M16C/R8C IAR Assembler

Reference Guide

for Renesas

M16C/1X–3X, 6X and R8C

Series of CPU Cores

COPYRIGHT NOTICE

DISCLAIMER

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

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

TRADEMARKS

IAR, IAR Embedded Workbench, IAR XAR Library Builder, IAR XLINK Linker, IAR XLIB Librarian, IAR MakeApp and IAR PreQual are trademarks owned by IAR Systems. C-SPY is a trademark registered in Sweden by IAR Systems. IAR visualSTATE is a registered trademark owned by IAR Systems.

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

EDITION NOTICE

Third edition: October 2004

Part number: AM16C-3

This guide applies to the M16C/R8C IAR Embedded Workbench™ version 3.x.

Contents

Tables

Preface

Welcome to the M16C/R8C IAR Assembler Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use the M16C/R8C IAR Assembler to best suit your application requirements.

Who should read this guide

You should read this guide if you plan to develop an application using assembler language for the M16C/R8C Series of CPU cores and need to get detailed reference information on how to use the M16C/R8C IAR Assembler. In addition, you should have working knowledge of the following:

- The architecture and instruction set of the M16C/R8C Series of CPU cores. Refer to the documentation from Renesas for information about the M16C/R8C Series of CPU cores
- 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 M16C/R8C IAR Assembler, you should read the *Introduction to the M16C/R8C IAR Assembler* chapter 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 toolkit, we recommend that you first read the initial chapters of the *IAR Embedded Workbench™ IDE User Guide*. They give product overviews, as well as tutorials that can help you get started.

What this guide contains

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

- *Introduction to the M16C/R8C 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 M16C/R8C Series of CPU cores is described in a series of guides. For information about:

- Using the IAR Embedded Workbench™ and the IAR C-SPY™ Debugger, refer to the *IAR Embedded Workbench™ IDE User Guide*
- Programming for the M16C/R8C IAR C/C++ Compiler, refer to the *M16C/R8C IAR C/C++ Compiler Reference Guide*
- Using the IAR XLINK Linker™, the IAR XAR Library Builder, and the IAR XLIB Librarian™, refer to the *IAR Linker and Library Tools Reference Guide*.
- Using the IAR DLIB Library functions, refer to the online help system
- Using the IAR CLIB Library functions, refer to the *IAR C Library Functions Reference Guide*, available from the online help system
- Using the runtime environment, refer to the *IAR Runtime Environment and Library User Guide*
- Porting application code and projects created with a previous M16C/R8C IAR Embedded Workbench IDE, refer to *M16C/R8C IAR Embedded Workbench Migration Guide*.

All of these guides are delivered in hypertext PDF 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.

Table 1: Typographic conventions used in this guide

Style	Used for
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
reference	A cross-reference within or to another part of this guide.
	Identifies instructions specific to the versions of the IAR Systems tools for the IAR Embedded Workbench interface.
	Identifies instructions specific to the command line versions of IAR Systems development tools.

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

Introduction to the M16C/R8C IAR Assembler

This chapter describes the source code format for the M16C/R8C IAR Assembler and provides programming hints.

Refer to Renesas's hardware documentation for syntax descriptions of the instruction mnemonics.

Source format

The format of an assembler source line is as follows:

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

where the components are as follows:

label	A label, which is assigned the value and type of the current program location counter (PLC). The `:` (colon) is optional if the label starts in the first column.
operation	An assembler instruction or directive. This must not start in the first column.
operands	An assembler instruction can have zero, one, or more operands. The data definition directives, for example `DC8`, can have any number of operands. For reference information about the data definition directives, see *Data definition or allocation directives*, page 73. Other assembler directives can have one, two, or three operands, separated by commas.
comment	Comment, preceded by a `;` (semicolon).

The fields can be 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 M16C/R8C IAR Assembler uses the default filename extensions `s34`, `asm`, and `msa` for source files.

Assembler expressions

Expressions can consist of 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, and range checking is only performed when 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 *Precedence of operators*, page 23.

The following operands are valid in an expression:

- User-defined symbols and labels.
- Constants, excluding floating-point constants.
- The program location counter (PLC) symbol, `$`.

These are described in greater detail in the following sections.

The valid operators are described in the chapter *Assembler operators*, page 23.

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.

USING SYMBOLS IN RELOCATABLE EXPRESSIONS

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

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

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

```
        .EXTERN third
        .RSEG DATA
first   .BLKB 5
second  .BLKB 3

        .RSEG CODE
start   …
```

Then in the segment CODE the following instructions are legal:

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

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

SYMBOLS

User-defined symbols can be up to 255 characters long, and all characters are significant.

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

For built-in symbols like instructions, registers, operators, and directives case is insignificant. 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 page 19 for additional information.

Notice that symbols and labels are byte addresses. For additional information, see *Generating lookup table*, page 74.

LABELS

Symbols used for memory locations are referred to as labels.

Program location counter (PLC)

The program location counter is called `$`. For example:

```
JMP    $       ; Loop forever
```

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'`
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 between zero and more 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).
""	Empty string (an ASCII null character).
\'	'
\\	\

Table 3: ASCII character constant formats

FLOATING-POINT CONSTANTS

The M16C/R8C IAR Assembler will accept floating-point values as constants and convert them into IEEE single-precision (signed 32-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.

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

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

PREDEFINED SYMBOLS

The M16C/R8C 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
__DATE__	Current date in dd/Mmm/yyyy format (string).
__FILE__	Current source filename (string).
__IAR_SYSTEMS_ASM__	IAR assembler identifier (number).
__LINE__	Current source line number (number).
__TID__	Target identity, consisting of two bytes (number). The high byte is the target identity, which is `28` for `AM16C`. The low byte is `00` for `AM16C`. Thus, the `__TID__` value is `0x1C00`.
__TIME__	Current time in `hh:mm:ss` format (string).
__VER__	Version number in integer format; for example, version 4.17 is returned as 417 (number).

Table 5: Predefined symbols

Notice that `__TID__` is related to the predefined symbol `__TID__` in the M16C/R8C IAR C/C++ Compiler. It is described in the *M16C/R8C IAR C/C++ Compiler Reference Guide*.

Including symbol values in code

To include a symbol value in the code, you use the symbol in one of the data definition directives.

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

```
timdat  BYTE     __TIME__,",",__DATE__,0  // time and date
        ...
        MOV      timdat,A0        ; load address of string
        JSR      printstring      ; routine to print string
```

Testing symbols for conditional assembly

To test a symbol at assembly time, use one of the conditional assembly directives. For example, to use some feature introduced in a specific version of the compiler, you would use the `__VER__` symbol as follows:

```
#if (__VER__ > 200)
...
...
#else
...
...
#endif
```

Register symbols

The following table shows the existing predefined register symbols:

Name	Address size	Description
R0L	8 bits	Data register, low part of R0
R0H	8 bits	Data register, high part of R0
R1L	8 bits	Data register, low part of R1
R1H	8 bits	Data register, high part of R1
R0	16 bits	Data register
R1	16 bits	Data register
R2	16 bits	Data register

Table 6: Predefined register symbols

Name	Address size	Description
R3	16 bits	Data register
A0	16 bits	Address register
A1	16 bits	Address register
FB	16 bits	Frame base register
SP	16 bits	Denotes either ISP or USP, depending on the state of the U flag of the FLG register
ISP	16 bits	Interrupt stack pointer
USP	16 bits	User stack pointer
SB	16 bits	Static base register
FLG	16 bits	Flag register
INTB	20 bits	Interrupt table register

Table 6: Predefined register symbols (Continued)

For some instructions you can combine R2 and R0, R3 and R1, or A1 and A0 to configure a 32-bit register (R2R0, R3R1, or A1A0, respectively).

Programming hints

This section gives hints on how to write efficient code for the M16C/R8C IAR Assembler. For information about projects including both assembler and C/C++ source files, see the *M16C/R8C IAR C/C++ Compiler Reference Guide*.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of M16C/R8C cores are included in the IAR product package, in the \m16c\inc directory. These header files define the processor-specific special function registers (SFRs) and interrupt vector numbers.

The header files are intended to be used also with the M16C/R8C IAR C/C++ Compiler, ICCM16C, and they are suitable to use as templates when creating new header files for other M16C/R8C cores.

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.

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, 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 `am16c` command:

```
am16c [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.s34`, use the following command to generate a list file to the default filename (`power2.lst`):

```
am16c 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`:

```
am16c 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`:

```
am16c power2 -Llist\
```

Note: The subdirectory you specify must already exist. The trailing backslash is required because the parameter is prepended to 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:

```
am16c -f extend.xcl
```

Error return codes

When using the M16C/R8C 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 7: Assembler error return codes

ASSEMBLER ENVIRONMENT VARIABLES

Options can also be specified using the `ASMM16C` 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 M16C/R8C IAR Assembler:

Environment variable	Description
`ASMM16C`	Specifies command line options; for example: `set ASMM16C=-L -ws`
`AM16C_INC`	Specifies directories to search for include files; for example: `set AM16C_INC=c:\myinc\`

Table 8: Assembler environment variables

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

```
ASMM16C=-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
`-b`	Makes a library module
`-c{DMEAO}`	Conditional list
-D*symbol*[=*value*]	Defines a symbol
`-d`	Disables `#ifdef` check
-E*number*	Maximum number of errors
-f *filename*	Extends the command line
`-G`	Opens standard input as source
-I*prefix*	Includes paths
`-i`	#included text
-L[*prefix*]	Lists to prefixed source name
-l *filename*	Lists to named file
-M*ab*	Macro quote characters
`-N`	Omit header from assembler listing
-O*prefix*	Sets object filename prefix
-o *filename*	Sets object filename
-p*lines*	Lines/page
`-r`	Generates debug information
`-S`	Set silent operation
`-s{+|-}`	Case sensitive user symbols
`-T`	Lists active lines only
-t*n*	Tab spacing
-U*symbol*	Undefines a symbol
-w[*string*][s]	Disables warnings
`-x{DI2}`	Includes cross-references

Table 9: Assembler options summary

Descriptions of assembler options

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

-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 16.

This option is identical to the **Macro execution info** option in the **Assembler** category in the IAR Embedded Workbench.

-b

`-b`

This option causes the object file to be a library module rather than a program module.

By default, the assembler produces a program module ready to be linked with the IAR XLINK Linker. Use the `-b` option if you instead want the assembler to make a library module for use with XLIB.

If the `NAME` directive is used in the source (to specify the name of the program module), the `-b` option is ignored, i.e. the assembler produces a program module regardless of the `-b` option.

This option is identical to the **Make a LIBRARY module** option in the **Assembler** category in the IAR Embedded Workbench.

-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 16 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
`-cA`	Assembled lines only
`-cO`	Multiline code

Table 10: Conditional list (-c)

This option is related to the options on the **List** page in the **Assembler** category in the IAR Embedded Workbench.

`-D` `D`*`symbol`*`[=`*`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: `am16c prog`
Test version: `am16c 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:

```
am16c prog -DFRAMERATE=3
```

This option is identical to the **#define** option in the **Assembler** category in the IAR Embedded Workbench.

`-d` `-d`

By default, `#ifdef` is checked against `#else` and `#endif` at the end of a module. You can use the `-d` option to disable the test. This will then allow programs like:

```
#define    FOO
#ifdef     FOO
           .MODULE m1
           .NOP
           .ENDMOD
#endif
           .MODULE m2
           .NOP
           .END
```

The `-d` option is identical to the **Disable #ifdef/#endif matching** option in the **Assembler** category in the IAR Embedded Workbench.

`-E` `-E`*`number`*

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

This option is identical to the **Max number of errors** option in the **Assembler** category in the IAR Embedded Workbench.

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

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

The `-f` option is particularly useful where there is a large number of options which are more conveniently placed in a file than on the command line itself.

Example

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

```
am16c prog -f extend.xcl
```

-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.

-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 `AM16C_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\` provided that the `AM16C_INC` environment variable is set.

This option is related to the **Include** option in the **Assembler** category in the IAR Embedded Workbench.

-i

`-i`

Includes `#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.

This option is related to the **Include** option in the **Assembler** category in the IAR Embedded Workbench.

-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 must not 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`:

```
am16c prog -Llist\
```

This option is related to the **List** options in the **Assembler** category in the IAR Embedded Workbench.

-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.

This option is related to the **List** options in the **Assembler** category in the IAR Embedded Workbench.

-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:

```
am16c filename -M'<>'
```

This option is identical to the **Macro quote chars** option in the **Assembler** category in the IAR Embedded Workbench.

`-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 16 for additional information.

This option is related to the **Include headers** option in the **Assembler** category in the IAR Embedded Workbench.

`-O` `-O`*`prefix`*

Use this option to set the prefix to be used on the name of the object file. Notice that you must not 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.r34` rather than to the default file `prog.r34`:

```
am16c prog -Oobj\
```

This option is related to the **Output directories** option in the **General Options** category in the IAR Embedded Workbench.

`-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, `r34` 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.r34` instead of the default `prog.r34`:

```
am16c prog -o obj
```

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

This option is related to the filename and directory that you specify when creating a new source file or project in the IAR Embedded Workbench.

`-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 16 for additional information.

This option is identical to the **Lines/page** option in the **Assembler** category in the IAR Embedded Workbench.

`-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.

This option is identical to the **Generate debug information** option in the **Assembler** category in the IAR Embedded Workbench.

-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.

-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 11: 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.

This option is identical to the **Case sensitive user symbols** option in the **Assembler** category in the IAR Embedded Workbench.

-T -T

Causes a listing to include only active lines, for example not those in false #if blocks. By default, all lines are listed.

This option is useful for reducing the size of listings by eliminating lines that do not generate or affect code.

The -T option is identical to the **Active lines only** option in the **Assembler** category in the IAR Embedded Workbench.

-t -t*n*

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 16 for additional information.

This option is identical to the **Tab spacing** option in the **Assembler** category in the IAR Embedded Workbench.

-U `-U`*`symbol`*

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

By default, the assembler provides certain predefined symbols; see *Predefined symbols*, page 5. 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:

```
am16c prog -U __TIME__
```

This option is identical to the **#undef** option in the **Assembler** category in the IAR Embedded Workbench.

-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 89, 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`*.
`-w+`*`m-n`*	Enables warnings *`m`* to *`n`*.
`-w-m-n`	Disables warnings *`m`* to *`n`*.

Table 12: Disabling assembler warnings (-w)

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:

```
am16c prog -w-0
```

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

```
am16c prog -w-0-8
```

This option is identical to the **Warnings** option in the **Assembler** category in the IAR Embedded Workbench.

`-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 16 for additional information.

The following parameters are available:

Command line option	Description
`-xD`	#defines
`-xI`	Internal symbols
`-x2`	Dual line spacing

Table 13: Including cross-references in assembler list file (-x)

This option is identical to the **Include cross reference** option in the **Assembler** category in the IAR Embedded Workbench.

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, i.e. first evaluated) to 7 (the lowest precedence, i.e. last evaluated).

The following rules determine how expressions are evaluated:

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

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

Summary of assembler operators

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

UNARY OPERATORS – 1

`–`	Unary minus.
`! (NOT)`	Logical NOT.
`~ (BINNOT)`	Bitwise NOT.
`LOW`	Low byte.
`HIGH`	High byte.
`BYTE2`	Second 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 – 2

`*`	Multiplication.
`/`	Division.
`% (MOD)`	Modulo.

ADDITIVE ARITHMETIC OPERATORS – 3

`+`	Addition.
`–`	Subtraction.

SHIFT OPERATORS – 4

`>> (SHR)`	Logical shift right.
`<< (SHL)`	Logical shift left.

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 *Assembler expressions*, page 2, for related information.

`*` Multiplication (2).

`*` 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
```

`+` Addition (3).

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.

– Subtraction (3).

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 (2).

/ 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).

< evaluates to 1 (true) if the left operand has a lower numeric value than the right operand.

Example

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

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

<= evaluates to 1 (true) if the left operand has a lower or equal numeric value to the right operand.

Example

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

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

<> 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).

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

Example

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

`> (GT)` Greater than (7).

> evaluates to 1 (true) if the left operand has a higher numeric value than the right operand.

Example

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

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

`>=` evaluates to 1 (true) if the left operand is equal to or has a higher numeric value than the right operand.

Example

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

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

Use `&&` to perform logical AND between its two integer operands. If both operands are non-zero the result is 1; otherwise it is zero.

Example

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

`& (BINAND)` Bitwise AND (5).

Use `&` to perform bitwise AND between the integer operands.

Example

```
B'1010 & B'0011 → B'0010
B'1010 & B'0101 → B'0000
B'1010 & B'0000 → B'0OOO
```

`~ (BINNOT)` Bitwise NOT (1).

Use ~ to perform bitwise NOT on its operand.

Example

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

| (BINOR) Bitwise OR (6).

Use | to perform bitwise OR on its operands.

Example

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

^ (BINXOR) Bitwise exclusive OR (6).

Use ^ to perform bitwise XOR on its operands.

Example

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

% (MOD) Modulo (2).

`%` 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 `!` to negate a logical argument.

Example

```
! B'0101 → 0
! B'0000 → 1
```

|| (OR) Logical OR (6).

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

Example

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

BYTE2 Second byte (1).

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

Example

```
BYTE2 0x12345678 → 0x56
```

BYTE3 Third byte (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 5, DATE 4, DATE 3
```

`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 (4).

Use << 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 (4).

Use >> 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 segment `CODE`.

UGT Unsigned greater than (7).

`UGT` evaluates to 1 (true) if the left operand has a larger value than the right operand. The operation treats its operands as unsigned values.

Example

`2 UGT 1 → 1`

`-1 UGT 1 → 1`

ULT Unsigned less than (7).

`ULT` evaluates to 1 (true) if the left operand has a smaller value than the right operand. The operation treats its operands as unsigned values.

Example

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

XOR Logical exclusive OR (6).

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. It then describes the syntax conventions and provides detailed reference information for each category of directives.

Summary of directives

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

Directive	Description	Section
`$`	Includes a file.	Assembler control
`#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
`#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
`ADDR`	Generates 24-bit constants.	Data definition or allocation
`ALIAS`	Assigns a permanent value local to a module.	Value assignment
`ALIGN`	Aligns the location counter by inserting zero-filled bytes.	Segment control
`ALIGNRAM`		Segment control
`ASEG`	Begins an absolute segment.	Segment control

Table 14: Assembler directives summary

Directive	Description	Section
ASEGN	Begins a named absolute segment	Segment control
ASSIGN	Assigns a temporary value.	Value assignment
BLKA	Allocates space for 24-bit data objects.	Data definition or allocation
BLKB	Allocates space for 8-bit data objects.	Data definition or allocation
BLKD	Allocates space for 64-bit data objects.	Data definition or allocation
BLKF	Allocates space for 32-bit data objects.	Data definition or allocation
BLKL	Allocates space for 32-bit data objects.	Data definition or allocation
BLKW	Allocates space for 16-bit data objects.	Data definition or allocation
BYTE	Generates 8-bit byte constants, including strings.	Data definition or allocation
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
DC8	Generates 8-bit byte constants, including strings.	Data definition or allocation
DC16	Generates 16-bit constants.	Data definition or allocation
DC24	Generates 24-bit constants.	Data definition or allocation
DC32	Generates 32-bit constants.	Data definition or allocation
DEFINE	Defines a file-wide value.	Value assignment
DF32	Generates 32-bit floating point constants.	Data definition or allocation
DS8	Allocates space for 8-bit data objects.	Data definition or allocation

Table 14: Assembler directives summary (Continued)

Directive	Description	Section
DS16	Allocates space for 16-bit data objects.	Data definition or allocation
DS24	Allocates space for 24-bit data objects.	Data definition or allocation
DS32	Allocates space for 32-bit data objects.	Data definition or allocation
ELSE	Assembles instructions if a condition is false.	Conditional assembly
END	Terminates the assembly of the last module in a file.	Module control
ENDIF	Ends an IF block.	Conditional assembly
ENDM	Ends a macro definition.	Macro processing
ENDMOD	Terminates the assembly of the current module.	Module control
ENDR	Ends a repeat structure.	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment
EVEN	Aligns the program counter to an even address.	Segment control
EXITM	Exits prematurely from a macro.	Macro processing
EXTERN	Imports an external symbol.	Symbol control
FLOAT	Generates 32-bit floating point constants.	Data definition or allocation
IF	Assembles instructions if a condition is true.	Conditional assembly
LIBRARY	Begins a library module.	Module control
LIMIT	Checks a value against limits.	Value assignment
LOCAL	Creates symbols local to a macro.	Macro processing
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

Table 14: Assembler directives summary (Continued)

Directive	Description	Section
LWORD	Generates 32-bit constants.	Data definition or allocation
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
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
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	Marks a symbol as required.	Symbol control
RSEG	Begins a relocatable segment.	Segment control
RTMODEL	Declares runtime model attributes.	Module control
SET	Assigns a temporary value.	Value assignment
sfr	Creates byte-access SFR labels.	Value assignment
sfrp	Creates word-access SFR labels.	Value assignment
SFRTYPE	Specifies SFR attributes.	Value assignment
STACK	Begins a stack segment.	Segment control
WORD	Generates 16-bit constants.	Data definition or allocation

Table 14: Assembler directives summary (Continued)

Note: The IAR Systems toolkit for the M16C/R8C Series of CPU cores also supports static overlay directives—FUNCALL, FUNCTION, LOCFRAME, and ARGFRAME—that are designed to ease coexistence of routines written in C and assembler language. These directives are described in the *M16C/R8C IAR C/C++ Compiler Reference Guide*. (Static overlay is not, however, relevant for this product.)

Syntax conventions

In the syntax definitions the following conventions are used:

- Parameters, representing what you would type, are shown in italics. So, for example, in:

  `ORG` *`expr`*

  `expr` represents an arbitrary expression.

- Optional parameters are shown in square brackets. So, for example, in:

  `END [`*`expr`*`]`

  the *`expr`* parameter is optional. An ellipsis indicates that the previous item can be repeated an arbitrary number of times. For example:

  `PUBLIC` *`symbol`* `[,`*`symbol`*`] …`

  indicates that `PUBLIC` can be followed by one or more symbols, separated by commas.

- Alternatives are enclosed in `{` and `}` brackets, separated by a vertical bar, for example:

  `LSTOUT{+|-}`

  indicates that the directive must be followed by either `+` or `-`.

LABELS AND COMMENTS

Where a label *must* precede a directive, this is indicated in the syntax, as in:

`label` `SET` *`expr`*

An optional label, which will assume the value and type of the current program location counter (PLC), can precede all directives. For clarity, this is not included in each syntax definition.

In addition, unless explicitly specified, all directives can be followed by a comment, preceded by `;` (semicolon).

PARAMETERS

The following table shows the correct form of the most commonly used types of parameter:

Parameter	What it consists of
`expr`	An expression; see *Assembler expressions*, page 2.

Table 15: Assembler directive parameters

Parameter	What it consists of
label	A symbolic label.
symbol	An assembler symbol.

Table 15: Assembler directive parameters (Continued)

DIRECTIVE FORMAT

Almost all directives can be written either as `MODULE` or as `.MODULE`. For the sake of convenience, they are referred to without the prefixed period (`.`) in the remainder of this chapter. Only the following directives cannot be written with a prefixed period: `#define`, `#elif`, `#else`, `#endif`, `#error`, `#if`, `#ifdef`, `#ifndef`, `#include`, `#message`, `#undef`.

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

PARAMETERS

expr	Optional expression (0–255) used by the IAR compiler to encode programming language, memory model, and processor configuration.
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 and XLIB when processing object files.
value	A text string specifying the value.

DESCRIPTION

Beginning a program module

Use `NAME` 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` to create libraries containing lots 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 underscore. 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 code, and you want to control the module consistency, refer to the *M16C/R8C IAR C/C++ Compiler Reference Guide*.

Examples

The following example defines three modules where:

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

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

MODULE MOD_2
```

```
  RTMODEL   "foo", "2"
  RTMODEL   "bar", "*"
  ...
ENDMOD

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

Symbol control directives

These directives control how symbols are shared between modules.

Directive	Description
EXTERN	Imports an external symbol.
PUBLIC	Exports symbols to other modules.
REQUIRE	Marks a symbol as referenced.

Table 17: Symbol control directives

SYNTAX

```
EXTERN symbol [,symbol] …
PUBLIC symbol [,symbol] …
REQUIRE symbol [,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. The symbols declared as PUBLIC can only be assigned values by using them as labels. Symbols declared PUBLIC can be relocated 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.

Importing symbols

Use EXTERN to import an untyped external symbol.

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

EXAMPLES

The following example defines a subroutine to print an error message, and exports the entry address err so that it can be called from other modules. It defines print as an external routine; the address will be resolved at link time.

```
     NAME   error
     EXTERN print
     PUBLIC err

err  RCALL  print
     DB     "** Error **"
     EVEN
     RET

     END
```

Segment control directives

The segment directives control how code and data are generated.

Directive	Description
ALIGN	Aligns the location counter by inserting zero-filled bytes.
ALIGNRAM	Aligns the location counter without inserting any bytes.
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.

Table 18: Segment control directives

Directive	Description
STACK	Begins a stack segment.

Table 18: Segment control directives (Continued)

SYNTAX

```
ALIGN align [,value2]
ALIGNRAM align [,value]
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 20. For example, `align 1` results in word alignment `2`.

align2 Exponent of the value to which the address should be aligned, in the range 0 to 8. For example, `align 1` results in word alignment `2`.

expr Address to set the location counter to.

flag `NOROOT`
This segment part may be discarded by the linker even 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`
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`
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 counter (PC)

Use ORG to set the program counter of the current segment to the value of an expression. The optional label will assume the value and type of the new 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 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 and ALIGNRAM aligns without inserting any 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.

EXAMPLES

Beginning an absolute segment

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

```
        EXTERN  undef, overflow, brk, add
        EXTERN  single, watch, DBC
```

```
        ASEG
        ORG     0FFFDCh
        LWORD   undef
        LWORD   overflow
        LWORD   brk
        LWORD   add
        LWORD   single
        LWORD   watch
        LWORD   DBC
        LWORD   NMI
        LWORD   reset

        ORG     0F0000h
reset   MOV.W   #0,R0    ;Start of the main program
        ;....

NMI     ;.....  Start of NMI routine

        END
```

Beginning a relocatable segment

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

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

```
        EXTERN  divrtn,mulrtn

        RSEG    table
        WORD    divrtn,mulrtn

        ORG     $+6
        WORD    subrtn

        RSEG    code
subrtn  MOV.W   R2,R0
        SUB.W   R3,R0
```

Beginning a stack segment

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

```
         STACK    rpnstack
parms    DS8      100
opers    DS8      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    DS24     1
         ENDMOD

         NAME     common2
         COMMON   data
up       DS8      1
         ORG      $+2
down     DS8      1
         END
```

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

Aligning a segment

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

```
         RSEG     data      ; Start a relocatable 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
                            ; and create a 64-byte table
results  DS8      64        ; 64 bytes
         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`	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.
`SET`	Assigns a temporary value.
`sfr`	Creates byte-access SFR labels.
`sfrp`	Creates word-access SFR labels.
`SFRTYPE`	Specifies SFR attributes.

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
[const] sfr register = value
[const] sfrp register = value
[const] SFRTYPE register attribute [,attribute] = value
```

PARAMETERS

attribute One or more of the following:

`BYTE`	The SFR must be accessed as a byte.
`READ`	You can read from this SFR.
`WORD`	The SFR must be accessed as a word.
`WRITE`	You can write to this SFR.

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*.
register	The special function register.
value	The SFR port address.

DESCRIPTION

Defining a temporary value

Use either of ASSIGN and SET to define a symbol that may be redefined, such as for use with macro variables. Symbols defined with SET 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.

Defining special function registers

Use sfr to create special function register labels with attributes READ, WRITE, and BYTE turned on. Use sfrp to create special function register labels with attributes READ, WRITE, or WORD turned on. Use SFRTYPE to create special function register labels with specified attributes.

Prefix the directive with const to disable the WRITE attribute assigned to the SFR. You will then get an error or warning message when trying to write to the SFR.

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 `cons` in a `REPT` loop to generate a table of the first 8 powers of 3:

```
         NAME     table
cons     SET      1

repeat   MACRO    times
         DC16     cons
cons     SET      cons*3
         IF       times>1
         repeat   times-1
         ENDIF
         ENDM

main     repeat   4
         END      main
```

It generates the following code:

```
     1    000000
     2    000000                  NAME    table
     3    000001          cons    SET     1
     4    000000
    12    000000
    13    000000          main    repeat  4
    13.1  000000 0100             DC16    cons
    13.2  000003          cons    SET     cons*3
    13.3  000002                  IF      4>1
    13    000002                  repeat  4-1
    13.1  000002 0300             DC16    cons
    13.2  000009          cons    SET     cons*3
    13.3  000004                  IF      4-1>1
    13    000004                  repeat  4-1-1
    13.1  000004 0900             DC16    cons
```

```
13.2  00001B         cons    SET     cons*3
13.3  000006                 IF      4-1-1>1
13    000006                 repeat  4-1-1-1
13.1  000006 1B00            DC16    cons
13.2  000051         cons    SET     cons*3
13.3  000008                 IF      4-1-1-1>1
13.4  000008                 repeat  4-1-1-1-1
13.5  000008                 ENDIF
13.6  000008                 ENDM
13.7  000008                 ENDIF
13.8  000008                 ENDM
13.9  000008                 ENDIF
13.10 000008                 ENDM
13.11 000008                 ENDIF
13.12 000008                 ENDM
14    000008                 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.W  locn,R0
         ADD.W  #value,R0
         RTS
         ENDMOD

         NAME   add2
value    EQU    88
         MOV.W  locn,R0
         ADD.W  #value,R0
         RTS
         END
```

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

Using special function registers

In this example a number of I/O ports are defined for a particular application. Although the CPU can handle them as bi-directional ports, their functionality is being restricted to allow the assembler to report inappropriate accesses.

BiPort is a byte-wide port which can be read and written to. InPort can only be read. Bothports allows both BiPort and InPort to be read simultaneously and OutPort can only be written to.

The definitions are followed by sample code performing legal actions:

```
        sfr      BiPort      = 0x3E0          ; Port P0
const   sfr      InPort      = 0x3E1          ; Port P1
const   sfrp     BothPorts   = 0x3E0          ; Ports P0&P1
        SFRTYPE  OutPort WRITE,BYTE = 0x3E4   ; Port P2
```

Based on these definitions the following accesses will be allowed:

```
        MOV.B    InPort,R0L
        MOV.B    R0L,OutPort
        MOV.B    BiPort,R0L
        MOV.B    R0L,BiPort
        MOV.W    BothPorts,R0
```

These ones will cause errors:

```
        MOV.B    R0L,InPort       ; Cannot write to InPort
        MOV.B    OutPort,R0L      ; Cannot read OutPort
        MOV.W    BiPort,R0        ; BiPort byte access only
        MOV.B    BothPorts,R0L    ; BothPorts word access
```

Using the LIMIT directive

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

```
speed     SET        50
LIMIT     speed,10,30,...speed out of range...

          END
```

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.

Table 20: Conditional assembly directives

Directive	Description
ENDIF	Ends an IF block.

Table 20: Conditional assembly directives (Continued)

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 adds a constant to a register a:

```
addd     MACRO    a,c
         IF       a=1
         INC.B    c
         ELSE
         ADD.B    #a,c
         ENDIF
         ENDM
```

It can be tested with the following program:

```
main     MOV.B    #17,R0L
         addd     2,R0L
         MOV.B    #22,R0L
         addd     1,R0L
         RTS
         END
```

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.
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	String to be substituted.
argument	A symbolic argument name.
expr	An expression.
formal	Argument into which each character of *actual* (REPTC) or each *actual* (REPTI) is substituted.
name	The name of the macro.
symbol	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.

Insert the target-specific file macro.fm here:

For example, you could define a macro ERROR as follows:

```
errmac  MACRO   text
        JSR     abort
        BYTE    text,0
        ENDM
```

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

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

The assembler will expand this to:

```
        JSR     abort
        BYTE    'Disk not ready',0
```

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

The previous example could therefore be written as follows:

```
errmac  MACRO
        JSR     abort
        BYTE    \1,0
        ENDM
```

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

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

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

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

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

Passing special characters

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

Import the target-specific file macroqch.fm here:

For example:

```
macld    MACRO  op
         MOV    op
         ENDM
```

The macro can be called using the macro quote characters:

```
         macld  <#1,R0>
         END
```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 16.

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:

```
chcount MACRO   parm
        IF      _args>10
        EXITM
        ENDIF
        BYTE    _args
        ENDM
        END
```

How macros are processed

There are three distinct phases in the macro process:

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

Repeating statements

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

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

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

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

EXAMPLES

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

Coding in-line for efficiency

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

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

```
        NAME    play
        sfr     IO_port=0x3E0
        RSEG    data
buffer  BYTE    512     \\buffer
        RSEG    code
play    MOV.W   #buffer,A0
loop    MOV.B   [A0],IO_port
        INC.W   A0
        CMP.W   #buffer+512,A0
        JNE     loop
        RTS
        END
```

The main program calls this routine as follows:

```
        JSR     play
```

For efficiency we can recode this as the following macro which takes the buffer as a parameter:

```
        NAME    play
        sfr     IO_port=0x3E0
play    MACRO   buf
        LOCAL   loop
        MOV.W   #buf,A0
loop    MOV.B   [A0],IO_port
        INC.W   A0
        CMP.W   #buf+512,A0
        JNE     loop
        ENDM
        END
```

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

Using REPTC and REPTI

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

```
         NAME   retc

         EXTERN plotc

banner   REPTC  chr, "Welcome"
         LDI    #'chr',R0L
         JSR    plotc
         ENDR

         END
```

This produces the following code:

```
     1    000000                  NAME retc
     2    000000
     3    000000                  EXTERN plotc
     4    000000
     5    000000           banner REPTC  chr, "Welcome"
     6    000000                  MOV.B  #'chr',R0L
     7    000000                  JSR    plotc
     8    000000                  ENDR
     8.1  000000 74C057           MOV.B  #'W',R0L
     8.2  000003 FD......         JSR    plotc
     8.3  000007 74C065           MOV.B  #'e',R0L
     8.4  00000A FD......         JSR    plotc
     8.5  00000E 74C06C           MOV.B  #'l',R0L
     8.6  000011 FD......         JSR    plotc
     8.7  000015 74C063           MOV.B  #'c',R0L
     8.8  000018 FD......         JSR    plotc
     8.9  00001C 74C06F           MOV.B  #'o',R0L
     8.10 00001F FD......         JSR    plotc
     8.11 000023 74C06D           MOV.B  #'m',R0L
     8.12 000026 FD......         JSR    plotc
     8.13 00002A 74C065           MOV.B  #'e',R0L
     8.14 00002D FD......         JSR    plotc
     9    000031
    10    000031                  END
```

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

```
        NAME    retc
        EXTERN  base,count,init
banner  REPTI   adds,base,count,init
```

```
MOV.W   #0,adds
ENDR
END
```

This produces the following code:

```
1    000000                     NAME    retc
2    000000                     EXTERN  base,count,init
3    000000             banner  REPTI   adds,base,count,init
4    000000                     MOV.W   #0,adds
5    000000                     ENDR
5.1  000000 D90F....            MOV.W   #0,base
5.2  000004 D90F....            MOV.W   #0,count
5.3  000008 D90F....            MOV.W   #0,init
6    00000C                     END
```

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, ELSE, or END.

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    lstcndtst
        EXTERN  print

        RSEG    prom
debug   SET     0

begin   IF      debug
        JSR     print
        ENDIF
```

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

This will generate the following listing:

```
 3   000000                       NAME    lstcndtst
 4   000000                       EXTERN  print
 5   000000
 6   000000                       RSEG    prom
 7   000000               debug   SET     0
 8   000000
 9   000000               begin   IF      debug
10   000000                       JSR     print
11   000000                       ENDIF
12   000000
13   000000                       LSTCND+
14   000000               begin2  IF      debug
16   000000                       ENDIF
17   000000                       END
```

The following example shows the effect of `LSTCOD+` on the generated code:

```
 1   000000                   NAME    lstcodtest
 2   000000                   EXTERN  print
 3   000000
 4   000000                   RSEG    tables
 5   000000
 6   000000 01000000*table1:  DC32    1,10,100,1000,10000
 7   000014
 8   000014                   LSTCOD+
 9   000014 01000000 table2:  DC32    1,10,100,1000,10000
            0A000000
            64000000
            E8030000
            10270000
10   000028
11   000028                   END
```

Controlling the listing of macros

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

```
dec2    MACRO   arg
        DEC.B   arg
        DEC.B   arg
```

```
        ENDM

        LSTMAC+
inc2    MACRO   arg
        INC.B   arg
        INC.B   arg
        ENDM

        EXTERN  memloc
begin   dec2    memloc

        LSTEXP-
        inc2    memloc
        RTS
        END     begin
```

This will produce the following output:

```
   5    000000
   6    000000                    LSTMAC+
   7    000000          inc2      MACRO   arg
   8    000000                    INC.B   arg
   9    000000                    INC.B   arg
  10    000000                    ENDM
  11    000000
  12    000000                    EXTERN  memloc
  13    000000          begin     dec2    memloc
  13.1  000000 AF....             DEC.B   memloc
  13.2  000003 AF....             DEC.B   memloc
  13.3  000006                    ENDM
  14    000006
  15    000006                    LSTEXP-
  16    000006                    inc2    memloc
  17    00000C F3                 RTS
  18    00000D
  19    00000D                    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 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.
`#message`	Generates a message on standard output.
`#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 SET 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.

Defining comments

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

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

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

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

```
#define five 5 ; comment

MOV.W  five+addr,R0
; Syntax error!
; Expands to "MOV.W 5 ; comment+addr,R0"

MOV.W  R1,five+addr
; Incorrect code!
; Expands to "MOV.W R1,5 ; comment+addr"
```

EXAMPLES

Using conditional directives

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

```
        NAME    ifedf
        EXTERN  input,output

#define adjust  10

main    MOV.W   input,A0
        MOV.W   [A0],R0

#ifdef  adjust
        ADD.W   adjust,R0

#else
#error  "'adjust' not defined"
#endif

#undef  adjust
        MOV.W [A0],R0
        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.s34`:

```
xch     MACRO   a,b
        PUSH.W  a
        MOV.W   b,a
        POP.W   b
        ENDM
```

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

```
        NAME    include
        EXTERN  result1, result2
; standard macro definitions
#include "macros.s34"

; program
```

```
main    xch     result1, result2
        RTS
        END     main
```

Data definition or allocation directives

These directives define temporary values or reserve memory. The column *Alias* in the following table shows the Renesas directive that corresponds to the IAR Systems directive:

Directive	Alias	Description	Expression restrictions
DC8	BYTE	Generates 8-bit constants, including strings.	
DC16	WORD	Generates 16-bit constants.	
DC24	ADDR	Generates 24-bit constants.	
DC32	LWORD	Generates 32-bit constants.	
DF32	FLOAT	Generates 32-bit floating-point constants.	
DS8	BLKB	Allocates space for 8-bit data objects.	No external references Absolute
DS16	BLKW	Allocates space for 16-bit data objects.	No external references Absolute
DS24	BLKA	Allocates space for 24-bit data objects.	No external references Absolute
DS32	BLKL, BLKF	Allocates space for 32-bit data objects.	No external references Absolute
DS64	BLKD	Allocates space for 64-bit data objects.	No external references Absolute

Table 24: Data definition or allocation directives

SYNTAX

```
BLKA expr [,expr] ...
BLKB expr [,expr] ...
BLKD expr [,expr] ...
BLKF expr [,expr] ...
BLKL expr [,expr] ...
BLKW expr [,expr] ...
DC8 expr [,expr] ...
DC16 expr [,expr] ...
DC24 expr [,expr] ...
DC32 expr [,expr] ...
```

```
DF32 expr [,expr] ...
DS8 expr [,expr] ...
DS16 expr [,expr] ...
DS24 expr [,expr] ...
DS32 expr [,expr] ...
DS64 expr [,expr] ...
```

PARAMETERS

expr	A valid absolute, relocatable, or external expression, or an ASCII string. ASCII strings will be zero filled to a multiple of the data size implied by the directive. Double-quoted strings will be zero-terminated.
value	A valid absolute expression or 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 unitialized memory
8-bit integers	DC8, BYTE	DS8, BLKB
16-bit integers	DC16, WORD	DS16, BLKW
24-bit integers	DC24, ADDR	DS24, BLKA
32-bit integers	DC32, LWORD	DS32, BLKL
32-bit floats	DF32, FLOAT	DS32, BLKF
64-bit floats	–	DS64, BLKD

Table 25: Using data definition or allocation directives

EXAMPLES

Generating lookup table

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

```
        NAME    table
table   WORD    addsubr,subsubr,clrsubr
addsubr ADD.W   R0,R1
        RTS

subsubr SUB.W   R0,R1
        RTS
```

```
clrsubr MOV.W   #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
`$`	Includes a file.
`/*comment*/`	C-style comment delimiter.
`//`	C++ style comment delimiter.
`CASEOFF`	Disables case sensitivity.
`CASEON`	Enables case sensitivity.
`RADIX`	Sets the default base.

Table 26: Assembler control directives

SYNTAX

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

PARAMETERS

comment	Comment ignored by the assembler.
expr	Default base; default 10 (decimal).
filename	Name of file to be included. The `$` character must be the first character on the line.

DESCRIPTION

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

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 use in conversion of constants from ASCII source to the internal binary format.

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

Including a source file

The following example uses `$` to include a file defining macros into the source file. For example, the following macros could be defined in `mymacros.s34`:

```
xch      MACRO    a,b
         PUSH.B   a
         MOV.B    a,b
         POP.B    b
         ENDM
```

The macro definitions can be included with a `$` directive, as in:

```
         NAME     include

; standard macro definitions

$mymacros.s34

; program
```

```
        EXTERN  var1,var2
main    xch     var1,var2
        RTS
        END     main
```

Defining comments

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

```
/*
Program to read serial input.
Version 3: 19.12.02
Author: mjp
*/
```

Changing the base

To set the default base to 16:

```
        RADIX  D'16
        MOV    12,A
```

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

To change the base from 16 to 10, *expr* must be written in hexadecimal format, for example:

```
RADIX  0x0A
```

Controlling case sensitivity

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

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

The following will generate a duplicate label error:

```
        CASEOFF

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

        END
```

Call frame information directives

These directives allow backtrace information to be defined.

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

Table 27: Call frame information directives

SYNTAX

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

Names block directives

```
CFI NAMES name
CFI ENDNAMES name
CFI RESOURCE resource : bits [, resource : bits] …
CFI VIRTUALRESOURCE resource : bits [, resource : bits] …
CFI RESOURCEPARTS resource part, part [, part] …
CFI STACKFRAME cfa resource type [, cfa resource type] …
CFI STATICOVERLAYFRAME cfa segment [, cfa segment] …
CFI BASEADDRESS cfa type [, cfa type] …
```

Extended names block directives

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

Common block directives

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

PARAMETERS

align	The power of two to which the address should be aligned. The allowed range for align is 0 to 31. As an example, the value 1 results in alignment on even addresses since 2^1 equals 2. The default align value is 0, for both `CODE` and `DATA` segments.
bits	The size of the resource in bits.
bytes	The size of the CFA in bytes. A constant value or an assembler expression that can be evaluated to a constant value.
cell	The name of a frame cell.
cfa	The name of a CFA (canonical frame address).
cfiexpr	A CFI expression (see *CFI expressions*, page 84).
column	A CFA name, a return address, or the name of a previously declared resource.
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.
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.
resource	The name of a resource.
segment	The name of the 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.

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 to define the *backtrace information* for the instructions in a program. The compiler normally generates this information, but for library functions and other code written purely in assembler language, backtrace information has to be added if you want to use the call frame stack in the debugger.

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

Backtrace rows and columns

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

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

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

Defining a names block

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

Start and end a names block with the directives:

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

where `name` is the name of the block.

Only one names block can be open at a time.

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

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

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

  The parameters are the name of the resource and the size of the resource in bits. A virtual resource is a logical concept, in contrast to a “physical” resource such as a processor register. 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 a composite resource, use the directive:

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

  The parts are separated with commas. The 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 (such as the stack pointer), and the segment type (to get the address width). 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 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, the base address CFA is not restored.

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/EC++ 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 to declare 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.

End a common block with the directive:

```
CFI ENDCOMMON name
```

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

Declare the initial value of a CFA or a resource by using the directives listed last in *Common block directives*, page 79.

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. Similarly to 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 function. The block starts when the function starts and ends when the function ends. Since any function consist of a consecutive sequence of instructions inside one segment, the data block will start and end within the same segment. For this reason, 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.

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 79.

CFI EXPRESSIONS

Call Frame Information expressions (CFI expressions) are used to define how the contents of columns are changed by the execution of an instruction.

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

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

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

Unary operators

Overall syntax: *OPERATOR*(*operand*)

Operator	Operand	Description
UMINUS	*cfiexpr*	Performs arithmetic negation on a CFI expression.
NOT	*cfiexpr*	Negates a logical CFI expression.
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.

Table 28: Unary operators in CFI expressions

Binary operators

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

Operator	Operands	Description
ADD	*cfiexpr*,*cfiexpr*	Addition
SUB	*cfiexpr*,*cfiexpr*	Subtraction
MUL	*cfiexpr*,*cfiexpr*	Multiplication
DIV	*cfiexpr*,*cfiexpr*	Division
MOD	*cfiexpr*,*cfiexpr*	Modulo
AND	*cfiexpr*,*cfiexpr*	Bitwise AND
OR	*cfiexpr*,*cfiexpr*	Bitwise OR
XOR	*cfiexpr*,*cfiexpr*	Bitwise XOR
EQ	*cfiexpr*,*cfiexpr*	Equal
NE	*cfiexpr*,*cfiexpr*	Not equal
LT	*cfiexpr*,*cfiexpr*	Less than
LE	*cfiexpr*,*cfiexpr*	Less than or equal
GT	*cfiexpr*,*cfiexpr*	Greater than
GE	*cfiexpr*,*cfiexpr*	Greater than or equal

Table 29: Binary operators in CFI expressions

Operator	Operands	Description
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.
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.
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.

Table 29: Binary operators in CFI expressions (Continued)

Ternary operators

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

Operator	Operands	Description
FRAME	*cfa,size,offset*	Get value from stack frame. The operands are: cfaAn identifier denoting a previously declared CFA. sizeA constant expression denoting a size in bytes. offsetA 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: condA CFA expression denoting a condition. trueAny CFA expression. falseAny 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*	Get value from memory. The operands are: sizeA constant expression denoting a size in bytes. typeA memory type. addrA 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

Consider a 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:

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

Table 31: Code sample with backtrace rows and columns

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

The SP column is empty since the CFA is defined in terms of the stack pointer. The RET column is the return address column—that is, the location of the return address. The R0 column has a '—' in the first line to indicate that the value of `R0` is undefined and can be discarded. The R1 column has `SAME` in the initial row to indicate that the value of the `R1` register will be restored on exit from the function.

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 NEAR

;; 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 NEAR
CFI R0 UNDEFINED
CFI R1 SAMEVALUE
CFI CFA SP + 2
```

```
CFI RET FRAME(CFA,-2)  ; Offset -2 from top of stack
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
     CFI    BLOCK func1 USING trivialCommon
func1:
     PUSH   R1
     CFI    CFA SP + 4
     CFI    R1 FRAME(CFA,-4)
     CFI    RET CFA - 4
     MOV    R1,#4
     CALL   func2
     POP    R0
     CFI    R1 R0
     CFI    CFA SP + 2
     CFI    RET CFA - 2
     MOV    R1,R0
     CFI    R1 SAMEVALUE
     RET
     CFI ENDBLOCK func1
```

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 issued as complete, self-explanatory messages. The message consists of the 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, 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
```

Diagnostic messages are displayed on the screen, as well as printed in the optional list file.

Severity levels

The diagnostic messages produced by the M16C/R8C IAR Assembler reflect problems or errors that are found in the source code or occur at assembly time.

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.

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 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.

ASSEMBLER INTERNAL ERROR MESSAGES

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, please report it to your software distributor or to IAR Technical Support. Please include information enough to reproduce the problem. This would typically include:

- 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.
- The version number of the assembler, which can be seen in the header of the list file generated by the assembler.

A

B

C

G

H

I

L

R

S