independent code for the constant data area and function addresses. See 14.5 Position-Independent Code and Data (PIC and PID), p. 148, for a complete discussion.

PowerPC

Example	Description
<pre>addis r3,r0,var@ha lwz r3,r3,var@l</pre>	Load r3 with the value pointed to by the 32 bit address of <i>var</i> . <i>var@ha</i> will extract the higher 16 bits and adjust them so that

Example	Description
	when adding the sign extended lower 16 bits, var@l , the resulting value will be the address of var .
<pre>addis r3,r0,(var+4)@ha lwz r3,(var+4)@l(r3)</pre>	Same as above, but use the address of var plus 4.
<pre>lwz r3,svar@sdarx(r0)</pre>	<p>Load r3 with the value of svar located in one of the Small Data Areas (SDA). There are three 64KB SDAs:</p> <p>One area pointed to by register r13 for regular small data. Usually in the sections .sdata and .sbss.</p> <p>One area pointed to by register r2 for constant small data. Usually in the sections .sdata2.</p> <p>One area located around address 0 (register r0).</p> <p>The assembler will generate the appropriate relocation information and the linker will patch the instruction to use the correct register and the correct offset.</p>
<pre>addis r3,r0,var@sdarx@ha lwz r3,var@sdax@l(r3)</pre>	Load r3 with the value of var by adding a 32 bit offset to one of the base registers described above. This is a way of getting position-independent data for data areas bigger than 64K.

RH850

Example	Description
<pre>movhi var@ha,r0,r10</pre>	Load r10 with the value pointed to be the 32-bit address of var . var@ha extracts the upper 16-bits and adjusts them so that when adding the sign extended lower 16-bits, var@l , the resulting value is the address of var .
<pre>ld.w var@l[r10],r10</pre>	
<pre>movhi (var+4)@ha,r0,r10</pre>	Same as above, but uses the address of var plus 4.
<pre>ld.w (var+4)@l[r10],r10</pre>	
<pre>st.h r6,val32@data[r0]</pre>	Expression relative to GP register (r4).
<pre>st.h r6,val32@code[r0]</pre>	Expression relative to TP register (r5)

Related information

[Registers on page 78](#)