SC14 IAR Assembler

Reference Guide

for National Semiconductor’s

SC14xxx Co-processors

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

EDITION NOTICE

First edition: November 2001

Part number: ASC14-1

Contents

Tables

Preface

Welcome to the SC14 IAR Assembler Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use the SC14 IAR Assembler for creating DIP or GenDSP output for the SC14 co-processor.

Who should read this guide

You should read this guide if you plan to develop an application using assembler language for the SC14 co-processor and need to get detailed reference information on how to use the SC14 IAR Assembler. In addition, you should have a working knowledge of the following:

- The architecture and instruction set of the SC14xxx co-processor. Refer to the documentation from National Semiconductor for information about the SC14xxx co-processor.
- 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 SC14 IAR Assembler, you should read the *Introduction to the SC14 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 *CR16C 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 SC14 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 SC14xxx co-processor is described in a series of guides. For information about:

- Using the IAR Embedded Workbench™ and the IAR C-SPY™ Debugger, refer to the *CR16C IAR Embedded Workbench™ IDE User Guide*
- Using the IAR XLINK Linker™ and the IAR XLIB Librarian™, refer to the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide*.

All of these guides are delivered in 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.
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

Introduction to the SC14 IAR Assembler

This chapter describes the source code format for the SC14 IAR Assembler.

Refer to National Semiconductor'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 two operands.

The data definition directives, for example DB and DC8, can have any number of operands. For reference information about the data definition directives, see *Data definition or allocation directives*, page 66.

`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 SC14 IAR Assembler uses the default file extensions `s44`, `asm`, and `msa` for source files. **Note:** The file extension for object files is `r45`, which corresponds to the file extension for CR16C object 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 21.

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 21.

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.

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 `-s` assembler option. See page 17 for additional information.

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)	′

Table 3: ASCII character constant formats

Format	Value
' ' (2 quotes)	Empty string (no value).
""	Empty string (an ASCII null character).
\'	'
\\	\

Table 3: ASCII character constant formats (Continued)

PREDEFINED SYMBOLS

The SC14 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 avilable:

Symbol	Value
__DATE__	Current date in dd/Mmm/yyyy format (string).
__FILE__	Current source filename (string).
__IAR_SYSTEMS_ASM__	IAR assembler identifier (`0x01`).
__LINE__	Current source line number (number).
__TID__	Target identity, consisting of two bytes (number). The high byte is the target identity, which is `45` for `ASC14`. The low byte is the processor option `*`16.
__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 4: Predefined symbols

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:

```
tim    DC8    __TIME__,",",__DATE__,0; time and date
```

Testing symbols for conditional assembly

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

Programming hints

This section gives hints on how to write efficient code for the SC14 IAR Assembler.

PROCESSOR-SPECIFIC FILES

In the previous DIP IAR Assembler, ADIP, the environment variable `QDIPINFO` was used to point out the `*.chp` files from which the assembler reads opcode information. This variable is no longer used. Instead, the information about the `*.chp` file location is entered in the registry when you install the product.

The ADIP assembler generated an output where each byte in a word was swapped. Since no DIP application was programmed using a normal programming tool, this error was never discovered. When adding the GenDSP format, the error was found and corrected. If old DIP programs are to be used in the new environment, without being recompiled, this needs to be taken in consideration.

When using direct jumps in the old ADIP environment, byte addresses were used, for example:

```
JMP   0x24
```

These were later solved by the linker to give the DIP a word address.

The ASC14 assembler uses word addresses directly, for example:

```
JMP   0x12
```

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.

Output formats

The relocatable and absolute output is in the same format for all IAR assemblers, because object code is always intended for processing with the IAR XLINK Linker.

In absolute formats the output from XLINK is, however, normally compatible with the chip vendor’s debugger programs (monitors), as well as with PROM programmers and stand-alone emulators from independent sources.

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 *CR16C 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 `asc14` command:

```
asc14 [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 `tutor.s44`, use the following command to generate a list file to the default filename (`tutor.lst`):

```
asc14 tutor -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`:

```
asc14 tutor -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`:

```
asc14 tutor -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.

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

```
asc14 -f extend.xcl
```

Error return codes

When using the SC14 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 5: Assembler error return codes

ASSEMBLER ENVIRONMENT VARIABLES

Options can also be specified using the `ASMSC14` 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 variable can be used with the SC14 IAR Assembler:

Environment variable	Description
`ASC14_INC`	Specifies directories to search for include files; for example: `set ASC14_INC=c:\myinc\`
`ASMSC14`	Specifies command line options; for example: `set ASMSC14=-L -ws`

Table 6: Assembler environment variables

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

```
ASMSC14=-l temp.lst
```

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[DEAOM]`	Conditional list
`-D`*`symbol`*`[=`*`value`*`]`	Defines a symbol
`-E`*`number`*	Maximum number of errors
`-f extend.xcl`	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`	Omits header from assembler listing
`-O`*`prefix`*	Sets object filename prefix
`-o` *`filename`*	Sets object filename
`-p`*`lines`*	Lines/page
`-r{e|n}`	Generates debug information
`-S`	Sets silent operation
`-s{+|-}`	Case sensitive user symbols
`-t`*`n`*	Tab spacing
`-U`*`symb`*	Undefines a symbol
`-vSC14xxx`	Specifies target processor
`-w[`*`string`*`][s]`	Disables warnings
`-x[DI2]`	Includes cross-references

Table 7: 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 13.

This option is identical to the **Macro execution info** option in the **ASC14** 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 **ASC14** category in the IAR Embedded Workbench.

-c

`-c[DEAOM]`

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 13 for additional information.

The following table shows the available parameters:

Command line option	Description
`-cA`	Assembled lines only
`-cD`	Disable list file
`-cE`	No macro expansions
`-cM`	Macro definitions
`-cO`	Multiline code

Table 8: Conditional list (-c)

This option is related to the **List file** options in the **ASC14** 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:	`asc14 prog`
Test version:	`asc14 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:

```
asc14 prog -Dframerate=3
```

This option is identical to the **#define** option in the **ASC14** 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 **ASC14** category in the IAR Embedded Workbench.

-f -f extend.xcl

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

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

Example

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

```
asc14 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 ASC14_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 `ASC14_INC` environment variable is set.

This option is related to the **#include** option in the **ASC14** 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 **ASC14** 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`:

```
asc14 prog -Llist\
```

This option is related to the **List** options in the **ASC14** 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 **ASC14** category in the IAR Embedded Workbench.

-M `-Mab`

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

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

Example

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

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

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

```
asc14 filename -M'<>'
```

This option is identical to the **Macro quote chars** option in the **ASC14** 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 13 for additional information.

This option is related to the **List file** option in the **ASC14** 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 cannot include a space before the prefix.

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

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

Example

To send the object code to the file `obj\prog.r45` rather than to the default file `prog.r45:`

```
asc14 prog -Oobj\
```

This option is related to the **Output directories** option in the **General** 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, `r45` 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.r45` instead of the default `prog.r45`:

```
asc14 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

`-plines`

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 13 for additional information.

This option is identical to the **Lines/page** option in the **ASC14** category in the IAR Embedded Workbench.

-r

`-r{e|n}`

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.

The following table shows the available parameters:

Command line option	Description
`-re`	Includes the full source file into the object file
`-rn`	Generates an object file without source information; symbol information will be available.

Table 9: Generating debug information (-r)

This option is identical to the **Generate debug information** option in the **ASC14** 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 10: 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 **ASC14** 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 13 for additional information.

This option is identical to the **Tab spacing** option in the **ASC14** category in the IAR Embedded Workbench.

-U -U*symb*

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

By default the assembler provides certain predefined symbols; see *Predefined symbols*, page 4. 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:

```
asc14 prog -U __TIME__
```

This option is identical to the **#undef** option in the **ASC14** category in the IAR Embedded Workbench.

-v -vSC14*xxx*

Use the -v option to specify the processor configuration:

Option	Derivative
-vSC14428 (default)	SC14428, SC14428-DSP

Table 11: Specifying the processor configuration (-v)

The -v option is identical to the **Processor configuration** option in the **General** 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 71, 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:

```
asc14 prog -w-0
```

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

```
asc14 prog -w-0-8
```

This option is identical to the **Warnings** option in the **ASC14** 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 13 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 **ASC14** 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 plus.
`–`	Unary minus.
`!`	Logical NOT.
`BINNOT (~)`	Bitwise NOT.
`LOW`	Low byte.
`HIGH`	High byte.
`BYTE2`	Second byte.

`BYTE3`	Third byte.
`BYTE4`	Fourth byte
`DATE`	Current time/date.
`SFB`	Segment begin.
`SFE`	Segment end.
`SIZEOF`	Segment size.

MULTIPLICATIVE ARITHMETIC OPERATORS – 3

`*`	Multiplication.
`/`	Division.
`%`	Modulo.

SHIFT OPERATORS – 3

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

ADDITIVE ARITHMETIC OPERATORS – 4

`+`	Addition.
`–`	Subtraction.

AND OPERATORS – 5

`AND [&&]`	Logical AND.
`BINAND [&]`	Bitwise AND.

OR OPERATORS – 6

`OR [||]`	Logical OR.
`BINOR [|]`	Bitwise OR.
`XOR`	Logical exclusive OR.
`BINXOR [^]`	Bitwise exclusive OR.

COMPARISON OPERATORS – 7

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

Description of operators

The following sections give detailed descriptions of each assembler operator. See *Assembler expressions*, page 2, for related information.

`*` Multiplication (3).

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

Examples

```
2*2 → 4
-2*2 → -4
```

`+` Unary plus (1).

Unary plus operator.

Examples

```
+3 → 3
3*+2 → 6
```

`+` Addition (4).

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

Examples

```
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 (4).

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

Examples

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

/ Division (3).

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

Examples

```
9/2 → 4
-12/3 → -4
9/2*6 → 24
```

AND [&&] Logical AND (5).

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

Examples

```
B'1010 && B'0011 → 1
```

```
B’1010 && B’0101 → 1
B’1010 && B’0000 → 0
```

BINAND [&] Bitwise AND (5).

Use BINAND to perform bitwise AND between the integer operands.

Examples

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

BINNOT [~] Bitwise NOT (1).

Use BINNOT to perform bitwise NOT on its operand.

Examples

```
~ B’1010 → B’11111111111111111111111111110101
```

BINOR [|] Bitwise OR (6).

Use BINOR to perform bitwise OR on its operands.

Examples

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

BINXOR [^] Bitwise exclusive OR (6).

Use BINXOR to perform bitwise XOR on its operands.

Examples

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

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.

Examples

```
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.

Examples

```
BYTE3 0x12345678 → 0x34
```

BYTE4 Fourth byte (1).

BYTE4 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.

Examples

```
BYTE4 0x12345678 → 0x12
```

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).

Examples

To assemble the date of assembly:

```
today: DC8 DATE 5, DATE 4, DATE 3
```

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

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

Examples

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

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

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

Examples

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

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

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

Examples

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

`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.

Examples

```
HIGH 0xABCD → 0xAB
```

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

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

Examples

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

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.

Examples

```
LOW 0xABCD → 0xCD
```

LT [<] Less than (7).

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

Examples

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

MOD [%] Modulo (3).

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

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

Examples

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

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

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

Examples

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

NOT [!] Logical NOT (1).

Use NOT to negate a logical argument.

Examples

```
! B'0101 → 0
! B'0000 → 1
```

OR [||] Logical OR (6).

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

Examples

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

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.

Examples

```
       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.

Examples

```
      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.

SHL [<<] Logical shift left (3).

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

Examples

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

SHR [>>] Logical shift right (3).

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

Examples

```
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.

Examples

```
      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.

Examples

```
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.

Examples

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

`XOR` Logical exclusive OR (6).

Use `XOR` to perform logical XOR on its two operands.

Examples

```
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
`ALIAS`	Assigns a permanent value local to a module.	Value assignment
`ALIGN`	Aligns the location counter by inserting zero-filled bytes.	Segment control
`ASEG`	Begins an absolute segment.	Segment control
`ASSIGN`	Assigns a temporary value.	Value assignment
`CASEOFF`	Disables case sensitivity.	Assembler control
`CASEON`	Enables case sensitivity.	Assembler control

Table 14: Assembler directives summary

Directive	Description	Section
COL	Sets the number of columns per page.	Listing control
COMMON	Begins a common segment.	Segment control
DB	Generates 8-bit byte constants, including strings.	Data definition or allocation
DC8	Generates 8-bit byte constants, including strings.	Data definition or allocation
DC16	Generates 16-bit word constants, including strings.	Data definition or allocation
DEFINE	Defines a file-wide value.	Value assignment
DQ15	Generates 16-bit fixed-point values.	Data definition or allocation
DS	Allocates space for 8-bit bytes.	Data definition or allocation
DS8	Allocates space for 8-bit bytes.	Data definition or allocation
DS16	Allocates space for 16-bit words.	Data definition or allocation
DW	Generates 16-bit word constants, including strings.	Data definition or allocation
ELSE	Assembles instructions if a condition is false.	Conditional assembly
ELSEIF	Specifies a new condition in an IF…ENDIF block.	Conditional assembly
END	Terminates the assembly of the last module in a file.	Module control
ENDIF	Ends an IF block.	Conditional assembly
ENDM	Ends a macro definition.	Macro processing
ENDMOD	Terminates the assembly of the current module.	Module control
EQU	Assigns a permanent value local to a module.	Value assignment
EVEN	Aligns the program counter to an even address.	Segment control
EXITM	Exits prematurely from a macro.	Macro processing
EXPORT	Exports symbols to other modules.	Symbol control
EXTERN	Imports an external symbol.	Symbol control
IF	Assembles instructions if a condition is true.	Conditional assembly
IMPORT	Imports an external symbol.	Symbol control

Table 14: Assembler directives summary (Continued)

Directive	Description	Section
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 assembly 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 assembly-listing output.	Listing control
LSTPAG	Controls the formatting of output into pages.	Listing control
LSTREP	Controls the listing of lines generated by repeat directives.	Listing control
LSTXRF	Generates a cross-reference table.	Listing control
MACRO	Defines a macro.	Macro processing
MODULE	Begins a library module.	Module control
NAME	Begins a program module.	Module control
ODD	Aligns the program counter to an odd address.	Segment control
ORG	Sets the location counter.	Segment control
PAGE	Generates a new page.	Listing control
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
RSEG	Begins a relocatable segment.	Segment control
RTMODEL	Declares run-time model attributes.	Module control
SET	Assigns a temporary value.	Value assignment
VAR	Assigns a temporary value.	Value assignment

Table 14: Assembler directives summary (Continued)

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 VAR 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.
label	A symbolic label.
symbol	An assembler symbol.

Table 15: Assembler directive parameters

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 run-time 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 run-time 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.

Symbol control directives

These directives control how symbols are shared between modules.

Directive	Description
`EXTERN (IMPORT)`	Imports an external symbol.
`PUBLIC (EXPORT)`	Exports symbols to other modules.

Table 17: Symbol control directives

SYNTAX

```
EXTERN symbol [,symbol] …
PUBLIC 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.

EXAMPLES

The following example defines two subroutines to set the power pins high or low. The `pow_high` and `pow_low` subroutines are defined as `PUBLIC` so that they can be called from other modules.

```
        MODULE   pow_high
        PUBLIC   pow_high
```

```
          RSEG    CODE

pow_high  P_EN         // make sure that power pins are enabled
          P_LDH   0xFF // set all pins high
          RTN
          ENDMOD       // end of module

          MODULE  pow_low
          PUBLIC  pow_low
          RSEG    CODE

pow_low   P_EN         // make sure that power pins are enabled
          P_LDH   0x00 // set all pins low
          RTN
          ENDMOD       // end of module

          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.
ASEG	Begins an 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

SYNTAX

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

PARAMETERS

address Address where this segment part will be placed.

align Exponent of the value to which the address should be aligned, in the range 0 to 30. 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; one of `UNTYPED` (the default), `CODE`, or `DATA`.

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.

Note: The use of ASEG is not recommended in the SC14 IAR Assembler. Instead, use RSEG to specify the DIP_CODE segment.

Beginning a relocatable segment

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

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

Beginning a common segment

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

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

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

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

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

Setting the program location counter (PLC)

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

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

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

Aligning a segment

Use ALIGN to align the program location counter to a specified address boundary. The expression gives the power of two to which the program counter should be aligned.

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

ALIGN aligns by inserting zero/filled bytes. The EVEN directive aligns the program counter to an even address (which is equivalent to ALIGN 1) and the ODD directive aligns the program counter to an odd address.

EXAMPLES

Beginning a common segment

The following example defines two common segments containing variables:

```
        NAME    common1
        COMMON  data
count   DS16    2
        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 odd address, and adds some data. It then aligns to a 16-byte boundary before creating a 64-byte table.

```
     NAME      ALIGN_EX
     RSEG      CODE
     ORG       $+1
     EVEN

     DC8       1,2

     ENDMOD
     END
```

It generates the following code:

```
   1         0000
   2         0000                NAME      ALIGN_EX
   3         0000                RSEG      CODE
   4         0001                ORG       $+1
   5         0001 00             EVEN
   6         0002
   7         0002 0201           DC8       1,2
   8         0004
   9         0004                ENDMOD
```

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.

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
```

PARAMETERS

expr	Value assigned to symbol or value to be tested.
label	Symbol to be defined.
message	A text message that will be printed when *expr* is out of range.

min, max The minimum and maximum values allowed for *expr*.

DESCRIPTION

Defining a temporary value

Use either 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.

Checking symbol values

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

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

EXAMPLES

Redefining a symbol

The following example uses SET to redefine the symbol cons in a loop to generate a table of the first 4 powers of 3:

```
cons    SET     1
```

```
rep      MACRO   times
         DB      cons
cons     SET     cons*3
         IF      times>1
         rep     times-1
         ENDIF
         ENDM

         rep     4

         END
```

It generates the following code:

```
   1        0000
   2        0001          cons    SET     1
   3        0000
  11        0000
  12        0000                  rep     4
  12.1      0000 01               DB      cons
  12.2      0003          cons    SET     cons*3
  12.3      0001                  IF      4>1
  12        0001                  rep     4-1
  12.1      0001 03               DB      cons
  12.2      0009          cons    SET     cons*3
  12.3      0002                  IF      4-1>1
  12        0002                  rep     4-1-1
  12.1      0002 09               DB      cons
  12.2      001B          cons    SET     cons*3
  12.3      0003                  IF      4-1-1>1
  12        0003                  rep     4-1-1-1
  12.1      0003 1B               DB      cons
  12.2      0051          cons    SET     cons*3
  12.3      0004                  IF      4-1-1-1>1
  12.4      0004                  rep     4-1-1-1-1
  12.5      0004                  ENDIF
  12.6      0004                  ENDM
  12.7      0004                  ENDIF
  12.8      0004                  ENDM
  12.9      0004                  ENDIF
  12.10     0004                  ENDM
  12.11     0004                  ENDIF
  12.12     0004                  ENDM
  13        0004
  14        0004                  END
```

Using local and global symbols

In the following example the symbol x defined in module Local_1 is local to that module; a distinct symbol of the same name is defined in module Local_2. The DEFINE directive is used for declaring y for use anywhere in the file:

```
     NAME      Local_1
     RSEG      CODE

x    EQU       1
y    DEFINE    2

     WT        x    ; WT  1
     WT        y    ; WT  2

     ENDMOD


     NAME      Local_2
     RSEG      CODE

x    EQU       4

     WT        x    ; WT  4
     WT        y    ; WT  2

     ENDMOD
     END
```

The symbol y defined in module Local_1 is also available to module Local_2.

It generates the following code:

```
   1         0000                NAME      Local_1
   2         0000                RSEG      CODE
   3         0000
   4         0001           x    EQU       1
   5         0002           y    DEFINE    2
   6         0000
   7       1 0000 0109           WT        x    ; WT  1
   8       1 0002 0209           WT        y    ; WT  2
   9         0004
  10         0004                ENDMOD

  11         0000
  12         0000
  13         0000                NAME      Local_2
  14         0000                RSEG      CODE
  15         0000
```

```
    16        0004          x    EQU       4
    17        0000
    18      1 0000 0409          WT        x    ; WT  4
    19      1 0002 0209          WT        y    ; WT  2
    20        0004
    21        0004               ENDMOD
```

Using the LIMIT directive

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

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

Conditional assembly directives

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

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

Table 20: Conditional assembly directives

SYNTAX

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

PARAMETERS

condition One of the following:

An absolute expression The expression must not contain forward or external references, and any non-zero value is considered as true.

string1=string2 The condition is true if *string1* and *string2* have the same length and contents.

string1<>string2 The condition is true if *string1* and *string2* have different length or contents.

DESCRIPTION

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

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

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

EXAMPLES

The following macro shows the usage both of macro quotes and of the IF, ELSE, and ENDIF directives. The first line in the macro checks the number of arguments passed to the macro.

```
MACRO   mac
        IF _args == 2
          DW   \0 + \1
        ELSE
          DW   \0
        ENDIF
        ENDMAC

        ASEG

        mac   2,2     ; two arguments passed to the macro
        mac  <2,2>    ; passed as one argument

        END
```

It produces the following code:

```
15         0000           ASEG
16         0000
17         0000           mac   2,2    ; two arguments passed to the
                                       ; macro
17.1       0000           IF _args == 2
17.2       0000 0400      DW   2 + 2
17.3       0002           ELSE
17.4       0002             DW   2
17.5       0002           ENDIF
17.6       0002           ENDMAC
18         0002           mac  <2,2>    ; passed as one argument
18.1       0002           IF _args == 2
18.2       0002             DW   2,2 +
18.3       0002           ELSE
18.4       0002 02000200    DW   2,2
18.5       0006           ENDIF
18.6       0006           ENDMAC
19         0006
20         0006           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.

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

```
errmac  MACRO   text
        JMP     abort
        DC8     text,0
        ENDM
```

This 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'
```

This will be expanded by the assembler to:

```
        JMP      abort
        DC8      'Disk not ready',0
```

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

The previous example could therefore be written as follows:

```
errmac  MACRO
        JMP       abort
        DC8       1,0
        ENDM
```

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

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

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

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

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

Passing special characters

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

For example:

```
        macro     <A, #42H>
        END
```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 14.

Using macro quotes

This macro shows the usage of macro quotes. The first call will generate the code "DW 2+2" while the second call, which uses macro quotes, will generate "DW 2,2".

```
MACRO   mac
        IF _args == 2
          DW   \0 + \1
        ELSE
          DW   \0
        ENDIF
        ENDMAC

        ASEG

        mac   2,2     ; two arguments passed to the macro
        mac  <2,2>    ; passed as one argument

        END
```

It will produce the following code:

```
15         0000                  ASEG
16         0000
17         0000                  mac   2,2     ; two arguments passed
                                              ;  to the macro
17.1       0000                  IF _args == 2
17.2       0000 0400               DW   2 + 2
17.3       0002                  ELSE
17.4       0002                    DW   2
17.5       0002                  ENDIF
17.6       0002                  ENDMAC
18         0002                  mac  <2,2>    ; passed as one argument
18.1       0002                  IF _args == 2
18.2       0002                    DW   2,2 +
18.3       0002                  ELSE
18.4       0002 02000200           DW   2,2
18.5       0006                  ENDIF
18.6       0006                  ENDMAC
19         0006
20         0006                  END
```

How macros are processed

There are three distinct phases in the macro process:

1 The assembler performs scanning and saving of macro definitions. The text between `MACRO` and `ENDM` is saved but not syntax checked. Include-file references `$`*`file`* are recorded and will be included during macro *expansion*.

2 A macro call forces the assembler to invoke the macro processor (expander). The macro expander switches (if not already in a macro) the assembler input stream from a source file to the output from the macro expander. The macro expander takes its input from the requested macro definition.
The macro expander has no knowledge of assembler symbols since it only deals with text substitutions at source level. Before a line from the called macro definition is handed over to the assembler, the expander scans the line for all occurrences of symbolic macro arguments, and replaces them with their expansion arguments.

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

Repeating statements

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

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

Only double quotes have a special meaning and its 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 macros sets the power pins:

```
MACRO  power_pins_high // no use of incoming argument
       P_EN            // make sure that power pins are enabled
       P_LDH   0xFF    // set all pins high
       ENDMAC          // end of macro

MACRO  power_pins_low  // no use of incoming argument
       P_EN            // make sure that power pins are enabled
       P_LDH   0x00    // set all pins low
       ENDMAC          // end of macro
```

The macros would be called with a statement such as:

```
power_pins_high
```

The following program calls the macros from the file `macros.s44`:

```
        RSEG    DIP_CODE
#include "macros.s44"

        WT      0x04
        JMP     init
        power_pins_high
        JMP     mute
        BR      QUIT

mute    A_RCV36
        RTN

init    A_NORM
        RTN

QUIT    WNT     0x01
        END
```

Using REPTC and REPTI

The following example assembles a series of `DB`s to store a version string in memory to a subroutine `plot` to plot each character in a string:

```
      RSEG   DIP_DATA

      REPTC  char, "Version 1.21A"
      DB     'char'
      ENDR
      END
```

This produces the following code:

```
1         0000                  RSEG   DIP_DATA
2         0000
3         0000                  REPTC  char, "Version 1.21A"
4         0000                  DB     'char'
5         0000                  ENDR
5.1       0000 56               DB     'V'
5.2       0001 65               DB     'e'
5.3       0002 72               DB     'r'
5.4       0003 73               DB     's'
5.5       0004 69               DB     'i'
5.6       0005 6F               DB     'o'
5.7       0006 6E               DB     'n'
5.8       0007 20               DB     ' '
5.9       0008 31               DB     '1'
5.10      0009 2E               DB     '.'
5.11      000A 32               DB     '2'
5.12      000B 31               DB     '1'
5.13      000C 41               DB     'A'
6         000D                  END
```

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

```
RSEG
EXTERN x,y,z

REPTI  number, x, y, z
DW     number
ENDR
END
```

This produces the following code:

```
1         0000                  RSEG
2         0000                  EXTERN x,y,z
3         0000
4         0000                  REPTI  number, x, y, z
5         0000                  DW     number
6         0000                  ENDR
6.1       0000 ....             DW      x
6.2       0002 ....             DW      y
6.3       0004 ....             DW      z
7         0006                  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 assembly-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:

```
   MACRO   mac
           LSTCND+
           IF _args == 2
             DW   \0 + \1
           ELSE
             DW   \0
           ENDIF
           ENDMAC


   RSEG    DIP_CODE

   mac     <2,2>
   mac      2,2

   END
```

This will generate the following listing:

```
  9         0000
 10         0000
 11         0000             RSEG    DIP_CODE
 12         0000
 13         0000             mac     <2,2>
 13.1       0000                     LSTCND+
 13.2       0000                     IF _args == 2
 13.3       0000                     ELSE
 13.4       0000 02000200              DW   2,2
 13.5       0004                     ENDIF
 13.6       0004                     ENDMAC
 14         0004             mac      2,2
 14.1       0004                     LSTCND+
 14.2       0004                     IF _args == 2
 14.3       0004 0400                  DW   2 + 2
 14.4       0006                     ELSE
 14.5       0006                     ENDMAC
 15         0006
 16         0006             END
```

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

```
  1         0000                ASEG
  2         0000
  3         0000                LSTCOD-
  4         0000 01000A00*      DW     1,10,100,1000,10000
  5         000A                LSTCOD+
  6         000A 01000A00       DW     1,10,100,1000,10000
                 6400E803 1027
  7         0014
  8         0014                END
```

Note: An asterisk (`*`) indicates that the line has been truncated.

Controlling the listing of macros

The following example shows the effect of `LSTEXP`:

```
       LSTEXP-
MACRO power_pins_high // no use of incoming argument
      P_EN            // make sure that power pins are enabled
      P_LDH    0xFF   // set all pins high
      ENDMAC          // end of macro

MACRO power_pins_low  // no use of incoming argument
```

```
        P_EN            // make sure that power pins are enabled
        P_LDH   0x00    // set all pins low
        ENDMAC          // end of macro
```

The macros are defined int the file `macros.s44` which is called from the following program:

```
        RSEG    DIP_CODE


#include "macros.s44"

        WT      0x04
        JMP     init
        power_pins_high
        JMP     mute
        BR      QUIT

mute    A_RCV36
        RTN

init    A_NORM
        RTN

QUIT    WNT     0x01
        END
```

This will produce the following output without expansion of the `power_pins_high` macro:

```
 1         0000                    RSEG    DIP_CODE
 2         0002
 3         0002
 4         0002         #include "macros.s44"
 5         0002
 6       4 0002 0904               WT      0x04

 7       7 0004 ....               JMP     init
 8         0006                    power_pins_high
 9      12 000A ....               JMP     mute
10       0 000C ....               BR      QUIT
11         000E
12       1 000E 8200    mute       A_RCV36
13       2 0010 0400               RTN
14         0012
15       1 0012 C500    init       A_NORM
16       2 0014 0400               RTN
17         0016
```

```
18      14 0016 0801     QUIT    WNT     0x01

19         0018                  END
```

Note: The second field from the left shows the cycle count. To make it easier to count the consumed cycles, the assembler has a built in cycle counter. This mechanism is automatically generated and the output is displayed in the list file. The cycle count represents the accumulated cycles so far in the program: the cycle count on the `WT` instruction, shown above, indicates that when this line is executed 4 cycles have been performed; when `JMP` is executed the count is incremented to 7 cycles.

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.

Table 23: C-style preprocessor directives

Directive	Description
`#message`	Generates a message on standard output.
`#undef`	Undefines a label.

Table 23: C-style preprocessor directives (Continued)

SYNTAX

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

PARAMETERS

`condition` One of the following:

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

`filename` Name of file to be included.

`label` Symbol to be defined, undefined, or tested.

`message` Text to be displayed.

text	Value to be assigned.

DESCRIPTION

Defining and undefining labels

Use `#define` to define a temporary label.

`#define` *`label value`*

is similar to:

`label` `VAR` *`value`*

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

Conditional directives

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

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

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

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

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

Including source files

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

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

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

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

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

Displaying errors

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

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 version 13    ; API version

   ASEG
   DW    version      ; Expands to "DW 13 ; API version"

   DW    version + 1  ; Expands to "DW 13 ; API version + 1"
                      ; So memory will contain 13 and not
                      ;  13+1 since '+ 1' is commented away
```

EXAMPLES

Using conditional directives

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

```
#define  deriv  14428

#ifdef  deriv
current DW     deriv
#else
#error  "'deriv' not defined"
#endif

        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 are defined in `macros.s44`:

```
MACRO  power_pins_high // no use of incoming argument
       P_EN            // make sure that power pins are enabled
       P_LDH   0xFF    // set all pins high
       ENDMAC          // end of macro

MACRO  power_pins_low  // no use of incoming argument
       P_EN            // make sure that power pins are enabled
       P_LDH   0x00    // set all pins low
       ENDMAC          // end of macro
```

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

```
        ASEG
        ORG     0x02

#include "macros.s44"

        WT      0x04
        JMP     init
        power_pins_high
        JMP     mute
        BR      QUIT

mute    A_RCV36
        RTN

init    A_NORM
        RTN

QUIT    WNT     0x01
        END
```

Data definition or allocation directives

These directives define temporary values or reserve memory.

Directive	Description
`DC8`, `DB`	Generates 8-bit byte constants, including strings.
`DC16`, `DW`	Generates 16-bit word constants, including strings.
`DQ15`	Generates 16-bit fixed-point constants.
`DS8`, `DS`	Allocates space for 8-bit bytes.
`DS16`	Allocates space for 16-bit words.

Table 24: Data definition or allocation directives

SYNTAX

```
DB expr
DC8 expr [,expr] ...
DC16 expr [,expr] ...
DQ15 value[,value] ...
DS expr[,expr]
DS8 expr [,expr] ...
DS16 expr [,expr] ...
DW 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 size. Double-quoted strings will be zero terminated.
value	A valid absolute expression or a floating-point constant.

DESCRIPTION

Use `DB`, `DC8`, `DC16`, `DQ15`, or `DW` to reserve and initialize memory space.

Use `DS`, `DS8`, or `DS16` to reserve uninitialized memory space.

EXAMPLES

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 25: 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.

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

```
RADIX  D'10
```

Controlling case sensitivity

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

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

EXAMPLES

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.s44`:

```
MACRO  power_pins_high // no use of incoming argument
       P_EN            // make sure that power pins are enabled
       P_LDH   0xFF    // set all pins high
       ENDMAC          // end of macro

MACRO  power_pins_low  // no use of incoming argument
       P_EN            // make sure that power pins are enabled
       P_LDH   0x00    // set all pins low
       ENDMAC          // end of macro
```

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

```
        RSEG    DIP_CODE

$include "mymacros.s44"

        WT      0x04
        JMP     init
        power_pins_high
        JMP     mute
        BR      QUIT
```

```
mute    A_RCV36
        RTN

init    A_NORM
        RTN

QUIT    WNT     0x01
        END
```

Defining comments

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

```
/*
Program to read serial input.
Version 3: dd.mm.yy
Author: mjp
*/
```

Changing the base

To set the default base to 16:

```
        RADIX  D’16
        WT     A,12
```

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

Controlling case sensitivity

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

```
label  WNT  0x01       ;stored as "LABEL"
       JMP  LABEL
```

The following will generate a duplicate label error:

```
label  WNT  0x01
LABEL  WNT  0x01       ;Error: "LABEL" already defined
       END
```

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. A typical diagnostic message from the assembler is produced in the form:

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

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

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

Severity levels

The diagnostic messages produced by the SC14 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 files generated by the assembler.