

# IAR Assembler

## Reference Guide

for Freescale's  
HCS12 Microcontroller Family



## **COPYRIGHT NOTICE**

Copyright © 1997–2010 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, C-SPY, visualSTATE, From Idea To Target, IAR KickStart Kit, IAR PowerPac, IAR YellowSuite, IAR Advanced Development Kit, IAR, and the IAR Systems logotype are trademarks or registered trademarks owned by IAR Systems AB. J-Link is a trademark licensed to IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Freescale is a registered trademark of Freescale Inc.

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

## **EDITION NOTICE**

Third edition: February 2010

Part number: AHSC12-3

This guide applies to version 3.x of IAR Embedded Workbench® for HCS12.

# Contents

Tables .....	ix
Preface .....	xi
<b>Who should read this guide</b> .....	xi
<b>How to use this guide</b> .....	xi
<b>What this guide contains</b> .....	xii
<b>Other documentation</b> .....	xii
<b>Document conventions</b> .....	xiii
Introduction to the IAR Assembler for HCS12 .....	1
<b>Introduction to assembler programming</b> .....	1
Getting started .....	1
<b>Modular programming</b> .....	2
<b>Source format</b> .....	2
<b>Assembler instructions</b> .....	3
<b>Expressions, operands, and operators</b> .....	3
Integer constants .....	4
ASCII character constants .....	4
TRUE and FALSE .....	5
Symbols .....	5
Labels .....	5
Register symbols .....	6
Program counter-relative addressing symbol—PCR .....	6
Predefined symbols .....	7
Absolute and relocatable expressions .....	8
Expression restrictions .....	9
<b>List file format</b> .....	10
Header .....	10
Body .....	10
Summary .....	10
Symbol and cross-reference table .....	10

<b>Programming hints</b> .....	11
Accessing special function registers .....	11
Using C-style preprocessor directives .....	11
<b>Assembler options</b> .....	13
<b>Setting assembler options</b> .....	13
Specifying parameters .....	14
Environment variables .....	14
Error return codes .....	15
<b>Summary of assembler options</b> .....	15
<b>Description of assembler options</b> .....	16
<b>Assembler operators</b> .....	29
<b>Precedence of operators</b> .....	29
<b>Summary of assembler operators</b> .....	30
Parenthesis operator – 1 .....	30
Function operators – 2 .....	30
Unary operators – 3 .....	30
Multiplicative arithmetic operators – 4 .....	31
Additive arithmetic operators – 5 .....	31
Shift operators – 6 .....	31
Comparison operators – 7 .....	31
Equivalence operators – 8 .....	31
Logical operators – 9-14 .....	31
Conditional operator – 15 .....	32
<b>Description of assembler operators</b> .....	32
<b>Assembler directives</b> .....	45
<b>Summary of assembler directives</b> .....	45
<b>Module control directives</b> .....	48
Syntax .....	48
Parameters .....	49
Descriptions .....	49
<b>Symbol control directives</b> .....	51
Syntax .....	51

Parameters .....	51
Descriptions .....	52
Examples .....	52
<b>Segment control directives .....</b>	<b>53</b>
Syntax .....	53
Parameters .....	54
Descriptions .....	55
Examples .....	56
<b>Value assignment directives .....</b>	<b>58</b>
Syntax .....	58
Parameters .....	58
Descriptions .....	59
Examples .....	59
<b>Conditional assembly directives .....</b>	<b>61</b>
Syntax .....	61
Parameters .....	62
Descriptions .....	62
Examples .....	62
<b>Macro processing directives .....</b>	<b>63</b>
Syntax .....	63
Parameters .....	63
Descriptions .....	64
Examples .....	67
<b>Listing control directives .....</b>	<b>69</b>
Syntax .....	70
Descriptions .....	70
Examples .....	71
<b>C-style preprocessor directives .....</b>	<b>73</b>
Syntax .....	73
Parameters .....	73
Descriptions .....	74
Examples .....	76
<b>Data definition or allocation directives .....</b>	<b>77</b>
Syntax .....	78

Parameters .....	78
Descriptions .....	78
Examples .....	79
<b>Assembler control directives</b> .....	80
Syntax .....	80
Parameters .....	80
Descriptions .....	80
Examples .....	81
<b>Function directives</b> .....	81
Syntax .....	82
Parameters .....	82
Descriptions .....	82
<b>Call frame information directives</b> .....	83
Syntax .....	84
Parameters .....	85
Descriptions .....	86
Simple rules .....	90
CFI expressions .....	92
Example .....	94
Pragma directives .....	97
<b>Summary of pragma directives</b> .....	97
<b>Descriptions of pragma directives</b> .....	97
Diagnostics .....	99
<b>Message format</b> .....	99
<b>Severity levels</b> .....	99
Setting the severity level .....	100
Internal error .....	100
Migrating assembler code .....	101
<b>The migration process</b> .....	101
Assembler options .....	101
Assembler operators .....	103
Assembler directives .....	103

Index ..... 105





# Tables

1: Typographic conventions used in this guide .....	xiii
2: Integer constant formats .....	4
3: ASCII character constant formats .....	4
4: Predefined register symbols .....	6
5: Predefined symbols .....	7
6: Symbol and cross-reference table .....	10
7: Environment variables .....	15
8: Error return codes .....	15
9: Assembler options summary .....	15
10: Generating a list of dependencies (--dependencies) .....	18
11: Conditional list options (-l) .....	23
12: Directing preprocessor output to file (--preprocess) .....	26
13: Assembler directives summary .....	45
14: Module control directives .....	48
15: Symbol control directives .....	51
16: Segment control directives .....	53
17: Value assignment directives .....	58
18: Conditional assembly directives .....	61
19: Macro processing directives .....	63
20: Listing control directives .....	69
21: C-style preprocessor directives .....	73
22: Data definition or allocation directives .....	77
23: Using data definition or allocation directives .....	78
24: Assembler control directives .....	80
25: Call frame information directives .....	83
26: Unary operators in CFI expressions .....	92
27: Binary operators in CFI expressions .....	92
28: Ternary operators in CFI expressions .....	93
29: Code sample with backtrace rows and columns .....	94
30: Pragma directives summary .....	97
31: Version 3.10 compiler options not available in version 3.20 .....	102

32: Renamed or modified options .....	102
33: Renamed or modified assembler directives .....	103
34: Obsolete assembler directives .....	104

# Preface

Welcome to the IAR Assembler Reference Guide for HCS12. The purpose of this guide is to provide you with detailed reference information that can help you to use the IAR Assembler for HCS12 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 HCS12 microcontroller and need to get detailed reference information on how to use the IAR Assembler for HCS12. In addition, you should have working knowledge of the following:

- The architecture and instruction set of the HCS12 microcontroller. Refer to the documentation from Freescale for information about the HCS12 microcontroller
- 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 HCS12, you should read the chapter *Introduction to the IAR Assembler for HCS12* in this reference guide.

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 Systems toolkit, we recommend that you first read the initial chapters of the *IAR Embedded Workbench® IDE User Guide*. They give product overviews, as well as tutorials that can help you get started. The *IAR Embedded Workbench® IDE User Guide* also contains a glossary.

---

## What this guide contains

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

- *Introduction to the IAR Assembler for HCS12* 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
- *Migrating assembler code* presents the major differences between the IAR Assembler for HCS12 version 3.20 and the IAR Assembler for HCS12 version 3.10, and describes the migration considerations.

---

## Other documentation

The complete set of IAR Systems development tools for the HCS12 microcontroller is described in a series of guides and online help files. For information about:

- Using the IAR Embedded Workbench® IDE with the IAR C-SPY® Debugger, refer to the *IAR Embedded Workbench® IDE User Guide*
- Programming for the IAR C/C++ Compiler for HCS12, refer to the *IAR C/C++ Compiler Reference Guide for HCS12*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR DLIB Library, refer to the online help system
- Using the IAR CLIB Library, refer to the *IAR C Library Functions Reference Guide*, available from the online help system
- Porting application code and projects created with a previous 68HC12 IAR Embedded Workbench IDE, refer to the *IAR Embedded Workbench® Migration Guide for HCS12*.

All of these guides are delivered in hypertext PDF or HTML format on the installation media. Some of them are also delivered as printed books.

## Document conventions

This guide uses the following typographic conventions:



Style	Used for
<code>computer</code>	Text that you enter or that appears on the screen.
<i>parameter</i>	A label representing the actual value you should enter as part of a command.
[option]	An optional part of a command.
{mandatory}	A mandatory part of a command.
a   b   c	Alternatives in a command.
<b>bold</b>	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>reference</i>	A cross-reference within this guide or to another guide.
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench interface.
	Identifies instructions specific to the command line interface.

Table 1: Typographic conventions used in this guide



# Introduction to the IAR Assembler for HCS12

This chapter contains the following sections:

- Introduction to assembler programming
- Modular programming
- Source format
- Assembler instructions
- Expressions, operands, and operators
- List file format
- Programming hints.

---

## Introduction to assembler programming

Even if you do not intend to write a complete application in assembler language, there may be situations where you will find it necessary to write parts of the code in assembler, for example, when using mechanisms in the HCS12 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 HCS12 microcontroller. Refer to Freescale's 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 *IAR Embedded Workbench® IDE User Guide*
- Read about the assembler language interface—also useful when mixing C and assembler modules—in the *IAR C/C++ Compiler Reference Guide for HCS12*
- In the IAR Embedded Workbench IDE, you can base a new project on a *template* for an assembler project.

---

## Modular programming

Typically, you write your assembler code in assembler source files. In each source file, you define one or several assembler *modules* by using the module control directives. By structuring your code in small modules—in contrast to one single monolithic module—you can organize your application code in a logical structure, which makes the code easier to understand, and which benefits:

- an efficient program development
- reuse of modules
- maintenance.

Each module has a name and a type, where the type can be either `PROGRAM` or `LIBRARY`. The linker will always include a `PROGRAM` module, whereas a `LIBRARY` module is only included in the linked code if other modules reference a public symbol in the module. A module consists of one or more segments.

A *segment* is a logical entity containing a piece of data or code that should be mapped to a physical location in memory. You place your code and data in segments by using the segment control directives. A segment can be either *absolute* or *relocatable*. An absolute segment always has a fixed address in memory, whereas the address for a relocatable segment is resolved at link time. By using segments, you can control how your code and data will be placed in memory. Each segment consists of many *segment parts*. A segment part is the smallest linkable unit, which allows the linker to include only those units that are referred to.

---

## 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 register
- a predefined symbol
- the program location counter (PLC)
- an expression
- a register prefixed or suffixed by + or - to indicate pre-/post-inc/dec addressing modes where appropriate
- an expression prefixed by < or T:, > or N:, or suffixed by :8 or :16 to indicate direct or extended addressing mode where appropriate.

*comment*

Comment, preceded by a ; (semicolon)  
C or C++ comments are also allowed.

The components are separated by spaces or tabs.

A source line may not 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 may be set up differently in different editors, it is recommended that you do not use tabs in your source files.

The IAR Assembler for HCS12 uses the default filename extensions `s12`, `asm`, and `msa` for source files.

---

## Assembler instructions

The IAR Assembler for HCS12 supports the syntax for assembler instructions as described in the chip manufacturer's hardware documentation.

---

## Expressions, operands, and operators

Expressions consist of expression operands and operators.

The assembler will accept a wide range of expressions, including both arithmetic and logical operations. All operators use 32-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*, page 29.

The following 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), \*.

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

## INTEGER CONSTANTS

Since all IAR Systems assemblers use 32-bit two's complement internal arithmetic, integers have a (signed) range from -2147483648 to 2147483647.

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:

Integer type	Example
Binary	1010b
Octal	0723, 1234q
Decimal	1234, -1, 1234d
Hexadecimal	0xFFFF, 0FFFFh,

*Table 2: Integer constant formats*

**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 may be used in ASCII strings. If the quote character itself is to be accessed, two consecutive quotes must be used:

Format	Value
'ABCD'	ABCD (four characters).
"ABCD"	ABCD'\0' (five characters the last ASCII null).
'A ' 'B'	A'B
'A ' ' '	A'

*Table 3: ASCII character constant formats*

Format	Value
' ' ' ' (4 quotes)	'
' ' (2 quotes)	Empty string (no value).
"" (2 double quotes)	Empty string (an ASCII null character).
'\'	'', for quote within a string, as in 'I'd love to'
\\	\, for \ within a string
\"	", for double quote within a string

Table 3: ASCII character constant formats (Continued)

## 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).

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. See `--case_insensitive`, page 17 for additional information.

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.

## 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*.

If you need to refer to the program location counter in your assembler source code you can use the \* sign. For example:

```
BRA * ; Loop forever
```

At link time, the \* sign will expand to the start address of the current instruction.

## REGISTER SYMBOLS

The following table shows the existing predefined register symbols:

Name	Address size	Description
A, B	8 bits	Accumulators
D	16 bits	Accumulator
X, Y	16 bits	Index registers
SP	16 bits	Stack pointer
PC	16 bits	Program counter
CCR	8 bits	Condition code register
CCRW	16 bits	Extended CCR *
DH, DL, XH, XL, YH, YL, SPH, SPL, CCRH, CCRL	8 bits	High/low part of registers (*) (**)

Table 4: Predefined register symbols

\* Core=hcs12x only

\*\* Only allowed in register transfer instructions (EXG, SEX, TFR).

## PROGRAM COUNTER-RELATIVE ADDRESSING SYMBOL—PCR

To simplify program counter-relative addressing, you can use the symbol PCR instead of PC for all instructions that accept indexed addressing mode with PC as base register.

When you use the register symbol PC, the offset is added to the program counter to obtain the effective address.

However, when you use the symbol PCR, the offset is not an offset but an address. The IAR Assembler for HCS12 will calculate the difference between the specified address and the PC and generate an instruction with a PC-relative offset, for example:

```
ORG      $1000
LDAA    14, PC
LDAB    LABEL, PCR
```

```

                ORG        $1010
LABEL: DC8     $80

```

After this code has been executed, both the registers A and B will contain 0x80, because both of the `LDAX` instructions will load the value from the label `LABEL`.

**Note:** The generated PC-relative instruction will not be optimized. It will use a 16-bit offset even if a 5-bit or 9-bit offset would be sufficient.

## PREDEFINED SYMBOLS

The IAR Assembler for HCS12 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. The strings returned by the assembler are enclosed in double quotes.

The following predefined symbols are available:

Symbol	Value
<code>__BUILD_NUMBER__</code>	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.
<code>__CORE__</code>	A unique integer that identifies the core option in use.
<code>__DATE__</code>	The current date in dd/Mmm/yyyy format (string).
<code>__FILE__</code>	The name of the current source file (string).
<code>__IAR_SYSTEMS_ASM__</code>	IAR assembler identifier (number).
<code>__LINE__</code>	The current source line number (number).
<code>__TID__</code>	Target identity, consisting of two bytes (number). The high byte is the target identity, which is 0x21 for HCS12. The low byte is the processor option 0.
<code>__SUBVERSION__</code>	An integer that identifies the version letter of the version number, for example the C in 4.21C, as an ASCII character.
<code>__TIME__</code>	The current time in hh:mm:ss format (string).
<code>__VER__</code>	The version number in integer format; for example, version 4.17 is returned as 417 (number).

Table 5: Predefined symbols

Notice that `__TID__` is related to the predefined symbol `__TID__` in the IAR C/C++ Compiler for HCS12. It is described in the *IAR C/C++ Compiler Reference Guide for HCS12*.

### Including symbol values in code

There are several data definition directives provided to 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:

```
timdat DC8    __TIME__,"",__DATE__,0; time and date
      ...
      LDX    #timdat        ; Load address of string
      JSR    printstring    ; Call string output routine
```

### Testing symbols for conditional assembly

To test a symbol at assembly time, you can 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 versions, you can do as follows:

```
#if (__VER__ > 320)      ; New assembler version
...
...
#else                    ; Old assembler version
...
...
#endif
```

See *Conditional assembly directives*, page 61.

## 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, in some cases, relocatable symbols that cancel each out.

Expressions that include symbols in relocatable segments cannot be resolved at assembly time, because they depend on the location of segments.

Such expressions are evaluated and resolved at link time, by the IAR XLINK Linker. There are no restrictions on the expression; any operator can be used on symbols from any segment, or any combination of segments.

For example, a program could define the segments DATA and CODE as follows:

```
          EXTERN    third
          RSEG      DATA
first DS8      5
second DS8     3
```

```

                RSEG      CODE
start  ...

```

Then in segment CODE the following instructions are legal:

```

                INC      #first+7
                INC      #first-7
                INC      #7+first
                INC      #(first/second)*third

```

**Note:** At assembly time, there will be no range check. The range check will occur at link time and, if the values are too large, there will be 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 (segment 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 may 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 will, 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 segment. 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 will be 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 has been included in the source file, a symbol and cross-reference table is produced.

The following information is provided for each symbol in the table:

Information	Description
Label	The label's user-defined name.
Mode	ABS (Absolute), or REL (Relocatable).
Type	The label type.
Segment	The name of the segment that this label is defined relative to.
Value/Offset	The value (address) of the label within the current module, relative to the beginning of the current segment part.

*Table 6: Symbol and cross-reference table*



---

## Programming hints

This section gives hints on how to write efficient code for the IAR Assembler for HCS12. For information about projects including both assembler and C or C++ source files, see the *IAR C/C++ Compiler Reference Guide for HCS12*.

### ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of HCS12 derivatives are included in the IAR Systems product package, in the `\hcs12\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 HCS12, and they are suitable to use as templates when creating new header files for other HCS12 derivatives.

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

```
#ifdef __IAR_SYSTEMS_ASM__
    (assembler-specific defines)
#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 assembler-style comments. For more information about comments, see *Assembler control directives*, page 80.



# Assembler options

This chapter first explains how to set the options from the command line, and gives an alphabetical summary of the assembler options. It then provides detailed reference information for each assembler option.



The *IAR Embedded Workbench® IDE User Guide* describes how to set assembler options in the IAR Embedded Workbench® IDE, and gives reference information about the available options.

---

## Setting assembler options

To set assembler options from the command line, include them on the command line after the `ahcs12` command, either before or after the source filename. For example, when assembling the source `prog.s12`, use the following command to generate an object file with debug information:

```
ahcs12 prog --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`:

```
ahcs12 prog -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:

```
ahcs12 prog -DDEBUG=1
```

Generally, the order of options on the command line, both relative to each other and to the source filename, is *not* significant. There is, however, 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, and it may have parameters. You specify it with double dashes, for example `--debug`.

## SPECIFYING PARAMETERS

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

For instance, an include file path of `\usr\include` can be specified either as:

```
-I\usr\include
```

or as

```
-I \usr\include
```

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

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

When a parameter is needed for an option with a long name, it can be specified 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 may be repeated, and may also have comma-separated values (without space), for example:

```
--diag_warning=Be0001,Be0002
```

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

```
ahcs12 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 (`--`). The following example will generate a list on standard output:

```
ahcs12 prog -l ---
```

## ENVIRONMENT VARIABLES

Assembler options can also be specified in the `ASMHC12` environment variable. The assembler automatically appends the value of this variable to every command line, so it provides a convenient method of specifying options that are required for every assembly.

The following environment variables can be used with the IAR Assembler for HCS12:

Environment variable	Description
AHCS12_INC	Specifies directories to search for include files; for example: AHCS12_INC=c:\program files\iar systems\em bedded workbench 4.n\hcs12\inc;c:\headers
ASMHCS12	Specifies command line options; for example: ASMHCS12=-l asm.lst

Table 7: Environment variables

## ERROR RETURN CODES

The IAR Assembler for HCS12 returns status information to the operating system which can be tested in a batch file.

The following command line error codes are supported:

Code	Description
0	Assembly successful, but there may have been warnings.
1	There were warnings, provided that the option <code>--warnings_affect_exit_code</code> was used.
2	There were non-fatal errors or fatal assembly errors (making the assembler abort).
3	There were crashing errors.

Table 8: Error return codes

## Summary of assembler options

The following table summarizes the assembler options available from the command line:

Command line option	Description
<code>--case_insensitive</code>	Case-insensitive user symbols
<code>--core</code>	Core in use
<code>-D</code>	Defines preprocessor symbols
<code>--debug</code>	Generates debug information
<code>--dependencies</code>	Lists file dependencies
<code>--diag_error</code>	Treats these diagnostics as errors
<code>--diag_remark</code>	Treats these diagnostics as remarks
<code>--diag_suppress</code>	Suppresses these diagnostics
<code>--diag_warning</code>	Treats these diagnostics as warnings

Table 9: Assembler options summary

Command line option	Description
--diagnostics_tables	Lists all diagnostic messages
--dir_first	Allows directives in the first column
--enable_multibytes	Enables support for multibyte characters
--error_limit	Specifies the allowed number of errors before the assembler stops
-f	Extends the command line
--header_context	Lists all referred source files
-I	Includes file paths
-l	Output list file
-M	Macro quote characters
--mnm_first	Allows mnemonics in the first column
--no_path_in_file_macros	Removes the path from the return value of the symbols <code>__FILE__</code> and <code>__BASE_FILE__</code>
--no_warnings	Disables all warnings
--no_wrap_diagnostics	Disables wrapping of diagnostic messages
-o	Sets object filename
--only_stdout	Uses standard output only
--preinclude	Includes an include file before reading the source file
--preprocess	Preprocessor output to file
-r	Generates debug information
--remarks	Enables remarks
--silent	Sets silent operation
--warnings_affect_exit_code	Warnings affect exit code
--warnings_are_errors	Treats all warnings as errors

Table 9: Assembler options summary (Continued)

## 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, there is no check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

---

```
--case_insensitive --case_insensitive
```

Use this option to make user symbols case insensitive.

By default, case sensitivity is on. This means that, for example, `LABEL` and `label` refer to different symbols. Use `--case_insensitive` to turn case sensitivity off, in which case `LABEL` and `label` will refer to the same symbol.

You can also use the assembler directives `CASEON` and `CASEOFF` to control case sensitivity for user-defined symbols. See *Assembler control directives*, page 80, for more information.

**Note:** The `--case_insensitive` option does not affect preprocessor symbols. Preprocessor symbols are always case sensitive, regardless of whether they are defined in the IAR Embedded Workbench IDE or on the command line. See *Defining and undefining preprocessor symbols*, page 74.



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

---

```
--core --core={hcs12|hcs12x}
```

Use this option to select the microcontroller for which the code is to be generated.



**Project>Options>General Options >Target>Device**

---

```
-D -Dsymbol [=value]
```

Defines a symbol to be used by the preprocessor with the name `symbol` and the value `value`. If no value is specified, 1 is used.

The `-D` option allows you to specify a value or choice on the command line instead of in the source file.

### Example

You may want to arrange your source to produce either the test or production version of your program dependent on whether the symbol `TESTVER` was defined. To do this use include sections such as:

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

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

```
Production version: ahcs12 prog
Test version:       ahcs12 prog -DTESTVER
```

Alternatively, your source might use a variable that you need to 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:

```
ahcs12 prog -DFRAMERATE=3
```



### Project>Options>Assembler>Preprocessor>Defined symbols

```
--debug, -r --debug
```

```
-r
```

The `--debug` option makes the assembler generate debug information that allows a symbolic debugger such as the IAR C-SPY® Debugger to be used on the program.

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

When you use this option, each source file opened by the assembler is listed in a file. The following modifiers are available:

Option modifier	Description
i	Include only the names of files (default)
m	Makefile style

Table 10: Generating a list of dependencies (`--dependencies`)

If a *filename* is specified, the assembler stores the output in that file.

If a *directory* is specified, the assembler stores the output in that directory, in a file with the extension `i`. The filename will be the same as the name of the assembled source file, unless a different name has been specified with the option `-o`, in which case that name will be used.

To specify the working directory, replace *directory* with a period (`.`).

If `--dependencies` or `--dependencies=i` is used, the name of each opened source file, including the full path if available, is output on a separate line. For example:

```
c:\iar\product\include\stdio.h
d:\myproject\include\foo.h
```



If `--dependencies=m` is used, 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.r12: c:\iar\product\include\stdio.h
foo.r12: d:\myproject\include\foo.h
```

### Example 1

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

```
ahcs12 prog --dependencies=i listing
```

### Example 2

To generate a listing of file dependencies to a file called `listing.i` in the `mypath` directory, you would use:

```
ahcs12 prog --dependencies \mypath\listing
```

**Note:** Both `\` and `/` can be used as directory delimiters.

### Example 3

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

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

```
%.r12 : %.c
$(ASM) $(ASMFLAGS) $< --dependencies=m $*.d
```

That is, besides 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 IAR Embedded Workbench IDE.

---

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

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 will not be generated, and the exit code will not be 0.

The following example classifies warning `As001` as an error:

```
--diag_error=As001
```



**Project>Options>Assembler >Diagnostics>Treat these as errors**

---

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

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 may cause strange behavior in the generated code.

The following example classifies the warning `As001` as a remark:

```
--diag_remark=As001
```



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

---

```
--diag_suppress --diag_suppress=tag, tag, ...
```

Use this option to suppress diagnostic messages. The following example suppresses the warnings `As001` and `As002`:

```
--diag_suppress=As001, As002
```



**Project>Options>Assembler >Diagnostics>Suppress these diagnostics**

---

```
--diag_warning --diag_warning=tag, tag, ...
```

Use this option to classify diagnostic messages as warnings.

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

The following example classifies the remark `As028` as a warning:

```
--diag_warning=As028
```



**Project>Options>Assembler >Diagnostics>Treat these as warnings**

---

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

Use this option to list all possible diagnostic messages in a named file. This can be very convenient, for example, if you have 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.

If a *filename* is specified, the assembler stores the output in that file.

If a *directory* is specified, the assembler stores the output in that directory, in a file with the name `diagnostics_tables.txt`. To specify the working directory, replace *directory* with a period (`.`).

### Example 1

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

```
--diagnostics_tables diag
```

### Example 2

If you want to generate a table to a file `diagnostics_tables.txt` in the working directory, you could use:

```
--diagnostics_tables .
```

**Note:** Both `\` and `/` can be used as directory delimiters.



This option is not available in the IAR Embedded Workbench IDE.

---

```
--dir_first --dir_first
```

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

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



**Project>Options>Assembler >Language>Allow directives in first column**

---

```
--enable_multibytes --enable_multibytes
```

By default, multibyte characters cannot be used in assembler source code. If you use this option, multibyte characters in the source code are interpreted according to the host computer's default setting for multibyte support.

Multibyte characters are allowed in comments, in string literals, and in character constants. They are transferred untouched to the generated code.



**Project>Options>Assembler>Language>Enable multibyte support**

---

```
--error_limit --error_limit=n
```

Use the `--error_limit` option to specify the number of errors allowed before the assembler stops. By default, 100 errors are allowed. *n* must be a positive number; 0 indicates no limit.



This option is not available in the IAR Embedded Workbench IDE.

---

```
-f -f filename
```

Extends the command line with text read from the specified file. Notice that there must be a space between the option itself and the filename.

The `-f` option is particularly useful where there is a large number of options which are more conveniently placed in a file than on the command line itself. For example, to run the assembler with further options taken from the file `extend.xcl`, use:

```
ahcs12 prog -f extend.xcl
```



To set this option, use:

**Project>Options>Assembler>Extra Options**

---

```
--header_context --header_context
```

Occasionally, it is necessary to 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 IAR Embedded Workbench IDE.

---

```
-I -Iprefix
```

Adds the `#include` file search prefix *prefix*.

By default, the assembler searches for `#include` files only in the current working directory and in the paths specified in the `AHCS12_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:

```
-Ic:\global\ -Ic:\thisproj\headers\
```

and then writing:

```
#include "asmlib.hdr"
```

in the source, will 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 `AHCS12_INC` environment variable, provided that this variable is set.



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

---

```
-1 -l[a][d][e][m][o][x][N] {filename|directory}
```

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

You can choose to include one or more of the following types of information:

Command line option	Description
-1a	Assembled lines only
-1d	The <code>LSTOUT</code> directive controls if lines are written to the list file or not. Using <code>-1d</code> turns the start value for this to off.
-1e	No macro expansions
-1m	Macro definitions
-1o	Multiline code
-1x	Includes cross-references
-1N	Do not include diagnostics

Table 11: Conditional list options (-l)

If a *filename* is specified, the assembler stores the output in that file.

If a *directory* is specified, the assembler stores the output in that directory, in a file with the extension `1st`. The filename will be the same as the name of the assembled source file, unless a different name has been specified with the option `-o`, in which case that name will be used.

To specify the working directory, replace *directory* with a period (`.`).

#### Example 1

To generate a listing to the file `list.1st`, use:

```
ahcs12 sourcefile -l list
```

**Example 2**

If you assemble the file `mysource.s12` and want to generate a listing to a file `mysource.lst` in the working directory, you could use:

```
ahcs12 mysource -l .
```

**Note:** Both `\` and `/` can be used as directory delimiters.



To set related options, select:

**Project>Options>Assembler >List**

---

```
-M -Mab
```

Specifies quote characters for macro arguments by setting the characters used for the left and right quotes of each macro argument to *a* and *b* respectively.

By default, the characters are `<` and `>`. The `-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.

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

```
ahcs12 filename -M'<>'
```

**Example**

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

to call a macro `print` with `>` as the argument.



**Project>Options>Assembler >Language>Macro quote characters**

---

```
--mnm_first --mnm_first
```

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

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



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

---

```
--no_path_in_file_macros --no_path_in_file_macros
```

Use this option to exclude the path but leave the filename as the return value of the predefined preprocessor symbols `__FILE__` and `__BASE_FILE__`.



This option is not available in the IAR Embedded Workbench IDE.

---

```
--no_warnings --no_warnings
```

By default the assembler issues standard warning messages. Use this option to disable all warning messages.



This option is not available in the IAR Embedded Workbench IDE.

---

```
--no_wrap_diagnostics --no_wrap_diagnostics
```

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 IAR Embedded Workbench IDE.

---

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

Use the `-o` option to specify an output file.

If a *filename* is specified, the assembler stores the object code in that file.

If a *directory* is specified, the assembler stores the object code in that directory, in a file with the same name as the name of the assembled source file, but with the extension `r12`. To specify the working directory, replace *directory* with a period (`.`).

### Example 1

To store the assembler output in a file called `obj.r12` in the `mypath` directory, you would use:

```
ahcs12 sourcefile -o \mypath\obj
```

### Example 2

If you assemble the file `mysource.s12` and want to store the assembler output in a file `mysource.r12` in the working directory, you could use:

```
ahcs12 mysource -o .
```

**Note:** Both \ and / can be used as directory delimiters. You must include a space between the option itself and the filename.



**Project>Options>General Options>Output>Output directories>Object files**

---

--only\_stdout --only\_stdout

Causes the assembler to use `stdout` also for messages that are normally directed to `stderr`.



This option is not available in the IAR Embedded Workbench IDE.

---

--preinclude --preinclude *includefile*

Use this option to make the compiler 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>Extra Options**

---

--preprocess --preprocess=[c][n][l] {*filename*|*directory*}

Use this option to direct preprocessor output to a named file.

The following table shows the mapping of the available preprocessor modifiers:

Command line option	Description
--preprocess=c	Preserve comments that otherwise are removed by the preprocessor, that is, C and C++ style comments. Assembler style comments are always preserved
--preprocess=n	Preprocess only
--preprocess=l	Generate #line directives

*Table 12: Directing preprocessor output to file (--preprocess)*

If a *filename* is specified, the assembler stores the output in that file.

If a *directory* is specified, the assembler stores the output in that directory, in a file with the extension `i`. The filename will be the same as the name of the assembled source file, unless a different name has been specified with the option `-o`, in which case that name will be used.

To specify the working directory, replace *directory* with a period (`.`).



**Example 1**

To store the assembler output with preserved comments to the file `output.i`, use:

```
ahcs12 sourcefile --preprocess=c output
```

**Example 2**

If you assemble the file `mysource.s12` and want to store the assembler output with `#line` directives to a file `mysource.i` in the working directory, you could use:

```
ahcs12 mysource --preprocess=1 .
```

**Note:** Both `\` and `/` can be used as directory delimiters.

**Project>Options>Assembler >Preprocessor>Preprocessor output to file**


---

```
-r, --debug --debug
```

```
-r
```

The `--debug` option makes the assembler generate debug information that allows a symbolic debugger such as the IAR C-SPY Debugger to be used on the program.

In order 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**


---

```
--remarks --remarks
```

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 may cause strange behavior in the generated code. By default remarks are not generated.

See *Severity levels*, page 99, for additional information about diagnostic messages.

**Project>Options>Assembler >Diagnostics>Enable remarks**


---

```
--silent --silent
```

The `--silent` option causes the assembler to operate without sending any messages to the standard output stream.

By default, the assembler sends various insignificant messages via the standard output stream. You can use the `--silent` option to prevent this. 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 IAR Embedded Workbench IDE.

---

`--warnings_affect_exit_code` `--warnings_affect_exit_code`

By default the exit code is not affected by warnings, only errors produce a non-zero exit code. With this option, warnings will generate a non-zero exit code.



This option is not available in the IAR Embedded Workbench IDE.

---

`--warnings_are_errors` `--warnings_are_errors`

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, you can 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
```

For additional information, see *--diag\_warning*, page 20.



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

# Assembler operators

This chapter first describes the precedence of the assembler operators, and then summarizes the operators, classified according to their precedence. Finally, this chapter provides reference information about each operator, presented in alphabetical order.

---

## Precedence of 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).

The following 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, the following expression evaluates to 1:

$7 / (1 + (2 * 3))$

**Note:** The precedence order in the IAR Assembler for HCS12 closely follows the precedence order of the ANSI C++ standard for operators, where applicable.

---

## Summary of assembler operators

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

### PARENTHESIS OPERATOR – 1

( )	Paranthesis.
-----	--------------

### FUNCTION OPERATORS – 2

BYTE1	First byte.
BYTE2	Second byte.
BYTE3	Third byte.
BYTE4	Fourth byte.
DATE	Current date/time.
HIGH	High byte.
HWRD	High word.
LOW	Low byte.
LWRD	Low word.
SFB	Segment begin.
SFE	Segment end.
SIZEOF	Segment size.
UPPER	Third byte.

### UNARY OPERATORS – 3

+	Unary plus.
BINNOT [~]	Bitwise NOT.
NOT [!]	Logical NOT.
-	Unary minus.

**MULTIPLICATIVE ARITHMETIC OPERATORS – 4**

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

**ADDITIVE ARITHMETIC OPERATORS – 5**

+	Addition.
-	Subtraction.

**SHIFT OPERATORS – 6**

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

**COMPARISON OPERATORS – 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 – 8**

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

**LOGICAL OPERATORS – 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 – 15

?:	Conditional operator.
----	-----------------------

---

## Description of assembler operators

The following sections give full descriptions of each assembler operator. The number within parentheses specifies the priority of the operator

---

### ( ) Parenthesis (1).

( and ) group expressions to be evaluated separately, overriding the default precedence order.

#### **Example**

1+2\*3 → 7  
(1+2)\*3 → 9

---

### \* Multiplication (4).

\* 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**

2\*2 → 4  
-2\*2 → -4

---

### + Unary plus (3).

Unary plus operator.

#### **Example**

+3 → 3  
3\*+2 → 6

---

**+** Addition (5).

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**

```
92+19 → 111
-2+2 → 0
-2+-2 → -4
```

---

**-** Unary minus (3).

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**

```
-3 → -3
3*-2 → -6
4--5 → 9
```

---

**-** Subtraction (5).

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 (4).

/ produces the integer quotient of the left operand divided by the right operand. 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 (15).

The result of this operator is 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 ? will be considered the first character of the label.

### Syntax

*condition* ? *expr* : *expr*

### Example

```
5 ? 6 : 7 → 6
0 ? 6 : 7 → 7
```

---

AND [&&] Logical AND (12).

Use AND to perform logical AND between its two integer operands. If both operands are non-zero the result is 1 (true), otherwise it will be 0 (false).

### Example

```
1010B AND 0011B → 1
1010B AND 0101B → 1
1010B AND 0000B → 0
```

---

BINAND [&] Bitwise AND (9).

Use BINAND to perform 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 BINAND 0011B → 0010B
1010B BINAND 0101B → 0000B
1010B BINAND 0000B → 0000B
```

---

BINNOT [~] Bitwise NOT (3).

Use BINNOT to perform bitwise NOT on its operand. Each bit in the 32-bit result is the complement of the corresponding bit in the operand.



**Example**

```
BINNOT 1010B → 1111111111111111111111111111111110101B
```

---

**BINOR** [`|`] Bitwise OR (11).

Use **BINOR** to perform 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 BINOR 0101B → 1111B
1010B BINOR 0000B → 1010B
```

---

**BINXOR** [`^`] Bitwise exclusive OR (10).

Use **BINXOR** to perform 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 BINXOR 0101B → 1111B
1010B BINXOR 0011B → 1001B
```

---

**BYTE1** First byte (2).

**BYTE1** takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the low byte (bits 7 to 0) of the operand.

**Example**

```
BYTE1 0x12345678 → 0x78
```

---

**BYTE2** Second byte (2).

**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**

```
BYTE2 0x12345678 → 0x56
```

---

BYTE3 Third byte (2).

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 (2).

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 date/time (2).

Use the DATE operator to specify 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)
DATE 6	Current year MOD 100 (1998 →98, 2000 →00, 2002 →02)

**Example**

To assemble the date of assembly:

today: DC8 DATE 5, DATE 4, DATE 3

---

EQ [=] [==] Equal (8).

= 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.

**Example**

```
1 = 2 → 0
2 == 2 → 1
'ABC' = 'ABCD' → 0
```

---

GE [ $\geq$ ] Greater than or equal (7).

$\geq$  evaluates to 1 (true) if the left operand is equal to or has a higher numeric value than the right operand, otherwise it will be 0 (false).

**Example**

```
1 >= 2 → 0
2 >= 1 → 1
1 >= 1 → 1
```

---

GT [ $>$ ] Greater than (7).

$>$  evaluates to 1 (true) if the left operand has a higher numeric value than the right operand, otherwise it will be 0 (false).

**Example**

```
-1 > 1 → 0
2 > 1 → 1
1 > 1 → 0
```

---

HIGH High byte (2).

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.

**Example**

```
HIGH 0xABCD → 0xAB
```

---

HWRD High word (2).

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.

**Example**

```
HWRD 0x12345678 → 0x1234
```

---

LE [`<=`] Less than or equal (7).

`<=` evaluates to 1 (true) if the left operand has a lower or equal numeric value to the right operand, otherwise it will be 0 (false).

**Example**

```
1 <= 2 → 1
2 <= 1 → 0
1 <= 1 → 1
```

---

LOW Low byte (2).

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.

**Example**

```
LOW 0xABCD → 0xCD
```

---

LT [`<`] Less than (7).

`<` evaluates to 1 (true) if the left operand has a lower numeric value than the right operand, otherwise it will be 0 (false).

**Example**

```
-1 < 2 → 1
2 < 1 → 0
2 < 2 → 0
```

---

LWRD Low word (2).

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
```

---

MOD [%] Modulo (4).

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.

$X \text{ MOD } Y$  is equivalent to  $X - Y * (X / Y)$  using integer division.

**Example**

```
2 MOD 2 → 0
12 MOD 7 → 5
3 MOD 2 → 1
```

---

NE [<>] [!=] Not equal (8).

<> 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
```

---

NOT [!] Logical NOT (3).

Use NOT to negate a logical argument.

**Example**

```
NOT 0101B → 0
NOT 0000B → 1
```

---

OR [|] Logical OR (14).

Use OR to perform a logical OR between two integer operands.

**Example**

```
1010B OR 0000B → 1
0000B OR 0000B → 0
```

---

SFB Segment begin (2).

SFB accepts a single operand to its right. The operand must be the name of a relocatable segment. The operator evaluates to the absolute address of the first byte of that segment. This evaluation takes place at link time.

### Syntax

SFB(*segment* [{+|-}*offset*])

### Parameters

*segment* The name of a relocatable segment, which must be defined before SFB is used.

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

### Example

```

NAME demo
RSEG segtab:CONST
start: DC16 SFB(mycode)

```

Even if the above code is linked with many other modules, *start* will still be set to the address of the first byte of the segment.

---

SFE Segment end (2).

SFE accepts a single operand to its right. The operand must be the name of a relocatable segment. The operator evaluates to the segment start address plus the segment size. This evaluation takes place at link time.

### Syntax

SFE (*segment* [{+ | -} *offset*])

### Parameters

*segment* The name of a relocatable segment, which must be defined before SFE is used.

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

**Example**

```

        NAME    demo
        RSEG    segtab:CONST
end:    DC16    SFE(mycode)

```

Even if the above code is linked with many other modules, `end` will still be set to the first byte after that segment (`mycode`).

The size of the segment `MY_SEGMENT` can be calculated as:

```
SFE(MY_SEGMENT) - SFB(MY_SEGMENT)
```

**SHL** [`<<`] Logical shift left (6).

Use `SHL` to shift 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 SHL 3 → 11100000B
000001111111111111B SHL 5 → 11111111111100000B
14 SHL 1 → 28

```

**SHR** [`>>`] Logical shift right (6).

Use `SHR` to shift 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 SHR 3 → 00001110B
111111111111111111B SHR 20 → 0
14 SHR 1 → 7

```

**SIZEOF** Segment size (2).

`SIZEOF` generates `SFE-SFB` for its argument, which should be the name of a relocatable segment; that is, it calculates the size in bytes of a segment. This is done when modules are linked together.

**Syntax**

```
SIZEOF (segment)
```

**Parameters**

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

**Example**

The following code sets `size` to the size of the segment `mycode`.

```

MODULE table
RSEG mycode:CODE ;forward declaration of mycode
RSEG segtab:CONST
size: DC32 SIZEOF(mycode)
ENDMOD

MODULE application
RSEG mycode:CODE
NOP ;placeholder for application code
ENDMOD

```

---

UGT Unsigned greater than (7).

UGT evaluates to 1 (true) if the left operand has a larger value than the right operand, otherwise it will be 0 (false). The operation treats its operands as unsigned values.

**Example**

```

2 UGT 1 → 1
-1 UGT 1 → 1

```

---

ULT Unsigned less than (7).

ULT evaluates to 1 (true) if the left operand has a smaller value than the right operand, otherwise it will be 0 (false). The operation treats the operands as unsigned values.

**Example**

```

1 ULT 2 → 1
-1 ULT 2 → 0

```

---

UPPER Third byte (2).

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 (13).

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 an alphabetical summary of the assembler directives and provides detailed reference information for each category of directives.

---

## Summary of assembler directives

The following table gives a summary of all the assembler directives.

Directive	Description	Section
<code>#define</code>	Assigns a value to a label.	C-style preprocessor
<code>#elif</code>	Introduces a new condition in a <code>#if...#endif</code> block.	C-style preprocessor
<code>#else</code>	Assembles instructions if a condition is false.	C-style preprocessor
<code>#endif</code>	Ends a <code>#if</code> , <code>#ifdef</code> , or <code>#ifndef</code> block.	C-style preprocessor
<code>#error</code>	Generates an error.	C-style preprocessor
<code>#if</code>	Assembles instructions if a condition is true.	C-style preprocessor
<code>#ifdef</code>	Assembles instructions if a symbol is defined.	C-style preprocessor
<code>#ifndef</code>	Assembles instructions if a symbol is undefined.	C-style preprocessor
<code>#include</code>	Includes a file.	C-style preprocessor
<code>#line</code>	Changes the line numbers.	C-style preprocessor
<code>#pragma</code>	Controls extension features.	C-style preprocessor
<code>#undef</code>	Undefines a label.	C-style preprocessor
<code>/*comment*/</code>	C-style comment delimiter.	Assembler control
<code>//</code>	C++-style comment delimiter.	Assembler control
<code>=</code>	Assigns a permanent value local to a module.	Value assignment
<code>ALIGN</code>	Aligns the program location counter by inserting zero-filled bytes.	Segment control
<code>ALIGNRAM</code>	Aligns the program location counter.	Segment control
<code>ARGFRAME</code>	Declares the space used for the arguments to a function.	Function
<code>ASEG</code>	Begins an absolute segment.	Segment control
<code>ASEGN</code>	Begins a named absolute segment.	Segment control
<code>ASSIGN</code>	Assigns a temporary value.	Value assignment

*Table 13: Assembler directives summary*

<b>Directive</b>	<b>Description</b>	<b>Section</b>
CASEOFF	Disables case sensitivity.	Assembler control
CASEON	Enables case sensitivity.	Assembler control
CFI	Specifies call frame information.	Call frame information
COMMON	Begins a common segment.	Segment control
DC8	Generates 8-bit constants, including strings.	Data definition or allocation
DC16	Generates 16-bit constants.	Data definition or allocation
DC24	Generates 24-bit constants.	Data definition or allocation
DC32	Generates 32-bit constants.	Data definition or allocation
DC64	Generates 64-bit constants.	Data definition or allocation
DEFINE	Defines a file-wide value.	Value assignment
DF32	Generates 32-bit floating-point constants.	Data definition or allocation
DF64	Generates 64-bit floating-point constants.	Data definition or allocation
DS8	Allocates space for 8-bit integers.	Data definition or allocation
DS16	Allocates space for 16-bit integers.	Data definition or allocation
DS24	Allocates space for 24-bit integers.	Data definition or allocation
DS32	Allocates space for 32-bit integers.	Data definition or allocation
DS64	Allocates space for 64-bit integers.	Data definition or allocation
ELSE	Assembles instructions if a condition is false.	Conditional assembly
ELSEIF	Specifies a new condition in an IF...ENDIF block.	Conditional assembly
END	Terminates the assembly of the last module in a file.	Module control
ENDIF	Ends an IF block.	Conditional assembly
ENDM	Ends a macro definition.	Macro processing

*Table 13: Assembler directives summary (Continued)*

Directive	Description	Section
ENDMOD	Terminates the assembly of the current module.	Module control
ENDR	Ends a repeat structure.	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment
EVEN	Aligns the program counter to an even address.	Segment control
EXITM	Exits prematurely from a macro.	Macro processing
EXTERN	Imports an external symbol.	Symbol control
FUNCALL	Declares that the function <i>caller</i> calls the function <i>callee</i> .	Function
FUNCTION	Declares a label name to be a function.	Function
IF	Assembles instructions if a condition is true.	Conditional assembly
LIBRARY	Begins a library module.	Module control
LIMIT	Checks a value against limits.	Value assignment
LOCAL	Creates symbols local to a macro.	Macro processing
LOCFRAME	Declares the space used for the locals in a function.	Function
LSTCND	Controls conditional assembler listing.	Listing control
LSTCOD	Controls multi-line code listing.	Listing control
LSTEXP	Controls the listing of macro generated lines.	Listing control
LSTMAC	Controls the listing of macro definitions.	Listing control
LSTOUT	Controls assembler-listing output.	Listing control
LSTREP	Controls the listing of lines generated by repeat directives.	Listing control
LSTXRF	Generates a cross-reference table.	Listing control
MACRO	Defines a macro.	Macro processing
MODULE	Begins a library module.	Module control
NAME	Begins a program module.	Module control
ODD	Aligns the program location counter to an odd address.	Segment control
ORG	Sets the program location counter.	Segment control
PROGRAM	Begins a program module.	Module control
PUBLIC	Exports symbols to other modules.	Symbol control
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.	Symbol control
RADIX	Sets the default base.	Assembler control

Table 13: Assembler directives summary (Continued)

Directive	Description	Section
REPT	Assembles instructions a specified number of times.	Macro processing
REPTC	Repeats and substitutes characters.	Macro processing
REPTI	Repeats and substitutes strings.	Macro processing
REQUIRE	Forces a symbol to be referenced.	Symbol control
RSEG	Begins a relocatable segment.	Segment control
RTMODEL	Declares runtime model attributes.	Module control
SET	Assigns a temporary value.	Value assignment
VAR	Assigns a temporary value.	Value assignment

Table 13: Assembler directives summary (Continued)

## Module control directives

Module control directives are used for marking the beginning and end of source program modules, and for assigning names and types to them. See *Expression restrictions*, page 9, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
END	Terminates the assembly of the last module in a file.	Only locally defined labels or integer constants
ENDMOD	Terminates the assembly of the current module.	Only locally defined labels or integer constants
LIBRARY	Begins a library module.	No external references Absolute
MODULE	Begins a library module.	No external references Absolute
NAME	Begins a program module.	No external references Absolute
PROGRAM	Begins a program module.	No external references Absolute
RTMODEL	Declares runtime model attributes.	Not applicable

Table 14: Module control directives

### SYNTAX

```

END [address]
ENDMOD [address]
LIBRARY symbol [(expr)]
MODULE symbol [(expr)]
NAME symbol [(expr)]

```

```
PROGRAM symbol [(expr)]
RTMODEL key, value
```

## PARAMETERS

<i>address</i>	An optional expression that determines the start address of the program. It can take any positive integer value.
<i>expr</i>	An optional expression used by the compiler to encode the runtime options. It must be within the range 0-255 and evaluate to a constant value. The expression is only meaningful if you are assembling source code that originates as assembler output from the compiler.
<i>key</i>	A text string specifying the key.
<i>symbol</i>	Name assigned to module, used by XLINK, XAR, and XLIB when processing object files.
<i>value</i>	A text string specifying the value.

## DESCRIPTIONS

### 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 library module

Use `MODULE` or `LIBRARY` to create libraries containing a number of small modules—like runtime systems for high-level languages—where each module often represents a single routine. With the multi-module facility, you can significantly reduce the number of source and object files needed.

Library modules are only copied into the linked code if other modules reference a public symbol in the module.

### Terminating a module

Use `ENDMOD` to define the end of a module.

## 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 terminates the last module in the file, if this is not done explicitly with an `ENDMOD` directive.

## Assembling multi-module files

Program entries must be either relocatable or absolute, and will show up in XLINK load maps, as well as in some of the hexadecimal absolute output formats. Program entries must not be defined externally.

The following rules apply when assembling multi-module files:

- At the beginning of a new module all user symbols are deleted, except for those created by `DEFINE`, `#define`, or `MACRO`, the location counters are cleared, and the mode is set to absolute.
- Listing control directives remain in effect throughout the assembly.

**Note:** `END` must always be placed after the *last* module, and there must not be any source lines (except for comments and listing control directives) between an `ENDMOD` and a `MODULE` directive.

If the `NAME` or `MODULE` directive is missing, the module will be assigned the name of the source file and the attribute `program`.

## 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++ Compiler Reference Guide for HCS12*.

## Examples

The following example defines three modules where:

- `MOD_1` and `MOD_2` *cannot* be linked together since they have different values for runtime model `f00`.



- MOD\_1 and MOD\_3 *can* be linked together since they have the same definition of runtime model `bar` and no conflict in the definition of `foo`.
- MOD\_2 and MOD\_3 *can* be linked together since they have no runtime model conflicts. The value `*` matches any runtime model value.

```

MODULE MOD_1
    RTMODEL    "foo", "1"
    RTMODEL    "bar", "XXX"
    ...
ENDMOD

MODULE MOD_2
    RTMODEL    "foo", "2"
    RTMODEL    "bar", "*"
    ...
ENDMOD

MODULE MOD_3
    RTMODEL    "bar", "XXX"
    ...
END

```

---

## Symbol control directives

These directives control how symbols are shared between modules.

Directive	Description
EXTERN	Imports an external symbol.
PUBLIC	Exports symbols to other modules.
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.
REQUIRE	Forces a symbol to be referenced.

Table 15: Symbol control directives

### SYNTAX

```

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

```

### PARAMETERS

*symbol*                      Symbol to be imported or exported.

## DESCRIPTIONS

### 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 are no restrictions on the 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 several times. Only one of those definitions will be used by `XLINK`. If a module containing a `PUBLIC` definition of a symbol is linked with one or more modules containing `PUBWEAK` definitions of the same symbol, `XLINK` will use the `PUBLIC` definition.

A symbol defined as `PUBWEAK` must be a label in a segment part, and it must be the *only* symbol defined as `PUBLIC` or `PUBWEAK` in that segment part.

**Note:** Library modules are only linked if a reference to a symbol in that module is made, and that symbol has not already been 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` to import an untyped external symbol.

The `REQUIRE` directive marks a symbol as referenced. This is useful if the segment part containing the symbol must be loaded for the code containing the reference to work, but the dependence is not otherwise evident.

## EXAMPLES

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.

Since 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 will be resolved at link time.

```

NAME      error
EXTERN    print
PUBLIC    err

err JSR    print
        DC8    "*** Error ***"
        RTS

        END

```

## Segment control directives

The segment directives control how code and data are located. See *Expression restrictions*, page 9, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
ALIGN	Aligns the program location counter by inserting zero-filled bytes.	No external references Absolute
ALIGNRAM	Aligns the program location counter.	No external references Absolute
ASEG	Begins an absolute segment.	No external references Absolute
ASEGN	Begins a named absolute segment.	No external references Absolute
COMMON	Begins a common segment.	No external references Absolute
EVEN	Aligns the program counter to an even address.	No external references Absolute
ODD	Aligns the program counter to an odd address.	No external references Absolute
ORG	Sets the location counter.	No external references Absolute (see below)
RSEG	Begins a relocatable segment.	No external references Absolute

Table 16: Segment control directives

### SYNTAX

```
ALIGN align [, value]
```

```

ALIGNRAM align
ASEG [start]
ASEGN segment [:type], address
COMMON segment [:type] [(align)]
EVEN [value]
ODD [value]
ORG expr
RSEG segment [:type] [flag] [(align)]

```

## PARAMETERS

<i>address</i>	Address where this segment part will be placed.
<i>align</i>	The power of two to which the address should be aligned, in most cases in the range 0 to 30. The default align value is 0.
<i>expr</i>	Address to set the location counter to.
<i>flag</i>	NOROOT, ROOT NOROOT means that the segment part is discarded by the linker if no symbols in this segment part are referred to. Normally all segment parts except startup code and interrupt vectors should set this flag. The default mode is ROOT which indicates that the segment part must not be discarded.  REORDER, NOREORDER REORDER allows the linker to reorder segment parts. For a given segment, all segment parts must specify the same state for this flag. The default mode is NOREORDER which indicates that the segment parts must remain in order.  SORT, NOSORT SORT means that the linker will sort the segment parts in decreasing alignment order. For a given segment, all segment parts must specify the same state for this flag. The default mode is NOSORT which indicates that the segment parts will not be sorted.
<i>segment</i>	The name of the segment.
<i>start</i>	A start address that has the same effect as using an ORG directive at the beginning of the absolute segment.
<i>type</i>	The memory type, typically CODE or DATA. In addition, any of the types supported by the IAR XLINK Linker.
<i>value</i>	Byte value used for padding, default is zero.

## DESCRIPTIONS

### Beginning an absolute segment

Use `ASEG` to set the absolute mode of assembly, which is the default at the beginning of a module.

If the parameter is omitted, the start address of the first segment is 0, and subsequent segments continue after the last address of the previous segment.

### Beginning a named absolute segment

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

This directive has the advantage of allowing you to specify the memory type of the segment.

### Beginning a relocatable segment

Use `RSEG` to set the current mode of the assembly to relocatable assembly mode. The assembler maintains separate location counters (initially set to zero) for all segments, which makes it possible to switch segments and mode anytime without the need to save the current segment location counter.

Up to 65536 unique, relocatable segments may be defined in a single module.

### Beginning a common segment

Use `COMMON` to place data in memory at the same location as `COMMON` segments from other modules that have the same name. In other words, all `COMMON` segments of the same name will start at the same location in memory and overlay each other.

Obviously, the `COMMON` segment type should not be used for overlaid executable code. A typical application would be when you want a number of different routines to share a reusable, common area of memory for data.

It can be practical to have the interrupt vector table in a `COMMON` segment, thereby allowing access from several routines.

The final size of the `COMMON` segment is determined by the size of largest occurrence of this segment. The location in memory is determined by the `XLINK -z` command; see the *IAR Linker and Library Tools Reference Guide*.

Use the *align* parameter in any of the above directives to align the segment start address.

## Setting the program location counter (PLC)

Use `ORG` to set the program location counter of the current segment to the value of an expression. The optional parameter will assume the value and type of the new location counter. When `ORG` is used in an absolute segment (`ASEG`), the parameter expression must be absolute. However, when `ORG` is used in a relative segment (`RSEG`), the expression may be either absolute or relative (and the value is interpreted as an offset relative to the segment start in both cases).

The program location counter is set to zero at the beginning of an assembler module.

## Aligning a segment

Use `ALIGN` to align the program location counter to a specified address boundary. The expression gives the power of two to which the program counter should be aligned and the permitted range is 0 to 8.

The alignment is made relative to the segment start; normally this means that the segment 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 byte value for padding must be within the range 0 to 255.

Use `ALIGNRAM` to align the program location counter by incrementing it; no data is generated. The expression can be within the range 0 to 30.

## EXAMPLES

### Beginning an absolute segment

The following example assembles interrupt routine entry instructions in the appropriate interrupt vectors using an absolute segment:

```

        EXTERN  irqsrv,nmisrv
        ASEG
        ORG    $1000
main LDA    #1
        RTS

        ORG    $FFF2
        FDB   irqsrv      ; IRQ interrupt
        ORG    $FFF4
        FDB   nmisrv     ; NMI interrupt
        ORG    $FFFE

```

```

FDB    main    ; Power on
END

```

The main power-on code is assembled in memory starting at \$1000.

### Beginning a relocatable segment

In the following example, the data following the first `RSEG` directive is placed in a relocatable segment called `table`; the `ORG` directive is used for creating a gap of six bytes in the table.

The code following the second `RSEG` directive is placed in a relocatable segment called `code`:

```

EXTERN    divrtn,mulrtn
RSEG     table
FDB     divrtn,mulrtn
ORG     *+6
FDB     subrtn
RSEG     code
subrtn  LDAA    #1
SBA
END

```

### Beginning a common segment

The following example defines two common segments containing variables:

```

NAME     common1
COMMON   data
count  RMB  4
ENDMOD

NAME     common2
COMMON   data
up      RMB  1
ORG     *+2
down   RMB  1
END

```

Because the common segments have the same name, `data`, the variables `up` and `down` refer to the same locations in memory as the first and last bytes of the 4-byte variable `count`.

## Aligning a segment

This example starts a relocatable segment, moves to an even address, and adds some data. It then aligns to a 64-byte boundary before creating a 64-byte table.

```

                                RSEG      data ; Start a relocatable data segment
                                EVEN      ; Ensure it's on an even boundary
target    DC16      1      ; target and best will be on an
                                ; even boundary
best      DC16      1
                                ALIGN     6      ; Now align to a 64 byte boundary
results   DS8       64     ; And create a 64 byte table
                                END

```

---

## Value assignment directives

These directives are used for assigning values to symbols.

Directive	Description
=, EQU	Assigns a permanent value local to a module.
ASSIGN, SET, VAR	Assigns a temporary value.
DEFINE	Defines a file-wide value.
LIMIT	Checks a value against limits.

Table 17: Value assignment directives

### SYNTAX

```

label = expr
label ASSIGN expr
label DEFINE expr
label EQU expr
LIMIT expr, min, max, message
label SET expr
label VAR expr

```

### PARAMETERS

<i>expr</i>	Value assigned to symbol or value to be tested.
<i>label</i>	Symbol to be defined.
<i>message</i>	A text message that will be printed when <i>expr</i> is out of range.
<i>min, max</i>	The minimum and maximum values allowed for <i>expr</i> .



## DESCRIPTIONS

### Defining a temporary value

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

### Defining a permanent local value

Use `EQU` or `=` to assign a value to a symbol.

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 and all modules following that module in the same source file. If a `DEFINE` directive is placed outside of a module, the symbol will be known to all modules following the directive in the same source file.

A symbol which has been 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.

### Checking symbol values

Use `LIMIT` to check that expressions lie within a specified range. If the expression is assigned a value outside the range, an error message will appear.

The check will occur as soon as the expression is resolved, which will be during linking if the expression contains external references.

## EXAMPLES

### Redefining a symbol

The following example uses `VAR` to redefine the symbol `cons` in a `REPT` loop to generate a table of the first 8 powers of 3:

```

                NAME    table
cons           VAR      1
buildit       MACRO    times
```

```

                DC16      cons
cons           VAR        cons*3
                IF        times>1
                buildit   times-1
                ENDIF
                ENDM
main          buildit 4
                END

```

It generates the following code:

1		NAME	table
2	000001	cons	VAR 1
10	000000	main	buildit 4
10.1	000000 0001		DC16 cons
10.2	000003	cons	VAR cons*3
10.3	000002		IF 4>1
10.4	000002		buildit 4-1
10.5	000002 0003		DC16 cons
10.6	000009	cons	VAR cons*3
10.7	000004		IF 4-1>1
10.8	000004		buildit 4-1-1
10.9	000004 0009		DC16 cons
10.10	00001B	cons	VAR cons*3
10.11	000006		IF 4-1-1>1
10.12	000006		buildit 4-1-1-1
10.13	000006 001B		DC16 cons
10.14	000051	cons	VAR cons*3
10.15	000008		IF 4-1-1-1>1
10.16	000008		ENDIF
10.17	000008		ENDIF
10.18	000008		ENDIF
10.19	000008		ENDIF
11	000008		END

### Using local and global symbols

In the following example the symbol `value` defined in module `add1` is local to that module; a distinct symbol of the same name is defined in module `add2`. The `DEFINE` directive is used for declaring `locn` for use anywhere in the file:

```

                NAME      add1
locn          DEFINE    100H
value        EQU       77
                LDAA     locn
                ADDA     value
                RTS
                ENDMOD

```

```

NAME      add2
value EQU  88
LDAA     locn
ADDA     value
RTS
END

```

The symbol `locn` defined in module `add1` is also available to module `add2`.

### Using the **LIMIT** directive

The following example sets the value of a variable called `speed` and then checks it, at assembly time, to see if it is in the range 10 to 30. This might be useful if `speed` is often changed at compile time, but values outside a defined range would cause undesirable behavior.

```

speed     SET      23
          LIMIT    speed,10,30, "Speed is out of range!"

```

---

## Conditional assembly directives

These directives provide logical control over the selective assembly of source code. See *Expression restrictions*, page 9, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
ELSE	Assembles instructions if a condition is false.	
ELSEIF	Specifies a new condition in an IF...ENDIF block.	No forward references No external references Absolute Fixed
ENDIF	Ends an IF block.	
IF	Assembles instructions if a condition is true.	No forward references No external references Absolute Fixed

Table 18: Conditional assembly directives

### SYNTAX

```

ELSE
ELSEIF condition
ENDIF
IF condition

```

## PARAMETERS

<i>condition</i>	One of the following:	
	An absolute expression	The expression must not contain forward or external references, and any non-zero value is considered as true.
	<i>string1==string2</i>	The condition is true if <i>string1</i> and <i>string2</i> have the same length and contents.
	<i>string1!=string2</i>	The condition is true if <i>string1</i> and <i>string2</i> have different length or contents.

## DESCRIPTIONS

Use the `IF`, `ELSE`, and `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 (i.e. it will not be assembled or syntax checked) until an `ELSE` or `ENDIF` directive is found.

Use `ELSEIF` to introduce a new condition after an `IF` directive. Conditional assembly directives may 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 may 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 may be nested to any level.

## EXAMPLES

The following macro adds a constant to the A register:

```

ADV    MACRO    v
        IF      v==1
        INCA
        ELSE
        ADDA    #v
        ENDIF
        ENDMAC

```

If the argument to the macro is 1, an `INCA` instruction is generated to save instruction cycles; otherwise an `ADDA` instruction is generated.

It could be tested with the following program:

```
main  LDAA    #0
      ADDV   1
      ADDV   2
      END
```

## Macro processing directives

These directives allow user macros to be defined. See *Expression restrictions*, page 9, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
ENDM	Ends a macro definition.	
ENDR	Ends a repeat structure.	
EXITM	Exits prematurely from a macro.	
LOCAL	Creates symbols local to a macro.	
MACRO	Defines a macro.	
REPT	Assembles instructions a specified number of times.	No forward references No external references Absolute Fixed
REPTC	Repeats and substitutes characters.	
REPTI	Repeats and substitutes text.	

Table 19: Macro processing directives

### SYNTAX

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

### PARAMETERS

*actual*        A string to be substituted.

*argument*    A symbolic argument name.

<i>expr</i>	An expression.
<i>formal</i>	An argument into which each character of <i>actual</i> (REPTC) or each <i>actual</i> (REPTI) is substituted.
<i>name</i>	The name of the macro.
<i>symbol</i>	A symbol to be local to the macro.

## DESCRIPTIONS

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.

**Note:** Avoid using C-type preprocessor directives within assembler macros, as this might lead to unexpected behavior, see *Using C-style preprocessor directives*, page 11.

## 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:

```

        EXTERN      abort
errmac MACRO      text
        JSR         abort
        DC8         text,0
        ENDM

```

This macro uses a parameter `text` to set up an error number for a routine `abort`. You would call the macro with a statement such as:

```
errmac 'Disk not ready'
```

The assembler will expand this to:

```

        JSR         abort
        DC8         'Disk not ready',0

```

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:

```
errmac2  MACRO
          JSR      abort
          DC8      \1,0
        ENDM
```

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:

```
mac1d   MACRO  op
          LDAA  op
        ENDM
```

The macro can be called using the macro quote characters:

```
mac1d  <3, X>
END
```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 24.

### Predefined macro symbols

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

```
FILL  MACRO
      IF  _args == 2
      REPT  \1
      DC8  \2
      ENDR
      ELSE
```

```

        DC8    \1
    ENDF
    ENDM

RSEG   CODE

        FILL  3, 4
        FILL  3

    END

```

It generates the following code:

10		
11	000000	RSEG CODE
12		
13	000000	FILL 3, 4
13.1	000000	IF _args == 2
13.2	000000	REPT 3
13.3	000000 04	DC8 4
13.4	000001 04	DC8 4
13.5	000002 04	DC8 4
13.6	000003	ENDR
13.7	000003	ELSE
13.8	000003	ENDIF
14	000003	FILL 3
14.1	000003	IF _args == 2
14.2	000003	ELSE
14.3	000003 03	DC8 3
14.4	000004	ENDIF
15		
16	000004	END

## How macros are processed

There are three distinct phases in the macro process:

- 1 The assembler performs scanning and saving of 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 will continue to be the output from the macro processor, until all lines of the current macro definition have been read.

### Repeating statements

Use the `REPT . . . ENDR` structure to assemble the same block of instructions a number of times. If *expr* evaluates to 0 nothing will be 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.

### EXAMPLES

This section gives examples of the different ways in which macros can make assembler programming easier.

#### 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.

The following example outputs bytes from a buffer to a port:

```

                EXTERN    port
                RSEG     DATA
buffer DS8      512          ;buffer

                RSEG     CODE
play  LDX      #buffer
loop  LDAA     0,X
      CPX      #buffer+512
      BNE     loop
      RTS

```

The main program calls this routine as follows:

```
        JSR        play
```

For efficiency we can rewrite this as the following macro:

```
play    MACRO
        LOCAL     loop
        LDX      #buffer
loop    LDAA     0,x
        CPX      #buffer+512
        BNE     loop
        ENDMAC

        RSEG     DATA
buffer  DS8     512          ,buffer
        RSEG     CODE
        play
        RTS
        END
```

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

## Using `REPTC` and `REPTI`

The following example assembles a series of calls to a subroutine `plotc` to plot each character in a string:

```
        NAME     reptc
        EXTERN   plotc
banner  REPTC    chr,"Welcome"
        LDAA    #'chr'
        JSR     plotc
        ENDR
        END
```

This produces the following code:

```
1          NAME     reptc
2    000000    EXTERN   plotc
3    000000    banner  REPTC    chr,"Welcome"
3.1  000000  8657    LDAA    #'W'
3.2  000002  16....    JSR     plotc
3.3  000005  8665    LDAA    #'e'
3.4  000007  16....    JSR     plotc
3.5  00000A  866C    LDAA    #'l'
3.6  00000C  16....    JSR     plotc
3.7  00000F  8663    LDAA    #'c'
```

```

3.8 000011 16... JSR plotc
3.9 000014 866F LDAA #'o'
3.10 000016 16... JSR plotc
3.11 000019 866D LDAA #'m'
3.12 00001B 16... JSR plotc
3.13 00001E 8665 LDAA #'e'
3.14 000020 16... JSR plotc
3.15 000023 ENDR
7 000023 END

```

The following example uses `REPTI` to clear a number of memory locations:

```

NAME repti
EXTERN base,count,init
banner REPTI adds,base,count,init
CLR adds
ENDR
END

```

This produces the following code:

```

1 NAME repti
2 000000 EXTERN base,count,init
3 000000 banner REPTI adds,base,count,init
3.1 000000 79... CLR base
3.2 000003 79... CLR count
3.3 000006 79... CLR init
3.4 000009 ENDR
6 000009 END

```

## Listing control directives

These directives provide control over the assembler list file.

Directive	Description
LSTCND	Controls conditional assembly listing.
LSTCOD	Controls multi-line code listing.
LSTEXP	Controls the listing of macro-generated lines.
LSTMAC	Controls the listing of macro definitions.
LSTOUT	Controls assembly-listing output.
LSTREP	Controls the listing of lines generated by repeat directives.
LSTXRF	Generates a cross-reference table.

Table 20: Listing control directives

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

## SYNTAX

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

## DESCRIPTIONS

### 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).

### 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 will 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.

### Controlling the listing of macros

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

Use `LSTMAC+` to list macro definitions. The default is `LSTMAC-`, which disables the listing of macro definitions.

### 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.

## EXAMPLES

### Turning the listing on or off

To disable the listing of a debugged section of program:

```
LSTOUT-
; Debugged section
LSTOUT+
; Not yet debugged
```

### Listing conditional code and strings

The following example shows how `LSTCND+` hides a call to a subroutine that is disabled by an `IF` directive:

```

NAME      lstcndtst
EXTERN   print
RSEG     prom
debug    SET      0

begin    IF      debug
        JSR     print
        ENDIF

        LSTCND+
begin2   IF      debug
        CALL   print
        ENDIF
        END
```

This will generate the following listing:

```

1                                     NAME      lstcndtst
2      000000                         EXTERN   print
3      000000                         RSEG     prom
4      000000          debug    SET      0
5
6      000000          begin    IF      debug
8      000000                         ENDIF
```

```

9
10                                LSTCND+
11    000000                    begin2 IF      debug
13    000000                                ENDF
14    000000                                END

```

### Controlling the listing of macros

```

dec2    MACRO  arg
        DEC   arg
        DEC   arg
        ENDM

        LSTMAC+
inc2    MACRO  arg
        INC   arg
        INC   arg
        ENDM

        EXTERN memloc
begin   dec2  memloc

        LSTEXP-
inc2    memloc
RTS
END     begin

```

This will produce the following output:

```

5
6                                LSTMAC+
7                                inc2    MACRO  arg
8                                INC     arg
9                                INC     arg
10                               ENDM
11
12    000000                    EXTERN  memloc
13    000000                    begin   dec2  memloc
13.1  000000 73....          DEC     memloc
13.2  000003 73....          DEC     memloc
14
15                               LSTEXP-
16    000006                    inc2    memloc
17    00000C 3D                RTS
18    00000D                    END     begin

```

## C-style preprocessor directives

The following C-language preprocessor directives are available:

Directive	Description
<code>#define</code>	Assigns a value to a preprocessor symbol.
<code>#elif</code>	Introduces a new condition in a <code>#if...#endif</code> block.
<code>#else</code>	Assembles instructions if a condition is false.
<code>#endif</code>	Ends a <code>#if</code> , <code>#ifdef</code> , or <code>#ifndef</code> block.
<code>#error</code>	Generates an error.
<code>#if</code>	Assembles instructions if a condition is true.
<code>#ifdef</code>	Assembles instructions if a preprocessor symbol is defined.
<code>#ifndef</code>	Assembles instructions if a preprocessor symbol is undefined.
<code>#include</code>	Includes a file.
<code>#line</code>	Changes the line numbers of the source code lines immediately following the <code>#line</code> directive, or the filename of the file being compiled.
<code>#pragma</code>	Controls extension features. The supported <code>#pragma</code> directives are described in the chapter <i>Pragma directives</i> .
<code>#undef</code>	Undefines a preprocessor symbol.

Table 21: C-style preprocessor directives

### SYNTAX

```
#define symbol text
#elif condition
#else
#endif
#error "message"
#if condition
#ifdef symbol
#ifndef symbol
#include {"filename" | <filename>}
#undef symbol
```

### PARAMETERS

*condition*      An absolute expression

The expression must not contain any assembler labels or symbols, and any non-zero value is considered as true.

<i>filename</i>	Name of file to be included.
<i>message</i>	Text to be displayed.
<i>symbol</i>	Preprocessor symbol to be defined, undefined, or tested.
<i>text</i>	Value to be assigned.

## DESCRIPTIONS

The preprocessor directives are processed before other directives. As an example avoid constructs like:

```
redef macro      ; avoid the following
#define \1 \2
    endm
```

since the \1 and \2 macro arguments will not be available during the preprocess.

Also be careful with comments; the preprocessor understands /\* \*/ and //, but not assembler comments. The following expression will evaluate to 3 since the comment character will be preserved by #define:

```
#define x 3      ; comment
exp EQU x*8+5
```

**Note:** It is important to avoid mixing the assembler language with the C-style preprocessor directives. Conceptually, they are different languages and mixing them may lead to unexpected behavior because an assembler directive is not necessarily accepted as a part of the C preprocessor language.

The following example illustrates some problems that may occur when assembler comments are used in the C-style preprocessor:

```
#define      five 5 ; comment

    STAA      [five,X]          ; Syntax error!
    ; Expands to "STAA      [5;comment,X]"

    LDAA      five + address    ; Incorrect code!
    ; Expanded to "LDAA 5 ; comment + address"
```

## Defining and undefining preprocessor symbols

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

```
#define symbol value
```



is similar to:

```
symbol SET 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 (i.e. it will not be assembled or syntax checked) until a `#endif` or `#else` directive is found.

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

`#if...#endif` and `#if...#else...#endif` blocks may 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.

## 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 <iohcs12.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 The directories specified using the `AHCS12_INC` environment variable, if any.

- When the assembler encounters the name of an `#include` file in double quotes such as:

```
#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 HCS12, and double quotes for header files that are part of your application.

### Displaying errors

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

### Defining comments

Use `/* ... */` to comment sections of the assembler listing.

Use `//` to mark the rest of the line as comment.

## EXAMPLES

### Using conditional preprocessor directives

The following example defines a label `adjust`, and then uses the conditional directive `#ifdef` to use the value if it is defined. If it is not defined `#error` displays an error:

```

                                NAME      ifdef
                                EXTERN     input,output
#define adjust                  10

main        LDAA      input
#ifdef     adjust
            ADDA      #adjust

#else
#error     "'adjust' not defined"
#endif
#undef     adjust
            STAA      output
            RTS
            END

```

## Including a source file

The following example uses `#include` to include a file defining macros into the source file. For example, the following macros could be defined in `exchange.s12`:

```
xch MACRO loc1,loc2
    LDAA loc1
    LDAB loc2
    STAA loc2
    STAB loc1
ENDMAC
```

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

```
        NAME include
mem1   DS8 1
mem2   DS8 1
#include "exchange.s33"
main   xch mem1,mem2
        RTS
        END
```

---

## Data definition or allocation directives

These directives define values or reserve memory. See *Expression restrictions*, page 9, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description
DC8	Generates 8-bit constants, including strings.
DC16	Generates 16-bit constants.
DC24	Generates 24-bit constants.
DC32	Generates 32-bit constants.
DC64	Generates 64-bit constants.
DF32	Generates 32-bit floating-point constants.
DF64	Generates 64-bit floating-point constants.
DS8	Allocates space for 8-bit integers.
DS16	Allocates space for 16-bit integers.
DS24	Allocates space for 24-bit integers.
DS32	Allocates space for 32-bit integers.
DS64	Allocates space for 64-bit integers.

Table 22: 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] ...
DS8  count
DS16 count
DS24 count
DS32 count
DS64 count

```

## PARAMETERS

*count*      A valid absolute expression specifying the number of elements to be reserved.

*expr*        A valid absolute, relocatable, or external expression, or an ASCII string. ASCII strings will be zero filled to a multiple of the data size implied by the directive. Double-quoted strings will be zero-terminated.

*value*       A valid absolute expression or floating-point constant.

## DESCRIPTIONS

Use the data definition and allocation directives according to the following table; it shows which directives reserve and initialize memory space or reserve uninitialized memory space, and their size.

Size	Reserve and initialize memory	Reserve uninitialized memory
8-bit integers	DC8	DS8
16-bit integers	DC16	DS16
24-bit integers	DC24	DS24
32-bit integers	DC32	DS32
64-bit integers	DC64	DS64
32-bit floats	DF32	DS32
64-bit floats	DF64	DS64

Table 23: Using data definition or allocation directives

## EXAMPLES

### Generating a lookup table

The following example generates a lookup table of addresses to routines:

```

                                NAME      table
                                RSEG      CONST
table    DC16      addsubr, subsubr, clrsubr

                                RSEG      CODE
addsubr:  ABA
          RTS
subsubr:  SBA
          RTS
clrsubr:  CLRA
          RTS
          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 0xA bytes:

```
table    DS8    0xA
```

## Assembler control directives

These directives provide control over the operation of the assembler. See *Expression restrictions*, page 9, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
<code>/*comment*/</code>	C-style comment delimiter.	
<code>//</code>	C++ style comment delimiter.	
<code>CASEOFF</code>	Disables case sensitivity.	
<code>CASEON</code>	Enables case sensitivity.	
<code>RADIX</code>	Sets the default base on all numeric values.	No forward references No external references Absolute Fixed

Table 24: Assembler control directives

### SYNTAX

```
/*comment*/
//comment
CASEOFF
CASEON
RADIX expr
```

### PARAMETERS

*comment*                    Comment ignored by the assembler.

*expr*                        Default base; default 10 (decimal).

### DESCRIPTIONS

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 `XLINK` should be written in upper case in the `XLINK` definition file.

## EXAMPLES

### Defining comments

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

```
/*
Program to read serial input.
Version 3: 19.2.06
Author: mjp
*/
```

### Changing the base

To set the default base to 16:

```
RADIX 16
LDAA #12
```

The immediate argument will then be interpreted as H'12.

### Controlling case sensitivity

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

```
label NOP ; Stored as "LABEL"
BRA LABEL
```

The following will generate a duplicate label error:

```
label NOP
LABEL NOP ; Error, "LABEL" already defined

END
```

---

## Function directives

The function directives are generated by the IAR C/C++ Compiler for HCS12 to pass information about functions and function calls to the IAR XLINK Linker. These directives can be seen if you create an assembler list file by using the compiler option **Output assembler file>Include compiler runtime information (-lA)**.

**Note:** These directives are primarily intended to support static overlay, a feature which is useful in smaller microcontrollers. The IAR C/C++ Compiler for HCS12 does not use static overlay, as it has no use for it.

## SYNTAX

```

FUNCTION <label>, <value>
ARGFRAME <segment>, <size>, <type>
LOCFRAME <segment>, <size>, <type>
FUNCCALL <caller>, <callee>

```

## PARAMETERS

<i>label</i>	A label to be declared as function.
<i>value</i>	Function information.
<i>segment</i>	The segment in which argument frame or local frame is to be stored.
<i>size</i>	The size of the argument frame or the local frame.
<i>type</i>	The type of argument or local frame; either <code>STACK</code> or <code>STATIC</code> .
<i>caller</i>	The caller to a function.
<i>callee</i>	The called function.

## DESCRIPTIONS

`FUNCTION` declares the *label* name to be a function. *value* encodes extra information about the function.

`FUNCCALL` declares that the function *caller* calls the function *callee*. *callee* can be omitted to indicate an indirect function call.

`ARGFRAME` and `LOCFRAME` declare how much space the frame of the function uses in different memories. `ARGFRAME` declares the space used for the arguments to the function, `LOCFRAME` the space for locals. *segment* is the segment in which the space resides. *size* is the number of bytes used. *type* is either `STACK` or `STATIC`, for stack-based allocation and static overlay allocation, respectively.

`ARGFRAME` and `LOCFRAME` always occur immediately after a `FUNCTION` or `FUNCCALL` directive.

After a `FUNCTION` directive for an external function, there can only be `ARGFRAME` directives, which indicate the maximum argument frame usage of any call to that function. After a `FUNCTION` directive for a defined function, there can be both `ARGFRAME` and `LOCFRAME` directives.

After a `FUNCCALL` directive, there will first be `LOCFRAME` directives declaring frame usage in the calling function at the point of call, and then `ARGFRAME` directives declaring argument frame usage of the called function.



## Call frame information directives

These directives allow backtrace information to be defined in the assembler source code. The benefit is that you can view the call frame stack when you debug your assembler code.

Directive	Description
CFI BASEADDRESS	Declares a base address CFA (Canonical Frame Address).
CFI BLOCK	Starts a data block.
CFI CODEALIGN	Declares code alignment.
CFI COMMON	Starts or extends a common block.
CFI CONDITIONAL	Declares data block to be a conditional thread.
CFI DATAALIGN	Declares data alignment.
CFI ENDBLOCK	Ends a data block.
CFI ENDCOMMON	Ends a common block.
CFI ENDNAMES	Ends a names block.
CFI FRAMECELL	Creates a reference into the caller's frame.
CFI FUNCTION	Declares a function associated with data block.
CFI INVALID	Starts range of invalid backtrace information.
CFI NAMES	Starts a names block.
CFI NOFUNCTION	Declares data block to not be associated with a function.
CFI PICKER	Declares data block to be a picker thread.
CFI REMEMBERSTATE	Remembers the backtrace information state.
CFI RESOURCE	Declares a resource.
CFI RESOURCEPARTS	Declares a composite resource.
CFI RESTORESTATE	Restores the saved backtrace information state.
CFI RETURNADDRESS	Declares a return address column.
CFI STACKFRAME	Declares a stack frame CFA.
CFI STATICOVERLAYFRAME	Declares a static overlay frame CFA.
CFI VALID	Ends range of invalid backtrace information.
CFI VIRTUALRESOURCE	Declares a virtual resource.
CFI <i>cfa</i>	Declares the value of a CFA.
CFI <i>resource</i>	Declares the value of a resource.

Table 25: Call frame information directives

## SYNTAX

The syntax definitions below show the syntax of each directive. The directives are grouped according to usage.

### 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 STATICOVERLAYFRAME cfa segment [, cfa segment] ...
CFI BASEADDRESS cfa type [, cfa type] ...
```

### Extended names block directives

```
CFI NAMES name EXTENDS namesblock
CFI ENDNAMES name
CFI FRAMECELL cell cfa(offset):size [, cell cfa(offset):size] ...
```

### Common block directives

```
CFI COMMON name USING namesblock
CFI ENDCOMMON name
CFI CODEALIGN codealignfactor
CFI DATAALIGN dataalignfactor
CFI RETURNADDRESS resource type
CFI cfa { NOTUSED | USED }
CFI cfa { resource | resource + constant | resource - constant }
CFI cfa cfiexpr
CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
CFI resource { resource | FRAME(cfa, offset) }
CFI resource cfiexpr
```

### Extended common block directives

```
CFI COMMON name EXTENDS commonblock USING namesblock
CFI ENDCOMMON name
```

### Data block directives

```
CFI BLOCK name USING commonblock
CFI ENDBLOCK name
CFI { NOFUNCTION | FUNCTION label }
CFI { INVALID | VALID }
CFI { REMEMBERSTATE | RESTORESTATE }
```

```
CFI PICKER
CFI CONDITIONAL label [, label] ...
CFI cfa { resource | resource + constant | resource - constant }
CFI cfa cfiexpr
CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
CFI resource { resource | FRAME(cfa, offset) }
CFI resource cfiexpr
```

## PARAMETERS

<i>bits</i>	The size of the resource in bits.
<i>cell</i>	The name of a frame cell.
<i>cfa</i>	The name of a CFA (canonical frame address).
<i>cfiexpr</i>	A CFI expression (see <i>CFI expressions</i> , page 92).
<i>codealignfactor</i>	The smallest 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 will shrink the produced backtrace information in size. The possible range is 1–256.
<i>commonblock</i>	The name of a previously defined common block.
<i>constant</i>	A constant value or an assembler expression that can be evaluated to a constant value.
<i>dataalignfactor</i>	The smallest factor of all frame sizes. If the stack grows towards higher addresses, the factor is negative; if it grows towards lower addresses, the factor is positive. 1 is the default, but a larger value will shrink the produced backtrace information in size. The possible ranges are -256 – -1 and 1 – 256.
<i>label</i>	A function label.
<i>name</i>	The name of the block.
<i>namesblock</i>	The name of a previously defined names block.
<i>offset</i>	The offset relative the CFA. An integer with an optional sign.
<i>part</i>	A part of a composite resource. The name of a previously declared resource.
<i>resource</i>	The name of a resource.
<i>segment</i>	The name of a segment.

<i>size</i>	The size of the frame cell in bytes.
<i>type</i>	The memory type, such as <code>CODE</code> , <code>CONST</code> or <code>DATA</code> . In addition, any of the memory types supported by the IAR XLINK Linker. It is used solely for the purpose of denoting an address space.

## DESCRIPTIONS

The call frame information directives (CFI directives) are an extension to the debugging format of the IAR C-SPY® Debugger. The CFI directives are used for defining the *backtrace information* for the instructions in a program. The compiler normally generates this information, but for library functions and other code written purely in assembler language, backtrace information has to be added if you want to use the call frame stack in the debugger.

The backtrace information is used to keep track of the contents of *resources*, such as registers or memory cells, in the assembler code. This information is used by the IAR C-SPY Debugger to go “back” in the call stack and show the correct values of registers or other resources before entering the function. In contrast with traditional approaches, this permits the debugger to run at full speed until it reaches a breakpoint, stop at the breakpoint, and retrieve backtrace information at that point in the program. The information can then be used to compute the contents of the resources in any of the calling functions—assuming they have call frame information as well.

### Backtrace rows and columns

At each location in the program where it is possible for the debugger to break execution, there is a *backtrace row*. Each backtrace row consists of a set of *columns*, where each column represents an item that should be tracked. There are three kinds of columns:

- The *resource columns* keep track of where the original value of a resource can be found.
- The canonical frame address columns (*CFA columns*) keep track of the top of the function frames.
- The *return address column* keeps track of the location of the return address.

There is always exactly one return address column and usually only one CFA column, although there may be more than one.

### Defining a names block

A *names block* is used to declare 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:

```
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 may appear: a resource declaration, a stack frame declaration, a static overlay frame declaration, or a base address declaration:

- To declare a resource, use one of the directives:

```
CFI RESOURCE resource : bits
CFI VIRTUALRESOURCE resource : bits
```

The parameters are the name of the resource and the size of the resource in bits. 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.

More than one resource can be declared by separating them with commas.

A resource may also be a composite resource, made up of at least two parts. To declare the composition of a composite resource, use the directive:

```
CFI RESOURCEPARTS resource part, 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:

```
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 segment type (to get the address space). More than one stack frame CFA can be declared by separating 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.

- To declare a static overlay frame CFA, use the directive:

```
CFI STATICOVERLAYFRAME cfa segment
```

The parameters are the name of the CFA and the name of the segment where the static overlay for the function is located. More than one static overlay frame CFA can be declared by separating them with commas.

- To declare a base address CFA, use the directive:

```
CFI BASEADDRESS cfa type
```

The parameters are the name of the CFA and the segment type. More than one base address CFA can be declared by separating them with commas.

A base address CFA is used to conveniently handle a CFA. In contrast to the stack frame CFA, there is no associated stack pointer resource to restore.

### Extending a names block

In some special cases you have to extend an existing names block with new resources. This occurs whenever there are routines that manipulate call frames other than their own, such as routines for handling, entering, and leaving C or C++ functions; these routines manipulate the caller's frame. Extended names blocks are normally used only by compiler developers.

Extend an existing names block with the directive:

```
CFI NAMES name EXTENDS namesblock
```

where *namesblock* is the name of the existing names block and *name* is the name of the new extended block. The extended block must end with the directive:

```
CFI ENDNAMES name
```

### 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:

```
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:

```
CFI RETURNADDRESS resource type
```

where *resource* is a resource defined in *namesblock* and *type* is the segment type. You have to declare the return address column for the common block.

End a common block with the directive:

```
CFI ENDCOMMON name
```

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

Inside a common block you can declare the initial value of a CFA or a resource by using the directives listed last in *Common block directives*, page 84. For more information on these directives, see *Simple rules*, page 90, and *CFI expressions*, page 92.

## Extending a common block

Since you can extend a names block with new resources, it is necessary to have a mechanism for describing the initial values of these new resources. For this reason, it is also possible to extend common blocks, effectively declaring the initial values of the extra resources while including the declarations of another common block. Just as in the case of extended names blocks, extended common blocks are normally only used by compiler developers.

Extend an existing common block with the directive:

```
CFI COMMON name EXTENDS commonblock USING namesblock
```

where *name* is the name of the new extended block, *commonblock* is the name of the existing common block, and *namesblock* is the name of a previously defined names block. The extended block must end with the directive:

```
CFI ENDCOMMON name
```

## Defining a data block

The *data block* contains the actual tracking information for one continuous piece of code. No segment control directive may appear inside a data block.

Start a data block with the directive:

```
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 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 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 may manipulate the values of the columns by using the directives listed last in *Data block directives*, page 84. For more information on these directives, see *Simple rules*, page 90, and *CFI expressions*, page 92.

## SIMPLE RULES

To describe the tracking information for individual columns, there is a set of simple rules with specialized syntax:

```
CFI cfa { NOTUSED | USED }
CFI cfa { resource | resource + constant | resource - constant }
CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
CFI resource { resource | FRAME(cfa, offset) }
```

These simple rules can be used 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, a full CFI expression can be used to describe the information (see *CFI expressions*, page 92). However, whenever possible, you should always use a simple rule instead of a CFI expression.

There are two different sets of simple rules: one for resources and one for CFAs.

### Simple rules for 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, that is, already correctly located, use `SAMEVALUE` as the location. Conceptually, this declares that the resource does not have to be restored since it already contains the correct value. For example, to declare that a register `REG` is restored to the same value, use the directive:

```
CFI REG 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) since 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 `REG` is a scratch register and does not have to be restored, use the directive:

```
CFI REG 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 `REG1` is temporarily located in a register `REG2` (and should be restored from that register), use the directive:

```
CFI REG1 REG2
```



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 `REG` is located at offset `-4` counting from the frame pointer `CFA_SP`, use the directive:

```
CFI REG FRAME(CFA_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:

```
CFI RET CONCAT
```

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

### Simple rules for CFAs

In contrast with 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 subroutine calling instruction. The CFA rules describe how to compute the address to the beginning of the current call frame. There are two different forms of CFAs, stack frames and static overlay frames, each declared in the associated names block. See *Names block directives*, page 84.

Each stack frame CFA is associated with a resource, such as the stack pointer. When going back one call frame the associated resource is restored to the current CFA. For stack frame CFAs there are two possible simple 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 resource 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 resource 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
```

For static overlay frame CFAs, there are only two possible declarations inside common and data blocks: `USED` and `NOTUSED`.

## CFI EXPRESSIONS

Call frame information expressions (CFI expressions) can be used when the descriptive power of the simple rules for resources and CFAs is not enough. However, you should always use a simple rule when one is available.

CFI expressions consist of operands and operators. Only the operators described below are allowed in a CFI expression. In most cases, they have an equivalent operator in the regular assembler expressions.

In the operand descriptions, *cfiexpr* denotes one of the following:

- A CFI operator with operands
- A numeric constant
- A CFA name
- A resource name.

### Unary operators

Overall syntax: *OPERATOR*(*operand*)

Operator	Operand	Description
UMINUS	<i>cfiexpr</i>	Performs arithmetic negation on a CFI expression.
NOT	<i>cfiexpr</i>	Negates a logical CFI expression.
COMPLEMENT	<i>cfiexpr</i>	Performs a bitwise NOT on a CFI expression.
LITERAL	<i>expr</i>	Get the value of the assembler expression. This can insert the value of a regular assembler expression into a CFI expression.

Table 26: Unary operators in CFI expressions

### Binary operators

Overall syntax: *OPERATOR*(*operand1*, *operand2*)

Operator	Operands	Description
ADD	<i>cfiexpr</i> , <i>cfiexpr</i>	Addition
SUB	<i>cfiexpr</i> , <i>cfiexpr</i>	Subtraction
MUL	<i>cfiexpr</i> , <i>cfiexpr</i>	Multiplication
DIV	<i>cfiexpr</i> , <i>cfiexpr</i>	Division
MOD	<i>cfiexpr</i> , <i>cfiexpr</i>	Modulo
AND	<i>cfiexpr</i> , <i>cfiexpr</i>	Bitwise AND
OR	<i>cfiexpr</i> , <i>cfiexpr</i>	Bitwise OR

Table 27: Binary operators in CFI expressions

Operator	Operands	Description
XOR	<i>cfiexpr, cfiexpr</i>	Bitwise XOR
EQ	<i>cfiexpr, cfiexpr</i>	Equal
NE	<i>cfiexpr, cfiexpr</i>	Not equal
LT	<i>cfiexpr, cfiexpr</i>	Less than
LE	<i>cfiexpr, cfiexpr</i>	Less than or equal
GT	<i>cfiexpr, cfiexpr</i>	Greater than
GE	<i>cfiexpr, cfiexpr</i>	Greater than or equal
LSHIFT	<i>cfiexpr, cfiexpr</i>	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.
RSHIFTL	<i>cfiexpr, cfiexpr</i>	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.
RSHIFTA	<i>cfiexpr, cfiexpr</i>	Arithmetic shift right of the left operand. The number of bits to shift is specified by the right operand. In contrast with RSHIFTL the sign bit will be preserved when shifting.

Table 27: Binary operators in CFI expressions (Continued)

## Ternary operators

Overall syntax: *OPERATOR(operand1, operand2, operand3)*

Operator	Operands	Description
FRAME	<i>cfa, size, offset</i>	Gets the value from a stack frame. The operands are: <i>cfa</i> An identifier denoting a previously declared CFA. <i>size</i> A constant expression denoting a size in bytes. <i>offset</i> A constant expression denoting an offset in bytes. Gets the value at address <i>cfa+offset</i> of size <i>size</i> .
IF	<i>cond, true, false</i>	Conditional operator. The operands are: <i>cond</i> A CFA expression denoting a condition. <i>true</i> Any CFA expression. <i>false</i> Any CFA expression. If the conditional expression is non-zero, the result is the value of the <i>true</i> expression; otherwise the result is the value of the <i>false</i> expression.

Table 28: Ternary operators in CFI expressions

Operator	Operands	Description
LOAD	<i>size, type, addr</i>	Gets the value from memory. The operands are: <i>size</i> A constant expression denoting a size in bytes. <i>type</i> A memory type. <i>addr</i> A CFA expression denoting a memory address. Gets the value at address <i>addr</i> in segment type <i>type</i> of size <i>size</i> .

Table 28: Ternary operators in CFI expressions (Continued)

## EXAMPLE

The following is a generic example and not an example specific to the HCS12 microcontroller. This will simplify the example and clarify the usage of the CFI directives. A target-specific example can be obtained by generating assembler output when compiling a C source file.

Consider a generic processor with a stack pointer *SP*, and two registers *R0* and *R1*. Register *R0* will be used as a scratch register (the register is destroyed by the function call), whereas register *R1* has to be restored after the function call. For reasons of simplicity, all instructions, registers, and addresses will have a width of 16 bits.

Consider the following short code sample with the corresponding backtrace rows and columns. At entry, assume that the stack contains a 16-bit return address. The stack grows from high addresses towards zero. The CFA denotes the top of the call frame, that is, the value of the stack pointer after returning from the function.

Address	CFA	SP	R0	R1	RET	Assembler code
0000	SP + 2		—	SAME	CFA - 2	func1: PUSH R1
0002	SP + 4			CFA - 4		MOV R1, #4
0004						CALL func2
0006						POP R0
0008	SP + 2			R0		MOV R1, R0
000A				SAME		RET

Table 29: Code sample with backtrace rows and columns

Each backtrace 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 backtrace row at address 0000 is the initial row and the result of the calling convention used for the function.

The SP column is empty since the CFA is defined in terms of the stack pointer. The RET column is the return address column—that is, the location of the return address. The R0 column has a ‘—’ in the first line to indicate that the value of R0 is undefined and 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:

```
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 CFA SP + 2
CFI R0 UNDEFINED
CFI R1 SAMEVALUE
CFI RET FRAME(CFA,-2) ; Offset -2 from top of frame
CFI ENDCOMMON trivialCommon
```

**Note:** SP may not be changed using a CFI directive since it is the resource associated with CFA.

### Defining the data block

Continuing the simple example, the data block would be:

```

RSEG CODE:CODE
CFI BLOCK func1block USING trivialCommon
CFI FUNCTION func1
func1:
PUSH R1
CFI CFA SP + 4
CFI R1 FRAME(CFA,-4)
MOV R1,#4
CALL func2
POP R0
CFI R1 R0
```

```
CFI    CFA SP + 2
MOV    R1,R0
CFI    R1 SAMEVALUE
RET
CFI ENDBLOCK func1block
```

Note that the CFI directives are placed *after* the instruction that affects the backtrace information.

# Pragma directives

This chapter describes the pragma directives of the IAR Assembler for HCS12.

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

The following table shows the pragma directives of the assembler:

<b>#pragma directive</b>	<b>Description</b>
#pragma diag_default	Changes the severity level of diagnostic messages
#pragma diag_error	Changes the severity level of diagnostic messages
#pragma diag_remark	Changes the severity level of diagnostic messages
#pragma diag_suppress	Suppresses diagnostic messages
#pragma diag_warning	Changes the severity level of diagnostic messages
#pragma message	Prints a message

*Table 30: Pragma directives summary*

---

## Descriptions of pragma directives

All pragma directives using = for value assignment should be entered like:

```
#pragma pragmaname=pragmavalue
```

or

```
#pragma pragmaname = pragmavalue
```

---

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

Changes the severity level back to default or as defined on the command line for the diagnostic messages with the specified tags. For example:

```
#pragma diag_default=Pe117
```

See the chapter *Diagnostics* for more information about diagnostic messages.

---

<code>#pragma diag_error</code>	<p><code>#pragma diag_error=tag, tag, ...</code></p> <p>Changes the severity level to <code>error</code> for the specified diagnostics. For example:</p> <pre>#pragma diag_error=Pe117</pre> <p>See the chapter <i>Diagnostics</i> for more information about diagnostic messages.</p>
<hr/>	
<code>#pragma diag_remark</code>	<p><code>#pragma diag_remark=tag, tag, ...</code></p> <p>Changes the severity level to <code>remark</code> for the specified diagnostics. For example:</p> <pre>#pragma diag_remark=Pe177</pre> <p>See the chapter <i>Diagnostics</i> for more information about diagnostic messages.</p>
<hr/>	
<code>#pragma diag_suppress</code>	<p><code>#pragma diag_suppress=tag, tag, ...</code></p> <p>Suppresses the diagnostic messages with the specified tags. For example:</p> <pre>#pragma diag_suppress=Pe117, Pe177</pre> <p>See the chapter <i>Diagnostics</i> for more information about diagnostic messages.</p>
<hr/>	
<code>#pragma diag_warning</code>	<p><code>#pragma diag_warning=tag, tag, ...</code></p> <p>Changes the severity level to <code>warning</code> for the specified diagnostics. For example:</p> <pre>#pragma diag_warning=Pe826</pre> <p>See the chapter <i>Diagnostics</i> for more information about diagnostic messages.</p>
<hr/>	
<code>#pragma message</code>	<p><code>#pragma message(string)</code></p> <p>Makes the assembler print a message on <code>stdout</code> when the file is assembled. For example:</p> <pre>#ifdef TESTING #pragma message("Testing") #endif</pre>



# Diagnostics

This chapter describes 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, as well as 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 27.

### 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 by use of the command-line option `--no_warnings`, see `--no_warnings`, page 25.

### Error

A diagnostic message that is produced when the assembler has found a construct which clearly violates the language rules, such that code cannot be produced. An error will produce a non-zero exit code.

### Fatal error

A diagnostic message that is produced when the assembler has found a condition that not only prevents code generation, but which makes further processing of the source code pointless. After the diagnostic has been issued, compilation terminates. A fatal error will produce 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.

See *Summary of assembler options*, page 15, for a description of the assembler options that are available for setting severity levels.

See the chapter *Pragma directives*, for a description of the pragma directives that are available for setting severity levels.

### INTERNAL ERROR

An internal error is a diagnostic message that signals that there has been a serious and unexpected failure due to a fault in the assembler. It is produced using the following form:

```
Internal error: message
```

where *message* is an explanatory message. If internal errors occur, they should be reported to your software distributor or IAR 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.

# Migrating assembler code

This chapter presents the major differences between the IAR Assembler for HCS12 version 3.20 and the IAR Assembler for HCS12 version 3.10, and describes the migration considerations.

---

## The migration process

In short, to migrate from version 3.10 to 3.20, you must consider the following:

- Assembler options
- Assembler operators
- Assembler directives.

To migrate your old project, follow the described migration process. Note that not all steps in the described migration process may be relevant for your project. Consider carefully what actions are needed in your case.

### ASSEMBLER OPTIONS

The command line options in version 3.20 follow two different syntax styles:

- Long option names containing one or more words prefixed with two dashes, and sometimes followed by an equal sign and a modifier, for example `--strict_ansi` and `--module_name=test`.
- Short option names consisting of a single letter prefixed with a single dash, and sometimes followed by a modifier, for example `-r`.

Some options appear in one style only, while other options appear as synonyms in both styles. A number of new command line options have been added. For a complete list of the available command line options, see the chapter *Assembler options*.

### Migrating project options

Because the available assembler options differ between version 3.20 and version 3.10, you should verify your option settings after you have converted an old project.



If you are using the command line interface, you can simply compare your makefile with the options listed in this chapter and modify the makefile accordingly.



If you are using the IAR Embedded Workbench IDE, many option settings are automatically converted during the project conversion. This means that you must verify the options manually.

For details about changes related to options, see Table 31, *Version 3.10 compiler options not available in version 3.20* and Table 32, *Renamed or modified options*.

## Removed options

The following table lists the command line options that have been removed:

Old option	Description
-B	Macro execution information
-b	Makes a library module
-h	Enables the use of space ( ' ) as the character for starting a comment
-N	Omits header from assembler listing
-O <i>prefix</i>	Sets object filename prefix
-p <i>lines</i>	Lines per page
-tn	Tab spacing
-U <i>symbol</i>	Undefines a symbol
-u	Use A6801 operators

Table 31: Version 3.10 compiler options not available in version 3.20

## Renamed or modified options

The following version 2.x command line options have been *renamed* and/or *modified*:

Old option	New option	Description
-c {DSMEAOC}	-l	Conditional listing; some variants are gone
-Enumber	--error_limit	Specifies the maximum of errors before the assembler stops
-G	-	Input from <code>stdin</code> *
-L[ <i>prefix</i> ], -l <i>filename</i>	-l[a d e m o x][N] { <i>filename directory</i> }	Generates list file; the modifiers specify the type of list file to create
-n	--enable_multibytes	Enables support for multibyte characters
-r[r][e]	-r, --debug	Generates debug information; the modifiers have been removed
-S	--silent	Sets silent operation
-s{+ -}	--case_insensitive	Case sensitive user symbols off
-w [ <i>string</i> ][s]	--no_warnings, --diag_suppress	Disables warnings

Table 32: Renamed or modified options

\* Specifying the character - (dash) on the command line using version 3.20 indicates input is taken from `stdin`, see *Specifying parameters*, page 14.

## ASSEMBLER OPERATORS

All assembler operators that are available in version 3.10 are also available in version 3.20. However, the following differences apply:

- The operator precedence differs between version 3.20 and version 3.10
- The following operators have new names formed by removing the dots around the old name, so that for example `.OPERATOR.` becomes just `OPERATOR`:

```
.NOT., .LOW., .HIGH., .LWRD., .HWRD., .DATE., .SFB., .SFE., .SIZEOF.,
.BINNOT., .MOD., .SHR., .SHL., .AND., .BINAND., .OR., .BINOR., .XOR.,
.BINXOR., .EQ., .NE., .GT., .LT., .UGT., .ULT., .GE., .LE.
```

- The operators `.BYT2.` and `.BYT3.` have become `BYTE2` and `BYTE3`, respectively

The old operators are still recognized by the assembler but they are obsolete, and it is recommended to change them to the new syntax.

For detailed information about operators in version 3.20, see the chapter *Assembler operators*, page 29.

## ASSEMBLER DIRECTIVES

There are some changes in version 3.20 related to available assembler directives in version 3.10.

### Removed assembler directives

The following directives have been removed:

- `CYCMAX`, `CYCMEAN`, `CYCMIN`, `CYCLES`
- All assembler directives related to structured assembly
- `sfrb`, `sfrtype`, `sfrw`
- `DCB`.

### Renamed or modified assembler directives

The following version 3.10 assembler directives have been *renamed* and/or *modified*:

In version 3.10	In version 3.20
<code>\$.INCLUDE</code>	<code>#include</code>
<code>#message</code>	<code>#pragma message</code>
<code>IFNC</code>	<code>IF s == t</code>
<code>IFxx</code>	<code>IF a &lt;relop&gt; b</code>
<code>STACK</code>	<code>RSEG</code> , and use <code>#</code> in the linker command file to allocate from the top

Table 33: Renamed or modified assembler directives

### Obsolete assembler directives

The following version 3.10 assembler directives are still recognized by the assembler, but they are obsolete. It is recommended to change them to their new equivalents:

In version 3.10	In version 3.20
ALIAS	=, EQU
DC . B, FCB, DC, FCC	DC8
DC . W, FDB	DC16
DC . L, FQB	DC32
DS . B, RMB, DS	DS8
DS . W	DS16
DS . L	DS32
ELSEC	ELSE
ENDMAC	ENDM
ENDS	ENDIF
EXPORT	PUBLIC
IMPORT	EXTERN

*Table 34: Obsolete assembler directives*

# A

absolute expressions . . . . .	8	DC8 . . . . .	77
absolute segments . . . . .	55	DC16 . . . . .	77
ADD (CFI operator) . . . . .	92	DC24 . . . . .	77
address field, in assembler list file . . . . .	10	DC32 . . . . .	77
AHCS12_INC (environment variable) . . . . .	15	DC64 . . . . .	77
ALIGN (assembler directive) . . . . .	53	DEFINE . . . . .	58
alignment, of segments . . . . .	56	DF32 . . . . .	77
ALIGNRAM (assembler directive) . . . . .	53	DF64 . . . . .	77
AND (assembler operator) . . . . .	34	DS8 . . . . .	77
AND (CFI operator) . . . . .	92	DS16 . . . . .	77
architecture, HCS12 . . . . .	xi	DS24 . . . . .	77
ARGFRAME (assembler directive) . . . . .	82	DS32 . . . . .	77
_args (predefined macro symbol) . . . . .	65	DS64 . . . . .	77
ASCII character constants . . . . .	4	ELSE . . . . .	61
ASEG (assembler directive) . . . . .	53	ELSEIF . . . . .	61
ASEGN (assembler directive) . . . . .	53	END . . . . .	48
asm (filename extension) . . . . .	3	ENDIF . . . . .	61
ASMHCS12 (environment variable) . . . . .	15	ENDM . . . . .	63
assembler control directives . . . . .	80	ENDMOD . . . . .	48
assembler diagnostics . . . . .	99	ENDR . . . . .	63
assembler directives		EQU . . . . .	58
ALIGN . . . . .	53	EVEN . . . . .	53
ALIGNRAM . . . . .	53	EXITM . . . . .	63
ARGFRAME . . . . .	82	EXTERN . . . . .	51
ASEG . . . . .	53	FUNCALL . . . . .	82
ASEGN . . . . .	53	FUNCTION . . . . .	82
assembler control . . . . .	80	function . . . . .	81
ASSIGN . . . . .	58	IF . . . . .	61
call frame information . . . . .	83	LIBRARY . . . . .	48
CASEOFF . . . . .	80	LIMIT . . . . .	58
CASEON . . . . .	80	list file control . . . . .	69
CFI directives . . . . .	83	LOCAL . . . . .	63
COMMON . . . . .	53	LOCFRAME . . . . .	82
conditional assembly . . . . .	61	LSTCND . . . . .	69
<i>See also</i> C-style preprocessor directives		LSTCOD . . . . .	69
C-style preprocessor . . . . .	73	LSTEXP . . . . .	69
data definition or allocation . . . . .	77	LSTMAC . . . . .	69
		LSTOUT . . . . .	69
		LSTREP . . . . .	69

LSTXRF .....	69	assembler environment variables .....	14
MACRO .....	63	assembler error return codes .....	15
macro processing .....	63	assembler instructions .....	3
MODULE .....	48	assembler labels .....	5
module control .....	48	format of .....	2
NAME .....	48	assembler list files	
ODD .....	53	address field .....	10
ORG .....	53	comments .....	80
PROGRAM .....	48	conditional code and strings .....	70
PUBLIC .....	51	cross-references, generating .....	23, 71
PUBWEAK .....	51	data field .....	10
RADIX .....	80	disabling .....	70
REPT .....	63	enabling .....	70
REPTC .....	63	filename, specifying .....	23
REPTI .....	63	generated lines, controlling .....	70
REQUIRE .....	51	macro-generated lines, controlling .....	70
RSEG .....	53	symbol and cross-reference table .....	10
RTMODEL .....	48	assembler macros	
segment control .....	53	arguments, passing to .....	65
SET .....	58	defining .....	64
summary .....	45	generated lines, controlling in list file .....	70
symbol control .....	51	in-line routines .....	67
value assignment .....	58	predefined symbol .....	65
VAR .....	58	processing .....	66
/*...*/ .....	80	quote characters, specifying .....	24
// .....	80	special characters, using .....	65
#define .....	73	assembler operators .....	29
#elif .....	73	AND .....	34
#else .....	73	BINAND .....	34
#endif .....	73	BINNOT .....	34
#error .....	73	BINOR .....	35
#if .....	73	BINXOR .....	35
#ifdef .....	73	BYTE1 .....	35
#ifndef .....	73	BYTE2 .....	35
#include .....	73	BYTE3 .....	36
#line .....	73	BYTE4 .....	36
#pragma .....	73, 97	DATE .....	36
#undef .....	73	EQ .....	36
= .....	58	GE .....	37



GT	37
HIGH	37
HWRD	37
in expressions	3
LE	38
LOW	38
LT	38
LWRD	38
MOD	39
NE	39
NOT	39
OR	39
precedence	29
SFB	40
SFE	40
SHL	41
SHR	41
SIZEOF	41
UGT	42
ULT	42
UPPER	42
XOR	43
^	35
-	33
!	39
!=	39
?:	34
()	32
*	32
/	33
&	34
&&	34
%	39
+	32–33
<	38
<<	41
<=	38
<>	39
=	36
==	36
>	37
>=	37
>>	41
	35
	39
~	34
assembler options	
specifying parameters	14
summary	15
typographic convention	xiii
-D	17
-f	22
-I	22
-l	23
-M	24
-o	25
-r	27
--case_insensitive	17
--core	17
--debug	18
--dependencies	18
--diagnostics_tables	20
--diag_error	19
--diag_remark	20
--diag_suppress	20
--diag_warning	20
--dir_first	21
--enable_multibytes	21
--header_context	22
--mnem_first	24
--no_path_in_file_macros	25
--no_warnings	25
--only_stdout	26
--preprocess	26
--remarks	27
--silent	27
--warnings_affect_exit_code	15, 28
--warnings_are_errors	28

assembler output, including debug information	18, 27
assembler source files, including	75
assembler source format	2
assembler subversion number	7
assembler symbols	5
exporting	52
importing	52
in relocatable expressions	8
local	60
predefined	7
redefining	59
ASSIGN (assembler directive)	58
assumptions (programming experience)	xi

## B

backtrace information, defining	83
BINAND (assembler operator)	34
BINNOT (assembler operator)	34
BINOR (assembler operator)	35
BINXOR (assembler operator)	35
__BUILD_NUMBER__ (predefined symbol)	7
BYTE1 (assembler operator)	35
BYTE2 (assembler operator)	35
BYTE3 (assembler operator)	36
BYTE4 (assembler operator)	36

## C

call frame information directives	83
case sensitivity, controlling	17, 80
CASEOFF (assembler directive)	80
CASEON (assembler directive)	80
--case_insensitive (assembler option)	17
CFI directives	83
CFI expressions	92
CFI operators	92
character constants, ASCII	4
command line, extending	22

comments	76
in assembler list file	80
in assembler source code	2
multi-line, using with assembler directives	81
common segments	55
COMMON (assembler directive)	53
compiler object file, specifying filename	25
compiler options	
migrating from old compiler version	101
--error_limit	22
--no_wrap_diagnostics	25
--preinclude	26
COMPLEMENT (CFI operator)	92
computer style, typographic convention	xiii
conditional assembly directives	61
<i>See also</i> C-style preprocessor directives	
conditional code and strings, listing	70
constants, default base of	80
constants, integer	4
conventions, typographic	xiii
copyright notice	ii
__CORE__ (predefined symbol)	7
--core (assembler option)	17
CRC, in assembler list file	10
cross-references, in assembler list file	23, 71
C-style preprocessor directives	73

## D

-D (assembler option)	17
data allocation directives	77
data definition directives	77
data field, in assembler list file	10
__DATE__ (predefined symbol)	7
DATE (assembler operator)	36
DC8 (assembler directive)	77
DC16 (assembler directive)	77
DC24 (assembler directive)	77
DC32 (assembler directive)	77

DC64 (assembler directive) . . . . . 77

--debug (assembler option) . . . . . 18

debug information, including in assembler output . . . . 18, 27

default base, for constants . . . . . 80

#define (assembler directive) . . . . . 73

DEFINE (assembler directive) . . . . . 58

--dependencies (assembler option) . . . . . 18

DF32 (assembler directive) . . . . . 77

DF64 (assembler directive) . . . . . 77

diagnostic messages . . . . . 99

    classifying as errors . . . . . 19

    classifying as remarks . . . . . 20

    classifying as warnings . . . . . 20

    disabling warnings . . . . . 25

    disabling wrapping of . . . . . 25

    enabling remarks . . . . . 27

    listing all . . . . . 20

    suppressing . . . . . 20

--diagnostics\_tables (assembler option) . . . . . 20

diag\_default (#pragma directive) . . . . . 97

--diag\_error (assembler option) . . . . . 19

diag\_error (#pragma directive) . . . . . 98

--diag\_remark (assembler option) . . . . . 20

diag\_remark (#pragma directive) . . . . . 98

--diag\_suppress (assembler option) . . . . . 20

diag\_suppress (#pragma directive) . . . . . 98

--diag\_warning (assembler option) . . . . . 20

diag\_warning (#pragma directive) . . . . . 98

directives. *See* assembler directives

--dir\_first (assembler option) . . . . . 21

disclaimer . . . . . ii

DIV (CFI operator) . . . . . 92

document conventions . . . . . xiii

DS8 (assembler directive) . . . . . 77

DS16 (assembler directive) . . . . . 77

DS24 (assembler directive) . . . . . 77

DS32 (assembler directive) . . . . . 77

DS64 (assembler directive) . . . . . 77

## E

edition, of this guide . . . . . ii

efficient coding techniques . . . . . 11

#elif (assembler directive) . . . . . 73

#else (assembler directive) . . . . . 73

ELSE (assembler directive) . . . . . 61

ELSEIF (assembler directive) . . . . . 61

--enable\_multibytes (assembler option) . . . . . 21

END (assembler directive) . . . . . 48

#endif (assembler directive) . . . . . 73

ENDIF (assembler directive) . . . . . 61

ENDM (assembler directive) . . . . . 63

ENDMOD (assembler directive) . . . . . 48

ENDR (assembler directive) . . . . . 63

environment variables . . . . . 14

    AHCS12\_INC . . . . . 15

    ASMHCS12 . . . . . 15

EQ (assembler operator) . . . . . 36

EQ (CFI operator) . . . . . 93

EQU (assembler directive) . . . . . 58

#error (assembler directive) . . . . . 73

error messages . . . . . 99

    classifying . . . . . 19

    #error, using to display . . . . . 76

error return codes . . . . . 15

EVEN (assembler directive) . . . . . 53

EXITM (assembler directive) . . . . . 63

experience, programming . . . . . xi

expressions . . . . . 3

expressions. *See* assembler expressions

extended command line file (extend.xcl) . . . . . 22

EXTERN (assembler directive) . . . . . 51

## F

-f (assembler option) . . . . . 22

false value, in assembler expressions . . . . . 5

fatal error messages . . . . . 100

__FILE__ (predefined symbol) . . . . .	7
file dependencies, tracking . . . . .	18
file extensions. <i>See</i> filename extensions	
file types	
assembler source . . . . .	3
extended command line . . . . .	22
#include, specifying path . . . . .	22
filename extensions	
asm . . . . .	3
msa . . . . .	3
s12 . . . . .	3
xcl . . . . .	22
filenames, specifying for assembler output . . . . .	25
filename, of object file . . . . .	25
formats, assembler source code. . . . .	2
FRAME (CFI operator) . . . . .	93
FUNCALL (assembler directive) . . . . .	82
function directives. . . . .	81
FUNCTION (assembler directive) . . . . .	82

## G

GE (assembler operator) . . . . .	37
GE (CFI operator) . . . . .	93
global value, defining . . . . .	59
glossary. . . . .	xi
GT (assembler operator) . . . . .	37
GT (CFI operator) . . . . .	93

## H

HCS12 architecture and instruction set. . . . .	xi
header files, SFR. . . . .	11
--header_context (assembler option) . . . . .	22
HIGH (assembler operator) . . . . .	37
hints for migration . . . . .	101
HWRD (assembler operator) . . . . .	37

## I

-I (assembler option) . . . . .	22
IAR Technical Support . . . . .	100
__IAR_SYSTEMS_ASM__ (predefined symbol) . . . . .	7
#if (assembler directive) . . . . .	73
IF (assembler directive) . . . . .	61
IF (CFI operator) . . . . .	93
#ifdef (assembler directive) . . . . .	73
#ifndef (assembler directive) . . . . .	73
#include files, specifying . . . . .	22
#include (assembler directive) . . . . .	73
include paths, specifying. . . . .	22
instruction set, HCS12 . . . . .	xi
integer constants . . . . .	4
internal error . . . . .	100
in-line coding, using macros . . . . .	67

## L

-l (assembler option) . . . . .	23
labels. <i>See</i> assembler labels	
LE (assembler operator) . . . . .	38
LE (CFI operator) . . . . .	93
library modules . . . . .	49
LIBRARY (assembler directive) . . . . .	48
LIMIT (assembler directive) . . . . .	58
__LINE__ (predefined symbol) . . . . .	7
#line (assembler directive) . . . . .	73
list file format . . . . .	10
body. . . . .	10
CRC. . . . .	10
header . . . . .	10
symbol and cross reference . . . . .	10
listing control directives . . . . .	69
LITERAL (CFI operator) . . . . .	92
LOAD (CFI operator) . . . . .	94
local value, defining . . . . .	59
LOCAL (assembler directive) . . . . .	63

location counter. <i>See</i> program location counter	
LOCFRAME (assembler directive) . . . . .	82
LOW (assembler operator) . . . . .	38
LSHIFT (CFI operator) . . . . .	93
LSTCND (assembler directive) . . . . .	69
LSTCOD (assembler directive) . . . . .	69
LSTEXP (assembler directives) . . . . .	69
LSTMAC (assembler directive) . . . . .	69
LSTOUT (assembler directive) . . . . .	69
LSTREP (assembler directive) . . . . .	69
LSTXRF (assembler directive) . . . . .	69
LT (assembler operator) . . . . .	38
LT (CFI operator) . . . . .	93
LWRD (assembler operator) . . . . .	38

## M

-M (assembler option) . . . . .	24
macro processing directives . . . . .	63
macro quote characters . . . . .	65
specifying . . . . .	24
MACRO (assembler directive) . . . . .	63
macros. <i>See</i> assembler macros	
memory space, reserving and initializing . . . . .	78
memory, reserving space in . . . . .	77
message (#pragma directive) . . . . .	98
messages, excluding from standard output stream . . . . .	27
--mnem_first (assembler option) . . . . .	24
MOD (assembler operator) . . . . .	39
MOD (CFI operator) . . . . .	92
module consistency . . . . .	50
module control directives . . . . .	48
MODULE (assembler directive) . . . . .	48
modules	
assembling multi-modules files . . . . .	50
terminating . . . . .	49
msa (filename extension) . . . . .	3
MUL (CFI operator) . . . . .	92
multibyte character support . . . . .	21

## N

NAME (assembler directive) . . . . .	48
NE (assembler operator) . . . . .	39
NE (CFI operator) . . . . .	93
NOT (assembler operator) . . . . .	39
NOT (CFI operator) . . . . .	92
--no_path_in_file_macros (assembler option) . . . . .	25
--no_warnings (assembler option) . . . . .	25
--no_wrap_diagnostics (compiler option) . . . . .	25

## O

-o (assembler option) . . . . .	25
ODD (assembler directive) . . . . .	53
--only_stdout (assembler option) . . . . .	26
operands	
format of . . . . .	2
in assembler expressions . . . . .	3
operations, format of . . . . .	2
operation, silent . . . . .	27
operators. <i>See</i> assembler operators	
option summary . . . . .	15
OR (assembler operator) . . . . .	39
OR (CFI operator) . . . . .	92
ORG (assembler directive) . . . . .	53

## P

parameters	
specifying . . . . .	14
typographic convention . . . . .	xiii
part number, of this guide . . . . .	ii
PLC. <i>See</i> program location counter	
porting, of code . . . . .	101
#pragma (assembler directive) . . . . .	73, 97
precedence, of assembler operators . . . . .	29
predefined register symbols . . . . .	6

predefined symbols . . . . .	7
in assembler macros . . . . .	65
__BUILD_NUMBER__ . . . . .	7
__CORE__ . . . . .	7
__DATE__ . . . . .	7
__FILE__ . . . . .	7
__IAR_SYSTEMS_ASM__ . . . . .	7
__LINE__ . . . . .	7
__SUBVERSION__ . . . . .	7
__TID__ . . . . .	7
__TIME__ . . . . .	7
__VER__ . . . . .	7
--preinclude (compiler option) . . . . .	26
--preprocess (assembler option) . . . . .	26
preprocessor symbols	
defining and undefining . . . . .	74
defining on command line . . . . .	17
prerequisites (programming experience) . . . . .	xi
program counter. <i>See</i> program location counter	
program location counter (PLC) . . . . .	6
setting . . . . .	56
program modules, beginning . . . . .	49
PROGRAM (assembler directive) . . . . .	48
programming experience, required . . . . .	xi
programming hints . . . . .	11
PUBLIC (assembler directive) . . . . .	51
publication date, of this guide . . . . .	ii
PUBWEAK (assembler directive) . . . . .	51

## R

-r (assembler option) . . . . .	27
RADIX (assembler directive) . . . . .	80
reference information, typographic convention . . . . .	xiii
registered trademarks . . . . .	ii
registers . . . . .	6
relocatable expressions . . . . .	8
relocatable segments, beginning . . . . .	55

remark (diagnostic message) . . . . .	99
classifying . . . . .	20
enabling . . . . .	27
--remarks (assembler option) . . . . .	27
repeating statements . . . . .	67
REPT (assembler directive) . . . . .	63
REPTC (assembler directive) . . . . .	63
REPTI (assembler directive) . . . . .	63
REQUIRE (assembler directive) . . . . .	51
RSEG (assembler directive) . . . . .	53
RSHIFTA (CFI operator) . . . . .	93
RSHIFTL (CFI operator) . . . . .	93
RTMODEL (assembler directive) . . . . .	48
rules, in CFI directives . . . . .	90
runtime model attributes, declaring . . . . .	50

## S

segment control directives . . . . .	53
segments	
absolute . . . . .	55
aligning . . . . .	56
common, beginning . . . . .	55
relocatable . . . . .	55
SET (assembler directive) . . . . .	58
severity level, of diagnostic messages . . . . .	99
specifying . . . . .	100
SFB (assembler operator) . . . . .	40
SFE (assembler operator) . . . . .	40
SFR. <i>See</i> special function registers	
SHL (assembler operator) . . . . .	41
SHR (assembler operator) . . . . .	41
--silent (assembler option) . . . . .	27
silent operation, specifying . . . . .	27
simple rules, in CFI directives . . . . .	90
SIZEOF (assembler operator) . . . . .	41
source files	
including . . . . .	75
list all referred . . . . .	22

source format, assembler	2
special function registers	11
standard error	26
standard output stream, disabling messages to	27
standard output, specifying	26
statements, repeating	67
stderr	26
stdout	26
SUB (CFI operator)	92
__SUBVERSION__ (predefined symbol)	7
Support, Technical	100
symbol and cross-reference table, in assembler list file	10
<i>See also</i> Include cross-reference	10
symbol control directives	51
symbol values, checking	59
symbols	
<i>See also</i> assembler symbols	
exporting to other modules	52
predefined, in assembler	7
predefined, in assembler macro	65
user-defined, case sensitive	17
syntax conventions	xiii
s12 (filename extension)	3

## T

Technical Support, IAR	100
temporary values, defining	59
terminology	xi
__TID__ (predefined symbol)	7
__TIME__ (predefined symbol)	7
time-critical code	67
trademarks	ii
true value, in assembler expressions	5
typographic conventions	xiii

## U

UGT (assembler operator)	42
--------------------------	----

ULT (assembler operator)	42
UMINUS (CFI operator)	92
#undef (assembler directive)	73
UPPER (assembler operator)	42
user symbols, case sensitive	17

## V

value assignment directives	58
values, defining	77
VAR (assembler directive)	58
__VER__ (predefined symbol)	7
version, IAR Embedded Workbench	ii
version, of assembler	7

## W

warnings	99
classifying	20
disabling	25
exit code	28
treating as errors	28
--warnings_affect_exit_code (assembler option)	15, 28
--warnings_are_errors (assembler option)	28

## X

xcl (filename extension)	22
XOR (assembler operator)	43
XOR (CFI operator)	93

## Symbols

^ (assembler operator)	35
__BUILD_NUMBER__ (predefined symbol)	7
__CORE__ (predefined symbol)	7
__DATE__ (predefined symbol)	7
__FILE__ (predefined symbol)	7
__IAR_SYSTEMS_ASM__ (predefined symbol)	7

__LINE__ (predefined symbol) . . . . .	7	!= (assembler operator) . . . . .	39
__SUBVERSION__ (predefined symbol) . . . . .	7	?: (assembler operator) . . . . .	34
__TID__ (predefined symbol) . . . . .	7	() (assembler operator) . . . . .	32
__TIME__ (predefined symbol) . . . . .	7	* (assembler operator) . . . . .	32
__VER__ (predefined symbol) . . . . .	7	/ (assembler operator) . . . . .	33
_args (predefined macro symbol) . . . . .	65	/*...*/ (assembler directive) . . . . .	80
- (assembler operator) . . . . .	33	// (assembler directive) . . . . .	80
-D (assembler option) . . . . .	17	& (assembler operator) . . . . .	34
-f (assembler option) . . . . .	22	&& (assembler operator) . . . . .	34
-I (assembler option) . . . . .	22	#define (assembler directive) . . . . .	73
-l (assembler option) . . . . .	23	#elif (assembler directive) . . . . .	73
-M (assembler option) . . . . .	24	#else (assembler directive) . . . . .	73
-o (assembler option) . . . . .	25	#endif (assembler directive) . . . . .	73
-r (assembler option) . . . . .	27	#error (assembler directive) . . . . .	73
--case_insensitive (assembler option) . . . . .	17	#if (assembler directive) . . . . .	73
--core (assembler option) . . . . .	17	#ifdef (assembler directive) . . . . .	73
--debug (assembler option) . . . . .	18	#ifndef (assembler directive) . . . . .	73
--dependencies (assembler option) . . . . .	18	#include files, specifying . . . . .	22
--diagnostics_tables (assembler option) . . . . .	20	#include (assembler directive) . . . . .	73
--diag_error (assembler option) . . . . .	19	#line (assembler directive) . . . . .	73
--diag_remark (assembler option) . . . . .	20	#pragma (assembler directive) . . . . .	73, 97
--diag_suppress (assembler option) . . . . .	20	#undef (assembler directive) . . . . .	73
--diag_warning (assembler option) . . . . .	20	% (assembler operator) . . . . .	39
--dir_first (assembler option) . . . . .	21	+ (assembler operator) . . . . .	32–33
--enable_multibytes (assembler option) . . . . .	21	< (assembler operator) . . . . .	38
--error_limit (compiler option) . . . . .	22	<< (assembler operator) . . . . .	41
--header_context (assembler option) . . . . .	22	<= (assembler operator) . . . . .	38
--mnem_first (assembler option) . . . . .	24	<> (assembler operator) . . . . .	39
--no_path_in_file_macros (assembler option) . . . . .	25	= (assembler directive) . . . . .	58
--no_warnings (assembler option) . . . . .	25	= (assembler operator) . . . . .	36
--no_wrap_diagnostics (compiler option) . . . . .	25	== (assembler operator) . . . . .	36
--only_stdout (assembler option) . . . . .	26	> (assembler operator) . . . . .	37
--preinclude (compiler option) . . . . .	26	>= (assembler operator) . . . . .	37
--preprocess (assembler option) . . . . .	26	>> (assembler operator) . . . . .	41
--remarks (assembler option) . . . . .	27	(assembler operator) . . . . .	35
--silent (assembler option) . . . . .	27	(assembler operator) . . . . .	39
--warnings_affect_exit_code (assembler option) . . . . .	15, 28	~ (assembler operator) . . . . .	34
--warnings_are_errors (assembler option) . . . . .	28	\$ (program location counter) . . . . .	6
! (assembler operator) . . . . .	39		