COPYRIGHT NOTICE
© 2013–2018 IAR Systems AB.

No part of this document may be reproduced without the prior written consent of IAR Systems AB. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

DISCLAIMER
The information in this document is subject to change without notice and does not represent a commitment on any part of IAR Systems. While the information contained herein is assumed to be accurate, IAR Systems assumes no responsibility for any errors or omissions.

In no event shall IAR Systems, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

TRADEMARKS
IAR Systems, IAR Embedded Workbench, Embedded Trust, IAR Connect, C-SPY, C-RUN, C-STAT, IAR Visual State, IAR KickStart Kit, I-jet, I-jet Trace, I-scope, IAR Academy, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Renesas is a registered trademark of Renesas Electronics Corporation. RH850 is a trademark of Renesas Electronics Corporation.

Adobe and Acrobat Reader are registered trademarks of Adobe Systems Incorporated.

All other product names are trademarks or registered trademarks of their respective owners.

EDITION NOTICE
Third edition: August 2018

Part number: ARH850-3

This guide applies to version 2.x of IAR Embedded Workbench® for the Renesas RH850 Family.

Internal reference: BB4, Mym8.1, asrct2010.3, V_110411, IJOA.
Contents

Tables .................................................................................................................................. 9
Preface ............................................................................................................................... 11
  Who should read this guide ................................................................. 11
  How to use this guide ....................................................................... 11
  What this guide contains ................................................................. 12
  Document conventions ..................................................................... 12
    Typographic conventions ............................................................ 12
    Naming conventions ..................................................................... 13
Introduction to the IAR Assembler for RH850 ........................................ 15
  Introduction to assembler programming ...................................... 15
    Getting started ............................................................................. 15
  Modular programming .................................................................... 16
  External interface details ............................................................... 17
    Assembler invocation syntax ...................................................... 17
    Passing options .......................................................................... 17
    Environment variables ............................................................... 18
    Error return codes ....................................................................... 18
Source format ......................................................................................... 19
Assembler instructions ........................................................................... 20
Expressions, operands, and operators .................................................... 20
  Integer constants .......................................................................... 20
  ASCII character constants .......................................................... 21
  Floating-point constants ............................................................. 21
  True and false .............................................................................. 22
  Symbols ......................................................................................... 22
  Labels .......................................................................................... 23
  Register symbols .......................................................................... 23
  Predefined symbols ....................................................................... 23
  Absolute and relocatable expressions .......................................... 26
  Expression restrictions ................................................................. 26
List file format ................................................................. 27
  Header ................................................................................. 27
  Body ..................................................................................... 27
  Summary .............................................................................. 28
  Symbol and cross-reference table .................................... 28

Programming hints .......................................................... 28
  Accessing special function registers .................................. 28
  Using C-style preprocessor directives ............................... 28

Tracking call frame usage ................................................... 29
  Call frame information overview ...................................... 29
  Call frame information in more detail ............................. 30
  Defining a names block .................................................... 31
  Defining a common block ................................................ 32
  Annotating your source code within a data block ............. 32
  Specifying rules for tracking resources and the stack depth .. 33
  Using CFI expressions for tracking complex cases .......... 35
  Stack usage analysis directives ........................................ 36
  Examples of using CFI directives ..................................... 36

Assembler options ................................................................ 39

Using command line assembler options ............................. 39
  Specifying command line options ................................... 39
  Specifying parameters .................................................... 40
  Extended command line file ............................................ 40

Summary of assembler options ........................................... 41

Description of assembler options ....................................... 42
  --case_insensitive ......................................................... 43
  --code_model ............................................................... 43
  --core ............................................................................. 44
  -D .................................................................................. 44
  --data_model ............................................................... 45
  --debug, -r ................................................................... 45
  --dependencies ............................................................ 46
  --diag_error ................................................................. 47
--diag_remark ................................................................. 47
--diag_suppress ............................................................... 48
--diag_warning ................................................................. 48
--diagnostics_tables ........................................................... 49
--dir_first ......................................................................... 49
--error_limit ....................................................................... 50
-f .......................................................................................... 50
--f .......................................................................................... 51
--fpu ...................................................................................... 51
--header_context ................................................................. 52
-l .............................................................................................. 52
-l .............................................................................................. 53
-M ............................................................................................ 54
--macro_positions_in_diagnostics ........................................ 54
--mnem_first ......................................................................... 55
--no_bom ............................................................................... 55
--no_path_in_file_macros ...................................................... 55
--no_system_include ............................................................ 56
--no_warnings ....................................................................... 56
--no_wrap_diagnostics .......................................................... 56
--only_stdout ......................................................................... 56
--output, -o ........................................................................... 57
--predef_macros ..................................................................... 57
--preinclude ......................................................................... 58
--preprocess .......................................................................... 58
--remarks ............................................................................... 59
--silent .................................................................................... 59
--source_encoding ................................................................. 59
--system_include_dir ............................................................ 60
--text_out .............................................................................. 60
--use_unix_directory_separators ........................................... 61
--utf8_text_in ....................................................................... 61
--warnings_affect_exit_code ............................................... 61
--warnings_are_errors ........................................................ 62
Assembler operators .......................................................... 63

Precedence of assembler operators 63

Summary of assembler operators ........................................ 63
  Parenthesis operator .......................................................... 63
  Function operators ............................................................. 64
  Unary operators ................................................................. 64
  Multiplicative arithmetic operators ...................................... 65
  Additive arithmetic operators ............................................. 65
  Shift operators ................................................................. 65
  Comparison operators ....................................................... 65
  Equivalence operators ....................................................... 65
  Logical operators ............................................................. 66
  Conditional operator ......................................................... 66

Description of assembler operators .................................... 66
  () Parenthesis ................................................................. 66
  * Multiplication ............................................................... 67
  + Unary plus ................................................................. 67
  + Addition ................................................................. 67
  – Unary minus .............................................................. 67
  – Subtraction .............................................................. 68
  / Division ................................................................. 68
  ? : Conditional operator ................................................ 68
  < Less than ............................................................... 69
  <= Less than or equal .................................................. 69
  != Not equal ............................................................... 69
  == Equal ................................................................. 69
  > Greater than ........................................................... 70
  >= Greater than or equal ................................................ 70
  && Logical AND ............................................................ 70
  & Bitwise AND ............................................................. 71
  ~ Bitwise NOT ............................................................ 71
  | Bitwise OR ............................................................. 71
  ^ Bitwise exclusive OR .................................................. 71
% Modulo .......................................................................................... 72
! Logical NOT .................................................................................... 72
|| Logical OR ..................................................................................... 72
<< Logical shift left ......................................................................... 72
>> Logical shift right ........................................................................ 73
BYTE1 First byte ............................................................................. 73
BYTE2 Second byte .......................................................................... 73
BYTE3 Third byte ........................................................................... 74
BYTE4 Fourth byte .......................................................................... 74
DATE Current time/date .................................................................... 74
H1 High word ..................................................................................... 75
HIGH High byte ................................................................................. 75
HWRD High word ............................................................................. 75
LOW Low byte .................................................................................. 75
LW1 Low word .................................................................................. 76
LWRD Low word .............................................................................. 76
RELEP ............................................................................................... 76
RELGP ............................................................................................... 76
RELTP ............................................................................................... 77
SFB section begin  ............................................................................. 77
SFE section end .................................................................................. 77
SIZEOF section size ........................................................................... 78
UGT Unsigned greater than .............................................................. 79
ULT Unsigned less than ........................................................................ 79
UPPER Third byte ............................................................................. 79
XOR Logical exclusive OR ............................................................... 80

Assembler directives ........................................................................ 81

Summary of assembler directives ................................................... 81
Description of assembler directives ............................................... 85
Module control directives ................................................................. 85
Symbol control directives ................................................................. 87
Section control directives ................................................................. 89
Value assignment directives ............................................................. 93
Tables

1: Typographic conventions used in this guide ......................................................... 12
2: Naming conventions used in this guide .............................................................. 13
3: Assembler environment variables ...................................................................... 18
4: Assembler error return codes ........................................................................... 18
5: Integer constant formats ....................................................................................... 20
6: ASCII character constant formats ...................................................................... 21
7: Floating-point constants ....................................................................................... 21
8: Predefined register symbols ................................................................................ 23
9: Predefined symbols ............................................................................................... 24
10: Symbol and cross-reference table ...................................................................... 28
11: Code sample with call frame information ........................................................... 37
12: Assembler options summary .............................................................................. 41
13: Assembler directives summary .......................................................................... 81
14: Module control directives ................................................................................... 85
15: Symbol control directives ................................................................................... 88
16: Section control directives ................................................................................... 90
17: Operand modifiers ............................................................................................... 93
18: Value assignment directives ............................................................................... 94
19: Macro processing directives ............................................................................... 97
20: Listing control directives .................................................................................... 104
21: C-style preprocessor directives .......................................................................... 109
22: Data definition or allocation directives ............................................................... 114
23: Assembler control directives ............................................................................. 116
24: Call frame information directives names block ................................................. 119
25: Call frame information directives common block .............................................. 121
26: Call frame information directives for data blocks ............................................. 122
27: Unary operators in CFI expressions .................................................................. 123
28: Binary operators in CFI expressions .................................................................. 123
29: Ternary operators in CFI expressions ................................................................ 124
30: Call frame information directives for tracking resources and CFAs ............ 125
31: Call frame information directives for stack usage analysis .......................... 125
32: Pragma directives summary ................................................................. 127
Preface

Welcome to the IAR Assembler Reference Guide for RH850. The purpose of this guide is to provide you with detailed reference information that can help you to use the IAR Assembler for RH850 to develop your application according to your requirements.

Who should read this guide

You should read this guide if you plan to develop an application, or part of an application, using assembler language for the RH850 microcontroller and need to get detailed reference information on how to use the IAR Assembler for RH850. In addition, you should have working knowledge of the following:

- The architecture and instruction set of the RH850 microcontroller (refer to the chip manufacturer’s documentation)
- General assembler language programming
- Application development for embedded systems
- The operating system of your host computer.

How to use this guide

When you first begin using the IAR Assembler for RH850, you should read the chapter Introduction to the IAR Assembler for RH850.

If you are an intermediate or advanced user, you can focus more on the reference chapters that follow the introduction.

If you are new to using the IAR Embedded Workbench, we recommend that you first work through the tutorials, which you can find in the IAR Information Center and which will help you get started using IAR Embedded Workbench.
What this guide contains

Below is a brief outline and summary of the chapters in this guide.

- *Introduction to the IAR Assembler for RH850* provides programming information. It also describes the source code format, and the format of assembler listings.
- *Assembler options* first explains how to set the assembler options from the command line and how to use environment variables. It then gives an alphabetical summary of the assembler options, and contains detailed reference information about each option.
- *Assembler operators* gives a summary of the assembler operators, arranged in order of precedence, and provides detailed reference information about each operator.
- *Assembler directives* gives an alphabetical summary of the assembler directives, and provides detailed reference information about each of the directives, classified into groups according to their function.
- *Pragma directives* describes the pragma directives available in the assembler.
- *Diagnostics* contains information about the formats and severity levels of diagnostic messages.

Document conventions

When, in the IAR Systems documentation, we refer to the programming language C, the text also applies to C++, unless otherwise stated.

When referring to a directory in your product installation, for example `rh850\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench N.n\rh850\doc`, where the initial digit of the version number reflects the initial digit of the version number of the IAR Embedded Workbench shared components.

**TYPOGRAPHIC CONVENTIONS**

The IAR Systems documentation set uses the following typographic conventions:

<table>
<thead>
<tr>
<th>Style</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer</td>
<td>• Source code examples and file paths.</td>
</tr>
<tr>
<td></td>
<td>• Text on the command line.</td>
</tr>
<tr>
<td></td>
<td>• Binary, hexadecimal, and octal numbers.</td>
</tr>
<tr>
<td>parameter</td>
<td>A placeholder for an actual value used as a parameter, for example <code>filename.h</code> where <code>filename</code> represents the name of the file.</td>
</tr>
</tbody>
</table>

Table 1: Typographic conventions used in this guide
The following naming conventions are used for the products and tools from IAR Systems®, when referred to in the documentation:

<table>
<thead>
<tr>
<th>Style</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>[option]</td>
<td>An optional part of a directive, where [ and ] are not part of the actual directive, but any [, ], {, or } are part of the directive syntax.</td>
</tr>
<tr>
<td>{option}</td>
<td>A mandatory part of a directive, where { and } are not part of the actual directive, but any [, ], {, or } are part of the directive syntax.</td>
</tr>
<tr>
<td>[option]</td>
<td>An optional part of a command.</td>
</tr>
<tr>
<td>[a</td>
<td>b</td>
</tr>
<tr>
<td>{a</td>
<td>b</td>
</tr>
<tr>
<td>bold</td>
<td>Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.</td>
</tr>
<tr>
<td>italic</td>
<td>• A cross-reference within this guide or to another guide.</td>
</tr>
<tr>
<td></td>
<td>• Emphasis.</td>
</tr>
<tr>
<td>...</td>
<td>An ellipsis indicates that the previous item can be repeated an arbitrary number of times.</td>
</tr>
<tr>
<td></td>
<td>Identifies instructions specific to the IAR Embedded Workbench® IDE interface.</td>
</tr>
<tr>
<td></td>
<td>Identifies instructions specific to the command line interface.</td>
</tr>
<tr>
<td></td>
<td>Identifies helpful tips and programming hints.</td>
</tr>
<tr>
<td></td>
<td>Identifies warnings.</td>
</tr>
</tbody>
</table>

Table 1: Typographic conventions used in this guide (Continued)

**NAMING CONVENTIONS**

The following naming conventions are used for the products and tools from IAR Systems®, when referred to in the documentation:

<table>
<thead>
<tr>
<th>Brand name</th>
<th>Generic term</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAR Embedded Workbench® for RH850</td>
<td>IAR Embedded Workbench®</td>
</tr>
<tr>
<td>IAR Embedded Workbench® IDE for RH850</td>
<td>the IDE</td>
</tr>
<tr>
<td>IAR C-SPY® Debugger for RH850</td>
<td>C-SPY, the debugger</td>
</tr>
<tr>
<td>IAR C-SPY® Simulator</td>
<td>the simulator</td>
</tr>
<tr>
<td>IAR C/C++ Compiler™ for RH850</td>
<td>the compiler</td>
</tr>
<tr>
<td>IAR Assembler™ for RH850</td>
<td>the assembler</td>
</tr>
<tr>
<td>IAR ILINK Linker™</td>
<td>ILINK, the linker</td>
</tr>
<tr>
<td>IAR DLIB Runtime Environment™</td>
<td>the DLIB runtime environment</td>
</tr>
</tbody>
</table>

Table 2: Naming conventions used in this guide
Introduction to the IAR Assembler for RH850

- Introduction to assembler programming
- Modular programming
- External interface details
- Source format
- Assembler instructions
- Expressions, operands, and operators
- List file format
- Programming hints
- Tracking call frame usage

Introduction to assembler programming

Even if you do not intend to write a complete application in assembler language, there might be situations where you find it necessary to write parts of the code in assembler, for example, when using mechanisms in the RH850 microcontroller that require precise timing and special instruction sequences.

To write efficient assembler applications, you should be familiar with the architecture and instruction set of the RH850 microcontroller. Refer to the Renesas hardware documentation for syntax descriptions of the instruction mnemonics.

GETTING STARTED

To ease the start of the development of your assembler application, you can:

- Work through the tutorials—especially the one about mixing C and assembler modules—that you find in the Information Center
- Read about the assembler language interface—also useful when mixing C and assembler modules—in the IAR C/C++ Development Guide for RH850
In the IAR Embedded Workbench IDE, you can base a new project on a template for an assembler project.

**Modular programming**

It is widely accepted that modular programming is a prominent feature of good software design. If you structure your code in small modules—in contrast to one single monolith—you can organize your application code in a logical structure, which makes the code easier to understand, and which aids:

- efficient program development
- reuse of modules
- maintenance.

The IAR development tools provide different facilities for achieving a modular structure in your software.

Typically, you write your assembler code in assembler source files; each file becomes a named module. If you divide your source code into many small source files, you will get many small modules. You can divide each module further into different subroutines.

A section is a logical entity containing a piece of data or code that should be mapped to a physical location in memory. Use the section control directives to place your code and data in sections. A section is relocatable. An address for a relocatable section is resolved at link time. Sections let you control how your code and data is placed in memory. A section is the smallest linkable unit, which allows the linker to include only those units that are referred to.

If you are working on a large project you will soon accumulate a collection of useful routines that are used by several of your applications. To avoid ending up with a huge amount of small object files, collect modules that contain such routines in a library object file. Note that a module in a library is always conditionally linked. In the IAR Embedded Workbench IDE, you can set up a library project, to collect many object files in one library. For an example, see the tutorials in the Information Center.

To summarize, your software design benefits from modular programming, and to achieve a modular structure you can:

- Create many small modules, one per source file
- In each module, divide your assembler source code into small subroutines (corresponding to functions on the C level)
- Divide your assembler source code into sections, to gain more precise control of how your code and data finally is placed in memory
● Collect your routines in libraries, which means that you can reduce the number of object files and make the modules conditionally linked.

**External interface details**

This section provides information about how the assembler interacts with its environment:

- **Assembler invocation syntax**, page 17
- **Passing options**, page 17
- **Environment variables**, page 18
- **Error return codes**, page 18

You can use the assembler either from the IAR Embedded Workbench IDE or from the command line. Refer to the *IDE Project Management and Building Guide for RH850* for information about using the assembler from the IAR Embedded Workbench IDE.

**ASSEMBLER INVOCATION SYNTAX**

The invocation syntax for the assembler is:

```
iasmrh850 [options] [sourcefile] [options]
```

For example, when assembling the source file `prog.s`, use this command to generate an object file with debug information:

```
iasmrh850 prog --debug
```

By default, the IAR Assembler for RH850 recognizes the filename extensions `.s`, `.asm`, and `.msa` for source files. The default filename extension for assembler output is `.o`.

Generally, the order of options on the command line, both relative to each other and to the source filename, is not significant. However, there is one exception: when you use the `-I` option, the directories are searched in the same order that they are specified on the command line.

If you run the assembler from the command line without any arguments, the assembler version number and all available options including brief descriptions are directed to `stdout` and displayed on the screen.

**PASSING OPTIONS**

You can pass options to the assembler in three different ways:

- Directly from the command line
Specify the options on the command line after the `iasmrh850` command; see "Assembler invocation syntax", page 17.

- Via environment variables
  The assembler automatically appends the value of the environment variables to every command line, so it provides a convenient method of specifying options that are required for every assembly; see "Environment variables", page 18.

- Via a text file by using the `-f` option; see "-f", page 50.

For general guidelines for the option syntax, an options summary, and more information about each option, see the "Assembler options" chapter.

### ENVIRONMENT VARIABLES

You can use these environment variables with the IAR Assembler:

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IASMRH850</td>
<td>Specifies command line options; for example: <code>set IASMRH850=la --warnings_are_errors</code></td>
</tr>
<tr>
<td>IASMRH850_INC</td>
<td>Specifies directories to search for include files; for example: <code>set IASMRH850_INC=c:\myinc\</code></td>
</tr>
</tbody>
</table>

Table 3: Assembler environment variables

For example, setting this environment variable always generates a list file with the name `temp.lst`:

```
set IASMRH850=-l temp.lst
```

For information about the environment variables used by the compiler and linker, see the *IAR C/C++ Development Guide for RH850*.

### ERROR RETURN CODES

When using the IAR Assembler from within a batch file, you might have to determine whether the assembly was successful to decide what step to take next. For this reason, the assembler returns these error return codes:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Assembly successful, warnings might appear.</td>
</tr>
<tr>
<td>1</td>
<td>Warnings occurred, provided that the option <code>--warnings_affect_exit_code</code> was used.</td>
</tr>
<tr>
<td>2</td>
<td>Non-fatal errors or fatal assembly errors occurred (making the assembler abort).</td>
</tr>
<tr>
<td>3</td>
<td>Crashing errors occurred.</td>
</tr>
</tbody>
</table>

Table 4: Assembler error return codes
Source format

The format of an assembler source line is as follows:

```
[label [:]] [operation] [operands] ; comment
```

where the components are as follows:

- **label**
  A definition of a label, which is a symbol that represents an address. If the label starts in the first column—that is, at the far left on the line—the colon (:) is optional.

- **operation**
  An assembler instruction or directive. This must not start in the first column—there must be some whitespace to the left of it.

- **operands**
  An assembler instruction or directive can have zero, one, or more operands. The operands are separated by commas. An operand can be:
  - a constant representing a numeric value or an address
  - a symbolic name representing a numeric value or an address (where the latter also is referred to as a label)
  - a floating-point constant
  - a register
  - a predefined symbol
  - the program location counter (PLC)
  - an expression.

- **comment**
  Comment, preceded by a ; (semicolon)

C or C++ comments are also allowed.

The components are separated by spaces or tabs.

A source line cannot exceed 2047 characters.

Tab characters, ASCII 09H, are expanded according to the most common practice; i.e. to columns 8, 16, 24 etc. This affects the source code output in list files and debug information. Because tabs might be set up differently in different editors, do not use tabs in your source files.
Assembler instructions

The IAR Assembler for RH850 supports the syntax for assembler instructions as described in the hardware documentation from Renesas. It complies with the requirement of the RH850 architecture on word alignment. Any instructions in a code section placed on an odd address results in an error.

Expressions, operands, and operators

Expressions consist of expression operands and operators.

The assembler accepts a wide range of expressions, including both arithmetic and logical operations. All operators use 64-bit two’s complement integers. Range checking is performed if a value is used for generating code.

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators; see also Assembler operators.

These operands are valid in an expression:

- Constants for data or addresses, excluding floating-point constants.
- Symbols—symbolic names—which can represent either data or addresses, where the latter also is referred to as labels.
- The program location counter (PLC), $ (dollar).

The operands are described in greater details on the following pages.

Note: You cannot have two symbols in one expression, or any other complex expression, unless the expression can be resolved at assembly time. If they are not resolved, the assembler generates an error.

INTEGER CONSTANTS

Because all IAR Systems assemblers use 64-bit two’s complement internal arithmetic, integers have a (signed) range from $-2^{63}$ to $2^{63}-1$.

Constants are written as a sequence of digits with an optional – (minus) sign in front to indicate a negative number.

Commas and decimal points are not permitted.

The following types of number representation are supported:

<table>
<thead>
<tr>
<th>Integer type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>1010b, b'1010</td>
</tr>
</tbody>
</table>

Table 5: Integer constant formats
Introduction to the IAR Assembler for RH850

Note: Both the prefix and the suffix can be written with either uppercase or lowercase letters.

ASCII CHARACTER CONSTANTS

ASCII constants can consist of any number of characters enclosed in single or double quotes. Only printable characters and spaces can be used in ASCII strings. If the quote character itself will be accessed, two consecutive quotes must be used:

**Table 6: ASCII character constant formats**

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘ABCD’</td>
<td>ABCD (four characters).</td>
</tr>
<tr>
<td>‘ABCDE’</td>
<td>ABCD ‘0’ (five characters the last ASCII null).</td>
</tr>
<tr>
<td>‘A’ ‘B’</td>
<td>A B</td>
</tr>
<tr>
<td>‘A’ ‘’</td>
<td>A</td>
</tr>
<tr>
<td>‘’ (4 quotes)</td>
<td>‘’</td>
</tr>
<tr>
<td>‘’ (2 quotes)</td>
<td>Empty string (no value).</td>
</tr>
<tr>
<td>‘ ‘ (2 double quotes)</td>
<td>‘’ (an ASCII null character).</td>
</tr>
<tr>
<td>‘\’</td>
<td>, for quote within a string, as in ‘I’d love to’</td>
</tr>
<tr>
<td>‘\’</td>
<td>, for \ within a string</td>
</tr>
<tr>
<td>‘\’</td>
<td>*, for double quote within a string</td>
</tr>
</tbody>
</table>

FLOATING-POINT CONSTANTS

The IAR Assembler accepts floating-point values as constants and converts them into IEEE single-precision (32-bit) floating-point format, double-precision (64-bit), or fractional format.

Floating-point numbers can be written in the format:

```
[+][-][digits][.digits][E|e][+][-]digits
```

This table shows some valid examples:

**Table 7: Floating-point constants**

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.23</td>
<td>1.023 x 10^1</td>
</tr>
</tbody>
</table>
Spaces and tabs are not allowed in floating-point constants.

Note: Floating-point constants do not give meaningful results when used in expressions.

TRUE AND FALSE

In expressions a zero value is considered false, and a non-zero value is considered true.

Conditional expressions return the value 0 for false and 1 for true.

SYMBOLS

User-defined symbols can be up to 255 characters long, and all characters are significant. Depending on what kind of operation a symbol is followed by, the symbol is either a data symbol or an address symbol where the latter is referred to as a label. A symbol before an instruction is a label and a symbol before, for example the EQU directive, is a data symbol. A symbol can be:

- absolute—its value is known by the assembler
- relocatable—its value is resolved at link time.

Symbols must begin with a letter, a–z or A–Z, ? (question mark), or _ (underscore). Symbols can include the digits 0–9 and $ (dollar).

Symbols may contain any printable characters if they are quoted with ' (backquote), for example:

`strange#label`

Case is insignificant for built-in symbols like instructions, registers, operators, and directives. For user-defined symbols, case is by default significant but can be turned on and off using the Case sensitive user symbols (--case_insensitive) assembler option. For more information, see --case_insensitive, page 43.

Use the symbol control directives to control how symbols are shared between modules. For example, use the PUBLIC directive to make one or more symbols available to other modules. The EXTERN directive is used for importing an untyped external symbol.

Note that symbols and labels are byte addresses. See also Data definition or allocation directives, page 113.

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.23456E-24</td>
<td>1.23456 x 10^-24</td>
</tr>
<tr>
<td>1.0E3</td>
<td>1.0 x 10^1</td>
</tr>
</tbody>
</table>

Table 7: Floating-point constants (Continued)
LABELS
Symbols used for memory locations are referred to as labels.

Program location counter (PLC)
The assembler keeps track of the start address of the current instruction. This is called the program location counter.

To refer to the program location counter in your assembler source code, use the $ (dollar) character. For example:

```
BR $ ; Loop forever
```

REGISTER SYMBOLS
This table shows the most common predefined register symbols:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP</td>
<td>Element pointer, alias for R30</td>
</tr>
<tr>
<td>GP</td>
<td>Alias for R4</td>
</tr>
<tr>
<td>HP</td>
<td>Alias for R2</td>
</tr>
<tr>
<td>LP</td>
<td>Link pointer, alias for R31</td>
</tr>
<tr>
<td>PC</td>
<td>Program counter</td>
</tr>
<tr>
<td>R0–R31</td>
<td>General purpose registers</td>
</tr>
<tr>
<td>SP</td>
<td>Stack pointer, alias for R3</td>
</tr>
<tr>
<td>TP</td>
<td>Alias for R5</td>
</tr>
<tr>
<td>ZERO</td>
<td>Zero register, alias for R0</td>
</tr>
<tr>
<td>VR0–VR31</td>
<td>Vector registers used by SIMD instructions.</td>
</tr>
</tbody>
</table>

Table 8: Predefined register symbols

For special registers, see the RH850 documentation from Renesas Electronics Corporation.

PREDEFINED SYMBOLS
The IAR Assembler for RH850 defines a set of symbols for use in assembler source files. The symbols provide information about the current assembly, allowing you to test them in preprocessor directives or include them in the assembled code.
These predefined symbols are available:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IASMRH850</strong></td>
<td>An integer that is set to 1 when the code is assembled with the IAR Assembler for RH850.</td>
</tr>
<tr>
<td><strong>BUILD_NUMBER</strong></td>
<td>A unique integer that identifies the build number of the assembler currently in use. The build number does not necessarily increase with an assembler that is released later.</td>
</tr>
<tr>
<td><strong>CORE</strong></td>
<td>An integer that identifies the processor core in use. It is defined to <strong>CORE_G3K</strong>, <strong>CORE_G3KH</strong>, <strong>CORE_G3M</strong>, <strong>CORE_G3MH</strong>, or <strong>CORE_G4MH</strong>. For portability reasons, these definitions also exist: <strong>CORE_V850</strong>, <strong>CORE_V850E</strong>, <strong>CORE_V850E2</strong>, <strong>CORE_V850E2M</strong>, <strong>CORE_V850E2S</strong> and <strong>CORE_V850E2H</strong>.</td>
</tr>
<tr>
<td><strong>CODE_MODEL</strong></td>
<td>An integer that identifies the code model in use. The symbol reflects the --code_model option and can be defined to <strong>CODE_MODEL_NORMAL</strong>, <strong>CODE_MODEL_PIC</strong> (for backward compatibility), or <strong>CODE_MODEL_LARGE</strong> (for backward compatibility).</td>
</tr>
<tr>
<td><strong>DATA_MODEL</strong></td>
<td>An integer that identifies the data model in use. The symbol reflects the --data_model option and can be defined to <strong>DATA_MODEL_TINY</strong>, <strong>DATA_MODEL_SMALL</strong>, <strong>DATA_MODEL_MEDIUM</strong>, or <strong>DATA_MODEL_LARGE</strong>.</td>
</tr>
<tr>
<td><strong>DATE</strong></td>
<td>The current date in dd/Mmm/yyyy format (string).</td>
</tr>
<tr>
<td><strong>FILE</strong></td>
<td>The name of the current source file (string).</td>
</tr>
<tr>
<td><strong>FPU</strong></td>
<td>The symbol reflects the --fpu option and is defined to <strong>FPU_NONE</strong>, <strong>FPU_SINGLE</strong>, or <strong>FPU_DOUBLE</strong>. If this symbol is defined to <strong>FPU_NONE</strong>, an error message will be emitted if FPU instructions are used. If the symbol is defined to <strong>FPU_SINGLE</strong>, an error message will be emitted if double-precision FPU instructions are used.</td>
</tr>
</tbody>
</table>

Table 9: Predefined symbols
including symbol values in code

several data definition directives make it possible to include a symbol value in the code.
these directives define values or reserve memory. to include a symbol value in the code,
use the symbol in the appropriate data definition directive.

for example, to include the time of assembly as a string for the program to display:

name    timeOfAssembly
section .text:CODE(2)
extern  printStr
code
time:   dc8         __TIME__        ; String representing the
timeOfAssembly
         ; time of assembly.
align    1
mov      time,r6     ; Load address of time
         ; string in r6.
jarl     printStr,lp ; Call string output routine.
end

Table 9: Predefined symbols (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IAR_SYSTEMS_ASM</strong></td>
<td>IAR assembler identifier (number). The current value is 9. Note that</td>
</tr>
<tr>
<td></td>
<td>the number could be higher in a future version of the product. This</td>
</tr>
<tr>
<td></td>
<td>symbol can be tested with #ifdef to detect whether the code was</td>
</tr>
<tr>
<td></td>
<td>assembled by an assembler from IAR Systems.</td>
</tr>
<tr>
<td><strong>LINE</strong></td>
<td>The current source line number (number).</td>
</tr>
<tr>
<td><strong>SUBVERSION</strong></td>
<td>An integer that identifies the subversion number of the assembler</td>
</tr>
<tr>
<td></td>
<td>version number, for example 3 in 1.2.3.4.</td>
</tr>
<tr>
<td><strong>TIME</strong></td>
<td>The current time in hh:mm:ss format (string).</td>
</tr>
<tr>
<td><strong>VER</strong></td>
<td>The version number in integer format; for example, version 4.17 is</td>
</tr>
<tr>
<td></td>
<td>returned as 417 (number).</td>
</tr>
</tbody>
</table>

Testing symbols for conditional assembly

To test a symbol at assembly time, use one of the conditional assembly directives. These
directives let you control the assembly process at assembly time.
For example, if you want to assemble separate code sections depending on whether you are using an old assembler version or a new assembler version, do as follows:

```
#if (__VER__ > 300)                 ; New assembler version
    ;…
#else                               ; Old assembler version
    ;…
#endif
```

For more information, see *Conditional assembly directives*, page 95.

**ABSOLUTE AND RELOCATABLE EXPRESSIONS**

Depending on what operands an expression consists of, the expression is either *absolute* or *relocatable*. Absolute expressions are those expressions that only contain absolute symbols or relocatable symbols that cancel each other out.

Expressions that include symbols in relocatable sections cannot be resolved at assembly time, because they depend on the location of sections. These are referred to as relocatable expressions.

Such expressions are evaluated and resolved at link time, by the IAR ILINK Linker. They can only be built up out of a maximum of one symbol reference and an offset after the assembler has reduced it.

For example, a program could define absolute and relocatable expressions as follows:

```
name    simpleExpressions
section MYCONST:CONST(2)
firstra8  5                ; A relocatable label.
second  equ 10 + 5         ; An absolute expression.
    dc8 first             ; Examples of some legal
    dc8 first + 1         ; relocatable expressions.
dc8     first + second
end
```

**Note:** At assembly time, there is no range check. The range check occurs at link time and, if the values are too large, there is a linker error.

**EXPRESSION RESTRICTIONS**

Expressions can be categorized according to restrictions that apply to some of the assembler directives. One such example is the expression used in conditional statements like IF, where the expression must be evaluated at assembly time and therefore cannot contain any external symbols.
The following expression restrictions are referred to in the description of each directive they apply to.

**No forward**
All symbols referred to in the expression must be known, no forward references are allowed.

**No external**
No external references in the expression are allowed.

**Absolute**
The expression must evaluate to an absolute value; a relocatable value (section offset) is not allowed.

**Fixed**
The expression must be fixed, which means that it must not depend on variable-sized instructions. A variable-sized instruction is an instruction that might vary in size depending on the numeric value of its operand.

**List file format**

The format of an assembler list file is as follows:

**HEADER**
The header section contains product version information, the date and time when the file was created, and which options were used.

**BODY**
The body of the listing contains the following fields of information:

- The line number in the source file. Lines generated by macros, if listed, have a . (period) in the source line number field.
- The address field shows the location in memory, which can be absolute or relative depending on the type of section. The notation is hexadecimal.
- The data field shows the data generated by the source line. The notation is hexadecimal. Unresolved values are represented by ..... (periods), where two periods signify one byte. These unresolved values are resolved during the linking process.
- The assembler source line.
Summary
The end of the file contains a summary of errors and warnings that were generated.

Symbol and Cross-Reference Table
When you specify the Include cross-reference option, or if the LSTXRF+ directive was included in the source file, a symbol and cross-reference table is produced.

This information is provided for each symbol in the table:

<table>
<thead>
<tr>
<th>Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>The symbol's user-defined name.</td>
</tr>
<tr>
<td>Mode</td>
<td>ABS (Absolute), or REL (Relocatable).</td>
</tr>
<tr>
<td>Sections</td>
<td>The name of the section that this symbol is defined relative to.</td>
</tr>
<tr>
<td>Value/Offset</td>
<td>The value (address) of the symbol within the current module, relative to</td>
</tr>
<tr>
<td></td>
<td>the beginning of the current section.</td>
</tr>
</tbody>
</table>

Table 10: Symbol and cross-reference table

Programming hints
This section gives hints on how to write efficient code for the IAR Assembler. For information about projects including both assembler and C or C++ source files, see the IAR C/C++ Development Guide for RH850.

Accessing Special Function Registers
Specific header files for several RH850 devices are included in the IAR Systems product package, in the rh850\inc directory. These header files define the processor-specific special function registers (SFRs) and interrupt vector numbers.

The header files are intended to be used also with the IAR C/C++ Compiler for RH850, and they are suitable to use as templates when creating new header files for other RH850 derivatives.

If any assembler-specific additions are needed in the header file, you can easily add these in the assembler-specific part of the file:

```c
#ifdef __IAR_SYSTEMS_ASM__
  ; Add your assembler-specific defines here.
#endif
```

Using C-Style Preprocessor Directives
The C-style preprocessor directives are processed before other assembler directives. Therefore, do not use preprocessor directives in macros and do not mix them with...
assembly-style comments. For more information about comments, see Assembler control directives, page 116.

C-style preprocessor directives like \#define are valid in the remainder of the source code file, while assembler directives like EQU only are valid in the current module.

Tracking call frame usage

In this section, these topics are described:

● Call frame information overview, page 29
● Call frame information in more detail, page 30

These tasks are described:

● Defining a names block, page 31
● Defining a common block, page 32
● Annotating your source code within a data block, page 32
● Specifying rules for tracking resources and the stack depth, page 33
● Using CFI expressions for tracking complex cases, page 35
● Stack usage analysis directives, page 36
● Examples of using CFI directives, page 36

For reference information, see:

● Call frame information directives for names blocks, page 119
● Call frame information directives for common blocks, page 120
● Call frame information directives for data blocks, page 121
● Call frame information directives for tracking resources and CFAs, page 122
● Call frame information directives for stack usage analysis, page 125

CALL FRAME INFORMATION OVERVIEW

Call frame information (CFI) is information about the call frames. Typically, a call frame contains a return address, function arguments, saved register values, compiler temporaries, and local variables. Call frame information holds enough information about call frames to support two important features:

● C-SPY can use call frame information to reconstruct the entire call chain from the current PC (program counter) and show the values of local variables in each function in the call chain.
● Call frame information can be used, together with information about possible calls for calculating the total stack usage in the application. Note that this feature might not be supported by the product you are using.

The compiler automatically generates call frame information for all C and C++ source code. Call frame information is also typically provided for each assembler routine in the system library. However, if you have other assembler routines and want to enable C-SPY to show the call stack when executing these routines, you must add the required call frame information annotations to your assembler source code. Stack usage can also be handled this way (by adding the required annotations for each function call), but you can also specify stack usage information for any routines in a stack usage control file (see the IAR C/C++ Development Guide for RH850), which is typically easier.

CALL FRAME INFORMATION IN MORE DETAIL

You can add call frame information to assembler files by using \texttt{cfi} directives. You can use these to specify:

● The start address of the call frame, which is referred to as the canonical frame address (CFA). There are two different types of call frames:
  ● On a stack—stack frames. For stack frames the CFA is typically the value of the stack pointer after the return from the routine.
  ● In static memory, as used in a static overlay system—static overlay frames. This type of call frame is not required by the RH850 microcontroller and is thus not supported.
● How to find the return address.
● How to restore various resources, like registers, when returning from the routine.

When adding the call frame information for each assembler module, you must:

1. Provide a names block where you describe the resources to be tracked.
2. Provide a common block where you define the resources to be tracked and specify their default values. This information must correspond to the calling convention used by the compiler.
3. Annotate the resources used in your source code, which in practice means that you describe the changes performed on the call frame. Typically, this includes information about when the stack pointer is changed, and when permanent registers are stored or restored on the stack.

To do this you must define a data block that encloses a continuous piece of source code where you specify rules for each resource to be tracked. When the descriptive power of the rules is not enough, you can instead use CFI expressions.
A full description of the calling convention might require extensive call frame information. In many cases, a more limited approach will suffice. The recommended way to create an assembler language routine that handles call frame information correctly is to start with a C skeleton function that you compile to generate assembler output. For an example, see the *IAR C/C++ Development Guide for RH850*.

**DEFINING A NAMES BLOCK**

A names block is used for declaring the resources available for a processor. Inside the names block, all resources that can be tracked are defined.

Start and end a names block with the directives:

```plaintext
CFI NAMES name
CFI ENDNAMES name
```

where `name` is the name of the block.

Only one names block can be open at a time.

Inside a names block, four different kinds of declarations can appear: a resource declaration, a stack frame declaration, a static overlay frame declaration, and a base address declaration:

- To declare a resource, use one of the directives:
  ```plaintext
  CFI RESOURCE resource : bits
  CFI VIRTUALRESOURCE resource : bits
  ```
  The parameters are the name of the resource and the size of the resource in bits. The name must be one of the register names defined in the Renesas ABI specification. A virtual resource is a logical concept, in contrast to a “physical” resource such as a processor register. Virtual resources are usually used for the return address.
  
  To declare more than one resource, separate them with commas.
  
  A resource can also be a composite resource, made up of at least two parts. To declare the composition of a composite resource, use the directive:
  ```plaintext
  CFI RESOURCEPARTS resource, part, ...
  ```
  The parts are separated with commas. The resource and its parts must have been previously declared as resources, as described above.

- To declare a stack frame CFA, use the directive:
  ```plaintext
  CFI STACKFRAME cfa resource type
  ```
  The parameters are the name of the stack frame CFA, the name of the associated resource (the stack pointer), and the memory type (to get the address space). To declare more than one stack frame CFA, separate them with commas.
When going “back” in the call stack, the value of the stack frame CFA is copied into the associated stack pointer resource to get a correct value for the previous function frame.

**DEFINING A COMMON BLOCK**

The common block is used for declaring the initial contents of all tracked resources. Normally, there is one common block for each calling convention used.

Start a common block with the directive:

```c
CFI COMMON name USING namesblock
```

where `name` is the name of the new block and `namesblock` is the name of a previously defined names block.

Declare the return address column with the directive:

```c
CFI RETURNADDRESS resource type
```

where `resource` is a resource defined in `namesblock` and `type` is the memory in which the calling function resides. You must declare the return address column for the common block.

Inside a common block, you can declare the initial value of a CFA or a resource by using the directives available for common blocks, see *Call frame information directives for common blocks*, page 120. For more information about how to use these directives, see *Specifying rules for tracking resources and the stack depth*, page 33 and *Using CFI expressions for tracking complex cases*, page 35.

End a common block with the directive:

```c
CFI ENDCOMMON name
```

where `name` is the name used to start the common block.

**ANNOTATING YOUR SOURCE CODE WITHIN A DATA BLOCK**

The data block contains the actual tracking information for one continuous piece of code.

Start a data block with the directive:

```c
CFI BLOCK name USING commonblock
```

where `name` is the name of the new block and `commonblock` is the name of a previously defined common block.
If the piece of code for the current data block is part of a defined function, specify the name of the function with the directive:

```
CFI FUNCTION label
```

where `label` is the code label starting the function.

If the piece of code for the current data block is not part of a function, specify this with the directive:

```
CFI NOFUNCTION
```

End a data block with the directive:

```
CFI ENDBLOCK name
```

where `name` is the name used to start the data block.

Inside a data block, you can manipulate the values of the resources by using the directives available for data blocks, see *Call frame information directives for data blocks*, page 121. For more information on how to use these directives, see *Specifying rules for tracking resources and the stack depth*, page 33, and *Using CFI expressions for tracking complex cases*, page 35.

**SPECIFYING RULES FOR TRACKING RESOURCES AND THE STACK DEPTH**

To describe the tracking information for individual resources, two sets of simple rules with specialized syntax can be used:

- **Rules for tracking resources**
  ```
  CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
  CFI resource { resource | FRAME(cfa, offset) }
  ```

- **Rules for tracking the stack depth (CFAs)**
  ```
  CFI cfa { NOTUSED | USED }
  CFI cfa { resource | resource + constant | resource - constant }
  ```

You can use these rules both in common blocks to describe the initial information for resources and CFAs, and inside data blocks to describe changes to the information for resources or CFAs.

In those rare cases where the descriptive power of the simple rules are not enough, you can use a full CFI expression with dedicated operators to describe the information, see *Using CFI expressions for tracking complex cases*, page 35. However, whenever possible, you should always use a rule instead of a CFI expression.
Rules for tracking resources

The rules for resources conceptually describe where to find a resource when going back one call frame. For this reason, the item following the resource name in a CFI directive is referred to as the location of the resource.

To declare that a tracked resource is restored, in other words, already correctly located, use SAMEVALUE as the location. Conceptually, this declares that the resource does not have to be restored because it already contains the correct value. For example, to declare that a register R11 is restored to the same value, use the directive:

```plaintext
CFI R11 SAMEVALUE
```

To declare that a resource is not tracked, use UNDEFINED as location. Conceptually, this declares that the resource does not have to be restored (when going back one call frame) because it is not tracked. Usually it is only meaningful to use it to declare the initial location of a resource. For example, to declare that R11 is a scratch register and does not have to be restored, use the directive:

```plaintext
CFI R11 UNDEFINED
```

To declare that a resource is temporarily stored in another resource, use the resource name as its location. For example, to declare that a register R11 is temporarily located in a register R12 (and should be restored from that register), use the directive:

```plaintext
CFI R11 R12
```

To declare that a resource is currently located somewhere on the stack, use FRAME(cfa, offset) as location for the resource, where cfa is the CFA identifier to use as “frame pointer” and offset is an offset relative the CFA. For example, to declare that a register R11 is located at offset –4 counting from the frame pointer CPA_SP, use the directive:

```plaintext
CFI R11 FRAME(CPA_SP,-4)
```

For a composite resource there is one additional location, CONCAT, which declares that the location of the resource can be found by concatenating the resource parts for the composite resource. For example, consider a composite resource RET with resource parts RETLO and RETHI. To declare that the value of RET can be found by investigating and concatenating the resource parts, use the directive:

```plaintext
CFI RET CONCAT
```

This requires that at least one of the resource parts has a definition, using the rules described above.

Rules for tracking the stack depth (CFAs)

In contrast to the rules for resources, the rules for CFAs describe the address of the beginning of the call frame. The call frame often includes the return address pushed by
the assembler call instruction. The CFA rules describe how to compute the address of the beginning of the current stack frame.

Each stack frame CFA is associated with a stack pointer. When going back one call frame, the associated stack pointer is restored to the current CFA. For stack frame CFAs there are two possible rules: an offset from a resource (not necessarily the resource associated with the stack frame CFA) or NOTUSED.

To declare that a CFA is not used, and that the associated stack pointer should be tracked as a normal resource, use NOTUSED as the address of the CFA. For example, to declare that the CFA with the name CFA_SP is not used in this code block, use the directive:

```
CFI CFA_SP NOTUSED
```

To declare that a CFA has an address that is offset relative the value of a resource, specify the stack pointer and the offset. For example, to declare that the CFA with the name CFA_SP can be obtained by adding 4 to the value of the SP resource, use the directive:

```
CFI CFA_SP SP + 4
```

**USING CFI EXPRESSIONS FOR TRACKING COMPLEX CASES**

You can use call frame information expressions (CFI expressions) when the descriptive power of the rules for resources and CFAs is not enough. However, you should always use a simple rule if there is one.

CFI expressions consist of operands and operators. Three sets of operators are allowed in a CFI expression:

- Unary operators
- Binary operators
- Ternary operators

In most cases, they have an equivalent operator in the regular assembler expressions.

In this example, R12 is restored to its original value. However, instead of saving it, the effect of the two post increments is undone by the subtract instruction.
Tracking call frame usage

AddTwo:
   cfi block addTwoBlock using myCommon
   cfi function addTwo
   cfi nocalls
   cfi r12 samevalue
   add @r12+, r13
   cfi r12 sub(r12, 2)
   add @r12+, r13
   cfi r12 sub(r12, 4)
   sub #4, r12
   cfi r12 samevalue
   ret
   cfi endblock addTwoBlock

For more information about the syntax for using the operators in CFI expressions, see
Call frame information directives for tracking resources and CFIs, page 122.

STACK USAGE ANALYSIS DIRECTIVES

The stack usage analysis directives (CFI FCALL, CFI TAILCALL, CFI
INDIRECTCALL, and CFI NOCALLS) are used for building a call graph which is needed
for stack usage analysis. These directives can be used only in data blocks. When the data
block is a function block (in other words, when the CFI FUNCTION directive has been
used in the data block), you should not specify a caller parameter. When a stack usage
analysis directive is used in code that is shared between functions, you must use the
caller parameter to specify which of the possible functions the information applies to.
The CFI FCALL, CFI TAILCALL, and CFI INDIRECTCALL directives must be placed
immediately before the instruction that performs the call. The CFI NOCALLS directive
can be placed anywhere in the data block.

EXAMPLES OF USING CFI DIRECTIVES

The following is a generic example of how to add and use the required CFI directives.
The example is not specific to the RH850 microcontroller. To obtain an example specific
to the microcontroller you are using, generate assembler output when you compile a C
source file.

Consider a generic processor with a stack pointer SP, and two registers R0 and R1.
Register R0 is used as a scratch register (the register may be destroyed by a function
call), whereas register R1 must be restored after the function call. To simplify, all
instructions, registers, and addresses are assumed to have a width of 16 bits.

Consider the following short code example with the corresponding call frame
information. At entry, assume that the stack contains a 16-bit return address. The stack
The CFA denotes the top of the call frame, in other words, the value of the stack pointer after returning from the function.

### Table 11: Code sample with call frame information

<table>
<thead>
<tr>
<th>Address</th>
<th>CFA</th>
<th>R0</th>
<th>R1</th>
<th>RET</th>
<th>Assembler code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SP + 2</td>
<td>undefined</td>
<td>SAME</td>
<td>CFA - 2</td>
<td>func1: PUSH R1</td>
</tr>
<tr>
<td>0002</td>
<td>SP + 4</td>
<td>CFA - 4</td>
<td></td>
<td></td>
<td>MOV R1,#4</td>
</tr>
<tr>
<td>0004</td>
<td></td>
<td></td>
<td>CALL func2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0006</td>
<td></td>
<td>POP</td>
<td>R0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0008</td>
<td>SP + 2</td>
<td>R0</td>
<td>MOV</td>
<td>R1,R0</td>
<td></td>
</tr>
<tr>
<td>000A</td>
<td></td>
<td>SAME</td>
<td>RET</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each row describes the state of the tracked resources before the execution of the instruction. As an example, for the MOV R1,R0 instruction the original value of the R1 register is located in the R0 register and the top of the function frame (the CFA column) is SP + 2. The row at address 0000 is the initial row and the result of the calling convention used for the function.

The RET column is the return address column—that is, the location of the return address. The value of R0 is undefined because it does not need to be restored on exit from the function. The R1 column has SAME in the initial row to indicate that the value of the R1 register will be restored to the same value it already has.

### Defining the names block

The names block for the small example above would be:

```c
  cfi     names trivialNames
  cfi     resource SP:16, R0:16, R1:16
  cfi     stackframe CFA SP DATA

; The virtual resource for the return address column.
  cfi     virtualresource RET:16
  cfi     endnames trivialNames
```

...
Defining the common block

The common block for the simple example above would be:

```
cfi    common trivialCommon using trivialNames
cfi    returnaddress RET DATA
cfi    CPA SP + 2
cfi    R0 undefined
cfi    R1 samevalue

; Offset -2 from top of frame.
cfi    RET frame(CFA,-2)
cfi    endcommon trivialCommon
```

**Note:** SP cannot be changed using a CFI directive as it is the resource associated with CFA.

Annotating your source code within a data block

You should place the CFI directives at the point where the call frame information has changed, in other words, immediately after the instruction that changes the call frame information.

Continuing the simple example, the data block would be:

```
rseg    CODE:CODE
cfi    block func1block using trivialCommon
cfi    function func1

func1   push    r1
cfi    CPA SP + 4
cfi    R1 frame(CFA,-4)
mov     r1,#4
call    func2
pop     r0
cfi    R1 R0
cfi    CPA SP + 2
mov     r1,r0
cfi    R1 samevalue
ret
```
Assembler options

- Using command line assembler options
- Summary of assembler options
- Description of assembler options

Using command line assembler options

Assembler options are parameters you can specify to change the default behavior of the assembler. You can specify options from the command line—which is described in more detail in this section—and from within the IAR Embedded Workbench® IDE.

The IDE Project Management and Building Guide for RH850 describes how to set assembler options in the IDE, and gives reference information about the available options.

SPECIFYING COMMAND LINE OPTIONS

To set assembler options from the command line, include them on the command line after the `iasmrh850` command, either before or after the source filename. For example, when assembling the source file `prog.s`, use this command to generate an object file with debug information:

```
iasmrh850 prog.s --debug
```

Some options accept a filename, included after the option letter with a separating space. For example, to generate a listing to the file `prog.lst`:

```
iasmrh850 prog.s -l prog.lst
```

Some other options accept a string that is not a filename. The string is included after the option letter, but without a space. For example, to define a symbol:

```
iasmrh850 prog.s -DDEBUG=1
```

Generally, the order of options on the command line, both relative to each other and to the source filename, is not significant. However, there is one exception: when you use the `-I` option, the directories are searched in the same order as they are specified on the command line.

Notice that a command line option has a short name and/or a long name:

- A short option name consists of one character, with or without parameters. You specify it with a single dash, for example `-r`. 
A long name consists of one or several words joined by underscores, with or without parameters. You specify it with double dashes, for example --debug.

**SPECIFYING PARAMETERS**

When a parameter is needed for an option with a short name, you can specify it either immediately following the option or as the next command line argument.

For instance, you can specify an include file path of \usr\include either as:

- `-I\usr\include`

or as

- `-I \usr\include`

**Note:** You can use / instead of \ as directory delimiter. A trailing slash or backslash can be added to the last directory name, but is not required.

Additionally, some options can take a parameter that is a directory name. The output file then receives a default name and extension.

When a parameter is needed for an option with a long name, you can specify it either immediately after the equal sign (=) or as the next command line argument, for example:

- `--diag_suppress=Pe0001`

or

- `--diag_suppress Pe0001`

Options that accept multiple values can be repeated, and can also have comma-separated values (without space), for example:

- `--diag_warning=Be0001,Be0002`

The current directory is specified with a period (.), for example:

- `iasmrh850 prog -l .`

A file specified by `-` (a single dash) is standard input or output, whichever is appropriate.

**Note:** When an option takes a parameter, the parameter cannot start with a dash (-) followed by another character. Instead you can prefix the parameter with two dashes (--). This example generates a list on standard output:

- `iasmrh850 prog -l ---`

**EXTENDED COMMAND LINE FILE**

In addition to accepting options and source filenames from the command line, the assembler can accept them from an extended command line file.
By default, extended command line files have the extension xcl, and can be specified using the -f command line option. For example, to read the command line options from extend.xcl, enter:
iasmrh850 -f extend.xcl

### Summary of assembler options

This table summarizes the assembler options available from the command line:

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--case_insensitive</td>
<td>Case-insensitive user symbols</td>
</tr>
<tr>
<td>--code_model</td>
<td>Sets the predefined symbol <strong>CODE_MODEL</strong></td>
</tr>
<tr>
<td>--core</td>
<td>Sets the predefined symbol <strong>CORE</strong></td>
</tr>
<tr>
<td>-D</td>
<td>Defines preprocessor symbols</td>
</tr>
<tr>
<td>--data_model</td>
<td>Sets the predefined symbol <strong>DATA_MODEL</strong></td>
</tr>
<tr>
<td>--debug</td>
<td>Generates debug information</td>
</tr>
<tr>
<td>--dependencies</td>
<td>Lists file dependencies</td>
</tr>
<tr>
<td>--diag_error</td>
<td>Treats these diagnostics as errors</td>
</tr>
<tr>
<td>--diag_remark</td>
<td>Treats these diagnostics as remarks</td>
</tr>
<tr>
<td>--diag_suppress</td>
<td>Suppresses these diagnostics</td>
</tr>
<tr>
<td>--diag_warning</td>
<td>Treats these diagnostics as warnings</td>
</tr>
<tr>
<td>--diagnostics_tables</td>
<td>Lists all diagnostic messages</td>
</tr>
<tr>
<td>--dir_first</td>
<td>Allows directives in the first column</td>
</tr>
<tr>
<td>--error_limit</td>
<td>Specifies the allowed number of errors before the assembler stops</td>
</tr>
<tr>
<td>-f</td>
<td>Extends the command line</td>
</tr>
<tr>
<td>--f</td>
<td>Extends the command line, optionally with a dependency</td>
</tr>
<tr>
<td>--fpu</td>
<td>Sets the predefined symbol <strong>FPU</strong></td>
</tr>
<tr>
<td>--header_context</td>
<td>Lists all referred source files</td>
</tr>
<tr>
<td>-I</td>
<td>Adds a search path for a header file</td>
</tr>
<tr>
<td>-l</td>
<td>Generates a list file</td>
</tr>
<tr>
<td>-M</td>
<td>Macro quote characters</td>
</tr>
<tr>
<td>--macro_positions_in_diagnostics</td>
<td>Obtains positions inside macros in diagnostic messages</td>
</tr>
</tbody>
</table>

Table 12: Assembler options summary
Description of assembler options

The following sections give detailed reference information about each assembler option.

Note that if you use the page Extra Options to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--enem_first</td>
<td>Allows mnemonics in the first column</td>
</tr>
<tr>
<td>--no_bom</td>
<td>Omits the Byte Order Mark for UTF-8 output files</td>
</tr>
<tr>
<td>--no_path_in_file_macros</td>
<td>Removes the path from the return value of the symbols <strong>FILE</strong> and <strong>BASE_FILE</strong></td>
</tr>
<tr>
<td>--no_system_include</td>
<td>Disables the automatic search for system include files</td>
</tr>
<tr>
<td>--no_warnings</td>
<td>Disables all warnings</td>
</tr>
<tr>
<td>--no_wrap_diagnostics</td>
<td>Disables wrapping of diagnostic messages</td>
</tr>
<tr>
<td>-o</td>
<td>Sets the object filename. Alias for --output.</td>
</tr>
<tr>
<td>--only_stdout</td>
<td>Uses standard output only</td>
</tr>
<tr>
<td>--output</td>
<td>Sets the object filename</td>
</tr>
<tr>
<td>--predef_macros</td>
<td>Lists the predefined symbols</td>
</tr>
<tr>
<td>--preinclude</td>
<td>Includes an include file before reading the source file</td>
</tr>
<tr>
<td>--preprocess</td>
<td>Preprocessor output to file</td>
</tr>
<tr>
<td>-r</td>
<td>Generates debug information. Alias for --debug.</td>
</tr>
<tr>
<td>--remarks</td>
<td>Enables remarks</td>
</tr>
<tr>
<td>--silent</td>
<td>Sets silent operation</td>
</tr>
<tr>
<td>--source_encoding</td>
<td>Specifies the encoding for source files</td>
</tr>
<tr>
<td>--system_include_dir</td>
<td>Specifies the path for system include files</td>
</tr>
<tr>
<td>--text_out</td>
<td>Specifies the encoding for text output files</td>
</tr>
<tr>
<td>--use_unix_directory_separators</td>
<td>Uses / as directory separator in paths</td>
</tr>
<tr>
<td>--utf8_text_in</td>
<td>Uses the UTF-8 encoding for text input files</td>
</tr>
<tr>
<td>--warnings_affect_exit_code</td>
<td>Warnings affect exit code</td>
</tr>
<tr>
<td>--warnings_are_errors</td>
<td>Treats all warnings as errors</td>
</tr>
</tbody>
</table>

Table 12: Assembler options summary (Continued)
--case_insensitive

Syntax

```
--case_insensitive
```

Description

Use this option to make user symbols case-insensitive. By default, case sensitivity is on.

You can also use the assembler directives `CASEON` and `CASEOFF` to control case sensitivity for user-defined symbols.

**Note:** The `--case_insensitive` option does not affect preprocessor symbols. Preprocessor symbols are always case-sensitive, regardless of whether they are defined in the IDE or on the command line.

Example

By default, for example, `LABEL` and `label` refer to different symbols. When `--case_insensitive` is used, `LABEL` and `label` instead refer to the same symbol.

See also

`Assembler control directives`, page 116 and information about defining and undefining preprocessor symbols under `C-style preprocessor directives`, page 108.

---

Project>Options>Assembler >Language>User symbols are case sensitive

--code_model

Syntax

```
--code_model={normal|pic|large}
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>(default)   Sets the predefined symbol <code>__CODE_MODEL__</code> to <code>__CODE_MODEL_NORMAL__</code>.</td>
</tr>
<tr>
<td>pic</td>
<td>Sets the predefined symbol <code>__CODE_MODEL__</code> to <code>__CODE_MODEL_PIC__</code>. Included for backward compatibility.</td>
</tr>
<tr>
<td>large</td>
<td>Sets the predefined symbol <code>__CODE_MODEL__</code> to <code>__CODE_MODEL_LARGE__</code>. Included for backward compatibility.</td>
</tr>
</tbody>
</table>

Description

Use this option to define the symbol `__CODE_MODEL__`.

See also

`Predefined symbols`, page 23.

This option is not available in the IDE.
Description of assembler options

--core

Syntax

--core={g3k|g3kh|g3m|g3mh|g4mh}

Parameters

g3k
Sets the predefined symbol __CORE__ to __CORE_G3K__.
g3kh
Sets the predefined symbol __CORE__ to __CORE_G3KH__.
g3m (default)
Sets the predefined symbol __CORE__ to __CORE_G3M__.
g3mh
Sets the predefined symbol __CORE__ to __CORE_G3MH__.
g4mh
Sets the predefined symbol __CORE__ to __CORE_G4MH__.

Description
Use this option to define the symbol __CORE__.

See also
Predefined symbols, page 23.

To set related options, select:
Project>Options>General Options>Target>Device

-D

Syntax

-Dsymbol[=value]

Parameters

symbol
The name of the symbol you want to define.
value
The value of the symbol. If no value is specified, 1 is used.

Description
Use this option to define a symbol to be used by the preprocessor.

Example
You might want to arrange your source code to produce either the test version or the production version of your application, depending on whether the symbol TESTVER was defined. To do this, use include sections such as:

```c
#ifdef TESTVER
... ; additional code lines for test version only
#endif
```

Then select the version required on the command line as follows:

Production version:  iasmRH850 prog
Test version:  iasmRH850 prog -DTESTVER
Alternatively, your source might use a variable that you must change often. You can then leave the variable undefined in the source, and use `-D` to specify the value on the command line; for example:

```
iasmRH850 prog -DFRAMERATE=3
```

**--data_model**

**Syntax**

```
--data_model=(tiny|small|medium|large)
```

**Parameters**

- `tiny` Sets the predefined symbol `__DATA_MODEL__` to `__DATA_MODEL_TINY__`
- `small` (default) Sets the predefined symbol `__DATA_MODEL__` to `__DATA_MODEL_SMALL__`
- `medium` Sets the predefined symbol `__DATA_MODEL__` to `__DATA_MODEL_MEDIUM__`
- `large` Sets the predefined symbol `__DATA_MODEL__` to `__DATA_MODEL_LARGE__`

**Description**

Use this option to define the symbol `__DATA_MODEL__`.

**See also**

*Predefined symbols*, page 23.

To set related options, select:

**Project>Options>Assembler>Preprocessor>Defined symbols**

**--debug, -r**

**Syntax**

```
--debug
-r
```

**Description**

Use this option to make the assembler generate debug information, which means the generated output can be used in a symbolic debugger such as IAR C-SPY® Debugger.
To reduce the size and link time of the object file, the assembler does not generate debug information by default.

**Project > Options > Assembler > Output > Generate debug information**

---

**--dependencies**

**Syntax**

```
--dependencies=[i][m] {filename|directory}
```

**Parameters**

- **No parameter**
  - The same affect as for the parameter `i`.
- **i (default)**
  - The names of the dependent files, including the full path if available, is output. For example:
    ```
c:\iar\product\include\stdio.h
d:\myproject\include\foo.h
    ```
- **m**
  - The output uses makefile style. For each source file, one line containing a makefile dependency rule is output. Each line consists of the name of the object file, a colon, a space, and the name of a source file. For example:
    ```
    foo.o: c:\iar\product\include\stdio.h
    foo.o: d:\myproject\include\foo.h
    ```
- **filename**
  - The output is stored in the specified file.
- **directory**
  - The output is stored in a file (filename extension `i`) which is stored in the specified directory.

**Description**

Use this option to list each source file opened by the assembler in a file.

**Example**

To generate a listing of file dependencies to the file `listing.i`, use:

```
iasmrh850 prog --dependencies=i listing
```

An example of using `--dependencies` with gmake:

```
1
```

Set up the rule for assembling files to be something like:

```
%.o : %.c
$(ASM) $(ASMFLAGS) $< --dependencies=m $*.d
```
That is, in addition to producing an object file, the command also produces a dependent file in makefile style (in this example using the extension .d).

2 Include all the dependent files in the makefile, using for example:

```
-include $(sources:.c=.d)
```

Because of the -, it works the first time, when the .d files do not yet exist.

This option is not available in the IDE.

---

**--diag_error**

**Syntax**

```
--diag_error=tag,tag,...
```

**Parameters**

tag The number of a diagnostic message, for example the message number As001.

**Description**

Use this option to classify diagnostic messages as errors.

An error indicates a violation of the assembler language rules, of such severity that object code is not generated, and the exit code will not be 0. The option can be used more than once on the command line.

**Example**

This example classifies warning As001 as an error:

```
--diag_error=As001
```

---

**--diag_remark**

**Syntax**

```
--diag_remark=tag,tag,...
```

**Parameters**

tag The number of a diagnostic message, for example the message number As001.

**Description**

Use this option to classify diagnostic messages as remarks.

A remark is the least severe type of diagnostic message and indicates a source code construct that might cause strange behavior in the generated code.
Example This example classifies the warning As001 as a remark:
--diag_remark=As001

Project>Options>Assembler >Diagnostics>Treat these as remarks

--diag_suppress

Syntax --diag_suppress=tag,tag,...

Parameters tag The number of a diagnostic message, for example the message number As001.

Description Use this option to suppress diagnostic messages.

Example This example suppresses the warnings As001 and As002:
--diag_suppress=As001,As002

Project>Options>Assembler >Diagnostics>Suppress these diagnostics

--diag_warning

Syntax --diag_warning=tag,tag,...

Parameters tag The number of a diagnostic message, for example the message number As001.

Description Use this option to classify diagnostic messages as warnings.

A warning indicates an error or omission that is of concern, but which does not cause the assembler to stop before the assembly is completed.

Example This example classifies the remark As028 as a warning:
--diag_warning=As028

Project>Options>Assembler >Diagnostics>Treat these as warnings
--diagnostics_tables

Syntax

```
--diagnostics_tables {filename|directory}
```

Parameters

- `filename`  The diagnostic messages are stored in the specified file.
- `directory` The diagnostic messages are stored in a file (filename extension `.i`) which is stored in the specified directory.

For information about specifying a filename or directory, see *Specifying parameters*, page 40.

Description

Use this option to list all possible diagnostic messages in a named file. This can be very convenient, for example, if you used a `#pragma` directive to suppress or change the severity level of any diagnostic messages, but forgot to document why.

This option cannot be given together with other options.

Example

To output a list of all possible diagnostic messages to the file `diag.txt`, use:

```
--diagnostics_tables diag
```

This option is not available in the IDE.

--dir_first

Syntax

```
--dir_first
```

Description

Use this option to make directive names (without a trailing colon) that start in the first column to be recognized as directives.

The default behavior of the assembler is to treat all identifiers starting in the first column as labels.

`Project>Options>Assembler >Language>Allow directives in first column`
Description of assembler options

--error_limit

Syntax

--error_limit=n

Parameters

n  The number of errors before the assembler stops the
assembly. n must be a positive integer; 0 indicates no limit.

Description

Use this option to specify the number of errors allowed before the assembler stops. By
default, 100 errors are allowed.

This option is not available in the IDE.

-f

Syntax

-f filename

Parameters

filename  The commands that you want to extend the command line
with are read from the specified file. Notice that there must
be a space between the option itself and the filename.

For information about specifying a filename, see Specifying parameters, page 40.

Example

To run the assembler with further options taken from the file extend.xcl, use:

iasmrh850 prog -f extend.xcl

See also

Extended command line file, page 40.

To set this option, use:

Project>Options>Assembler>Extra Options
Assembler options

--f

**Syntax**

```
--f filename
```

**Parameters**

*filename* The commands that you want to extend the command line with are read from the specified file. Notice that there must be a space between the option itself and the filename.

For information about specifying a filename, see *Specifying parameters*, page 40.

**Description**

Use this option to make the assembler read command line options from the named file, with the default filename extension xcl.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.

If you use the assembler option --dependencies, extended command line files specified using --f will generate a dependency, but those specified using -f will not generate a dependency.

**See also**

--dependencies, page 46 and -f, page 50.

To set this option, use **Project>Options>Assembler>Extra Options**.

--fpu

**Syntax**

```
--fpu={single|double}
```

**Parameters**

*single* Sets the predefined symbol __FPU__ to __FPU_SINGLE__.

*double* Sets the predefined symbol __FPU__ to __FPU_DOUBLE__.

**Description**

Use this option to define the symbol __FPU__. If this option is not specified, the predefined symbol __FPU__ is defined as __FPU_NONE__. 
Description of assembler options

See also

Predefined symbols, page 23.

To set related options, select:

Project>Options>General Options>Target>FPU

--header_context

Syntax

--header_context

Description

Occasionally, you must know which header file that was included from what source line, to find the cause of a problem. Use this option to list, for each diagnostic message, not only the source position of the problem, but also the entire include stack at that point.

This option is not available in the IDE.

-I

Syntax

-I path

Parameters

path The search path for include files.

Description

Use this option to specify paths to be used by the preprocessor. This option can be used more than once on the command line.

By default, the assembler searches for include files in the current working directory, in the system header directories, and in the paths specified in the $ASM86RH850_INC environment variable. The -I option allows you to give the assembler the names of directories which it will also search if it fails to find the file in the current working directory.

Example

For example, using the options:

-If:\global\ -If:\thisproj\headers\ 
and then writing:

#include "asmlib.hdr"

in the source code, make the assembler search first in the current directory, then in the directory c:\global\, and then in the directory C:\thisproj\headers\. Finally,
the assembler searches the directories specified in the IASMBH850_INC environment variable, provided that this variable is set, and in the system header directories.

**Project>Options>Assembler>Preprocessor>Additional include directories**

- **l**

  **Syntax**
  
  `-l[a][d][e][m][o][x][N][H] {filename|directory}

  **Parameters**
  
  a
  
  Assembled lines only.
  
  d
  
  The LSTOUT directive controls if lines are written to the list file or not. Using `-ld` turns the start value for this to off.
  
  e
  
  No macro expansions.
  
  m
  
  Macro definitions.
  
  o
  
  Multiline code.
  
  x
  
  Includes cross-references.
  
  N
  
  Do not include diagnostics.
  
  H
  
  Includes header file source lines.
  
  filename
  
  The output is stored in the specified file.
  
  directory
  
  The output is stored in a file (filename extension `i`) which is stored in the specified directory.

  For information about specifying a filename or directory, see *Specifying parameters*, page 40.

  **Description**
  
  By default, the assembler does not generate a listing. Use this option to generate a listing to a file.

  **Example**
  
  To generate a listing to the file `list.lst`, use:
  
  `iasm sourcefile -l list`

  To set related options, select:
  
  **Project>Options>Assembler >List**
Description of assembler options

-\texttt{M}

\textbf{Syntax}\hspace{1cm} \texttt{-Mab}

\textbf{Parameters}\hspace{1cm} \textit{ab} \hspace{1cm} The characters to be used as left and right quotes of each macro argument, respectively.

\textbf{Description}\hspace{1cm} Use this option to sets the characters to be used as left and right quotes of each macro argument to \textit{a} and \textit{b} respectively.

By default, the characters are < and >. The \texttt{-M} option allows you to change the quote characters to suit an alternative convention or simply to allow a macro argument to contain < or > themselves.

\textbf{Example}\hspace{1cm} For example, using the option:

\texttt{-M[]}

in the source you would write, for example:

\begin{verbatim}
print [>]
\end{verbatim}

to call a macro \verb|print| with \texttt{>} as the argument.

Note: Depending on your host environment, it might be necessary to use quote marks with the macro quote characters, for example:

\begin{verbatim}
iasmrh850 filename -M'<>'
\end{verbatim}

---

\textbf{--macro\_positions\_in\_diagnostics}

\textbf{Syntax}\hspace{1cm} \texttt{--macro\_positions\_in\_diagnostics}

\textbf{Description}\hspace{1cm} Use this option to obtain position references inside macros in diagnostic messages. This is useful for detecting incorrect source code constructs in macros.

To set this option, use \texttt{Project>Options>Assembler>Extra Options}.
--mnem_first

Syntax
--mnem_first

Description
Use this option to make mnemonics names (without a trailing colon) starting in the first column be recognized as mnemonics.

The default behavior of the assembler is to treat all identifiers starting in the first column as labels.

Project>Options>Assembler>Language>Allow mnemonics in first column

--no_bom

Syntax
--no_bom

Description
Use this option to omit the Byte Order Mark (BOM) when generating a UTF-8 output file.

See also
--text_out, page 60. For more information about encodings, see the IAR C/C++ Development Guide for RH850.

Project>Options>Assembler>Encodings>Text output file encoding

--no_path_in_file_macros

Syntax
--no_path_in_file_macros

Description
Use this option to exclude the path from the return value of the predefined preprocessor symbols __FILE__ and __BASE_FILE__.

This option is not available in the IDE.
Description of assembler options

--no_system_include
Syntax: --no_system_include
Description: By default, the assembler automatically locates the system include files. Use this option to disable the automatic search for system include files. In this case, you might need to set up the search path by using the -I assembler option.

---
Project>Options>Assembler>Preprocessor>Ignore standard include directories

--no_warnings
Syntax: --no_warnings
Description: By default, the assembler issues standard warning messages. Use this option to disable all warning messages.

This option is not available in the IDE.

--no_wrap_diagnostics
Syntax: --no_wrap_diagnostics
Description: By default, long lines in assembler diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.

This option is not available in the IDE.

--only_stdout
Syntax: --only_stdout
Description: Use this option to make the assembler direct messages to stdout instead of to stderr.

This option is not available in the IDE.
**--output, -o**

**Syntax**

```plaintext
--output {filename|directory}
o {filename|directory}
```

**Parameters**

- `filename` The object code is stored in the specified file.
- `directory` The object code is stored in a file (filename extension `.o`) which is stored in the specified directory.

For information about specifying a filename or directory, see [Specifying parameters](#), page 40.

**Description**

By default, the object code produced by the assembler is located in a file with the same name as the source file, but with the extension `.o`. Use this option to specify a different output filename for the object code output.

[Project>Options>General Options>Output>Output directories>Object files]

**--predef_macros**

**Syntax**

```plaintext
--predef_macros {filename|directory}
```

**Parameters**

- `filename` The list of predefined macros is stored in the specified file.
- `directory` The list of predefined macros is stored in a file (filename extension `predef`) which is stored in the specified directory.

For information about specifying a filename or directory, see [Specifying parameters](#), page 40.

**Description**

Use this option to list the predefined symbols. When using this option, make sure to also use the same options as for the rest of your project.

Note that this option requires that you specify a source file on the command line.

This option is not available in the IDE.
**--preinclude**

Syntax  
--preinclude includefile

Parameters  
includefile The header file to be included.

Description  
Use this option to make the assembler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.

To set this option, use:

*Project>Options>Assembler>Preprocessor>Preinclude file*

**--preprocess**

Syntax  
--preprocess=[c][n][s] {filename|directory}

Parameters  
No parameter A preprocessed file.

c Preserves C and C++ style comments that otherwise are removed by the preprocessor. Assembler style comments are always preserved.

n Preprocess only.

s Suppress #line directives.

filename The output is stored in the specified file.

directory The output is stored in a file (filename extension .i) which is stored in the specified directory. The filename is the same as the name of the assembled source file.

For information about specifying a filename or directory, see *Specifying parameters*, page 40.

Description  
Use this option to direct preprocessor output to a named file.
Example
To store the assembler output with preserved comments to the file output.i, use:
iasmrh850 sourcefile --preprocess=c output

**--remarks**

**Syntax**
--remarks

**Description**
Use this option to make the assembler generate remarks, which is the least severe type of diagnostic message and which indicates a source code construct that might cause strange behavior in the generated code. By default, remarks are not generated.

**See also**
Severity levels, page 131.

**--silent**

**Syntax**
--silent

**Description**
By default, the assembler sends various minor messages via the standard output stream. Use this option to make the assembler operate without sending any messages to the standard output stream.

The assembler sends error and warning messages to the error output stream, so they are displayed regardless of this setting.

This option is not available in the IDE.

**--source_encoding**

**Syntax**
--source_encoding {locale|utf8}

**Parameters**
locale The default source encoding is the system locale encoding.
utf8 The default source encoding is the UTF-8 encoding.
Description of assembler options

Description

When reading a source file with no Byte Order Mark (BOM), use this option to specify the encoding.

If this option is not specified and the source file does not have a BOM, the Raw encoding will be used.

See also

For more information about encodings, see the IAR C/C++ Development Guide for RH850.

Project>Options>Assembler>Encodings>Default source file encoding

--system_include_dir

Syntax

--system_include_dir path

Parameters

path

The path to the system include files.

Description

By default, the assembler automatically locates the system include files. Use this option to explicitly specify a different path to the system include files. This might be useful if you have not installed IAR Embedded Workbench in the default location.

This option is not available in the IDE.

--text_out

Syntax

--text_out {utf8|utf16le|utf16be|locale}

Parameters

utf8

Uses the UTF-8 encoding

utf16le

Uses the UTF-16 little-endian encoding

utf16be

Uses the UTF-16 big-endian encoding

locale

Uses the system locale encoding

Description

Use this option to specify the encoding to be used when generating a text output file.

The default for the assembler list files is to use the same encoding as the main source file. The default for all other text files is UTF-8 with a Byte Order Mark (BOM).

If you want text output in UTF-8 encoding without a BOM, use the option --no_bom.
See also  --no_bom, page 55. For more information about encodings, see the IAR C/C++ Development Guide for RH850.

---

**--use_unix_directory_separators**

**Syntax**  
--use_unix_directory_separators

**Description**  
Use this option to make DWARF debug information use / (instead of \) as directory separators in file paths.

This option can be useful if you have a debugger that requires directory separators in UNIX style.

To set this option, use Project>Options>Assembler>Extra Options.

---

**--utf8_text_in**

**Syntax**  
--utf8_text_in

**Description**  
Use this option to specify that the assembler shall use UTF-8 encoding when reading a text input file with no Byte Order Mark (BOM).

**Note:** This option does not apply to source files.

See also  The IAR C/C++ Development Guide for RH850 for more information about encodings.

---

**--warnings_affect_exit_code**

**Syntax**  
--warnings_affect_exit_code

**Description**  
By default, the exit code is not affected by warnings, only errors produce a non-zero exit code. Use this option to make warnings generate a non-zero exit code.

This option is not available in the IDE.
Description of assembler options

**--warnings_are_errors**

**Syntax**

```
--warnings_are_errors
```

**Description**

Use this option to make the assembler treat all warnings as errors. If the assembler encounters an error, no object code is generated.

If you want to keep some warnings, use this option in combination with the option **--diag_warning**. First make all warnings become treated as errors and then reset the ones that should still be treated as warnings, for example:

```
--diag_warning=As001
```

**See also**


Project>Options>Assembler >Diagnostics>Treat all warnings as errors
Assembler operators

● Precedence of assembler operators
● Summary of assembler operators
● Description of assembler operators

Precedence of assembler operators
Each operator has a precedence number assigned to it that determines the order in which the operator and its operands are evaluated. The precedence numbers range from 1 (the highest precedence, that is, first evaluated) to 15 (the lowest precedence, that is, last evaluated).

These rules determine how expressions are evaluated:
● The highest precedence operators are evaluated first, then the second highest precedence operators, and so on until the lowest precedence operators are evaluated.
● Operators of equal precedence are evaluated from left to right in the expression.
● Parentheses ( and ) can be used for grouping operators and operands and for controlling the order in which the expressions are evaluated. For example, this expression evaluates to 1:

\[ 7/((1+(2*3)) \]

Summary of assembler operators
The following tables give a summary of the operators, in order of precedence. Synonyms, where available, are shown after the operator name.

PARENTHESIS OPERATOR
Precedence: 1

( ) Parenthesis.
FUNCTION OPERATORS
Precedence: 2

BYTE1          First byte.
BYTE2          Second byte.
BYTE3          Third byte.
BYTE4          Fourth byte.
DATE           Current date/time.
HI1            High word.
HIGH           High byte.
HWRD           High word.
LOW            Low byte.
LW1            Low word.
LWRD           Low word.
RELEP          Returns signed offset of value to the EP register.
RELGP          Returns signed offset of value to the GP register.
RELTP          Returns signed offset of value to the TP register.
SFB            Section begin.
SFE            Section end.
SIZEOF         Section size.
UPPER          Third byte.

UNARY OPERATORS
Precedence: 3

+              Unary plus.
BINNOT [-]     Bitwise NOT.
NOT [!]        Logical NOT.
-              Unary minus.
MULTIPLICATIVE ARITHMETIC OPERATORS
Precedence: 4

* Multiplication.
/ Division.
MOD [%] Modulo.

ADDITIVE ARITHMETIC OPERATORS
Precedence: 5

+ Addition.
− Subtraction.

SHIFT OPERATORS
Precedence: 6

SHL [<<] Logical shift left.
SHR [>>] Logical shift right.

COMPARISON OPERATORS
Precedence: 7

GE [>=] Greater than or equal.
GT [>] Greater than.
LE [<=] Less than or equal.
LT [<] Less than.
UGT Unsigned greater than.
ULT Unsigned less than.

EQUIVALENCE OPERATORS
Precedence: 8
Description of assembler operators

EQ [=] [==] Equal.
NE [<>] [!=] Not equal.

LOGICAL OPERATORS
Precedence: 9–14
BINAND [&] Bitwise AND (9).
BINXOR [^] Bitwise exclusive OR (10).
BINOR [||] Bitwise OR (11).
AND [&&] Logical AND (12).
XOR Logical exclusive OR (13).
OR [||] Logical OR (14).

CONDITIONAL OPERATOR
Precedence: 15
? : Conditional operator.

Description of assembler operators
This section gives detailed descriptions of each assembler operator.
See also Expressions, operands, and operators, page 20.

() Parenthesis
Precedence 1
Description ( and ) group expressions to be evaluated separately, overriding the default precedence order.
Example 1+2*3 \rightarrow 7
(1+2)*3 \rightarrow 9
* **Multiplication**

Precedence 4

Description * produces the product of its two operands. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

Example

\[
\begin{align*}
2 \times 2 & \rightarrow 4 \\
-2 \times 2 & \rightarrow -4 \\
\end{align*}
\]

**Unary plus**

Precedence 3

Description Unary plus operator; performs nothing.

Example

\[
\begin{align*}
+3 & \rightarrow 3 \\
3^+2 & \rightarrow 6 \\
\end{align*}
\]

**Addition**

Precedence 5

Description The + addition operator produces the sum of the two operands which surround it. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

Example

\[
\begin{align*}
92 + 19 & \rightarrow 111 \\
-2 + 2 & \rightarrow 0 \\
-2 + -2 & \rightarrow -4 \\
\end{align*}
\]

**Unary minus**

Precedence 3

Description The unary minus operator performs arithmetic negation on its operand.

The operand is interpreted as a 32-bit signed integer and the result of the operator is the two’s complement negation of that integer.

Example

\[
\begin{align*}
-3 & \rightarrow -3 \\
3^--2 & \rightarrow -6 \\
4^--5 & \rightarrow 9 \\
\end{align*}
\]
Description of assembler operators

– Subtraction

Precedence 5

Description The subtraction operator produces the difference when the right operand is taken away from the left operand. The operands are taken as signed 32-bit integers and the result is also signed 32-bit integer.

Example
92 - 19 -> 73
-2 - 2 -> -4
-2 -- 2 -> 0

/ Division

Precedence 4

Description / produces the integer quotient of the left operand divided by the right operator. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

Example
9/2 -> 4
-12/3 -> -4
9/2 * 6 -> 24

? : Conditional operator

Syntax condition ? expr : expr

Precedence 15

Description ? results in the first expr if condition evaluates to true and the second expr if condition evaluates to false.

Note: The question mark and a following label must be separated by space or a tab, otherwise the ? is considered the first character of the label.

Example
5 ? 6 : 7 -> 6
0 ? 6 : 7 -> 7
< Less than
Precedence 7
Description < or LT evaluates to 1 (true) if the left operand has a lower numeric value than the right operand, otherwise it is 0 (false).
Example
-1 < 2 -> 1
2 < 1 -> 0
2 < 2 -> 0

<= Less than or equal
Precedence 7
Description <= or LE evaluates to 1 (true) if the left operand has a numeric value that is lower than or equal to the right operand, otherwise it is 0 (false).
Example
1 <= 2 -> 1
2 <= 1 -> 0
1 <= 1 -> 1

<>, != Not equal
Precedence 8
Description <>, !, or NE evaluates to 0 (false) if its two operands are identical in value or to 1 (true) if its two operands are not identical in value.
Example
1 <> 2 -> 1
2 <> 2 -> 0
'A' <> 'B' -> 1

=, == Equal
Precedence 8
Description =, ==, or EQ evaluates to 1 (true) if its two operands are identical in value, or to 0 (false) if its two operands are not identical in value.
Description of assembler operators

Example
1 = 2 \rightarrow 0
2 == 2 \rightarrow 1
'ABC' = 'ABCD' \rightarrow 0

> Greater than

Precedence 7

Description > or GT evaluates to 1 (true) if the left operand has a higher numeric value than the right operand, otherwise it is 0 (false).

Example
-1 > 1 \rightarrow 0
2 > 1 \rightarrow 1
1 > 1 \rightarrow 0

>= Greater than or equal

Precedence 7

Description >= or GE evaluates to 1 (true) if the left operand is equal to or has a higher numeric value than the right operand, otherwise it is 0 (false).

Example
1 >= 2 \rightarrow 0
2 >= 1 \rightarrow 1
1 >= 1 \rightarrow 1

&& Logical AND

Precedence 12

Description && or AND performs logical AND between its two integer operands. If both operands are non-zero the result is 1 (true), otherwise it is 0 (false).

Example
1010B && 0011B \rightarrow 1
1010B && 0101B \rightarrow 1
1010B && 0000B \rightarrow 0
& Bitwise AND

Precedence 9

Description
& or BINAND performs bitwise AND between the integer operands. Each bit in the 32-bit result is the logical AND of the corresponding bits in the operands.

Example
1010B & 0011B -> 0010B
1010B & 0101B -> 0000B
1010B & 0000B -> 0000B

~ Bitwise NOT

Precedence 3

Description
~ or BINNOT performs bitwise NOT on its operand. Each bit in the 32-bit result is the complement of the corresponding bit in the operand.

Example
~ 1010B -> 11111111111111111111111111110101B

| Bitwise OR

Precedence 11

Description
| or BINOR performs bitwise OR on its operands. Each bit in the 32-bit result is the inclusive OR of the corresponding bits in the operands.

Example
1010B | 0101B -> 1111B
1010B | 0000B -> 1010B

^ Bitwise exclusive OR

Precedence 10

Description
^ or BINXOR performs bitwise XOR on its operands. Each bit in the 32-bit result is the exclusive OR of the corresponding bits in the operands.

Example
1010B ^ 0101B -> 1111B
1010B ^ 0011B -> 1001B
% Modulo

Precedence 4

Description % or MOD produces the remainder from the integer division of the left operand by the right operand. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

\[ X \% Y \text{ is equivalent to } X - Y \times (X/Y) \text{ using integer division.} \]

Example

\[
\begin{align*}
2 \% 2 & \rightarrow 0 \\
12 \% 7 & \rightarrow 5 \\
3 \% 2 & \rightarrow 1
\end{align*}
\]

! Logical NOT

Precedence 3

Description ! or NOT negates a logical argument.

Example

\[
\begin{align*}
! 0101B & \rightarrow 0 \\
! 0000B & \rightarrow 1
\end{align*}
\]

|| Logical OR

Precedence 14

Description || or OR performs a logical OR between two integer operands.

Example

\[
\begin{align*}
1010B \text{ || } 0000B & \rightarrow 1 \\
0000B \text{ || } 0000B & \rightarrow 0
\end{align*}
\]

<< Logical shift left

Precedence 6

Description << or SHL shifts the left operand, which is always treated as unsigned, to the left. The number of bits to shift is specified by the right operand, interpreted as an integer value between 0 and 32.
Example:
- $00011100B \ll 3 \rightarrow 11100000B$
- $0000011111111111B \ll 5 \rightarrow 11111111111100000B$
- $14 \ll 1 \rightarrow 28$

**>> Logical shift right**

Precedence: 6

Description: $\gg$ or $\text{SHR}$ shifts the left operand, which is always treated as unsigned, to the right. The number of bits to shift is specified by the right operand, interpreted as an integer value between 0 and 32.

Example:
- $01110000B \gg 3 \rightarrow 00001110B$
- $1111111111111111B \gg 20 \rightarrow 0$
- $14 \gg 1 \rightarrow 7$

**BYTE1 First byte**

Precedence: 2

Description: $\text{BYTE1}$ takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the unsigned, 8-bit integer value of the lower order byte of the operand.

Example:
- $\text{BYTE1} \ 0xABCD \rightarrow 0xCD$

**BYTE2 Second byte**

Precedence: 2

Description: $\text{BYTE2}$ takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the middle-low byte (bits 15 to 8) of the operand.

Example:
- $\text{BYTE2} \ 0x12345678 \rightarrow 0x56$
Description of assembler operators

**BYTE3 Third byte**

- **Precedence**: 2
- **Description**: `BYTE3` takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the middle-high byte (bits 23 to 16) of the operand.
- **Example**: `BYTE3 0x12345678` → `0x34`

**BYTE4 Fourth byte**

- **Precedence**: 2
- **Description**: `BYTE4` takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the high byte (bits 31 to 24) of the operand.
- **Example**: `BYTE4 0x12345678` → `0x12`

**DATE Current time/date**

- **Precedence**: 2
- **Description**: `DATE` gets the time when the current assembly began. The `DATE` operator takes an absolute argument (expression) and returns:
  - `DATE 1`: Current second (0–59).
  - `DATE 2`: Current minute (0–59).
  - `DATE 3`: Current hour (0–23).
  - `DATE 4`: Current day (1–31).
  - `DATE 5`: Current month (1–12).
- **Example**: To specify the date of assembly:
  `today: DC8 DATE 5, DATE 4, DATE 3`
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>HI1 High word</td>
<td>HI1 gets the 16-bit high part of an address to be used by the MOVHI instruction in conjunction with MOVEA or register-indirect operands. These are the only situations where HI1 is used.</td>
<td>MOVHI HI1(my_symbol), reg1, reg2</td>
</tr>
<tr>
<td>HIGH High byte</td>
<td>HIGH takes a single operand to its right which is interpreted as an unsigned, 16-bit integer value. The result is the unsigned 8-bit integer value of the higher order byte of the operand.</td>
<td>HIGH 0xABCD -&gt; 0xAB</td>
</tr>
<tr>
<td>HWRD High word</td>
<td>HWRD takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the high word (bits 31 to 16) of the operand.</td>
<td>HWRD 0x12345678 -&gt; 0x1234</td>
</tr>
<tr>
<td>LOW Low byte</td>
<td>LOW takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the unsigned, 8-bit integer value of the lower order byte of the operand.</td>
<td>LOW 0xABCD -&gt; 0xCD</td>
</tr>
</tbody>
</table>
**LWI Low word**

**Precedence**
2

**Description**
LWI gets the signed low 16-bit part of an address to be used either by either a MOVEA instruction or a register-indirect operand in conjunction with the MOVHI instruction. These are the only situations where LW1 is used.

**Example**
MOVEA LW1(my_symbol), reg1, reg2
LD.B LW1 my_symbol[reg1], reg2

**LWRD Low word**

**Precedence**
2

**Description**
LWRD takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the low word (bits 15 to 0) of the operand.

**Example**
LWRD 0x12345678 → 0x5678

**RELEP**

**Precedence**
2

**Description**
RELEP takes a 32-bit value/symbol and returns the signed offset to the EP register.

**Example**
LD.B RELEP(my_symbol)[EP], reg1

**RELGP**

**Precedence**
2

**Description**
RELGP takes a 32-bit value/symbol and returns the signed offset to the GP register.

**Example**
LD.B RELGP(my_symbol)[GP], reg1
Assembler operators

RELTP

Precedence 2

Description RELTP takes a 32-bit value/symbol and returns the signed offset to the TP register.

Example LD.B RELTP(my_symbol)[TP],reg1

SFB section begin

Syntax SFB(section [(+|-)offset])

Precedence 2

Parameters

section The name of a section, which must be defined before SFB is used.

offset An optional offset from the start address. The parentheses are optional if offset is omitted.

Description SFB accepts a single operand to its right. The operator evaluates to the absolute address of the first byte of that section. This evaluation occurs at linking time.

Example

name sectionBegin
section MYCODE:CODE(2) ; Forward declaration of MYCODE.
section MYCONST:CONST(2)
start dc32 sfb(MYCODE)
end

Even if this code is linked with many other modules, start is still set to the address of the first byte of the section.

SFE section end

Syntax SFE (section [+ | -] offset)

Precedence 2

Parameters

section The name of a section, which must be defined before SFE is used.
Description of assembler operators

**SFE**

SFE accepts a single operand to its right. The operator evaluates to the address of the first byte after the section end. This evaluation occurs at linking time.

**Example**

```assembly
name    sectionEnd
section MYCODE:CODE(2)  ; Forward declaration of MYCODE.
section MYCONST:CONST(2)
end     dc32    sfe(MYCODE)
end
```

Even if this code is linked with many other modules, end is still set to the first byte after the section MYCODE.

The size of the section MYCODE can be achieved by using the **SIZEOF** operator.

---

**SIZEOF** section size

**Syntax**

`SIZEOF section`

**Precedence**

2

**Parameters**

`section` The name of a relocatable section, which must be defined before **SIZEOF** is used.

**Description**

**SIZEOF** generates **SFE**-**SFB** for its argument. That is, it calculates the size in bytes of a section. This is done when modules are linked together.

**Example**

These two files set size to the size of the section MYCODE.

Table.s:

```assembly
module  table
section MYCODE:CODE  ; Forward declaration of MYCODE.
section SEGTAB:CONST(2)
data
size    dc32    sizeof(MYCODE)
end
```

---

IAR Assembler Reference Guide

for RH850

78
Application.s:

```assembly
module  application
section MYCODE:CODE(2)

code

nop                  ; Placeholder for application.

end
```

**UGT Unsigned greater than**

- **Precedence**: 7
- **Description**: UGT evaluates to 1 (true) if the left operand has a larger value than the right operand, otherwise it is 0 (false). The operation treats the operands as unsigned values.
- **Example**: 2 UGT 1 -> 1
  -1 UGT 1 -> 1

**ULT Unsigned less than**

- **Precedence**: 7
- **Description**: ULT evaluates to 1 (true) if the left operand has a smaller value than the right operand, otherwise it is 0 (false). The operation treats the operands as unsigned values.
- **Example**: 1 ULT 2 -> 1
  -1 ULT 2 -> 0

**UPPER Third byte**

- **Precedence**: 2
- **Description**: UPPER takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the middle-high byte (bits 23 to 16) of the operand.
- **Example**: UPPER 0x12345678 -> 0x34
**XOR Logical exclusive OR**

**Precedence**  13

**Description**  XOR evaluates to 1 (true) if either the left operand or the right operand is non-zero, but to 0 (false) if both operands are zero or both are non-zero. Use XOR to perform logical XOR on its two operands.

**Example**

```
0101B XOR 1010B -> 0
0101B XOR 0000B -> 1
```
Assembler directives

This chapter gives a summary of the assembler directives and provides detailed reference information for each category of directives.

Summary of assembler directives

The assembler directives are classified into these groups according to their function:

- Module control directives, page 85
- Symbol control directives, page 87
- Section control directives, page 89
- Value assignment directives, page 93
- Conditional assembly directives, page 95
- Macro processing directives, page 96
- Listing control directives, page 104
- C-style preprocessor directives, page 108
- Data definition or allocation directives, page 113
- Assembler control directives, page 116
- Function directives, page 118
- Call frame information directives for names blocks, page 119.
- Call frame information directives for common blocks, page 120
- Call frame information directives for data blocks, page 121
- Call frame information directives for tracking resources and CFAs, page 122
- Call frame information directives for stack usage analysis, page 125

This table gives a summary of all the assembler directives:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>_args</td>
<td>Is set to number of arguments passed to macro.</td>
<td>Macro processing</td>
</tr>
<tr>
<td>#define</td>
<td>Assigns a value to a label.</td>
<td>C-style preprocessor</td>
</tr>
<tr>
<td>#elif</td>
<td>Introduces a new condition in an #if...#endif block.</td>
<td>C-style preprocessor</td>
</tr>
<tr>
<td>#else</td>
<td>Assembles instructions if a condition is false.</td>
<td>C-style preprocessor</td>
</tr>
<tr>
<td>#endif</td>
<td>Ends an #if, #ifdef, or #ifndef block.</td>
<td>C-style preprocessor</td>
</tr>
</tbody>
</table>

Table 13: Assembler directives summary
### Summary of assembler directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#error</code></td>
<td>Generates an error.</td>
<td>C-style preprocessor</td>
</tr>
<tr>
<td><code>#if</code></td>
<td>Assembles instructions if a condition is true.</td>
<td>C-style preprocessor</td>
</tr>
<tr>
<td><code>#ifdef</code></td>
<td>Assembles instructions if a symbol is defined.</td>
<td>C-style preprocessor</td>
</tr>
<tr>
<td><code>#ifndef</code></td>
<td>Assembles instructions if a symbol is undefined.</td>
<td>C-style preprocessor</td>
</tr>
<tr>
<td><code>#include</code></td>
<td>Includes a file.</td>
<td>C-style preprocessor</td>
</tr>
<tr>
<td><code>#line</code></td>
<td>Changes the line numbers.</td>
<td>C-style preprocessor</td>
</tr>
<tr>
<td><code>#pragma</code></td>
<td>Controls extension features.</td>
<td>C-style preprocessor</td>
</tr>
<tr>
<td><code>#undef</code></td>
<td>Undefines a label.</td>
<td>C-style preprocessor</td>
</tr>
<tr>
<td><code>/*comment*/</code></td>
<td>C-style comment delimiter.</td>
<td>Assembler control</td>
</tr>
<tr>
<td><code>//</code></td>
<td>C++ style comment delimiter.</td>
<td>Assembler control</td>
</tr>
<tr>
<td><code>=</code></td>
<td>Assigns a permanent value local to a module.</td>
<td>Value assignment</td>
</tr>
<tr>
<td><code>ALIGN</code></td>
<td>Aligns the program location counter by inserting zero-filled bytes.</td>
<td>Section control</td>
</tr>
<tr>
<td><code>ALIGNRAM</code></td>
<td>Aligns the program location counter.</td>
<td>Section control</td>
</tr>
<tr>
<td><code>ASSIGN</code></td>
<td>Assigns a temporary value.</td>
<td>Value assignment</td>
</tr>
<tr>
<td><code>CASEOFF</code></td>
<td>Disables case sensitivity.</td>
<td>Assembler control</td>
</tr>
<tr>
<td><code>CASEON</code></td>
<td>Enables case sensitivity.</td>
<td>Assembler control</td>
</tr>
<tr>
<td><code>CFI</code></td>
<td>Specifies call frame information.</td>
<td>Call frame information</td>
</tr>
<tr>
<td><code>DCB</code></td>
<td>Generates 8-bit constants, including strings.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td><code>DC16</code></td>
<td>Generates 16-bit constants.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td><code>DC24</code></td>
<td>Generates 24-bit constants.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td><code>DC32</code></td>
<td>Generates 32-bit constants.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td><code>DC64</code></td>
<td>Generates 64-bit constants.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td><code>DEFINE</code></td>
<td>Defines a file-wide value.</td>
<td>Value assignment</td>
</tr>
<tr>
<td><code>DF32</code></td>
<td>Generates 32-bit floating-point constants.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td><code>DF64</code></td>
<td>Generates 64-bit floating-point constants.</td>
<td>Data definition or allocation</td>
</tr>
</tbody>
</table>

*Table 13: Assembler directives summary* (Continued)
<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQ15</td>
<td>Generates 16-bit fractional constants.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DQ31</td>
<td>Generates 32-bit fractional constants.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DS</td>
<td>Allocates space for 8-bit integers.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DS8</td>
<td>Allocates space for 8-bit integers.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DS16</td>
<td>Allocates space for 16-bit integers.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DS24</td>
<td>Allocates space for 24-bit integers.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DS32</td>
<td>Allocates space for 32-bit integers.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DS64</td>
<td>Allocates space for 64-bit integers.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>ELSE</td>
<td>Assembles instructions if a condition is false.</td>
<td>Conditional assembly</td>
</tr>
<tr>
<td>ELSEIF</td>
<td>Specifies a new condition in an IF...ENDIF block.</td>
<td>Conditional assembly</td>
</tr>
<tr>
<td>END</td>
<td>Ends the assembly of the last module in a file.</td>
<td>Module control</td>
</tr>
<tr>
<td>ENDR</td>
<td>Ends a repeat structure.</td>
<td>Macro processing</td>
</tr>
<tr>
<td>EVEN</td>
<td>Aligns the program counter to an even address.</td>
<td>Section control</td>
</tr>
<tr>
<td>EXITM</td>
<td>Exits prematurely from a macro.</td>
<td>Macro processing</td>
</tr>
<tr>
<td>EXTERN</td>
<td>Imports an external symbol.</td>
<td>Symbol control</td>
</tr>
<tr>
<td>EXTWEAK</td>
<td>Imports an external symbol (which can be undefined).</td>
<td>Symbol control</td>
</tr>
<tr>
<td>IF</td>
<td>Assembles instructions if a condition is true.</td>
<td>Conditional assembly</td>
</tr>
<tr>
<td>IMPORT</td>
<td>Imports an external symbol.</td>
<td>Symbol control</td>
</tr>
<tr>
<td>LIBRARY</td>
<td>Begins a module; an alias for PROGRAM and NAME.</td>
<td>Module control</td>
</tr>
</tbody>
</table>

Table 13: Assembler directives summary (Continued)
### Summary of assembler directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCAL</td>
<td>Creates symbols local to a macro.</td>
<td>Macro processing</td>
</tr>
<tr>
<td>LSTCND</td>
<td>Controls conditional assembler listing.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTCOD</td>
<td>Controls multi-line code listing.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTEXP</td>
<td>Controls the listing of macro generated lines.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTMAC</td>
<td>Controls the listing of macro definitions.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTOUT</td>
<td>Controls assembler-listing output.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTPAG</td>
<td>Retained for backward compatibility reasons; recognized but ignored.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTREP</td>
<td>Controls the listing of lines generated by repeat directives.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTXRF</td>
<td>Generates a cross-reference table.</td>
<td>Listing control</td>
</tr>
<tr>
<td>MACRO</td>
<td>Defines a macro.</td>
<td>Macro processing</td>
</tr>
<tr>
<td>MODULE</td>
<td>Begins a module; an alias for PROGRAM and NAME.</td>
<td>Module control</td>
</tr>
<tr>
<td>NAME</td>
<td>Begins a program module.</td>
<td>Module control</td>
</tr>
<tr>
<td>ODD</td>
<td>Aligns the program location counter to an odd address.</td>
<td>Section control</td>
</tr>
<tr>
<td>OVERLAY</td>
<td>Recognized but ignored.</td>
<td>Symbol control</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>Begins a module.</td>
<td>Module control</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>Exports symbols to other modules.</td>
<td>Symbol control</td>
</tr>
<tr>
<td>PUBWEAK</td>
<td>Exports symbols to other modules, multiple definitions allowed.</td>
<td>Symbol control</td>
</tr>
<tr>
<td>RADIX</td>
<td>Sets the default base.</td>
<td>Assembler control</td>
</tr>
<tr>
<td>REPT</td>
<td>Assembles instructions a specified number of times.</td>
<td>Macro processing</td>
</tr>
<tr>
<td>REPTC</td>
<td>Repeats and substitutes characters.</td>
<td>Macro processing</td>
</tr>
<tr>
<td>REPTI</td>
<td>Repeats and substitutes strings.</td>
<td>Macro processing</td>
</tr>
<tr>
<td>REQUIRE</td>
<td>Forces a symbol to be referenced.</td>
<td>Symbol control</td>
</tr>
<tr>
<td>RSEG</td>
<td>Begins a section.</td>
<td>Section control</td>
</tr>
<tr>
<td>RTMODEL</td>
<td>Declares runtime model attributes.</td>
<td>Module control</td>
</tr>
<tr>
<td>SECTION</td>
<td>Begins a section.</td>
<td>Section control</td>
</tr>
<tr>
<td>SECTION_TYPE</td>
<td>Sets ELF type and flags for a section.</td>
<td>Section control</td>
</tr>
<tr>
<td>SET</td>
<td>Assigns a temporary value.</td>
<td>Value assignment</td>
</tr>
</tbody>
</table>

*Table 13: Assembler directives summary (Continued)*
Assembler directives

Module control directives

Syntax

END

NAME symbol

PROGRAM symbol

RTMODEL key, value

Parameters

key A text string specifying the key.
symbol Name assigned to module.
value A text string specifying the value.

Description

Module control directives are used for marking the beginning and end of source program modules, and for assigning names to them. For information about the restrictions that apply when using a directive in an expression, see Expression restrictions, page 26.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Expression restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>END</td>
<td>Ends the assembly of the last module in a file.</td>
<td>Only locally defined labels or integer constants</td>
</tr>
<tr>
<td>NAME</td>
<td>Begins a module; alias to PROGRAM.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>Begins a module.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>RTMODEL</td>
<td>Declares runtime model attributes.</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Table 14: Module control directives
Beginning a program module

Use NAME or PROGRAM to begin a program module, and to assign a name for future reference by the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian.

Program modules are unconditionally linked by XLINK, even if other modules do not reference them.

Beginning a module

Use any of the directives NAME or PROGRAM to begin an ELF module, and to assign a name.

A module is included in the linked application, even if other modules do not reference them. For more information about how modules are included in the linked application, read about the linking process in the IAR C/C++ Development Guide for RH850.

Note: There can be only one module in a file.

Terminating the source file

Use END to indicate the end of the source file. Any lines after the END directive are ignored. The END directive also ends the module in the file.

 Declaring runtime model attributes

Use RTMODEL to enforce consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key value, or the special value *. Using the special value * is equivalent to not defining the attribute at all. It can however be useful to explicitly state that the module can handle any runtime model.

A module can have several runtime model definitions.

Note: The compiler runtime model attributes start with double underscores. In order to avoid confusion, this style must not be used in the user-defined assembler attributes.

If you are writing assembler routines for use with C or C++ code, and you want to control the module consistency, refer to the IAR C/C++ Development Guide for RH850.

The following examples defines three modules in one source file each, where:

- MOD_1 and MOD_2 cannot be linked together since they have different values for runtime model CAN.
- MOD_1 and MOD_3 can be linked together since they have the same definition of runtime model RTOS and no conflict in the definition of CAN.
- MOD_2 and MOD_3 can be linked together since they have no runtime model conflicts. The value * matches any runtime model value.
Assembler directives

Assembler source file f1.s:

module  mod_1
  rtmodel "CAN", "ISO11519"
  rtmodel "Platform", "M7"
  ; ...
end

Assembler source file f2.s:

module  mod_2
  rtmodel "CAN", "ISO11898"
  rtmodel "Platform", "*
  ; ...
end

Assembler source file f3.s:

module  mod_3
  rtmodel "Platform", "M7"
  ; ...
end

Symbol control directives

Syntax

EXTERN symbol [,symbol] ...
EXTWEAK symbol [,symbol] ...
IMPORT symbol [,symbol] ...
PUBLIC symbol [,symbol] ...
PUBWEAK symbol [,symbol] ...
REQUIRE symbol

Parameters

label Label to be used as an alias for a C/C++ symbol.
symbol Symbol to be imported or exported.
Description of assembler directives

Description

These directives control how symbols are shared between modules:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTERN, IMPORT</td>
<td>Imports an external symbol.</td>
</tr>
<tr>
<td>EXTWEEK</td>
<td>Imports an external symbol. The symbol can be undefined.</td>
</tr>
<tr>
<td>OVERLAY</td>
<td>Recognized but ignored.</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>Exports symbols to other modules.</td>
</tr>
<tr>
<td>PUBWEAK</td>
<td>Exports symbols to other modules, multiple definitions allowed.</td>
</tr>
<tr>
<td>REQUIRE</td>
<td>Forces a symbol to be referenced.</td>
</tr>
</tbody>
</table>

Table 15: Symbol control directives

Exporting symbols to other modules

Use **PUBLIC** to make one or more symbols available to other modules. Symbols defined **PUBLIC** can be relocatable or absolute, and can also be used in expressions (with the same rules as for other symbols).

The **PUBLIC** directive always exports full 32-bit values, which makes it feasible to use global 32-bit constants also in assemblers for 8-bit and 16-bit processors. With the **LOW**, **HIGH**, **>>,** and **<<** operators, any part of such a constant can be loaded in an 8-bit or 16-bit register or word.

There can be any number of **PUBLIC**-defined symbols in a module.

Exporting symbols with multiple definitions to other modules

**PUBWEAK** is similar to **PUBLIC** except that it allows the same symbol to be defined in more than one module. Only one of those definitions is used by ILINK. If a module containing a **PUBLIC** definition of a symbol is linked with one or more modules containing **PUBWEAK** definitions of the same symbol, ILINK uses the **PUBLIC** definition.

Note: Library modules are only linked if a reference to a symbol in that module is made, and that symbol was not already linked. During the module selection phase, no distinction is made between **PUBLIC** and **PUBWEAK** definitions. This means that to ensure that the module containing the **PUBLIC** definition is selected, you should link it before the other modules, or make sure that a reference is made to some other **PUBLIC** symbol in that module.

Importing symbols

Use **EXTERN** or **IMPORT** to import an untyped external symbol.

The **REQUIRE** directive marks a symbol as referenced. This is useful if the section containing the symbol must be loaded even if the code is not referenced.
Example

The following example defines a subroutine to print an error message, and exports the entry address `err` so that it can be called from other modules.

Because the message is enclosed in double quotes, the string will be followed by a zero byte.

It defines `print` as an external routine; the address is resolved at link time.

```assembly
name errorMessage
section .text:CODE(2)
extern print
public err
code

err:    br          print
dc8     "** Error **"
end
```

Section control directives

Syntax

- **ALIGN**: `align [,value]`
- **ALIGNRAM**: `align`
- **ASEGN**: `section [:type] [:flag] [,address]`
- **EVEN**: `[value]`
- **ODD**: `[value]`
- **RSEG**: `section [:type] [:flag] [[align]]`
- **SECTION**: `segment :type [:flag] [[align]]`
- **SECTION_TYPE**: `type-expr {,flags-expr}`

Parameters

- **align**: The power of two to which the address should be aligned. The default align value is 0, except for code sections where the default is 1.
Description of assembler directives

flag

ROOT, NOROOT

ROOT (the default mode) indicates that the section fragment must not be discarded.

NOROOT means that the section fragment is discarded by the linker if no symbols in this section fragment are referred to. Normally, all section fragments except startup code and interrupt vectors should set this flag.

REORDER, NOREORDER

NOREORDER (the default mode) starts a new fragment in the section with the given name, or a new section if no such section exists.

REORDER starts a new section with the given name.

section

The name of the section. The section name is a user-defined symbol that follows the rules described in Symbols, page 22.

type

The memory type, which can be either CODE, CONST, or DATA.

value

Byte value used for padding, default is zero.

type-expr

A constant expression identifying the ELF type of the section.

flags-expr

A constant expression identifying the ELF flags of the section.

Description

The section directives control how code and data are located. For information about the restrictions that apply when using a directive in an expression, see Expression restrictions, page 26.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Expression restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGN</td>
<td>Aligns the program location counter by inserting zero-filled bytes.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>ALIGNRAM</td>
<td>Aligns the program location counter.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>ASEGN</td>
<td>Begins a named absolute segment.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>EVEN</td>
<td>Aligns the program counter to an even address.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>ODD</td>
<td>Aligns the program counter to an odd address.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>RSEG</td>
<td>Begins an ELF section; alias to SECTION.</td>
<td>No external references Absolute</td>
</tr>
</tbody>
</table>

Table 16: Section control directives
Assembler directives

Beginning a named absolute section

Use ASEGN to start a named absolute section located at the address `address`.

Beginning a relocatable section

Use SECTION (or RSEG) to start a new section. The assembler maintains separate location counters (initially set to zero) for all sections, which makes it possible to switch sections and mode anytime without having to save the current program location counter.

Note: The first instance of a SECTION or RSEG directive must not be preceded by any code generating directives, such as DC8 or DS8, or by any assembler instructions.

To set the ELF type, and possibly the ELF flags for the newly created section, use SECTION_TYPE. By default, the values of the flags are zero. For information about valid values, refer to the ELF documentation.

In the following example, the data following the first SECTION directive is placed in a relocatable section called TABLE.

The code following the second SECTION directive is placed in a relocatable section called CODE:

```assembly
module calculator
extern operator
extern addOperator, subOperator

section TABLE:CONST(8)
data
operatorTable:
dc8 addOperator, subOperator
```
section CODE:CODE(2)
code

calculator:    ld.w   operator[r0],r1
              ld.bu   0[r1],r1
              ld.bu   operatorTable[r0],r7
              cmp     r1,r7
              be      add
              ld.bu   operatorTable+1[r0],r7
              cmp     r1,r7
              be      sub

;...

dispose     0,(lp),[lp]

add:        ;...

dispose     0,(lp),[lp]

sub:        ;...

dispose     0,(lp),[lp]
end

Aligning a section

Use ALIGN to align the program location counter to a specified address boundary. You do this by specifying an expression for the power of two to which the program counter should be aligned. That is, a value of 1 aligns to an even address and a value of 2 aligns to an address evenly divisible by 4.

The alignment is made relative to the section start; normally this means that the section alignment must be at least as large as that of the alignment directive to give the desired result.

ALIGN aligns by inserting zero/filled bytes, up to a maximum of 255. The EVEN directive aligns the program counter to an even address (which is equivalent to ALIGN 1) and the ODD directive aligns the program location counter to an odd address. The value used for padding bytes must be within the range 0 to 255.

Use ALIGNRAM to align the program location counter by incrementing it; no data is generated. The parameter align can be within the range 0 to 8.
As an example, the operand modifier F: is needed to determine whether
Description of assembler directives

JARL disp22,reg2

or

JARL disp32,reg2

shall be used. For example:

JARL F:max,R2

These directives are used for assigning values to symbols:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=, EQU</td>
<td>Assigns a permanent value local to a module.</td>
</tr>
<tr>
<td>ASSIGN, SET, VAR</td>
<td>Assigns a temporary value.</td>
</tr>
<tr>
<td>DEFINE</td>
<td>Defines a file-wide value.</td>
</tr>
</tbody>
</table>

Table 18: Value assignment directives

Defining a temporary value

Use ASSIGN, SET, or VAR to define a symbol that might be redefined, such as for use with macro variables. Symbols defined with ASSIGN, SET, or VAR cannot be declared PUBLIC.

This example uses SET to redefine the symbol cons in a loop to generate a table of the first 8 powers of 3:

<table>
<thead>
<tr>
<th>name</th>
<th>table</th>
</tr>
</thead>
<tbody>
<tr>
<td>cons</td>
<td>set</td>
</tr>
</tbody>
</table>

; Generate table of powers of 3.

```asm
cr_tabl macro times
   dc32 cons
   cons set cons * 3
   if times > 1
      cr_tabl times - 1
   endif
endm

section .text:CODE(2)
table    cr_tabl 4
end
```

Defining a permanent local value

Use EQU or = to create a local symbol that denotes a number or offset. The symbol is only valid in the module in which it was defined, but can be made available to other modules with a PUBLIC directive (but not with a PUBWEAK directive).
Use `EXTERN` to import symbols from other modules.

**Defining a permanent global value**

Use `DEFINE` to define symbols that should be known to the module containing the directive. After the `DEFINE` directive, the symbol is known.

A symbol which was given a value with `DEFINE` can be made available to modules in other files with the `PUBLIC` directive.

Symbols defined with `DEFINE` cannot be redefined within the same file. Also, the expression assigned to the defined symbol must be constant.

**Conditional assembly directives**

**Syntax**

```
ELSE
ELSEIF condition
ENDIF
IF condition
```

**Parameters**

`condition` One of these:

- An absolute expression
  - The expression must not contain forward or external references, and any non-zero value is considered as true.

- `string1==string2`
  - The condition is true if `string1` and `string2` have the same length and contents.

- `string1!=string2`
  - The condition is true if `string1` and `string2` have different length or contents.

**Description**

Use the `IF`, `ELSE`, `ELSEIF`, and `ENDIF` directives to control the assembly process at assembly time. If the condition following the `IF` directive is not true, the subsequent instructions do not generate any code (that is, it is not assembled or syntax checked) until an `ELSEIF` condition is true or `ELSE` or `ENDIF` directive is found.

Use `ELSEIF` to introduce a new condition after an `IF` directive. Conditional assembly directives can be used anywhere in an assembly, but have their greatest use in conjunction with macro processing.
All assembler directives (except for END) as well as the inclusion of files can be disabled by the conditional directives. Each IF directive must be terminated by an ENDIF directive. The ELSE and ELSEIF directives are optional, and if used, they must be inside an IF...ENDIF block. IF...ENDIF and IF...ELSE...ENDIF blocks can be nested to any level.

Example

This example uses a macro to add a constant to a direct page memory location:

```assembly
addMem      macro   loc,val         ; loc is a direct page memory
            ; location, and val is an
            ; 8-bit value to add to that
            ; location.
if      val = 0
        ; Do nothing.
elseif  val < 16
        mov     loc,r6
        ld.w   0[r6],r7
        add    val,r7
        st.w   r7,0[r6]
else
        mov     loc,r6
        ld.w   0[r6],r7
        addi   val,r7,r7
        st.w   r7,0[r6]
endif
endm
module  addWithMacro
section CODE:CODE
code
addSome     addMem  0xa0,0          ; Add 0 to memory loc. 0xa0.
addMem  0xa0,1          ; Add 1 to the same address.
addMem  0xa0,2          ; Add 1 to the same address.
addMem  0xa0,3          ; Add 3 to the same address.
addMem  0xa0,47         ; Add 47 to the same address.
jmp     [lp]
end
```

Macro processing directives

Syntax

```assembly
_macro   _args
ENDM
ENDR
```
EXITM
LOCAL symbol [,symbol] ...
name MACRO [argument] [,argument] ...
REPT expr
REPTC formal,actual
REPTI formal,actual [,actual] ...

Parameters
actual Strings to be substituted.
argument Symbolic argument names.
expr An expression.
formal An argument into which each character of actual(REPTC) or each string of actual(REPTI) is substituted.
name The name of the macro.
symbol Symbols to be local to the macro.

Description
These directives allow user macros to be defined. For information about the restrictions that apply when using a directive in an expression, see Expression restrictions, page 26.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Expression restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>_args</td>
<td>Is set to number of arguments passed to macro.</td>
<td>No forward references</td>
</tr>
<tr>
<td>ENDM</td>
<td>Ends a macro definition.</td>
<td>No external references</td>
</tr>
<tr>
<td>ENDR</td>
<td>Ends a repeat structure.</td>
<td>Absolute</td>
</tr>
<tr>
<td>EXITM</td>
<td>Exits prematurely from a macro.</td>
<td>Fixed</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Creates symbols local to a macro.</td>
<td></td>
</tr>
<tr>
<td>MACRO</td>
<td>Defines a macro.</td>
<td></td>
</tr>
<tr>
<td>REPT</td>
<td>Assembles instructions a specified number of times.</td>
<td></td>
</tr>
<tr>
<td>REPTC</td>
<td>Repeats and substitutes characters.</td>
<td></td>
</tr>
<tr>
<td>REPTI</td>
<td>Repeats and substitutes text.</td>
<td></td>
</tr>
</tbody>
</table>

Table 19: Macro processing directives
A macro is a user-defined symbol that represents a block of one or more assembler source lines. Once you have defined a macro, you can use it in your program like an assembler directive or assembler mnemonic.

When the assembler encounters a macro, it looks up the macro’s definition, and inserts the lines that the macro represents as if they were included in the source file at that position.

Macros perform simple text substitution effectively, and you can control what they substitute by supplying parameters to them.

The macro process consists of three distinct phases:

1. The assembler scans and saves macro definitions. The text between MACRO and ENDM is saved but not syntax checked.
2. A macro call forces the assembler to invoke the macro processor (expander). The macro expander switches (if not already in a macro) the assembler input stream from a source file to the output from the macro expander. The macro expander takes its input from the requested macro definition.
   The macro expander has no knowledge of assembler symbols since it only deals with text substitutions at source level. Before a line from the called macro definition is handed over to the assembler, the expander scans the line for all occurrences of symbolic macro arguments, and replaces them with their expansion arguments.
3. The expanded line is then processed as any other assembler source line. The input stream to the assembler continues to be the output from the macro processor, until all lines of the current macro definition have been read.

**Defining a macro**

You define a macro with the statement:

```
name    MACRO [argument] [,argument] ...
```

Here `name` is the name you are going to use for the macro, and `argument` is an argument for values that you want to pass to the macro when it is expanded.

For example, you could define a macro `errMac` as follows:

```
name    errMacro
section .text:CODE(2)
errMac   macro   text
extern  abort
jarl abort,lp
dc8     text,0
even     endm
```
errMac 'Disk not ready'
end

The assembler expands this to:

name errMacro
section .text:CODE(2)
errMac 'Disk not ready'
extern abort
jarl abort,lp
dc8 'Disk not ready', 0

even
csect

end

If you omit a list of one or more arguments, the arguments you supply when calling the macro are called \1 to \9 and \A to \Z.

The previous example could therefore be written as follows:

name errMacro
section .text:CODE(2)
errMac macro text
extern abort
jarl abort,lp
dc8 \1, 0

even
csect
end

Use the EXITM directive to generate a premature exit from a macro.

EXITM is not allowed inside REPT...ENDR, REPTC...ENDR, or REPTI...ENDR blocks.

Use LOCAL to create symbols local to a macro. The LOCAL directive must be used before the symbol is used.

Each time that a macro is expanded, new instances of local symbols are created by the LOCAL directive. Therefore, it is legal to use local symbols in recursive macros.

Note: It is illegal to redefine a macro.

Passing special characters
Macro arguments that include commas or white space can be forced to be interpreted as one argument by using the matching quote characters < and > in the macro call.
For example:

```
name ldrMacro
section .text:CODE(2)

ldrMac macro op
    mov    op
endm

ldrMac <0x19a0,r1>
end
```

The macro can be called using the macro quote characters:

```
ldaMac <0x19a0,X>
```

You can redefine the macro quote characters with the `-M` command line option; see `-M`, page 54.

**Predefined macro symbols**

The symbol `_args` is set to the number of arguments passed to the macro. This example shows how `_args` can be used:

```
fill        macro
    if      _args == 2
        rept    \2
        dc8     \1
        endr
    else
        dc8     \1
        endif
endm

module filler
section .text:CODE(2)
fill 3
fill 4, 3
end
```
It generates this code:

```assembly
module filler
section .text:CODE(2)
fill 3
if _args == 2
else
dc8 3
endif
fill 4, 3
if _args == 2
rept 3
dc8 4
dc8 4
dc8 4
endr
else
endif
en

Repeating statements

Use the REPT...ENDR structure to assemble the same block of instructions several times. If expr evaluates to 0 nothing is generated.

Use REPTC to assemble a block of instructions once for each character in a string. If the string contains a comma it should be enclosed in quotation marks.

Only double quotes have a special meaning and their only use is to enclose the characters to iterate over. Single quotes have no special meaning and are treated as any ordinary character.

Use REPTI to assemble a block of instructions once for each string in a series of strings. Strings containing commas should be enclosed in quotation marks.

This example assembles a series of calls to a subroutine plot to plot each character in a string:

```
name    reptc
extern  plotc
section .text:CODE(2)

banner:    reptc   chr,"Welcome"
mov     'chr',r6
jarl    plotc,lp
endr
end
```
This produces this code:

```
13                                name    reptc
14    000000                      extern  plotc
15 000000                       section .text:CODE(2)
16
17 000000              banner: reptc chr,"Welcome"
17.1 000000 0626               mov    'W',R6
17.2 000006 ........     jarl    plotc,lp
17.3 00000A 0626        mov    'e',R6
17.4 000010 ........     jarl    plotc,lp
17.5 000014 0626   mov    'l',R6
17.6 00001A ........     jarl    plotc,lp
17.7 00001E 0626    mov    'c',R6
17.8 000024 ........     jarl    plotc,lp
17.9 000028 0626   mov    'o',R6
17.10 00002E ........    jarl    plotc,lp
17.11 000032 0626      mov    'm',R6
17.12 000038 ........    jarl    plotc,lp
17.13 00003C 0626    mov    'e',R6
17.14 000042 ........     jarl    plotc,lp
17.15 000046    endr
21 000046                      end
```

This example uses **REPTI** to clear several memory locations:

```
name    repti
extern  base, count, init
section .text:CODE(2)

banner:    repti    addr, base, count, init
st.w    r0,addr[r0]
endr
endr
```

**IAR Assembler Reference Guide**

for RH850

102
Assembler directives

This produces this code:

```
13                           name    repti
14    000000                 extern  base, count, init
15    000000                 section .text:CODE(2)
16
17    000000         banner: repti   addr, base, count, init
17.1  000000 0760 ....       st.w    r0,base[r0]
17.2  000004 0760 ....       st.w    r0,count[r0]
17.3  000008 0760 ....       st.w    r0,init[r0]
17.4  00000C                 endr
20    00000C                 end
```

Coding inline for efficiency

In time-critical code it is often desirable to code routines inline to avoid the overhead of a subroutine call and return. Macros provide a convenient way of doing this.

This example outputs bytes from a buffer to a port:

```
name    ioBufferSubroutine
public  copyBuffer
p0          equ     0xFFC10000  ; Define of the port p0
; data register.

section RAM_BLOCK:DATA
buffer      ds8     256

section .text:CODE(2)
copyBuffer: mov     p0,r6
             mov     buffer,r7
             mov     256,r8      ; Initialize the counter.
loop:       ld.b    0[r7],r9
             add     1,r7
             st.b    r9,0[r8]
             loop    r8,loop     ; Loop 256 to copy the buffer.
             jmp     [lp]
end
```

The main program calls this routine as follows:

doCopy      jarl    copyBuffer,lp
```
```
Description of assembler directives

section RAM_BLOCK:DATA
buffer   ds8   256

copyBuffer: macro
local   loop
mov     p0,r6
mov     buffer,r7
mov     256,r8 ; Initialize the counter.
loop:   ld.b    0[r7],r9
add     1,r7
st.b    r9,0[r8]
loop    r8,loop ; Loop 256 to copy the buffer.
jmp     [lp]
endm

section .text:CODE(2)
copyBuffer
end

Notice the use of the LOCAL directive to make the label loop local to the macro; otherwise an error is generated if the macro is used twice, as the loop label already exists.

Listing control directives

Syntax
LSTCND{+|-}
LSTCOD{+|-}
LSTEXP{+|-}
LSTMAC{+|-}
LSTOUT{+|-}
LSTREP{+|-}
LSTXRF{+|-}

Parameters

Description
These directives provide control over the assembler list file:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTCND</td>
<td>Controls conditional assembly listing.</td>
</tr>
<tr>
<td>LSTCOD</td>
<td>Controls multi-line code listing.</td>
</tr>
</tbody>
</table>

Table 20: Listing control directives

IAR Assembler Reference Guide
for RH850
Assembler directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTEXP</td>
<td>Controls the listing of macro-generated lines.</td>
</tr>
<tr>
<td>LSTMAC</td>
<td>Controls the listing of macro definitions.</td>
</tr>
<tr>
<td>LSTOUT</td>
<td>Controls assembly-listing output.</td>
</tr>
<tr>
<td>LSTREP</td>
<td>Controls the listing of lines generated by repeat directives.</td>
</tr>
<tr>
<td>LSTXRF</td>
<td>Generates a cross-reference table.</td>
</tr>
</tbody>
</table>

Table 20: Listing control directives (Continued)

Note: The directives COL, LSTPAGE, PAGE, and PAGSIZ are included for backward compatibility reasons; they are recognized but no action is taken.

Turning the listing on or off

Use LSTOUT- to disable all list output except error messages. This directive overrides all other listing control directives.

The default is LSTOUT+, which lists the output (if a list file was specified).

To disable the listing of a debugged section of program:

```
lstout- ; This section has already been debugged.
lstout+ ; This section is currently being debugged.
end
```

Listing conditional code and strings

Use LSTCND+ to force the assembler to list source code only for the parts of the assembly that are not disabled by previous conditional IF statements.

The default setting is LSTCND-, which lists all source lines.

Use LSTCOD+ to list more than one line of code for a source line, if needed; that is, long ASCII strings produce several lines of output.

The default setting is LSTCOD-, which restricts the listing of output code to just the first line of code for a source line.

Using the LSTCND and LSTCOD directives does not affect code generation.
This example shows how \texttt{LSTCND+} hides a call to a subroutine that is disabled by an \texttt{IF} directive:

```assembly
name    lstcndTest
extern  print
section .text:CODE(2)

debug   set     0
begin   if      debug
        jarl    print,lp
endif

lstcnd+
begin2  if      debug
        jarl    print
endif

end
```

This generates the following listing:

```
13                                name
14    000000                      print
15    000000              extern  print
16                                section .text:CODE(2)
17    000000              debug   set     0
18    000000              begin   if      debug
19                                jarl    print,lp
20    000000                      endif
21                                lstcnd+
22    000000              begin2  if      debug
23    000000                      endif
24                                lstcnd+
25    000000                      end
```

### Controlling the listing of macros

Use \texttt{LSTEXP-} to disable the listing of macro-generated lines. The default is \texttt{LSTEXP+}, which lists all macro-generated lines.

Use \texttt{LSTMAC+} to list macro definitions. The default is \texttt{LSTMAC-}, which disables the listing of macro definitions.
This example shows the effect of LSTMAC and LSTEXP:

```
name lstmacTest
extern memLoc
section .text:CODE(2)

dec2:   macro   arg
        mov     arg,r6
        ld.w    0[r6],r7
        add     2,r7
        st.w    r7,0[r6]
        endm
        lstmac+

inc2:   macro   arg
        mov     arg,r6
        ld.w    0[r6],r7
        add     -2,r7
        st.w    r7,0[r6]
        endm
        lstexp-

begin:  dec2    memLoc
        lstexp-
        inc2    memLoc
        jmp     [lp]
        lstexp+

; Restore default values for
; listing control directives.
        lstmac-
        lstexp-

end
```
This produces the following output:

```
1    NAME    dec2
2
3    dec2    MACRO    arg
4    subw    $2,arg
5    ENDM
6
7    000000    LSTMAC+
8
9
10   000000   begin:    dec2    R6
10.1  000000 263A    subw    $2,R6
11
12   000000    LSTEXP-
13   000002    inc2    R7
14
15
16   000000    ; restore defaults
17   000000    LSTMAC-
18   000000    LSTEXP+
19
20   000004    END
```

### Controlling the listing of generated lines

Use `LSTREP-` to turn off the listing of lines generated by the directives `REPT`, `REPTC`, and `REPTI`.

The default is `LSTREP+`, which lists the generated lines.

### Generating a cross-reference table

Use `LSTXRF+` to generate a cross-reference table at the end of the assembler list for the current module. The table shows values and line numbers, and the type of the symbol.

The default is `LSTXRF-`, which does not give a cross-reference table.

### C-style preprocessor directives

**Syntax**

```
#define symbol text
#elif condition
#else
#endif
#error "message"
```
Assembler directives

```c
#if condition
#endif
#ifdef symbol
#endif
#ifndef symbol
#endif
#include "filename" | <filename>
#line line-no "filename"
#undef symbol
```

### Parameters

- **condition**: An absolute assembler expression, see Expressions, operands, and operators, page 20.
  The expression must not contain any assembler labels or symbols, and any non-zero value is considered as true. The C preprocessor operator defined can be used.

- **filename**: Name of file to be included or referred.

- **line-no**: Source line number.

- **message**: Text to be displayed.

- **symbol**: Preprocessor symbol to be defined, undefined, or tested.

- **text**: Value to be assigned.

### Description

The assembler has a C-style preprocessor that follows the C99 standard. These C-language preprocessor directives are available:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#define</code></td>
<td>Assigns a value to a preprocessor symbol.</td>
</tr>
<tr>
<td><code>#elif</code></td>
<td>Introduces a new condition in an <code>#if...#endif</code> block.</td>
</tr>
<tr>
<td><code>#else</code></td>
<td>Assembles instructions if a condition is false.</td>
</tr>
<tr>
<td><code>#endif</code></td>
<td>Ends an <code>#if</code>, <code>#ifdef</code>, or <code>#ifndef</code> block.</td>
</tr>
<tr>
<td><code>#error</code></td>
<td>Generates an error.</td>
</tr>
<tr>
<td><code>#if</code></td>
<td>Assembles instructions if a condition is true.</td>
</tr>
<tr>
<td><code>#ifdef</code></td>
<td>Assembles instructions if a preprocessor symbol is defined.</td>
</tr>
<tr>
<td><code>#ifndef</code></td>
<td>Assembles instructions if a preprocessor symbol is undefined.</td>
</tr>
<tr>
<td><code>#include</code></td>
<td>Includes a file.</td>
</tr>
<tr>
<td><code>#line</code></td>
<td>Changes the source references in the debug information.</td>
</tr>
</tbody>
</table>

*Table 21: C-style preprocessor directives*
You should not mix assembler language and C-style preprocessor directives. Conceptually, they are different languages and mixing them might lead to unexpected behavior because an assembler directive is not necessarily accepted as a part of the C preprocessor language.

Note that the preprocessor directives are processed before other directives. As an example avoid constructs like:

```asm
redef       macro                   ; Avoid the following!
#define \1 \2
endm
```

because the \1 and \2 macro arguments are not available during the preprocessing phase.

### Defining and undefining preprocessor symbols

Use `#define` to define a value of a preprocessor symbol.

```asm
#define symbol value
```

Use `#undef` to undefine a symbol; the effect is as if it had not been defined.

### Conditional preprocessor directives

Use the `#if...#else...#endif` directives to control the assembly process at assembly time. If the condition following the `#if` directive is not true, the subsequent instructions will not generate any code (that is, it will not be assembled or syntax checked) until an `#endif` or `#else` directive is found.

All assembler directives (except for `END`) and file inclusion can be disabled by the conditional directives. Each `#if` directive must be terminated by an `#endif` directive. The `#else` directive is optional and, if used, it must be inside an `#if...#endif` block.

`#if...#endif` and `#if...#else...#endif` blocks can be nested to any level.

Use `#ifdef` to assemble instructions up to the next `#else` or `#endif` directive only if a symbol is defined.

Use `#ifndef` to assemble instructions up to the next `#else` or `#endif` directive only if a symbol is undefined.
This example defines the labels `tweak` and `adjust`. If `adjust` is defined, then the memory location calibration constant is decremented by an amount that depends on `adjust`, in this case 14.

```
module calibrate
extern calibrationConstant
section .text:CODE(2)

#define tweak 1
#define adjust 3

calibrate
  mov calibrationConstant, r6
  ld.w 0[r6], r7
#ifdef tweak
#if adjust==1
  add 4, r7
#elif adjust==2
  add 8, r7
#elif adjust==3
  add 14, r7
#endif
#endif /* ifdef tweak */
  st.w r7, 0[r6]
  jmp [lp]
end
```

Including source files

Use `#include` to insert the contents of a file into the source file at a specified point. The filename can be specified within double quotes or within angle brackets.

Following is the full description of the assembler’s `#include` file search procedure:

- If the name of the `#include` file is an absolute path, that file is opened.
- When the assembler encounters the name of an `#include` file in angle brackets such as:
  ```
  #include <ior7f7010352atp.h>
  ```
  it searches the following directories for the file to include:
  1. The directories specified with the `-I` option, in the order that they were specified.
  2. Any directories specified using the `ARH850_INC` environment variable.
  3. The automatically set up library system include directories. See `--no_system_include`, page 56 and `--system_include_dir`, page 60.
When the assembler encounters the name of an `#include` file in double quotes such as:

```c
#include "vars.h"
```

it searches the directory of the source file in which the `#include` statement occurs, and then performs the same sequence as for angle-bracketed filenames.

If there are nested `#include` files, the assembler starts searching the directory of the file that was last included, iterating upwards for each included file, searching the source file directory last.

Use angle brackets for header files provided with the IAR Assembler for RH850, and double quotes for header files that are part of your application.

This example uses `#include` to include a file defining macros into the source file. For example, these macros could be defined in `Macros.inc`:

```c
add     -4,sp
st.w    \1,0[sp]
mov     \2,\1
ld.w    0[sp],\2
add     4,sp
```

The macro definitions can then be included, using `#include`, as in this example:

**Displaying errors**

Use `#error` to force the assembler to generate an error, such as in a user-defined test.

**Changing the source line numbers**

Use the `#line` directive to change the source line numbers and the source filename used in the debug information. `#line` operates on the lines following the `#line` directive.

**Comments in C-style preprocessor directives**

If you make a comment within a define statement, use:

- the C comment delimiters /* ... */ to comment sections
- the C++ comment delimiter // to mark the rest of the line as comment.

Do not use assembler comments within a define statement as it leads to unexpected behavior.
This expression evaluates to 3 because the comment character is preserved by `#define`:

```
#define x 3    ; This is a misplaced comment.
module misplacedComment1
expression equ x * 8 + 5
;...
end
```

This example illustrates some problems that might occur when assembler comments are used in the C-style preprocessor:

```
#define five 5    ; This comment is not OK.
#define six 6     // This comment is OK.
#define seven 7   /* This comment is OK. */
module misplacedComment2
section MYCONST:CONST(2)

DC32 five, 11, 12
; The previous line expands to:
; "DC32 5    ; This comment is not OK., 11, 12"

DC32 six + seven, 11, 12
; The previous line expands to:
; "DC32 6 + 7, 11, 12"
end
```

**Data definition or allocation directives**

**Syntax**

```
DC8 expr [,expr] ...
DC16 expr [,expr] ...
DC24 expr [,expr] ...
DC32 expr [,expr] ...
DC64 expr [,expr] ...
DF32 value [,value] ...
DF64 value [,value] ...
DQ15 value [,value] ...
DQ31 value [,value] ...
DS.[size] count
DS count
DS8 count
DS16 count
DS24 count
DS32 count
DS64 count
```
Description of assembler directives

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>A valid absolute expression specifying the number of elements to be reserved.</td>
</tr>
<tr>
<td>expr</td>
<td>A valid absolute, relocatable, or external expression, or an ASCII string. ASCII strings are zero filled to a multiple of the data size implied by the directive. Double-quoted strings are zero-terminated. For DC64, expr cannot be relocatable or external.</td>
</tr>
<tr>
<td>value</td>
<td>A valid absolute expression or floating-point constant.</td>
</tr>
</tbody>
</table>

Description

These directives define values or reserve memory.

Use DC8, DC16, DC24, DC32, DC64, DF32, or DF64 to create a constant, which means an area of bytes is reserved big enough for the constant.

Use DS, DS8, DS16, DS24, DS32, or DS64 to reserve a number of uninitialized bytes.

For information about the restrictions that apply when using a directive in an expression, see Expression restrictions, page 26.

The column Alias in the following table shows the Renesas directive that corresponds to the IAR Systems directive.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC8</td>
<td></td>
<td>Generates 8-bit constants, including strings.</td>
</tr>
<tr>
<td>DC16</td>
<td></td>
<td>Generates 16-bit constants.</td>
</tr>
<tr>
<td>DC24</td>
<td></td>
<td>Generates 24-bit constants.</td>
</tr>
<tr>
<td>DC32</td>
<td></td>
<td>Generates 32-bit constants.</td>
</tr>
<tr>
<td>DC64</td>
<td></td>
<td>Generates 64-bit constants.</td>
</tr>
<tr>
<td>DF32</td>
<td></td>
<td>Generates 32-bit floating-point constants.</td>
</tr>
<tr>
<td>DF64</td>
<td></td>
<td>Generates 64-bit floating-point constants.</td>
</tr>
<tr>
<td>DQ15</td>
<td></td>
<td>Generates 16-bit fractional constants.</td>
</tr>
<tr>
<td>DQ31</td>
<td></td>
<td>Generates 32-bit fractional constants.</td>
</tr>
<tr>
<td>DS8</td>
<td>DS</td>
<td>Allocates space for 8-bit integers.</td>
</tr>
<tr>
<td>DS16</td>
<td></td>
<td>Allocates space for 16-bit integers.</td>
</tr>
<tr>
<td>DS24</td>
<td></td>
<td>Allocates space for 24-bit integers.</td>
</tr>
<tr>
<td>DS32</td>
<td></td>
<td>Allocates space for 32-bit integers.</td>
</tr>
<tr>
<td>DS64</td>
<td></td>
<td>Allocates space for 64-bit integers.</td>
</tr>
</tbody>
</table>

Table 22: Data definition or allocation directives
Generating a lookup table

This example generates a constant table of 8-bit data that is accessed via the call instruction and added up to a sum.

```
module sumTableAndIndex
section .text:CONST(2)
table:   dc8  12
        dc8  15
        dc8  17
        dc8  16
        dc8  14
        dc8  13
        dc8  11
        dc8  9
section .text:CODE(2)
count:  set  0
addTable: mov  table,r6
rept   7
if     count == 7
exitm
endif
ld.b  count[r6],r10
set    count + 1
add    1,r10
st.b  r10,count[r6]
endr
jmp      [lp]
end
```

Defining strings

To define a string:

```
myMsg   DC8 'Please enter your name'
```

To define a string which includes a trailing zero:

```
myCstr  DC8 "This is a string."
```

To include a single quote in a string, enter it twice; for example:

```
errMsg  DC8 'Don''t understand!!'
```
Reserving space

To reserve space for 10 bytes:

table DS8 10

Assembler control directives

Syntax

- /*comment*/
- //comment
- CASEOFF
- CASEON
- RADIX expr

Parameters

- comment: Comment ignored by the assembler.
- expr: Default base; default 10 (decimal).

Description

These directives provide control over the operation of the assembler. For information about the restrictions that apply when using a directive in an expression, see Expression restrictions, page 26.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Expression restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>comment</em>/</td>
<td>C-style comment delimiter.</td>
<td></td>
</tr>
<tr>
<td>//</td>
<td>C++ style comment delimiter.</td>
<td></td>
</tr>
<tr>
<td>CASEOFF</td>
<td>Disables case sensitivity.</td>
<td></td>
</tr>
<tr>
<td>CASEON</td>
<td>Enables case sensitivity.</td>
<td></td>
</tr>
<tr>
<td>RADIX expr</td>
<td>Sets the default base on all numeric values.</td>
<td>No forward references</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No external references</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Absolute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fixed</td>
</tr>
</tbody>
</table>

Table 23: Assembler control directives

Use /*...*/ to comment sections of the assembler listing.
Use // to mark the rest of the line as comment.
Use RADIX to set the default base for constants. The default base is 10.
Controlling case sensitivity

Use `CASEON` or `CASEOFF` to turn on or off case sensitivity for user-defined symbols. By default, case sensitivity is on.

When `CASEOFF` is active all symbols are stored in upper case, and all symbols used by ILINK should be written in upper case in the ILINK definition file.

When `CASEOFF` is set, `label` and `LABEL` are identical in this example:

```assembly
module caseSensitivity1
section .text:CODE(2)

caseoff
label   nop               ; Store as "LABEL".
br      LABEL
end
```

The following will generate a duplicate label error:

```assembly
module caseSensitivity2
section .text:CODE(2)
caseoff
label   nop               ; Store as "LABEL".
LABEL   nop               ; Error, "LABEL" already defined.
end
```

Defining comments

This example shows how `/* ... */` can be used for a multi-line comment:

```assembly
/* Program to read serial input. Version 1: 19.2.11 Author: mjwp */
```

See also [C-style preprocessor directives], page 108.
Description of assembler directives

Changing the base

To set the default base to 16:

```assembly
module radix
section .text:CODE(2)
radix 16            ; With the default base set
mov 12,r6           ; to 16, the immediate value
;...                ; of the load instruction is
; interpreted as 0x12.

; To reset the base from 16 to 10 again, the argument must be
; written in hexadecimal format.
radix 0xa           ; Reset the default base to 10.
mov 12,r6           ; Now, the immediate value of
;...                ; the load instruction is
; interpreted as 0x0c.
end
```

Function directives

**Syntax**

`CALL_GRAPH_ROOT function [,.category]`

**Parameters**

- `function` The function, a symbol.
- `category` An optional call graph root category, a string.

**Description**

Use this directive to specify that, for stack usage analysis purposes, the function `function` is a call graph root. You can also specify an optional category, a quoted string.

The compiler will generate this directive in assembler list files, when needed.

**Example**

`CALL_GRAPH_ROOT my_interrupt, "interrupt"

**See also**

*Call frame information directives for stack usage analysis*, page 125, for information about CFI directives required for stack usage analysis.

*IAR C/C++ Development Guide for RH850* for information about how to enable and use stack usage analysis.
Call frame information directives for names blocks

Syntax

Names block directives:

- `CFI NAMES name`
- `CFI ENDNAMES name`
- `CFI RESOURCE resource: bits [, resource : bits]`...
- `CFI VIRTUALRESOURCE resource : bits [, resource : bits]`...
- `CFI RESOURCEPARTS resource part, part [, part]`...
- `CFI STACKFRAME cfa resource type [, cfa resource type]`...
- `CFI BASEADDRESS cfa type [, cfa type]`...

Parameters

- `bits` The size of the resource in bits.
- `cfa` The name of a CFA (canonical frame address).
- `name` The name of the block.
- `namesblock` The name of a previously defined names block.
- `offset` The offset relative the CFA. An integer with an optional sign.
- `part` A part of a composite resource. The name of a previously declared resource.
- `resource` The name of a resource.
- `size` The size of the frame cell in bytes.
- `type` The segment memory type, such as `CODE`, `CONST` or `DATA`. In addition, any of the memory types supported by the IAR ILINK Linker. It is only used for denoting an address space.

Description

Use these directives to define a names block:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CFI BASEADDRESS</code></td>
<td>Declares a base address CFA (Canonical Frame Address).</td>
</tr>
<tr>
<td><code>CFI ENDNAMES</code></td>
<td>Ends a names block.</td>
</tr>
<tr>
<td><code>CFI FRAMECELL</code></td>
<td>Creates a reference into the caller’s frame.</td>
</tr>
<tr>
<td><code>CFI NAMES</code></td>
<td>Starts a names block.</td>
</tr>
<tr>
<td><code>CFI RESOURCE</code></td>
<td>Declares a resource.</td>
</tr>
</tbody>
</table>

Table 24: Call frame information directives names block
Description of assembler directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI RESOURCEPARTS</td>
<td>Declares a composite resource.</td>
</tr>
<tr>
<td>CFI STACKFRAME</td>
<td>Declares a stack frame CFA.</td>
</tr>
<tr>
<td>CFI VIRTUALRESOURCE</td>
<td>Declares a virtual resource.</td>
</tr>
</tbody>
</table>

Table 24: Call frame information directives names block (Continued)

Example

Examples of using CFI directives, page 36

See also

Tracking call frame usage, page 29

Call frame information directives for common blocks

Syntax

Common block directives:

CFI COMMON name USING namesblock
CFI ENDCOMMON name
CFI CODEALIGN codealignfactor
CFI DATAALIGN dataalignfactor
CFI DEFAULT { UNDEFINED | SAMEVALUE }
CFI RETURNADDRESS resource type

Parameters

codealignfactor The smallest common factor of all instruction sizes. Each CFI directive for a data block must be placed according to this alignment. 1 is the default and can always be used, but a larger value reduces the produced call frame information in size. The possible range is 1–256.

commonblock The name of a previously defined common block.

dataalignfactor The smallest common factor of all frame sizes. If the stack grows toward higher addresses, the factor is negative; if it grows toward lower addresses, the factor is positive. 1 is the default, but a larger value reduces the produced call frame information in size. The possible ranges are –256 to –1 and 1 to 256.

name The name of the block.

namesblock The name of a previously defined names block.

resource The name of a resource.
Assembler directives

Description
Use these directives to define a common block:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI CODEALIGN</td>
<td>Declares code alignment.</td>
</tr>
<tr>
<td>CFI COMMON</td>
<td>Starts or extends a common block.</td>
</tr>
<tr>
<td>CFI DATAALIGN</td>
<td>Declares data alignment.</td>
</tr>
<tr>
<td>CFI DEFAULT</td>
<td>Declares the default state of all resources.</td>
</tr>
<tr>
<td>CFI ENDCOMMON</td>
<td>Ends a common block.</td>
</tr>
<tr>
<td>CFI RETURNADDRESS</td>
<td>Declares a return address column.</td>
</tr>
</tbody>
</table>

Table 25: Call frame information directives common block

In addition to these directives you might also need the call frame information directives for specifying rules or CFI expressions for resources and CFAs, see Call frame information directives for tracking resources and CFAs, page 122.

Example
Examples of using CFI directives, page 36

See also
Tracking call frame usage, page 29

Call frame information directives for data blocks

Syntax
CFI BLOCK name USING commonblock
CFI ENDBLOCK name
CFI { NOFUNCTION | FUNCTION label }
CFI { INVALID | VALID }
CFI { REMEMBERSTATE | RESTORESTATE }
CFI PICKER
CFI CONDITIONAL label [, label]...

Parameters
commonblock       The name of a previously defined common block.
label             A function label.
Description of assembler directives

Description

These directives allow call frame information to be defined in the assembler source code:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI BLOCK</td>
<td>Starts a data block.</td>
</tr>
<tr>
<td>CFI CONDITIONAL</td>
<td>Declares a data block to be a conditional thread.</td>
</tr>
<tr>
<td>CFI ENDBLOCK</td>
<td>Ends a data block.</td>
</tr>
<tr>
<td>CFI FUNCTION</td>
<td>Declares a function associated with a data block.</td>
</tr>
<tr>
<td>CFI INVALID</td>
<td>Starts a range of invalid call frame information.</td>
</tr>
<tr>
<td>CFI NOFUNCTION</td>
<td>Declares a data block to not be associated with a function.</td>
</tr>
<tr>
<td>CFI PICKER</td>
<td>Declares a data block to be a picker thread. Used by the compiler for keeping track of execution paths when code is shared within or between functions.</td>
</tr>
<tr>
<td>CFI REMEMBERSTATE</td>
<td>Remembers the call frame information state.</td>
</tr>
<tr>
<td>CFI RESTORESTATE</td>
<td>Restores the saved call frame information state.</td>
</tr>
<tr>
<td>CFI VALID</td>
<td>Ends a range of invalid call frame information.</td>
</tr>
</tbody>
</table>

Table 26: Call frame information directives for data blocks

In addition to these directives you might also need the call frame information directives for specifying rules or CFI expressions for resources and CFAs, see **Call frame information directives for tracking resources and CFAs**, page 122.

Example

Examples of using CFI directives, page 36

See also

**Tracking call frame usage**, page 29

### Call frame information directives for tracking resources and CFAs

**Syntax**

```
CFI cfa { resource | resource + constant | resource - constant }

CFI cfa cfiexpr

CFI resource { UNDEFINED | SAMEVALUE | CONCAT }

CFI resource { resource | FRAME(cfa, offset) }

CFI resource cfiexpr
```

**name**

The name of the block.

**Description**

The name of the block.
Parameters

cfa
The name of a CFA (canonical frame address).
cfiexpr
A CFI expression, which can be one of these:
- A CFI operator with operands
- A numeric constant
- A CFA name
- A resource name.

constant
A constant value or an assembler expression that can be evaluated to a constant value.

offset
The offset relative the CFA. An integer with an optional sign.

resource
The name of a resource.

Unary operators

Overall syntax: \texttt{OPERATOR(operand)}

<table>
<thead>
<tr>
<th>CFI operator</th>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPLEMENT</td>
<td>cfiexpr</td>
<td>Performs a bitwise NOT on a CFI expression.</td>
</tr>
<tr>
<td>LITERAL</td>
<td>expr</td>
<td>Get the value of the assembler expression. This can insert the value of a regular assembler expression into a CFI expression.</td>
</tr>
<tr>
<td>NOT</td>
<td>cfiexpr</td>
<td>Negates a logical CFI expression.</td>
</tr>
<tr>
<td>UMINUS</td>
<td>cfiexpr</td>
<td>Performs arithmetic negation on a CFI expression.</td>
</tr>
</tbody>
</table>

Table 27: Unary operators in CFI expressions

Binary operators

Overall syntax: \texttt{OPERATOR(operand1,operand2)}

<table>
<thead>
<tr>
<th>CFI operator</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>cfiexpr,cfiexpr</td>
<td>Addition</td>
</tr>
<tr>
<td>AND</td>
<td>cfiexpr,cfiexpr</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>DIV</td>
<td>cfiexpr,cfiexpr</td>
<td>Division</td>
</tr>
<tr>
<td>EQ</td>
<td>cfiexpr,cfiexpr</td>
<td>Equal</td>
</tr>
<tr>
<td>GE</td>
<td>cfiexpr,cfiexpr</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>GT</td>
<td>cfiexpr,cfiexpr</td>
<td>Greater than</td>
</tr>
<tr>
<td>LE</td>
<td>cfiexpr,cfiexpr</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>LSHIFT</td>
<td>cfiexpr,cfiexpr</td>
<td>Logical shift left of the left operand. The number of bits to shift is specified by the right operand. The sign bit will not be preserved when shifting.</td>
</tr>
</tbody>
</table>

Table 28: Binary operators in CFI expressions
Description of assembler directives

<table>
<thead>
<tr>
<th>CFI operator</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LT</td>
<td>cfiexpr,cfiexpr</td>
<td>Less than</td>
</tr>
<tr>
<td>MOD</td>
<td>cfiexpr,cfiexpr</td>
<td>Modulo</td>
</tr>
<tr>
<td>MUL</td>
<td>cfiexpr,cfiexpr</td>
<td>Multiplication</td>
</tr>
<tr>
<td>NE</td>
<td>cfiexpr,cfiexpr</td>
<td>Not equal</td>
</tr>
<tr>
<td>OR</td>
<td>cfiexpr,cfiexpr</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>RSHIPTA</td>
<td>cfiexpr,cfiexpr</td>
<td>Arithmetic shift right of the left operand. The number of bits to shift is specified by the right operand. In contrast with RSHIPTL, the sign bit is preserved when shifting.</td>
</tr>
<tr>
<td>RSHIPTL</td>
<td>cfiexpr,cfiexpr</td>
<td>Logical shift right of the left operand. The number of bits to shift is specified by the right operand. The sign bit will not be preserved when shifting.</td>
</tr>
<tr>
<td>SUB</td>
<td>cfiexpr,cfiexpr</td>
<td>Subtraction</td>
</tr>
<tr>
<td>XOR</td>
<td>cfiexpr,cfiexpr</td>
<td>Bitwise XOR</td>
</tr>
</tbody>
</table>

Table 28: Binary operators in CFI expressions (Continued)

Ternary operators

Overall syntax: OPERATOR(operand1, operand2, operand3)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAME</td>
<td>cfa, size, offset</td>
<td>Gets the value from a stack frame. The operands are: cfa, an identifier that denotes a previously declared CFA. size, a constant expression that denotes a size in bytes. offset, a constant expression that denotes a size in bytes. Gets the value at address cfa+offset of size size.</td>
</tr>
<tr>
<td>IF</td>
<td>cond, true, false</td>
<td>Conditional operator. The operands are: cond, a CFI expression that denotes a condition. true, any CFI expression. false, any CFI expression. If the conditional expression is non-zero, the result is the value of the true expression; otherwise the result is the value of the false expression.</td>
</tr>
<tr>
<td>LOAD</td>
<td>size, type, addr</td>
<td>Gets the value from memory. The operands are: size, a constant expression that denotes a size in bytes. type, a memory type. addr, a CFI expression that denotes a memory address. Gets the value at address addr in the segment memory type type of size size.</td>
</tr>
</tbody>
</table>

Table 29: Ternary operators in CFI expressions
**Assemble directives**

Use these directives to track resources and CFAs in common blocks and data blocks:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI cfa</td>
<td>Declares the value of a CFA.</td>
</tr>
<tr>
<td>CFI resource</td>
<td>Declares the value of a resource.</td>
</tr>
</tbody>
</table>

*Table 30: Call frame information directives for tracking resources and CFAs*

**Example**

Examples of using CFI directives, page 36

**See also**

Tracking call frame usage, page 29

---

## Call frame information directives for stack usage analysis

### Syntax

- CFI FUNCALL `{ caller } callee`
- CFI INDIRECTCALL `{ caller }
- CFI NOCALLS `{ caller }
- CFI TAILCALL `{ callee }

### Parameters

- `callee` The label of the called function.
- `caller` The label of the calling function.

### Description

These directives allow call frame information to be defined in the assembler source code:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI FUNCALL</td>
<td>Declares function calls for stack usage analysis.</td>
</tr>
<tr>
<td>CFI INDIRECTCALL</td>
<td>Declares indirect calls for stack usage analysis.</td>
</tr>
<tr>
<td>CFI NOCALLS</td>
<td>Declares absence of calls for stack usage analysis.</td>
</tr>
<tr>
<td>CFI TAILCALL</td>
<td>Declares tail calls for stack usage analysis.</td>
</tr>
</tbody>
</table>

*Table 31: Call frame information directives for stack usage analysis*

**See also**

Tracking call frame usage, page 29

The *IAR C/C++ Development Guide for RH850* for information about stack usage analysis.
Description of assembler directives
Pragma directives

This chapter describes the pragma directives of the IAR Assembler for RH850.

The pragma directives control the behavior of the assembler, for example whether it outputs warning messages. The pragma directives are preprocessed, which means that macros are substituted in a pragma directive.

Summary of pragma directives

This table lists the pragma directives of the assembler:

<table>
<thead>
<tr>
<th>#pragma directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>diag_default</td>
<td>Changes the severity level of diagnostic messages</td>
</tr>
<tr>
<td>diag_error</td>
<td>Changes the severity level of diagnostic messages</td>
</tr>
<tr>
<td>diag_remark</td>
<td>Changes the severity level of diagnostic messages</td>
</tr>
<tr>
<td>diag_suppress</td>
<td>Suppresses diagnostic messages</td>
</tr>
<tr>
<td>diag_warning</td>
<td>Changes the severity level of diagnostic messages</td>
</tr>
<tr>
<td>message</td>
<td>Prints a message</td>
</tr>
</tbody>
</table>

Table 32: Pragma directives summary

Descriptions of pragma directives

The following pages describe each pragma directive.

Note that all pragma directives using = for value assignment should be entered like:

```
#pragma pragmaname=pragmavalue
```

or

```
#pragma pragmaname = pragmavalue
```

diag_default

**Syntax**

```
#pragma diag_default=tag,tag,...
```

**Parameters**

- `tag` The number of a diagnostic message, for example the message number Pe117.
Descriptions of pragma directives

Description
Use this pragma directive to change the severity level back to the default, or to the severity level defined on the command line by any of the options --diag_error, --diag_remark, --diag_suppress, or --diag_warning, for the diagnostic messages specified with the tags.

Example
#pragma diag_default=Pe117

See also
The chapter Diagnostics.

diag_error
Syntax
#pragma diag_error=tag,tag,...

Parameters
tag The number of a diagnostic message, for example the message number Pe117.

Description
Use this pragma directive to change the severity level to error for the specified diagnostic messages.

Example
#pragma diag_error=Pe117

See also
The chapter Diagnostics.

diag_remark
Syntax
#pragma diag_remark=tag,tag,...

Parameters
tag The number of a diagnostic message, for example the message number Pe117.

Description
Use this pragma directive to change the severity level to remark for the specified diagnostic messages.

Example
#pragma diag_remark=Pe177

See also
The chapter Diagnostics.
diag_suppress

Syntax
#pragma diag_suppress=tag,tag,...

Parameters

- **tag**: The number of a diagnostic message, for example the message number Pe117.

Description
Use this pragma directive to suppress the specified diagnostic messages.

Example
#pragma diag_suppress=Pe117,Pe177

See also
The chapter Diagnostics.

diag_warning

Syntax
#pragma diag_warning=tag,tag,...

Parameters

- **tag**: The number of a diagnostic message, for example the message number Pe826.

Description
Use this pragma directive to change the severity level to warning for the specified diagnostic messages.

Example
#pragma diag_warning=Pe826

See also
The chapter Diagnostics.

message

Syntax
#pragma message(string)

Parameters

- **string**: The message that you want to direct to the standard output stream.

Description
Use this pragma directive to make the assembler print a message on stdout when the file is assembled.
Descriptions of pragma directives

Example

```c
#ifdef TESTING
#pragma message("Testing")
#endif
```
Diagnostics

The following pages describe the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.

Message format

All diagnostic messages are issued as complete, self-explanatory messages. A typical diagnostic message from the assembler is produced in the form:

filename, linenumber, level [tag]: message

where filename is the name of the source file in which the error was encountered; linenumber is the line number at which the assembler detected the error; level is the level of seriousness of the diagnostic; tag is a unique tag that identifies the diagnostic message; message is a self-explanatory message, possibly several lines long.

Diagnostic messages are displayed on the screen, and printed in the optional list file. In the IAR Embedded Workbench IDE, diagnostic messages are displayed in the Build messages window.

Severity levels

The diagnostics are divided into different levels of severity:

REMARK
A diagnostic message that is produced when the assembler finds a source code construct that can possibly lead to erroneous behavior in the generated code. Remarks are, by default, not issued but can be enabled, see --remarks, page 59.

WARNING
A diagnostic message that is produced when the assembler finds a programming error or omission which is of concern but not so severe as to prevent the completion of compilation. Warnings can be disabled with the command line option --no_warnings, see --no_warnings, page 56.

ERROR
A diagnostic message that is produced when the assembler finds a construct which clearly violates the language rules, such that code cannot be produced. An error produces a non-zero exit code.
Severity levels

FATAL ERROR
A diagnostic message that is produced when the assembler finds a condition that not only prevents code generation, but which makes further processing of the source code pointless. After the diagnostic is issued, assembly ends. A fatal error produces a non-zero exit code.

SETTING THE SEVERITY LEVEL
The diagnostic messages can be suppressed or the severity level can be changed for all types of diagnostics except for fatal errors and some of the regular errors.

For information about the assembler options that are available for setting severity levels, see Summary of assembler options, page 41.

For information about the pragma directives that are available for setting severity levels, see the chapter Pragma directives.

INTERNAL ERROR
An internal error is a diagnostic message that signals that there was a serious and unexpected failure due to a fault in the assembler. It is produced using this form:

Internal error: message

where message is an explanatory message. If internal errors occur, they should be reported to your software distributor or IAR Systems Technical Support. Please include information enough to reproduce the problem. This would typically include:

- The product name
- The version number of the assembler, which can be seen in the header of the list files generated by the assembler
- Your license number
- The exact internal error message text
- The source file of the program that generated the internal error
- A list of the options that were used when the internal error occurred.
importing ............................................. 88
in relocatable expressions .................. 26
predefined ......................................... 23
assembling, invocation syntax .............. 26
assembly messages format .................... 17
ASSIGN (assembler directive) .............. 94
assumptions (assembler directive) ........ 11

B
bitwise AND (assembler operator) .......... 71
bitwise exclusive OR (assembler operator) 71
bitwise NOT (assembler operator) .......... 71
bold style, in this guide ....................... 13
__BUILD_NUMBER__ (predefined symbol) .... 24
BYTE1 (assembler operator) ................. 73
BYTE2 (assembler operator) ................. 73
BYTE3 (assembler operator) ................. 74
BYTE4 (assembler operator) ................. 74

C
 CALL_GRAPH_ROOT (assembler directive) ... 118
case sensitivity, controlling ................ 43, 117
CASEOFF (assembler directive) ............. 116
CASEON (assembler directive) ............... 116
--case_insensitive (assembler option) ..... 43
CFA, CFI directives for tracking ............ 122
CFI BASEADDRESS (assembler directive) ...119
CFI BLOCK (assembler directive) .......... 122
CFI cfa (assembler directive) .............. 125
CFI CODEALIGN (assembler directive) ..... 121
CFI COMMON (assembler directive) ........ 121
CFI CONDITIONAL (assembler directive) ... 122
CFI DATAALIGN (assembler directive) ..... 121
CFI DEFAULT (assembler directive) ........ 121
CFI directives for common blocks .......... 120
CFI directives for data blocks .............. 121
CFI directives for names blocks ............ 119
CFI directives for stack usage analysis ...... 125
CFI directives for tracking resources and CFAs 122
CFI ENDBLOCK (assembler directive) ...... 122
CFI ENDCOMMON (assembler directive) ..... 121
CFI ENDNAMES (assembler directive) ...... 119
CFI expressions ............................... 35
CFI FRAMECELL (assembler directive) ...... 119
CFI FUNCTION (assembler directive) ........ 125
CFI INDIRECTCALL (assembler directive) .. 125
CFI INVALID (assembler directive) ......... 122
CFI NAMES (assembler directive) .......... 119
CFI NOCALLS (assembler directive) ....... 125
CFI NOFUNCTION (assembler directive) ..... 122
CFI PICKER (assembler directive) .......... 122
CFI REMEMBERSTATE (assembler directive) ...122
CFI RESOURCE (assembler directive) ...... 119
CFI resource (assembler directive) ......... 125
CFI RESOURCEPARTS (assembler directive) ...120
CFI RESTORESTATE (assembler directive) ... 122
CFI RETURNADDRESS (assembler directive) ...121
CFI STACKFRAME (assembler directive) ..... 120
CFI TAILCALL (assembler directive) ....... 125
CFI VALID (assembler directive) .......... 122
CFI VIRTUALRESOURCE (assembler directive) .... 120
code model
identifying ........................................ 24
specifying ........................................ 43
__CODE_MODEL__ (predefined symbol) ....... 24
--code_model (assembler option) .......... 43
COL (assembler directive) ................. 105
command line options
part of invocation syntax .................... 17
typographic convention ...................... 17
command line, extending .................... 50

IAR Assembler Reference Guide
for RH850

AFE1_AFE2-1:1
command prompt icon, in this guide .......................... 13
corrections
  in assembler list file ........................................ 116
  in assembler source code .................................... 19
  in C-style preprocessor directives ......................... 112
  multi-line, using with assembler directives ............ 117
common block (call frame information) ..................... 30
common blocks, CFI directives for .......................... 120
common block, defining ....................................... 32
COMPLEMENT (CFI operator) ................................... 123
computer style, typographic convention .................... 12
conditional assembly directives .............................. 95
See also C-style preprocessor directives
conditional code and strings, listing ........................ 105
constants
  default base of ............................................. 116
  integer ................................................................ 20
conventions, used in this guide ................................. 12
copyright notice .................................................. 2
__CORE__ (predefined symbol) ................................ 24
core
  identifying ..................................................... 24
  selecting (--core) .............................................. 44
  specifying on command line .................................. 44
--core (assembler option) .................................... 44
--core (compiler option) .................................... 44
CRC, in assembler list file ..................................... 28
cross-references, in assembler list file
  generating (LSTXRF) ......................................... 108
  generating (-l) ............................................... 53
current time/date (assembler operator) ....................... 74
C++ terminology .................................................. 12
data blocks, CFI directives for ................................ 121
data definition directives ....................................... 113
data field, in assembler list file ............................... 27
data model
  identifying ..................................................... 24
  specifying ...................................................... 45
__DATA_MODEL__ (predefined symbol) ......................... 24
--data_model (assembler option) ............................. 45
__DATE__ (predefined symbol) .................................. 47
DATE (assembler operator) ................................... 74
DC8 (assembler directive) .................................... 114
DC16 (assembler directive) ................................... 114
DC24 (assembler directive) ................................... 114
DC32 (assembler directive) ................................... 114
DC64 (assembler directive) ................................... 114
--debug (assembler option) ................................... 45
diagnosis messages ............................................... 131
defining a common block ....................................... 32
defining a names block ......................................... 31
--dependencies (assembler option) ............................ 46
DF32 (assembler directive) .................................. 114
DF64 (assembler directive) .................................. 114
classifying as errors .......................................... 47
classifying as remarks ........................................ 47
classifying as warnings ....................................... 48
disabling warnings ............................................. 56
disabling wrapping of .......................................... 56
enabling remarks ............................................... 59
listing all ....................................................... 49
suppressing ...................................................... 48
--diagnostics_tables (assembler option) ..................... 49
diag_default (pragma directive) ............................... 127
--diag_error (assembler option) ............................. 47
diag_error (pragma directive) ................................ 128
--diag_remak (assembler option) ............................. 47
diag_remark (#pragma directive) ................. 128
diag_suppress (assembler option) ............... 48
diag_suppress (pragma directive) ............... 129
diag_warning (assembler option) ............... 48
diag_warning (pragma directive) ............... 129
diag_warning (pragma directive) ............... 129
directives. See assembler directives
--dir_first (assembler option) .................. 49
diagnostic messages .................................. 131
classifying ........................................... 47
classifying . See assembler directives
even (assembler directive) ....................... 90
extended command line file
   for assembler ..................................... 51
extended command line file (extend.xcl) ....... 40, 50
EXTERN (assembler directive) ................. 88
EXTWEAK (assembler directive) ................. 88
FILE .................................................. 2
false value, in assembler expressions ........... 22
fatal error messages ................................ 132
__FILE__ (predefined symbol) ..................... 24
file dependencies, tracking ...................... 46
file extensions. See filename extensions
file types
   extended command line .......................... 40, 50
   #include, specifying path ...................... 52
filename extensions
   xcl ................................................. 40, 50
filenames, specifying for assembler object file . 57
first byte (assembler operator) ................ 73
floating-point constants ......................... 21
floating-point unit
   identifying ....................................... 24
   specifying ....................................... 51
formats
   assembler source code .......................... 19
diagnostic messages ............................... 131
in list files ........................................ 27
fourth byte (assembler operator) ............... 74

E
edition, of this guide ................................ 2
efficient coding techniques ...................... 28
#elif (assembler directive) ...................... 109
#else (assembler directive) ..................... 109
END (assembler directive) ......................... 85
#endif (assembler directive) ..................... 109
ENDM (assembler directive) ....................... 97
ENDR (assembler directive) ....................... 97
environment variables
   assembler ......................................... 18
EQ (CFI operator) .................................. 123
EQU (assembler directive) ....................... 94
equal (assembler operator) ....................... 69
#error (assembler directive) .................... 109
error messages ...................................... 131
#error, using to display ......................... 112
--error_limit (assembler option) .............. 50
EXITM (assembler directive) ..................... 97
experience, programming ......................... 11
expressions ......................................... 20
false value, in assembler expressions ........... 22
fatal error messages ................................ 132
__FILE__ (predefined symbol) ..................... 24
file dependencies, tracking ...................... 46
file extensions. See filename extensions
file types
   extended command line .......................... 40, 50
   #include, specifying path ...................... 52
filename extensions
   xcl ................................................. 40, 50
filenames, specifying for assembler object file . 57
first byte (assembler operator) ................ 73
floating-point constants ......................... 21
floating-point unit
   identifying ....................................... 24
   specifying ....................................... 51
formats
   assembler source code .......................... 19
diagnostic messages ............................... 131
in list files ........................................ 27
fourth byte (assembler operator) ............... 74

F
-f (assembler option) .............................. 40, 50
--f (assembler option) ............................ 51
false value, in assembler expressions ........... 22
fatal error messages ................................ 132
__FILE__ (predefined symbol) ..................... 24
file dependencies, tracking ...................... 46
file extensions. See filename extensions
file types
   extended command line .......................... 40, 50
   #include, specifying path ...................... 52
filename extensions
   xcl ................................................. 40, 50
filenames, specifying for assembler object file . 57
first byte (assembler operator) ................ 73
floating-point constants ......................... 21
floating-point unit
   identifying ....................................... 24
   specifying ....................................... 51
formats
   assembler source code .......................... 19
diagnostic messages ............................... 131
in list files ........................................ 27
fourth byte (assembler operator) ............... 74

IAR Assembler Reference Guide
for RH850

AFE1_AFE2-1:1
<table>
<thead>
<tr>
<th>Symbol/Operator</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FPU</strong> (predefined symbol)</td>
<td>24</td>
</tr>
<tr>
<td>--fpu (assembler option)</td>
<td>51</td>
</tr>
<tr>
<td>FRAME (CFI operator)</td>
<td>124</td>
</tr>
<tr>
<td>function directives</td>
<td>118</td>
</tr>
<tr>
<td>F: (operand modifier)</td>
<td>93</td>
</tr>
<tr>
<td>GE (CFI operator)</td>
<td>123</td>
</tr>
<tr>
<td>greater than or equal (assembler operator)</td>
<td>70</td>
</tr>
<tr>
<td>GT (CFI operator)</td>
<td>123</td>
</tr>
<tr>
<td>header files, SFR</td>
<td>28</td>
</tr>
<tr>
<td>--header_context (assembler option)</td>
<td>52</td>
</tr>
<tr>
<td>high byte (assembler operator)</td>
<td>75</td>
</tr>
<tr>
<td>high word (assembler operator HI1)</td>
<td>75</td>
</tr>
<tr>
<td>HIGH (assembler operator)</td>
<td>75</td>
</tr>
<tr>
<td>HI1 (assembler operator)</td>
<td>75</td>
</tr>
<tr>
<td>HWRD (assembler operator)</td>
<td>75</td>
</tr>
<tr>
<td>-l (assembler option)</td>
<td>53</td>
</tr>
<tr>
<td>IAR Technical Support</td>
<td>132</td>
</tr>
<tr>
<td><strong>IAR_SYSTEMS_ASM</strong> (predefined symbol)</td>
<td>25</td>
</tr>
<tr>
<td>#if (assembler directive)</td>
<td>109</td>
</tr>
<tr>
<td>IF (CFI operator)</td>
<td>124</td>
</tr>
<tr>
<td>#ifdef (assembler directive)</td>
<td>109</td>
</tr>
<tr>
<td>ifndef (assembler directive)</td>
<td>109</td>
</tr>
<tr>
<td>IMPORT (assembler directive)</td>
<td>88</td>
</tr>
<tr>
<td>#include files, specifying</td>
<td>52</td>
</tr>
<tr>
<td>#include (assembler directive)</td>
<td>109</td>
</tr>
<tr>
<td>include paths, specifying</td>
<td>52</td>
</tr>
<tr>
<td>inline coding, using macros</td>
<td>103</td>
</tr>
<tr>
<td>installation directory</td>
<td>12</td>
</tr>
<tr>
<td>integer constants</td>
<td>20</td>
</tr>
<tr>
<td>internal error</td>
<td>132</td>
</tr>
<tr>
<td>invocation syntax</td>
<td>17</td>
</tr>
<tr>
<td>italic style, in this guide</td>
<td>12–13</td>
</tr>
<tr>
<td>-l (assembler option)</td>
<td>53</td>
</tr>
<tr>
<td>labels. See assembler labels</td>
<td>123</td>
</tr>
<tr>
<td>LE (CFI operator)</td>
<td>123</td>
</tr>
<tr>
<td>less than or equal (assembler operator)</td>
<td>69</td>
</tr>
<tr>
<td>LIBRARY (assembler directive)</td>
<td>83</td>
</tr>
<tr>
<td>lightbulb icon, in this guide</td>
<td>13</td>
</tr>
<tr>
<td><strong>LINE</strong> (predefined symbol)</td>
<td>25</td>
</tr>
<tr>
<td>#line (assembler directive)</td>
<td>109</td>
</tr>
<tr>
<td>linker options</td>
<td></td>
</tr>
<tr>
<td>typographic convention</td>
<td>13</td>
</tr>
<tr>
<td>list file format</td>
<td>27</td>
</tr>
<tr>
<td>body</td>
<td>27</td>
</tr>
<tr>
<td>CRC</td>
<td>28</td>
</tr>
<tr>
<td>header</td>
<td>27</td>
</tr>
<tr>
<td>symbol and cross reference list files</td>
<td></td>
</tr>
<tr>
<td>control directives for</td>
<td>104</td>
</tr>
<tr>
<td>generating (-l)</td>
<td>53</td>
</tr>
<tr>
<td>LITERAL (CFI operator)</td>
<td>123</td>
</tr>
<tr>
<td>LOAD (CFI operator)</td>
<td>124</td>
</tr>
<tr>
<td>local value, defining</td>
<td>94</td>
</tr>
<tr>
<td>LOCAL (assembler directive)</td>
<td>97</td>
</tr>
<tr>
<td>logical AND (assembler operator)</td>
<td>70</td>
</tr>
<tr>
<td>logical exclusive OR (assembler operator)</td>
<td>80</td>
</tr>
<tr>
<td>logical NOT (assembler operator)</td>
<td>72</td>
</tr>
<tr>
<td>logical OR (assembler operator)</td>
<td>72</td>
</tr>
<tr>
<td>logical shift left (assembler operator)</td>
<td>72</td>
</tr>
<tr>
<td>logical shift right (assembler operator)</td>
<td>73</td>
</tr>
</tbody>
</table>
low byte (assembler operator) ........................................ 75
low word (assembler operator LW1) ............................ 76
low word (assembler operator) .................................... 76
LOW (assembler operator) ........................................... 75
LSHIFT (CFI operator) ................................................. 123
LSTCND (assembler directive) .................................... 104
LSTCOD (assembler directive) .................................... 104
LSTEXP (assembler directives) ................................... 105
LSTMAC (assembler directive) .................................... 105
LSTOUT (assembler directive) .................................... 105
LSTPAGE (assembler directive) ................................... 105
LSTREP (assembler directive) .................................... 105
LSTXRF (assembler directive) .................................... 105
macro processing directives ....................................... 96
macro quote characters .............................................. 99
specifying ............................................................... 54
MACRO (assembler directive) ..................................... 97
macros. See assembler macros ..................................... 54
--macro_positions_in_diagnostics (compiler option) ........ 54
memory, reserving space in ......................................... 114
message (pragma directive) ......................................... 129
messages, excluding from standard output stream ............. 59
--mmem_first (assembler option) ................................ 55
MOD (CFI operator) .................................................... 124
module consistency .................................................. 86
module control directives .......................................... 85
modules, beginning .................................................. 86
MUL (CFI operator) .................................................... 124
multiplication (assembler operator) ............................ 67
M: (operand modifier) ................................................. 93
NAME (assembler directive) ......................................... 85
names block (call frame information) ......................... 30
names blocks, CFI directives for ................................. 119
names block, defining ............................................... 31
naming conventions ................................................ 13
NE (CFI operator) ..................................................... 124
not equal (assembler operator) ................................... 69
NOT (CFI operator) .................................................. 123
--no_bom (assembler option) ..................................... 55
--no_path_in_file_macros (assembler option) ............... 55
--no_system_include (assembler option) ..................... 56
--no_warnings (assembler option) .............................. 56
--no_wrap_diagnostics (assembler option) ................. 56
-o (assembler option) ............................................... 42
ODD (assembler directive) .......................................... 90
--only_stdout (assembler option) ............................... 56
operand modifiers (for value assignment directives) ....... 93
operands
  format of .......................................................... 19
  in assembler expressions ....................................... 20
operations, format of .............................................. 19
operation, silent .................................................... 59
operators. See assembler operators ............................... 55
option summary ...................................................... 41
OR (CFI operator) .................................................... 124
--output (assembler option) .................................... 57
OVERLAY (assembler directive) ................................. 88
PAGE (assembler directive) ....................................... 105
PAGSIZ (assembler directive) ................................... 105
parameters
  specifying .......................................................... 40
--source_encoding (assembler option) .......................... 59
special function registers ...................................... 28
stack usage analysis, CFI directives for .................. 125
standard error ..................................................... 56
standard output stream, disabling messages to ........... 59
standard output, specifying ..................................... 56
statements, repeating ............................................. 101
stdout, direct messages to ..................................... 56
SUB (CFI operator) .................................................. 124
subtraction (assembler operator) ............................. 68
__SUBVERSION__ (predefined symbol) ......................... 25
Support, Technical ................................................ 132
symbol and cross-reference table, in assembler list file .... 28
See also Include cross-reference
symbol control directives ........................................ 87
symbols
See also assembler symbols
exporting to other modules ..................................... 88
predefined, in assembler ........................................ 23
predefined, in assembler macro ................................ 100
user-defined, case sensitive ................................... 43
--system_include_dir (assembler option) .................... 60

T
Technical Support, IAR ............................................ 132
temporary values, defining .................................... 94
terminology .......................................................... 12
--text_out (assembler option) .................................. 60
third byte (assembler operator) ............................... 74
__TIME__ (predefined symbol) .................................. 25
time-critical code ................................................ 103
tools icon, in this guide ......................................... 13
trademarks .......................................................... 2
true value, in assembler expressions ......................... 22
typographic conventions ......................................... 12

U
UGT (assembler operator) ........................................ 79
ULT (assembler operator) ......................................... 79
UMINUS (CFI operator) ........................................... 123
unary minus (assembler operator) .............................. 67
unary plus (assembler operator) ............................... 67
#undef (assembler directive) ..................................... 110
unsigned greater than (assembler operator) ................ 79
unsigned less than (assembler operator) ..................... 79
UPPER (assembler operator) ..................................... 79
user symbols, case sensitive .................................... 43
--use_unix_directory_separators (compiler option) ......... 61
--utf8_text_in (assembler option) ............................ 61

V
value assignment directives ..................................... 93
values, defining .................................................... 114
VAR (assembler directive) ....................................... 94
__VER__ (predefined symbol) .................................. 25
version
of this guide ....................................................... 2

W
warnings ............................................................. 131
classifying ........................................................ 48
disabling .......................................................... 56
exit code ........................................................... 61
treating as errors ............................................... 62
warnings icon, in this guide ................................... 13
--warnings_affect_exit_code (assembler option) ............. 18, 61
--warnings_are_errors (assembler option) .................... 62

X
xcl (filename extension) ........................................ 40, 50
XOR (assembler operator) ......................................... 80
Symbols

_args (assembler directive) ............................................. 97
_args (predefined macro symbol) .................................. 100
_BUILD_NUMBER_ (predefined symbol) .......................... 24
_CODE_MODEL_ (predefined symbol) ............................... 24
_CORE_ (predefined symbol) ........................................ 24
_DATA_MODEL_ (predefined symbol) ............................... 24
_DATE_ (predefined symbol) ......................................... 24
_FILE_ (predefined symbol) .......................................... 24
_FPU_ (predefined symbol) ........................................... 24
_IAR_SYSTEMS_ASM_ (predefined symbol) ....................... 25
_SUBVERSION_ (predefined symbol) ............................. 25
_TIME_ (predefined symbol) ......................................... 25
_VER_ (predefined symbol) .......................................... 25
-D (assembler option) ................................................ 44
-f (assembler option) ................................................ 40, 50
-I (assembler option) ................................................ 52
-M (assembler option) ................................................ 54
-o (assembler option) ................................................ 42
-r (assembler option) ................................................ 42
-case_insensitive (assembler option) ......................... 43
-code_model (assembler option) ................................. 43
-core (assembler option) ........................................... 44
-core (compiler option) ............................................. 44
-data_model (assembler option) ............................... 45
-debug (assembler option) ......................................... 45
-dependencies (assembler option) ........................... 46
-diagnostics_tables (assembler option) ................. 49
-diag_error (assembler option) .................................. 47
-diag_remark (assembler option) .............................. 47
-diag_suppress (assembler option) ......................... 48
-diag_warning (assembler option) ............................ 48
-dir_first (assembler option) .................................. 49
--error_limit (assembler option) ............................... 50
--f (assembler option) .............................................. 51
--fpu (assembler option) ......................................... 51
--header_context (assembler option) ..................... 52
--macro_positions_in_diagnostics (compiler option) .... 54
--mmem_first (assembler option) ............................. 55
--no_bom (assembler option) ................................... 55
--no_path_in_file_macros (assembler option) .......... 55
--no_system_include (assembler option) ................. 56
--no_warnings (assembler option) ......................... 56
--no_wrap_diagnostics (assembler option) ............ 56
--only_stdout (assembler option) ........................... 56
--output (assembler option) ................................... 57
--predef_macros (assembler option) ....................... 57
--preinclude (assembler option) ............................ 58
--warnings_are_errors (assembler option) ............ 59
--source_encoding (assembler option) ................... 59
--system_include_dir (assembler option) .............. 60
--text_out (assembler option) ................................ 60
--use_unix_directory_separators (compiler option) ... 61
--util8_text_in (assembler option) ....................... 61
--warnings_affect_exit_code (assembler option) .... 18, 61
--warnings_are_errors (assembler option) ............ 62
! (assembler operator) ............................................. 72
!= (assembler operator) .......................................... 69
?: (assembler operator) ........................................... 68
() (assembler operator) ........................................... 66
* (assembler operator) ........................................... 67
/ (assembler operator) ........................................... 68
/"..."/ (assembler directive) .................................. 116
// (assembler directive) ...................................... 116
& (assembler operator) ........................................... 71
&& (assembler operator) .......................................... 70
#define (assembler directive) ................................ 109
#endif (assembler directive) .................................. 109
#define (assembler directive) ................................ 109
#define (assembler directive) ................................ 109
#error (assembler directive) ......................... 109
#if (assembler directive) .......................... 109
#ifdef (assembler directive) ...................... 109
#ifndef (assembler directive) .................... 109
#include files, specifying .......................... 52
#include (assembler directive) .................... 109
#line (assembler directive) ....................... 109
#pragma (assembler directive) ..................... 110
#undef (assembler directive) ........................ 110
^ (assembler operator) ............................. 71
+ (assembler operator) ............................. 67
< (assembler operator) ............................. 69
<= (assembler operator) ............................ 69
<> (assembler operator) ............................. 69
= (assembler directive) ............................ 94
= (assembler operator) ............................. 69
== (assembler operator) ............................ 69
> (assembler operator) ............................. 70
>= (assembler operator) ............................ 70
>> (assembler operator) ............................. 73
| (assembler operator) ............................. 71
|| (assembler operator) ............................ 72
~ (assembler operator) ............................. 71
$ (program location counter) ..................... 23