PIC18 IAR Assembler

Reference Guide

for Microchip® Technology Inc's

PIC18 Microcontroller

EDITION NOTICE
Second edition: March 2002

Part number: APIC18-2

This guide replaces the PIC18 IAR Assembler Reference Guide, part number APIC18-1.

Contents

Tables

Preface

Welcome to the PIC18 IAR Assembler Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use the PIC18 IAR Assembler to best suit your application requirements.

Who should read this guide

You should read this guide if you plan to develop an application using assembler language for the PIC18 microcontroller and need to get detailed reference information on how to use the PIC18 IAR Assembler. In addition, you should have working knowledge of the following:

- The architecture and instruction set of the PIC18 microcontroller. Refer to the documentation from Microchip® for information about the PIC18 microcontroller
- General assembler language programming
- Application development for embedded systems
- The operating system of your host machine.

How to use this guide

When you first begin using the PIC18 IAR Assembler, you should read the *Introduction to the IAR Assembler* chapter in this reference guide.

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

If you are new to using the IAR toolkit, we recommend that you first read the initial chapters of the *PIC18 IAR Embedded Workbench™ User Guide*. They give product overviews, as well as tutorials that can help you get started.

What this guide contains

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

- *Introduction to the IAR Assembler* provides programming information. It also describes the source code format, and the format of assembler listings.
- *Assembler options* first explains how to set the assembler options from the command line and how to use environment variables. It then gives an alphabetical summary of the assembler options, and contains detailed reference information about each option.
- *Assembler operators* gives a summary of the assembler operators, arranged in order of precedence, and provides detailed reference information about each operator.

- *Assembler directives* gives an alphabetical summary of the assembler directives, and provides detailed reference information about each of the directives, classified into groups according to their function.
- *#pragma directives* describes the `#pragma` directives available in the assembler.
- *Assembler diagnostics* contains information about the formats and severity levels of diagnostic messages.

Other documentation

The complete set of IAR Systems development tools for the PIC18 microcontroller is described in a series of guides. For information about:

- Using the IAR Embedded Workbench™ and the IAR C-SPY™ Debugger, refer to the *PIC18 IAR Embedded Workbench™ User Guide*
- Programming for the PIC18 IAR C Compiler, refer to the *PIC18 IAR C Compiler Reference Guide*
- Using the IAR XLINK Linker™ and the IAR XLIB Librarian™, refer to the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide*.
- Using the IAR C Library, refer to the *IAR C Library Functions Reference Guide*

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

Document conventions

This guide uses the following typographic conventions:

Style	Used for
`computer`	Text that you enter or that appears on the screen.
`parameter`	A label representing the actual value you should enter as part of a command.
`[option]`	An optional part of a command.
`{a | b | c}`	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
reference	A cross-reference within or to another part of this guide.
	Identifies instructions specific to the versions of the IAR Systems tools for the IAR Embedded Workbench interface.
	Identifies instructions specific to the command line versions of IAR Systems development tools.

Table 1: Typographic conventions used in this guide

Introduction to the IAR Assembler

This chapter describes the source code format for the PIC18 IAR Assembler. It also provides programming hints for the assembler and describes the format of assembler list files.

Refer to the hardware documentation from Microchip® for syntax descriptions of the instruction mnemonics.

Source format

The format of an assembler source line is as follows:

```
[label[:]] [operation] [operand] [; 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. See also *--mnem_first*, page 18.
operand	An assembler instruction can have zero, one, two, or three operands. The data definition directives, for example `DB` and `DC8`, can have any number of operands. For reference information about the data definition directives, see *Data definition or allocation directives*, page 74. Other assembler directives can have one, two, or three operands, separated by commas.
comment	Comment, preceded by a `;` (semicolon).

The fields can be separated by spaces or tabs. A source line can be of unlimited length.

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

The PIC18 IAR Assembler recognizes the default file extensions `s49`, `asm`, and `msa` for source files.

Assembler instructions

The PIC18 IAR Assembler supports the syntax for assembler instructions as described in the *Microchip PIC18CXX2 Data Sheet*. It complies with the requirement of the PIC18 architecture on word alignment. Any instructions in a code segment placed on an odd address will result in an error.

Assembler expressions

Assembler expressions can consist of operands and operators.

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

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators. For more information, see *Precedence of operators*, page 21.

The following operands are valid in an expression:

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

These are described in greater detail in the following sections.

The valid operators are described in the chapter *Assembler operators*, page 21.

TRUE AND FALSE

In expressions a zero value is considered FALSE, and a non-zero value is considered TRUE.

Conditional expressions return the value 0 for FALSE and 1 for TRUE.

EXPRESSION RESTRICTIONS

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

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

No forward

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

No external

No external references in the expression are allowed.

Absolute

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

Fixed

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

USING SYMBOLS IN RELOCATABLE EXPRESSIONS

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

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

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

```
        EXTERN  third
        RSEG    DATA
first:  DS      5
second: DS      3
        ENDMOD
        RSEG    CODE
start   …
```

Then in the segment `CODE` the following instructions are legal:

```
        INCF     first+7,1
        INCF     first-7,1
        INCF     7+first,1
        INCF     (first/second)*third,1
```

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

SYMBOLS

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

Symbols must begin with a letter, a–z or A–Z, ? (question mark), or _ (underscore). Symbols can include the digits 0–9 and `$` (dollar). (Note, however, that the preprocessor does not allow symbols to include a `$`.)

For built-in symbols like instructions, registers, operators, and directives case is insignificant. For user-defined symbols case is by default significant but can be turned on and off using the **User symbols are case sensitive** (`--case_insensitive`) assembler option; see page 12 for additional information. You can also use the assembler directives `CASEON` and `CASEOFF` to control case sensitivity for user-defined symbols; see *Assembler control directives*, page 78, for more information.

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

LABELS

Symbols used for memory locations are referred to as labels.

Program location counter (PLC)

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

```
GOTO   A:$      ; Loop forever
```

INTEGER CONSTANTS

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

Constants are written as a sequence of digits with an optional - (minus) sign in front to indicate a negative number. Commas and decimal points are not permitted.

The following types of number representation are supported:

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

Table 2: Integer constant formats

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

ASCII CHARACTER CONSTANTS

ASCII constants consist of zero or 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'`	The string `ABCD` (four characters).
`"ABCD"`	The string `ABCD'\0'` (five characters the last ASCII null).
`'A''B'`	The string `A'B`
`'A'''`	The string `A'`
`''''` (4 quotes)	The character constant `'`
`''` (2 quotes)	Empty string (no value).
`""`	Empty string (an ASCII null character).
`\'`	`'` (as a character constant or character inside a string)
`\\`	`\` (as a character constant or character inside a string)

Table 3: ASCII character constant formats

FLOATING-POINT CONSTANTS

The PIC18 IAR Assembler will accept floating-point values as constants and convert them into IEEE single-precision (signed 32-bit) floating-point format or fractional format.

Floating-point numbers can be written in the format:

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

The following table shows some valid examples:

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

Table 4: Floating-point constants

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

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

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

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

For information about DQ15, see *Data definition or allocation directives*, page 74.

PREDEFINED SYMBOLS

The PIC18 IAR Assembler defines a set of symbols for use in assembler source files. The symbols provide information about the current assembly, allowing you to test them in preprocessor directives or include them in the assembled code. The strings returned by the assembler are enclosed in double quotes.

Symbol	Value
__APIC18__	PIC18 IAR Assembler identifier (number). The current identifier is 1.
__DATE__	Date of assembly in Mmm dd yyyy format (string).
__FILE__	Current source filename (string).
__IAR_SYSTEMS_ASM__	IAR assembler identifier (number). The current identifier is 2.
__LINE__	Current source line number (number).
__TID__	Target identity, consisting of two bytes (number). The high byte is the target identity, which is 49 for APIC18. The low byte is the processor option 00.
__TIME__	Time of assembly in hh:mm:ss format (string).
__VER__	Version number in integer format; for example, version 4.17 is returned as 417 (number).

Table 5: Predefined symbols

Note: __TID__ is related to the predefined symbol __TID__ in the PIC18 IAR C Compiler, which is described in the *PIC18 IAR C Compiler Reference Guide*.

Including symbol values in code

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

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

```
timdat: DT    __TIME__,",",__DATE__,0   ;time and date
```

Testing symbols for conditional assembly

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

For example, you may want to test the target identifier to verify that the code is assembled for the proper target processor. You could do this using the `__TID__` symbol as follows:

```
#define TARGET ((__TID__& 0xFF00)>>8)
#if (TARGET==49)
…
…    (code for PIC18CXXX)
…
#else
…
…    (code for another chip)
…
#endif
```

Programming hints

This section gives hints on how to write efficient code for the PIC18 IAR Assembler. For information about projects including both assembler and C source files, see the *PIC18 IAR C Compiler Reference Guide*.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of PIC18 derivatives are included in the IAR product package, in the `\pic18\inc` directory. These header files define the processor-specific special function registers (SFRs) and interrupt vector numbers.

The header files are intended to be used also with the PIC18 IAR C Compiler, `ICCPIC18`, and they are suitable to use as templates when creating new header files for other PIC18 derivatives.

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

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

USING C-STYLE PREPROCESSOR DIRECTIVES

The C-style preprocessor directives are processed before other assembler directives. Therefore, do not use preprocessor directives in assembler macros and do not mix them with assembler-style comments.

Example

This example will not give the intended behavior since the assembler comment `;` is unknown to the C-style preprocessor:

```
#define SPEED 5 ; speed value
#define TEMP  6 ; temperature

DC8 SPEED,TEMP
```

The resulting line will be expanded as:

```
DC8 5 ; speed value,6 ; temperature
```

which is probably not the intended result. Instead you should use C or C++ style comments when commenting preprocessor directives.

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 *PIC18 IAR Embedded Workbench™ 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, include them on the command line after the `apic18` command, either before or after the source filename. For example, when assembling the source `prog.s49`, use the following command to generate an object file with debug information:

```
apic18 prog --debug
```

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

```
apic18 prog -l prog.lst
```

Some other options accept a string that is not a filename. The string is included after the option letter, but without a space. For example, to define a symbol:

```
apic18 prog -DDEBUG=1
```

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

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

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

SPECIFYING PARAMETERS

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

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

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

or as

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

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

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

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

```
--diag_suppress=Pe0001
```

or

```
--diag_suppress Pe0001
```

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

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

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

```
apic18 prog -l .
```

A file specified by `'-'` is standard input or output, whichever is appropriate.

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

```
apic18 prog -l ---
```

ENVIRONMENT VARIABLES

Assembler options can also be specified in the `ASMPIC18` environment variable. The assembler automatically appends the value of this variable to every command line, so it provides a convenient method of specifying options that are required for every assembly.

The following environment variables can be used with the PIC18 IAR Assembler:

Environment variable	Description
APIC18_INC	Specifies directories to search for include files; for example: APIC18_INC=c:\program files\iar systems\embedded workbench 3.n\pic18\inc;c:\headers
ASMPIC18	Specifies command line options; for example: ASMPIC18=-l asm.lst

Table 6: Environment variables

ERROR RETURN CODES

The PIC18 IAR Assembler returns status information to the operating system which can be tested in a batch file.

The following command line error codes are supported:

Code	Description
0	Assembly successful, but there may have been warnings.
1	There were warnings, provided that the option --warnings_affect_exit_code was used.
2	There were non-fatal errors or fatal assembly errors (making the assembler abort).
3	There were crashing errors.

Table 7: Error return codes

Summary of assembler options

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

Command line option	Description
`--case_insensitive`	Case-insensitive user symbols
`-D`*`symbol`*`[=`*`value`*`]`	Defines preprocessor symbols
`--debug`	Generates debug information
`--diag_error=`*`tag`*`,`*`tag`*`,...`	Treats these diagnostics as errors
`--diag_remark=`*`tag`*`,`*`tag`*`,...`	Treats these diagnostics as remarks
`--diag_suppress=`*`tag`*`,`*`tag`*`,...`	Suppresses these diagnostics
`--diag_warning=`*`tag`*`,`*`tag`*`,...`	Treats these diagnostics as warnings
`--dir_first`	Allows directives in the first column
`-f extend.xcl`	Extends the command line

Table 8: Assembler options summary

Command line option	Description
`-I`*`prefix`*	Includes file paths
`-l[d][e][a][o][m][x]` *`filename`*	Lists to named file
`--library_module`	Makes a library module
`-M`*`ab`*	Macro quote characters
`--macro_info`	Macro execution information
`--mnem_first`	Allows mnemonics in the first column
`--module_name=`*`name`*	Sets object module name
`--no_warnings`	Disables all warnings
`-o` *`filename`*	Sets object filename
`--only_stdout`	Uses standard output only
`--preprocess=[c][n][l]` *`filename`*	Preprocessor output to file
`-r`	Generates debug information
`--remarks`	Enables remarks
`--silent`	Sets silent operation
`--warnings_affect_exit_code`	Warnings affect exit code
`--warnings_are_errors`	Treats all warnings as errors

Table 8: Assembler options summary (Continued)

Description of assembler options

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

`--case_insensitive` `--case_insensitive`

Use this option to make user symbols case insensitive.

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

You can also use the assembler directives `CASEON` and `CASEOFF` to control case sensitivity for user-defined symbols. See *Assembler control directives*, page 78, for more information.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Language**.

-D -D*symbol*[=*value*]

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

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

Example

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:    apic18 prog
test version:          apic18 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:

```
apic18 prog -Dframerate=3
```

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Preprocessor**.

--debug, -r --debug

-r

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

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Output**.

--diag_error

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

Use this option to classify diagnostic messages as errors.

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

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

```
--diag_error=As001
```

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Diagnostics**.

--diag_remark

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

Use this option to classify diagnostic messages as remarks.

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

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

```
--diag_remark=As001
```

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Diagnostics**.

--diag_suppress

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

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

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Diagnostics**.

--diag_warning

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

Use this option to classify diagnostic messages as warnings.

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

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

```
--diag_warning=As028
```

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Diagnostics**.

`--dir_first` `--dir_first`

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

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Language**.

`-f` `-f extend.xcl`

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:

```
apic18 prog -f extend.xcl
```

`-I` `-I`*`prefix`*

Includes paths to be used by the preprocessor.

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 `APIC18_INC` environment variable. The `-I` option allows you to give the assembler the names of directories which it will also search if it fails to find the file in the current working directory.

Example

For example, using the options:

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

and then writing:

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

in the source, will make the assembler search first in the current directory, then in the directory `c:\global\`, and then in the directory `C:\thisproj\headers\`. Finally, the assembler searches the directories specified in the `APIC18_INC` environment variable, provided that this variable is set.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Preprocessor**.

`-l` `-l[d][e][a][o][m][x] filename`

Use this option if you want the assembler to generate a list file with the indicated *filename*, and specify which information to include in the list file. If no extension is specified, `lst` is used. Notice that you must include a space before the filename.

You can choose to include one or more of the following types of information:

Command line option	Description
`-ld`	Disables list file
`-le`	No macro expansions
`-la`	Assembled lines only
`-lo`	Multiline code
`-lm`	Macro definitions
`-lx`	Includes cross-references

Table 9: Conditional list options (-l)

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >List**.

`--library_module` `--library module`

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 `--library module` option if you instead want the assembler to make a library module for use with the IAR XLIB Librarian™.

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Output**.

`-M` `-Mab`

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

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

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

```
apic18 filename -M'<>'
```

Example

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

to call a macro print with > as the argument.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Language**.

`--macro_info` `--macro_info`

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 mainly used in conjunction with the list file option `-l`. For additional information, see page 16.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >List**.

--mnem_first `--mnem_first`

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

Use this option to make mnemonics names (without a trailing colon) starting in the first column to be recognized as mnemonics.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Language**.

--module_name `--module_name=`*`name`*

By default the internal name of the object module is the name of the source file, without a directory name or extension. To set the object module name explicitly, use this option, for example:

```
apic18 prog --module_name=main
```

This option is particularly useful when several modules have the same filename, since the resulting duplicate module name would normally cause a linker error; for example, when the source file is a temporary file generated by a preprocessor.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Output**.

--no_warnings `--no_warnings`

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Diagnostics**.

-o `-o` *`filename`*

Sets the filename to be used for the object file. If no extension is specified, `r49` is used.

For example, the following command puts the object code to the file `obj.r49` instead of the default `prog.r49`:

```
apic18 prog -o obj
```

Note: You must include a space between the option itself and the filename.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Output**.

--only_stdout

```
--only_stdout
```

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

--preprocess

```
--preprocess=[c][n][l] filename
```

Use this option to direct preprocessor output to the named file, *`filename`*`.i`.

The filename consists of the filename itself, optionally preceded by a path name and optionally followed by an extension. If no extension is given, the extension `i` is used. In the syntax description above, note that space is allowed in front of the filename.

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

Command line option	Description
`--preprocess=c`	Preserve comments
`--preprocess=n`	Preprocess only
`--preprocess=l`	Generate `#line` directives

Table 10: Directing preprocessor output to file (--preprocess)

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Preprocessor**.

-r, --debug

```
--debug
-r
```

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

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Output**.

--remarks

```
--remarks
```

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

See *Severity levels*, page 97, for additional information about diagnostic messages.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Diagnostics**.

`--silent` `--silent`

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

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

`--warnings_affect_exit_code` `--warnings_affect_exit_code`

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Diagnostics**.

`--warnings_are_errors` `--warnings_are_errors`

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

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

```
--diag_warning=As001
```

For additional information, see *--diag_warning*, page 14.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>APIC18 >Diagnostics**.

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 15 (the lowest precedence, i.e. last evaluated).

The following rules determine how expressions are evaluated:

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

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

Note: The precedence order in the PIC18 IAR Assembler differs from that in the PIC16/17 IAR Assembler. In the PIC18 IAR Assembler it closely follows the precedence order of the ANSI C++ standard for operators.

Summary of assembler operators

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

PARENTHESIS OPERATOR – 1

`()`	Parenthesis.

FUNCTION OPERATORS – 2

`BYTE1`	First byte.
`BYTE2`	Second byte.
`BYTE3`	Third byte.
`BYTE4`	Fourth byte.
`DATE`	Current date/time.
`HIGH`	High byte.
`HWRD`	High word.
`LOC`	Local variable reference.
`LOW`	Low byte.
`LWRD`	Low word.
`PRM`	Parameter reference
`SFB`	Segment begin.
`SFE`	Segment end.
`SIZEOF`	Segment size.
`UPPER`	Third byte.

UNARY OPERATORS – 3

`+`	Unary plus.
`BINNOT [~]`	Bitwise NOT.
`NOT [!]`	Logical NOT.
`-`	Unary minus.

MULTIPLICATIVE ARITHMETIC OPERATORS – 4

`*`	Multiplication.
`/`	Division.
`MOD [%]`	Modulo.

ADDITIVE ARITHMETIC OPERATORS – 5

`+`	Addition.
–	Subtraction.

SHIFT OPERATORS – 6

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

COMPARISON OPERATORS – 7

`GE [>=]`	Greater than or equal.
`GT [>]`	Greater than.
`LE [<=]`	Less than or equal.
`LT [<]`	Less than.
`UGT`	Unsigned greater than.
`ULT`	Unsigned less than.

EQUIVALENCE OPERATORS – 8

`EQ [=] [==]`	Equal.
`NE [<>] [!=]`	Not equal.

LOGICAL OPERATORS – 9-14

`BINAND [&]`	Bitwise AND (9).
`BINXOR [^]`	Bitwise exclusive OR (10).
`BINOR [|]`	Bitwise OR (11).
`AND [&&]`	Logical AND (12).
`XOR`	Logical exclusive OR (13).
`OR [||]`	Logical OR (14).

CONDITIONAL OPERATOR – 15

`?:`	Conditional operator.

Description of assembler operators

The following sections give full descriptions of each assembler operator.

`()` Parenthesis (1).

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

Example

```
1+2*3 → 7
(1+2)*3 → 9
```

`*` Multiplication (4).

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

Example

```
2*2 → 4
-2*2 → -4
```

`+` Unary plus (3).

Unary plus operator.

Example

```
+3 → 3
3*+2 → 6
```

\+ Addition (5).

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

Example

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

\- Unary minus (3).

The unary minus operator performs arithmetic negation on its operand.

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

\- Subtraction (5).

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

Example

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

/ Division (4).

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

Example

```
9/2 → 4
```

```
-12/3 → -4
9/2*6 → 24
```

`?:` Conditional operator (15).

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

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

Syntax

condition ? *expr* : *expr*

Example

```
5 ? 6 : 7 →6
0 ? 6 : 7 →7
```

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

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

Example

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

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

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

Example

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

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

Use `BINNOT` to perform bitwise NOT on its operand.

Example

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

`BINOR [|]` Bitwise OR (11).

Use `BINOR` to perform bitwise OR on its operands.

Example

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

`BINXOR [^]` Bitwise exclusive OR (10).

Use `BINXOR` to perform bitwise XOR on its operands.

Example

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

`BYTE1` First byte (2).

`BYTE1` takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the low byte (bits 7 to 0) of the operand.

Example

```
BYTE1 0x12345678 → 0x78
```

`BYTE2` Second byte (2).

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

Example

```
BYTE2 0x12345678 → 0x56
```

`BYTE3` Third byte (2).

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

Example

```
BYTE3 0x12345678 → 0x34
```

`BYTE4` Fourth byte (2).

`BYTE4` takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the high byte (bits 31 to 24) of the operand.

Example

```
BYTE4 0x12345678 → 0x12
```

`DATE` Current date/time (2).

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

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

`DATE 1`	Current second (0–59)
`DATE 2`	Current minute (0–59)
`DATE 3`	Current hour (0–23)
`DATE 4`	Current day (1–31)
`DATE 5`	Current month (1–12)
`DATE 6`	Current year `MOD 100` (1998 → 98, 2000 → 00, 2002 → 02)

Example

To assemble the date of assembly:

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

`EQ [=] [==]` Equal (8).

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

Example

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

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

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

Example

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

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

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

Example

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

`HIGH` High byte (2).

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

Example

```
HIGH 0xABCD → 0xAB
```

`HWRD` High word (2).

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

Example

```
HWRD 0x12345678 → 0x1234
```

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

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

Example

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

`LOC` Local variable reference (2)

`LOC` evaluates to an absolute address in the memory area block used for a function's local variables in a specific segment. This evaluation takes place at link time.

`LOC` is intended for functions using static overlays. The memory area block for local variables must have been defined using the `LOCFRAME` assembler directive.

See also the *PIC18 IAR C Compiler Reference Guide* for information about the assembler language interface.

Syntax

`LOC(`*`function`*`, `*`segment`*`, `*`offset`*`)`

Parameters

`function`	The name of the function.
`segment`	The name of a memory segment, which must be defined before `LOC` is used.
`offset`	An offset from the start address.

Example

```
        LEA     LOC(func,X,0)
```

This will load the address of the first local variable of `func` into the EA register.

`LOW` Low byte (2).

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

Example

```
LOW 0xABCD → 0xCD
```

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

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

Example

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

`LWRD` Low word (2).

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

Example

```
LWRD 0x12345678 → 0x5678
```

`MOD [%]` Modulo (4).

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

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

Example

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

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

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

Example

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

`NOT [!]` Logical NOT (3).

Use `NOT` to negate a logical argument.

Example

```
NOT 0101B → 0
NOT 0000B → 1
```

`OR [||]` Logical OR (14).

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

Example

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

`PRM` Parameter reference (2).

`PRM` evaluates to an absolute address in the memory area block used for a function's parameters in a specific segment. This evaluation takes place at link time.

`PRM` is intended for functions using static overlays. The memory area block for parameters must have been defined using the `ARGFRAME` assembler directive.

See also the *PIC18 IAR C Compiler Reference Guide* for information about the assembler language interface.

Syntax

`PRM(`*`function`*`, `*`segment`*`, `*`offset`*`)`

Parameters

`function`	The name of the function.
`segment`	The name of a memory segment, which must be defined before `PRM` is used.
`offset`	An offset from the start address.

Example

```
        LEA     PRM(func,X,0)
```

This will load the address of the first parameter of `func` into the EA register.

SFB Segment begin (2).

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

Syntax

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

Parameters

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

Example

```
        NAME  demo
        RSEG  segtab: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 (2).

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

Syntax

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

Parameters

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

Example

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

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

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

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

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

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

Example

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

`SHR [>>]` Logical shift right (6).

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

Example

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

`SIZEOF` Segment size (2).

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

Syntax

```
SIZEOF (segment)
```

Parameters

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

Example

```
      NAME   demo
      RSEG   segtab:CODE
size: DC16   SIZEOF(CODE)
```

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

`UGT` Unsigned greater than (7).

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

Example

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

`ULT` Unsigned less than (7).

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

Example

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

`UPPER` Third byte (2).

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

Example

```
UPPER 0x12345678 → 0x34
```

XOR Logical exclusive OR (13).

Use XOR to perform logical XOR on its two operands.

Example

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

Assembler directives

This chapter gives an alphabetical summary of the assembler directives. It then describes the syntax conventions and provides complete reference information for each category of directives.

Summary of assembler directives

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

Directive	Description	Section
`#define`	Assigns a value to a label.	C-style preprocessor
`#elif`	Introduces a new condition in a `#if`...`#endif` block.	C-style preprocessor
`#else`	Assembles instructions if a condition is false.	C-style preprocessor
`#endif`	Ends a `#if`, `#ifdef`, or `#ifndef` block.	C-style preprocessor
`#error`	Generates an error.	C-style preprocessor
`#if`	Assembles instructions if a condition is true.	C-style preprocessor
`#ifdef`	Assembles instructions if a symbol is defined.	C-style preprocessor
`#ifndef`	Assembles instructions if a symbol is undefined.	C-style preprocessor
`#include`	Includes a file.	C-style preprocessor
`#pragma`	Controls extension features.	C-style preprocessor
`#undef`	Undefines a label.	C-style preprocessor
`/*comment*/`	C-style comment delimiter.	Assembler control
`//`	C++ style comment delimiter.	Assembler control
`=`	Assigns a permanent value local to a module.	Value assignment
`ALIGN`	Aligns the program location counter by inserting zero-filled bytes.	Segment control
`ALIGNRAM`	Aligns the program location counter.	Segment control
`ASEG`	Begins an absolute segment.	Segment control
`ASEGN`	Begins a named absolute segment.	Segment control
`ASSIGN`	Assigns a temporary value.	Value assignment
`CASEOFF`	Disables case sensitivity.	Assembler control
`CASEON`	Enables case sensitivity.	Assembler control

Table 11: Assembler directives summary

Directive	Description	Section
COMMON	Begins a common segment.	Segment control
DB	Generates 8-bit constants, including strings.	Data definition or allocation
DC8	Generates 8-bit constants, including strings.	Data definition or allocation
DC16	Generates 16-bit constants.	Data definition or allocation
DC24	Generates 24-bit constants.	Data definition or allocation
DC32	Generates 32-bit constants.	Data definition or allocation
DC64	Generates 64-bit constants.	Data definition or allocation
DEFINE	Defines a file-wide value.	Value assignment
DF	Generates floating-point constants (in Microchip modified IEEE format)	Data definition or allocation
DF32	Generates 32-bit floating-point constants.	Data definition or allocation
DF64	Generates 64-bit floating-point constants.	Data definition or allocation
DL	Generates 32-bit constants.	Data definition or allocation
DP	Generates 24-bit constants.	Data definition or allocation
DQ15	Generates 16-bit fractional constants.	Data definition or allocation
DS	Allocates space for 8-bit integers.	Data definition or allocation
DS8	Allocates space for 8-bit integers.	Data definition or allocation
DS16	Allocates space for 16-bit integers.	Data definition or allocation
DS24	Allocates space for 24-bit integers.	Data definition or allocation
DS32	Allocates space for 32-bit integers.	Data definition or allocation

Table 11: Assembler directives summary (Continued)

Directive	Description	Section
DS64	Allocates space for 64-bit integers.	Data definition or allocation
DT	Generates byte constants with RETLW instructions	Data definition or allocation
DW	Generates 16-bit constants.	Data definition or allocation
ELSE	Assembles instructions if a condition is false.	Conditional assembly
ELSEIF	Specifies a new condition in an IF...ENDIF block.	Conditional assembly
END	Terminates the assembly of the last module in a file.	Module control
ENDIF	Ends an IF block.	Conditional assembly
ENDM	Ends a macro definition.	Macro processing
ENDMOD	Terminates the assembly of the current module.	Module control
ENDR	Ends a repeat structure	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment
EVEN	Aligns the program counter to an even address.	Segment control
EXITM	Exits prematurely from a macro.	Macro processing
EXTERN	Imports an external symbol.	Symbol control
IF	Assembles instructions if a condition is true.	Conditional assembly
LIBRARY	Begins a library module.	Module control
LIMIT	Checks a value against limits.	Value assignment
LOCAL	Creates symbols local to a macro.	Macro processing
LSTCND	Controls conditional assembler listing.	Listing control
LSTCOD	Controls multi-line code listing.	Listing control
LSTEXP	Controls the listing of macro generated lines.	Listing control
LSTMAC	Controls the listing of macro definitions.	Listing control
LSTOUT	Controls assembler-listing output.	Listing control
LSTREP	Controls the listing of lines generated by repeat directives.	Listing control
LSTXRF	Generates a cross-reference table.	Listing control
MACRO	Defines a macro.	Macro processing
MODULE	Begins a library module.	Module control

Table 11: Assembler directives summary (Continued)

Directive	Description	Section
NAME	Begins a program module.	Module control
ODD	Aligns the program location counter to an odd address.	Segment control
ORG	Sets the program location counter.	Segment control
PROGRAM	Begins a program module.	Module control
PUBLIC	Exports symbols to other modules.	Symbol control
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.	Symbol control
RADIX	Sets the default base.	Assembler control
REPT	Assembles instructions a specified number of times.	Macro processing
REPTC	Repeats and substitutes characters.	Macro processing
REPTI	Repeats and substitutes strings.	Macro processing
REQUIRE	Forces a symbol to be referenced.	Symbol control
RADIX	Sets the default base.	Assembler control
RES	Allocates space for 16-bit integers.	Data definition or allocation
RSEG	Begins a relocatable segment.	Segment control
RTMODEL	Declares runtime model attributes.	Module control
SET	Assigns a temporary value.	Value assignment
VAR	Assigns a temporary value.	Value assignment

Table 11: Assembler directives summary (Continued)

Note: The IAR Systems toolkit for the PIC18 microcontroller also supports the static overlay directives FUNCALL, FUNCTION, LOCFRAME, and ARGFRAME that are designed to ease coexistence of routines written in C and assembler language. These directives are described in the *PIC18 IAR C Compiler Reference Guide*. (Static overlay is not, however, relevant for this product.)

Syntax conventions

In the syntax definitions, the following conventions are used:

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

  ```
  ORG expr
  ```

expr represents an arbitrary expression.

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

  `END [expr]`

  the *expr* parameter is optional.

- An ellipsis indicates that the previous item can be repeated an arbitrary number of times. For example:

  `PUBLIC symbol [,symbol] …`

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

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

  `LSTOUT{+|-}`

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

LABELS AND COMMENTS

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

`label SET expr`

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

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

PARAMETERS

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

Parameter	What it consists of
expr	An expression; see *Assembler expressions*, page 2.
label	A symbolic label.
symbol	An assembler symbol.

Table 12: Assembler directive syntax conventions

Module control directives

Module control directives are used for marking the beginning and end of source program modules, and for assigning names and types to them. See *Expression restrictions*, page 2, for a description of the restrictions that apply when using a direcctive in an expression.

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

Table 13: Module control directives

SYNTAX

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

PARAMETERS

expr — Optional expression used by the compiler to encode the runtime options. For END/ENDMOD, *expr* can take any positive integer value. For the other module control directives it must be within the range 0-255.

key — A text string specifying the key.

symbol — Name assigned to module, used by XLINK and XLIB when processing object files.

value A text string specifying the value.

DESCRIPTIONS

Beginning a program module

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

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

Beginning a library module

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

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

Terminating a module

Use ENDMOD to define the end of a module.

Terminating the source file

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

Assembling multi-module files

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

The following rules apply when assembling multi-module files:

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

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

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

Declaring runtime model attributes

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

A module can have several runtime model definitions.

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

If you are writing assembler routines for use with C code, and you want to control the module consistency, refer to the *PIC18 IAR C Compiler Reference Guide.*

Examples

The following example defines three modules where:

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

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

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

MODULE MOD_3
```

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

Symbol control directives

These directives control how symbols are shared between modules.

Directive	Description
EXTERN	Imports an external symbol.
PUBLIC	Exports symbols to other modules.
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.
REQUIRE	Forces a symbol to be referenced.

Table 14: Symbol control directives

SYNTAX

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

PARAMETERS

symbol	Symbol to be imported or exported.

DESCRIPTIONS

Exporting symbols to other modules

Use PUBLIC to make one or more symbols available to other modules. The symbols declared as PUBLIC can only be assigned values by using them as labels. Symbols declared PUBLIC can be relocated or absolute, and can also be used in expressions (with the same rules as for other symbols).

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

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

Exporting symbols with multiple definitions to other modules

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

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

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

Importing symbols

Use EXTERN to import an untyped external symbol.

The REQUIRE directive marks a symbol as referenced. This is useful if the segment part containing the symbol must be loaded 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.

Since the message is enclosed in double quotes, the string will be followed by a zero byte.

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

```
     NAME         error
     EXTERN       print
     PUBLIC       err

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

     END
```

Segment control directives

The segment directives control how code and data are generated. See *Expression restrictions*, page 2, for a description of the restrictions that apply when using a direcctive in an expression.

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

Table 15: Segment control directives

SYNTAX

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

PARAMETERS

address Address where this segment part will be placed.

align	The power of two to which the address should be aligned, in most cases in the range 0 to 30. For example, `align 1` results in alignment on even addresses since 2^1 equals 2. The default align value is 0, except for code segments where the default is 1.
expr	Address to set the location counter to.

flag	`NOROOT`
	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. The default mode is `ROOT` which indicates that the segment part must not be discarded.
	
	`REORDER`
	Allows the linker to reorder segment parts. For a given segment, all segment parts must specify the same state for this flag. The default mode is `NOREORDER` which indicates that the segment parts must remain in order.
	
	`SORT`
	The linker will sort the segment parts in decreasing alignment order. For a given segment, all segment parts must specify the same state for this flag. The default mode is `NOSORT` which indicates that the segment parts will not be sorted.

segment	The name of the segment.
start	A start address that has the same effect as using an `ORG` directive at the beginning of the absolute segment.
type	The memory type, typically `CODE` or `DATA`. In addition, any of the types supported by the IAR XLINK Linker.
value	Byte value used for padding, default is zero.

DESCRIPTIONS

Beginning an absolute segment

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

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

Beginning a named absolute segment

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

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

Beginning a relocatable segment

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

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

Beginning a common segment

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

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

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

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

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

Setting the program location counter (PLC)

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

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

Aligning a segment

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

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

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

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

EXAMPLES

Beginning an absolute segment

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

```
        MODULE    reset

        EXTERN    main

        ASEG
        ORG       0           ; RESET vector address

reset:  GOTO      main        ; Instruction that
                              ; executes on startup

        end
```

Beginning a relocatable segment

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

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

```
        EXTERN     subrtn, divrtn

        RSEG       table
```

```
functable:
       DC16        subrtn, divrtn

       RSEG        code

main:
       MOWLW      0x12
       ADDWF      0x20,0
       RETURN

       END
```

Beginning a common segment

The following example defines two common segments containing variables:

```
         NAME     common1
         COMMON   data
count    DC16     1
         ENDMOD
         NAME     common2
         COMMON   data
up       DS8      1
         DS8      2
down     DS8      1
         END
```

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

Aligning a segment

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

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

Value assignment directives

These directives are used for assigning values to symbols.

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

Table 16: Value assignment directives

SYNTAX

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

PARAMETERS

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

DESCRIPTIONS

Defining a temporary value

Use SET, VAR or ASSIGN to define a symbol that may be redefined, such as for use with macro variables. Symbols defined with SET, VAR or ASSIGN cannot be declared PUBLIC.

Defining a permanent local value

Use EQU or = to assign a value to a symbol.

Use EQU or = to create a local symbol that denotes a number or offset. The symbol is only valid in the module in which it was defined, but can be made available to other modules with a PUBLIC directive.

Use EXTERN to import symbols from other modules.

Defining a permanent global value

Use DEFINE to define symbols that should be known to the module containing the directive and all modules following that module in the same source file. If a DEFINE directive is placed outside of a module, the symbol will be known to all modules following the directive in the same source file.

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

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

Checking symbol values

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

The check will occur as soon as the expression is resolved, which will be during linking if the expression contains external references.

EXAMPLES

Redefining a symbol

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

```
        NAME    table

; Generate table of powers of 3

cons    SET     1

cr_tabl MACRO   times
        DW      cons
cons    SET     cons*3
        IF      times>1
        cr_tabl times-1
```

```
        ENDIF
        ENDM

table:
        cr_tabl 4
        END     table
```

It generates the following code:

```
16    000000             NAME table
17
18    000000             ; Generate table of powers of 3
19
20    000000            cons     SET    1
21
22                      cr_tabl  MACRO  times
23                       DW      cons
24                      cons     SET    cons*3
25                       IF      times>1
26                       cr_tabl times-1
27                       ENDIF
28                       ENDM
29
30    000000            table:
31    000000             cr_tabl 4
31.1  000000 0100         DW      cons
31.2  000000            cons     SET     cons*3
31.3  000000             IF      4>1
31.4  000002             cr_tabl 4-1
31.5  000002 0300         DW      cons
31.6  000000            cons     SET     cons*3
31.7  000000             IF      4-1>1
31.8  000004             cr_tabl 4-1-1
31.9  000004 0900         DW      cons
31.10 000000            cons     SET     cons*3
31.11 000000             IF      4-1-1>1
31.12 000006             cr_tabl 4-1-1-1
31.13 000006 1B00         DW      cons
31.14 000000            cons     SET     cons*3
31.15 000000             IF      4-1-1-1>1
31.16 000000             ENDIF
31.18 000000             ENDIF
31.20 000000             ENDIF
31.22 000000             ENDIF
32    000008             END table
33
```

Using local and global symbols

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

```
        NAME    add1
        PUBLIC  add12
R0      DEFINE  0x20
value   EQU     12

add12:
        MOVLW   value
        ADDWF   R0,1
        RETURN
        ENDMOD

        NAME    add2
        PUBLIC  add20
value   EQU     20

add20:
        MOVLW   value
        ADDWF   R0,1
        RETURN

        END
```

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

Using the LIMIT directive

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

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

Conditional assembly directives

These directives provide logical control over the selective assembly of source code. See *Expression restrictions*, page 2, for a description of the restrictions that apply when using a directive in an expression.

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

Table 17: Conditional assembly directives

SYNTAX

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

PARAMETERS

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

DESCRIPTIONS

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

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

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

EXAMPLES

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

```
add     MACRO   a,b       ; A should be a register file,
                          ; b a literal
        IF      b=1
        INCF    a,1
        ELSE
        MOVLW   b
        ADDWF   a,1
        ENDIF
        ENDM
```

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

It could be tested with the following program:

```
R0      DEFINE  0x20
R1      DEFINE  0x21
main:
        MOVLW   0x0F
        MOVWF   R0
        add     R0, 0x12
        add     R1, 1
        RETURN

        END
```

Macro processing directives

These directives allow user macros to be defined. See *Expression restrictions*, page 2, for a description of the restrictions that apply when using a direcctive in an expression.

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

Table 18: Macro processing directives

SYNTAX

```
ENDM
ENDR
EXITM
LOCAL symbol [,symbol] …
name MACRO [argument] [,argument] …
REPT expr
```

```
REPTC formal,actual
REPTI formal,actual [,actual] …
```

PARAMETERS

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

DESCRIPTIONS

A macro is a user-defined symbol that represents a block of one or more assembler source lines. Once you have defined a macro you can use it in your program like an assembler directive or assembler mnemonic.

When the assembler encounters a macro, it looks up the macro's definition, and inserts the lines that the macro represents as if they were included in the source file at that position.

Macros perform simple text substitution effectively, and you can control what they substitute by supplying parameters to them.

Defining a macro

You define a macro with the statement:

name MACRO [*argument*] [,*argument*] …

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

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

```
errmac  MACRO   errno
        MOVLW   errno
        CALL    abort
        ENDM
```

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

```
        errmac  2
```

The assembler will expand this to:

```
        MOVLW   2
        CALL    abort
```

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
        MOVLW   \1
        CALL    abort
        ENDM
```

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

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

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

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

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

Passing special characters

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

For example:

```
macld    MACRO  op
         MOVFF  op
         ENDM
```

The macro can be called using the macro quote characters:

```
         macld  <0x800, 0x900>
         END
```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 17.

Predefined macro symbols

The symbol `_args` is set to the number of arguments passed to the macro. The following example shows how `_args` can be used:

```
DO_CONST  MACRO
     IF _args == 2
       DC8  \1,\2
     ELSE
       DC8  \1
     ENDIF
ENDM

RSEG      CODE

DO_CONST  3, 4
DO_CONST  3

END
```

How macros are processed

There are three distinct phases in the macro process:

1 The assembler performs scanning and saving of macro definitions. The text between `MACRO` and `ENDM` is saved but not syntax checked.
2 A macro call forces the assembler to invoke the macro processor (expander). The macro expander switches (if not already in a macro) the assembler input stream from a source file to the output from the macro expander. The macro expander takes its input from the requested macro definition.
The macro expander has no knowledge of assembler symbols since it only deals with text substitutions at source level. Before a line from the called macro definition is handed over to the assembler, the expander scans the line for all occurrences of symbolic macro arguments, and replaces them with their expansion arguments.
3 The expanded line is then processed as any other assembler source line. The input stream to the assembler will continue to be the output from the macro processor, until all lines of the current macro definition have been read.

Repeating statements

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

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

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

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

EXAMPLES

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

Coding 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 adds two 16-bit constants found in R1 and R2, and returns the result in R1. The program does not handle the case where overflow occurs in the high byte when propagating the carry from the low byte addition:

```
#define R1      0x20
#define R2      0x22
#define STATUS  3
#define CARRY   0
        ORG     0
start:  GOTO    main

        RSEG    CODE
add16:
        MOVF    R2+1,0        ; Read R2LOW to w
        ADDWF   R1+1,1        ; ADD and store in R1LOW
        MOVF    R2,0          ; Load high part
        BTFSS   STATUS, CARRY
        GOTO    no_carry
        ADDLW   1             ; Take care of carry

no_carry:
        ADDWF   R1            ; Store result
        RETURN


        RSEG    CODE
        PUBLIC main
main:   MOVLW  0xF0
        MOVWF  R1 + 1
        MOVLW  0x77
        MOVWF  R2 + 1
        MOVLW  0x10
        MOVWF  R1
        MOVLW  0x10
```

```
        MOVWF   R2
        CALL    add16

loop:   GOTO    loop

        end     main
```

The main program calls this routine as follows:

```
        CALL    add16
```

For efficiency we can recode this using a macro:

```
#define R1      0x20
#define R2      0x22
#define STATUS 3
#define CARRY   0

        ORG     0
start:  GOTO    main

        RSEG    CODE
add16   MACRO
        MOVF    R2+1,0          ; Read R2LOW to w
        ADDWF   R1+1,1          ; ADD and store in R1LOW
        MOVF    R2,0            ; Load high part
        BTFSS   STATUS, CARRY
        GOTO    no_carry
        ADDLW   1               ; Take care of carry

no_carry:
        ADDWF   R1              ; Store result
        ENDM

        RSEG    CODE
        PUBLIC main
main:   MOVLW   0xF0
        MOVWF   R1 + 1
        MOVLW   0x77
        MOVWF   R2 + 1
        MOVLW   0x10
        MOVWF   R1
        MOVLW   0x10
        MOVWF   R2
        add16

loop:   GOTO    loop

        end     main
```

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

```
add16
```

Using REPTC and REPTI

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

```
        NAME   reptc

        EXTERN plotc
R0      DEFINE 0x20
banner  REPTC  chr, "Welcome"

        MOVLW  'chr'
        MOVWF  R0          ; Pass char in R0 as parameter
        CALL   plotc
        ENDR

        END
```

This produces the following code:

```
    16    000000             NAME   reptc
    17
    18
    19    000000             EXTERN plotc
    20    000000            R0      DEFINE 0x20
    21                      banner  REPTC  chr, "Welcome"
    22
    23                       MOVLW  'chr'
    24                       MOVWF  R0    ; Pass char in R0 as parameter
    25                       CALL   plotc
    26                       ENDR
    26.1  000000 570E        MOVLW  'W'
    26.2  000002 206E        MOVWF  R0    ; Pass char in R0 as parameter
    26.3  000004 ........    CALL   plotc
    26.4  000008 650E        MOVLW  'e'
    26.5  00000A 206E        MOVWF  R0    ; Pass char in R0 as parameter
    26.6  00000C ........    CALL   plotc
    26.7  000010 6C0E        MOVLW  'l'
    26.8  000012 206E        MOVWF  R0    ; Pass char in R0 as parameter
    26.9  000014 ........    CALL   plotc
    26.10 000018 630E        MOVLW  'c'
    26.11 00001A 206E        MOVWF  R0    ; Pass char in R0 as parameter
    26.12 00001C ........    CALL   plotc
    26.13 000020 6F0E        MOVLW  'o'
```

```
26.14 000022 206E        MOVWF  R0    ; Pass char in R0 as parameter
26.15 000024 ........    CALL   plotc
26.16 000028 6D0E        MOVLW  'm'
26.17 00002A 206E        MOVWF  R0    ; Pass char in R0 as parameter
26.18 00002C ........    CALL   plotc
26.19 000030 650E        MOVLW  'e'
26.20 000032 206E        MOVWF  R0    ; Pass char in R0 as parameter
26.21 000034 ........    CALL   plotc
27
28
29    000038             END
30
31
```

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

```
        NAME   repti

        EXTERN base, count, init

banner  REPTI  adds, base, count, init

        CLRF   adds
        ENDR

        END
```

This produces the following code:

```
16    000000             NAME    repti
17
18    000000             EXTERN  base, count, init
19
20
21                     banner    REPTI    adds, base, count, init
22
23                       CLRF    adds
24                       ENDR
24.1  000000 ....        CLRF    base
24.2  000002 ....        CLRF    count
24.3  000004 ....        CLRF    init
25
26
27    000006             END
28
```

Listing control directives

These directives provide control over the assembler list file.

Directive	Description
LSTCND	Controls conditional assembly listing.
LSTCOD	Controls multi-line code listing.
LSTEXP	Controls the listing of macro-generated lines.
LSTMAC	Controls the listing of macro definitions.
LSTOUT	Controls assembly-listing output.
LSTREP	Controls the listing of lines generated by repeat directives.
LSTXRF	Generates a cross-reference table.

Table 19: Listing control directives

SYNTAX

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

DESCRIPTIONS

Turning the listing on or off

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

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

Listing conditional code and strings

Use `LSTCND+` to force the assembler to list source code only for the parts of the assembly that are not disabled by previous conditional `IF` statements, `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; that is, long ASCII strings will produce several lines of output.

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

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

Controlling the listing of macros

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

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

Controlling the listing of generated lines

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

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

Generating a cross-reference table

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

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

EXAMPLES

Turning the listing on or off

To disable the listing of a debugged section of program:

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

Listing conditional code and strings

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

```
        NAME    lstcndtst
        EXTERN  print

        RSEG    prom

debug   SET     0
begin   IF      debug
```

```
        CALL    print
        ENDIF

        LSTCND+
begin2  IF      debug
        CALL    print
        ENDIF

        END
```

This will generate the following listing:

```
    16    000000                NAME lstcndtst
    17    000000                EXTERN print
    18
    19    000000                RSEG prom
    20
    21    000000          debug SET    0
    22    000000          begin IF     debug
    23                          CALL print
    24    000000                ENDIF
    25
    26    000000                LSTCND+
    27    000000          begin2 IF    debug
    28                          CALL   print
    29    000000                ENDIF
    30
    31    000000                END
```

Controlling the listing of macros

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

```
dec2    MACRO  arg
        DECF   arg,1
        DECF   arg,1
        ENDM

        LSTMAC-
inc2    MACRO  arg
        INCF   arg,1
        INCF   arg,1
        ENDM

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

```
        RETURN

        END     begin
```

This will produce the following output:

```
 1                                      NAME    dec2
 2
 3                          dec2        MACRO   arg
 4                                      subw    $2,arg
 5                                      ENDM
 6
 7    000000                            LSTMAC+
 8
 9
10    000000                begin:      dec2    R6
10.1  000000 263A                       subw    $2,R6
11
12    000000                            LSTEXP-
13    000002                            inc2    R7
14
15
16    000000                            ; restore defaults
17    000000                            LSTMAC-
18    000000                            LSTEXP+
19
20    000004                            END
```

C-style preprocessor directives

The following C-language preprocessor directives are available:

Directive	Description
`#define`	Assigns a value to a label.
`#elif`	Introduces a new condition in a `#if...#endif` block.
`#else`	Assembles instructions if a condition is false.
`#endif`	Ends a `#if`, `#ifdef`, or `#ifndef` block.
`#error`	Generates an error.
`#if`	Assembles instructions if a condition is true.
`#ifdef`	Assembles instructions if a symbol is defined.
`#ifndef`	Assembles instructions if a symbol is undefined.
`#include`	Includes a file.

Table 20: C-style preprocessor directives

Directive	Description
#pragma	Controls extension features. The supported #pragma directives are described in the chapter *#pragma directives*.
#undef	Undefines a label.

Table 20: C-style preprocessor directives (Continued)

SYNTAX

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

PARAMETERS

condition	One of the following:	
	An absolute expression	The expression must not contain any assembler labels or symbols, and any non-zero value is considered as true.
	string1==string	The condition is true if *string1* and *string2* have the same length and contents.
	string1!=string2	The condition is true if *string1* and *string2* have different length or contents.
filename	Name of file to be included.	
label	Symbol to be defined, undefined, or tested.	
message	Text to be displayed.	
text	Value to be assigned.	

DESCRIPTIONS

The preprocessor directives are processed before other directives. As an example avoid constructs like

```
redef macro
#define \1 \2
      endm
```

since the `\1` and `\2` macro arguments will not be available during the preprocess.

Also be careful with comments; the preprocessor understands `/* */` and `//`. The following expression will evaluate to 3 since the comment character will be preserved by `#define`:

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

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

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

```
#define    five 5 ; comment

    MOVFF  five+addr,0x20     ;syntax error!
    ; Expands to "MOVFF 5 ; comment+addr,0x20"

    MOVFF  0x33,five + addr   ; incorrect code!
    ; Expanded to "MOVFF 0x33,5 ; comment + addr"
```

Defining and undefining labels

Use `#define` to define a temporary label.

`#define` *`label value`*

is similar to:

`label` `SET` *`value`*

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

Conditional directives

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

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

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

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

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

Including source files

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

When the assembler encounters the name of an `#include` file in double quotes such as:

```
#include "vars.h"
```

it searches the directory of the source file in which the `#include` statement occurs, and then performs the same sequence as for angle-bracketed filenames (see below).

If there are nested `#include` files, the assembler starts searching the directory of the file that was last included, iterating upwards for each included file, searching the source file directory last.

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

1 The directories specified by the `-I` option, or options.
2 The directories specified by the `APIC18_INC` environment variable.

Use angle brackets for standard header files, and double quotes for header files that are part of your application.

Displaying errors

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

Defining comments

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

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

EXAMPLES

Using conditional directives

The following example defines the labels `tweek` and `adjust`. If `adjust` is defined, then register 16 is decremented by an amount that depends on `adjust`, in this case 30.

```
#define tweek 1
#define adjust 3

#ifdef  tweek
#if     adjust==1
        SUBLW 4
#elif   adjust==2
        SUBLW 20
#elif   adjust==3
        SUBLW 30
#endif
#endif                                          /* ifdef tweek */
```

Including a source file

The following example uses `#include` to include a file defining macros into the source file. For example, the following macros could be defined in `Macros.s49`:

```
; exchange a and b using c as temporary
xch      MACRO   a,b, c
         MOVF    a,0
         MOVWF   c
         MOVF    b,0
         MOVWF   a
         MOVF    c,0
         MOVWF   b
         ENDM
```

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

```
         NAME    include

R0 DEFINE 0x20
R1 DEFINE 0x21
```

```
R2 DEFINE 0x22

; standard macro definitions

#include "macros.s49"

; program

main:
         xch     R0,R1,R2
         RETURN
         END     main
```

Data definition or allocation directives

These directives define temporary values or reserve memory. The column *Alias* in the following table shows the Microchip® directive that corresponds to the IAR Systems directive. See *Expression restrictions*, page 2, for a description of the restrictions that apply when using a direcctive in an expression.

Directive	Alias	Description	Expression restrictions
DC8	DB	Generates 8-bit constants, including strings.	
DC16	DW	Generates 16-bit constants.	
DC24	DP	Generates 24-bit constants.	
DC32	DL	Generates 32-bit constants.	
DC64		Generates 64-bit constants.	
DF32	DF	Generates 32-bit floating-point constants.	
DF64		Generates 64-bit floating-point constants.	
DQ15		Generates 16-bit fractional constants.	
DS8	DS	Allocates space for 8-bit integers.	No external references Absolute
DS16	RES	Allocates space for 16-bit integers.	No external references Absolute
DS24		Allocates space for 24-bit integers.	No external references Absolute
DS32		Allocates space for 32-bit integers.	No external references Absolute

Table 21: Data definition or allocation directives

Directive	Alias	Description	Expression restrictions
DS64		Allocates space for 64-bit integers.	No external references Absolute

Table 21: Data definition or allocation directives (Continued)

SYNTAX

```
DB expr[,expr]
DC8 expr [,expr] ...
DC16 expr [,expr] ...
DC24 expr [,expr] ...
DC32 expr [,expr] ...
DC64 expr [,expr] ...
DF aexpr [,aexpr]
DF32 value [,value] ...
DF64 value [,value] ...
DL expr[,expr]
DP expr[,expr]
DQ15 value [,value] ...
DS aexpr
DS8 aexpr
DS16 aexpr
DS24 aexpr
DS32 aexpr
DS64 aexpr
DW expr[,expr]
RES expr [,expr] ...
```

PARAMETERS

aexpr A valid absolute expression, or an ASCII string. ASCII strings will be zero filled to a multiple of the data size implied by the directive. Double-quoted strings will be zero-terminated.

expr A valid absolute, relocatable, or external expression, or an ASCII string. ASCII strings will be zero filled to a multiple of the data size implied by the directive. Double-quoted strings will be zero-terminated.

value A valid absolute expression or floating-point constant.

DESCRIPTIONS

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

Size	Reserve and initialize memory	Reserve unitialized memory
8-bit integers	DC8, DB	DS8, DS
16-bit integers	DC16, DW	DS16
24-bit integers	DC24, DP	DS24
32-bit integers	DC32, DL	DS32
64-bit integers	DC64	DS64
32-bit floats	DF32, DF	DS32
64-bit floats	DF64	DS64

Table 22: Using data definition or allocation directives

EXAMPLES

Generating a lookup table

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

```
          NAME    table

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

          RSEG    CODE
sum       DEFINE  0x20
          COUNT   SET 0

fsum:
          REPT    7
          IF      COUNT == 7
          EXITM
          ENDIF
          CALL    table+COUNT        ; load table data in
```

```
                                        ; WREG
        ADDWF   sum,1                   ; ADD up
COUNT   SET     COUNT+1
        ENDR
        MOVF    sum,0                   ; Get sum into WREG
        RETURN

        END
```

Defining strings

To define a string:

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

To define a string which includes a trailing zero:

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

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

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

Reserving space

To reserve space for `0xA` bytes:

```
table   DS8   0xA
```

Assembler control directives

These directives provide control over the operation of the assembler. See *Expression restrictions*, page 2, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
`/*comment*/`	C-style comment delimiter.	
`//`	C++ style comment delimiter.	
`CASEOFF`	Disables case sensitivity.	
`CASEON`	Enables case sensitivity.	
`RADIX`	Sets the default base.	No forward references No external references Absolute Fixed

Table 23: Assembler control directives

SYNTAX

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

PARAMETERS

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

DESCRIPTIONS

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

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

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

Controlling case sensitivity

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

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

EXAMPLES

Defining comments

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

```
/*
Program to read serial input.
Version 1: 19.2.02
Author: mjp
*/
```

Changing the base

To set the default base to 16:

```
        RADIX  16'D
        MOVLW  12
```

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

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

```
        RADIX  0x0A
```

Controlling case sensitivity

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

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

                         ; The following will generate a
                         ; duplicate label error:

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

      END
```

Call frame information directives

These directives allow backtrace information to be defined.

Directive	Description
`CFI BASEADDRESS`	Declares a base address CFA (Canonical Frame Address).
`CFI BLOCK`	Starts a data block.
`CFI CODEALIGN`	Declares code alignment.
`CFI COMMON`	Starts or extends a common block.
`CFI CONDITIONAL`	Declares data block to be a conditional thread.
`CFI DATAALIGN`	Declares data alignment.
`CFI ENDBLOCK`	Ends a data block.
`CFI ENDCOMMON`	Ends a common block.
`CFI ENDNAMES`	Ends a names block.
`CFI FRAMECELL`	Creates a reference into the caller's frame.
`CFI FUNCTION`	Declares a function associated with data block.
`CFI INVALID`	Starts range of invalid backtrace information.

Table 24: Call frame information directives

Directive	Description
CFI NAMES	Starts a names block.
CFI NOFUNCTION	Declares data block to not be associated with a function.
CFI PICKER	Declares data block to be a picker thread.
CFI REMEMBERSTATE	Remembers the backtrace information state.
CFI RESOURCE	Declares a resource.
CFI RESOURCEPARTS	Declares a composite resource.
CFI RESTORESTATE	Restores the saved backtrace information state.
CFI RETURNADDRESS	Declares a return address column.
CFI STACKFRAME	Declares a stack frame CFA.
CFI STATICOVERLAYFRAME	Declares a static overlay frame CFA.
CFI VALID	Ends range of invalid backtrace information.
CFI VIRTUALRESOURCE	Declares a virtual resource.
CFI *cfa*	Declares the value of a CFA.
CFI *resource*	Declares the value of a resource.

Table 24: Call frame information directives (Continued)

SYNTAX

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

Names block directives

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

Extended names block directives

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

Common block directives

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

Extended common block directives

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

Data block directives

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

PARAMETERS

align	The power of two to which the address should be aligned. The allowed range for align is 0 to 31. As an example, the value 1 results in alignment on even addresses since 2^1 equals 2. The default align value is 0, except for segments of type CODE where the default is 1.
bits	The size of the resource in bits.
cell	The name of a frame cell.
cfa	The name of a CFA (canonical frame address).
cfiexpr	A CFI expression (see *CFI expressions*, page 89).
commonblock	The name of a previously defined common block.

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

DESCRIPTIONS

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

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

Backtrace rows and columns

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

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

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

Defining a names block

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

Start and end a names block with the directives:

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

where `name` is the name of the block.

Only one names block can be open at a time.

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

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

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

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

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

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

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

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

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

  ```
  CFI STACKFRAME cfa resource type
  ```

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

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

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

  ```
  CFI STATICOVERLAYFRAME cfa segment
  ```

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

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

  ```
  CFI BASEADDRESS cfa type
  ```

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

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

Extending a names block

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

Extend an existing names block with the directive:

```
CFI NAMES name EXTENDS namesblock
```

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

```
CFI ENDNAMES name
```

Defining a common block

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

Start a common block with the directive:

```
CFI COMMON name USING namesblock
```

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

Declare the return address column with the directive:

```
CFI RETURNADDRESS resource type
```

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

End a common block with the directive:

```
CFI ENDCOMMON name
```

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

Inside a common block you can declare the initial value of a CFA or a resource by using the directives listed last in *Common block directives*, page 81. For more information on these directives, see *Simple rules*, page 87, and *CFI expressions*, page 89.

Extending a common block

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

Extend an existing common block with the directive:

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

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

```
CFI ENDCOMMON name
```

Defining a data block

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

Start a data block with the directive:

```
CFI BLOCK name USING commonblock
```

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

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

```
CFI FUNCTION label
```

where label is the code label starting the function.

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

```
CFI NOFUNCTION
```

End a data block with the directive:

```
CFI ENDBLOCK name
```

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

Inside a data block you may manipulate the values of the columns by using the directives listed last in *Data block directives*, page 81. For more information on these directives, see *Simple rules*, page 87, and *CFI expressions*, page 89.

SIMPLE RULES

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

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

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

In those rare cases where the descriptive power of the simple rules are not enough, a full CFI expression can be used to describe the information (see *CFI expressions*, page 89). However, whenever possible, you should always use a simple rule instead of a CFI expression.

There are two different sets of simple rules: one for resources and one for CFAs.

Simple rules for resources

The rules for resources conceptually describe where to find a resource when going back one call frame. For this reason, the item following the resource name in a CFI directive is referred to as the *location* of the resource.

To declare that a tracked resource is restored, that is, already correctly located, use SAMEVALUE as the location. Conceptually, this declares that the resource does not have to be restored since it already contains the correct value. For example, to declare that a register REG is restored to the same value, use the directive:

```
CFI REG SAMEVALUE
```

To declare that a resource is not tracked, use UNDEFINED as location. Conceptually, this declares that the resource does not have to be restored (when going back one call frame) since it is not tracked. Usually it is only meaningful to use it to declare the initial location of a resource. For example, to declare that REG is a scratch register and does not have to be restored, use the directive:

```
CFI REG UNDEFINED
```

To declare that a resource is temporarily stored in another resource, use the resource name as its location. For example, to declare that a register REG1 is temporarily located in a register REG2 (and should be restored from that register), use the directive:

```
CFI REG1 REG2
```

To declare that a resource is currently located somewhere on the stack, use FRAME(*cfa, offset*) as location for the resource, where *cfa* is the CFA identifier to use as "frame pointer" and *offset* is an offset relative the CFA. For example, to declare that a register REG is located at offset -4 counting from the frame pointer CFA_SP, use the directive:

```
CFI REG FRAME(CFA_SP,-4)
```

For a composite resource there is one additional location, CONCAT, which declares that the location of the resource can be found by concatenating the resource parts for the composite resource. For example, consider a composite resource RET with resource parts RETLO and RETHI. To declare that the value of RET can be found by investigating and concatenating the resource parts, use the directive:

```
CFI RET CONCAT
```

This requires that at least one of the resource parts has a definition, using the rules described above.

Simple rules for CFAs

In contrast with the rules for resources, the rules for CFAs describe the address of the beginning of the call frame. The call frame often includes the return address pushed by the subroutine calling instruction. The CFA rules describe how to compute the address to the beginning of the current call frame. There are two different forms of CFAs, stack frames and static overlay frames, each declared in the associated names block. See *Names block directives*, page 81.

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

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

```
CFI CFA_SP NOTUSED
```

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

```
CFI CFA_SP SP + 4
```

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

CFI EXPRESSIONS

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

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

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

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

Unary operators

Overall syntax: *OPERATOR*(*operand*)

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

Table 25: Unary operators in CFI expressions

Operator	Operand	Description
LITERAL	*expr*	Get the value of the assembler expression. This can insert the value of a regular assembler expression into a CFI expression.

Table 25: Unary operators in CFI expressions (Continued)

Binary operators

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

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

Table 26: Binary operators in CFI expressions

Ternary operators

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

Operator	Operands	Description
FRAME	*cfa*,*size*,*offset*	Get value from stack frame. The operands are: cfa An identifier denoting a previously declared CFA. size A constant expression denoting a size in bytes. offset A constant expression denoting an offset in bytes. Gets the value at address *cfa*+*offset* of size *size*.
IF	*cond*,*true*,*false*	Conditional operator. The operands are: cond A CFA expression denoting a condition. true Any CFA expression. false Any CFA expression. If the conditional expression is non-zero, the result is the value of the *true* expression; otherwise the result is the value of the *false* expression.
LOAD	*size*,*type*,*addr*	Get value from memory. The operands are: size A constant expression denoting a size in bytes. type A memory type. addr A CFA expression denoting a memory address. Gets the value at address *addr* in segment type *type* of size *size*.

Table 27: Ternary operators in CFI expressions

EXAMPLE

The following is a generic example and not an example specific to the PIC18 microcontroller. This will simplify the example and clarify the usage of the CFI directives. A target-specific example can be obtained by generating assembler output when compiling a C source file.

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

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

Address	CFA	SP	R0	R1	RET	Assembler code
0000	SP + 2		—	SAME	CFA - 2	func1: PUSH R1
0002	SP + 4			CFA - 4		MOV R1,#4

Table 28: Code sample with backtrace rows and columns

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

Table 28: Code sample with backtrace rows and columns (Continued)

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

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

Defining the names block

The names block for the small example above would be:

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

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

Defining the common block

The common block for the simple example above would be:

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

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

Defining the data block

Continuing the simple example, the data block would be:

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

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

#pragma directives

This chapter describes the #pragma directives of the PIC18 IAR Assembler.

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

Summary of #pragma directives

The following table shows the `#pragma` directives of the assembler:

#pragma directive	Description
#pragma diag_default	Changes the severity level of diagnostic messages
#pragma diag_error	Changes the severity level of diagnostic messages
#pragma diag_remark	Changes the severity level of diagnostic messages
#pragma diag_suppress	Suppresses diagnostic messages
#pragma diag_warning	Changes the severity level of diagnostic messages
#pragma message	Prints a message

Table 29: #pragma directives summary

Descriptions of #pragma directives

All `#pragma` directives should be entered like:

```
#pragma pragmaname=pragmavalue
```

or

```
#pragma pragmaname = pragmavalue
```

The memory in which the segment resides is optionally specified using the following syntax:

#pragma diag_default

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

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

See the chapter *Assembler diagnostics* for more information about diagnostic messages.

Example

```
#pragma diag_default=Pe117
```

`#pragma diag_error`

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

Changes the severity level to `error` for the specified diagnostics. See the chapter *Assembler diagnostics* for more information about diagnostic messages.

Example

```
#pragma diag_error=Pe117
```

`#pragma diag_remark`

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

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

```
#pragma diag_remark=Pe177
```

See the chapter *Assembler diagnostics* for more information about diagnostic messages.

`#pragma diag_suppress`

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

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

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

See the chapter *Assembler diagnostics* for more information about diagnostic messages.

`#pragma diag_warning`

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

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

```
#pragma diag_warning=Pe826
```

See the chapter *Assembler diagnostics* for more information about diagnostic messages.

`#pragma message`

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

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

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

Assembler diagnostics

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

Message format

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

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

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

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

Severity levels

The diagnostics are divided into different levels of severity:

Remark

A diagnostic message that is produced when the assembler finds a source code construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued but can be enabled, see *--remarks*, page 19.

Warning

A diagnostic message that is produced when the assembler finds a programming error or omission which is of concern but not so severe as to prevent the completion of compilation. Warnings can be disabled by use of the command-line option `--no_warnings`, see page 18.

Error

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

Fatal error

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

SETTING THE SEVERITY LEVEL

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

See *Summary of assembler options*, page 11, for a description of the assembler options that are available for setting severity levels.

INTERNAL ERROR

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

```
Internal error: message
```

where *message* is an explanatory message. If internal errors occur they should be reported to your software distributor or IAR Technical Support. Please include information enough to reproduce the problem. This would typically include:

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

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Symbols