M32C IAR ASSEMBLER

Reference Guide

for Renesas

M32C and M16C/8x Series of CPU Cores

COPYRIGHT NOTICE

DISCLAIMER

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

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

TRADEMARKS

Second edition: June 2004

Part number: AM32C-2

WELCOME

Welcome to the M32C IAR Assembler Reference Guide.

This guide provides reference information about the IAR Systems Assembler for the M32C and M16C/8x Series of CPU cores, and applies to the command line version of this tool.

Before reading this guide we recommend you to read the initial chapters of the *IAR Embedded Workbench™ IDE User Guide*, where you will find information about installing the IAR Systems development tools, product overviews, and tutorials that will help you get started. The *IAR Embedded Workbench™ IDE User Guide* also contains complete reference information about the IAR Embedded Workbench™ and the M32C IAR C-SPY™ Debugger.

For information about programming with the M32C IAR C Compiler, refer to the *M32C IAR C/C++ Compiler Reference Guide*.

For information about using the IAR XLINK Linker™ and IAR XLIB Librarian™, refer to the *IAR Linker and Library Tools Reference Guide*, which is available from the M32C IAR Embedded Workbench™ **Help** menu.

ABOUT THIS GUIDE

This guide consists of the following chapters:

Introduction to the M32C Assembler provides a brief summary of the M32C Assembler and gives programming hints.

Assembler options first explains how to set the M32C Assembler options and how to use environment variables. It then gives an alphabetical summary of the assembler options, and contains complete reference information about each option.

Assembler file formats describes the source format for the M32C Assembler, and the format of assembler listings.

Assembler operators gives a summary of the assembler operators, arranged in order of precedence, and provides a complete alphabetical list of the M32C Assembler operators, with a full description of each one.

Assembler directives gives an alphabetical summary of the M32C Assembler directives, and provides complete reference information about the M32C Assembler directives, classified into groups according to their function.

Assembler diagnostics provides a list of error and warning messages specific to the M32C Assembler.

ASSUMPTIONS

This guide assumes that you already have a working knowledge of the following:

- The architecture of the M32C and M16C/8x Series CPU cores.
- The M32C and M16C/8x Series assembler instruction set.
- Windows 98/2000/Me/NT, depending on your host system.
- The IAR Systems development tools and the project model, as described in the *IAR Embedded Workbench™ IDE User Guide*.

Note: The illustrations in this guide show the IAR Embedded Workbench running in a Windows-style environment, and their appearance will be slightly different if you are using another platform.

CONVENTIONS

This guide uses the following typographical conventions:

Style	*Used for*
`computer`	Text that you type in, or that appears on the screen.
`parameter`	A label representing the actual value you should type 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	Cross-references to another part of this guide, or to another guide.
	Identifies instructions specific to the versions of the IAR development tools for the IAR Embedded Workbench interface.
	Identifies instructions specific to the command line versions of IAR development tools.

CONTENTS

INTRODUCTION TO THE M32C ASSEMBLER

This chapter describes the key features of the IAR Systems M32C Assembler, and provides some programming hints.

KEY FEATURES

The IAR Systems M32C Assembler is a powerful relocating macro assembler with a versatile set of directives.

The assembler incorporates a high degree of compatibility with the CPU core manufacturer's assembler to ensure that software originally developed using that assembler can be transferred to the IAR Systems Assembler with a few modifications.

The IAR Systems M32C Assembler provides the following features:

GENERAL

- One pass assembly, for fast execution.
- Integration with the IAR XLINK Linker™ and IAR XLIB Librarian™.
- Integration with other IAR Systems software for the M32C and M16C/8x Series of CPU cores.
- Self-explanatory error messages.

ASSEMBLER FEATURES

- Up to 65536 relocatable segments per module.
- 32-bit arithmetic and IEEE floating-point constants.
- 255 significant characters in symbols.
- Powerful recursive macro facilities.
- Number of symbols and program size limited only by available memory.
- Support for complex expressions with external references.
- Forward references allowed to any depth.
- Macros in Intel/Motorola style.

- Support for C language preprocessor directives.

PROGRAMMING HINTS

ACCESSING SPECIAL FUNCTION REGISTERS

A header file that defines the special function registers (SFRs) is included in the M32C Assembler delivery. The header file is called `iom32c.h`.

Since the header file is intended to be used with the M32C IAR C/EC++ Compiler, `ICCM32C`, the SFR declaration is made with macros. The macros that convert the declaration to assembler or compiler syntax are defined in the `iomacros.h` file.

The `iom32c.h`. header file is also suitable to use as a template when creating new header files for any future M32C or M16C/80 derivatives.

Example
The Universal Asynchronous Receiver Transmitter (UART0) transmitter buffer register at address `0x362` of the M32C CPU core is defined in the `iom32c.h`. file as:

```
__SFR(__U0TB, 0x362, __REG16, __READ_WRITE)
```

Note: The `__REG16` size definition and the `__READ_WRITE` attribute definition are used in the compiler exclusively.

The declaration is converted by the file `iomacros.h` to:

```
__U0TB DEFINE 0x362
```

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

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

C-STYLE PREPROCESSOR DIRECTIVES

The C-style preprocessor directives are processed before other assembler directives. Therefore, do not use preprocessor directives in assembler macros and do not mix them with assembler-style comments. For additional information, see *C-style preprocessor directives*, page 86.

ASSEMBLER OPTIONS

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

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

SETTING ASSEMBLER OPTIONS

To set assembler options from the command line, you include them on the command line, after the `am32c` command:

```
am32c [options] [sourcefile] [options]
```

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

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

For example, when assembling the source file `power2.s48`, use the following command to generate a list file to the default filename (`power2.lst`):

```
am32c power2 -L
```

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

```
am32c power2 -l list.lst
```

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

```
am32c power2 -Llist\
```

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

EXTENDED COMMAND LINE FILE

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

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

```
am32c -f extend.xcl
```

Error return codes

When using the M32C 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

ASSEMBLER ENVIRONMENT VARIABLES

Options can also be specified using the `ASMM32C` environment variable. The assembler appends the value of this variable to every command line, so it provides a convenient method of specifying options that are required for every assembly.

The following environment variables can be used with the M32C IAR Assembler:

Environment variable	*Description*
`ASMM32C`	Specifies command line options; for example: `set ASMM32C=-L -ws`
`AM32C_INC`	Specifies directories to search for include files; for example: `set AM32C_INC=c:\myinc\`

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

```
ASMM32C=-l temp.lst
```

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

OPTIONS SUMMARY

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

Command line option	*Description*
`-B`	Macro execution information
`-b`	Make a library module
`-c{DMEAO}`	Conditional list
`-D`*`symb`*`[=`*`value`*`]`	Define symbol
`-E`*`number`*	Maximum number of errors
`-f extend.xcl`	Extend the command line
`-G`	Open standard input as source
`-I`*`prefix`*	Include paths
`-i`	#included text
`-L[`*`prefix`*`]`	List to prefixed source name
`-l` *`filename`*	List to named file
`-M`*`ab`*	Macro quote characters
`-N`	No header
`-O`*`prefix`*	Set object filename prefix
`-o` *`filename`*	Set object filename
`-p`*`lines`*	Lines/page
`-r`	Generate debug information
`-S`	Set silent operation

Command line option	*Description*
`-s{+|-}`	Case sensitive user symbols
`-t`*n*	Tab spacing
`-U`*symb*	Undefine symbol
`-v[0|1]`	Processor configuration
`-w[`*string*`][s]`	Disable warnings
`-x{DI2}`	Include cross-reference

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

-B

Prints macro execution information. This option is mainly used in conjunction with the list file options `-L` or `-l`; for additional information, see page 11.

SYNTAX

```
-B
```

DESCRIPTION

Causes the assembler to 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 identical to the **Macro execution info** option in the **AM32C** category in the IAR Embedded Workbench.

-b

Makes a library module to be used with the IAR XLIB Librarian.

SYNTAX

```
-b
```

DESCRIPTION

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 **AM32C** category in the IAR Embedded Workbench.

-c

Conditional list. This option is mainly used in conjunction with the list file options `-L` and `-l`; see page 11 for additional information.

SYNTAX

```
-c{DMEAO}
```

DESCRIPTION

Sets one or more of the following:

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

This option is related to the **List** options in the **AM32C** category in the IAR Embedded Workbench.

-D

Defines a symbol to be used by the preprocessor.

SYNTAX

`D`*`symb`*`[=`*`value`*`]`

DESCRIPTION

Defines a symbol with the name *`symb`* 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.

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:       am32c prog
test version:             am32c 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:

```
am32c prog -Dframerate=3
```

This option is identical to the **#define** option in the **AM32C** category in the IAR Embedded Workbench.

-E

Sets maximum number of errors to be reported.

SYNTAX

`-E`*`number`*

DESCRIPTION

Sets the maximum number of errors the assembler reports.

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.

-f

Extends the command line.

SYNTAX

```
-f extend.xcl
```

DESCRIPTION

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

```
am32c prog -f extend.xcl
```

-G

Opens standard input as source.

SYNTAX

```
-G
```

DESCRIPTION

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

Includes paths to be used by the preprocessor.

SYNTAX

`-I`*`prefix`*

DESCRIPTION

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

By default, the assembler searches for `#include` files only in the current working directory and in the paths specified in the `AM32C_INC` environment variable. The `-I` option allows you to give the assembler the names of directories which it will also search if it fails to find the file in the current working directory.

For example, using the options:

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

and then writing:

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

in the source, will make the assembler search first in the current directory, then in the directory `c:\global\`, and finally in the directory `c:\thisproj\headers\` provided that the `AM32C_INC` environment variable is set.

This option is related to the **Include** option in the **AM32C** category in the IAR Embedded Workbench.

-i

Includes `#include` text to be used by the preprocessor.

SYNTAX

```
-i
```

DESCRIPTION

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 **#included text** option in the **AM32C** category in the IAR Embedded Workbench.

-L

Generates a list file with the prefixed source file name.

SYNTAX

`-L[`*`prefix`*`]`

DESCRIPTION

Causes the assembler to generate a listing and send it to the file *`prefixsourcename`*`.lst`. Notice that you must not include a space before the prefix.

By default, the assembler does not generate a list file. To simply generate a listing, you use the `-L` option without a prefix. The listing is sent to the file with the same name as the source, but extension `lst`.

The `-L` option lets you specify a prefix, for example to direct the list file to a subdirectory:

```
am32c prog -Llist\
```

This sends the list file to `list\prog.lst` rather than the default `prog.lst`.

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

This option is related to the **List** options in the **AM32C** category in the IAR Embedded Workbench.

-l

Generates a list file with the specified filename.

SYNTAX

`-l filename`

DESCRIPTION

Causes the assembler to generate a listing and send it to the named file. 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 **AM32C** category in the IAR Embedded Workbench.

-M

Specifies quote characters for macro arguments.

SYNTAX

`-Mab`

DESCRIPTION

Sets the characters used for the left and right quotes of each macro argument to *a* and *b* respectively.

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

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:

```
am32c filename -M'<>'
```

This option is identical to the **Macro quote chars** option in the **AM32C** category in the IAR Embedded Workbench.

-N

Omits the header from assembler list file. This option is useful in conjunction with the list file options `-L` or `-l`; see page 11 for additional information.

SYNTAX

```
-N
```

DESCRIPTION

By default the assembler list file contains a header section. Use this option to omit the header section that is normally printed in the beginning of the list file.

This option is related to the **List** options in the **AM32C** category in the IAR Embedded Workbench.

-O

Sets the object filename prefix.

SYNTAX

```
-Oprefix
```

DESCRIPTION

Set the prefix to be used on the filename of the object file. Notice that you must not include a space before the prefix.

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

```
am32c prog -Oobj\
```

This sends the object to `obj\prog.r48` rather than to the default file `prog.r48`.

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

-o

Sets the object filename.

SYNTAX

`-o filename`

DESCRIPTION

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, `r48` is used.

For example, the following command puts the object code to the file `obj.r48` instead of the default `prog.r48`:

```
am32c prog -o obj
```

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

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

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

Sets number of lines per page. This option is used in conjunction with the list options `-L` or `-l`; see page 11 for additional information.

SYNTAX

`-p lines`

DESCRIPTION

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

This option is identical to the **Lines/page** option in the **AM32C** category in the IAR Embedded Workbench.

-r

Generates debug information to be used with C-SPY.

SYNTAX

```
-r
```

DESCRIPTION

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

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

This option is identical to the **Generate debug information** option in the **AM32C** category in the IAR Embedded Workbench.

-S

Specifies silent operation.

SYNTAX

```
-S
```

DESCRIPTION

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. You can 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

Makes user symbols case sensitive.

SYNTAX

```
-s{+|-}
```

DESCRIPTION

The `-s` option determines 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

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 **AM32C** category in the IAR Embedded Workbench.

-t

Specifies the tab spacing. This option is useful in conjunction with the list options `-L` or `-l`; see page 11 for additional information.

SYNTAX

```
-tn
```

DESCRIPTION

The `-t` option sets the number of character positions per tab stop to *n*, which must be in the range 2 to 9.

By default, the assembler sets eight character positions per tab stop.

This option is identical to the **Tab spacing** option in the **AM32C** category in the IAR Embedded Workbench.

-U

Undefines a predefined symbol.

SYNTAX

`-U`*`symb`*

DESCRIPTION

The `-U` option undefines the symbol *`symb`*.

By default, the assembler provides certain predefined symbols; see *Predefined symbols*, page 27. 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.

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

```
am32c prog -U __TIME__
```

This option is identical to the **#undef** option in the **AM32C** category in the IAR Embedded Workbench.

-v

Specifies the processor configuration.

SYNTAX

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

DESCRIPTION

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

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

The following table shows how the `-v` options are mapped to the processor options:

Option	*Processor*
`-v0`	M32C
`-v1`	M16C/80

If no processor configuration option is specified, the assembler uses the `-v0` option by default.

The `-v` option is identical to the **Processor configuration** option in the **General** category in the IAR Embedded Workbench.

-w

Disables warnings.

SYNTAX

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

DESCRIPTION

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

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

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

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.

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

```
am32c prog -w-0
```

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

```
am32c prog -w-0-8
```

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

This option is identical to the **Warnings** option in the **AM32C** category in the IAR Embedded Workbench.

-x

Includes cross-references in the assembler list file. This option is useful in conjunction with the list options `-L` or `-l`; see page 11 for additional information.

SYNTAX

```
-x{DI2}
```

DESCRIPTION

Causes the assembler to generate a cross-reference list at the end of the list file. See the chapter *Assembler file formats*, page 21, for details.

The following options are available:

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

This option is identical to the **Include cross-reference** option in the **AM32C** category in the IAR Embedded Workbench.

ASSEMBLER FILE FORMATS

This chapter describes the source format for the M32C IAR Assembler, and the format of assembler listings.

SOURCE FORMAT

The format of an assembler source line is as follows:

```
[label [:]] operation [.size] [:format] [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.
size	Size specifier: Short (`.S`), Byte (`.B`), Word (`.W`), Address (`.A`), or Long (`.L`).
format	Format specifier: Generic (`:G`), Quick (`:Q`), Short (`:S`), or Zero (`:Z`).
operands	One, two, or three operands, separated by commas.
comment	Comment, preceded by a `;` (semicolon). C++ style comments starting with `//` (double slash) are also allowed.
\	Line continuation character.

The fields can be separated by spaces or tabs.

A source line may not exceed 2048 characters.

Tab characters (ASCII 09H), are expanded according to the most common practice; i.e. to columns 8, 16, 24 etc.

EXPRESSIONS AND OPERATORS

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 to generate code.

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators. The valid operands in an expression are:

- 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*.

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 where the segments are located by the IAR XLINK Linker™.

Such expressions are evaluated and resolved at link time, by the linker. There are no restrictions on the expression; any operator can be used on symbols from any segment, or any combination of segments. For example, a program could define the segments `DATA` and `CODE` as follows:

```
          .MODULE EX_1
          .EXTERN third

          .RSEG DATA
first     DS8    9
second    DS8    3

          .RSEG   CODE
```

```
INC.B   first-7
INC.B   first+7
INC.B   first*third-second
.ENDMOD
```

The following list shows what the assembler list file looks like:

```
000000 A68E..... INC.B   first-7
000005 A68E..... INC.B   first+7
00000A A68E..... INC.B   first*third-second
00000F           .END
```

The expressions are evaluated and resolved by XLINK:

```
xlink -Z(CODE)CODE=80000 -Z(DATA)DATA=500 -Dthird=2
filename
```

After resolving the relocatable symbols the following absolute code is generated:

```
080000 A68EF90400 INC.B   first-7; 0x04F9
080005 A68E070500 INC.B   first+7; 0x0507
08000A A68EF70400 INC.B   first*third-second; 0x04F7
08000F            .END
```

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 user-defined symbols case is significant. Case is insignificant for built-in symbols like instructions, registers, operators, and directives. For user-defined symbols, case sensitivity can be turned on and off, see *-s*, page 16.

LABELS

Symbols used for memory locations are referred to as labels.

Location counter

The location counter is called `$` (dollar). For example:

```
JMP             $          ; Loop forever
```

FORMAT MODIFIERS

The assembler will normally assemble an instruction into the smallest possible number of bytes. The format modifiers can be used to instruct the assembler to use a less efficient format. This feature might be useful to keep the execution speed or code size fixed with varying data.

The following examples demonstrate how the same instruction can generate different amounts of code, and how the format specifiers can be used to limit the optimizations that are applied.

In this example the assembler chooses the most compact format for a variety of MOV commands, taking between 4 bytes and 1 byte.

```
.MODULE EX_2
.RSEG CODE
MOV.W   #0x1234,R1
MOV.W   #0x1234,R0
MOV.W   #0x3,R0
MOV.W   #0x0,R0
MOV.B   #0x3,R0L
MOV.B   #0x0,R0L
```

Now the same instructions as above are forced into the most general format (the variations in length are caused by the size of the immediate data):

```
MOV.W:G   #0x1234,R1
MOV.W:G   #0x1234,R0
MOV.W:G   #0x3,R0
MOV.W:G   #0x0,R0
MOV.B:G   #0x3,R0L
MOV.B:G   #0x0,R0L
```

In the next example, a single instruction is forced into all the formats. If a format is not specified, the most efficient one is used.

```
MOV.B     #0x0,R0L
MOV.B:Z   #0x0,R0L
MOV.B:S   #0x0,R0L
MOV.B:Q   #0x0,R0L
MOV.B:G   #0x0,R0L
.ENDMOD
```

The following list file is produced:

```
15    000000 99EF3412          MOV.W   #0x1234,R1
16    000004 053412            MOV.W   #0x1234,R0
17    000007 F9A3              MOV.W   #0x3,R0
18    000009 03                MOV.W   #0x0,R0
19    00000A F8A3              MOV.B   #0x3,R0L
20    00000C 02                MOV.B   #0x0,R0L
21    00000D
22    00000D 99EF3412          MOV.W:G #0x1234,R1
23    000011 99AF3412          MOV.W:G #0x1234,R0
24    000015 99AF0300          MOV.W:G #0x3,R0
25    000019 99AF0000          MOV.W:G #0x0,R0
26    00001D 98AF03            MOV.B:G #0x3,R0L
27    000020 98AF00            MOV.B:G #0x0,R0L
28    000023
29    000023 02                MOV.B   #0x0,R0L
30    000024 02                MOV.B:Z #0x0,R0L
31    000025 0400              MOV.B:S #0x0,R0L
32    000027 F8A0              MOV.B:Q #0x0,R0L
33    000029 98AF00            MOV.B:G #0x0,R0L
```

Attempting to force an instruction into a format that is too small will result in an error. For additional information about error messages, refer to the chapter *Assembler diagnostics*.

For detailed information about the formats available for each command, refer to the manufacturer's data book.

Note: Errors may also be detected in the range check during the linking process. For additional information, refer to the *IAR Linker and Library Tools Reference Guide*.

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	`B'1010`
Octal	`1234Q, 1234O, Q'1234, O'1234`
Decimal	`1234, -1, D'1234`
Hexadecimal	`0FFFFH, 0xFFFF, H'FFFF, X'FFFF`

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

ASCII CHARACTER CONSTANTS

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

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

REAL NUMBER CONSTANTS

The M32C Assembler will accept real numbers as constants and convert them into IEEE single-precision (signed 32-bit) real-number format.

Floating-point numbers can be written in the format:

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

Some valid examples are as follows:

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

Spaces and tabs are not allowed in real constants.

Note: Floating-point numbers will not give meaningful results when used in expressions.

PREDEFINED SYMBOLS

The M32C 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.

Symbol	*Value*
`__DATE__`	Current date in `Mmm dd yyyy` format (string).
`__FILE__`	Current source filename (string).
`__IAR_SYSTEMS_ASM__`	IAR assembler identifier (number).
`__LINE__`	Current source line number (number).
`__TID__`	Target identity, consisting of two bytes (number). The low byte is the target identity, which is `48` for the AM32C. The high byte is not used.
`__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).

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 and date of assembly as a string for the program to display:

```
timdat  DC8    __TIME__,",",__DATE__,0
        ...
        MOV.L  timdat,A0        ; load address of string
        JSR    printstring      ; routine to print string
```

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

For example, if you have assembler source files intended for use with different assemblers, you may want to test that the code is appropriate for a specific assembler. You could do this using the `__IAR_SYSTEMS_ASM__` symbol as follows:

```
#ifdef __IAR_SYSTEMS_ASM__
  ...
#else
  ...
#endif
```

REGISTER SYMBOLS

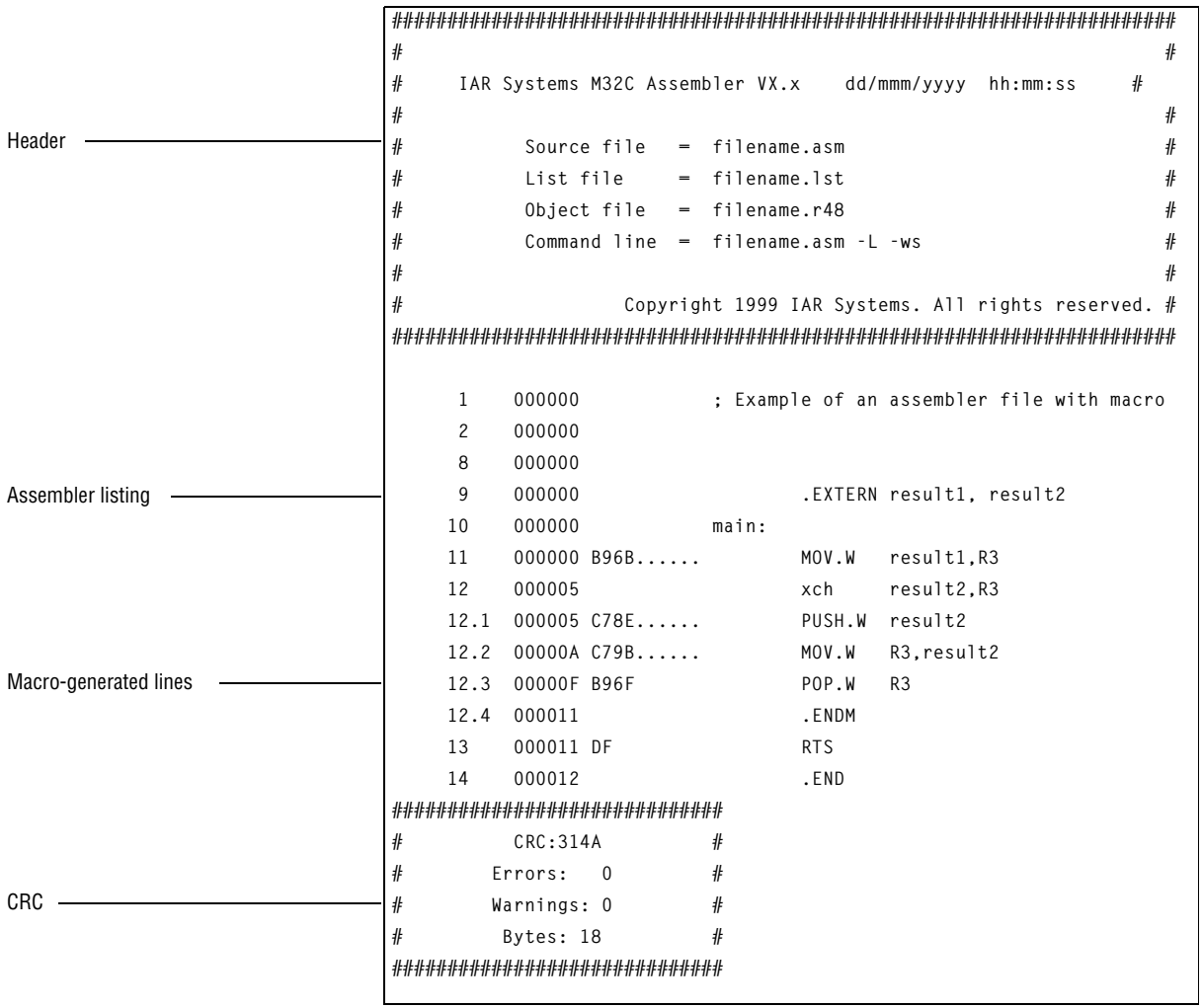
The following table shows the existing predefined register symbols:

Name	*Description*
`R0L, R0H, R1L, R1H`	8-bit byte register (part of word register)
`R0, R1, R2, R3`	16-bit (word) register
`R2R0, R3R1`	32-bit (long) register pair
`R1R2R0`	48-bit register group
`A0, A1`	24-bit address register
`SP, ISP, USP`	24-bit stack pointer register
`SB, FB`	24-bit base register

Name	*Description*
FLG	Status flag register
INTB	Interrupt-table base register
SVP, VCT, SVF	High-speed interrupt registers
DMD*x*, DCT*x* DRC*x*, DMA*x* DSA*x*, DRA*x*	DMAC-related registers, *x* signifies the channel number

LISTING FORMAT

The format of the M32C Assembler listing is as follows:

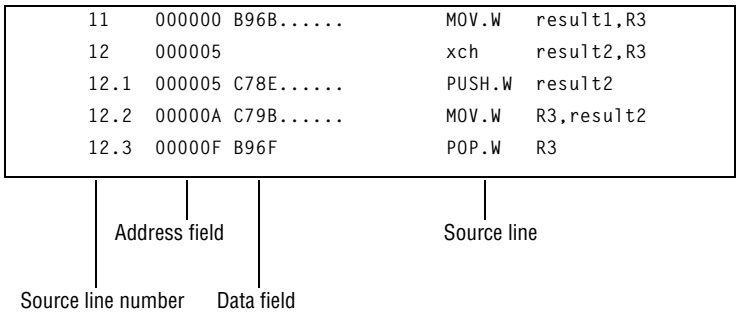



```
#############################################################################
#                                                                           #
#     IAR Systems M32C Assembler VX.x    dd/mmm/yyyy  hh:mm:ss     #
#                                                                           #
#           Source file   =  filename.asm                                   #
#           List file     =  filename.lst                                   #
#           Object file   =  filename.r48                                   #
#           Command line  =  filename.asm -L -ws                            #
#                                                                           #
#                    Copyright 1999 IAR Systems. All rights reserved. #
#############################################################################

      1    000000            ; Example of an assembler file with macro
      2    000000
      8    000000
      9    000000                      .EXTERN result1, result2
     10    000000            main:
     11    000000 B96B......           MOV.W   result1,R3
     12    000005                      xch     result2,R3
     12.1  000005 C78E......           PUSH.W  result2
     12.2  00000A C79B......           MOV.W   R3,result2
     12.3  00000F B96F                 POP.W   R3
     12.4  000011                      .ENDM
     13    000011 DF                   RTS
     14    000012                      .END
################################
#          CRC:314A          #
#        Errors:   0         #
#        Warnings: 0         #
#         Bytes: 18          #
################################
```

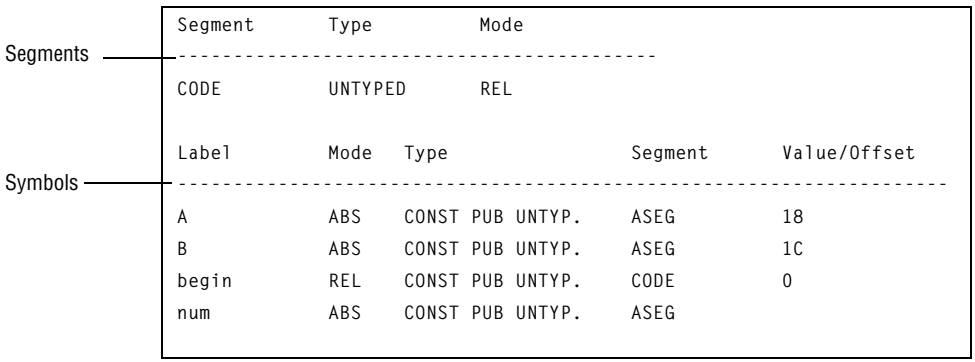
The assembly list contains the following fields of information:

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



SYMBOL AND CROSS-REFERENCE TABLE

If the LSTXRF+ directive has been included, or the option **Include cross reference** (-x) has been specified, a symbol and cross-reference table of the following type will be produced:




```
Segment       Type          Mode
-----------------------------------------
CODE          UNTYPED       REL

Label         Mode   Type                 Segment       Value/Offset
-------------------------------------------------------------------------
A             ABS    CONST PUB UNTYP.     ASEG          18
B             ABS    CONST PUB UNTYP.     ASEG          1C
begin         REL    CONST PUB UNTYP.     CODE          0
num           ABS    CONST PUB UNTYP.     ASEG
```

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

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

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 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 complete 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 (lowest number) 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 to group operators and operands and to control the order in which the expressions are evaluated. For example, the following expression evaluates to 1:

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

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

SUMMARY OF ASSEMBLER OPERATORS

UNARY OPERATORS – 1

+	Unary plus.
-	Unary minus.
NOT (!)	Logical NOT.
LOW	Low byte.
HIGH	High byte.
BYTE1	First byte.
BYTE2	Second byte.
BYTE3	Third byte.
BYTE4	Fourth byte.
LWRD	Low word.
HWRD	High word.
DATE	Current date/time.
SFB	Segment begin.
SFE	Segment end.
SIZEOF	Segment size.
BINNOT (~)	Bitwise NOT.

MULTIPLICATIVE ARITHMETIC OPERATORS – 2

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

ADDITIVE ARITHMETIC OPERATORS – 3

+	Addition.
-	Subtraction.

SHIFT OPERATORS – 4

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

AND OPERATORS – 5

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

OR OPERATORS – 6

`OR (||)`	Logical OR.
`XOR`	Logical exclusive OR.
`BINOR (|)`	Bitwise 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.

The following sections give full descriptions of each assembler operator.

*****

Multiplication (2).

DESCRIPTION

* 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).

DESCRIPTION

Unary plus operator.

EXAMPLES

```
+3 → 3
3*+2 → 6
```

+

Addition (3).

DESCRIPTION

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).

DESCRIPTION

The unary minus operator performs arithmetic negation on its operand.

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

–

Subtraction (3).

DESCRIPTION

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 (2).

DESCRIPTION

/ 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).

DESCRIPTION

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

BINAND(&)

Bitwise AND (5).

DESCRIPTION

Use BINAND to perform bitwise AND between the integer operands.

EXAMPLES

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

BINNOT (~)

Bitwise NOT (1).

DESCRIPTION

Use BINNOT to perform bitwise NOT on its operand.

EXAMPLE

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

BINOR (|)

Bitwise OR (6).

DESCRIPTION

Use BINOR to perform bitwise OR on its operands.

EXAMPLES

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

BINXOR (^)

Bitwise exclusive OR (6).

DESCRIPTION

Use BINXOR to perform bitwise XOR on its operands.

EXAMPLES

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

BYTE1

First byte (1).

DESCRIPTION

BYTE1 takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the unsigned, 8-bit integer value of the lower order byte of the operand.

EXAMPLE

```
BYTE1 0xABCD → 0xCD
```

BYTE2

Second byte (1).

DESCRIPTION

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

EXAMPLE

```
BYTE2 0x12345678 → 0x56
```

BYTE3

Third byte (1).

DESCRIPTION

BYTE3 takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the middle-high byte (bits 23 to 16) of the operand.

EXAMPLE

```
BYTE3 0x12345678 → 0x34
```

BYTE4

Fourth byte (1).

DESCRIPTION

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.

EXAMPLE

```
BYTE4 0x12345678 → 0x12
```

DATE

Current date/time (1).

DESCRIPTION

Use the DATE operator to specify when the current assembly began.

The DATE operator takes an absolute argument (expression) and returns:

DATE 1	Current second (0–59).
DATE 2	Current minute (0–59).
DATE 3	Current hour (0–23).
DATE 4	Current day (1–31).

DATE 5	Current month (1–12).
DATE 6	Current year MOD 100 (1998 →98, 2000 →00, 2002 →02).

EXAMPLE

To assemble the date of assembly:

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

EQ (=, ==)

Equal (7).

DESCRIPTION

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

GE(>=)

Greater than or equal (7).

DESCRIPTION

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

EXAMPLES

```
1 GE 2 → 0
2 GE 1 → 1
1 GE 1 → 1
```

GT (>)

Greater than (7).

DESCRIPTION

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

EXAMPLES

```
-1 GT 1 → 0
2 GT 1 → 1
1 GT 1 → 0
```

HIGH

Second byte (1).

DESCRIPTION

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

EXAMPLE

```
HIGH 0xABCD → 0xAB
```

HWRD (MSW)

High word (1).

DESCRIPTION

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

EXAMPLE

```
HWRD 0x12345678 → 0x1234
```

LE (< =)

Less than or equal (7).

DESCRIPTION

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

EXAMPLES

```
1 LE 2 → 1
2 LE 1 → 0
1 LE 1 → 1
```

LOW

Low byte (1).

DESCRIPTION

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

EXAMPLE

```
LOW 0xABCD → 0xCD
```

LT (<)

Less than (7).

DESCRIPTION

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

EXAMPLES

```
-1 LT 2 → 1
2 LT 1 → 0
2 LT 2 → 0
```

LWRD (LSW)

Low word (1).

DESCRIPTION

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

EXAMPLE

```
LWRD 0x12345678 → 0x5678
```

MOD (%)

Modulo (2).

DESCRIPTION

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

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

EXAMPLES

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

NE (< >, ! =)

Not equal (7).

DESCRIPTION

`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 NE 2 → 1
2 NE 2 → 0
'A' NE 'B' → 1
```

NOT (!)

Logical NOT (1).

DESCRIPTION

Use `NOT` to negate a logical argument.

EXAMPLES

```
NOT B'0101 → 0
NOT B'0000 → 1
```

OR (||)

Logical OR (6).

DESCRIPTION

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

EXAMPLES

```
B'1010 OR B'0000 → 1
B'0000 OR 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 (4).

DESCRIPTION

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 SHL 3 → B'11100000
B'00000111111111111 SHL 5 → B'1111111111100000
14 SHL 1 → 28
```

SHR (> >)

Logical shift right (4).

DESCRIPTION

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 SHR 3 → B'00001110
B'111111111111111 SHR 20 → 0
14 SHR 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).

DESCRIPTION

UGT evaluates to 1 (true) if the left operand has a larger absolute value than the right operand.

EXAMPLES

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

ULT

Unsigned less than (7).

DESCRIPTION

ULT evaluates to 1 (true) if the left operand has a smaller absolute value than the right operand.

EXAMPLES

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

XOR

Logical exclusive OR (6).

DESCRIPTION

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 complete reference information for each category of directives:

SUMMARY OF DIRECTIVES

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

Directive	*Description*	*Section*
`#define`	Assigns a value to a label.	C-style preprocessor
`#elif`	Introduces a new condition in an `#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

Directive	*Description*	*Section*
#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
$	Includes a file.	Assembler control
/*comment*/	C-style comment delimiter.	Assembler control
//	C++ style comment delimiter.	Assembler control
=	Assigns a permanent value local to a module.	Value assignment
.ADDR	Generates 24-bit triple byte constants.	Data definition or allocation directives
.ALIAS	Assigns a permanent value local to a module.	Value assignment
.ALIGN	Aligns the location counter by inserting zero-filled bytes.	Segment control
.ALIGNRAM	Aligns the program counter.	Segment control
.ASEG	Begins an absolute segment.	Segment control
.ASSIGN	Assigns a temporary value.	Value assignment
.BLKA	Allocates space for 24-bit triple byte constants.	Data definition or allocation directives

Directive	*Description*	*Section*
.BLKB	Allocates space for 8-bit bytes.	Data definition or allocation directives
.BLKF	Reserves memory space without initializing for float (32 bits).	Data definition or allocation directives
.BLKL	Allocates space for 32-bit double word constants.	Data definition or allocation directives
.BLKW	Allocates space for 16-bit words.	Data definition or allocation directives
.BYTE	Generates 8-bit byte constants.	Data definition or allocation directives
.CASEOFF	Disables case sensitivity.	Assembler control
.CASEON	Enables case sensitivity.	Assembler control
.COL	Sets the number of columns per page.	Listing control
.COMMON	Begins a common segment.	Segment control
DC8	Generates 8-bit byte constants.	Data definition or allocation
DC16	Generates 16-bit word constants.	Data definition or allocation
DC24	Generates 24-bit triple byte constants.	Data definition or allocation
DC32	Generates 32-bit double word constants.	Data definition or allocation
.DEFINE	Defines a file-wide value.	Value assignment
DS8	Allocates space for 8-bit bytes.	Data definition or allocation

Directive	*Description*	*Section*
DS16	Allocates space for 16-bit words.	Data definition or allocation
DS24	Allocates space for 24-bit triple byte constants.	Data definition or allocation
DS32	Allocates space for 32-bit double word constants.	Data definition or allocation
.ELSE	Assembles instructions if a condition is false.	Conditional assembly
.ELSEIF	Specifies a new condition in an .IF … .ENDIF block.	Conditional assembly
.END	Terminates the assembly of the last module in a file.	Module control
.ENDIF	Ends an .IF block.	Conditional assembly
.ENDM	Ends a macro definition.	Macro processing
.ENDMOD	Terminates the assembly of the current module.	Module control
.ENDR	Ends a repeat structure.	Macro processing
.EQU	Assigns a permanent value local to a module.	Value assignment
.EVEN	Aligns the program counter to an even address.	Segment control
.EXITM	Exits prematurely from a macro.	Macro processing
.EXPORT	Exports symbols to other modules.	Symbol control directives
.EXTERN	Imports an external symbol.	Symbol control
.FLOAT	Initializes float (32-bit) constants.	Data definition or allocation
.IF	Assembles instructions if a condition is true.	Conditional assembly

Directive	*Description*	*Section*
.IFC	Assembles instructions if two strings are equal.	Conditional assembly
.IFNC	Assembles instructions if two strings are not equal.	Conditional assembly
.IMPORT	Initializes float (32-bit) constants.	Symbol control directives
.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
.LWORD	Generates 32-bit double word constants.	Data definition or allocation directives
.MACRO	Defines a macro.	Macro processing
.MODULE	Begins a library module.	Module control
.NAME	Begins a program module.	Module control

Directive	*Description*	*Section*
.ODD	Aligns the program counter to an odd address.	Segment control directives
.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
.PUBWEAK	Exports symbols to other modules; multiple definitions allowed.	Symbol control directives
.RADIX	Sets the default base.	Assembler control
.REPT	Assembles instructions a specified number of times.	Macro processing
.REPTC	Repeats and substitutes characters.	Macro processing
.REPTI	Repeats and substitutes strings.	Macro processing
.REQUIRE	Marks a symbol as required.	Symbol control
.RSEG	Begins a relocatable segment.	Segment control
.RTMODEL	Declares run-time model attributes.	Module control
.SET	Assigns a temporary value.	Value assignment
.SFRTYPE	Specifies SFR attributes.	Value assignment directives
sfr	Creates byte-access SFR labels.	Value assignment directives
sfrp	Creates word-access SFR labels.	Value assignment directives
.STACK	Begins a stack segment.	Segment control

Directive	*Description*	*Section*
.VAR	Assigns a temporary value.	Value assignment
.WORD	Generates 16-bit word constants.	Data definition or allocation directives

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:

```
.LOCAL symbol [,symbol] …
```

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

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

```
.LSTOUT{+ | -}
```

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

LABELS AND COMMENTS

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

```
label .SET expr
```

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

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

PARAMETERS

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

Parameter	*What it consists of*
symbol	An assembler symbol.
label	A symbolic label.
expr	An expression; see *Expressions and operators*, page 22.

The following sections give full descriptions of each category of directives.

MODULE CONTROL DIRECTIVES

Module control directives are used to mark the beginning and end of source program modules, and to assign names and types to them.

Directive	*Description*
.NAME (.PROGRAM)	Begins a program module.
.MODULE (.LIBRARY)	Begins a library module.
.ENDMOD	Terminates the assembly of the current module.
.END	Terminates the assembly of the last module in a file.
.RTMODEL	Declares run-time model attributes.

SYNTAX

```
.NAME symbol [(expr)]
.MODULE symbol [(expr)]
.ENDMOD [label]
.END [label]
.RTMODEL key, value
```

PARAMETERS

symbol	Name assigned to module, used by XLIB when referencing the module.
expr	Optional expression (0–255) used by the IAR C Compiler to encode programming language, memory model, and processor configuration.
label	An expression or label that can be resolved at assembly time. It is output in the object code as a program entry address.
key	A text string specifying the key.
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 also 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.

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 externally defined.

The following rules apply when assembling multi-module files:

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

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

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

Declaring run-time model attributes
Use `.RTMODEL` to enforce compatibility between modules.

All modules that are linked together and that are defining the same run-time attribute key, must have the same value for the corresponding key value, or the special value * (asterisk).

Using the special value * is equivalent to not defining the attribute at all. It can, however, be useful to state explicitly that the module can handle any run-time model.

Each module can have several run-time model definitions.

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

If you are writing assembler routines for use with C code, and you want to control compatibility between modules, refer to the chapter *Assembly Language Interface* in the *M32C IAR C/C++ Compiler Reference Guide.*

EXAMPLES

The following example defines three modules where:

- `MOD_1` and `MOD_2` *cannot* be linked together since they have different values for run-time model **"`foo`"**.
- `MOD_1` and `MOD_3` *can* be linked together since they have the same definition of run-time model "bar" and no conflict in the definition of **"`foo`"**.

- MOD_2 and MOD_3 *can* be linked together since they have no run-time model conflicts. The value "*" matches any run-time model value.

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

.MODULE MOD_2
  .RTMODEL   "foo", "2"
  .RTMODEL   "bar", "*"
  ...
.ENDMOD

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

SYMBOL CONTROL DIRECTIVES

These directives control how symbols are shared between modules.

Directive	*Description*
.PUBLIC (.EXPORT)	Exports symbols to other modules.
.PUBWEAK	Exports symbols to other modules; multiple definitions allowed.
.EXTERN (.IMPORT)	Imports an external symbol.
.REQUIRE	Marks a symbol as required.

SYNTAX

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

PARAMETERS

symbol Symbol to be imported or exported.

DESCRIPTION

Exporting symbols to other modules
Use .PUBLIC to make one or more symbols available to other modules. The symbols declared as .PUBLIC can only be assigned values by using them as labels. .PUBLIC declared symbols 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.

.PUBWEAK is similar to .PUBLIC except that it allows the same symbol to be declared several times. Only one of those declarations will be picked by the linker. All declarations of the symbol must be equivalent.

Importing symbols
Use .EXTERN to import an untyped external symbol.

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

EXAMPLES

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

```
            .NAME    error
            .EXTERN  print
            .PUBLIC  err

errstr      DC8      "** Error **",0
err         PUSH.L   errstr
            JSR      print
            RTS

            .END
```

SEGMENT CONTROL DIRECTIVES

The segment directives control how code and data are generated.

Directive	*Description*
.ASEG	Begins an absolute segment.
.RSEG	Begins a relocatable segment.
.STACK	Begins a stack segment.
.COMMON	Begins a common segment.
.ORG	Sets the location counter.
.ALIGN	Aligns the location counter by inserting zero-filled bytes.
.ALIGNRAM	Aligns the program counter without inserting bytes.
.EVEN	Aligns the program counter to an even address.
.ODD	Aligns the program counter to an odd address.

SYNTAX

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

PARAMETERS

start	A start address that has the same effect as using an .ORG directive at the beginning of the absolute segment.
segment	The name of the segment.
type	The memory type; one of: UNTYPED (the default), CODE, or DATA. In addition, the following types are provided for compatibility with the M32C IAR C Compiler: NEAR, NEARDATA, NEARCONST, NEARCODE FAR, FARDATA, FARCONST, FARCODE HUGE, HUGEDATA, HUGECONST, HUGECODE
flag	NOROOT This segment part may be discarded by the linker if no symbols in this segment part are referred to. Normally all segment parts except startup code and interrupt vectors should set this flag. REORDER Allows the linker to reorder segment parts. For a given segment, all segment parts must specify the same state for this flag. 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.
address	Address where this segment part will be placed.
expr	Address to set the location counter to.
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.
value	Byte value used for padding, default is zero.

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

Setting the location counter
Use `.ORG` to set the 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, that is, it is not valid to use `.ORG 10` during `.RSEG`, since the expression is absolute; instead use `.ORG $+10`. The expression must not contain any forward or external references.

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

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

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

Use `.ALIGNRAM` to align the location counter to a specified address boundary. The expression gives the power of two to which the program counter should be aligned. `.ALIGNRAM` aligns by incrementing the data; no data is generated.

EXAMPLES

Beginning an absolute segment
The following example uses an absolute segment to initialize the reset vector to point to the `main` label, which could indicate the start of a program.

```
.EXTERN main       ; 'main' is a public label
.ASEG
.ORG    0xFFFFFC   ; reset vector location
DC32    main       ; initialize the reset vector
.ENDMOD
```

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

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

```
        .EXTERN  divrtn, mulrtn

        .RSEG    table
        DC16     divrtn, mulrtn
        .ORG     $+6
        DC16     subrtn

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

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

```
        .STACK   rpnstack
parms   DS8      100
opers   DS8      100
        .END
```

The data is allocated from high to low addresses.

Beginning a common segment
The following example defines two common segments containing variables:

```
        .NAME    common1
        .COMMON  data
count   DS32     1
        .ENDMOD

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

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

VALUE ASSIGNMENT DIRECTIVES

These directives are used to assign values to symbols.

Directive	*Description*
`.SET (.VAR, .ASSIGN)`	Assigns a temporary value.
`.EQU (.ALIAS,=)`	Assigns a permanent value local to a module.
`.DEFINE`	Defines a file-wide value.
`.LIMIT`	Checks a value against limits.
`.SFRTYPE`	Specifies SFR attributes.
`sfr`	Creates byte-access SFR labels.
`sfrp`	Creates word-access SFR labels.
`const`	Makes a value read-only.

SYNTAX

```
label .SET expr
label .EQU expr
label = expr
label .DEFINE expr
.LIMIT expr, min, max, message
.SFRTYPE register attribute [,attribute] = value
[const] sfr register = value
[const] sfrp register = value
const value
```

PARAMETERS

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

register A user-defined identifier.

attribute One or more of the following:

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

value The SFR value.

DESCRIPTION

Defining a temporary value
Use .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.

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.

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

message A text message that will be printed when the symbol is out of range.

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

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

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

EXAMPLES

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

```
        .NAME    table
cons    .SET     1

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

main    repeat   4
        .END
```

It generates the following code:

```
   127    000000                          .NAME   table
   128    000001          cons    .SET    1
   129    000000
   136    000000
   137    000000          main:   repeat  4
   137.1  000003          cons    .SET    cons*3
   137.2  000000                          .IF     4>1
   137    000000                          repeat 4-1
```

```
137.1  000009          cons    .SET    cons*3
137.2  000000                          .IF     4-1>1
137    000000                          repeat 4-1-1
137.1  00001B          cons    .SET    cons*3
137.2  000000                          .IF     4-1-1>1
137    000000                          repeat 4-1-1-1
137.1  000051          cons    .SET    cons*3
137.2  000000                          .IF     4-1-1-1>1
137.3  000000                          repeat 4-1-1-1-1
137.4  000000                          .ENDIF
137.5  000000                          .ENDM
137.6  000000                          .ENDIF
137.7  000000                          .ENDM
137.8  000000                          .ENDIF
137.9  000000                          .ENDM
137.10 000000                          .ENDIF
137.11 000000                          .ENDM
138    000000                          .ENDMOD
```

Using local and global symbols

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

```
        .NAME    add1
locn    .DEFINE  0x100
value   .EQU     77
        MOV.W    locn,R0
        ADD.W    #value,R0
        RTS
        .ENDMOD

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

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

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.

SYNTAX

```
.IF condition
.ELSE
.ELSEIF
.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. Conditional assembler directives may be used anywhere in an assembly, but have their greatest use in conjunction with macro processing.

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

.IF … .ENDIF and .IF … .ELSE … .ENDIF blocks may be nested to any level.

EXAMPLES

The following macro adds a byte constant to any location. The a should be tied to the byte constant and the c should be tied to the location:

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

If the argument to the macro is 1 it generates an INC instruction to save instruction cycles; otherwise it generates an ADD instruction. It could be tested with the following program:

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

MACRO PROCESSING DIRECTIVES

These directives allow user macros to be defined.

Directive	*Description*
.MACRO	Defines a macro.
.ENDM	Ends a macro definition.
.EXITM	Exits prematurely from a macro.
.LOCAL	Creates symbols local to a macro.

Directive	*Description*
.REPT	Assembles instructions a specified number of times.
.REPTC	Repeats and substitutes characters.
.REPTI	Repeats and substitutes strings.
.ENDR	Ends a repeat structure.

SYNTAX

```
name .MACRO [argument] …
.ENDM
.EXITM
.LOCAL symbol [,symbol] …
.REPT expr
.REPTC formal,actual
.REPTI formal,actual [,actual] …
.ENDR
```

PARAMETERS

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

DESCRIPTION

A macro is a user-defined symbol that represents a block of one or more assembler source line. 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.

Although macros effectively perform simple text substitution, 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
        JSR      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'
```

The assembler will expand this to:

```
        JSR      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
        JSR      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 macro quote characters in the macro call.

For example:

```
macld       .MACRO      two_op
            MOV.W       two_op
            .ENDM
```

It could be called using:

```
macld_main:
            macld       <#1,R0>
            .END
```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 12.

Predefined macro symbols
The symbol `_args` is set to the number of arguments passed to the macro.

How macros are processed
There are three distinct phases in the macro process:

- The assembler performs scanning and saving of macro definitions. The text between `.MACRO` and `.ENDM` is saved but not syntax-checked. Include-file references `$file` are recorded and will be included during macro *expansion*.
- A macro call forces the assembler to invoke the macro processor (expander) which switches (if not already in a macro) the assembler input stream from a source file to the output from the macro expander (which takes its input from the requested macro definition).

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

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

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

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

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 subroutine outputs bytes from a buffer to a port:

```
        .NAME   play
sfr     IO_port=0x3E0

        .RSEG   data
buffer  DC8     512                     //buffer

        .RSEG   code
play:   MOV.W   #buffer,A0
loop:   MOV.B   [A0],IO_port
        INC.W   A0
        CMP.W   #buffer+512,A0
        JNE     loop
        RTS
        .END
```

The main program calls this routine as follows:

```
        JSR     play
```

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

```
        .NAME   play
sfr     IO_port=0x3E0

        .RSEG   data
buffer  DC8     512
        .RSEG   code
play:   MOV.W   #buffer,A0
loop:   MOV.B   [A0],IO_PORT
        INC.W   A0
        CMP.W   #buffer+512,A0
        JNE     loop
        RTS
        .ENDMOD
        .END
```

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

To use in-line code the main program is then simply altered to:

```
        play    buffer
```

Using .REPTC and .REPTI

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

```
       .NAME   retc1
       .EXTERN plotc
banner .REPTC  chr,  "Welcome"
       MOV.B   #'chr',R0L
       JSR     plotc
       .ENDR
       .ENDM
```

This produces the following code:

```
   238    000000                  .NAME   retc1
   239    000000                  .EXTERN plotc
```

```
240    000000            banner  .REPTC  chr, "Welcome"
241    000000                    MOV.B   #'chr',R0L
242    000000                    JSR             plotc
243    000000                    .ENDR
243.1  000000 0457               MOV.B   #'W',R0L
243.2  000002 CD......           JSR             plotc
243.3  000006 0465               MOV.B   #'e',R0L
243.4  000008 CD......           JSR             plotc
243.5  00000C 046C               MOV.B   #'l',R0L
243.6  00000E CD......           JSR             plotc
243.7  000012 0463               MOV.B   #'c',R0L
243.8  000014 CD......           JSR             plotc
243.9  000018 046F               MOV.B   #'o',R0L
243.10 00001A CD......           JSR             plotc
243.11 00001E 046D               MOV.B   #'m',R0L
243.12 000020 CD......           JSR             plotc
243.13 000024 0465               MOV.B   #'e',R0L
243.14 000026 CD......           JSR             plotc
244    00002A                    .ENDMOD
```

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

```
        .NAME   retc2

        .EXTERN base,count,init
banner  .REPTI  adds,base,count,init
        MOV.W   #0,adds
        .ENDR
        .ENDM
```

This produces the following code:

```
   250    000000                      .NAME   retc2
   251    000000                      .EXTERN base,count,init
   252    000000             banner   .REPTI  adds,base,count,init
   253    000000                      MOV.W   #0,adds
   254    000000                      .ENDR
   254.1  000000 F7A0......           MOV.W   #0,base
   254.2  000005 F7A0......           MOV.W   #0,count
   254.3  00000A F7A0......           MOV.W   #0,init
   255    00000F                      .ENDMOD
```

LISTING CONTROL DIRECTIVES

These directives provide control over the assembler listing.

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

Directive	*Description*
.PAGE	Generates a new page.

SYNTAX

```
.LSTCND{+ | -}
.LSTCOD{+ | -}
.LSTEXP{+ | -}
.LSTMAC{+ | -}
.LSTOUT{+ | -}
.LSTPAG{+ | -}
.LSTREP{+ | -}
.LSTXRF{+ | -}
.COL columns
.PAGSIZ lines
.PAGE
```

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 list 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 list 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.

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

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 assembly 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.

Formatting listed output
Use .COL to set the number of columns per page of the assembly list. The default number of columns is 80.

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

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

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

Use .PAGE to generate a new page in the assembly listing if paging is active.

EXAMPLES

Turning the listing on or off

To disable the listing of a debugged section of program:

```
        .NAME lstcndtst
        .LSTOUT-
; Debugged section, needs no listing
        .LSTOUT+
; Not yet debugged
```

Listing conditional code and strings

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

```
        .NAME    lstcndtst
        .EXTERN  print
        .RSEG    prom
debug   .SET     0
begin1:
        .IF      debug
                 .JSR     print
        .ENDIF
        .LSTCND+
begin2
        .IF      debug
                 .JSR     print
        .ENDIF
        .END
```

This will generate the following listing:

```
   269    000000                   .NAME   lstcndtst
   270    000000                   .EXTERN print
   271    000000                   .RSEG   prom
   272    000000           debug   .SET    0
   273    000000           begin1:
   274    000000                   .IF     debug
   275    000000                           JSR     print
   276    000000                   .ENDIF
   277    000000                   .LSTCND+
   278    000000           begin2:
   279    000000                   .IF     debug
   281    000000                   .ENDIF
```

```
   282    000000                    .ENDMOD
```

The following example shows the effect of `.LSTCOD-` on the code generated by a `DC8` directive:

```
        .NAME    lstcodtbl
table1  DC8      1,2,3,4,5,6
        .LSTCOD-
table2  DC8      1,2,3,4,5,6
        .END
```

This will produce the following output:

```
   286    000000                    .NAME   lstcodtbl
   287    000000                    .LSTCOD+
   288    000000 0102030405 table1  DC8     1,2,3,4,5,6
                 06
   289    000006                    .LSTCOD-
   290    000006 0102030405*table2  DC8     1,2,3,4,5,6
   291    00000C                    .ENDMOD
```

Controlling the listing of macros

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

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

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

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

This will produce the following output:

```
296    000000                    .MODULE EX_9
297    000000
302    000000
303    000000                    .LSTMAC+
304    000000            inc2    .MACRO  arg
305    000000                            INC.B   arg
306    000000                            INC.B   arg
307    000000                            .ENDM
308    000000
309    000000                    .EXTERN memlock
310    000000            begin:
311    000000                            dec2    memlock
311.1  000000 B68E......                  DEC.B   memlock
311.2  000005 B68E......                  DEC.B   memlock
311.3  00000A                             .ENDM
312    00000A                    .LSTEXP-
313    00000A                    inc2    memlock
314    000014 DF                 RTS
315    000015                    .ENDMOD
```

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.
#undef	Undefines a label.
#if	Assembles instructions if a condition is true.
#ifdef	Assembles instructions if a symbol is defined.
#ifndef	Assembles instructions if a symbol is undefined.
#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.
#include	Includes a file.
#message	Generates a message on standard output.
#error	Generates an error.
/*comment*/	C-style comment delimiter.
//	C++ style comment delimiter.

SYNTAX

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

PARAMETERS

label	Symbol to be defined, undefined, or tested.	
text	Value to be assigned.	
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.	
message	Text to be displayed.	

DESCRIPTION

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

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

```
#define five 5 ; comment
MOV.W five+addr,R0      ; syntax error!
   ; expanded to "MOV.W 5 ; comment+addr,R0"
MOV.W R0,five+addr      ; incorrect code!
   ; expanded to "MOV.W R0,5 ; comment+addr"
```

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.

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

```
/*
Program to read serial input.
Version 2: 11.1.99
Author: mjp
*/
```

EXAMPLES

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

```
        .NAME    ifdef
        .EXTERN  input, output

#define adjust   10

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

#ifdef  adjust
        ADD.W    adjust,R0
#else
#error  "'adjust' not defined"
#endif
```

```
#undef  adjust
        MOV.W    [A0],R0
        RTS
        .END
```

Including a source file

The following example uses #include to include a file defining macros into the source file. For example, the following macros could be defined in xchmacro.s48:

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

xch_w   .MACRO   a,b
        PUSH.W   a
        MOV.W    b,a
        POP.W    b
        .END
```

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

```
        .NAME  include1
        .EXTERN result1,result2
#include “xchmacro.s48”

inc_main:
        xch_w   result1, result2
        xch_b   result1, result2
        xch_w   result1, result2
        .END
```

DATA DEFINITION OR ALLOCATION DIRECTIVES

These directives define temporary values or reserve memory.

Directive	*Description*
DC8 (.BYTE)	Generates 8-bit byte constants.
DC16 (.WORD)	Generates 16-bit word constants.
DC24 (.ADDR)	Generates 24-bit 3-byte constants.
DC32 (.LWORD)	Generates 32-bit double word constants.
DS8 (.BLKB)	Allocates space for 8-bit bytes.
DS16 (.BLKW)	Allocates space for 16-bit words.
DS24 (.BLKA)	Allocates space for 24-bit 3-byte constants.
DS32 (.BLKL)	Allocates space for 32-bit double word constants.
.BLKF	Reserves memory space for float (32-bit) without initializing.
.FLOAT	Initializes float (32-bit) constants.

SYNTAX

```
DC8 expr [,expr] ...
DC16 expr [,expr] ...
DC24 expr [,expr] ...
DC32 expr [,expr] ...
DS8 expr [,expr] ...
DS16 expr [,expr] ...
DS24 expr [,expr] ...
DS32 expr [,expr] ...
.BLKF expr [,expr] ...
.FLOAT 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.

DESCRIPTION

Use DS8, DS16, DS24, DS32, and .BLKF to allocate space. The memory contents are not initialized in any way.

Use DC8, DC16, DC24, DC32, and .FLOAT to initialize and reserve memory space.

EXAMPLES

Generating lookup table

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

```
        .NAME    table

table   DC16     addsubr, subsubr, clrsubr
addsubr ADD.W    R0,R1
        RTS

subsubr SUB.W    R0,R1
        RTS

clrsubr MOV.W    #0,R0
        RTS

        .END
```

Defining strings

To define a string:

```
mymess     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:

```
errmess    DC8 'Don''t understand!'
```

Reserving space

To reserve space for ten bytes:

```
table      DS8  0xA
```

ASSEMBLER CONTROL DIRECTIVES

These directives provide control over the operation of the assembler.

Directive	*Description*
`$`	Includes a file.
`.RADIX`	Sets the default base.
`.CASEON`	Enables case sensitivity.
`.CASEOFF`	Disables case sensitivity.

SYNTAX

```
$filename
.RADIX expr
.CASEON
.CASEOFF
```

PARAMETERS

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

DESCRIPTION

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

Use `.RADIX` to set the default base for use in conversion of constants from ASCII source to the internal binary format.

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

```
.RADIX 0x0A
```

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 macros.s48:

```
xch     .MACRO   add1,add2
        PUSH.B   add1
        MOV.B    add1,add2
        POP.B    add2
        .ENDM
```

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

```
        .NAME  include
; Standard macro definitions
$macros.s48
; Program
        .EXTERN  var1,var2
main    exch     var1, var2
        RTS
        .END
```

Changing the base
To set the default base to 16:

```
     .RADIX 16D
     MOV.W  #12,A0
```

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

Controlling case sensitivity
When .CASEOFF is set, label and LABEL are identical in the following example:

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

The following will generate a duplicate label error:

```
label      NOP
LABEL      NOP   ; Error: "LABEL" already defined
           .END
```

ASSEMBLER DIAGNOSTICS

This chapter lists the error and warning messages for the M32C Assembler.

INTRODUCTION

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

All errors are issued as complete, self-explanatory messages. The error message consists of the erroneous source line, with a pointer to the faulty spot, followed by the source line number and diagnostics. If include files are used, error messages will be preceded by the source line number and name of *current* file:

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

The error messages produced by the assembler fall into the following categories:

- Command line error messages.
- Assembly warning messages.
- Assembly error messages.
- Assembly fatal error messages.
- Memory overflow messages.
- Assembler internal error messages.

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, mis-spelled, or missing command line switches.

ASSEMBLY ERROR MESSAGES

Assembly error messages are produced when the assembler has found a construct which violates the language rules. These are listed in the section *Error messages*, page 97.

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. These are listed in the section *Warning messages*, page 107.

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. The fatal error messages are identified as `Fatal` in the error messages list.

MEMORY OVERFLOW MESSAGES

The assembler is a memory-based program that, in the case of a system with a small primary memory or in the case of very large source files, may run out of memory. This is identified by the special message:

```
* * * ASSEMBLER OUT OF MEMORY * * *
Dynamic memory used: nnnnnn bytes
```

If such a situation occurs, the solution is either to add system memory or to split source files into smaller modules.

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 and should be reported to your software distributor or to IAR Technical Support. Please include information enough to reproduce the problem. This would typically include:

- The 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.
- Version number of the M32C IAR Assembler.

ERROR MESSAGES

GENERAL

The following section lists the general error messages.

0 Invalid syntax

The assembler could not decode the expression.

1 Too deep #include nesting (max. is 10)

Fatal. The assembler limit for nesting of `#include` files was exceeded. A recursive `#include` could be the reason.

2 Failed to open #include file < name >

Fatal. Could not open a `#include` file. The file does not exist in the specified directories. Check the `-I` prefixes.

3 Invalid #include file name

Fatal. A `#include` file name must be written `<file>` or `"file"`.

4 Unexpected end of file encounted

Fatal. End of file encountered within a conditional assembly, the repeat directive, or during macro expansion. A probable cause is the missing end of conditional assembly.

5 Too long source line (max. is 2048 characters) truncated

The source line length exceeds the assembler limit.

6 Bad constant

A character that is not a legal digit was encountered.

7 Hexadecimal constant without digits

The prefix `0x` or `0X` of a hexadecimal constant found without any hexadecimal digits following.

8 Invalid floating point constant

A too large floating-point constant or invalid syntax of floating-point constant was encountered.

9 Too many errors encountered (> 100).

The maximum number of errors can be set using the command line option -E; see *-E*, page 9.

10 Space or tab expected

11 Too deep block nesting (max is 50)

The preprocessor directives are nested too deeply.

12 String too long (max is 2045)

The assembler string length limit was exceeded.

13 Missing delimiter in literal or character constant

No closing delimiter `'` or `"` was found in character or literal constant.

14 Missing #endif

A `#if`, `#ifdef`, or `#ifndef` was found but had no matching `#endif`.

15 Invalid character encountered: *char*; ignored

16 Identifier expected

A name of a label or symbol was expected.

17 ')' expected

18 No such pre-processor command: *command*

was followed by an unknown identifier.

19 Unexpected token found in pre-processor line

The preprocessor line was not empty after the argument part was read.

20 Argument to #define too long (max is 2048)

21 Too many formal parameters for #define (max is 37)

22 Macro parameter *parameter* redefined

A `#define` symbol's formal parameter was repeated.

23 ',' or ')' expected

24 Unmatched #else, #endif or #elif

Fatal. Missing `#if`, `#ifdef`, or `#ifndef`.

25 #error < error > .

Printout via the `#error` directive.

26 '(' expected

27 Too many active macro parameters (max is 256)

Fatal. Preprocessor limit exceeded.

28 Too many nested parameterized macros (max is 50)

Fatal. Preprocessor limit exceeded.

29 Too deep macro nesting (max is 100)

Fatal. Preprocessor limit exceeded.

30 Actual macro parameter too long (max is 512)

A single macro (in `#define`) argument may not exceed the length of a source line.

31 Macro < macro > called with too many parameters

The number of parameters used was greater than the number in the macro declaration.

32 Macro < macro > called with too few parameters

The number of parameters used was less than the number in the macro declaration (`#define`).

33 Too many MACRO arguments

The number of assembler macros exceeds 32.

34 May not be redefined

Assembler macros may not be redefined.

35 No name on macro

An assembler macro definition without a label was encountered.

36 Illegal formal parameter in macro

A parameter that was not an identifier was found.

37 ENDM or EXITM not in macro

An `ENDM` directive or `EXITM` directive encountered outside a macro.

38 '>' expected but found end-of-line

A < was found but no matching >.

39 END before start of module

The end-of-module directive has no matching `MODULE` directive.

40 Bad instruction

The mnemonic/directive does not exist.

41 Bad label

Labels must begin with A-Z, a-z, _, or ?. The succeeding characters must be A-Z, a-z, 0-9, _, or ?. Labels cannot have the same name as a predefined symbol.

42 Duplicate label

The label has already appeared in the label field or has been declared as EXTERN.

43 Illegal effective address

The addressing mode (operands) is not allowed for this mnemonic.

44 ',' expected

A comma was expected but not found.

45 Name duplicated

The name of RSEG, STACK, or COMMON segments is already used but for something else.

46 Segment type expected

In RSEG, STACK, or COMMON directive : was found but the segment type that should follow was not valid.

47 Segment name expected

The RSEG, STACK, and COMMON directives need a name.

48 Value out of range *range*

The value exceeds its limits.

49 Alignment already set

RSEG, STACK, and COMMON segment do not allow alignment to be set more than once. Use ALIGN, EVEN, or ODD instead.

50 Undefined symbol: *symbol*

The symbol did not appear in label field or in an `EXTERN` or `sfr` declaration.

51 Can't be both PUBLIC and EXTERN

Symbols can be declared as either `PUBLIC` or `EXTERN`.

52 EXTERN not allowed

Reference to `EXTERN` symbols is not allowed in this context.

53 Expression must be absolute

The expression cannot involve relocatable or external symbols.

54 Expression can not be forward

The assembler must be able to solve the expression the first time this expression is encountered.

55 Illegal size

The maximum size for expressions is 32 bits.

56 Too many digits

The value exceeds the size of the destination.

57 Unbalanced conditional assembly directives

Missing conditional assembly `IF` or `ENDIF`.

58 ELSE without IF

Missing conditional assembly `IF`.

59 ENDIF without IF

Missing conditional assembly `IF`.

60 Unbalanced structured assembly directives

Missing structured assembly `IF` or `ENDIF`.

61 '+' or '-' expected

A plus or minus sign is missing.

62 Illegal operation on extern or public symbol

An illegal operation has been used on a public or external symbol; eg `SET`.

63 Illegal operation on non-constant label

It is illegal to make a non-constant symbol `PUBLIC` or `EXTERN`.

64 Extern or unsolved expression

The expression must be solved at assembly time, i.e. not include external references.

65 '=' expected

Equals sign was missing.

66 Segment too long (max is *max*)

The length of `ASEG`, `RSEG`, `STACK`, or `COMMON` segments is larger than the addressable length.

67 Public did not appear in label field

A symbol was declared `PUBLIC` but no label with the same name was found in the source file.

68 End of block-repeat without start

The repeat directive `REPT` was not found although the `ENDR` directive was.

69 Segment must be relocatable

The operation is not allowed on `ASEG`.

70 Limit exceeded: *error text*, value is: *value* (decimal)

The value exceeded the limits set with the LIMIT directive. The error text is set by the user in the LIMIT directive.

71 Symbol *symbol* has already been declared EXTERN

An attempt to redeclare an EXTERN as EXTERN was made.

72 Symbol *symbol* has already been declared PUBLIC

An attempt to redeclare a PUBLIC as PUBLIC was made.

73 End-of-module missing

A PROGRAM or MODULE directive was encountered before ENDMOD was found.

74 Expression must yield non-negative result

The expression was evaluated to a negative number, whereas a positive number was required.

75 Repeat directive unbalanced

This error is caused by a REPT directive without a matching ENDR, or a an ENDR directive without a matching REPT.

76 End of repeat directive is missing

A REPT directive without a closing ENDR was encountered.

77 LOCALs not allowed in this context, (*symbol*)

Local symbols must be declared within macro definitions.

78 End of macro expected

An assembler macro is being defined but there was no end-of-macro.

79 End of repeat expected

One of the repeat directives is active, but there was no end-of-repeat found.

80 End of conditional assembly expected

Conditional assembly is active but there was no end of if.

81 End of structured assembly expected

One of the directives for structured assembly is active but has no matching `END`.

82 Misplaced end of structured assembly

A directive that terminates one of the structured assembly directives was found but no matching `START` directive is active.

83 Error in SFR attribute definition

The `SFRTYPE` directive was used with unknown attributes.

84 Illegal symbol type in symbol

The symbol cannot be used in this context since it has the wrong type.

85 Wrong number of arguments

Expected a different number of arguments.

86 Number expected

Characters other than digits were encountered.

87 Label must be public or extern

The label must be declared with `PUBLIC` or `EXTERN`.

88 Label not defined with DEFFN

The label has to be defined via `DEFFN` before used in this context.

89 Sorry DEMO version, bytecount exceeded (*max bytes*)

90 Different parts of ASEG have overlapping code

91 Internal error

92 Empty macro stack overflow

93 Macro stack overflow

94 Attempt to access out-of-stack value

95 Invalid macro operator

96 No such macro argument

97 Sorry Lite version, bytecount exceeded (*max bytes*)

98 Option -re cannot handle code in include files, use -r or -rn instead

99 #include within macro not supported

100 Duplicate segment definitions

Segment redefinition with different attributes; for example, an `RSEG` segment cannot be used as a `COMMON` segment.

M32C-SPECIFIC ERROR MESSAGES

In addition to the general errors, the M32C assembler may generate the following errors:

400 Branch too long

401 Too many operands

402 :8 or :16 expected

403 :8, :16 or :24 expected

404 :16 or :24 expected

405 :11, :19 or :27 expected

406 :19 or :27 expected

407 Size specifier (.B .W etc) required

408 The register *register* is not allowed here

409 Illegal flag-register flag

410 Size specifier not compatible with operand

WARNING MESSAGES

GENERAL

The following section lists the general warning messages.

0 Unreferenced label

The label was not used as an operand, nor was it declared public.

1 Nested comment

A C comment was nested.

2 Unknown escape sequence

A backslash (\) found in a character constant or string literal was followed by an unknown escape character.

3 Non-printable character

A non-printable character was found in a literal or character constant.

4 Macro or define expected

5 Floating point value out-of-range

Floating point value is too large to be represented by the floating point system of the target.

6 Floating point division by zero

7 Wrong usage of string operator (# or ##); ignored.

The current implementation restricts use of the # and ## operators to the token field of parameterized macros. In addition, the # operator must precede a formal parameter.

8 Macro parameter(s) not used

9 Macro redefined

10 Unknown macro

11 Empty macro argument

12 Recursive macro

13 Redefinition of Special Function Register

The special function register (SFR) has already been defined.

14 Division by zero

Division by 0 in constant expression.

15 Constant truncated

The constant was longer than the size of the destination.

16 Suspicious sfr expression

A special function register (SFR) is used in an expression, and the assembler cannot check access rights.

17 Empty module *module*, module skipped

An empty module was created by using END directly after ENDMOD or MODULE, followed by ENDMOD without any statements in between.

18 End of program while in include file

The program ended while a file was being included.

19 Symbol *symbol* duplicated

20 Bit symbol cannot be used as operand

A symbol was declared using the bit directive, but since the bit address is not calculated the symbol should not be used.

21 Label did not appear in label field

22 Set segment alignment the same *value* or larger

When the alignment set by ALIGN is larger than the segment alignment it may be lost at link time.

M32C-SPECIFIC WARNING MESSAGES

In addition to the general warnings, the M32C IAR Assembler may generate the following warnings:

400 Number out of range

401 SFR neither defined as READ nor WRITE

402 More than one SFR size attribute defined, using default (byte)

403 No SFR size attribute defined, using default (byte)

404 Displacement out of bounds

405 Accessing SFR incorrectly, check read/write flags

406 Accessing SFR using incorrect size

407 :8 applied, ignoring upper byte

408 :16 applied

409 Illegal register

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W

X

Symbols