

IAR Assembler™

Reference Guide

for STMicroelectronics'
STM8 Microcontroller Family



ASTM8-1

 IAR
SYSTEMS

COPYRIGHT NOTICE

Copyright © 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.

STMicroelectronics is a registered trademark of STMicroelectronics Corporation.
STM8 is a trademark of STMicroelectronics Corporation.

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

EDITION NOTICE

First edition: March 2010

Part number: ASTM8-1

This guide applies to version 1.x of IAR Embedded Workbench® for STM8.

Internal reference: AFE2, Too6.0, ascrt2010.2, IMAE.

Contents

| | |
|--|------|
| Tables | ix |
| Preface | xi |
| Who should read this guide | xi |
| How to use this guide | xi |
| What this guide contains | xii |
| Other documentation | xii |
| Document conventions | xii |
| Typographic conventions | xiii |
| Naming conventions | xiii |
| Introduction to the IAR Assembler for STM8 | 1 |
| Introduction to assembler programming | 1 |
| Getting started | 1 |
| Modular programming | 2 |
| External interface details | 3 |
| Assembler invocation syntax | 3 |
| Passing options | 3 |
| Environment variables | 4 |
| Error return codes | 4 |
| Source format | 5 |
| Assembler instructions | 5 |
| Expressions, operands, and operators | 6 |
| Integer constants | 6 |
| ASCII character constants | 7 |
| Floating-point constants | 7 |
| TRUE and FALSE | 8 |
| Symbols | 8 |
| Labels | 8 |
| Register symbols | 9 |
| Predefined symbols | 9 |
| Absolute and relocatable expressions | 11 |

| | |
|---|----|
| Expression restrictions | 11 |
| Operand modifiers | 12 |
| List file format | 13 |
| Header | 13 |
| Body | 13 |
| Summary | 13 |
| Symbol and cross-reference table | 13 |
| Programming hints | 14 |
| Accessing special function registers | 14 |
| Using C-style preprocessor directives | 14 |
| Assembler options | 15 |
| Setting command line assembler options | 15 |
| Specifying parameters | 16 |
| Summary of assembler options | 16 |
| Description of assembler options | 18 |
| Assembler operators | 31 |
| Precedence of operators | 31 |
| Summary of assembler operators | 32 |
| Parenthesis operator – 1 | 32 |
| Function operators – 2 | 32 |
| Unary operators – 3 | 32 |
| Multiplicative arithmetic operators – 4 | 33 |
| Additive arithmetic operators – 5 | 33 |
| Shift operators – 6 | 33 |
| Comparison operators – 7 | 33 |
| Equivalence operators – 8 | 33 |
| Logical operators – 9-14 | 33 |
| Conditional operator – 15 | 34 |

| | |
|---|----|
| Description of assembler operators | 34 |
| Assembler directives | 47 |
| Summary of assembler directives | 47 |
| Module control directives | 51 |
| Syntax | 51 |
| Parameters | 51 |
| Descriptions | 51 |
| Symbol control directives | 53 |
| Syntax | 53 |
| Parameters | 53 |
| Descriptions | 53 |
| Examples | 54 |
| Section control directives | 55 |
| Syntax | 55 |
| Parameters | 55 |
| Descriptions | 56 |
| Examples | 57 |
| Value assignment directives | 58 |
| Syntax | 58 |
| Parameters | 58 |
| Descriptions | 59 |
| Examples | 59 |
| Conditional assembly directives | 60 |
| Syntax | 61 |
| Parameters | 61 |
| Descriptions | 61 |
| Examples | 62 |
| Macro processing directives | 63 |
| Syntax | 63 |
| Parameters | 63 |
| Descriptions | 64 |
| Examples | 67 |

| | |
|---|-----|
| Listing control directives | 70 |
| Syntax | 70 |
| Descriptions | 71 |
| Examples | 72 |
| C-style preprocessor directives | 74 |
| Syntax | 74 |
| Parameters | 75 |
| Descriptions | 75 |
| Examples | 78 |
| Data definition or allocation directives | 79 |
| Syntax | 80 |
| Parameters | 80 |
| Descriptions | 80 |
| Examples | 81 |
| Assembler control directives | 82 |
| Syntax | 82 |
| Parameters | 82 |
| Descriptions | 82 |
| Examples | 83 |
| Call frame information directives | 84 |
| Syntax | 85 |
| Parameters | 86 |
| Descriptions | 87 |
| Simple rules | 91 |
| CFI expressions | 93 |
| Example | 95 |
| Pragma directives | 99 |
| Summary of pragma directives | 99 |
| Descriptions of pragma directives | 99 |
| Diagnostics | 101 |
| Message format | 101 |
| Severity levels | 101 |
| Setting the severity level | 102 |

| | |
|----------------------|------------|
| Internal error | 102 |
| Index | 103 |

Tables

| | |
|--|------|
| 1: Typographic conventions used in this guide | xiii |
| 2: Naming conventions used in this guide | xiii |
| 3: Assembler environment variables | 4 |
| 4: Assembler error return codes | 4 |
| 5: Integer constant formats | 6 |
| 6: ASCII character constant formats | 7 |
| 7: Floating-point constants | 7 |
| 8: Predefined register symbols | 9 |
| 9: Predefined symbols | 9 |
| 10: Operand modifiers | 12 |
| 11: Symbol and cross-reference table | 13 |
| 12: Assembler options summary | 16 |
| 13: Generating a list of dependencies (--dependencies) | 20 |
| 14: Conditional list options (-l) | 25 |
| 15: Directing preprocessor output to file (--preprocess) | 29 |
| 16: Assembler directives summary | 47 |
| 17: Module control directives | 51 |
| 18: Symbol control directives | 53 |
| 19: Section control directives | 55 |
| 20: Value assignment directives | 58 |
| 21: Conditional assembly directives | 60 |
| 22: Macro processing directives | 63 |
| 23: Listing control directives | 70 |
| 24: C-style preprocessor directives | 74 |
| 25: Data definition or allocation directives | 79 |
| 26: Assembler control directives | 82 |
| 27: Call frame information directives | 84 |
| 28: Unary operators in CFI expressions | 93 |
| 29: Binary operators in CFI expressions | 94 |
| 30: Ternary operators in CFI expressions | 95 |
| 31: Code sample with backtrace rows and columns | 96 |

| | |
|-------------------------------------|----|
| 32: Pragma directives summary | 99 |
|-------------------------------------|----|

Preface

Welcome to the IAR Assembler Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use the IAR Assembler for STM8 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 STM8 microcontroller and need to get detailed reference information on how to use the IAR Assembler. In addition, you should have working knowledge of the following:

- The architecture and instruction set of the STM8 microcontroller. Refer to the documentation from STMicroelectronics for information about the STM8 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, you should read the chapter *Introduction to the IAR Assembler for STM8* 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, and tutorials that can help you get started.

What this guide contains

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

- *Introduction to the IAR Assembler for STM8* 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.

Other documentation

The complete set of IAR Systems development tools for the STM8 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 STM8 and using the IAR ILINK Linker, refer to the *IAR C/C++ Development Guide for STM8*
- Using the IAR DLIB Library, refer to the online help system.

All of these guides are delivered in hypertext PDF or HTML format on the installation media.

Document conventions

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

When referring to a directory in your product installation, for example `stm8\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.n\stm8\doc`.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:





| Style | Used for |
|---|--|
| <code>computer</code> | <ul style="list-style-type: none"> • Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers. |
| <i>parameter</i> | A placeholder for an actual value used as a parameter, for example <i>filename.h</i> where <i>filename</i> represents the name of the file. |
| [option] | An optional part of a command. |
| a b c | Alternatives in a command. |
| {a b c} | A mandatory part of a command with alternatives. |
| bold | Names of menus, menu commands, buttons, and dialog boxes that appear on the screen. |
| <i>italic</i> | <ul style="list-style-type: none"> • A cross-reference within this guide or to another guide. • Emphasis. |
| ... | An ellipsis indicates that the previous item can be repeated an arbitrary number of times. |
|  | Identifies instructions specific to the IAR Embedded Workbench® IDE interface. |
|  | Identifies instructions specific to the command line interface. |
|  | Identifies helpful tips and programming hints. |
|  | Identifies warnings. |

Table 1: Typographic conventions used in this guide

NAMING CONVENTIONS

The following naming conventions are used for the products and tools from IAR Systems® referred to in this guide:

| Brand name | Generic term |
|--------------------------------------|-------------------------|
| IAR Embedded Workbench® for STM8 | IAR Embedded Workbench® |
| IAR Embedded Workbench® IDE for STM8 | the IDE |
| IAR C-SPY® Debugger for STM8 | C-SPY, the debugger |
| IAR C-SPY® Simulator | the simulator |
| IAR C/C++ Compiler™ for STM8 | the compiler |

Table 2: Naming conventions used in this guide

| Brand name | Generic term |
|-------------------------|---------------------|
| IAR Assembler™ for STM8 | the assembler |
| IAR ILINK™ Linker | ILINK, the linker |
| IAR DLIB Library™ | the DLIB library |

Table 2: Naming conventions used in this guide (Continued)

Introduction to the IAR Assembler for STM8

This chapter contains these sections:

- Introduction to assembler programming
- Modular programming
- External interface details
- 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 might be situations where you find it necessary to write parts of the code in assembler, for example, when using mechanisms in the STM8 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 STM8 microcontroller. Refer to STMicroelectronics'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++ Development Guide for STM8*

- In the IAR Embedded Workbench IDE, you can base a new project on a *template* for an assembler project.

Modular programming

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

- efficient program development
- reuse of modules
- maintenance.

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

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

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

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

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

- Create many small modules, one per source file
- In each module, divide your assembler source code into small subroutines (corresponding to *functions* on the C level)
- Divide your assembler source code into *sections*, to gain more precise control of how your code and data finally is placed in memory

- Collect your routines in libraries, which means that you can reduce the number of object files and make the modules conditionally linked.

External interface details

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

You can use the assembler either from the IAR Embedded Workbench IDE or from the command line. Refer to the *IAR Embedded Workbench® IDE User Guide* for information about using the assembler from the IAR Embedded Workbench IDE.

ASSEMBLER INVOCATION SYNTAX

The invocation syntax for the assembler is:

```
istm8asm [options][sourcefile][options]
```

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

```
istm8asm prog --debug
```

By default, the IAR Assembler for STM8 recognizes the filename extensions `s`, `asm`, and `msa` for source files. The default filename extension for assembler output is `o`.

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

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

PASSING OPTIONS

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

- Directly from the command line
 - Specify the options on the command line after the `istm8asm` command; see *Assembler invocation syntax*, page 3.
- Via environment variables
 - The assembler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 4.
- Via a text file by using the `-f` option; see *-f*, page 24.

For general guidelines for the option syntax, an options summary, and a detailed description of each option, see the *Assembler options* chapter.

ENVIRONMENT VARIABLES

Assembler options can also be specified in the `ASMSTM8` 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.

You can use these environment variables with the IAR Assembler for STM8:

| Environment variable | Description |
|---------------------------|--|
| <code>IASMSTM8</code> | Specifies command line options; for example: <code>set IASMSTM8=-la . --warnings_are_errors</code> |
| <code>IASMSTM8_INC</code> | Specifies directories to search for include files; for example: <code>set IASMSTM8_INC=c:\myinc\</code> |

Table 3: Assembler environment variables

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

```
set IASMSTM8=-l temp.lst
```

For information about the environment variables used by the compiler and linker, see the *IAR C/C++ Development Guide for STM8*.

ERROR RETURN CODES

When using the IAR Assembler for STM8 from within a batch file, you might have to determine whether the assembly was successful to decide what step to take next. For this reason, the assembler returns these error return codes:

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

Table 4: Assembler error return codes

Source format

The format of an assembler source line is as follows:

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

where the components are as follows:

| | |
|------------------|--|
| <i>label</i> | 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. |
| <i>operation</i> | An assembler instruction or directive. This must not start in the first column—there must be some whitespace to the left of it. |
| <i>operands</i> | An assembler instruction or directive can have zero, one, two, or three operands. The operands are separated by commas. An operand can be: <ul style="list-style-type: none"> • a constant representing a numeric value or an address • a symbolic name representing a numeric value or an address (where the latter also is referred to as a label) • a floating-point constant • a register • a predefined symbol • the program location counter (PLC) • an expression. |
| <i>comment</i> | Comment, preceded by a ; (semicolon) C or C++ comments are also allowed. |

The components are separated by spaces or tabs.

A source line can 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 might be set up differently in different editors, do not use tabs in your source files.

Assembler instructions

The IAR Assembler for STM8 supports the syntax for assembler instructions as described in the hardware documentation from STMicroelectronics.

Expressions, operands, and operators

Expressions consist of expression operands and operators.

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

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators; see also *Assembler operators*, page 31.

These operands are valid in an expression:

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

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

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

INTEGER CONSTANTS

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

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

Commas and decimal points are not permitted.

The following types of number representation are supported:

| Integer type | Example |
|--------------|------------------------|
| Binary | 1010b, b'1010 |
| Octal | 1234q, q'1234 |
| Decimal | 1234, -1, d'1234 |
| Hexadecimal | 0FFFFh, 0xFFFF, h'FFFF |

Table 5: 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 can be used in ASCII strings. If the quote character itself will 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' |
| '' '' (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 6: ASCII character constant formats

FLOATING-POINT CONSTANTS

The IAR Assembler for STM8 accepts floating-point values as constants and convert them into IEEE single-precision (signed 32-bit or signed 64-bit) floating-point format or fractional format.

Floating-point numbers can be written in the format:

$$[+|-] [digits] . [digits] [{E|e} [+|-] digits]$$

This table shows some valid examples:

| Format | Value |
|-------------|---------------------------|
| 10.23 | 1.023×10^1 |
| 1.23456E-24 | 1.23456×10^{-24} |
| 1.0E3 | 1.0×10^3 |

Table 7: Floating-point constants

Spaces and tabs are not allowed in floating-point constants.

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

When a fractional format is used—for example, DQ15—the range that can be represented is $-1.0 \leq x < 1.0$. Any value outside that range is silently saturated into the maximum or minimum value that can be represented.

If the word length of the fractional data is n , the fractional number will be represented as the 2-complement number: $x * 2^{(n-1)}$.

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 18 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.

Note that symbols and labels are byte addresses. For additional information, see *Generating a lookup table*, page 81.

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

```
JRA    $    ; Loop forever
```

REGISTER SYMBOLS

This table shows the existing predefined register symbols:

| Name | Size | Description |
|------|---------|-------------------------|
| A | 8 bits | Accumulator |
| X | 16 bits | Index register |
| Y | 16 bits | Index register |
| SP | 16 bits | Stack pointer |
| XL | 8 bits | Low half of X |
| XH | 8 bits | High half of X |
| YL | 8 bits | Low half of Y |
| YH | 8 bits | High half of Y |
| CC | 8 bits | Condition code register |

Table 8: Predefined register symbols

PREDEFINED SYMBOLS

The IAR Assembler for STM8 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.

These predefined symbols are available:

| Symbol | Value |
|------------------|---|
| __IASMSTM8__ | An integer that is set to 1 when the code is assembled with the IAR Assembler for STM8. |
| __BUILD_NUMBER__ | 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. |
| __DATE__ | The current date in dd/Mmm/yyyy format (string). |
| __FILE__ | The name of the current source file (string). |

Table 9: Predefined symbols

| Symbol | Value |
|----------------------------------|---|
| <code>__IAR_SYSTEMS_ASM__</code> | IAR assembler identifier (number). The current value is 7. Note that the number could be higher in a future version of the product. This symbol can be tested with <code>#ifdef</code> to detect whether the code was assembled by an assembler from IAR Systems. |
| <code>__LINE__</code> | The current source line number (number). |
| <code>__SUBVERSION__</code> | An integer that identifies the subversion number of the assembler version number, for example 3 in 1.2.3.4. |
| <code>__TIME__</code> | The current time in <code>hh:mm:ss</code> format (string). |
| <code>__VER__</code> | The version number in integer format; for example, version 4.17 is returned as 417 (number). |

Table 9: Predefined symbols (Continued)

Including symbol values in code

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

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

```

        name      timeOfAssembly
        extern   printStr
        section  .text:code

time    dc8      __TIME__          ; String representing the
                                           ; time of assembly.
        ldw     x,#time            ; Load address of time
                                           ; string in x.
        call    printStr          ; Call string output routine.
        end

```

Testing symbols for conditional assembly

To test a symbol at assembly time, use one of the conditional assembly directives. These directives let you control the assembly process at assembly time.

For example, if you want to assemble separate code sections depending on whether you are using an old assembler version or a new assembler version, do as follows:

```

#if ( __VER__ > 300)                ; New assembler version
;...
;...
#else                                ; Old assembler version
;...

```



```

;...
#endif

```

See *Conditional assembly directives*, page 60.

ABSOLUTE AND RELOCATABLE EXPRESSIONS

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

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

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

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

```

                name    simpleExpressions
                section .data:data
size           define 12
first         dc8     5                ; An absolute expression.

                dc8     first           ; Examples of some legal
                dc8     first + 1       ; relocatable expressions.
                dc8     first + 8 * size
end

```

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

EXPRESSION RESTRICTIONS

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

The following expression restrictions are referred to in the description of each directive they apply to.

No forward

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

No external

No external references in the expression are allowed.

Absolute

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

Fixed

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

OPERAND MODIFIERS

These prefixes can be used for modifying operands:

| Modifier | Description |
|----------|---|
| S: | Forces the assembler to use an 8-bit address (short addressing) |
| L: | Forces the assembler to use a 16-bit address (long addressing) |

Table 10: Operand modifiers

Note: The lowercase prefixes *s* and *l* are equivalent to the uppercase counterparts.

Example

The operand modifier *S*: is needed to determine whether

```
LD    A,shortmem
```

or

```
LD    A,longmem
```

shall be used. For example:

```
LD    A,S:max
```

List file format

The format of an assembler list file is as follows:

HEADER

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

BODY

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

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

SUMMARY

The *end* of the file contains a summary of errors and warnings that were generated.

SYMBOL AND CROSS-REFERENCE TABLE

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

This information is provided for each symbol in the table:

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

Table 11: Symbol and cross-reference table

Programming hints

This section gives hints on how to write efficient code for the IAR Assembler for STM8. For information about projects including both assembler and C or C++ source files, see the *IAR C/C++ Development Guide for STM8*.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several STM8 devices are included in the IAR Systems product package, in the `\stm8\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 STM8, and they are suitable to use as templates when creating new header files for other STM8 devices.

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

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

USING C-STYLE PREPROCESSOR DIRECTIVES

The C-style preprocessor directives are processed before other assembler directives. Therefore, do not use preprocessor directives in macros and do not mix them with assembler-style comments. For more information about comments, see *Assembler control directives*, page 82.

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 command line assembler options

To set assembler options from the command line, include them on the command line after the `astm8` command, either before or after the source filename. For example, when assembling the source file `prog.s`, use this command to generate an object file with debug information:

```
iastm8 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`:

```
iastm8 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:

```
iastm8 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. However, there is one exception: when you use the `-I` option, the directories are searched in the same order as they are specified on the command line.

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

- A short option name consists of one character, with or without parameters. You specify it with a single dash, for example `-r`.
- A long name consists of one or several words joined by underscores, and it can 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, you can specify it either immediately following the option or as the next command line argument.

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

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

or as

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

Note: You can use `/` instead of `\` as directory delimiter. A trailing 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 then receives a default name and extension.

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

```
--diag_suppress=Pe0001
```

or

```
--diag_suppress Pe0001
```

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

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

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

```
iastm8 prog -l .
```

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

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

```
iastm8 prog -l ---
```

Summary of assembler options

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

| Command line option | Description |
|---------------------------------|-------------------------------|
| <code>--case_insensitive</code> | Case-insensitive user symbols |

Table 12: Assembler options summary

| Command line option | Description |
|---------------------------------------|--|
| <code>--code_model</code> | Specifies the code model |
| <code>--core</code> | Specifies a CPU core |
| <code>-D</code> | Defines preprocessor symbols |
| <code>--data_model</code> | Specifies the data model |
| <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 |
| <code>--diagnostics_tables</code> | Lists all diagnostic messages |
| <code>--dir_first</code> | Allows directives in the first column |
| <code>--enable_multibytes</code> | Enables support for multibyte characters |
| <code>--error_limit</code> | Specifies the allowed number of errors before the assembler stops |
| <code>-f</code> | Extends the command line |
| <code>--header_context</code> | Lists all referred source files |
| <code>-I</code> | Add search path for header file |
| <code>-l</code> | Generates list file |
| <code>-M</code> | Macro quote characters |
| <code>--mnm_first</code> | Allows mnemonics in the first column |
| <code>--no_path_in_file_macros</code> | Removes the path from the return value of the symbols <code>__FILE__</code> and <code>__BASE_FILE__</code> |
| <code>--no_warnings</code> | Disables all warnings |
| <code>--no_wrap_diagnostics</code> | Disables wrapping of diagnostic messages |
| <code>-o</code> | Sets object filename. This is an alias for <code>--output</code> . |
| <code>--only_stdout</code> | Uses standard output only |
| <code>--output</code> | Sets object filename |
| <code>--predef_macros</code> | Lists the predefined symbols |
| <code>--preinclude</code> | Includes an include file before reading the source file |
| <code>--preprocess</code> | Preprocessor output to file |
| <code>-r</code> | Generates debug information. This is an alias for <code>--debug</code> . |

Table 12: Assembler options summary (Continued)

| Command line option | Description |
|--|-------------------------------|
| <code>--remarks</code> | Enables remarks |
| <code>--silent</code> | Sets silent operation |
| <code>--warnings_affect_exit_code</code> | Warnings affect exit code |
| <code>--warnings_are_errors</code> | Treats all warnings as errors |

Table 12: 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` 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 82, 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 75.



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

`--code_model` `--code_model={small|medium|large}`

Use this option to inform the assembler of which code model that is used. All modules of your application must use the same code model.

| | |
|-------------------------------|-------------------------------|
| <code>small</code> | The small code model is used |
| <code>medium (default)</code> | The medium code model is used |

`large` The large code model is used

This option also controls the value of the symbol `__CODE_MODEL__`. See *Predefined symbols*, page 9. For information about the different code models, see the *IAR C/C++ Development Guide for STM8*.



Project>Options>General Options >Target>Code model

`--core` `--core {stm8}`

Use this option to select the processor core for which the code will be generated.



This option is not available in the IDE because this setting is automatically enabled.

`-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 might 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:  iasmstm8 prog
Test version:       iasmstm8 prog -DTESTVER
```

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

```
iasmstm8 prog -DFRAMERATE=3
```



Project>Options>Assembler>Preprocessor>Defined symbols

```
--data_model --data_model={small|medium|large}
```

Use this option to inform the assembler of which data model that is used. All modules of your application must use the same data model.

| | |
|------------------|-------------------------------|
| small | The small data model is used |
| medium (default) | The medium data model is used |
| large | The large code data is used |

This option also controls the value of the symbol `__DATA_MODEL__`. See *Predefined symbols*, page 9. For information about the different data models, see the *IAR C/C++ Development Guide for STM8*.



Project>Options>General Options >Target>Data model

```
--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.

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. These modifiers are available:

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

Table 13: 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 is the same as the name of the assembled source file, unless a different name was specified with the `-o` option, in which case that name is 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.o: c:\iar\product\include\stdio.h
foo.o: d:\myproject\include\foo.h
```

Example 1

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

```
astm8 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:

```
astm8 prog --dependencies \mypath\listing
```

Note: You can use both `\` and `/` as directory delimiters.

Example 3

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

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

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

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

This 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 might cause strange behavior in the generated code.

This 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. This 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 does not cause the assembler to stop before the assembly is completed.

This 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 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 .
```

You can use both `\` and `/` 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 if there are many options which are more conveniently placed in a file than on the command line itself.

Example

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

```
iasmstm8 prog -f extend.xcl
```



To set this option, use:

Project>Options>Assembler>Extra Options

--header_context --header_context

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



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

-I -I*path*

Use this option to specify paths to be used by the preprocessor, by adding the `#include` file search prefix *path*.

By default, the assembler searches for `#include` files only in the current working directory and in the paths specified in the `ASTM8_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, makes 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 `ASTM8_INC` environment variable, provided that this variable is set.



Project>Options>Assembler >Preprocessor>Additional include directories

```
-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 |
|---------------------|--|
| -la | Assembled lines only |
| -ld | The <code>LSTOUT</code> directive controls if lines are written to the list file or not. Using <code>-ld</code> turns the start value for this to off. |
| -le | No macro expansions |
| -lm | Macro definitions |
| -lo | Multiline code |
| -lx | Includes cross-references |
| -lN | Do not include diagnostics |

Table 14: 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 `lst`. The filename is the same as the name of the assembled source file, unless a different name was specified with the `-o` option, in which case that name is used.

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

Example 1

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

```
astm8 sourcefile -l list
```

Example 2

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

```
astm8 mysource -l .
```

Note: You can use both `\` and `/` as directory delimiters.



To set related options, select:

Project>Options>Assembler >List

`-M -Mab`

This option sets the characters to be used as 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.

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.

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

```
iasmstm8 filename -M'<>'
```



Project>Options>Assembler >Language>Macro quote characters

```
--mnem_first --mnem_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 recognized as mnemonics.



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

```
--no_fragments --no_fragments
```

Use this option to disable section fragment handling. Normally, the toolset uses IAR proprietary information for transferring section fragment information to the linker. The linker uses this information to remove unused code and data, and thus further minimize the size of the executable image.



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

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

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



This option is not available in the 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.

```
--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.

```
--output, -o --output {filename|path}
-o {filename|path}
```

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

For more syntax information, see *Setting command line assembler options*, page 15.



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

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

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

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 `predef` filename extension.

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

For more syntax information, see *Setting command line assembler options*, page 15.



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

```
--preinclude --preinclude includefile
```

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



To set this option, use:

Project>Options>Assembler>Extra Options

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

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

This 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 15: 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 is the same as the name of the assembled source file, unless a different name was specified with the `-o` option, in which case that name is 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:

```
astm8 sourcefile --preprocess=c output
```

Example 2

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

```
astm8 mysource --preprocess=l .
```

Note: You can use both `\` and `/` as directory delimiters.



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

```
--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 might cause strange behavior in the generated code. By default, remarks are not generated.



See *Severity levels*, page 101, 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. 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 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, 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 22.



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).

These rules determine how expressions are evaluated:

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

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

Note: The precedence order in the IAR Assembler for STM8 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 precedence. Synonyms, where available, are shown in brackets after the operator name.

PARENTHESIS OPERATOR – 1

| | |
|-----|--------------|
| () | Parenthesis. |
|-----|--------------|

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 | Section begin. |
| SFE | Section end. |
| SIZEOF | Section 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 ? is 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 is 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 is 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 is 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 is 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 is 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 Section begin (2).

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

Syntax

```
SFB(section [{+|-}offset])
```

Parameters

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

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

Example

```

name      segmentBegin
section .mycode:code ; Forward declaration of
                        ; .mycode.
section .segtab:const
start    dc16      sfb(.mycode)
end
```

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

SFE Section end (2).

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

Syntax

```
SFE (section [{+ | -} offset])
```

Parameters

section The name of a relocatable section, 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      segmentEnd
section  .mycode:code          ; Forward declaration
                                   ; of .mycode.

section  .segtab:const
dc16     sfe(.mycode)
end
end

```

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

The size of the section `MY_SECTION` can be calculated as:

```
SFE(MY_SECTION) - SFB(MY_SECTION)
```

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 Section size (2).

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

Syntax

SIZEOF (*section*)

Parameters

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

Example

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

Table.s:

```

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

```

Application.s:

```

                                module application
                                section MYCODE:CODE(2)
                                nop ; Placeholder for application.
                                end

```

UGT Unsigned greater than (7).

UGT evaluates to 1 (true) if the left operand has a larger value than the right operand, otherwise it is 0 (false). The operation treats 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 is 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 assembler directives are classified into these groups according to their function:

- *Module control directives*, page 51
- *Symbol control directives*, page 53
- *Section control directives*, page 55
- *Value assignment directives*, page 58
- *Conditional assembly directives*, page 60
- *Macro processing directives*, page 63
- *Listing control directives*, page 70
- *C-style preprocessor directives*, page 74
- *Data definition or allocation directives*, page 79
- *Assembler control directives*, page 82
- *Call frame information directives*, page 84.

This table gives a summary of all the assembler directives.

| Directive | Description | Section |
|-----------------------|--|----------------------|
| <code>_args</code> | Is set to number of arguments passed to macro. | Macro processing |
| <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 |

Table 16: Assembler directives summary

| Directive | Description | Section |
|--------------------------|---|-------------------------------|
| <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. | Section control |
| <code>ALIGNRAM</code> | Aligns the program location counter. | Section control |
| <code>ASSIGN</code> | Assigns a temporary value. | Value assignment |
| <code>CASEOFF</code> | Disables case sensitivity. | Assembler control |
| <code>CASEON</code> | Enables case sensitivity. | Assembler control |
| <code>CFI</code> | Specifies call frame information. | Call frame information |
| <code>DB</code> | Generates 8-bit constants, including strings. | Data definition or allocation |
| <code>DC8</code> | Generates 8-bit constants, including strings. | Data definition or allocation |
| <code>DC16</code> | Generates 16-bit constants. | Data definition or allocation |
| <code>DC24</code> | Generates 24-bit constants. | Data definition or allocation |
| <code>DC32</code> | Generates 32-bit constants. | Data definition or allocation |
| <code>DC64</code> | Generates 64-bit constants. | Data definition or allocation |
| <code>DEFINE</code> | Defines a file-wide value. | Value assignment |
| <code>DF32</code> | Generates 32-bit floating-point constants. | Data definition or allocation |
| <code>DF64</code> | Generates 64-bit floating-point constants. | Data definition or allocation |
| <code>DQ15</code> | Generates 16-bit fractional constants. | Data definition or allocation |

Table 16: Assembler directives summary (Continued)

| Directive | Description | Section |
|------------------|---|-------------------------------|
| DQ31 | Generates 32-bit fractional 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 |
| DW | Generates 16-bit constants. | 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 | Ends 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 |
| 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. | Section control |
| EXITM | Exits prematurely from a macro. | Macro processing |
| EXTERN | Imports an external symbol. | Symbol control |
| IF | Assembles instructions if a condition is true. | Conditional assembly |
| IMPORT | Imports an external symbol. | Symbol control |
| LIBRARY | Begins a module; an alias for PROGRAM and NAME. | Module control |
| LOCAL | Creates symbols local to a macro. | Macro processing |
| LSTCND | | 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 |

Table 16: Assembler directives summary (Continued)

| Directive | Description | Section |
|------------------|--|-------------------|
| LSTPAG | Retained for backward compatibility reasons; recognized but ignored. | 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 module; an alias for PROGRAM and NAME. | Module control |
| NAME | Begins a program module. | Module control |
| ODD | Aligns the program location counter to an odd address. | Section control |
| OVERLAY | Recognized but ignored. | Symbol control |
| PROGRAM | Begins a 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 |
| 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 section. | Section control |
| RTMODEL | Declares runtime model attributes. | Module control |
| SECTION | Begins a section. | Section control |
| SECTION_TYPE | Sets ELF type and flags for a section. | Section control |
| SET | Assigns a temporary value. | Value assignment |
| VAR | Assigns a temporary value. | Value assignment |

Table 16: 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 to them. See *Expression restrictions*, page 11, for a description of the restrictions that apply when using a directive in an expression.

| Directive | Description | Expression restrictions |
|-----------|---|--|
| END | Ends the assembly of the last module in a file. | Only locally defined labels or integer constants |
| NAME | Begins a module; alias to PROGRAM. | No external references Absolute |
| PROGRAM | Begins a module. | No external references Absolute |
| RTMODEL | Declares runtime model attributes. | Not applicable |

Table 17: Module control directives

SYNTAX

END

NAME *symbol*

PROGRAM *symbol*

RTMODEL *key, value*

PARAMETERS

key A text string specifying the key.

symbol Name assigned to module.

value A text string specifying the value.

DESCRIPTIONS

Beginning a module

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

A module is included in the linked application, even if other modules do not reference them. For more information about how modules are included in the linked application, read about the linking process in the *IAR C/C++ Development Guide for STM8*.

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

Terminating the source file

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

Declaring runtime model attributes

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

A module can have several runtime model definitions.

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

If you are writing assembler routines for use with C or C++ code, and you want to control the module consistency, refer to the *IAR C/C++ Development Guide for STM8*.

Examples

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

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

Assembler source file `f1.s`:

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

Assembler source file `f2.s`:

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

Assembler source file `f3.s`:

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

Symbol control directives

These directives control how symbols are shared between modules.

| Directive | Description |
|----------------|---|
| EXTERN, IMPORT | Imports an external symbol. |
| OVERLAY | Recognized but ignored. |
| PUBLIC | Exports symbols to other modules. |
| PUBWEAK | Exports symbols to other modules, multiple definitions allowed. |
| REQUIRE | Forces a symbol to be referenced. |

Table 18: Symbol control directives

SYNTAX

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

PARAMETERS

label Label to be used as an alias for a C/C++ symbol.

symbol Symbol to be imported or exported.

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

Exporting symbols with multiple definitions to other modules

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

A section cannot contain both a public symbol and a pubweak symbols.

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

Importing symbols

Use EXTERN or IMPORT to import an untyped external symbol.

The REQUIRE directive marks a symbol as referenced. This is useful if the section containing the symbol must be loaded 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.

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

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

```

name      errorMessage
extern   print
public   err
section  .text:code

err      call    print
         dc8     "*** Error ***"
         ret

         end
```

Section control directives

The section directives control how code and data are located. See *Expression restrictions*, page 11, 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 |
| 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 |
| RSEG | Begins an ELF section; alias to SECTION. | No external references Absolute |
| SECTION | Begins an ELF section. | No external references Absolute |
| SECTION_TYPE | Sets ELF type and flags for a section. | |

Table 19: Section control directives

SYNTAX

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

```
ALIGNRAM align
```

```
EVEN [value]
```

```
ODD [value]
```

```
RSEG section :type [flag] [(align)]
```

```
SECTION segment :type [flag] [(align)]
```

```
SECTION_TYPE type-expr {, flags-expr}
```

PARAMETERS

align The power of two to which the address should be aligned, in most cases in the range 0 to 30.
The default align value is 0.

| | |
|-------------------|--|
| <i>flag</i> | NOROOT, ROOT NOROOT means that the sectionfragment is discarded by the linker if no symbols in this sectionfragment are referred to. Normally, all sectionfragments except startup code and interrupt vectors should set this flag. The default mode is ROOT which indicates that the sectionfragment must not be discarded. REORDER, NOREORDER REORDER starts a new section with the given name. The default mode is NOREORDER which starts a new fragment in the section with the given name, or a new section if no such section exists. |
| <i>section</i> | The name of the section. |
| <i>type</i> | The memory type, which can be either CODE, CONST, or DATA. |
| <i>value</i> | Byte value used for padding, default is zero. |
| <i>type-expr</i> | A constant expression identifying the ELF type of the section. |
| <i>flags-expr</i> | A constant expression identifying the ELF flags of the section. |

DESCRIPTIONS

Beginning a relocatable section

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

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

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

Aligning a section

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

`ALIGN` aligns by inserting zero/filled bytes, up to a maximum of 255. The `EVEN` directive aligns the program counter to an even address (which is equivalent to `ALIGN 1`) and the `ODD` directive aligns the program location counter to an odd address. The 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 a relocatable section

In the following example, the data following the first `RSEG` directive is placed in a section called `table`.

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

```

                                module calculate
                                extern operator
                                extern addOperator, subOperator

                                section .table:const(8)
operatorTable:
                                dc8      addOperator, subOperator

                                section .text:code
calculate  ld      a,operator
                                ldw     x, #operatorTable
                                cp      a, (x)
                                jreq    add
                                cp      a, (1,x)
                                jreq    sub
                                ;...
                                ret

add        ;...
                                ret

sub        ;...
                                ret

                                end

```

Aligning a section

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

```

                                name    alignment
                                section .data:data    ; Start a relocatable data
                                                                ; segment.
                                even                ; Ensure it is on an even
                                                                ; boundary.
target    dc16    1                ; target and best will be on
best      dc16    1                ; an even boundary.
                                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. |

Table 20: Value assignment directives

SYNTAX

```

label = expr
label ASSIGN expr
label DEFINE const_expr
label EQU expr
label SET expr
label VAR expr

```

PARAMETERS

| | |
|-------------------|---|
| <i>const_expr</i> | Constant value assigned to symbol. |
| <i>expr</i> | Value assigned to symbol or value to be tested. |
| <i>label</i> | Symbol to be defined. |

DESCRIPTIONS

Defining a temporary value

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

Defining a permanent local value

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

Use `EXTERN` to import symbols from other modules.

Defining a permanent global value

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

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

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

EXAMPLES

Redefining a symbol

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

```

                name    table
cons           set     1

; Generate table of powers of 3.
cr_tabl       macro    times
                dc32    cons
cons          set     cons * 3
                if      times > 1
                cr_tabl times - 1
                endif
                endm

                section .text:CODE(2)

```

```
table      cr_tabl 4
end
```

It generates this code:

```
10          name      table
11      000001      cons      set      1
12
13          ; Generate table of powers of 3.
21
22      000000          section .text:code(2)
23      000000      table      cr_tabl 4
23.1      000000 00000001      dc32      cons
23.2      000003      cons      set      cons * 3
23.3      000004          if      4 > 1
23.4      000004          cr_tabl 4 - 1
23.5      000004 00000003      dc32      cons
23.6      000009      cons      set      cons * 3
23.7      000008          if      4 - 1 > 1
23.8      000008          cr_tabl 4 - 1 - 1
23.9      000008 00000009      dc32      cons
23.10     00001B      cons      set      cons * 3
23.11     00000C          if      4 - 1 - 1 > 1
23.12     00000C          cr_tabl 4 - 1 - 1 - 1
23.13     00000C 0000001B      dc32      cons
23.14     000051      cons      set      cons * 3
23.15     000010          if      4 - 1 - 1 - 1 > 1
23.16     000010          endif
23.17     000010          endif
23.18     000010          endif
23.19     000010          endif
24      000010          end
```

Conditional assembly directives

These directives provide logical control over the selective assembly of source code. See *Expression restrictions*, page 11, 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 |

Table 21: Conditional assembly directives

| Directive | Description | Expression restrictions |
|-----------|--|--|
| ENDIF | Ends an IF block. | |
| IF | Assembles instructions if a condition is true. | No forward references No external references Absolute Fixed |

Table 21: Conditional assembly directives (Continued)

SYNTAX

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

PARAMETERS

| | | |
|------------------|----------------------------------|---|
| <i>condition</i> | One of these: | |
| | An absolute expression | The expression must not contain forward or external references, and any non-zero value is considered as true. |
| | <i>string1</i> == <i>string2</i> | The condition is true if <i>string1</i> and <i>string2</i> have the same length and contents. |
| | <i>string1</i> != <i>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 ENDF directives to control the assembly process at assembly time. If the condition following the IF directive is not true, the subsequent instructions do not generate any code (that is, it is not assembled or syntax checked) until an ELSE or ENDF directive is found.

Use ELSEIF to introduce a new condition after an IF directive. Conditional assembly directives can be used anywhere in an assembly, but have their greatest use in conjunction with macro processing.

All assembler directives (except for END) as well as the inclusion of files can be disabled by the conditional directives. Each IF directive must be terminated by an ENDF

directive. The `ELSE` directive is optional, and if used, it must be inside an `IF...ENDIF` block. `IF...ENDIF` and `IF...ELSE...ENDIF` blocks can be nested to any level.

EXAMPLES

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

```
; If the second argument to the addMem macro is 1, 2, or 3,
; it generates the equivalent number of inc instructions. For any
; other non-zero value of the second argument, it generates an
; ld, an add, and an ld instruction.
```

```
addMem      macro    loc,val          ; loc is a direct page memory
                                           ; location, and val is an
                                           ; 8-bit value to add to that
                                           ; location.

            if      val = 0          ; Do nothing.

            elseif  val = 1
            inc     loc
            elseif  val = 2
            inc     loc
            inc     loc
            elseif  val = 3
            inc     loc
            inc     loc
            inc     loc
            else
            ld      a, loc
            add     a, #val
            ld      loc, a
            endif
            endm

            module  addWithMacro
            section .text:code

addSome     addMem  0xa0,0           ; Add 0 to memory location
0xa0.

            addMem 0xa0,1           ; Add 1 to the same address.
            addMem 0xa0,2           ; Add 2 to the same address.
            addMem 0xa0,3           ; Add 3 to the same address.
            addMem 0xa0,47          ; Add 47 to the same address.
            ret
            end
```

Macro processing directives

These directives allow user macros to be defined. See *Expression restrictions*, page 11, for a description of the restrictions that apply when using a directive in an expression.

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

Table 22: Macro processing directives

SYNTAX

```

_args
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.

expr 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.

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:

```
errMac      name    errMacro
            macro   text
            extern  abort
            call    abort
            dc8     text,0
            endm

            end
```

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

```
errMac 'Disk not ready'
```

The assembler expands this to:

```
jsr    abort
dc8    'Disk not ready',0
even
```

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:

```

errMac      name      errMacro
            macro     text
            extern    abort
            call      abort
            dc8       \1,0
            endm

            end

```

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

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

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

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

Note: It is illegal to *redefine* a macro.

Passing special characters

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

For example:

```

ldaMac      name      ldaMacro
            macro     op
            ldw       op
            endm

            end

```

The macro can be called using the macro quote characters:

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

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 26.

Predefined macro symbols

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

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

          module  fill
          section .text:code(2)
          fill    3
          fill    4, 3
          end
```

It generates this code:

| | | | |
|------|-----------|---------|---------------|
| 20 | | module | fill |
| 21 | 000000 | section | .text:code(2) |
| 22 | 000000 | fill | 3 |
| 22.1 | 000000 | if | _args == 2 |
| 22.2 | 000000 | else | |
| 22.3 | 000000 03 | dc8 | 3 |
| 22.4 | 000001 | endif | |
| 23 | 000001 | fill | 4, 3 |
| 23.1 | 000001 | if | _args == 2 |
| 23.2 | 000001 | rept | 3 |
| 23.3 | 000001 04 | dc8 | 4 |
| 23.4 | 000002 04 | dc8 | 4 |
| 23.5 | 000003 04 | dc8 | 4 |
| 23.6 | 000004 | endr | |
| 23.7 | 000004 | else | |
| 23.8 | 000004 | endif | |
| 24 | 000004 | end | |

How macros are processed

The macro process consists of three distinct phases:

- 1 The assembler scans and saves macro definitions. The text between `MACRO` and `ENDM` is saved but not syntax checked.
- 2 A macro call forces the assembler to invoke the macro processor (expander). The macro expander switches (if not already in a macro) the assembler input stream from a source file to the output from the macro expander. The macro expander takes its input from the requested macro definition.

The macro expander has no knowledge of assembler symbols since it only deals with text substitutions at source level. Before a line from the called macro definition is handed over to the assembler, the expander scans the line for all occurrences of symbolic macro arguments, and replaces them with their expansion arguments.

- 3 The expanded line is then processed as any other assembler source line. The input stream to the assembler continues to be the output from the macro processor, until all lines of the current macro definition have been read.

Repeating statements

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

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

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

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

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.

This example outputs bytes from a buffer to a port:

```
name      ioBufferSubroutine
```

```

public copyBuffer
ptbd      equ      0x5231      ; Definition of the USART
                                   ; data register.

        section .data:data
buffer    ds8      256

        section .text:code
copyBuffer push    #0          ; Initialize the loop
                                   ; counter.

        clrW     x
loop      ld      a, (buffer,x)
        ld      ptbd, a
        incW     x
        inc     (1,sp)
        jrne    loop          ; Have we copied 256 bytes?
        pop     a
        ret

        end

```

The main program calls this routine as follows:

```
doCopy    jsr     copyBuffer
```

For efficiency, we can recode this using a macro:

```

name      ioBufferInline
ptbd      equ      0x5231      ; Definition of the USART
                                   ; data register.

        section .data:data
buffer    ds8      256

copyBuffer macro
        local   loop
        push    #0          ; Initialize the loop
counter.

        clrW     x
loop      ld      a, (buffer,x)
        ld      ptbd, a
        incW     x
        inc     (1,sp)
        jrne    loop          ; Have we copied 256 bytes?
        pop     a
        endm

        section .text:code
copyBuffer
        end

```

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

Using REPTC and REPTI

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

```

                name    reptc
                extern  plotc
                section .text:code

banner         reptc  chr, "Welcome"
                ld      a, #'chr'
                call    plotc
                endr
                end

```

This produces this code:

```

     9
    10 000000                name    reptc
    11 000000                extern  plotc
    12                                section .text:code
    13 000000                banner  reptc  chr, "Welcome"
    13.1 000000 A657          ld      a, #'W'
    13.2 000002 CD0000        call    plotc
    13.3 000005 A665          ld      a, #'e'
    13.4 000007 CD0000        call    plotc
    13.5 00000A A66C          ld      a, #'l'
    13.6 00000C CD0000        call    plotc
    13.7 00000F A663          ld      a, #'c'
    13.8 000011 CD0000        call    plotc
    13.9 000014 A66F          ld      a, #'o'
    13.10 000016 CD0000       call    plotc
    13.11 000019 A66D          ld      a, #'m'
    13.12 00001B CD0000       call    plotc
    13.13 00001E A665          ld      a, #'e'
    13.14 000020 CD0000       call    plotc
    13.15 000023                endr
    17 000023                end

```

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

```

                name    repti
                extern  base, count, init
                section .text:code

```

```

banner      repti  adds, base, count, init
            clr    adds
            endr

            end

```

This produces this code:

```

9
10      000000      name      repti
11      000000      extern   base, count, init
12
13      000000      banner    repti  adds, base, count, init
13.1    000000  725F0000      clr    base
13.2    000004  725F0000      clr    count
13.3    000008  725F0000      clr    init
13.4    00000C      endr
16
17      00000C      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 23: Listing control directives

Note: The directives COL, LSTPAGE, PAGE, and PAGESIZ 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 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-
; This section has already been debugged.
lstout+
; This section is currently being debugged.
end
```

Listing conditional code and strings

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

```

name    lstcndTest
extern  print
section .text:code

debug   set    0
begin   if    debug
        call   print
        endif

        lstcnd+
begin2  if    debug
        call   print
        endif

end
```

This generates the following listing:

| | | | | |
|----|--------|--------|---------|------------|
| 9 | | | name | lstcndTest |
| 10 | 000000 | | extern | print |
| 11 | 000000 | | section | .text:code |
| 12 | | | | |
| 13 | 000000 | debug | set | 0 |
| 14 | 000000 | begin | if | debug |
| 15 | | | call | print |
| 16 | 000000 | | endif | |
| 17 | | | | |
| 18 | | | lstcnd+ | |
| 19 | 000000 | begin2 | if | debug |
| 21 | 000000 | | endif | |
| 22 | | | | |
| 23 | 000000 | | end | |

Controlling the listing of macros

This example shows the effect of `LSTMAC` and `LSTEXP`:

```

                name    lstmacTest
                extern  memLoc
                section .text:code

dec2           macro  arg
                dec    arg
                dec    arg
                endm

                lstmac+
inc2           macro  arg
                inc    arg
                inc    arg
                endm

begin         dec2   memLoc
                lstexp-
                inc2  memLoc
                ret

; Restore default values for
; listing control directives.

                lstmac-
                lstexp+

                end

```

This produces the following output:

```

 9
10      000000
11      000000
12
17
18
19                inc2           lstmac+
20                inc           arg
21                inc           arg
22                endm
23
24      000000                begin  dec2   memLoc
24.1    000000 725A0000      dec    memLoc
24.2    000004 725A0000      dec    memLoc
25                lstexp-

```

```

26      000008                inc2    memLoc
27      000010 81            ret
28
29                                ; Restore default values for
30                                ; listing control directives.
31
32                                lstmac-
33                                lstexp+
34
35      000011                end

```

C-style preprocessor directives

The assembler has a C-style preprocessor that follows the C99 standard.

These 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 an <code>#if...#endif</code> block. |
| <code>#else</code> | Assembles instructions if a condition is false. |
| <code>#endif</code> | Ends an <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 source references in the debug information. |
| <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 24: C-style preprocessor directives

SYNTAX

```

#define symbol text
#elif condition
#else
#endif
#error "message"

```



```

#if condition
#ifdef symbol
#ifndef symbol
#include {"filename" | <filename>}
#line line-no {"filename"}
#undef symbol

```

PARAMETERS

| | | |
|------------------|--|--|
| <i>condition</i> | 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 or referred. | |
| <i>line-no</i> | Source line number. | |
| <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

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

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

```

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

```

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

Defining and undefining preprocessor symbols

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

```
#define symbol value
```

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

Conditional preprocessor directives

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

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

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

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

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 <iostm8.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 `ASTM8_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 STM8, 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.

Comments in C-style preprocessor directives

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

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

Do not use assembler comments within a define statement as it leads to unexpected behavior.

This expression evaluates to 3 because the comment character is preserved by `#define`:

```
#define x 3      ; This is a misplaced comment.

        module misplacedComment1
expression equ   x * 8 + 5
           ; ...
        end
```

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

```
#define five 5      ; This comment is not OK.
#define six 6       // This comment is OK.
#define seven 7     /* This comment is OK. */

        module misplacedComment2
        section MYCONST:CONST(2)

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

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

        end
```

Changing the source line numbers

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

EXAMPLES

Using conditional preprocessor directives

This example defines the labels `tweak` and `adjust`. If `adjust` is defined, then register 16 is decremented by an amount that depends on `adjust`, in this case 30.

```

                                module  calibrate
                                extern  calibrationConstant
                                section .text:code

#define    tweak  1
#define    adjust 3

calibrate  ld      a, calibrationConstant
#ifdef    tweak
#if       adjust==1
          sub     a, #4
#elif    adjust==2
          sub     a, #20
#elif    adjust==3
          sub     a, #30
#endif
#endif
                                /* ifdef tweak */
                                ld      calibrationConstant, a
                                ret

                                end

```

Including a source file

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

```

; Exchange content of two bytes in memory.
; Use the stack for temporary storage.

```

```

xch      macro  a,b
          push  \1
          push  \2
          pop   \1
          pop   \2
        endm

```

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

```

        program includeFile
        section .text:code

; Standard macro definitions.
#include "Macros.inc"

        extern v, w, z

xchRegs    xch    v,w
           xch    z,v
           ret

           end

```

Data definition or allocation directives

These directives define values or reserve memory. The column *Alias* in the following table shows the STMicroelectronics directive that corresponds to the IAR Systems directive. See *Expression restrictions*, page 11, for a description of the restrictions that apply when using a directive in an expression.

| Directive | Alias | Description |
|-----------|-------|---|
| DC8 | DB | Generates 8-bit constants, including strings. |
| DC16 | DW | 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. |
| DQ15 | | Generates 16-bit fractional constants. |
| DQ31 | | Generates 32-bit fractional 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 25: Data definition or allocation directives

SYNTAX

```

DB expr [, expr] ...
DC8 expr [, expr] ...
DC16 expr [, expr] ...
DC24 expr [, expr] ...
DC32 expr [, expr] ...
DC64 expr [, expr] ...
DF32 value [, value] ...
DF64 value [, value] ...
DQ15 value [, value] ...
DQ31 value [, value] ...
DS8 count
DS16 count
DS24 count
DS32 count
DS64 count
DW expr [, expr] ...

```

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 are zero filled to a multiple of the data size implied by the directive. Double-quoted strings are zero-terminated.*

value A valid absolute expression or floating-point constant.

* For DC64, the *expr* cannot be relocatable or external.

DESCRIPTIONS

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

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

EXAMPLES

Generating a lookup table

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

```

                                module  sumTableAndIndex
                                section  .data:const

table    dc8    12
          dc8    15
          dc8    17
          dc8    16
          dc8    14
          dc8    11
          dc8     9

                                section  .text:code
count    set     0

addTable  clr     a

          rept    7
          if     count == 7
          exitm
          endif
          adc    a, table + count
count    set     count + 1
          endr

          ret

          end

```

Defining strings

To define a string:

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

To define a string which includes a trailing zero:

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

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

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

Reserving space

To reserve space for 10 bytes:

```
table DS8 10
```

Assembler control directives

These directives provide control over the operation of the assembler. See *Expression restrictions*, page 11, 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 26: 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 `ILINK` should be written in upper case in the `ILINK` definition file.

EXAMPLES

Defining comments

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

```
/*
Program to read serial input.
Version 1: 19.2.02
Author: mjp
*/
```

See also, *Comments in C-style preprocessor directives*, page 77.

Changing the base

To set the default base to 16:

```
module radix
section .text:code

radix 16 ; With the default base set
ld a, 12 ; to 16, the immediate value
;... ; of the load instruction is
; interpreted as 0x12.

; To reset the base from 16 to 10 again, the argument must be
; written in hexadecimal format.

radix 0x0a ; Reset the default base to 10.
ld a, 12 ; Now, the immediate value of
;... ; the load instruction is
; interpreted as 0x0c.

end
```

Controlling case sensitivity

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

```
module caseSensitivity1
section .text:code
```

```

                                caseoff
label      nop                    ; Stored as "LABEL".
           jra      LABEL
           end

```

The following will generate a duplicate label error:

```

                                module caseSensitivity2
                                section .text:code

                                caseoff
label      nop                    ; Stored as "LABEL".
LABEL     nop                    ; Error, "LABEL" already defined.
           end

```

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. |

Table 27: Call frame information directives

| Directive | Description |
|------------------------|---|
| 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 27: Call frame information directives (Continued)

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 section [, cfa section] ...
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 93). |

| | |
|------------------------|---|
| <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 shrinks 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 toward higher addresses, the factor is negative; if it grows toward lower addresses, the factor is positive. 1 is the default, but a larger value shrinks the produced backtrace information in size. The possible ranges are -256 to -1 and 1 to 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>section</i> | The name of a section. |
| <i>size</i> | The size of the frame cell in bytes. |
| <i>type</i> | The memory type, such as CODE, CONST or DATA. In addition, any of the memory types supported by the IAR ILINK 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 must 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 might 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 can 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.

To declare more than one resource, separate them with commas.

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

```
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 section type (to get the address space). To declare more than one stack frame CFA, separate them with commas.

When going “back” in the call stack, the value of the stack frame CFA is copied into the associated stack pointer resource to get a correct value for the previous function frame.

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

```
CFI STATICOVERLAYFRAME cfa section
```

The parameters are the name of the CFA and the name of the section where the static overlay for the function is located. To declare more than one static overlay frame CFA, separate 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 section type. To declare more than one base address CFA, separate 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 must 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 section type. You must 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 85. For more information on these directives, see *Simple rules*, page 91, and *CFI expressions*, page 93.

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 section control directive can 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 can manipulate the values of the columns by using the directives listed last in *Data block directives*, page 86. For more information on these directives, see *Simple rules*, page 91, and *CFI expressions*, page 93.

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) }
```

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

In those rare cases where the descriptive power of the simple rules are not enough, you can use a full CFI expression to describe the information (see *CFI expressions*, page 93). 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 85.

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

You can use call frame information expressions (CFI expressions) 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 these:

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

Unary operators

Overall syntax: `OPERATOR(operand)`

| Operator | Operand | Description |
|-------------------------|----------------|---|
| <code>COMPLEMENT</code> | <i>cfiexpr</i> | Performs a bitwise NOT on a CFI expression. |

Table 28: Unary operators in CFI expressions

| Operator | Operand | Description |
|----------|----------------|---|
| 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. |
| NOT | <i>cfiexpr</i> | Negates a logical CFI expression. |
| UMINUS | <i>cfiexpr</i> | Performs arithmetic negation on a CFI expression. |

Table 28: Unary operators in CFI expressions (Continued)

Binary operators

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

| Operator | Operands | Description |
|----------|-------------------------|--|
| ADD | <i>cfiexpr, cfiexpr</i> | Addition |
| AND | <i>cfiexpr, cfiexpr</i> | Bitwise AND |
| DIV | <i>cfiexpr, cfiexpr</i> | Division |
| EQ | <i>cfiexpr, cfiexpr</i> | Equal |
| GE | <i>cfiexpr, cfiexpr</i> | Greater than or equal |
| GT | <i>cfiexpr, cfiexpr</i> | Greater than |
| LE | <i>cfiexpr, cfiexpr</i> | Less 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. |
| LT | <i>cfiexpr, cfiexpr</i> | Less than |
| MOD | <i>cfiexpr, cfiexpr</i> | Modulo |
| MUL | <i>cfiexpr, cfiexpr</i> | Multiplication |
| NE | <i>cfiexpr, cfiexpr</i> | Not equal |
| OR | <i>cfiexpr, cfiexpr</i> | Bitwise OR |
| 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 is 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. |
| SUB | <i>cfiexpr, cfiexpr</i> | Subtraction |
| XOR | <i>cfiexpr, cfiexpr</i> | Bitwise XOR |

Table 29: Binary operators in CFI expressions

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. |
| 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 section type <i>type</i> of size <i>size</i> . |

Table 30: Ternary operators in CFI expressions

EXAMPLE

The following is a generic example and not an example specific to the STM8 microcontroller. This simplifies the example and clarifies the usage of the CFI directives. To obtain a target-specific example, generate assembler output when you compile a C source file.

Consider a generic processor with a stack pointer *SP*, and two registers *R0* and *R1*. Register *R0* is used as a scratch register (the register is destroyed by the function call), whereas register *R1* must be restored after the function call. For reasons of simplicity, all instructions, registers, and addresses 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 toward 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 31: 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 cannot 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 STM8.

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

Summary of pragma directives

This table shows the pragma directives of the assembler:

| #pragma directive | Description |
|--------------------------|---|
| #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 32: 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, and printed in the optional list file. In the IAR Embedded Workbench IDE, diagnostic messages are displayed in the Build messages window.

Severity levels

The diagnostics are divided into different levels of severity:

Remark

A diagnostic message that is produced when the assembler finds a source code construct that can possibly lead to erroneous behavior in the generated code. Remarks are, by default, not issued but can be enabled, see `--remarks`, page 29.

Warning

A diagnostic message that is produced when the assembler finds a programming error or omission which is of concern but not so severe as to prevent the completion of compilation. Warnings can be disabled with the command line option `--no_warnings`, see `--no_warnings`, page 27.

Error

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

Fatal error

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

SETTING THE SEVERITY LEVEL

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

See *Summary of assembler options*, page 16, 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 was a serious and unexpected failure due to a fault in the assembler. It is produced using this form:

Internal error: *message*

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

- The product name
- The version number of the assembler, which can be seen in the header of the list files generated by the assembler
- Your license number
- The exact internal error message text
- The source file of the program that generated the internal error
- A list of the options that were used when the internal error occurred.

A

- absolute expressions 11
- ADD (CFI operator) 94
- address field, in assembler list file 13
- ALIGN (assembler directive) 55
- alignment, of sections 56
- ALIGNRAM (assembler directive) 55
- AND (assembler operator) 36
- AND (CFI operator) 94
- architecture, STM8 xi
- _args (assembler directive) 63
- _args (predefined macro symbol) 66
- ASCII character constants 7
- asm (filename extension) 3
- assembler control directives 82
- assembler diagnostics 101
- assembler directives
 - assembler control 82
 - call frame information (CFI) 84
 - conditional assembly 60
 - See also* C-style preprocessor directives
 - C-style preprocessor 74
 - data definition or allocation 79
 - list file control 70
 - macro processing 63
 - module control 51
 - segment control 55
 - summary 47
 - symbol control 53
 - value assignment 58
 - #pragma 99
- assembler environment variables 4
- assembler expressions 6
- assembler instructions 5
- assembler labels 8
 - format of 5
- assembler list files
 - address field 13
 - comments 82
 - conditional code and strings 71
 - cross-references, generating 25, 71
 - data field 13
 - disabling 71
 - enabling 71
 - filename, specifying 25
 - generated lines, controlling 71
 - macro-generated lines, controlling 71
 - symbol and cross-reference table 13
- assembler macros
 - arguments, passing to 66
 - defining 64
 - generated lines, controlling in list file 71
 - in-line routines 67
 - predefined symbol 66
 - processing 67
 - quote characters, specifying 26
 - special characters, using 65
- assembler operators 31
 - in expressions 6
 - precedence 31
- assembler options
 - passing to assembler 3
 - specifying parameters 16
 - summary 16
- assembler output, including debug information 20
- assembler source files, including 76
- assembler source format 5
- assembler subversion number 10
- assembler symbols 8
 - exporting 53
 - importing 54
 - in relocatable expressions 11
 - predefined 9
 - redefining 59
- assembler, invocation syntax 3
- assembling, syntax 3
- ASSIGN (assembler directive) 58

assumptions (programming experience) xi

B

backtrace information, defining 84
BINAND (assembler operator) 36
BINNOT (assembler operator) 36
BINOR (assembler operator) 37
BINXOR (assembler operator) 37
bold style, in this guide xiii
__BUILD_NUMBER__ (predefined symbol) 9
BYTE1 (assembler operator) 37
BYTE2 (assembler operator) 37
BYTE3 (assembler operator) 38
BYTE4 (assembler operator) 38

C

call frame information directives 84
case sensitivity, controlling 18, 83
CASEOFF (assembler directive) 82
CASEON (assembler directive) 82
--case_insensitive (assembler option) 18
CFI directives 84
CFI expressions 93
CFI operators 93
character constants, ASCII 7
--code_model (assembler option) 18
COL (assembler directive) 70
command line options
 part of invocation syntax 3
 passing 3
 typographic convention xiii
command line, extending 24
command prompt icon, in this guide xiii
comments
 in assembler list file 82
 in assembler source code 5
 multi-line, using with assembler directives 83

comments, in C-style preprocessor directives 77
COMPLEMENT (CFI operator) 93
computer style, typographic convention xiii
conditional assembly directives 60
 See also C-style preprocessor directives
conditional code and strings, listing 71
constants
 default base of 82
 integer 6
conventions, used in this guide xii
copyright notice ii
--core (assembler option) 19
CRC, in assembler list file 13
cross-references, in assembler list file 25, 71
C-style preprocessor directives 74
C++ terminology xii

D

-D (assembler option) 19
data allocation directives 79
data definition directives 79
data field, in assembler list file 13
--data_model (assembler option) 20
__DATE__ (predefined symbol) 9
DATE (assembler operator) 38
DB (assembler directive) 79
DC8 (assembler directive) 79
DC16 (assembler directive) 79
DC24 (assembler directive) 79
DC32 (assembler directive) 79
DC64 (assembler directive) 79
--debug (assembler option) 20
debug information, including in assembler output 20
default base, for constants 82
#define (assembler directive) 74
DEFINE (assembler directive) 58
--dependencies (assembler option) 20
DF32 (assembler directive) 79

- DF64 (assembler directive) 79
 - diagnostic messages 101
 - classifying as errors 21
 - classifying as remarks 22
 - classifying as warnings 22
 - disabling warnings 27
 - disabling wrapping of 27
 - enabling remarks 29
 - listing all 23
 - suppressing 22
 - diagnostics_tables (assembler option) 23
 - diag_default (#pragma directive) 99
 - diag_error (assembler option) 21
 - diag_error (#pragma directive) 100
 - diag_remark (assembler option) 22
 - diag_remark (#pragma directive) 100
 - diag_suppress (assembler option) 22
 - diag_suppress (#pragma directive) 100
 - diag_warning (assembler option) 22
 - diag_warning (#pragma directive) 100
 - directives. *See* assembler directives
 - dir_first (assembler option) 23
 - disclaimer ii
 - DIV (CFI operator) 94
 - document conventions xii
 - DQ15 (assembler directive) 79
 - DQ31 (assembler directive) 79
 - DS8 (assembler directive) 79
 - DS16 (assembler directive) 79
 - DS24 (assembler directive) 79
 - DS32 (assembler directive) 79
 - DS64 (assembler directive) 79
 - DW (assembler directive) 79
- ## E
- edition, of this guide ii
 - efficient coding techniques 14
 - #elif (assembler directive) 74
 - #else (assembler directive) 74
 - ELSE (assembler directive) 60
 - ELSEIF (assembler directive) 60
 - enable_multibytes (assembler option) 23
 - END (assembler directive) 51
 - #endif (assembler directive) 74
 - ENDIF (assembler directive) 61
 - ENDM (assembler directive) 63
 - ENDR (assembler directive) 63
 - environment variables
 - assembler 4
 - IASMSTM8 4
 - IASTM8_INC 4
 - EQ (assembler operator) 38
 - EQ (CFI operator) 94
 - EQU (assembler directive) 58
 - #error (assembler directive) 74
 - error messages 101
 - classifying 21
 - #error, using to display 77
 - error_limit (assembler option) 24
 - EVEN (assembler directive) 55
 - EXITM (assembler directive) 63
 - experience, programming xi
 - expressions 6
 - extended command line file (extend.xcl) 24
 - EXTERN (assembler directive) 53
- ## F
- f (assembler option) 24
 - false value, in assembler expressions 8
 - fatal error messages 102
 - __FILE__ (predefined symbol) 9
 - file dependencies, tracking 20
 - file extensions. *See* filename extensions
 - file types
 - assembler source 3
 - extended command line 24

| | |
|---|----|
| #include, specifying path | 24 |
| filename extensions | |
| asm | 3 |
| msa | 3 |
| s | 3 |
| xcl | 24 |
| filenames, specifying for assembler object file | 28 |
| floating-point constants | 7 |
| formats, assembler source code | 5 |
| fractions | 7 |
| FRAME (CFI operator) | 95 |

G

| | |
|-----------------------------------|----|
| GE (assembler operator) | 39 |
| GE (CFI operator) | 94 |
| global value, defining | 59 |
| GT (assembler operator) | 39 |
| GT (CFI operator) | 94 |

H

| | |
|---|----|
| header files, SFR. | 14 |
| --header_context (assembler option) | 24 |
| HIGH (assembler operator) | 39 |
| HWRD (assembler operator) | 39 |

I

| | |
|---|------|
| -I (assembler option) | 24 |
| IAR Technical Support | 102 |
| __IAR_SYSTEMS_ASM__ (predefined symbol) | 10 |
| IASMSTM8 (environment variable) | 4 |
| __IASTM8__ (predefined symbol) | 9 |
| IASTM8_INC (environment variable) | 4 |
| icons, in this guide | xiii |
| #if (assembler directive) | 74 |
| IF (assembler directive) | 61 |
| IF (CFI operator) | 95 |

| | |
|--|------|
| #ifdef (assembler directive) | 74 |
| #ifndef (assembler directive) | 74 |
| IMPORT (assembler directive) | 53 |
| #include files, specifying | 24 |
| #include (assembler directive) | 74 |
| include paths, specifying | 24 |
| instruction set, STM8 | xi |
| integer constants | 6 |
| internal error | 102 |
| invocation syntax | 3 |
| in-line coding, using macros | 67 |
| italic style, in this guide | xiii |

L

| | |
|---|------|
| L: (operand modifier) | 12 |
| -l (assembler option) | 25 |
| labels. <i>See</i> assembler labels | |
| LE (assembler operator) | 40 |
| LE (CFI operator) | 94 |
| LIBRARY (assembler directive) | 49 |
| lightbulb icon, in this guide | xiii |
| __LINE__ (predefined symbol) | 10 |
| #line (assembler directive) | 74 |
| list file format | 13 |
| body | 13 |
| CRC | 13 |
| header | 13 |
| symbol and cross reference | 13 |
| listing control directives | 70 |
| LITERAL (CFI operator) | 94 |
| LOAD (CFI operator) | 95 |
| local value, defining | 59 |
| LOCAL (assembler directive) | 63 |
| LOW (assembler operator) | 40 |
| LSHIFT (CFI operator) | 94 |
| LSTCND (assembler directive) | 70 |
| LSTCOD (assembler directive) | 70 |
| LSTEXP (assembler directives) | 70 |

| | |
|---|----|
| LSTMAC (assembler directive) | 70 |
| LSTOUT (assembler directive) | 70 |
| LSTPAGE (assembler directive) | 70 |
| LSTREP (assembler directive) | 70 |
| LSTXRF (assembler directive) | 70 |
| LT (assembler operator) | 40 |
| LT (CFI operator) | 94 |
| LWRD (assembler operator) | 40 |

M

| | |
|---|-----|
| -M (assembler option) | 26 |
| macro processing directives | 63 |
| macro quote characters | 65 |
| specifying | 26 |
| MACRO (assembler directive) | 63 |
| macros. <i>See</i> assembler macros | |
| memory space, reserving and initializing | 80 |
| memory, reserving space in | 79 |
| message (#pragma directive) | 100 |
| messages, excluding from standard output stream | 30 |
| --mnm_first (assembler option) | 27 |
| MOD (assembler operator) | 41 |
| MOD (CFI operator) | 94 |
| module consistency | 52 |
| module control directives | 51 |
| modules, beginning | 51 |
| msa (filename extension) | 3 |
| MUL (CFI operator) | 94 |
| multibyte character support | 23 |

N

| | |
|--------------------------------------|------|
| NAME (assembler directive) | 51 |
| naming conventions | xiii |
| NE (assembler operator) | 41 |
| NE (CFI operator) | 94 |
| NOT (assembler operator) | 41 |
| NOT (CFI operator) | 94 |

| | |
|---|----|
| --no_fragments (assembler option) | 27 |
| --no_path_in_file_macros (assembler option) | 27 |
| --no_warnings (assembler option) | 27 |
| --no_wrap_diagnostics (assembler option) | 27 |

O

| | |
|--|----|
| -o (assembler option) | 28 |
| ODD (assembler directive) | 55 |
| --only_stdout (assembler option) | 28 |
| operand modifiers | 12 |
| operands | |
| format of | 5 |
| in assembler expressions | 6 |
| operations, format of | 5 |
| operation, silent | 30 |
| operators. <i>See</i> assembler operators | |
| option summary | 16 |
| OR (assembler operator) | 41 |
| OR (CFI operator) | 94 |
| --output (assembler option) | 28 |
| OVERLAY (assembler directive) | 53 |

P

| | |
|--|--------|
| PAGE (assembler directive) | 70 |
| PAGSIZ (assembler directive) | 70 |
| parameters | |
| specifying | 16 |
| typographic convention | xiii |
| part number, of this guide | ii |
| #pragma (assembler directive) | 74, 99 |
| precedence, of assembler operators | 31 |
| predefined register symbols | 9 |
| predefined symbols | 9 |
| in assembler macros | 66 |
| --predef_macros (assembler option) | 28 |
| prefix to operands | 12 |
| --preinclude (assembler option) | 28 |

| | |
|--|----|
| --preprocess (assembler option) | 29 |
| preprocessor symbols | |
| defining and undefining | 75 |
| defining on command line | 19 |
| prerequisites (programming experience) | xi |
| program location counter (PLC) | 8 |
| PROGRAM (assembler directive) | 51 |
| programming experience, required | xi |
| programming hints | 14 |
| PUBLIC (assembler directive) | 53 |
| publication date, of this guide | ii |
| PUBWEAK (assembler directive) | 53 |

R

| | |
|---|------|
| -r (assembler option) | 20 |
| RADIX (assembler directive) | 82 |
| reference information, typographic convention | xiii |
| registered trademarks | ii |
| registers | 9 |
| relocatable expressions | 11 |
| remark (diagnostic message) | 101 |
| classifying | 22 |
| enabling | 29 |
| --remarks (assembler option) | 29 |
| repeating statements | 67 |
| REPT (assembler directive) | 63 |
| REPTC (assembler directive) | 63 |
| REPTI (assembler directive) | 63 |
| REQUIRE (assembler directive) | 53 |
| RSEG (assembler directive) | 55 |
| RSHIFTA (CFI operator) | 94 |
| RSHIFTL (CFI operator) | 94 |
| RTMODEL (assembler directive) | 51 |
| rules, in CFI directives | 91 |
| runtime model attributes, declaring | 52 |

S

| | |
|--|-----|
| S: (operand modifier) | 12 |
| s (filename extension) | 3 |
| SECTION (assembler directive) | 55 |
| sections | |
| aligning | 56 |
| beginning | 56 |
| SECTION_TYPE (assembler directive) | 55 |
| segment control directives | 55 |
| SET (assembler directive) | 58 |
| severity level, of diagnostic messages | 101 |
| specifying | 102 |
| SFB (assembler operator) | 42 |
| SFE (assembler operator) | 42 |
| SFR. <i>See</i> special function registers | |
| SHL (assembler operator) | 43 |
| SHR (assembler operator) | 43 |
| --silent (assembler option) | 30 |
| silent operation, specifying | 30 |
| simple rules, in CFI directives | 91 |
| SIZEOF (assembler operator) | 44 |
| source files, including | 76 |
| source files, list all referred | 24 |
| source format, assembler | 5 |
| source line numbers, changing | 78 |
| special function registers | 14 |
| standard error | 28 |
| standard output stream, disabling messages to | 30 |
| standard output, specifying | 28 |
| statements, repeating | 67 |
| stderr | 28 |
| stdout | 28 |
| STM8 architecture and instruction set | xi |
| SUB (CFI operator) | 94 |
| __SUBVERSION__ (predefined symbol) | 10 |
| Support, Technical | 102 |
| symbol and cross-reference table, in assembler list file | 13 |
| <i>See also</i> Include cross-reference | 13 |

| | |
|-----------------------------------|----|
| symbol control directives | 53 |
| symbols | |
| <i>See also</i> assembler symbols | |
| exporting to other modules | 54 |
| predefined, in assembler | 9 |
| predefined, in assembler macro | 66 |
| user-defined, case sensitive | 18 |

T

| | |
|--------------------------------------|------|
| Technical Support, IAR | 102 |
| temporary values, defining | 59 |
| terminology | xii |
| __TIME__ (predefined symbol) | 10 |
| time-critical code | 67 |
| tools icon, in this guide | xiii |
| trademarks | ii |
| true value, in assembler expressions | 8 |
| typographic conventions | xiii |

U

| | |
|------------------------------|----|
| UGT (assembler operator) | 44 |
| ULT (assembler operator) | 45 |
| UMINUS (CFI operator) | 94 |
| #undef (assembler directive) | 74 |
| UPPER (assembler operator) | 45 |
| user symbols, case sensitive | 18 |

V

| | |
|-----------------------------|----|
| value assignment directives | 58 |
| values, defining | 79 |
| VAR (assembler directive) | 58 |
| __VER__ (predefined symbol) | 10 |
| version | |
| IAR Embedded Workbench | ii |
| of assembler | 10 |

W

| | |
|--|-------|
| warnings | 101 |
| classifying | 22 |
| disabling | 27 |
| exit code | 30 |
| treating as errors | 30 |
| warnings icon, in this guide | xiii |
| --warnings_affect_exit_code (assembler option) | 4, 30 |
| --warnings_are_errors (assembler option) | 30 |

X

| | |
|--------------------------|----|
| xcl (filename extension) | 24 |
| XOR (assembler operator) | 45 |
| XOR (CFI operator) | 94 |

Symbols

| | |
|---|----|
| ^ (assembler operator) | 37 |
| _args (assembler directive) | 63 |
| _args (predefined macro symbol) | 66 |
| __BUILD_NUMBER__ (predefined symbol) | 9 |
| __DATE__ (predefined symbol) | 9 |
| __FILE__ (predefined symbol) | 9 |
| __IAR_SYSTEMS_ASM__ (predefined symbol) | 10 |
| __IASTM8__ (predefined symbol) | 9 |
| __LINE__ (predefined symbol) | 10 |
| __SUBVERSION__ (predefined symbol) | 10 |
| __TIME__ (predefined symbol) | 10 |
| __VER__ (predefined symbol) | 10 |
| - (assembler operator) | 35 |
| -D (assembler option) | 19 |
| -f (assembler option) | 24 |
| -I (assembler option) | 24 |
| -l (assembler option) | 25 |
| -M (assembler option) | 26 |
| -o (assembler option) | 28 |
| -r (assembler option) | 20 |

| | | | |
|--|-------|--|--------|
| --case_insensitive (assembler option) | 18 | #define (assembler directive) | 74 |
| --code_model (assembler option) | 18 | #elif (assembler directive) | 74 |
| --core (assembler option) | 19 | #else (assembler directive) | 74 |
| --data_model (assembler option) | 20 | #endif (assembler directive) | 74 |
| --debug (assembler option) | 20 | #error (assembler directive) | 74 |
| --dependencies (assembler option) | 20 | #if (assembler directive) | 74 |
| --diagnostics_tables (assembler option) | 23 | #ifdef (assembler directive) | 74 |
| --diag_error (assembler option) | 21 | #ifndef (assembler directive) | 74 |
| --diag_remark (assembler option) | 22 | #include files, specifying | 24 |
| --diag_suppress (assembler option) | 22 | #include (assembler directive) | 74 |
| --diag_warning (assembler option) | 22 | #line (assembler directive) | 74 |
| --dir_first (assembler option) | 23 | #pragma (assembler directive) | 74, 99 |
| --enable_multibytes (assembler option) | 23 | #undef (assembler directive) | 74 |
| --error_limit (assembler option) | 24 | % (assembler operator) | 41 |
| --header_context (assembler option) | 24 | + (assembler operator) | 34–35 |
| --mnem_first (assembler option) | 27 | < (assembler operator) | 40 |
| --no_fragments (assembler option) | 27 | << (assembler operator) | 43 |
| --no_path_in_file_macros (assembler option) | 27 | <= (assembler operator) | 40 |
| --no_warnings (assembler option) | 27 | <> (assembler operator) | 41 |
| --no_wrap_diagnostics (assembler option) | 27 | = (assembler directive) | 58 |
| --only_stdout (assembler option) | 28 | = (assembler operator) | 38 |
| --output (assembler option) | 28 | == (assembler operator) | 38 |
| --predef_macros (assembler option) | 28 | > (assembler operator) | 39 |
| --preinclude (assembler option) | 28 | >= (assembler operator) | 39 |
| --preprocess (assembler option) | 29 | >> (assembler operator) | 43 |
| --remarks (assembler option) | 29 | (assembler operator) | 37 |
| --silent (assembler option) | 30 | (assembler operator) | 41 |
| --warnings_affect_exit_code (assembler option) | 4, 30 | ~ (assembler operator) | 36 |
| --warnings_are_errors (assembler option) | 30 | \$ (program location counter) | 8 |
| ! (assembler operator) | 41 | | |
| != (assembler operator) | 41 | | |
| ?: (assembler operator) | 36 | | |
| () (assembler operator) | 34 | | |
| * (assembler operator) | 34 | | |
| / (assembler operator) | 35 | | |
| /*...*/ (assembler directive) | 82 | | |
| // (assembler directive) | 82 | | |
| & (assembler operator) | 36 | | |
| && (assembler operator) | 36 | | |