ColdFire® IAR Assembler

Reference Guide

for Freescale's

ColdFire Microcontroller Family

EDITION NOTICE

First edition: September 2007

Part number: ACF-1

This guide applies to version 1.x of the ColdFire® IAR Embedded Workbench® IDE.

Internal reference: ISUD.

Contents

Tables vii

Preface ix

Who should read this guide ix

How to use this guide ix

What this guide contains x

Other documentation x

Document conventions xi

Introduction to the ColdFire IAR Assembler 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 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 assembler options 15
Specifying parameters 16
Summary of assembler options 17
Description of assembler options 18

Assembler operators 33

Precedence of operators 33
Summary of assembler operators 34
Parenthesis operator – 1 34
Function operators – 2 34
Unary operators – 3 34
Multiplicative arithmetic operators – 4 35
Additive arithmetic operators – 5 35
Shift operators – 6 35
Comparison operators – 7 35
Equivalence operators – 8 35
Logical operators – 9-14 35
Conditional operator – 15 36
Description of assembler operators 36

Assembler directives 49

Summary of assembler directives 49
Module control directives 53
Syntax 53
Parameters 54

Descriptions 54
Symbol control directives 56
Syntax 57
Parameters 57
Descriptions 57
Examples 58
Segment control directives 59
Syntax 59
Parameters 60
Descriptions 60
Examples 62
Value assignment directives 64
Syntax 64
Parameters 64
Descriptions 64
Examples 65
Conditional assembly directives 68
Syntax 68
Parameters 68
Descriptions 69
Examples 69
Macro processing directives 70
Syntax 70
Parameters 70
Descriptions 71
Examples 74
Listing control directives 77
Syntax 78
Descriptions 78
Examples 79
C-style preprocessor directives 81
Syntax 81
Parameters 82
Descriptions 82

Examples 84
Data definition or allocation directives 85
Syntax 86
Parameters 87
Descriptions 87
Examples 87
Assembler control directives 88
Syntax 89
Parameters 89
Descriptions 89
Examples 89
Function directives 90
Syntax 91
Parameters 91
Descriptions 91
Call frame information directives 92
Syntax 93
Parameters 94
Descriptions 95
Simple rules 99
CFI expressions 101
Example 103

Pragma directives 107

Summary of pragma directives 107
Descriptions of pragma directives 107

Diagnostics 109

Message format 109
Severity levels 109
Setting the severity level 110
Internal error 110

Index 111

Tables

1: Typographic conventions used in this guide xi
2: Assembler environment variables 4
3: Assembler error return codes 4
4: Integer constant formats 6
5: ASCII character constant formats 7
6: Floating-point constants 7
7: Predefined register symbols 9
8: Predefined symbols 9
9: Symbol and cross-reference table 14
10: Assembler options summary 17
11: Generating a list of dependencies (--dependencies) 20
12: Conditional list options (-l) 25
13: Directing preprocessor output to file (--preprocess) 29
14: Assembler directives summary 49
15: Module control directives 53
16: Symbol control directives 56
17: Segment control directives 59
18: Value assignment directives 64
19: Conditional assembly directives 68
20: Macro processing directives 70
21: Listing control directives 77
22: C-style preprocessor directives 81
23: Data definition or allocation directives 85
24: Assembler control directives 88
25: Call frame information directives 92
26: Unary operators in CFI expressions 101
27: Binary operators in CFI expressions 101
28: Ternary operators in CFI expressions 102
29: Code sample with backtrace rows and columns 103
30: Pragma directives summary 107

Preface

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

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

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

If you are new to using the IAR Systems toolkit, we recommend that you first read the initial chapters of the *IAR Embedded Workbench® IDE User Guide*. They give product overviews, as well as tutorials that can help you get started.

What this guide contains

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

- *Introduction to the ColdFire IAR Assembler* 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 ColdFire 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 ColdFire IAR C/C++ Compiler, refer to the *ColdFire® IAR C/C++ Compiler Reference Guide*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR DLIB Library, refer to the online help system.

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

Document conventions

This guide uses the following typographic conventions:

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

Table 1: Typographic conventions used in this guide

Introduction to the ColdFire IAR Assembler

This chapter contains the following 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 may be situations where you will find it necessary to write parts of the code in assembler, for example, when using mechanisms in the ColdFire 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 ColdFire microcontroller. Refer to Freescale's hardware documentation for syntax descriptions of the instruction mnemonics.

GETTING STARTED

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

- Work through the tutorials—especially the one about mixing C and assembler modules—that you find in the *IAR Embedded Workbench® IDE User Guide*
- Read about the assembler language interface—also useful when mixing C and assembler modules—in the *ColdFire® IAR C/C++ Compiler Reference Guide*

- 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. By structuring 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. In each source file, you define one or several assembler *modules* by using the module control directives. Each module has a name and a type, where the type can be either `PROGRAM` or `LIBRARY`. The linker will always include a `PROGRAM` module, whereas a `LIBRARY` module is only included in the linked code if other modules refer to a public symbol in the module. Each module can be further divided into subroutines.

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

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, you can collect modules that contain such routines in a *library* object file. In the IAR Embedded Workbench IDE, it is possible to 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, either one per source file, or many modules per file by using the module directives

- In each module, divide your assembler source code into small subroutines (corresponding to *functions* on the C level)
- Divide your assembler source code into *segments*, to gain more precise control of how your code and data finally will be 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:

```
acf [options][sourcefile][options]
```

For example, when assembling the source file `prog.s68`, use the following command to generate an object file with debug information:

```
acf prog --debug
```

By default, the ColdFire IAR Assembler recognizes the filename extensions `s68`, `asm`, and `msa` for source files. The default filename extension for assembler output is `r68`.

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

There are three different ways of passing options to the assembler:

- Directly from the command line

  Specify the options on the command line after the `acf` 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 23.

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 `ASMCF` environment variable. The assembler automatically appends the value of this variable to every command line, so it provides a convenient method of specifying options that are required for every assembly.

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

Environment variable	Description
`ASMCF`	Specifies command line options; for example: `set ASMCF=-L -ws`
`ASMCF_INC`	Specifies directories to search for include files; for example: `set ASMCF_INC=c:\myinc\`

Table 2: Assembler environment variables

For example, setting the following environment variable will always generate a list file with the name `temp.lst`:

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

For information about the environment variables used by the IAR XLINK Linker and the IAR XLIB Librarian, see the *IAR Linker and Library Tools Reference Guide*.

ERROR RETURN CODES

When using the ColdFire IAR Assembler from within a batch file, you may need to determine whether the assembly was successful in order to decide what step to take next. For this reason, the assembler returns the following error return codes:

Return code	Description
`0`	Assembly successful, warnings may appear.
`1`	There were warnings, provided that the option --warnings_affect_exit_code was used.
`2`	There were non-fatal errors or fatal assembly errors (making the assembler abort).
3	There were crashing errors.

Table 3: Assembler error return codes

Source format

The format of an assembler source line is as follows:

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

where the components are as follows:

label	A definition of a label, which is a symbol that represents an address. If the label starts in the first column—that is, at the far left on the line—the `:` (colon) is optional.
operation	An assembler instruction or directive. This must not start in the first column—there must be some whitespace to the left of it.
operands	An assembler instruction or directive can have zero, one, or more operands. The operands are separated by commas. An operand can be: • a constant representing a numeric value or an address • a symbolic name representing a numeric value or an address (where the latter also is referred to as a label) • a floating-point constant • a register • a predefined symbol • the program location counter (PLC) • an expression.
comment	Comment, preceded by a `;` (semicolon) C or C++ comments are also allowed.

The components are separated by spaces or tabs.

A source line may not exceed 2047 characters.

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

Assembler instructions

The ColdFire IAR Assembler supports the syntax for assembler instructions as described in Freescale's documentation *ColdFire® Family Programmer's Reference Manual* (document number CFPRM, Rev3, 03/2005).

Expressions, operands, and operators

Expressions consist of expression operands and operators.

The assembler will accept a wide range of expressions, including both arithmetic and logical operations. All operators use 32-bit two's complement integers. Range checking is performed if a value is used for generating code.

Expressions are evaluated from left to right, unless this order is overridden by the precedence of operators; see also *Assembler operators*, page 33.

The following operands are valid in an expression:

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

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

INTEGER CONSTANTS

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

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

Commas and decimal points are not permitted.

The following types of number representation are supported:

Integer type	Example
Binary	`1010b`
Octal	`1234q`, `0123`
Decimal	`1234`, `-1`, `1234d`
Hexadecimal	`0FFFFh`, `0xFFFF`

Table 4: Integer constant formats

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

ASCII CHARACTER CONSTANTS

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

Format	Value
`'ABCD'`	`ABCD` (four characters).
`"ABCD"`	`ABCD'\0'` (five characters the last ASCII null).
`'A''B'`	`A'B`
`'A'''`	`A'`
`''''` (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 5: ASCII character constant formats

FLOATING-POINT CONSTANTS

The ColdFire IAR Assembler will accept floating-point values as constants and convert them into IEEE single-precision (signed 32-bit) floating-point format and double-precision (signed 64-bit), or fractional format.

Floating-point numbers can be written in the format:

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

The following 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 6: Floating-point constants

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

Note: Floating-point constants will 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 <= 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 long addresses. For additional information, see *Generating a lookup table*, page 87.

LABELS

Symbols used for memory locations are referred to as labels.

Program location counter (PLC)

The assembler keeps track of the start address of the current instruction. This is called the *program location counter.*

If you need to refer to the program location counter in your assembler source code you can use the $ (dollar) sign. For example:

```
BRA   $      ; Loop forever
```

REGISTER SYMBOLS

The following table shows the existing predefined register symbols:

Name	Size	Description
`A0–A6`	32 bits	Address registers
`A7/SP`	32 bits	Supervisor stack pointer
`OTHER_A7`	32 bits	User stack pointer
`D0–D7`	32 bits	Data registers
`PC`	32 bits	Program counter
`SR/CCR`	16 bits/8 bits	Status register / Conditional code register
`VBR`	32 bits	Vector base register

Table 7: Predefined register symbols

PREDEFINED SYMBOLS

The ColdFire IAR Assembler defines a set of symbols for use in assembler source files. The symbols provide information about the current assembly, allowing you to test them in preprocessor directives or include them in the assembled code. The strings returned by the assembler are enclosed in double quotes.

The following predefined symbols are available:

Symbol	Value
`__ACF__`	An integer that is set to `1` when the code is assembled with the ColdFire IAR Assembler.
`__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).
`__DIV__`	An integer that identifies whether support for the `DIV/MOD` instructions is enabled. The symbol reflects the `--no_div` option and is defined to `__NO_DIV__` or `__DIV_INSTR__`. These symbolic names can be used when testing the `__DIV__` symbol.
`__FILE__`	The name of the current source file (string).

Table 8: Predefined symbols

Symbol	Value
`__FPU__`	An integer that identifies whether support for the floating-point unit is enabled. The symbol reflects the `--fpu` option and is defined to `__NO_FPU__` or `__FP_UNIT__`. These symbolic names can be used when testing the `__FPU__` symbol.
`__IAR_SYSTEMS_ASM__`	IAR assembler identifier (number).
`__ISA__`	An integer that identifies the code model in use. The symbol reflects the `--isa` option and is defined to `__ISA_A__`, `__ISA_A_PLUS__`, `__ISA_B__`, or `__ISA_C__`. These symbolic names can be used when testing the `__ISA__` symbol.
`__LINE__`	The current source line number (number).
`__MAC__`	An integer that identifies whether support for MAC instructions is enabled. The symbol reflects the `--mac` option and is defined to `__NO_MAC__`, `__EMAC__`, `__EMAC_B__`, or `__MAC_UNIT__`. These symbolic names can be used when testing the `__MAC__` symbol.
`__SUPERVISOR__`	An integer that identifies whether support for the supervisor instructions is enabled. The symbol reflects the `--supervisor` option and is defined to `__NO_SUPERVISOR__` or `__SUPERVISOR_INSTR__`. These symbolic names can be used when testing the `__SUPERVISOR__` symbol.
`__TID__`	Target identity, consisting of two bytes (number). The high byte is the target identity, which is `68` for `ACF`. The low byte is the processor option *16. The value of the symbol is `0x4400`.
`__SUBVERSION__`	An integer that identifies the version letter of the version number, for example the C in 4.21C, as an ASCII character.
`__TIME__`	The current time in `hh:mm:ss` format (string).
`__VER__`	The version number in integer format; for example, version 4.17 is returned as 417 (number).

Table 8: Predefined symbols (Continued)

Note: The symbol `__TID__` is related to the predefined symbol `__TID__` in the ColdFire IAR C/C++ Compiler. It is described in the *ColdFire® IAR C/C++ Compiler Reference Guide*.

Including symbol values in code

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

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

```
      MODULE  time

      EXTERN  printstr

      RSEG    CODE

tim   DC8     __TIME__    ; Time string

      MOVE.B #tim, D0     ; Load address of string in D0
      JSR   (printstr).L  ; Call string output routine
      END
```

Testing symbols for conditional assembly

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

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

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

See *Conditional assembly directives*, page 68.

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 segments cannot be resolved at assembly time, because they depend on the location of segments. These are referred to as relocatable expressions.

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

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

```
        NAME prog1

        EXTERN third

        RSEG DATA

first:  DC8   5
second: DC8   3

        RSEG CODE
start …
```

Then in the segment CODE, the following relocatable expressions are legal:

```
        DC8 first
        DC8 first+1
        DC8 1+first
        DC8 (first/second)*third
```

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

EXPRESSION RESTRICTIONS

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

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

No forward

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

No external

No external references in the expression are allowed.

Absolute

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

Fixed

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

List file format

The format of an assembler list file is as follows:

HEADER

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

BODY

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

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

SUMMARY

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

SYMBOL AND CROSS-REFERENCE TABLE

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

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

Information	Description
Symbol	The symbol's user-defined name.
Mode	ABS (Absolute), or REL (Relocatable).
Segments	The name of the segment 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 segment part.

Table 9: Symbol and cross-reference table

Programming hints

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

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of ColdFire devices are included in the IAR Systems product package, in the `\cf\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 ColdFire IAR C/C++ Compiler, and they are suitable to use as templates when creating new header files for other ColdFire derivatives.

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

```
#ifdef __IAR_SYSTEMS_ASM__
  (assembler-specific defines)
#endif
```

USING C-STYLE PREPROCESSOR DIRECTIVES

The C-style preprocessor directives are processed before other assembler directives. Therefore, do not use preprocessor directives in macros and do not mix them with assembler-style comments. For more information about comments, see *Assembler control directives*, page 88.

Assembler options

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

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

Setting assembler options

To set assembler options from the command line, include them on the command line after the `acf` command, either before or after the source filename. For example, when assembling the source file `prog.s68`, use the following command to generate an object file with debug information:

```
acf 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`:

```
acf 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:

```
acf prog -DDEBUG=1
```

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

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

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

SPECIFYING PARAMETERS

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

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

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

or as

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

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

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

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

```
--diag_suppress=Pe0001
```

or

```
--diag_suppress Pe0001
```

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

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

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

```
acf prog -l .
```

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

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

```
acf prog -l ---
```

Summary of assembler options

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

Command line option	Description
`--case_insensitive`	Case-insensitive user symbols
`-D`	Defines preprocessor symbols
`--debug`	Generates debug information
`--dependencies`	Lists file dependencies
`--diag_error`	Treats these diagnostics as errors
`--diag_remark`	Treats these diagnostics as remarks
`--diag_suppress`	Suppresses these diagnostics
`--diag_warning`	Treats these diagnostics as warnings
`--diagnostics_tables`	Lists all diagnostic messages
`--dir_first`	Allows directives in the first column
`--enable_multibytes`	Enables support for multibyte characters
`--error_limit`	Specifies the allowed number of errors before the assembler stops
`-f`	Extends the command line
`--fpu`	Enables floating-point (FPU) instructions
`--header_context`	Lists all referred source files
`-I`	Includes file paths
`--isa`	Selects instruction set for architecture version
`-l`	Output list file
`-M`	Macro quote characters
`--mac`	Enables support for multiply-accumulate (MAC)
`--mnem_first`	Allows mnemonics in the first column
`--no_div`	Disables support for `DIVS`/`DIVU` and `REMS`/`REMU` instructions
`--no_path_in_file_macros`	Removes the path from the return value of the symbols `__FILE__` and `__BASE_FILE__`
`--no_supervisor`	Disables support for supervisor instructions
`--no_warnings`	Disables all warnings
`--no_wrap_diagnostics`	Disables wrapping of diagnostic messages

Table 10: Assembler options summary

Command line option	Description
`-o`	Sets object filename
`--only_stdout`	Uses standard output only
`--preinclude`	Includes an include file before reading the source file
`--preprocess`	Preprocessor output to file
`-r`	Generates debug information
`--remarks`	Enables remarks
`--silent`	Sets silent operation
`--warnings_affect_exit_code`	Warnings affect exit code
`--warnings_are_errors`	Treats all warnings as errors

Table 10: Assembler options summary (Continued)

Description of assembler options

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

Note that if you use the page **Extra Options** to specify specific command line options, there is no check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--case_insensitive

`--case_insensitive`

Use this option to make user symbols case insensitive.

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

You can also use the assembler directives `CASEON` and `CASEOFF` to control case sensitivity for user-defined symbols. See *Assembler control directives*, page 88, 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 82.

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

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

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

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

Example

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

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

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

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

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

```
acf prog -DFRAMERATE=3
```

Project>Options>Assembler>Preprocessor>Defined symbols

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

`-r`

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

In order to reduce the size and link time of the object file, the assembler does not generate debug information by default.

Project>Options>Assembler >Output>Generate debug information

`--dependencies`

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

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

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

Table 11: Generating a list of dependencies (--dependencies)

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

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

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

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

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

If `--dependencies=m` is used, the output uses makefile style. For each source file, one line containing a makefile dependency rule is output. Each line consists of the name of the object file, a colon, a space, and the name of a source file. For example:

```
foo.r68: c:\iar\product\include\stdio.h
foo.r68: d:\myproject\include\foo.h
```

Example 1

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

```
acf 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:

```
acf prog --dependencies \mypath\listing
```

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

Example 3

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

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

```
%.r68 : %.c
      $(ASM) $(ASMFLAGS) $< --dependencies=m $*.d
```

That is, besides producing an object file, the command also produces a dependent file in makefile style (in this example using the extension `.d`).

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

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

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

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

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

Use this option to classify diagnostic messages as errors.

An error indicates a violation of the assembler language rules, of such severity that object code will not be generated, and the exit code will not be 0.

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

```
--diag_error=As001
```

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

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

Use this option to classify diagnostic messages as remarks.

A remark is the least severe type of diagnostic message and indicates a source code construct that may cause strange behavior in the generated code.

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

```
--diag_remark=As001
```

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

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

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

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

Project>Options>Assembler >Diagnostics>Suppress these diagnostics

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

Use this option to classify diagnostic messages as warnings.

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

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

```
--diag_warning=As028
```

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

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

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

This option cannot be given together with other options.

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

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

Example 1

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

```
--diagnostics_tables diag
```

Example 2

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

```
--diagnostics_tables .
```

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

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

`--dir_first` `--dir_first`

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

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

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

---

`--enable_multibytes` `--enable_multibytes`

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

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

Project>Options>Assembler>Language>Enable multibyte support

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

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

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

`-f` `-f` *`filename`*

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

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

```
acf prog -f extend.xcl
```

To set this option, use:

Project>Options>Assembler>Extra Options

--fpu

`--fpu`

Use this option to enable floating-point (FPU) instructions.

To set this option, use:

Project>Options>Assembler>Extra Options

--header_context

`--header_context`

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

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

-I

`-I`*`prefix`*

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

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

Example

For example, using the options:

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

and then writing:

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

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

Project>Options>Assembler >Preprocessor>Additional include directories

--isa `--isa={isa_a|isa_a+|isa_b|isa_c}`

Use this option to select the version of the instruction set architecture to be used. If you do not use the option to specify what instruction set to use, the compiler uses the `isa_a` instruction set as default.

See also the device documentation delivered from the chip manufacturer.

Project>Options>General Options>Target>ISA variant

-l `-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 `LSTOUT` directive controls if lines are written to the list file or not. Using `-ld` 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 12: 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 will be the same as the name of the assembled source file, unless a different name has been specified with the option `-o`, in which case that name will be used.

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

Example 1

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

```
acf sourcefile -l list
```

Example 2

If you assemble the file `mysource.s68` and want to generate a listing to a file `mysource.lst` in the working directory, you could use:

```
acf mysource -l .
```

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

To set related options, select:

Project>Options>Assembler >List

`-M` `-Mab`

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

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

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

```
acf filename -M'<>'
```

Example

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

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

Project>Options>Assembler >Language>Macro quote characters

`--mac` `--mac={mac|emac|emab_b}`

Use this option to enable support for the multiply–accumulate instructions. If you do not use the option to specify what instruction set to use, the compiler does not support MAC instructions.

See also the device documentation delivered from the chip manufacturer.

Project>Options>General Options>Target>MAC variant

--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_div

```
--no_div
```

Use this option to disable support for the `DIVS`/`DIVU` and `REMS`/`REMU` instructions.

See also the device documentation delivered from the chip manufacturer.

Project>Options>General Options>Target>No DIV/REM instruction

--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_supervisor

```
--no_supervisor
```

Use this option to disable support for supervisor instructions.

See also the device documentation delivered from the chip manufacturer.

To set this option, use:

Project>Options>Assembler>Extra Options

--no_warnings

```
--no_warnings
```

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

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

`--no_wrap_diagnostics`

`--no_wrap_diagnostics`

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

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

`-o`

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

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

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

If a *`directory`* is specified, the assembler stores the object code in that directory, in a file with the same name as the name of the assembled source file, but with the extension `r68`. To specify the working directory, replace *`directory`* with a period (`.`).

Example 1

To store the assembler output in a file called `obj.r68` in the `mypath` directory, you would use:

```
acf sourcefile -o \mypath\obj
```

Example 2

If you assemble the file `mysource.s68` and want to store the assembler output in a file `mysource.r68` in the working directory, you could use:

```
acf mysource -o .
```

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

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

`--only_stdout`

`--only_stdout`

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

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

`--preinclude` `--preinclude includefile`

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

To set this option, use:

Project>Options>Assembler>Extra Options

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

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

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

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

Table 13: Directing preprocessor output to file (--preprocess)

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

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

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

Example 1

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

```
acf sourcefile --preprocess=c output
```

Example 2

If you assemble the file `mysource.s68` and want to store the assembler output with `#line` directives to a file `mysource.i` in the working directory, you could use:

```
acf mysource --preprocess=l .
```

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

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

`-r, --debug`

`--debug`

-r

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

In order to reduce the size and link time of the object file, the assembler does not generate debug information by default.

Project>Options>Assembler >Output>Generate debug information

`--remarks`

`--remarks`

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

See *Severity levels*, page 109, for additional information about diagnostic messages.

Project>Options>Assembler >Diagnostics>Enable remarks

`--silent`

`--silent`

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

By default, the assembler sends various insignificant messages via the standard output stream. You can use the `--silent` option to prevent this. The assembler sends error and warning messages to the error output stream, so they are displayed regardless of this setting.

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

`--warnings_affect_exit_code`

`--warnings_affect_exit_code`

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

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

`--warnings_are_errors`

`--warnings_are_errors`

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

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

```
--diag_warning=As001
```

For additional information, see --*diag_warning*, page 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).

The following rules determine how expressions are evaluated:

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

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

Note: The precedence order in the ColdFire IAR Assembler 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`	Segments begin.
`SFE`	Segments end.
`SIZEOF`	Segments 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 precedence of the operator

`()` Parenthesis (1).

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

Example

```
1+2*3 → 7
(1+2)*3 → 9
```

`*` Multiplication (4).

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

Example

```
2*2 → 4
-2*2 → -4
```

`+` Unary plus (3).

Unary plus operator.

Example

```
+3 → 3
3*+2 → 6
```

`+` Addition (5).

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

Example

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

`-` Unary minus (3).

The unary minus operator performs arithmetic negation on its operand.

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

Example

```
-3 → -3
3*-2 → -6
4--5 → 9
```

`-` Subtraction (5).

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

Example

```
92-19 → 73
-2-2 → -4
-2--2 → 0
```

`/` Division (4).

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

Example

```
9/2 → 4
-12/3 → -4
9/2*6 → 24
```

`?:` Conditional operator (15).

The result of this operator is the first *expr* if *condition* evaluates to true and the second *expr* if *condition* evaluates to false.

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

Syntax

```
condition ? expr : expr
```

Example

```
5 ? 6 : 7 →6
0 ? 6 : 7 →7
```

`AND [&&]` Logical AND (12).

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

Example

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

`BINAND [&]` Bitwise AND (9).

Use `BINAND` to perform bitwise AND between the integer operands. Each bit in the 32-bit result is the logical AND of the corresponding bits in the operands.

Example

```
1010B BINAND 0011B → 0010B
1010B BINAND 0101B → 0000B
1010B BINAND 0000B → 0000B
```

`BINNOT [~]` Bitwise NOT (3).

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

Example

```
BINNOT 1010B → 11111111111111111111111111110101B
```

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 [>=] Greater than or equal (7).

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

Example

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

GT [>] Greater than (7).

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

Example

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

HIGH High byte (2).

`HIGH` takes a single operand to its right which is interpreted as an unsigned, 16-bit integer value. The result is the unsigned 8-bit integer value of the higher order byte of the operand.

Example

```
HIGH 0xABCD → 0xAB
```

HWRD High word (2).

`HWRD` takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the high word (bits 31 to 16) of the operand.

Example

```
HWRD 0x12345678 → 0x1234
```

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

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

Example

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

`LOW` Low byte (2).

`LOW` takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the unsigned, 8-bit integer value of the lower order byte of the operand.

Example

```
LOW 0xABCD → 0xCD
```

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

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

Example

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

`LWRD` Low word (2).

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

Example

```
LWRD 0x12345678 → 0x5678
```

MOD [%] Modulo (4).

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

X MOD Y is equivalent to X-Y*(X/Y) using integer division.

Example

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

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

<> evaluates to 0 (false) if its two operands are identical in value or to 1 (true) if its two operands are not identical in value.

Example

```
1 <> 2 → 1
2 <> 2 → 0
'A' <> 'B' → 1
```

NOT [!] Logical NOT (3).

Use NOT to negate a logical argument.

Example

```
NOT 0101B → 0
NOT 0000B → 1
```

OR [||] Logical OR (14).

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

Example

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

`SFB` Segment begin (2).

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

Syntax

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

Parameters

segment	The name of a relocatable segment, which must be defined before `SFB` is used.
offset	An optional offset from the start address. The parentheses are optional if *offset* is omitted.

Example

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

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

`SFE` Segment end (2).

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

Syntax

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

Parameters

segment	The name of a relocatable segment, which must be defined before `SFE` is used.
offset	An optional offset from the start address. The parentheses are optional if *offset* is omitted.

Example

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

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

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

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

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

Use `SHL` to shift the left operand, which is always treated as `unsigned`, to the left. The number of bits to shift is specified by the right operand, interpreted as an integer value between 0 and 32.

Example

```
00011100B SHL 3 → 11100000B
0000011111111111B SHL 5 → 1111111111100000B
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
1111111111111111B SHR 20 → 0
14 SHR 1 → 7
```

`SIZEOF` Segment size (2).

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

Syntax

```
SIZEOF (segment)
```

Parameters

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

Example

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

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

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

`UGT` Unsigned greater than (7).

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

Example

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

`ULT` Unsigned less than (7).

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

Example

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

`UPPER` Third byte (2).

`UPPER` takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the middle-high byte (bits 23 to 16) of the operand.

Example

```
UPPER 0x12345678 → 0x34
```

`XOR` Logical exclusive OR (13).

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

Example

```
0101B XOR 1010B → 0
0101B XOR 0000B → 1
```

Assembler directives

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

Summary of assembler directives

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

- *Module control directives*, page 53
- *Symbol control directives*, page 56
- *Segment control directives*, page 59
- *Value assignment directives*, page 64
- *Conditional assembly directives*, page 68
- *Macro processing directives*, page 70
- *Listing control directives*, page 77
- *C-style preprocessor directives*, page 81
- *Data definition or allocation directives*, page 85
- *Assembler control directives*, page 88
- *Function directives*, page 90
- *Call frame information directives*, page 92.

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

Directive	Description	Section
`_args`	Is set to number of arguments passed to macro.	Macro processing
`#define`	Assigns a value to a preprocessor symbol.	C-style preprocessor
`#elif`	Introduces a new condition in a `#if`…`#endif` block.	C-style preprocessor
`#else`	Assembles instructions if a condition is false.	C-style preprocessor
`#endif`	Ends a `#if`, `#ifdef`, or `#ifndef` block.	C-style preprocessor
`#error`	Generates an error.	C-style preprocessor
`#if`	Assembles instructions if a condition is true.	C-style preprocessor
`#ifdef`	Assembles instructions if a symbol is defined.	C-style preprocessor

Table 14: Assembler directives summary

Directive	Description	Section
`#ifndef`	Assembles instructions if a symbol is undefined.	C-style preprocessor
`#include`	Includes a file.	C-style preprocessor
`#line`	Changes the line numbers.	C-style preprocessor
`#pragma`	Controls extension features.	C-style preprocessor
`#undef`	Undefines a label.	C-style preprocessor
`/*comment*/`	C-style comment delimiter.	Assembler control
`//`	C++ style comment delimiter.	Assembler control
`=`	Assigns a permanent value local to a module.	Value assignment
`ALIGN`	Aligns the program location counter by inserting zero-filled bytes.	Segment control
`ALIGNRAM`	Aligns the program location counter.	Segment control
`ARGFRAME`	Declares the space used for the arguments to a function.	Function
`ASEG`	Begins an absolute segment.	Segment control
`ASEGN`	Begins a named absolute segment.	Segment control
`ASSIGN`	Assigns a temporary value.	Value assignment
`BLOCK`	Specifies the block number for an alias created by the `SYMBOL` directive.	Symbol control
`CASEOFF`	Disables case sensitivity.	Assembler control
`CASEON`	Enables case sensitivity.	Assembler control
`CFI`	Specifies call frame information.	Call frame information
`COMMON`	Begins a common segment.	Segment control
`DB`	Generates 8-bit constants, including strings.	Data definition or allocation
`DC8`	Generates 8-bit constants, including strings.	Data definition or allocation
`DC16`	Generates 16-bit constants.	Data definition or allocation
`DC24`	Generates 24-bit constants.	Data definition or allocation
`DC32`	Generates 32-bit constants.	Data definition or allocation

Table 14: Assembler directives summary (Continued)

Directive	Description	Section
`DC64`	Generates 64-bit constants.	Data definition or allocation
`DEFINE`	Defines a file-wide value.	Value assignment
`DF32`	Generates 32-bit floating-point constants.	Data definition or allocation
`DF64`	Generates 64-bit floating-point constants.	Data definition or allocation
`DQ15`	Generates 16-bit fractional constants.	Data definition or allocation
`DQ31`	Generates 32-bit fractional constants.	Data definition or allocation
`DS`	Allocates space for 8-bit integers.	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`	Terminates the assembly of the last module in a file.	Module control
`ENDIF`	Ends an `IF` block.	Conditional assembly
`ENDM`	Ends a macro definition.	Macro processing
`ENDMOD`	Terminates the assembly of the current module.	Module control
`ENDR`	Ends a repeat structure.	Macro processing
`EQU`	Assigns a permanent value local to a module.	Value assignment
`EVEN`	Aligns the program counter to an even address.	Segment control

Table 14: Assembler directives summary (Continued)

Directive	Description	Section
`EXITM`	Exits prematurely from a macro.	Macro processing
`EXTERN`	Imports an external symbol.	Symbol control
`FUNCALL`	Declares that the function *`caller`* calls the function *`callee`*.	Function
`FUNCTION`	Declares a label name to be a function.	Function
`IF`	Assembles instructions if a condition is true.	Conditional assembly
`IMPORT`	Imports an external symbol.	Symbol control
`LIBRARY`	Begins a library module.	Module control
`LIMIT`	Checks a value against limits.	Value assignment
`LOCAL`	Creates symbols local to a macro.	Macro processing
`LOCFRAME`	Declares the space used for the locals in a function.	Function
`LSTCND`		Listing control
`LSTCOD`	Controls multi-line code listing.	Listing control
`LSTEXP`	Controls the listing of macro generated lines.	Listing control
`LSTMAC`	Controls the listing of macro definitions.	Listing control
`LSTOUT`	Controls assembler-listing output.	Listing control
`LSTREP`	Controls the listing of lines generated by repeat directives.	Listing control
`LSTXRF`	Generates a cross-reference table.	Listing control
`MACRO`	Defines a macro.	Macro processing
`MODULE`	Begins a library module.	Module control
`NAME`	Begins a program module.	Module control
`ODD`	Aligns the program location counter to an odd address.	Segment control
`ORG`	Sets the program location counter.	Segment control
`PROGRAM`	Begins a program module.	Module control
`PUBLIC`	Exports symbols to other modules.	Symbol control
`PUBWEAK`	Exports symbols to other modules, multiple definitions allowed.	Symbol control
`RADIX`	Sets the default base.	Assembler control
`REPT`	Assembles instructions a specified number of times.	Macro processing
`REPTC`	Repeats and substitutes characters.	Macro processing
`REPTI`	Repeats and substitutes strings.	Macro processing

Table 14: Assembler directives summary (Continued)

Directive	Description	Section
`REQUIRE`	Forces a symbol to be referenced.	Symbol control
`RSEG`	Begins a relocatable segment.	Segment control
`RTMODEL`	Declares runtime model attributes.	Module control
`SET`	Assigns a temporary value.	Value assignment
`SYMBOL`	Creates an alias that can be used for referring to a C/C++ symbol.	Symbol control
`VAR`	Assigns a temporary value.	Value assignment

Table 14: Assembler directives summary (Continued)

Module control directives

Module control directives are used for marking the beginning and end of source program modules, and for assigning names and types to them. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

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

Table 15: Module control directives

SYNTAX

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

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

PARAMETERS

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

DESCRIPTIONS

Beginning a program module

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

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

Beginning a library module

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

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

Terminating a module

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

Terminating the source file

Use END to indicate the end of the source file. Any lines after the END directive are ignored. The END directive also terminates the last module in the file, if this is not done explicitly with an ENDMOD directive.

Assembling multi-module files

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

The following rules apply when assembling multi-module files:

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

Note: END must always be placed after the *last* module, and there must not be any source lines (except for comments and listing control directives) between an ENDMOD and the next module (beginning with MODULE, LIBRARY, NAME, or PROGRAM).

If any of the directives NAME, MODULE, LIBRARY, or PROGRAM is missing, the module will be assigned the name of the source file and the attribute program.

Declaring runtime model attributes

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

A module can have several runtime model definitions.

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

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

Examples

The following example defines three modules where:

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

```
MODULE MOD_1
  RTMODEL   "foo", "1"
  RTMODEL   "bar", "XXX"
  ...
ENDMOD
MODULE MOD_2
  RTMODEL   "foo", "2"
  RTMODEL   "bar", "*"
  ...
ENDMOD
MODULE MOD_3
  RTMODEL   "bar", "XXX"
  ...
END
```

Symbol control directives

These directives control how symbols are shared between modules.

Directive	Description
BLOCK	Specifies the block number for an alias created by the SYMBOL directive.
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.
SYMBOL	Creates an alias for a C/C++ symbol.

Table 16: Symbol control directives

SYNTAX

```
label BLOCK old_label, block_number
EXTERN symbol [,symbol] …
IMPORT symbol [,symbol] …
PUBLIC symbol [,symbol] …
PUBWEAK symbol [,symbol] …
REQUIRE symbol
label SYMBOL "C/C++_symbol" [,old_label]
```

PARAMETERS

block_number	Block number of the alias created by the `SYMBOL` directive.
C/C++_symbol	C/C++ symbol to create an alias for.
label	Label to be used as an alias for a C/C++ symbol.
old_label	Alias created earlier by a `SYMBOL` directive.
symbol	Symbol to be imported or exported.

DESCRIPTIONS

Exporting symbols to other modules

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

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

There are no restrictions on the number of `PUBLIC`-defined symbols in a module.

Exporting symbols with multiple definitions to other modules

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

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

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

Importing symbols

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

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

Referring to scoped C/C++ symbols

Use the `SYMBOL` directive to create an alias for a C/C++ symbol. The alias can be used for referring to the C/C++ symbol. The symbol and the alias must be located within the same scope.

Use the `BLOCK` directive to provide the block scope for the alias.

Typically, the `SYMBOL` and the `BLOCK` directives are for compiler internal use only, for example when referring to objects inside classes or namespaces. For detailed information about how to use these directives, declare and define your C/C++ symbol, compile, and view the assembler listfile output.

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 will be resolved at link time.

```
        NAME        error
        EXTERN      print
        PUBLIC      err
        RSEG        CODE

err     JSR         (print).L
        DB          "** Error **"
        EVEN
        RTS
        END
```

Segment control directives

The segment directives control how code and data are located. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

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

Table 17: Segment control directives

SYNTAX

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

PARAMETERS

address	Address where this segment part will be placed.
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, except for code segments where the default is 1.
expr	Address to set the location counter to.
flag	NOROOT, ROOT NOROOT means that the segment part is discarded by the linker if no symbols in this segment part are referred to. Normally, all segment parts except startup code and interrupt vectors should set this flag. The default mode is ROOT which indicates that the segment part must not be discarded. REORDER, NOREORDER REORDER allows the linker to reorder segment parts. For a given segment, all segment parts must specify the same state for this flag. The default mode is NOREORDER which indicates that the segment parts must remain in order. SORT, NOSORT SORT means that the linker will sort the segment parts in decreasing alignment order. For a given segment, all segment parts must specify the same state for this flag. The default mode is NOSORT which indicates that the segment parts will not be sorted.
segment	The name of the segment.
start	A start address that has the same effect as using an ORG directive at the beginning of the absolute segment.
type	The memory type, typically CODE or DATA. In addition, any of the types supported by the IAR XLINK Linker.
value	Byte value used for padding, default is zero.

DESCRIPTIONS

Beginning an absolute segment

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

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

Beginning a named absolute segment

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

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

Beginning a relocatable segment

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

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

Beginning a common segment

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

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

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

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

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

Setting the program location counter (PLC)

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

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

Aligning a segment

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

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

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

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

EXAMPLES

Beginning an absolute segment

The following example assembles the jump to the function main in address 0. On RESET, the chip sets PC to address 0.

```
       MODULE     reset

       EXTERN     main

       ASEG
       ORG        0           ; RESET vector address

reset: GOTO       main        ; Instruction that
                              ; executes on startup

       end
```

Beginning a relocatable segment

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

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

```
       EXTERN           subrtn, divrtn

       RSEG             table
```

```
functable:
       DC32             subrtn, divrtn

       RSEG             code

main:
       MOWE.L     #0x12, D0
       ADD.L      #0x20, D0
       RTS

       END
```

Beginning a common segment

The following example defines two common segments containing variables:

```
        NAME    common1
        COMMON  data
count   DC32    1
        ENDMOD
        NAME    common2
        COMMON  data
up      DS8     1
        DS8     2
down    DS8     1
        END
```

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

Aligning a segment

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

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

Value assignment directives

These directives are used for assigning values to symbols.

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

Table 18: Value assignment directives

SYNTAX

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

PARAMETERS

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

DESCRIPTIONS

Defining a temporary value

Use `ASSIGN`, `SET`, or `VAR` to define a symbol that may 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 and all modules following that module in the same source file. If a DEFINE directive is placed outside of a module, the symbol will be known to all modules following the directive in the same source file.

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

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

Checking symbol values

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

The check will occur as soon as the expression is resolved, which will be during linking if the expression contains external references. The *min* and *max* expressions cannot involve references to forward or external labels, that is they must be resolved when encountered.

EXAMPLES

Redefining a symbol

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

```
        NAME  table

; Generate table of power of 3

cons    SET 1

cr_tabl MACRO times
        DC32  cons
cons    SET   cons*3
        IF times > 1
```

```
          cr_tabl  times-1
        ENDIF
        ENDM

        RSEG    CODE
table   cr_tabl 4

        END table
```

It generates the following code:

```
     1                                   NAME  table
     2
     3                           ; Generate table of power of 3
     4
     5    000001                 cons    SET 1
     6
     7                           cr_tabl MACRO times
     8                                   DC32  cons
     9                           cons    SET   cons*3
    10                                   IF times > 1
    11                                     cr_tabl  times-1
    12                                   ENDIF
    13                                   ENDM
    14
    15    000000                         RSEG    CODE
    16    000000                 table   cr_tabl 4
    16.1  000000 00000001                DC32  cons
    16.2  000003                 cons    SET   cons*3
    16.3  000004                         IF 4 > 1
    16.4  000004                           cr_tabl  4-1
    16.5  000004 00000003                DC32  cons
    16.6  000009                 cons    SET   cons*3
    16.7  000008                         IF 4-1 > 1
    16.8  000008                           cr_tabl  4-1-1
    16.9  000008 00000009                DC32  cons
    16.10 00001B                 cons    SET   cons*3
    16.11 00000C                         IF 4-1-1 > 1
    16.12 00000C                           cr_tabl  4-1-1-1
    16.13 00000C 0000001B                DC32  cons
    16.14 000051                 cons    SET   cons*3
    16.15 000010                         IF 4-1-1-1 > 1
    16.16 000010                         ENDIF
    16.17 000010                         ENDIF
    16.18 000010                         ENDIF
    16.19 000010                         ENDIF
    17
    18    000010                         END table
```

Using local and global symbols

In the following example the symbol `value` defined in module `add1` is local to that module; a distinct symbol of the same name is defined in module `add2`. The `DEFINE` directive is used for declaring `R0` for use anywhere in the file:

```
        NAME    add1
        PUBLIC  add12
R0      DEFINE  0x20
value   EQU 12
        RSEG    CODE
add12:
        MOVE.L  #value, D0
        ADD.L   #R0, D0
        RTS
        ENDMOD

        NAME    add2
        PUBLIC  add20
value   EQU 20
        RSEG    CODE
add20:
        MOVE.L  #R0, D0
        ADD.L   #value, D0
        RTS

        END
```

The symbol `R0` defined in module `add1` is also available to module `add2`.

Using the LIMIT directive

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

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

Conditional assembly directives

These directives provide logical control over the selective assembly of source code. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

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

Table 19: Conditional assembly directives

SYNTAX

`ELSE`

`ELSEIF` *`condition`*

`ENDIF`

`IF` *`condition`*

PARAMETERS

`condition` One of the following:

An absolute expression	The expression must not contain forward or external references, and any non-zero value is considered as true.
`string1==string2`	The condition is true if *`string1`* and *`string2`* have the same length and contents.
`string1!=string2`	The condition is true if *`string1`* and *`string2`* have different length or contents.

DESCRIPTIONS

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

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

All assembler directives (except for END) as well as the inclusion of files may be disabled by the conditional directives. Each IF directive must be terminated by an ENDIF directive. The ELSE directive is optional, and if used, it must be inside an IF...ENDIF block. IF...ENDIF and IF...ELSE...ENDIF blocks may be nested to any level.

EXAMPLES

The following macro adds a constant to a byte variable in memory:

```
xadd  MACRO   a ,b  ; a is a register, D0-D7,
                    ; b is a literal
      IF b == 1
        ADDQ.L #1, a
      ELSE
        ADD.L  #b, a
      ENDIF
      ENDM
```

If the argument to the macro is 1, it generates an INC instruction; otherwise it generates an ADD instruction.

It could be tested with the following program:

```
main:
      xadd     D0, 0x12
      xadd     D1, 1
      RTS

      END
```

Macro processing directives

These directives allow user macros to be defined. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

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

Table 20: 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.

`formal`	An argument into which each character of *`actual`* (`REPTC`) or each *`actual`* (`REPTI`) is substituted.
`name`	The name of the macro.
`symbol`	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 `ERROR` as follows:

```
errmac  MACRO   text
        JSR     abort
        DB      text,0
        EVEN
        ENDM
```

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 will expand this to:

```
        JSR     abort
        DB      '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  MACRO
        CALL    abort
        DB      \1,0
        EVEN
        ENDM
```

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

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

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

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

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

Passing special characters

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

For example:

```
macld    MACRO  op
         MOVE.L  op
         ENDM
```

The macro can be called using the macro quote characters:

```
         macld  <#0x800, D0>
         END
```

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. The following example shows how `_args` can be used:

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

RSEG  CODE

      FILL  3, 4
      FILL  3
      END
```

It generates the following code:

```
 1                           FILL  MACRO
 2                                 IF _args == 2
 3                                   REPT \1
 4                                     DC8 \2
 5                                   ENDR
 6                                 ELSE
 7                                   DC8 \1
 8                                 ENDIF
 9                                 ENDM
10
11    000000                       RSEG  CODE
12    000000                       FILL  3, 4
12.1  000000                       IF _args == 2
12.2  000000                         REPT 3
12.3  000000 04                        DC8 4
12.4  000001 04                        DC8 4
12.5  000002 04                        DC8 4
12.6  000003                         ENDR
12.7  000003                       ELSE
12.8  000003                       ENDIF
13    000003                       FILL  3
13.1  000003                       IF _args == 2
13.2  000003                       ELSE
13.3  000003 03                      DC8 3
13.4  000004                       ENDIF
14    000004                       END
```

How macros are processed

There are three distinct phases in the macro process:

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

   The macro expander has no knowledge of assembler symbols since it only deals with text substitutions at source level. Before a line from the called macro definition is handed over to the assembler, the expander scans the line for all occurrences of symbolic macro arguments, and replaces them with their expansion arguments.
3. The expanded line is then processed as any other assembler source line. The input stream to the assembler will continue to be the output from the macro processor, until all lines of the current macro definition have been read.

Repeating statements

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

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

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

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

EXAMPLES

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

Coding inline for efficiency

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

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

```
        NAME    play

        RSEG    CSTACK

        ASEGN   CODE, 0x0000
        DC32    SFE(CSTACK)
        DC32    main

P0      DEFINE  0xFF8010

        RSEG    DATA
buffer  DS8     25
watch   DC8     0xFF

        RSEG    CODE
play    MOVEA.L #buffer, A0
        MOVE.B  (buffer, A0), D0
loop    ADDQ.L  #1, A0
        MOVE.B  D0, (P0).L
        MOVE.B  (buffer, A0), D0
        CMPI.B  #watch, D0
        BNE.B   loop
        RTS

main    JSR     (play).L
        RTS
        END
```

The main program calls this routine as follows:

```
doplay  CALL    play
```

For efficiency we can recode this using a macro:

```
        NAME    play

        RSEG    STACK

        ASEGN   CODE, 0x0000
        DC32    SFE(STACK)
        DC32    main

play    MACRO
        LOCAL   loop
        MOVEA.L #buffer, A0
        MOVE.B  (buffer, A0), D0
loop    ADDQ.L  #1, A0
```

```
        MOVE.B  D0, (P0).L
        MOVE.B  (buffer, A0), D0
        CMPI.B  #watch, D0
        BNE.B   loop
        ENDM

P0      DEFINE  0xFF8010

        RSEG    DATA
buffer  DS8     25
watch   DC8     0xFF

        RSEG    CODE
main    play
        play
        RTS
        END
```

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

Using REPTC and REPTI

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

```
        NAME      reptc
        EXTERN    plotc
P0      DEFINE    0xFF8010
banner  REPTC     chr, "Welcome"
          MOVE.B  #'chr', D0
          MOVE.B  D0, (P0).L
          JSR     (plotc).L
        ENDR
        END
```

This produces the following code:

```
1                                   NAME      reptc
2    000000                         EXTERN    plotc
3    FF8010                P0       DEFINE    0xFF8010
4    000000                banner   REPTC     chr, "Welcome"
4.1  000000 103C0057                  MOVE.B  #'W', D0
4.2  000004 13C000FF8010              MOVE.B  D0, (P0).L
4.3  00000A 4EB9........              JSR     (plotc).L
4.4  000010 103C0065                  MOVE.B  #'e', D0
4.5  000014 13C000FF8010              MOVE.B  D0, (P0).L
4.6  00001A 4EB9........              JSR     (plotc).L
```

```
4.7  000020 103C006C               MOVE.B  #'l', D0
4.8  000024 13C000FF8010           MOVE.B  D0, (P0).L
4.9  00002A 4EB9........           JSR     (plotc).L
4.10 000030 103C0063               MOVE.B  #'c', D0
4.11 000034 13C000FF8010           MOVE.B  D0, (P0).L
4.12 00003A 4EB9........           JSR     (plotc).L
4.13 000040 103C006F               MOVE.B  #'o', D0
4.14 000044 13C000FF8010           MOVE.B  D0, (P0).L
4.15 00004A 4EB9........           JSR     (plotc).L
4.16 000050 103C006D               MOVE.B  #'m', D0
4.17 000054 13C000FF8010           MOVE.B  D0, (P0).L
4.18 00005A 4EB9........           JSR     (plotc).L
4.19 000060 103C0065               MOVE.B  #'e', D0
4.20 000064 13C000FF8010           MOVE.B  D0, (P0).L
4.21 00006A 4EB9........           JSR     (plotc).L
4.22 000070                       ENDR
9    000070                       ENDMOD
```

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

```
        NAME      repti
        EXTERN    base, count, init
banner  REPTI     adds, base, count, init
          CLR.B   (adds).L
        ENDR
        END
```

This produces the following code:

```
12                                   NAME     repti
13    000000                         EXTERN   base, count, init
14    000000                  banner REPTI    adds, base, count, init
14.1  000000 4239........              CLR.B  (base).L
14.2  000006 4239........              CLR.B  (count).L
14.3  00000C 4239........              CLR.B  (init).L
14.4  000012                         ENDR
17    000012                         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.

Table 21: Listing control directives

Directive	Description
`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 21: Listing control directives (Continued)

SYNTAX

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

DESCRIPTIONS

Turning the listing on or off

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

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

Listing conditional code and strings

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

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

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

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

Using the `LSTCND` and `LSTCOD` directives does *not* affect code generation.

Controlling the listing of macros

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

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

Controlling the listing of generated lines

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

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

Generating a cross-reference table

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

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

EXAMPLES

Turning the listing on or off

To disable the listing of a debugged section of program:

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

Listing conditional code and strings

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

```
        NAME    lstcndtst
        EXTERN  print

        RSEG    prom

debug   SET 0

begin   IF debug
          JSR   (print).L
        ENDIF

        LSTCND+
begin2  IF debug
          JSR   (print).L
        ENDIF
        END
```

This will generate the following listing:

```
     1                                    NAME    lstcndtst
     2    000000                          EXTERN  print
     3
     4    000000                          RSEG    prom
     5
     6    000000              debug       SET 0
     7
     8    000000              begin       IF debug
     9                                      JSR   (print).L
    10    000000                          ENDIF
    11
    12                                    LSTCND+
    13    000000              begin2      IF debug
    15    000000                          ENDIF
    16    000000                          END
```

Controlling the listing of macros

The following example shows the effect of `LSTMAC` and `LSTEXP`:

```
         NAME     incdec

dec2     MACRO    arg
         MOVE.L   (arg).L,   D0
         SUBQ.L   #2, D0
         ENDM

         LSTMAC-
inc2     MACRO    arg
         MOVE.L   (arg).L,   D0
         ADDQ.L   #2, D0
         ENDM

         EXTERN   memlock
begin    dec2     memlock
         LSTEXP-
         inc2     memlock
         RTS
         END
```

This will produce the following output:

```
     1                                    NAME    incdec
     2
     3                        dec2        MACRO   arg
     4                                    MOVE.L  (arg).L,  D0
     5                                    SUBQ.L  #2, D0
```

```
 6                                  ENDM
 7
 8                                  LSTMAC-
13
14    000000                        EXTERN  memlock
15    000000               begin    dec2    memlock
15.1  000000 2039........           MOVE.L  (memlock).L,  D0
15.2  000006 5580                   SUBQ.L  #2, D0
16                                  LSTEXP-
17    000008                        inc2    memlock
18    000010 4E75                   RTS
19    000012                        END
```

C-style preprocessor directives

The following C-language preprocessor directives are available:

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

Table 22: C-style preprocessor directives

SYNTAX

`#define` *`symbol text`*

`#elif` *`condition`*

`#else`

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

PARAMETERS

condition	An absolute expression	The expression must not contain any assembler labels or symbols, and any non-zero value is considered as true.
filename	Name of file to be included.	
message	Text to be displayed.	
symbol	Preprocessor symbol to be defined, undefined, or tested.	
text	Value to be assigned.	

DESCRIPTIONS

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

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 will not be 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 a #endif or #else directive is found.

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

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

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

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

Including source files

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

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

- If the name of the #include file is an absolute path, that file is opened.
- When the assembler encounters the name of an #include file in angle brackets such as:

  ```
  #include <ioM52223.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.

  1 The directories specified using the ACF_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 ColdFire IAR Assembler, 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.

The following expression will evaluate to 3 because the comment character will be preserved by `#define`:

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

The following 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 */

  DC32 five,11,12
  ; Expands to "DC32 5        ; this comment is not OK"

  DC32 six + seven,11,12      ; OK
  ; Expanded to "DC32 6+7,11,12"
```

EXAMPLES

Using conditional preprocessor directives

The following example defines the labels `tweak` and `adjust`. If `adjust` is defined, then register `D0` is decremented by an amount that depends on `adjust`, in this case 30.

```
#define tweak 1
#define adjust 3

#ifdef  tweak
#if     adjust==1
        SUB.L #4, D0
```

```
#elif   adjust==2
        SUB.L #20, D0
#elif   adjust==3
        SUB.L #30, D0
#endif
#endif                                      /* ifdef tweak */
```

Including a source file

The following example uses `#include` to include a file defining macros into the source file. For example, the following macros could be defined in `macros.s68`:

```
; exchange a and b using c as temporary
xch      MACRO   a, b, c
         MOVE.L  a,c
         MOVE.L  b,a
         MOVE.L  c,b
         ENDM
```

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

```
         NAME    include

; standard macro definitions

#include "macros.s68"

; program

main:
         xch     D0,D1,D2
         RTS
         END     main
```

Data definition or allocation directives

These directives define values or reserve memory. The column *Alias* in the following table shows the Freescale directive that corresponds to the IAR Systems directive. See *Expression restrictions*, page 12, 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.

Table 23: Data definition or allocation directives

Directive	Alias	Description
`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`	`DS`	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 23: Data definition or allocation directives (Continued)

SYNTAX

```
DB expr [,expr] …
DC8 expr [,expr] ...
DC16 expr [,expr] ...
DC24 expr [,expr] ...
DC32 expr [,expr] ...
DC64 expr [,expr] ...
 expr [,expr] …
 expr [,expr] …
 expr [,expr] …
 value [,value] …
 value [,value] …DF value [,value] …
DF32 value [,value] …
DF64 value [,value] …
DQ15 value [,value] …
DQ31 value [,value] …
DS count
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 will be zero filled to a multiple of the data size implied by the directive. Double-quoted strings will be zero-terminated.*
value	A valid absolute expression or floating-point constant.

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

DESCRIPTIONS

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

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

EXAMPLES

Generating a lookup table

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

```
        NAME    table

        PUBLIC  fsum

        RSEG    CODE
table:  DC8     12
        DC8     15
        DC8     17
        DC8     16
        DC8     14
        DC8     11
        DC8     9

        RSEG    CODE

sum     DEFINE  0x20

COUNT   SET     0
```

```
fsum:
        CLR.L     D0
        CLR.L     D7
        REPT 7
          IF COUNT == 7
            EXITM
          ENDIF
          MOVE.B  (table+COUNT).L, D0
          ADD.L   D0, D7
COUNT     SET     COUNT+1
        ENDR
        MOVE.B    D7, D0          ; Get sum into D0
        RTS
        END
```

Defining strings

To define a string:

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

To define a string which includes a trailing zero:

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

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

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

Reserving space

To reserve space for `10` bytes:

```
table   DS8   10
```

Assembler control directives

These directives provide control over the operation of the assembler. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
`/*comment*/`	C-style comment delimiter.	
`//`	C++ style comment delimiter.	
`CASEOFF`	Disables case sensitivity.	

Table 24: Assembler control directives

Directive	Description	Expression restrictions
CASEON	Enables case sensitivity.	
RADIX	Sets the default base on all numeric values.	No forward references No external references Absolute Fixed

Table 24: Assembler control directives (Continued)

SYNTAX

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

PARAMETERS

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

DESCRIPTIONS

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

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

Use `RADIX` to set the default base for constants. The default base is 10.

Controlling case sensitivity

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

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

EXAMPLES

Defining comments

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

```
/*
Program to read serial input.
```

```
Version 1: 19.2.07
Author: mjp
*/
```

See also, *Comments in C-style preprocessor directives*, page 84.

Changing the base

To set the default base to 16:

```
        RADIX  16'D
        MOVE.L #0x12, D0
```

The immediate argument will then be interpreted as the hexadecimal constant 12, that is decimal 18.

To reset the base from 16 to 10 again, the argument must be written in hexadecimal format, for example:

```
        RADIX  0x0A
```

Controlling case sensitivity

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

```
label NOP                 ; Stored as "LABEL"
      JMP (LABEL).L

                          ; The following will generate a
                          ; duplicate label error:

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

      END
```

Function directives

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

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

SYNTAX

```
ARGFRAME segment, size, type
FUNCALL caller, callee
FUNCTION label,value
LOCFRAME segment, size, type
```

PARAMETERS

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

DESCRIPTIONS

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

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

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

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

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

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

Call frame information directives

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

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

Table 25: Call frame information directives

SYNTAX

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

Names block directives

CFI NAMES *name*

CFI ENDNAMES *name*

CFI RESOURCE *resource* : *bits* [, *resource* : *bits*] …

CFI VIRTUALRESOURCE *resource* : *bits* [, *resource* : *bits*] …

CFI RESOURCEPARTS *resource part, part* [, *part*] ...

CFI STACKFRAME *cfa resource type* [, *cfa resource type*] …

CFI STATICOVERLAYFRAME *cfa segment* [, *cfa segment*] …

CFI BASEADDRESS *cfa type* [, *cfa type*] …

Extended names block directives

CFI NAMES *name* EXTENDS *namesblock*

CFI ENDNAMES *name*

CFI FRAMECELL *cell cfa* (*offset*): *size* [, *cell cfa* (*offset*) : *size*] …

Common block directives

CFI COMMON *name* USING *namesblock*

CFI ENDCOMMON *name*

CFI CODEALIGN *codealignfactor*

CFI DATAALIGN *dataalignfactor*

CFI RETURNADDRESS *resource type*

CFI *cfa* { NOTUSED | USED }

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

CFI *cfa cfiexpr*

CFI *resource* { UNDEFINED | SAMEVALUE | CONCAT }

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

CFI *resource cfiexpr*

Extended common block directives

CFI COMMON *name* EXTENDS *commonblock* USING *namesblock*

CFI ENDCOMMON *name*

Data block directives

```
CFI BLOCK name USING commonblock
CFI ENDBLOCK name
CFI { NOFUNCTION | FUNCTION label }
CFI { INVALID | VALID }
CFI { REMEMBERSTATE | RESTORESTATE }
CFI PICKER
CFI CONDITIONAL label [, label] …
CFI cfa { resource | resource + constant | resource - constant }
CFI cfa cfiexpr
CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
CFI resource { resource | FRAME(cfa, offset) }
CFI resource cfiexpr
```

PARAMETERS

bits	The size of the resource in bits.
cell	The name of a frame cell.
cfa	The name of a CFA (canonical frame address).
cfiexpr	A CFI expression (see *CFI expressions*, page 101).
codealignfactor	The smallest factor of all instruction sizes. Each CFI directive for a data block must be placed according to this alignment. 1 is the default and can always be used, but a larger value will shrink the produced backtrace information in size. The possible range is 1–256.
commonblock	The name of a previously defined common block.
constant	A constant value or an assembler expression that can be evaluated to a constant value.
dataalignfactor	The smallest factor of all frame sizes. If the stack grows towards higher addresses, the factor is negative; if it grows towards lower addresses, the factor is positive. 1 is the default, but a larger value will shrink the produced backtrace information in size. The possible ranges are -256 to -1 and 1 to 256.
label	A function label.
name	The name of the block.

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

DESCRIPTIONS

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

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

Backtrace rows and columns

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

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

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

Defining a names block

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

Start and end a names block with the directives:

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

where `name` is the name of the block.

Only one names block can be open at a time.

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

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

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

  The parameters are the name of the resource and the size of the resource in bits. A virtual resource is a logical concept, in contrast to a “physical” resource such as a processor register. Virtual resources are usually used for the return address.

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

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

  ```
  CFI RESOURCEPARTS resource part, part, …
  ```

  The parts are separated with commas. The resource and its parts must have been previously declared as resources, as described above.

- To declare a stack frame CFA, use the directive:

  ```
  CFI STACKFRAME cfa resource type
  ```

  The parameters are the name of the stack frame CFA, the name of the associated resource (the stack pointer), and the segment type (to get the address space). More than one stack frame CFA can be declared by separating them with commas.

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

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

  ```
  CFI STATICOVERLAYFRAME cfa segment
  ```

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

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

  ```
  CFI BASEADDRESS cfa type
  ```

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

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

Extending a names block

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

Extend an existing names block with the directive:

```
CFI NAMES name EXTENDS namesblock
```

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

```
CFI ENDNAMES name
```

Defining a common block

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

Start a common block with the directive:

```
CFI COMMON name USING namesblock
```

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

Declare the return address column with the directive:

```
CFI RETURNADDRESS resource type
```

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

End a common block with the directive:

```
CFI ENDCOMMON name
```

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

Inside a common block you can declare the initial value of a CFA or a resource by using the directives listed last in *Common block directives*, page 93. For more information on these directives, see *Simple rules*, page 99, and *CFI expressions*, page 101.

Extending a common block

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

Extend an existing common block with the directive:

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

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

```
CFI ENDCOMMON name
```

Defining a data block

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

Start a data block with the directive:

```
CFI BLOCK name USING commonblock
```

where *name* is the name of the new block and *commonblock* is the name of a previously defined common block.

If the piece of code is part of a defined function, specify the name of the function with the directive:

```
CFI FUNCTION label
```

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

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

```
CFI NOFUNCTION
```

End a data block with the directive:

```
CFI ENDBLOCK name
```

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

Inside a data block you may manipulate the values of the columns by using the directives listed last in *Data block directives*, page 94. For more information on these directives, see *Simple rules*, page 99, and *CFI expressions*, page 101.

SIMPLE RULES

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

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

These simple rules can be used both in common blocks to describe the initial information for resources and CFAs, and inside data blocks to describe changes to the information for resources or CFAs.

In those rare cases where the descriptive power of the simple rules are not enough, a full CFI expression can be used to describe the information (see *CFI expressions*, page 101). 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 93.

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

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

```
CFI CFA_SP NOTUSED
```

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

```
CFI CFA_SP SP + 4
```

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

CFI EXPRESSIONS

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

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

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

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

Unary operators

Overall syntax: *`OPERATOR`*`(`*`operand`*`)`

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

Table 26: Unary operators in CFI expressions

Binary operators

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

Operator	Operands	Description
`ADD`	*`cfiexpr`*,*`cfiexpr`*	Addition
`AND`	*`cfiexpr`*,*`cfiexpr`*	Bitwise AND
`DIV`	*`cfiexpr`*,*`cfiexpr`*	Division
`EQ`	*`cfiexpr`*,*`cfiexpr`*	Equal

Table 27: Binary operators in CFI expressions

Operator	Operands	Description
`GE`	*`cfiexpr`*,*`cfiexpr`*	Greater than or equal
`GT`	*`cfiexpr`*,*`cfiexpr`*	Greater than
`LE`	*`cfiexpr`*,*`cfiexpr`*	Less than or equal
`LSHIFT`	*`cfiexpr`*,*`cfiexpr`*	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`	*`cfiexpr`*,*`cfiexpr`*	Less than
`MOD`	*`cfiexpr`*,*`cfiexpr`*	Modulo
`MUL`	*`cfiexpr`*,*`cfiexpr`*	Multiplication
`NE`	*`cfiexpr`*,*`cfiexpr`*	Not equal
`OR`	*`cfiexpr`*,*`cfiexpr`*	Bitwise OR
`RSHIFTA`	*`cfiexpr`*,*`cfiexpr`*	Arithmetic shift right of the left operand. The number of bits to shift is specified by the right operand. In contrast with RSHIFTL the sign bit will be preserved when shifting.
`RSHIFTL`	*`cfiexpr`*,*`cfiexpr`*	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`	*`cfiexpr`*,*`cfiexpr`*	Subtraction
`XOR`	*`cfiexpr`*,*`cfiexpr`*	Bitwise XOR

Table 27: Binary operators in CFI expressions (Continued)

Ternary operators

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

Operator	Operands	Description
`FRAME`	*`cfa,size,offset`*	Gets the value from a stack frame. The operands are: *`cfa`* An identifier denoting a previously declared CFA. *`size`* A constant expression denoting a size in bytes. *`offset`* A constant expression denoting an offset in bytes. Gets the value at address *`cfa+offset`* of size *`size`*.

Table 28: Ternary operators in CFI expressions

Operator	Operands	Description
`IF`	*`cond,true,false`*	Conditional operator. The operands are: *`cond`* A CFA expression denoting a condition. *`true`* Any CFA expression. *`false`* Any CFA expression. If the conditional expression is non-zero, the result is the value of the *`true`* expression; otherwise the result is the value of the *`false`* expression.
`LOAD`	*`size,type,addr`*	Gets the value from memory. The operands are: *`size`* A constant expression denoting a size in bytes. *`type`* A memory type. *`addr`* A CFA expression denoting a memory address. Gets the value at address *`addr`* in segment type *`type`* of size *`size`*.

Table 28: Ternary operators in CFI expressions (Continued)

EXAMPLE

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

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

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

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

Table 29: Code sample with backtrace rows and columns

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

The SP column is empty since the CFA is defined in terms of the stack pointer. The RET column is the return address column—that is, the location of the return address. The R0 column has a '—' in the first line to indicate that the value of `R0` is undefined and does not need to be restored on exit from the function. The R1 column has `SAME` in the initial row to indicate that the value of the `R1` register will be restored to the same value it already has.

Defining the names block

The names block for the small example above would be:

```
CFI NAMES trivialNames
CFI RESOURCE SP:16, R0:16, R1:16
CFI STACKFRAME CFA SP DATA

;; The virtual resource for the return address column
CFI VIRTUALRESOURCE RET:16
CFI ENDNAMES trivialNames
```

Defining the common block

The common block for the simple example above would be:

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

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

Defining the data block

Continuing the simple example, the data block would be:

```
    RSEG   CODE:CODE
    CFI    BLOCK func1block USING trivialCommon
    CFI    FUNCTION func1
```

```
func1:
     PUSH   R1
     CFI    CFA SP + 4
     CFI    R1 FRAME(CFA,-4)
     MOV    R1,#4
     CALL   func2
     POP    R0
     CFI    R1 R0
     CFI    CFA SP + 2
     MOV    R1,R0
     CFI    R1 SAMEVALUE
     RET
     CFI ENDBLOCK func1block
```

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

Pragma directives

This chapter describes the pragma directives of the ColdFire IAR Assembler.

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

Summary of pragma directives

The following table shows the pragma directives of the assembler:

#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 30: Pragma directives summary

Descriptions of pragma directives

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

`#pragma` *`pragmaname`*`=`*`pragmavalue`*

or

`#pragma` *`pragmaname`* `=` *`pragmavalue`*

#pragma diag_default

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

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

```
#pragma diag_default=Pe117
```

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

`#pragma diag_error`

`#pragma diag_error=`*`tag`*`,`*`tag`*`,...`

Changes the severity level to `error` for the specified diagnostics. For example:

```
#pragma diag_error=Pe117
```

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

`#pragma diag_remark`

`#pragma diag_remark=`*`tag`*`,`*`tag`*`,...`

Changes the severity level to `remark` for the specified diagnostics. For example:

```
#pragma diag_remark=Pe177
```

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

`#pragma diag_suppress`

`#pragma diag_suppress=`*`tag`*`,`*`tag`*`,...`

Suppresses the diagnostic messages with the specified tags. For example:

```
#pragma diag_suppress=Pe117,Pe177
```

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

`#pragma diag_warning`

`#pragma diag_warning=`*`tag`*`,`*`tag`*`,...`

Changes the severity level to `warning` for the specified diagnostics. For example:

```
#pragma diag_warning=Pe826
```

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

`#pragma message`

`#pragma message(`*`string`*`)`

Makes the assembler print a message on `stdout` when the file is assembled. For example:

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

Diagnostics

This chapter describes the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.

Message format

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

```
filename,linenumber level[tag]: message
```

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

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

Severity levels

The diagnostics are divided into different levels of severity:

Remark

A diagnostic message that is produced when the assembler finds a source code construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued but can be enabled, see *--remarks*, page 30.

Warning

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

Error

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

Fatal error

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

SETTING THE SEVERITY LEVEL

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

See *Summary of assembler options*, page 17, for a description of the assembler options that are available for setting severity levels.

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

INTERNAL ERROR

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

```
Internal error: message
```

where `message` is an explanatory message. If internal errors occur, they should be reported to your software distributor or IAR 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
absolute segments 60
__ACF__ (predefined symbol) 9
ACF_INC (environment variable) 4
ADD (CFI operator) 101
address field, in assembler list file 13
ALIGN (assembler directive) 59
alignment, of segments 62
ALIGNRAM (assembler directive) 59
AND (assembler operator) 38
AND (CFI operator) 101
architecture, ColdFire ix
ARGFRAME (assembler directive) 91
_args (assembler directive) 70
_args (predefined macro symbol) 72
ASCII character constants. 7
ASEG (assembler directive) 59
ASEGN (assembler directive) 59
asm (filename extension) 3
ASMCF (environment variable) 4
assembler BLOCK (assembler directive) 56
assembler control directives 88
assembler diagnostics 109
assembler directives
  assembler control 88
  call frame information (CFI) 92
  conditional assembly 68
    See also C-style preprocessor directives
  C-style preprocessor 81
  data definition or allocation 85
  function 90
  list file control 77
  macro processing 70
  module control 53
  segment control 59
  summary 49
  symbol control 56
  value assignment 64
  #pragma 107
assembler environment variables 4
assembler expressions. 6
assembler instructions. 5
assembler invocation syntax 3
assembler labels 8
  format of 5
assembler list files
  address field. 13
  comments. 89
  conditional code and strings. 78
  cross-references, generating 25, 79
  data field 13
  disabling 78
  enabling 78
  filename, specifying. 25
  generated lines, controlling 79
  macro-generated lines, controlling. 78
  symbol and cross-reference table. 13
assembler macros
  arguments, passing to. 72
  defining 71
  generated lines, controlling in list file 78
  in-line routines. 74
  predefined symbol 72
  processing 74
  quote characters, specifying. 26
  special characters, using. 72
assembler object file, specifying filename. 28
assembler operators 33
  in expressions. 6
  precedence 33
assembler options
  passing to assembler 3
  specifying parameters 16
  summary 17
  typographic convention xi
assembler output, including debug information 19, 30

assembler source files, including 83
assembler source format 5
assembler subversion number 10
assembler symbols 8
 exporting 57
 importing 58
 in relocatable expressions 11
 local 67
 predefined 9
 redefining 65
assembling, syntax 3
ASSIGN (assembler directive) 64
assumptions (programming experience) ix

B

backtrace information, defining 92
BINAND (assembler operator) 38
BINNOT (assembler operator) 38
BINOR (assembler operator) 39
BINXOR (assembler operator) 39
BLOCK (assembler directive). 56
__BUILD_NUMBER__ (predefined symbol) 9
BYTE1 (assembler operator) 39
BYTE2 (assembler operator) 39
BYTE3 (assembler operator) 40
BYTE4 (assembler operator) 40

C

call frame information directives 92
case sensitivity, controlling 18, 89
CASEOFF (assembler directive). 88
CASEON (assembler directive) 89
--case_insensitive (assembler option) 18
CFI directives 92
CFI expressions 101
CFI operators 101
character constants, ASCII 7
ColdFire architecture and instruction set. ix
command line options
 part of invocation syntax 3
 passing. 3
command line, extending 23
comments
 in assembler list file 89
 in assembler source code 5
 multi-line, using with assembler directives 89
comments, in C-style preprocessor directives 84
common segments 61
COMMON (assembler directive) 59
COMPLEMENT (CFI operator) 101
computer style, typographic convention xi
conditional assembly directives 68
 See also C-style preprocessor directives
conditional code and strings, listing 78
constants
 default base of 89
 integer 6
conventions, typographic xi
CRC, in assembler list file 13
cross-references, in assembler list file. 25, 79
C-style preprocessor directives 81

D

-D (assembler option) 19
data allocation directives. 85
data definition directives. 85
data field, in assembler list file 13
__DATE__ (predefined symbol) 9
DATE (assembler operator). 40
DB (assembler directive) 85
DC8 (assembler directive). 85
DC16 (assembler directive). 85
DC24 (assembler directive). 86
DC32 (assembler directive). 86
DC64 (assembler directive). 86

--debug (assembler option) 19
debug information, including in assembler output 19, 30
default base, for constants. 89
#define (assembler directive) 81
DEFINE (assembler directive) 64
--dependencies (assembler option) 20
DF32 (assembler directive). 86
DF64 (assembler directive). 86
diagnostic messages 109
  classifying as errors 21
  classifying as remarks 21
  classifying as warnings 22
  disabling warnings. 27
  disabling wrapping of 28
  enabling remarks 30
  listing all 22
  suppressing 22
--diagnostics_tables (assembler option) 22
diag_default (#pragma directive) 107
--diag_error (assembler option). 21
diag_error (#pragma directive) 108
--diag_remark (assembler option). 21
diag_remark (#pragma directive) 108
--diag_suppress (assembler option). 22
diag_suppress (#pragma directive) 108
--diag_warning (assembler option) 22
diag_warning (#pragma directive) 108
directives. *See* assembler directives
--dir_first (assembler option) 23
__DIV__ (predefined symbol) 9
DIV (CFI operator). 101
document conventions. xi
DQ15 (assembler directive) 86
DQ31 (assembler directive) 86
DS (assembler directive). 86
DS8 (assembler directive). 86
DS16 (assembler directive). 86
DS24 (assembler directive). 86
DS32 (assembler directive). 86
DS64 (assembler directive). 86
DW (assembler directive) 85

E

efficient coding techniques 14
#elif (assembler directive). 81
#else (assembler directive) 81
ELSE (assembler directive). 68
ELSEIF (assembler directive) 68
--enable_multibytes (assembler option) 23
END (assembler directive) 53
#endif (assembler directive) 81
ENDIF (assembler directive) 68
ENDM (assembler directive) 70
ENDMOD (assembler directive). 53
ENDR (assembler directive) 70
environment variables
  ACF_INC. 4
  ASMCF 4
  assembler 4
EQ (assembler operator) 40
EQ (CFI operator). 101
EQU (assembler directive) 64
#error (assembler directive) 81
error messages 109
  classifying 21
  #error, using to display. 84
--error_limit (assembler option) 23
EVEN (assembler directive) 59
EXITM (assembler directive) 70
experience, programming ix
expressions 6
extended command line file (extend.xcl). 23
EXTERN (assembler directive) 56

F

-f (assembler option). 23

false value, in assembler expressions 8
fatal error messages 110
__FILE__ (predefined symbol). 9
file dependencies, tracking 20
file extensions. *See* filename extensions
file types
    assembler source 3
    extended command line 23
    #include, specifying path 24
filename extensions
    asm 3
    msa 3
    s68 3
    xcl 23
filenames, specifying for assembler output 28
filename, of object file 28
floating-point constants. 7
format, for assembler source code. 5
__FPU__ (predefined symbol) 10
--fpu (assembler option) 24
fractions 8
FRAME (CFI operator). 102
FUNCALL (assembler directive) 91
function directives. 90
FUNCTION (assembler directive) 91

G

GE (assembler operator) 41
GE (CFI operator). 102
global value, defining 65
GT (assembler operator) 41
GT (CFI operator). 102

H

header files, SFR. 14
--header_context (assembler option) 24
HIGH (assembler operator). 41
HWRD (assembler operator) 41

I

-I (assembler option). 24
IAR Technical Support 110
_ _IAR_SYSTEMS_ASM__ (predefined symbol) 10
#if (assembler directive) 81
IF (assembler directive) 68
IF (CFI operator). 103
#ifdef (assembler directive). 81
#ifndef (assembler directive). 81
IMPORT (assembler directive) 56
#include files, specifying 24
#include (assembler directive) 81
include paths, specifying. 24
instruction set
    ColdFire. ix
    identifying (__ISA__) 10
integer constants 6
internal error 110
invocation syntax 3
in-line coding, using macros 74
__ISA__ (predefined symbol). 10
--isa (assembler option). 25

L

-l (assembler option). 25
labels. *See* assembler labels
LE (assembler operator) 42
LE (CFI operator) 102
library modules 54
LIBRARY (assembler directive) 52–53
LIMIT (assembler directive) 64
__LINE__ (predefined symbol) 10
#line (assembler directive) 81
list file format 13
    body 13

CRC 13
header 13
symbol and cross reference
listing control directives 77
LITERAL (CFI operator) 101
LOAD (CFI operator) 103
local value, defining 65
LOCAL (assembler directive) 70
location counter. *See* program location counter
LOCFRAME (assembler directive) 91
LOW (assembler operator) 42
LSHIFT (CFI operator) 102
LSTCND (assembler directive) 77
LSTCOD (assembler directive) 77
LSTEXP (assembler directives) 77
LSTMAC (assembler directive) 78
LSTOUT (assembler directive) 78
LSTREP (assembler directive) 78
LSTXRF (assembler directive) 78
LT (assembler operator) 42
LT (CFI operator) 102
LWRD (assembler operator) 42

M

-M (assembler option) 26
__MAC__ (predefined symbol) 10
--mac (assembler option) 26
macro processing directives 70
macro quote characters 72
specifying 26
MACRO (assembler directive) 70
macros. *See* assembler macros
memory space, reserving and initializing 87
memory, reserving space in 85
message (#pragma directive) 108
messages, excluding from standard output stream 30
--mnem_first (assembler option) 27
MOD (assembler operator) 43
MOD (CFI operator) 102
module consistency 55
module control directives 53
MODULE (assembler directive) 53
modules
assembling multi-modules files 55
terminating 54
msa (filename extension) 3
MUL (CFI operator) 102
multibyte character support 23

N

NAME (assembler directive) 53
NE (assembler operator) 43
NE (CFI operator) 102
NOT (assembler operator) 43
NOT (CFI operator) 101
--no_div (assembler option) 27
--no_path_in_file_macros (assembler option) 27
--no_supervisor (assembler option) 27
--no_warnings (assembler option) 27
--no_wrap_diagnostics (assembler option) 28

O

-o (assembler option) 28
ODD (assembler directive) 59
--only_stdout (assembler option) 28
operands
format of 5
in assembler expressions 6
operations, format of 5
operation, silent 30
operators. *See* assembler operators
option summary 17
OR (assembler operator) 43
OR (CFI operator) 102
ORG (assembler directive) 59

OVERLAY (assembler directive) 56

P

parameters
  specifying 16
  typographic convention xi
PLC. *See* program location counter
#pragma (assembler directive) 81, 107
precedence, of assembler operators. 33
predefined register symbols 9
predefined symbols. 9
  in assembler macros. 72
--preinclude (assembler option) 29
--preprocess (assembler option) 29
preprocessor symbols
  defining and undefining 82
  defining on command line 19
prerequisites (programming experience). ix
program counter. *See* program location counter
program location counter (PLC) 8
  setting 61
program modules, beginning. 54
PROGRAM (assembler directive). 53
programming experience, required ix
programming hints 14
PUBLIC (assembler directive) 56
PUBWEAK (assembler directive). 56

R

-r (assembler option). 30
RADIX (assembler directive) 89
reference information, typographic convention. xi
registered trademarks ii
registers 9
relocatable expressions 11
relocatable segments, beginning 61
remark (diagnostic message). 109
  classifying 21
  enabling 30
--remarks (assembler option) 30
repeating statements 74
REPT (assembler directive) 70
REPTC (assembler directive) 70
REPTI (assembler directive). 70
REQUIRE (assembler directive). 56
RSEG (assembler directive) 59
RSHIFTA (CFI operator) 102
RSHIFTL (CFI operator) 102
RTMODEL (assembler directive). 53
rules, in CFI directives 99
runtime model attributes, declaring. 55

S

segment control directives. 59
segments
  absolute 60
  aligning 62
  common, beginning 61
  relocatable 61
SET (assembler directive). 64
severity level, of diagnostic messages. 109
  specifying 110
SFB (assembler operator) 44
SFE (assembler operator) 44
SFR. *See* special function registers
SHL (assembler operator). 45
SHR (assembler operator). 45
--silent (assembler option) 30
silent operation, specifying 30
simple rules, in CFI directives. 99
SIZEOF (assembler operator). 45
source files
  including 83
  list all referred 24

source format, assembler 5
special function registers. 14
standard error 28
standard output stream, disabling messages to 30
standard output, specifying 28
statements, repeating. 74
stderr. 28
stdout 28
SUB (CFI operator) 102
__SUBVERSION__ (predefined symbol). 10
__SUPERVISOR__ (predefined symbol) 10
Support, Technical 110
symbol and cross-reference table, in assembler list file. . . . 13
    See also Include cross-reference
symbol control directives 56
symbol values, checking 65
SYMBOL (assembler directive) 56
symbols
    See also assembler symbols
    exporting to other modules. 57
    predefined, in assembler 9
    predefined, in assembler macro 72
    user-defined, case sensitive 18
syntax conventions xi
s68 (filename extension). 3

T

Technical Support, IAR 110
temporary values, defining 64
__TID__ (predefined symbol). 10
__TIME__ (predefined symbol) 10
time-critical code 74
trademarks ii
true value, in assembler expressions 8
typographic conventions xi

U

UGT (assembler operator) 46
ULT (assembler operator) 46
UMINUS (CFI operator). 101
#undef (assembler directive) 81
UPPER (assembler operator) 46
user symbols, case sensitive 18

V

value assignment directives. 64
values, defining. 85
VAR (assembler directive) 64
__VER__ (predefined symbol) 10
version, of assembler 10

W

warnings 109
    classifying 22
    disabling 27
    exit code. 30
    treating as errors 30
--warnings_affect_exit_code (assembler option). 4, 30
--warnings_are_errors (assembler option). 30

X

xcl (filename extension) 23
XOR (assembler operator) 47
XOR (CFI operator) 102

Symbols

! (assembler operator). 43
!= (assembler operator). 43
#define (assembler directive) 81
#elif (assembler directive). 81

#else (assembler directive) 81
#endif (assembler directive) 81
#error (assembler directive) 81
#if (assembler directive) 81
#ifdef (assembler directive). 81
#ifndef (assembler directive). 81
#include files, specifying 24
#include (assembler directive) 81
#line (assembler directive) 81
#pragma (assembler directive) 81, 107
#undef (assembler directive). 81
$ (program location counter). 8
% (assembler operator) 43
& (assembler operator) 38
&& (assembler operator) 38
() (assembler operator) 36
* (assembler operator) 36
+ (assembler operator) 36–37
- (assembler operator) 37
-D (assembler option) 19
-f (assembler option). 23
-I (assembler option). 24
-l (assembler option). 25
-M (assembler option). 26
-o (assembler option) 28
-r (assembler option). 30
--case_insensitive (assembler option) 18
--debug (assembler option) 19
--dependencies (assembler option) 20
--diagnostics_tables (assembler option) 22
--diag_error (assembler option). 21
--diag_remark (assembler option) 21
--diag_suppress (assembler option). 22
--diag_warning (assembler option) 22
--dir_first (assembler option) 23
--enable_multibytes (assembler option) 23
--error_limit (assembler option) 23
--fpu (assembler option) 24
--header_context (assembler option). 24
--isa (assembler option). 25
--mac (assembler option) 26
--mnem_first (assembler option). 27
--no_div (assembler option) 27
--no_path_in_file_macros (assembler option). 27
--no_supervisor (assembler option). 27
--no_warnings (assembler option). 27
--no_wrap_diagnostics (assembler option) 28
--only_stdout (assembler option) 28
--preinclude (assembler option) 29
--preprocess (assembler option) 29
--remarks (assembler option) 30
--silent (assembler option) 30
--warnings_affect_exit_code (assembler option). 4, 30
--warnings_are_errors (assembler option). 30
/ (assembler operator) 37
/*...*/ (assembler directive). 88
// (assembler directive) 88
< (assembler operator) 42
<< (assembler operator) 45
<= (assembler operator) 42
<> (assembler operator) 43
= (assembler directive) 64
= (assembler operator) 40
== (assembler operator) 40
> (assembler operator) 41
>= (assembler operator) 41
>> (assembler operator) 45
?: (assembler operator) 38
^ (assembler operator). 39
_args (assembler directive) 70
_args (predefined macro symbol) 72
__ACF__ (predefined symbol) 9
__BUILD_NUMBER__ (predefined symbol) 9
__DATE__ (predefined symbol) 9
__DIV__ (predefined symbol) 9
__FILE__ (predefined symbol). 9
__FPU__ (predefined symbol) 10
__IAR_SYSTEMS_ASM__ (predefined symbol) 10

__ISA__ (predefined symbol) 10
__LINE__ (predefined symbol) 10
__MAC__ (predefined symbol) 10
__SUBVERSION__ (predefined symbol). 10
__SUPERVISOR__ (predefined symbol) 10
__TID__ (predefined symbol). 10
__TIME__ (predefined symbol) 10
__VER__ (predefined symbol) 10
| (assembler operator) 39
|| (assembler operator). 43
~ (assembler operator) 38