H8 IAR Assembler

Reference Guide

for Renesas

H8/300H and H8S Microcomputer Families

EDITION NOTICE

First edition: June 2006

Part number: AH8-1

This guide applies to version 2.x of H8 IAR Embedded Workbench®.

Contents

Tables

Preface

Welcome to the H8 IAR Assembler Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use the H8 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 H8/300H and H8S microcomputer families and need to get detailed reference information on how to use the H8 IAR Assembler. In addition, you should have working knowledge of the following:

- The architecture and instruction set of the H8/300H and H8S microcomputer families; for information refer to the documentation from Renesas
- General assembler language programming
- Application development for embedded systems
- The operating system of your host machine.

How to use this guide

When you first begin using the H8 IAR Assembler, you should read the chapter *Introduction to the H8 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. The *IAR Embedded Workbench® IDE User Guide* also contains a glossary.

What this guide contains

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

- *Introduction to the H8 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.
- *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 H8/300H and H8S microcomputers is described in a series of guides. 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 H8 IAR C/C++ Compiler, refer to the *H8 IAR C/EC++ 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.
- Porting application code and projects created with a previous H8 IAR Embedded Workbench IDE, refer to the *H8 IAR Embedded Workbench® Migration Guide*.

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:

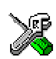
Style	Used for
`computer`	Text that you enter or that appears on the screen.
`parameter`	A label representing the actual value you should enter as part of a command.
`[option]`	An optional part of a command.
`{a | b | c}`	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog 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 H8 IAR Assembler

This chapter contains the following sections:

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

Introduction to assembler programming

Even if you do not intend to write a complete application in assembler language, there may be situations where you will find it necessary to write parts of the code in assembler, for example, when using mechanisms in the H8/300H and H8S microcomputer families 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 H8/300H and H8S microcomputer families. Refer to Renesas hardware documentation for syntax descriptions of the instruction mnemonics.

GETTING STARTED

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

- Work through the tutorials—especially the one about mixing C and assembler modules—that you find in the *IAR Embedded Workbench® IDE User Guide*
- Read about the assembler language interface—also useful when mixing C and assembler modules—in the *H8 IAR C/EC++ Compiler Reference Guide*
- In the IAR Embedded Workbench IDE, you can base a new project on a *template* for an assembler project.

Modular programming

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

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

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

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

Source format

The format of an assembler source line is as follows:

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

where the components are as follows:

label	A definition of a label, which is a symbol that represents an address. If the label starts in the first column—that is, to the leftmost 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.

The H8 IAR Assembler uses the default filename extensions `s37`, `asm`, and `msa` for source files.

Assembler instructions

The H8 IAR Assembler supports the syntax for assembler instructions as described in the chip manufacturer's hardware documentation. It complies with the requirement of the H8/300H and H8S architecture on word alignment. Any instructions in a code segment placed on an odd address will result in a warning.

Expressions, operands, and operators

Expressions consist of expression operands and operators.

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

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators; see also *Assembler operators*, page 27. The valid operators are described in the chapter *Assembler operators*, page 27.

The following operands are valid in an expression:

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

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

INTEGER CONSTANTS

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

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

Commas and decimal points are not permitted.

The following types of number representation are supported:

Integer type	Example
Binary	`1010b`, `b'1010`
Octal	`1234q`, `q'1234, '\123'`
Decimal	`1234`, `-1`, `d'1234`
Hexadecimal	`0FFFFh`, `0xFFFF`, `h'FFFF`

Table 2: Integer constant formats

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

ASCII CHARACTER CONSTANTS

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

Format	Value
'ABCD'	ABCD (four characters).
"ABCD"	ABCD'\0' (five characters the last ASCII null).
'A''B'	A'B
'A'''	A'
'''' (4 quotes)	'
'' (2 quotes)	Empty string (no value).

Table 3: ASCII character constant formats

Format	Value
"" (2 double quotes)	Empty string (an ASCII null character).
\'	', for quote within a string, as in 'I\'d love to'
\\	\, for \ within a string
\"	", for double quote within a string

Table 3: ASCII character constant formats (Continued)

FLOATING-POINT CONSTANTS

The H8 IAR Assembler will accept floating-point values as constants and convert them into IEEE single-precision (signed 64-bit) floating-point format 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 4: 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.

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** (`-s`) assembler option. See *-s*, page 23 for additional information.

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

Note that symbols and labels are byte addresses. For additional information, see *Generating a lookup table*, page 76.

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:

```
       MOV.B      @0x20, R0L  ; Test value
       BNE        ($ + 4):8   ; If non-zero, skip add
       ADD.B      #1, R2H
       MOV        R2H, R3H    ; Branch ends here
...

       BRA        $           ; Loop forever
```

REGISTER SYMBOLS

The register symbols follow the notation used in the Renesas H8/300H and H8S hardware documentation:

Name	Address size	Description
`ER0–ER7`	32 bits	General purpose registers
`E0–E7`	16 bits	General purpose registers
`R0–R7`	16 bits	General purpose registers
`R0L–R7L, R0H–R7H`	8 bits	General purpose registers

Table 5: Predefined register symbols

Name	Address size	Description
CCR	8 bits	Condition code register
EXR	8 bits	Extended control register (H8S only)
MAC, MACL, MACH	64, 32, 10 bits	Multiply-accumulate registers (only some devices in the H8S family)

Table 5: Predefined register symbols

PREDEFINED SYMBOLS

The H8 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
__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.
__CORE__	An integer that identifies the chip core in use.
__CORE_H8300H__	An integer that identifies the H8/300H chip core.
__CORE_H8S__	An integer that identifies the H8S chip core.
__DATE__	The current date in dd/Mmm/yyyy format (string).
__FILE__	The name of the current source file (string).
__IAR_SYSTEMS_ASM__	IAR assembler identifier (number).
__LINE__	The current source line number (number).
__OPERATING_MODE__	An integer that identifies the operating mode in use.
__OPERATING_MODE_NORMAL__	An integer that identifies the Normal operating mode.
__OPERATING_MODE_ADVANCED__	An integer that identifies the Advanced operating mode.
__TID__	Target identity, consisting of two bytes (number). The high byte is the target identity, which is 0x25 for AH8. The low byte is 00 when assembling for the Normal operating mode (-mn) and 01 when assembling for the Advanced operating mode (-ma).

Table 6: Predefined symbols

Symbol	Value
__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 6: Predefined symbols (Continued)

Notice that __TID__ is related to the predefined symbol __TID__ in the H8 IAR C/C++ Compiler. It is described in the *H8 IAR C/EC++ 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:

```
tim     DC8     __TIME__        ; Time string
        ...
        MOV.W     #tim, R0
        JSR       @printstring ; Call string print routine
```

Testing symbols for conditional assembly

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

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

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

See *Conditional assembly directives*, page 57.

ABSOLUTE AND RELOCATABLE EXPRESSIONS

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

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

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

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

```
        NAME    TEST
        ORG     0
        DC.W    start

        RSEG    DATA
first   DS.B    5
second  DS.B    3
        ENDMOD

        RSEG    CODE
start   …
```

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

```
        ADD.B   #first+7,R0L
        ADD.B   #first-7,R0L
        ADD.B   #7+first,R0L
        ADD.B   #(first/second)*start,R0L
```

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

EXPRESSION RESTRICTIONS

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

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

No forward

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

No external

No external references in the expression are allowed.

Absolute

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

Fixed

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

List file format

The format of an assembler list file is as follows:

HEADER

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

BODY

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

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

SUMMARY

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

SYMBOL AND CROSS-REFERENCE TABLE

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

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

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

Table 7: Symbol and cross-reference table

Programming hints

This section gives hints on how to write efficient code for the H8 IAR Assembler. For information about projects including both assembler and C or C++ source files, see the *H8 IAR C/EC++ Compiler Reference Guide*.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of H8/300H and H8S derivatives are included in the IAR Systems product package, in the `h8\inc` directory. These header files define the processor-specific special function registers (SFRs) and interrupt vector numbers.

In general, special function registers are defined with their memory address as the value, while individual SFR bits are defined with the bit number as the value (with the least significant bit having the value 0).

The header files are intended to be used also with the H8 IAR C/C++ Compiler, and they are suitable for use as templates when creating new header files for other H8/300H and H8S 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 77.

Assembler options

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

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

Setting command line options

To set assembler options from the command line, you include them on the command line, after the `ah8` command:

```
ah8 [options] [sourcefile] [options]
```

These items must be separated by one or more spaces or tab characters.

If all the optional parameters are omitted the assembler will display a list of available options a screenful at a time. Press Enter to display the next screenful.

For example, when assembling the source file `power2.s37`, use the following command to generate a list file to the default filename (`power2.lst`):

```
ah8 power2 -L
```

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

```
ah8 power2 -l list.lst
```

Some other options accept a string that is not a filename. This is included after the option letter, but without a space. For example, to generate a list file to the default filename but in the subdirectory named `list`:

```
ah8 power2 -Llist\
```

Note: The subdirectory you specify must already exist. The trailing backslash is required to separate the name of the subdirectory and the default filename.

EXTENDED COMMAND LINE FILE

In addition to accepting options and source filenames from the command line, the assembler can accept them from an extended command line file.

By default, extended command line files have the extension `xcl`, and can be specified using the `-f` command line option. For example, to read the command line options from `extend.xcl`, enter:

```
ah8 -f extend.xcl
```

ERROR RETURN CODES

When using the H8 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 (only if the `-ws` option is used)
2	There were errors

Table 8: Assembler error return codes

ASSEMBLER ENVIRONMENT VARIABLES

Options can also be specified using the `ASMH8` environment variable. The assembler 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 H8 IAR Assembler:

Environment variable	Description
ASMH8	Specifies command line options; for example: `set ASMH8=-L -ws`
AH8_INC	Specifies directories to search for include files; for example: `set AH8_INC=c:\myinc\`

Table 9: Assembler environment variables

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

```
ASMH8=-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*.

Summary of assembler options

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

Command line option	Description
-B	Macro execution information
--bus_width	Specifies bus width in Advanced operating mode
-c{DMEAO}	Conditional list
-D*symbol*[=*value*]	Defines a symbol
-E*number*	Maximum number of errors
-f *filename*	Extends the command line
-G	Opens standard input as source
-I*prefix*	Includes paths
-i	Lists #included text
-L[*prefix*]	Lists to prefixed source name
-l *filename*	Lists to named file
-M*ab*	Macro quote characters
-m{n|a}	Selects operating mode
-N	Omit header from assembler listing
-n	Enables support for multibyte characters
-O*prefix*	Sets object filename prefix
-o *filename*	Sets object filename
--old_style_expressions	Operators use same precedence as in version 1.x
-p*lines*	Lines/page
-r	Generates debug information
-S	Sets silent operation
-s{+|-}	Case sensitive user symbols
-t*n*	Tab spacing
-U*symbol*	Undefines a symbol
-v[0|1|2]	Selects processor option
-w[*string*][s]	Disables warnings
-X	Unreferenced externals in object file
-x{DI2}	Includes cross-references

Table 10: Assembler options summary

Descriptions 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.

-B

`-B`

Use this option to make the assembler print macro execution information to the standard output stream on every call of a macro. The information consists of:

- The name of the macro
- The definition of the macro
- The arguments to the macro
- The expanded text of the macro.

This option is mainly used in conjunction with the list file options `-L` or `-l`; for additional information, see page 19.

To set this option, use:

Project>Options>Assembler>Extra Options

--bus_width

`-bus_width`

When you use the Advanced operating mode, use this option to specify the width of the address bus for the device you are assembling for.

Project>Options>General Options>Target>Address bus width

-c

`-c{DMEAO}`

Use this option to control the contents of the assembler list file. This option is mainly used in conjunction with the list file options `-L` and `-l`; see page 19 for additional information.

The following table shows the available parameters:

Command line option	Description
`-cD`	Disable list file
`-cM`	Macro definitions
`-cE`	No macro expansions

Table 11: Conditional list (-c)

Command line option	Description
-cA	Assembled lines only
-cO	Multiline code

Table 11: Conditional list (-c) (Continued)

To set related options, select:

Project>Options>Assembler>List

-D `-Dsymbol[=value]`

Use this option to define a preprocessor symbol 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

For example, you could 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 in the command line as follows:

Production version: `ah8 prog`
Test version: `ah8 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:

```
ah8 prog -DFRAMERATE=3
```

Project>Options>Assembler>Preprocessor>Defined symbols

-E `-Enumber`

This option specifies the maximum number of errors that the assembler will report.

By default, the maximum number is 100. The `-E` option allows you to decrease or increase this number to see more or fewer errors in a single assembly.

Project>Options>Assembler>Diagnostics>Max number of errors

-f

`-f` *`filename`*

This option extends the command line with text read from the file named `extend.xcl`. 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.

Example

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

```
ah8 prog -f extend.xcl
```

To set this option, use:

Project>Options>Assembler>Extra Options

-G

`-G`

This option causes the assembler to read the source from the standard input stream, rather than from a specified source file.

When `-G` is used, no source filename may be specified.

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

-I

`-I`*`prefix`*

Use this option to specify paths to be used by the preprocessor by adding 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 `AH8_INC` environment variable. The `-I` option allows you to give the assembler the names of directories where it will also search if it fails to find the file in the current working directory.

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 finally in the directory `c:\thisproj\headers\`.

You can also specify the include path with the `AH8_INC` environment variable, see *Assembler environment variables*, page 14.

Project>Options>Assembler>Preprocessor>Additional include directories

---

-i `-i`

Lists `#include` files in the list file.

By default, the assembler does not list `#include` file lines since these often come from standard files and would waste space in the list file. The `-i` option allows you to list these file lines.

To set this option, use:

Project>Options>Assembler>Extra Options

-L `-L[prefix]`

By default the assembler does not generate a list file. Use this option to make the assembler generate one and sent it to file [*prefix*]*sourcename*.lst.

To simply generate a listing, use the `-L` option without a prefix. The listing is sent to the file with the same name as the source, but the extension will be `lst`.

The `-L` option lets you specify a prefix, for example to direct the list file to a subdirectory. Notice that you cannot include a space before the prefix.

`-L` may not be used at the same time as `-l`.

Example

To send the list file to `list\prog.lst` rather than the default `prog.lst`:

```
ah8 prog -Llist\
```

To set related options, select:

Project>Options>Assembler>List

-l `-l filename`

Use this option to make the assembler generate a listing and send it to the file *filename*. If no extension is specified, `lst` is used. Notice that you must include a space before the filename.

By default, the assembler does not generate a list file. The `-l` option generates a listing, and directs it to a specific file. To generate a list file with the default filename, use the `-L` option instead.

To set related options, select:

Project>Options>Assembler>List

`-M` `-M`*`ab`*

This option sets the characters to be used as left and right quotes of each macro argument to *`a`* and *`b`* respectively.

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

Example

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

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

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

`ah8` *`filename`* `-M'<>'`

Project>Options>Assembler>Language>Macro quote characters

`-m` `-m[n|a]`

Use the `-m` option to specify the operating mode.

The following list summarizes the differences between the `-m` options:

`-mn`	Normal operating mode
`-ma`	Advanced operating mode

Project>Options>General Options>Target>Operating mode

-N

`-N`

Use this option to omit the header section that is printed by default in the beginning of the list file.

This option is useful in conjunction with the list file options `-L` or `-l`; see page 19 for additional information.

To set this option, use:

Project>Options>Assembler>Extra Options

-n

`-n`

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 C and C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.

Project>Options>Assembler>Language>Enable multibyte support

-O

`-O`*`prefix`*

Use this option to set the prefix to be used on the name of the object file. Notice that you cannot include a space before the prefix.

By default the prefix is null, so the object filename corresponds to the source filename (unless `-o` is used). The `-O` option lets you specify a prefix, for example to direct the object file to a subdirectory.

Notice that `-O` may not be used at the same time as `-o`.

Example

To send the object code to the file `obj\prog.r37` rather than to the default file `prog.r37`:

```
ah8 prog -Oobj\
```

Project>Options>General Options>Output>Output directories

-o

`-o` *`filename`*

This option sets the filename to be used for the object file. Notice that you must include a space before the filename. If no extension is specified, `r37` is used.

The option `-o` may not be used at the same time as the option `-O`.

Example

For example, the following command puts the object code to the file `obj.r37` instead of the default `prog.r37`:

```
ah8 prog -o obj
```

Notice that you must include a space between the option itself and the filename.

Project>Options>General Options>Output>Output directories

--old_style_expressions

`--old_style_expressions`

Use the `-old_style_expressions` option to make operators have the same precedence as in version 1.x. For more information, see the *H8 IAR Embedded Workbench® Migration Guide.*

To set this option, use:

Project>Options>Assembler>Extra Options

-p

`-p`*`lines`*

The `-p` option sets the number of lines per page to *`lines`*, which must be in the range 10 to 150.

This option is used in conjunction with the list options `-L` or `-l`; see page 19 for additional information.

To set this option, use:

Project>Options>Assembler>Extra Options

-r

`-r`

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

By default, the assembler does not generate debug information, to reduce the size and link time of the object file. You must use the `-r` option if you want to use a debugger with the program.

Project>Options>Assembler>Output>Generate debug information

-S `-S`

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

By default, the assembler sends various insignificant messages via the standard output stream. Use the `-S` 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.

-s `-s{+|-}`

Use the `-s` option to control whether the assembler is sensitive to the case of user symbols:

Command line option	**Description**
`-s+`	Case sensitive user symbols
`-s-`	Case insensitive user symbols

Table 12: Controlling case sensitivity in user symbols (-s)

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

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

-t `-tn`

By default the assembler sets 8 character positions per tab stop. The `-t` option allows you to specify a tab spacing to *n*, which must be in the range 2 to 9.

This option is useful in conjunction with the list options `-L` or `-l`; see page 19 for additional information.

To set this option, use:

Project>Options>Assembler>Extra Options

-U `-Usymbol`

Use the `-U` option to undefine the predefined symbol *`symbol`*.

By default, the assembler provides certain predefined symbols; see *Predefined symbols*, page 7. The `-U` option allows you to undefine such a predefined symbol to make its name available for your own use through a subsequent `-D` option or source definition.

Example

To use the name of the predefined symbol `__TIME__` for your own purposes, you could undefine it with:

```
ah8 prog -U __TIME__
```

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

-v `-v[0|1|2]`

Use the `-v` option to specify the processor core.

The following list summarizes the differences between the `-v` options:

`-v0`	H8/300H (default)
`-v1`	H8S, without MAC instructions
`-v2`	H8S. with MAC instructions

In the IAR Embedded Workbench IDE, the appropriate processor core will automatically be used based on the options **Core** and **MAC**.

-w `-w[string][s]`

By default, the assembler displays a warning message when it detects an element of the source which is legal in a syntactical sense, but may contain a programming error; see *Assembler diagnostics*, page 95, for details.

Use this option to disable warnings. The `-w` option without a range disables all warnings. The `-w` option with a range performs the following:

Command line option	**Description**
`-w+`	Enables all warnings
`-w-`	Disables all warnings
`-w+n`	Enables just warning *n*
`-w-n`	Disables just warning *n*

Table 13: Disabling assembler warnings (-w)

Command line option	Description
`-w+m-n`	Enables warnings *m* to *n*
`-w-m-n`	Disables warnings *m* to *n*

Table 13: Disabling assembler warnings (-w) (Continued)

Only one `-w` option may be used on the command line.

By default, the assembler generates exit code 0 for warnings. Use the `-ws` option to generate exit code 1 if a warning message is produced.

Example

To disable just warning 0 (unreferenced label), use the following command:

```
ah8 prog -w-0
```

To disable warnings 0 to 8, use the following command:

```
ah8 prog -w-0-8
```

To set related options, select:

Project>Options>Assembler>Diagnostics

-X

`-X`

Use this option to force all unreferenced externally declared symbols to be included in the object file.

To set this option, use:

Project>Options>Assembler>Extra Options

-x

`-x{DI2}`

Use this option to make the assembler include a cross-reference table at the end of the list file.

This option is useful in conjunction with the list options `-L` or `-l`; see page 19 for additional information.

The following parameters are available:

Command line option	Description
`-xD`	#defines
`-xI`	Internal symbols

Table 14: Including cross-references in assembler list file (-x)

Command line option	Description
`-x2`	Dual line spacing

Table 14: Including cross-references in assembler list file (-x)

To set the `-x` option without any parameters, use:

Project>Options>Assembler>List>Include cross reference

To set the `-x` option together with any of the parameters, use:

Project>Options>Assembler>Extra Options

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 7 (the lowest precedence, that is, last evaluated).

Note: There are no operators that have the precedence level 2.

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))
```

Summary of assembler operators

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

UNARY OPERATORS – 1

`+`	Unary plus.
`-`	Unary minus.
`NOT, !`	Logical NOT.
`BINNOT, ~`	Bitwise NOT.
`LOW`	Low byte.

`HIGH`	High byte.
`BYTE3`	Third byte.
`LWRD`	Low word.
`HWRD`	High word.
`DATE`	Current time/date.
`SFB`	Segment begin.
`SFE`	Segment end.
`SIZEOF`	Segment size.

MULTIPLICATIVE ARITHMETIC OPERATORS – 3

`*`	Multiplication.
`/`	Division.
`%, MOD`	Modulo.
`>>, SHR`	Logical shift right.
`<<, SHL`	Logical shift left.

ADDITIVE ARITHMETIC OPERATORS – 4

`+`	Addition.
`-`	Subtraction.

AND OPERATORS – 5

`&&, AND`	Logical AND.
`&, BINAND`	Bitwise AND.

OR OPERATORS – 6

`||, OR`	Logical OR.
`|, BINOR`	Bitwise OR.
`XOR`	Logical exclusive OR.
`^, BINXOR`	Bitwise exclusive OR.

COMPARISON OPERATORS – 7

=, ==, EQ	Equal.
<>, !=, NE	Not equal.
>, GT	Greater than.
<, LT	Less than.
UGT	Unsigned greater than.
ULT	Unsigned less than.
>=, GE	Greater than or equal.
<=, LE	Less than or equal.

Description of operators

The following sections give detailed descriptions of each assembler operator. See *Expressions, operands, and operators*, page 3, for related information. The number within parentheses specifies the priority of the operator.

* Multiplication (3).

* 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 (1).

Unary plus operator.

Example

```
+3 → 3
3*+2 → 6
```

+ Addition (4).

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 (1).

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 (4).

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 (3).

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

Example

```
9/2 → 4
-12/3 → -4
9/2*6 → 24
```

`<, LT` Less than (7).

`<`, alternatively `LT`, 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
```

`<=, LE` Less than or equal (7)

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

Example

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

`<>, !=, NE` Not equal (7).

`<>`, alternatively `!=` or `NE`, evaluates to 0 (false) if its two operands are identical in value or to 1 (true) if its two operands are not identical in value.

Example

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

`=, ==, EQ` Equal (7).

`=`, alternatively `==` or `EQ`, evaluates to 1 (true) if its two operands are identical in value, or to 0 (false) if its two operands are not identical in value.

Note: The meaning or precedence of the `==` operator is affected when using the `--old_style_expression` option. For more information, see the *H8 IAR Embedded Workbench® Migration Guide*.

Example

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

`>, GT` Greater than (7).

`>`, alternatively `GT`, 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
```

`>=, GE` Greater than or equal (7).

`>=`, alternatively `GE`, 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
```

`&&, AND` Logical AND (5).

Use `&&`, alternatively `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).

Note: The meaning or precedence of the `&&` operator is affected when using the `--old_style_expression` option. For more information, see the *H8 IAR Embedded Workbench® Migration Guide.*

Example

```
B'1010 && B'0011 → 1
B'1010 && B'0101 → 1
B'1010 && B'0000 → 0
```

&, BINAND

Bitwise AND (5).

Use &, alternatively 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.

Note: The meaning or precedence of the & operator is affected when using the `--old_style_expression` option. For more information, see the *H8 IAR Embedded Workbench® Migration Guide.*

Example

```
B'1010 & B'0011 → B'0010
B'1010 & B'0101 → B'0000
B'1010 & B'0000 → B'0000
```

~, BINNOT

Bitwise NOT (1).

Use ~, alternatively 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.

Note: The meaning or precedence of the ~ operator is affected when using the `--old_style_expression` option. For more information, see the *H8 IAR Embedded Workbench® Migration Guide.*

Example

```
~ B'1010 → B'11111111111111111111111111110101
```

|, BINOR

Bitwise OR (6).

Use |, alternatively 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.

Note: The meaning or precedence of the | operator is affected when using the `--old_style_expression` option. For more information, see the *H8 IAR Embedded Workbench® Migration Guide.*

Example

```
B'1010 | B'0101 → B'1111
B'1010 | B'0000 → B'1010
```

^, BINXOR Bitwise exclusive OR (6).

Use ^, alternatively 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.

Note: The meaning or precedence of the ^ operator is affected when using the --old_style_expression option. For more information, see the *H8 IAR Embedded Workbench® Migration Guide*.

Example

```
B'1010 ^ B'0101 → B'1111
B'1010 ^ B'0011 → B'1001
```

%, MOD Modulo (3).

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

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

Example

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

!, NOT Logical NOT (1).

Use !, alternatively NOT, to negate a logical argument.

Example

```
! B'0101 → 0
! B'0000 → 1
```

||, OR Logical OR (6).

Use ||, alternatively OR, to perform a logical OR between two integer operands.

Note: The meaning or precedence of the || operator is affected when using the --old_style_expression option. For more information, see the *H8 IAR Embedded Workbench® Migration Guide*.

Example

```
B'1010 || B'0000 → 1
B'0000 || B'0000 → 0
```

`BYTE3` Third byte (1).

`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
```

`DATE` Current time/date (1).

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 6, DATE 5, DATE 4
```

`HIGH` High byte (1).

`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 (1).

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

LOW Low byte (1).

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

LWRD Low word (1).

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

SFB Segment begin (1).

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.

Description

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 linking time.

Example

```
       NAME  demo
       RSEG  CODE
start: DC16  SFB(CODE)
```

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 (1).

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.

Description

`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 linking time.

Example

```
      NAME  demo
      RSEG  CODE
end:  DC16  SFE(CODE)
```

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

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

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

`<<, SHL` Logical shift left (3).

Use `<<`, alternatively `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

```
B'00011100 << 3 → B'11100000
B'0000011111111111 << 5 → B'1111111111100000
14 << 1 → 28
```

`>>, SHR` Logical shift right (3).

Use `>>`, alternatively `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

```
B'01110000 >> 3 → B'00001110
B'1111111111111111 >> 20 → 0
14 >> 1 → 7
```

`SIZEOF` Segment size (1).

Syntax

`SIZEOF` *`segment`*

Parameters

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

Description

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

Example

```
        NAME   demo
        RSEG   CODE
```

```
size: DC16   SIZEOF CODE
```

sets `size` to the size of the segment `CODE`.

`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 the 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
```

`XOR` Logical exclusive OR (6).

`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

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

Assembler directives

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

Summary of assembler directives

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

Directive	Description	Section
`#define`	Assigns a value to a label.	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
`#ifndef`	Assembles instructions if a symbol is undefined.	C-style preprocessor
`#include`	Includes a file.	C-style preprocessor
`#pragma`	Recognized but ignored.	C-style preprocessor
`#line`	Recognized but ignored.	C-style preprocessor
`#message`	Generates a message on standard output.	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
`ALIAS`	Assigns a permanent value local to a module.	Value assignment
`ALIGN`	Aligns the location counter by inserting zero-filled bytes.	Segment control
`ALIGNRAM`	Aligns without inserting.	Segment control
`ARGFRAME`	Declares the space used for the arguments to a function.	Function
`ASEG`	Begins an absolute segment.	Segment control

Table 15: Assembler directives summary

Directive	Description	Section
ASEGN	Begins an absolute segment.	Segment control
ASSIGN	Assigns a temporary value.	Value assignment
CASEOFF	Disables case sensitivity.	Assembler control
CASEON	Enables case sensitivity.	Assembler control
CFI	Specifies call frame information.	Call frame information
COL	Sets the number of columns per page.	Listing control
COMMON	Begins a common segment.	Segment control
DC[.*size*]	Generates *size*-bit words constants, including strings.	Data definition or allocation
DC16	Generates 16-bit word constants, including strings.	Data definition or allocation
DC24	Generates 24-bit word constants.	Data definition or allocation
DC32	Generates 32-bit long word constants.	Data definition or allocation
DC8	Generates 8-bit byte constants, including strings.	Data definition or allocation
DEFINE	Defines a file-wide value.	Value assignment
DS[.*size*]	Allocates space for *size*-bit words.	Data definition or allocation
DS16	Allocates space for 16-bit words.	Data definition or allocation
DS24	Allocates space for 24-bit words.	Data definition or allocation
DS32	Allocates space for 32-bit words.	Data definition or allocation
DS8	Allocates space for 8-bit bytes.	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

Table 15: Assembler directives summary (Continued)

Directive	Description	Section
ENDM	Ends a macro definition.	Macro processing
ENDMOD	Terminates the assembly of the current module.	Module control
ENDR	Ends a REPT, REPTC or REPTI structure.	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment
EVEN	Aligns the program counter to an even address.	Segment control
EXITM	Exits prematurely from a macro.	Macro processing
EXPORT	Exports symbols to other modules.	Symbol control
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	Controls conditional assembler listing.	Listing control
LSTCOD	Controls multi-line code listing.	Listing control
LSTEXP	Controls the listing of macro generated lines.	Listing control
LSTMAC	Controls the listing of macro definitions.	Listing control
LSTOUT	Controls assembler-listing output.	Listing control
LSTPAG	Controls the formatting of output into pages.	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 counter to an odd address.	Segment control
ORG	Sets the location counter.	Segment control
PAGE	Generates a new page.	Listing control

Table 15: Assembler directives summary (Continued)

Directive	Description	Section
PAGSIZ	Sets the number of lines per page.	Listing 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
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
STACK	Begins a stack segment.	Segment control
VAR	Assigns a temporary value.	Value assignment

Table 15: 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.

Directive	Description
END	Terminates the assembly of the last module in a file.
ENDMOD	Terminates the assembly of the current module.
LIBRARY	Begins a library module.
MODULE	Begins a library module.
NAME	Begins a program module.
PROGRAM	Begins a program module.
RTMODEL	Declares runtime model attributes.

Table 16: Module control directives

SYNTAX

```
END [label]
ENDMOD [label]
```

```
LIBRARY symbol
MODULE symbol
NAME symbol
PROGRAM symbol
RTMODEL key, value
```

PARAMETERS

key	A text string specifying the key.
label	An expression or label that can be resolved at assembly time. It is output in the object code as a program entry address.
symbol	Name assigned to module, used by XLINK XAR, and XLIB when processing object files.
value	A text string specifying the value.

DESCRIPTION

Beginning a program module

Use `NAME`, alternatively `PROGRAM`, to begin a program module, and to assign a name for future reference by the IAR XLINK Linker 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`, alternatively `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 last module

Use `END` to indicate the end of the source file. Any lines after the `END` directive are ignored.

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 used in the *last* module, and there must not be any source lines (except for comments and listing control directives) between an `ENDMOD` and a `MODULE` directive.

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

Declaring runtime model attributes

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

A module can have several runtime model definitions.

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

If you are writing assembler routines for use with C or C++ code, and you want to control the module consistency, refer to the *H8 IAR C/EC++ 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
EXTERN, IMPORT	Imports an external symbol.
PUBLIC, EXPORT	Exports symbols to other modules.
PUBWEAK	Exports symbols to other modules; multiple definitions allowed.
REQUIRE	Forces a symbol to be referenced.

Table 17: Symbol control directives

SYNTAX

EXTERN *symbol* [,*symbol*] …
PUBLIC *symbol* [,*symbol*] …
PUBWEAK *symbol* [,*symbol*] …
REQUIRE *symbol*

PARAMETERS

symbol Symbol to be imported or exported.

DESCRIPTION

Exporting symbols to other modules

Use PUBLIC to make one or more symbols available to other modules. Symbols declared 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-declared symbols in a module.

Exporting symbols with multiple definitions to other modules

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

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

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

Importing symbols

Use EXTERN to import an untyped external symbol.

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

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

```
     NAME   login
     EXTERN print
     PUBLIC err

err  JSR    @print
     DC.B   "Please login:"
     RTS

     END    err
```

Segment control directives

The segment directives control how code and data are located.

Directive	Description
ALIGN	Aligns the location counter by inserting zero-filled bytes.
ALIGNRAM	Aligns the program location counter.
ASEG	Begins an absolute segment.
ASEGN	Begins a named absolute segment.
COMMON	Begins a common segment.
EVEN	Aligns the program counter to an even address.
ODD	Aligns the program counter to an odd address.
ORG	Sets the location counter.
RSEG	Begins a relocatable segment.
STACK	Begins a stack segment.

Table 18: Segment control directives

SYNTAX

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

```
RSEG segment [:type] [flag] [(align)]
RSEG segment [:type], address
STACK segment [:type] [(align)]
```

PARAMETERS

address	Address where this segment part will be placed.
align	Exponent of the value to which the address should be aligned, in the range 0 to 30.
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.

DESCRIPTION

Beginning an absolute segment

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

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

Beginning a named absolute segment

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

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

Beginning a relocatable segment

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

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

Beginning a stack segment

Use `STACK` to allocate code or data allocated from high to low addresses (in contrast with the `RSEG` directive that causes low-to-high allocation).

Note: The contents of the segment are not generated in reverse order.

Beginning a common segment

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

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

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

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

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

Setting the program location counter (PLC)

Use ORG to set the program location counter of the current segment to the value of an expression. The optional label will assume the value and type of the new location counter.

The result of the expression must be of the same type as the current segment, i.e. it is not valid to use ORG 10 during RSEG, since the expression is absolute; use ORG $+10 instead. The expression must not contain any forward or external references.

All program location counters are set to zero at the beginning of an assembly 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.

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. The EVEN directive aligns the program counter to an even address (which is equivalent to ALIGN 1) and the ODD directive aligns the program counter to an odd address.

Use ALIGNRAM to align the program location counter by incrementing it; no data is generated. The expression can be within the range 0 to 31.

EXAMPLES

Beginning an absolute segment

The following example makes the subroutine subr start in a next new page after address 123:

```
  10   0000007B         subr  ASEG        123 (8)
  11   00000100 F90A          MOV         #10,R1L
  12   00000102 5090          MULXU       R1L,R0
  13   00000104 5470          RTS
  14   00000106
  15   00000106               END         main
```

After assembling this code subr will have the value 100.

Beginning a relocatable segment

The following directive aligns the start address of segment `MYSEG` (upwards) to the nearest 8 byte (2**3) page boundary:

```
RSEG MYSEG :CODE(3)
```

Note that only the first segment directive for a particular segment can contain an alignment operand.

Beginning a stack segment

The following example defines two 100-byte stacks in a relocatable segment called `rpnstack`:

```
         STACK   rpnstack
parms    DS.B    100
opers    DS.B    100
         END
```

The data is allocated from high to low addresses.

Beginning a common segment

The following example defines two common segments containing variables:

```
         NAME    common1
         COMMON  data
count    DS.B    4
         ENDMOD

         NAME    common2
         COMMON  data
up       DS.B    1
         ORG     $+2
down     DS.B    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`.

Setting the location counter

The following example uses `ORG` to leave a gap of 256 bytes:

```
        NAME    org
        ORG     $+256
begin   MOV     #12,R3L
        MULXU   R3L,R2
        RTS
        END     begin
```

Aligning a segment

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

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

Value assignment directives

These directives are used for assigning values to symbols.

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

Table 19: Value assignment directives

SYNTAX

```
label = expr
label ALIAS expr
label ASSIGN expr
label DEFINE expr
label EQU expr
```

```
LIMIT expr, min, max, message
label SET expr
label VAR expr
```

PARAMETERS

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*.

DESCRIPTION

Defining a temporary value

Use either 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 assign a value to a symbol.

Use EQU 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.

Use EXTERN to import symbols from other modules.

Defining a permanent global value

Use DEFINE to define symbols that should be known to all modules in the source file.

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

Symbols defined with DEFINE cannot be redefined within the same file.

Checking symbol values

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

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

EXAMPLES

Redefining a symbol

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

```
         NAME     table
main     ; Generate table of powers of 3
const    SET      1
         REPT     8
         DC.W     const
const    SET      const*3
         ENDR
         END      main
```

Using local and global symbols

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

```
         NAME   add1
locn     DEFINE 100H
value    EQU    77
         MOV    @locn,R1L
         ADD    #value,R1L
         ENDMOD

         NAME   add2
value    EQU    88
         MOV    @locn,R2L
         ADD    #value,R2L
         END
```

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

Using the LIMIT directive

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

```
speed     VAR        23
LIMIT     speed,10,30,...speed out of range...
```

Conditional assembly directives

These directives provide logical control over the selective assembly of source code.

Directive	Description
`IF`	Assembles instructions if a condition is true.
`ELSE`	Assembles instructions if a condition is false.
`ELSEIF`	Specifies a new condition in an `IF`...`ENDIF` block.
`ENDIF`	Ends an `IF` block.

Table 20: Conditional assembly directives

SYNTAX

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

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.

DESCRIPTION

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

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

All assembler directives (except `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 assembles instructions to multiply `R0L` by a constant, but omits them if the argument is 1:

```
     NAME   mult

mult MACRO  k
     IF     k <> 1
     MOV    #k,R1L
     MULXU  R1L,R0
     ENDIF
     ENDM
```

It could be tested with the following program:

```
main mov    #23,R0L
     mult   7
     END    main
```

Macro processing directives

These directives allow user macros to be defined.

Directive	Description
`ENDM`	Ends a macro definition.
`ENDR`	Ends a repeat structure.
`EXITM`	Exits prematurely from a macro.
`LOCAL`	Creates symbols local to a macro.

Table 21: Macro processing directives

Directive	Description
MACRO	Defines a macro.
REPT	Assembles instructions a specified number of times.
REPTC	Repeats and substitutes characters.
REPTI	Repeats and substitutes strings.

Table 21: Macro processing directives

SYNTAX

```
ENDM
ENDR
EXITM
LOCAL symbol [,symbol] …
name MACRO [,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.

DESCRIPTION

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:

```
macroname MACRO [,arg] [,arg] …
```

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

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

```
getbyte MACRO   port
        JSR     @waitdata
        MOV.B   port,R0L
        ENDM
```

This macro uses a parameter `port` to specify the port address to read from. You would call the macro with a statement such as:

```
        getbyte  @0x8000
```

The assembler will expand this to:

```
        JSR       @waitdata
        MOV.B     @0x8000,R0L
```

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:

```
getbyte  MACRO
         JSR      @waitdata
         MOV.B    \1,R0L
         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:

```
mac1     MACRO  regs
         ADD    regs   ; add content of reg1 to reg2
         ENDM
```

The macro can be called using the macro quote characters:

```
         mac1  <R1L,R2L>
         END
```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 20.

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 for checking that there are at most 10 arguments:

```
COUNT MACRO parm
     IF     _args > 10
     EXITM
     ENDIF
     DC.B   _args
     ENDM
```

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. Include-file references `$file` are recorded and will be included during macro *expansion*.

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.

Extending the instruction set

Because of the way in which microprocessor instruction sets evolve, they are not always as symmetrical as you would like. By writing macros, you can add definitions for instructions that you would like to have included in the instruction set, and use them just like the built-in instructions.

For example, the SUB instruction does not support immediate data for byte operands. If you frequently need this operation, you can define a subi macro to do this as follows:

```
subi    MACRO   immed,reg
        ORC     #H'01,CCR
        SUBX    #(immed-1),reg
        ENDM
```

This could then be used in a program as follows:

```
main    subi    27,R0L
        RTS
        END
```

Coding inline for efficiency

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

The following example outputs bytes from a 256-byte buffer to a port:

```
        EXTERN  port

        RSEG    RAM
buffer  DS.B    256

        RSEG    PROM
; Plays 256 bytes from buffer to port
play    MOV     #0,R1
loop    MOV     @(buffer,R1),R2L
        MOV     R2L,@port
        INC     R1L
        BNE     loop
        RTS
        END
```

The main program calls this routine as follows:

```
doplay  JSR     @play
```

For efficiency we can recode this using a macro:

```
; Plays 256 bytes from buffer to port
play    MACRO
        LOCAL   loop
        MOV     #0,R1
loop    MOV     @(buffer,R1),R2L
        MOV     R2L,@port
        INC     R1L
        BNE     loop
        ENDM
        END
```

Note 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.

To use inline code, the main program is then simply altered to:

```
doplay  play
```

Exiting from a macro

The following example defines a macro to rotate the register `R1L` a specified number of times, `r`. It uses `EXITM` to exit from the macro if `r` is 8, because no rotates are needed:

```
        NAME    rotate

rotate  MACRO   r
        IF      r = 8
        EXITM
```

```
        ENDIF
        REPT    r
        ROTR    R1L
        ENDR
        END
```

Using REPT

The following example uses `REPT` to assemble a table of powers of 3:

```
        NAME    table

main    ; Generate table of powers of 3
calc    SET     1
        REPT    8
        DC.W    calc
calc    SET     calc * 3
        ENDR
        END     main
```

It generates the following code:

```
1    00000000                 NAME    table
2    00000000
3    00000000          main    ; Generate table of powers of 3
4    00000001          calc    SET     1
5    00000000                  REPT    8
6    00000000                  DC.W    calc
7    00000000          calc    SET     calc * 3
8    00000000                  ENDR
8.1  00000000 0001             DC.W    calc
8.2  00000003          calc    SET     calc * 3
8.3  00000002 0003             DC.W    calc
8.4  00000009          calc    SET     calc * 3
8.5  00000004 0009             DC.W    calc
8.6  0000001B          calc    SET     calc * 3
8.7  00000006 001B             DC.W    calc
8.8  00000051          calc    SET     calc * 3
8.9  00000008 0051             DC.W    calc
8.10 000000F3          calc    SET     calc * 3
8.11 0000000A 00F3             DC.W    calc
8.12 000002D9          calc    SET     calc * 3
8.13 0000000C 02D9             DC.W    calc
8.14 0000088B          calc    SET     calc * 3
8.15 0000000E 088B             DC.W    calc
8.16 000019A1          calc    SET     calc * 3
9    00000010                  END     main
```

Using REPTC and REPTI

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

```
prompt   REPTC  chr, "Login:"
         MOV    #char,R0L
         JSR    @putc
         ENDR
```

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

```
main     MOV    #0,R0L
         REPTI  zero,flag,temp,"(base,R1)"
         MOV    R0L,@zero
         ENDR
```

Listing control directives

These directives provide control over the assembler list file.

Directive	Description
`COL`	Sets the number of columns per page.
`LSTCND`	Controls conditional assembly listing.
`LSTCOD`	Controls multi-line code listing.
`LSTEXP`	Controls the listing of macro-generated lines.
`LSTMAC`	Controls the listing of macro definitions.
`LSTOUT`	Controls assembler-listing output.
`LSTPAG`	Controls the formatting of output into pages.
`LSTREP`	Controls the listing of lines generated by repeat directives.
`LSTXRF`	Generates a cross-reference table.
`PAGE`	Generates a new page.
`PAGSIZ`	Sets the number of lines per page.

Table 22: Listing control directives

SYNTAX

```
COL columns
LSTCND{+|-}
LSTCOD{+|-}
LSTEXP{+|-}
LSTMAC{+|-}
LSTOUT{+|-}
```

```
LSTPAG{+|-}
LSTREP{+|-}
LSTXRF{+|-}
PAGE
PAGSIZ lines
```

PARAMETERS

columns	An absolute expression in the range 80 to 132, default is 80
lines	An absolute expression in the range 10 to 150, default is 44

DESCRIPTION

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 restrict the listing of output code to just the first line of code for a source line.

The default setting is `LSTCOD+`, which lists more than one line of code for a source line, if needed; i.e. long ASCII strings will produce several lines of output. Code generation is *not* affected.

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.

Specifying the list file format

Use COL to set the number of columns per page of the assembler list. The default number of columns is 80.

Use PAGSIZ to set the number of printed lines per page of the assembler list. The default number of lines per page is 44.

Use LSTPAG+ to format the assembler output list into pages.

The default is LSTPAG-, which gives a continuous listing.

Use PAGE to generate a new page in the assembler list file if paging is active.

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    lstcnd
        EXTERN  print

        RSEG    prom
debug   SET     0

begin   IF      debug
        JSR     @print
        ENDIF
```

```
        LSTCND+
begin2  IF      debug
        JSR     @print
        ENDIF
        END     begin
```

This will generate the following listing:

```
      1    00000000                  NAME    lstcnd
      2    00000000                  EXTERN  print
      3    00000000
      4    00000000                  RSEG    prom
      5    00000000          debug   SET     0
      6    00000000
      7    00000000          begin   IF      debug
      8    00000000                  JSR     print
      9    00000000                  ENDIF
     10    00000000
     11    00000000                  LSTCND+
     12    00000000          begin2  IF      debug
     13    00000000                  ENDIF
     14    00000000                  END     begin
```

The following example shows the effect of LSTCOD+ on the generated code generated by a DC directive:

```
        NAME    lstcod
table1  DC      1,10,100,1000,10000
        LSTCOD+
table2  DC      1,10,100,1000,10000
        END
```

This will generate the following listing:

```
     1    00000000                    NAME    lstcod
     2    00000000 0001000A*table1    DC      1,10,100,1000,10000
     3    0000000A                    LSTCOD+
     4    0000000A 0001000A table2    DC      1,10,100,1000,10000
                   006403E8
                   2710
     5    00000014                    END
```

Controlling the listing of macros

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

```
        NAME    lstmac

times2  MACRO   reg
        SHAL    reg
        ENDM

        LSTMAC+
div2    MACRO   reg
        SHLR    reg
        ENDM

begin
        times2  R2L

        LSTEXP-
        div2    R1H
        RTS
        END     begin
```

This will produce the following output:

```
    1    00000000                    NAME   lstmac
    2    00000000
    6    00000000
    7    00000000                    LSTMAC+
    8    00000000            div2    MACRO  reg
    9    00000000                    SHLR   reg
   10    00000000                    ENDM
   11    00000000
   12    00000000            begin
   13    00000000                    times2 R2L
   13.1  00000000 108A               SHAL   R2L
   13.2  00000002                    ENDM
   14    00000002
   15    00000002                    LSTEXP-
   16    00000002                    div2   R1H
   17    00000004 5470               RTS
   18    00000006                    END    begin
```

Formatting listed output

The following example formats the output into pages of 66 lines each with 132 columns. The LSTPAG directive organizes the listing into pages, starting each module on a new page. The PAGE directive inserts additional page breaks.

```
PAGSIZ 66  ; Page size
COL 132
LSTPAG+
...
ENDMOD
MODULE
...
PAGE
...
```

C-style preprocessor directives

The C-style preprocessor directives are processed before other assembler directives. The following C-language preprocessor directives are available:

Directive	Description
#define	Assigns a value to a label.
#elif	Introduces a new condition in a #if...#endif block.
#else	Assembles instructions if a condition is false.
#endif	Ends a #if, #ifdef, or #ifndef block.
#error	Generates an error.
#if	Assembles instructions if a condition is true.
#ifdef	Assembles instructions if a symbol is defined.
#ifndef	Assembles instructions if a symbol is undefined.
#include	Includes a file.
#line	This directive is recognized but ignored.
#message	Generates a message on standard output.
#pragma	This directive is recognized but ignored.
#undef	Undefines a label.

Table 23: C-style preprocessor directives

SYNTAX

```
#define label text
#elif condition
#else
```

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

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=string	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.
filename	Name of file to be included.	
label	Symbol to be defined, undefined, or tested.	
message	Text to be displayed.	
text	Value to be assigned.	

DESCRIPTION

Defining and undefining labels

Use `#define` to define a temporary label.

```
#define label value
```

is similar to:

```
label VAR value
```

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

Conditional directives

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

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

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

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

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

Including source files

Use `#include` to insert the contents of a file into the source file at a specified point.

`#include "`*`filename`*`"` searches the following directories in the specified order:

1. The source file directory.
2. The directories specified by the `-I` option, or options.
3. The current directory.

`#include <`*`filename`*`>` searches the following directories in the specified order:

1. The directories specified by the `-I` option, or options.
2. The current directory.

Displaying errors

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

Ignoring #pragma and #line

Lines with `#pragma` or `#line` are ignored by the assembler, making it easier to have header files common to C and assembler.

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 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 */

  MOV  #five,R5         ;syntax error!
  ; Expands to "MOV #5 ; this comment is not OK,R5"

  MOV  #(six + seven),R5; OK
  ; Expanded to "MOV #(6+7),R5"
```

EXAMPLES

Using conditional directives

The following example defines the label `adjust`, and then uses the conditional directive `#ifdef` to use the value if it is defined:

```
        EXTERN  input
#define adjust  10

main    MOV     @input,R0L
#ifdef  adjust
        ADD.B   #adjust,R0L
#else
        ADD.B   #7,R0L
#endif
        MOV     R0L,@input
        RTS
        END
```

Including a source file

The following example uses `#include` to include a file defining macros into the source file. For example, the following macros could be defined in `macros.s37`:

```
subi    MACRO   const,reg
        ORC     #H'01,CCR
        SUBX    #(const-1),reg
        ENDM

addi    MACRO   const,reg
        ADD.B   #const,reg
        ENDM
```

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

```
        NAME   include

; standard macro definitions
#include "macros.s37"

; program
main    MOV    #123,R0L
        subi   99,R0L
        RTS
        END    main
```

Displaying errors

The following example generates an error if a label is undefined:

```
main    MOV    #3,R0
#ifndef level
#error  "Not defined"
#endif
        RTS
        ENDM   main
```

Data definition or allocation directives

These directives define values or reserve memory:

Directive	Description	Expression restrictions
`DC[.size]`	Generates constants, including strings.	
`DC8`	Generates 8-bit constants, including strings.	
`DC16`	Generates 16-bit constants.	
`DC24`	Generates 24-bit constants.	
`DC32`	Generates 32-bit constants.	
`DS[.size]`	Allocates space.	No external references Absolute
`DS8`	Allocates space for 8-bit constants.	No external references Absolute
`DS16`	Allocates space for 16-bit constants.	No external references Absolute
`DS24`	Allocates space for 24-bit constants.	No external references Absolute
`DS32`	Allocates space for 32-bit constants.	No external references Absolute

Table 24: Data definition or allocation directives

SYNTAX

```
DC[.size] expr [,expr] ...
DC8 expr [,expr] ...
DC16 expr [,expr] ...
DC24 expr [,expr] ...
DC32 expr [,expr] ...
DS[.size] count
DS8 count
DS16 count
DS24 count
DS32 count
```

PARAMETERS

expr Specifies the constant, which can be 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.*

size — For DC, the element size of generated values. For DS, the size of each element reserved space for. One of:

.B	Byte
.W	Word (the default); that is 2 bytes
.L	Long word; that is 4 bytes
.S	Single; that is 4-byte floating-point constant*
.D	Double; that is 8-byte floating-point constant*

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

* **For DC.S and DC.D, the parameter expr is limited to a valid absolute expression or a floating-point constant.**

DESCRIPTIONS

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

Size	Reserve and initialize memory	Reserve uninitialized memory
8-bit integers	DC8, DC.B	DS8, DS.B
16-bit integers	DC16, DC.W	DS16, DS.W
24-bit integers	DC24	DS24
32-bit integers	DC32, DC.L	DS32, DS.L
32-bit floats	DC32, DC.S	DS32, DS.S
64-bit floats	DC.D	DS.D

Table 25: Using data definition or allocation directives

EXAMPLES

Generating a lookup table

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

```
            NAME        table
table       DC          addsubr2,subsubr,clrsubr

addsubr     ADD         R1,R0
            RTS
```

```
subsubr     SUB         R1,R0
            RTS

clrsubr     MOV         #0,R0
            RTS

            END
```

Defining strings

To define a string:

```
mymsg   DC8 'Please enter your name'
```

To define a string which includes a trailing zero:

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

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

```
errmsg  DC8 'Don''t understand!'
```

Reserving space

To reserve space for `0xA` bytes:

```
table   DS8   0xA
```

Assembler control directives

These directives provide control over the operation of the assembler.

Directive	Description
`/*comment*/`	C-style comment delimiter.
`//`	C++ style comment delimiter.
`CASEOFF`	Disables case sensitivity.
`CASEON`	Enables case sensitivity.
`RADIX`	Sets the default base on all numeric values.

Table 26: Assembler control directives

SYNTAX

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

PARAMETERS

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

DESCRIPTION

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 off.

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 2: 01.10.05
Author: mjp
*/
```

Changing the base

To set the default base to 16:

```
        RADIX  D'16
        MOV    #12,R3
```

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

Controlling case sensitivity

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

```
label   NOP       ; Stored as "LABEL"
        JMP       @LABEL
```

However, the following will generate a duplicate label error:

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

Function directives

The function directives are generated by the H8 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 call frame information** (`-lA`).

Note: These directives are primarily intended to support static overlay, a feature which is useful in smaller microcontrollers. The H8 IAR C/C++ Compiler does not use static overlay, as it has no use for it.

SYNTAX

```
FUNCTION <label>,<value>
ARGFRAME &lt;segment&gt;, &lt;size&gt;, &lt;type&gt;
LOCFRAME &lt;segment&gt;, &lt;size&gt;, &lt;type&gt;
FUNCALL <caller>, <callee>
```

PARAMETERS

label	A label to be declared as a function.
value	Function information.
segment	The segment in which the argument frame or local frame will 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`.
caller	A caller to a function.
callee	The called function.

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.

Table 27: Call frame information directives

Directive	Description
`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 27: Call frame information directives (Continued)

SYNTAX

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

Names block directives

```
CFI NAMES name
CFI ENDNAMES name
CFI RESOURCE resource : bits [, resource : bits] …
CFI VIRTUALRESOURCE resource : bits [, resource : bits] …
CFI RESOURCEPARTS resource part, part [, part] …
CFI STACKFRAME cfa resource type [, cfa resource type] …
CFI STATICOVERLAYFRAME cfa 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 89).
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 – -1 and 1 – 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 to 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 81. For more information on these directives, see *Simple rules*, page 87, and *CFI expressions*, page 89.

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 82. For more information on these directives, see *Simple rules*, page 87, and *CFI expressions*, page 89.

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 89). 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 81.

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.

Table 28: Unary operators in CFI expressions

Operator	Operand	Description
UMINUS	*cfiexpr*	Performs arithmetic negation on a CFI expression.

Table 28: Unary operators in CFI expressions (Continued)

Binary operators

Overall syntax: `OPERATOR(operand1,operand2)`

Operator	Operands	Description
ADD	*cfiexpr*,*cfiexpr*	Addition
AND	*cfiexpr*,*cfiexpr*	Bitwise AND
DIV	*cfiexpr*,*cfiexpr*	Division
EQ	*cfiexpr*,*cfiexpr*	Equal
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 29: Binary operators in CFI expressions

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*.
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 30: Ternary operators in CFI expressions

EXAMPLE

The following is a generic example and not an example specific to the H8/300H and H8S microcomputers. 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

Table 31: Code sample with backtrace rows and columns

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

Table 31: Code sample with backtrace rows and columns (Continued)

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.

Assembler 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 displayed on the screen, as well as printed in the optional list file.

All messages are issued as complete, self-explanatory messages. The message consists of the incorrect source line, with a pointer to where the problem was detected, followed by the source line number and the diagnostic message. If include files are used, error messages will be preceded by the source line number and the name of the *current* file:

```
        ADS     B,C
-----------^
"subfile.h",4  Error[40]: bad instruction
```

Severity levels

The diagnostic messages produced by the H8 IAR Assembler reflect problems or errors that are found in the source code or occur at assembly time.

OPTIONS FOR DIAGNOSTICS

There are two assembler options for diagnostics. You can:

- Disable or enable all warnings, ranges of warnings, or individual warnings, see-*w*, page 24
- Set the number of maximum errors before the compilation stops, see *-E*, page 17.

COMMAND LINE ERROR MESSAGES

Command line errors occur when the assembler is invoked with incorrect parameters. The most common situation is when a file cannot be opened, or with duplicate, misspelled, or missing command line options.

ASSEMBLY WARNING MESSAGES

Assembly warning messages are produced when the assembler has found a construct which is probably the result of a programming error or omission.

ASSEMBLY ERROR MESSAGES

Assembly error messages are produced when the assembler has found a construct which violates the language rules.

ASSEMBLY FATAL ERROR MESSAGES

Assembly fatal error messages are produced when the assembler has found a user error so severe that further processing is not considered meaningful. After the diagnostic message has been issued the assembly is immediately terminated. These error messages are identified as `Fatal` in the error messages list.

ASSEMBLER INTERNAL ERROR MESSAGES

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.

During assembly a number of internal consistency checks are performed and if any of these checks fail, the assembler will terminate after giving a short description of the problem. Such errors should normally not occur. However, if you should encounter an error of this type, it should be reported to your software distributor or to IAR Technical Support. Please include information enough to reproduce the problem. This would typically include:

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

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Symbols