IAR Embedded Workbench®

IAR Assembler Reference Guide

for the Texas Instruments

MSP430 Microcontroller Family

A430-4

IAR SYSTEMS

DISCLAIMER

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

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

TRADEMARKS

IAR Systems, IAR Embedded Workbench, C-SPY, C-RUN, C-STAT, visualSTATE, Focus on Your Code, IAR KickStart Kit, IAR Experiment!, I-jet, I-jet Trace, I-scope, IAR Academy, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Texas Instruments is a registered trademark of Texas Instruments Corporation. MSP430 is a trademark of Texas Instruments Corporation.

Adobe and Acrobat Reader are registered trademarks of Adobe Systems Incorporated.

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

EDITION NOTICE

Fourth edition: June 2016

Part number: A430-4

This guide applies to version 6.x of IAR Embedded Workbench® for Texas Instruments's MSP430 microcontroller family.

Internal reference: M20, asrct2010.3, IMAE.

Contents

Tables

Preface

Welcome to the IAR Assembler Reference Guide for MSP430. The purpose of this guide is to provide you with detailed reference information that can help you to use the IAR Assembler for MSP430 to develop your application according to your requirements.

Who should read this guide

You should read this guide if you plan to develop an application, or part of an application, using assembler language for the MSP430 microcontroller and need to get detailed reference information on how to use the IAR Assembler for MSP430. In addition, you should have working knowledge of the following:

- The architecture and instruction set of the MSP430 microcontroller (refer to the chip manufacturer's documentation)
- General assembler language programming
- Application development for embedded systems
- The operating system of your host computer.

How to use this guide

When you first begin using the IAR Assembler Reference Guide, you should read the chapter *Introduction to the IAR Assembler for MSP430*.

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 Embedded Workbench, we recommend that you first work through the tutorials, which you can find in the IAR Information Center and which will help you get started using IAR Embedded Workbench.

What this guide contains

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

- *Introduction to the IAR Assembler for MSP430* provides programming information. It also describes the source code format, and the format of assembler listings.
- *Assembler options* first explains how to set the assembler options from the command line and how to use environment variables. It then gives an alphabetical summary of the assembler options, and contains detailed reference information about each option.
- *Assembler operators* gives a summary of the assembler operators, arranged in order of precedence, and provides detailed reference information about each operator.
- *Assembler directives* gives an alphabetical summary of the assembler directives, and provides detailed reference information about each of the directives, classified into groups according to their function.
- *Assembler diagnostics* contains information about the formats and severity levels of diagnostic messages.

Document conventions

When, in the IAR Systems documentation, we refer to the programming language C, the text also applies to C++, unless otherwise stated.

When referring to a directory in your product installation, for example `430\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench` *`N.n`*`\430\doc`, where the initial digit of the version number reflects the initial digit of the version number of the IAR Embedded Workbench shared components.

TYPOGRAPHIC CONVENTIONS

The IAR Systems documentation set uses the following typographic conventions:

Style	Used for
`computer`	• Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
`parameter`	A placeholder for an actual value used as a parameter, for example *`filename`*`.h` where *`filename`* represents the name of the file.
[`option`**]**	An optional part of a directive, where **[** and **]** are not part of the actual directive, but any `[`, `]`, `{`, or `}` are part of the directive syntax.

Table 1: Typographic conventions used in this guide

Style	Used for
{option**}**	A mandatory part of a directive, where **{** and **}** are not part of the actual directive, but any `[`, `]`, `{`, or `}` are part of the directive syntax.
`[option]`	An optional part of a command.
`[a|b|c]`	An optional part of a command with alternatives.
`{a|b|c}`	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
italic	• A cross-reference within this guide or to another guide. • Emphasis.
…	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

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

NAMING CONVENTIONS

The following naming conventions are used for the products and tools from IAR Systems®, when referred to in the documentation:

Brand name	Generic term
IAR Embedded Workbench® for MSP430	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for MSP430	the IDE
IAR C-SPY® Debugger for MSP430	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for MSP430	the compiler
IAR Assembler™ for MSP430	the assembler
IAR XLINK Linker™	XLINK, the linker
IAR XAR Library Builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Runtime Environment™	the DLIB runtime environment

Table 2: Naming conventions used in this guide

Brand name	Generic term
IAR CLIB Runtime Environment™	the CLIB runtime environment

Table 2: Naming conventions used in this guide (Continued)

Introduction to the IAR Assembler for MSP430

- Introduction to assembler programming
- Modular programming
- External interface details
- Source format
- Assembler instructions
- Expressions, operands, and operators
- List file format
- Programming hints
- Tracking call frame usage

Introduction to assembler programming

Even if you do not intend to write a complete application in assembler language, there might be situations where you find it necessary to write parts of the code in assembler, for example, when using mechanisms in the MSP430 microcontroller that require precise timing and special instruction sequences.

To write efficient assembler applications, you should be familiar with the architecture and instruction set of the MSP430 microcontroller. Refer to the Texas Instruments hardware documentation for syntax descriptions of the instruction mnemonics.

GETTING STARTED

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

- Work through the tutorials—especially the one about mixing C and assembler modules—that you find in the Information Center
- Read about the assembler language interface—also useful when mixing C and assembler modules—in the *IAR C/C++ Compiler Reference Guide for MSP430*

- In the IAR Embedded Workbench IDE, you can base a new project on a *template* for an assembler project.

Modular programming

It is widely accepted that modular programming is a prominent feature of good software design. If you structure your code in small modules—in contrast to one single monolith—you can organize your application code in a logical structure, which makes the code easier to understand, and which aids:

- efficient program development
- reuse of modules
- maintenance.

The IAR development tools provide different facilities for achieving a modular structure in your software.

Typically, you write your assembler code in assembler source files. In each source file you define one or several assembler *modules*, using the module control directives. Each module has a name and a type, where the type can be either `PROGRAM` or `LIBRARY`. The linker always includes a `PROGRAM` module, whereas a `LIBRARY` module is only included in the linked code if other modules refer to a public symbol in the module. You can divide each module further into subroutines.

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

If you are working on a large project you will soon accumulate a collection of useful routines that are used by several of your applications. To avoid ending up with a huge amount of small object files, collect modules that contain such routines in a *library* object file. In the IAR Embedded Workbench IDE, you can set up a library project, to collect many object files in one library. For an example, see the tutorials in the Information Center.

To summarize, your software design benefits from modular programming, and to achieve a modular structure you can:

- Create many small modules, either one per source file, or many modules per file by using the module directives

- In each module, divide your assembler source code into small subroutines (corresponding to *functions* on the C level)
- Divide your assembler source code into *segments*, to gain more precise control of how your code and data finally is placed in memory
- Collect your routines in libraries, which means that you can reduce the number of object files and make the modules conditionally linked.

External interface details

This section provides information about how the assembler interacts with its environment:

- *Assembler invocation syntax*, page 15
- *Passing options*, page 16
- *Environment variables*, page 16
- *Error return codes*, page 16

You can use the assembler either from the IAR Embedded Workbench IDE or from the command line. Refer to the *IAR Embedded Workbench® IDE User Guide for MSP430* for information about using the assembler from the IAR Embedded Workbench IDE.

ASSEMBLER INVOCATION SYNTAX

The invocation syntax for the assembler is:

```
a430 [options][sourcefile][options]
```

For example, when assembling the source file `prog.s43`, use this command to generate an object file with debug information:

```
a430 prog -r
```

By default, the IAR Assembler for MSP430 recognizes the filename extensions `s43`, `asm`, and `msa` for source files. The default filename extension for assembler output is `r43`.

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

If you run the assembler from the command line without any arguments, the assembler version number and all available options including brief descriptions are directed to `stdout` and displayed on the screen.

PASSING OPTIONS

You can pass options to the assembler in three different ways:

- Directly from the command line

  Specify the options on the command line after the a430 command; see *Assembler invocation syntax*, page 15.
- Via environment variables

  The assembler automatically appends the value of the environment variables to every command line, so it provides a convenient method of specifying options that are required for every assembly; see *Environment variables*, page 16.
- Via a text file by using the `-f` option; see *-f*, page 42.

For general guidelines for the option syntax, an options summary, and more information about each option, see the *Assembler options* chapter.

ENVIRONMENT VARIABLES

You can use these environment variables with the IAR Assembler:

Environment variable	Description
`ASM430`	Specifies command line options; for example: `set ASM430=-L -ws`
`ASM430_INC`	Specifies directories to search for include files; for example: `set ASM430_INC=c:\myinc\`

Table 3: Assembler environment variables

For example, setting this environment variable always generates a list file with the name `temp.lst`:

```
set ASM430=-l temp.lst
```

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

ERROR RETURN CODES

When using the IAR Assembler from within a batch file, you might have to determine whether the assembly was successful to decide what step to take next. For this reason, the assembler returns these error return codes:

Return code	Description
`0`	Assembly successful, warnings might appear.
`1`	Warnings occurred (only if the `-ws` option is used).

Table 4: Assembler error return codes

Return code	Description
2	Errors occurred.

Table 4: Assembler error return codes (Continued)

Source format

The format of an assembler source line is as follows:

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

where the components are as follows:

label	A definition of a label, which is a symbol that represents an address. If the label starts in the first column—that is, at the far left on the line—the `:` (colon) is optional.
operation	An assembler instruction or directive. This must not start in the first column—there must be some whitespace to the left of it.
operands	An assembler instruction or directive can have zero, one, or more operands. The operands are separated by commas or whitespaces.
comment	Comment, preceded by a `;` (semicolon) C or C++ comments are also allowed.

The components are separated by spaces or tabs.

A source line cannot exceed 2047 characters.

Tab characters, ASCII `09H`, are expanded according to the most common practice; i.e. to columns 8, 16, 24 etc. This affects the source code output in list files and debug information. Because tabs might be set up differently in different editors, do not use tabs in your source files.

Assembler instructions

The IAR Assembler for MSP430 supports the syntax for assembler instructions as described in the Texas Instruments hardware documentation. It complies with the requirement of the MSP430 architecture on word alignment. Any instructions in a code segment placed on an odd address results in an error.

8-bit instructions have the suffix `.b`, 16-bit instructions have the suffix `.w`, and 20-bit instructions have the suffix `.a`.

Expressions, operands, and operators

Expressions consist of expression operands and operators.

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

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators; see also *Assembler operators*.

These operands are valid in an expression:

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

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

INTEGER CONSTANTS

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

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

Commas and decimal points are not permitted.

The following types of number representation are supported:

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

Table 5: Integer constant formats

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

ASCII CHARACTER CONSTANTS

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

Format	Value
`'ABCD'`	`ABCD` (four characters).
`"ABCD"`	`ABCD'\0'` (five characters the last ASCII null).
`'A''B'`	`A'B`
`'A'''`	`A'`
`''''` (4 quotes)	`'`
`''` (2 quotes)	Empty string (no value).
`""` (2 double quotes)	Empty string (an ASCII null character).
`\'`	`'`, for quote within a string, as in `'I\'d love to'`
`\\`	`\`, for `\` within a string
`\"`	`"`, for double quote within a string

Table 6: ASCII character constant formats

FLOATING-POINT CONSTANTS

The IAR Assembler accepts floating-point values as constants and converts 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`*`]`

This 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 7: Floating-point constants

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

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

The MSP430 single and double precision floating-point format

The IAR Assembler for MSP430 supports the Texas Instruments single and double precision floating-point format. For a description of this format, see the MSP430 documentation provided by Texas Instruments.

TRUE AND FALSE

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

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

SYMBOLS

User-defined symbols can be up to 255 characters long, and all characters are significant. Depending on what kind of operation a symbol is followed by, the symbol is either a data symbol or an address symbol where the latter is referred to as a label. A symbol before an instruction is a label and a symbol before, for example the `EQU` directive, is a data symbol. A symbol can be:

- absolute—its value is known by the assembler
- relocatable—its value is resolved at link time.

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

Symbols may contain any printable characters if they are quoted with ` (backquote), for example:

```
`strange#label`
```

Case is insignificant for built-in symbols like instructions, registers, operators, and directives. For user-defined symbols, case is by default significant but can be turned on and off using the **Case sensitive user symbols** (`-s`) assembler option. For more information, see *-s*, page 51.

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

Note that symbols and labels are byte addresses. For more information, see Data definition or allocation directives, page 111.

LABELS

Symbols used for memory locations are referred to as labels.

Program location counter (PLC)

The assembler keeps track of the start address of the current instruction. This is called the *program location counter*.

If you must refer to the program location counter in your assembler source code, use the $ (dollar) sign. For example:

```
        BR      $         ; Loop forever
```

REGISTER SYMBOLS

This table shows the existing predefined register symbols:

Name	Size	Description
R4–R15	16 bits	General purpose registers
PC	16 bits	Program counter
SP	16 bits	Stack pointer
SR	16 bits	Status register

Table 8: Predefined register symbols

PREDEFINED SYMBOLS

The 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.

These predefined symbols are available:

Symbol	Value
`__A430__`	An integer that is set to `1` when the code is assembled with the IAR Assembler for MSP430.
`__BUILD_NUMBER__`	A unique integer that identifies the build number of the assembler currently in use. The build number does not necessarily increase with an assembler that is released later.
`__CORE__`	An integer that identifies the processor core in use. The symbol reflects the `-v` option and is defined to `__430_CORE__` for the MSP430 architecture and to `__430X_CORE__` for the MSP430X architecture. These symbolic names can be used when testing the `__CORE__` symbol.

Table 9: Predefined symbols

Symbol	Value
`__CODE_MODEL__`	An integer that identifies the code model. The symbol reflects the `--code_model` option and is defined to `__CODE_MODEL_SMALL__` for the small code model and to `__CODE_MODEL_LARGE__` for the large code model. These symbolic names can be used when testing the `__CODE_MODEL__` symbol.
`__DATA_MODEL__`	An integer that identifies the data model. The symbol reflects the `--data_model` option and is defined to one of `__DATA_MODEL_SMALL__`, `__DATA_MODEL_MEDIUM__`, and `__DATA_MODEL_LARGE__`. These symbolic names can be used when testing the `__DATA_MODEL__` symbol.
`__DATE__`	The current date in `dd/Mmm/yyyy` format (string).
`__FILE__`	The name of the current source file (string).
`__IAR_SYSTEMS_ASM__`	IAR assembler identifier (number). Note that the number could be higher in a future version of the product. This symbol can be tested with `#ifdef` to detect whether the code was assembled by an assembler from IAR Systems.
`__LINE__`	The current source line number (number).
`__REGISTER_MODEL__`	An integer that identifies whether the data model supports 20-bit registers. For the Small data model, this is equal to `__REGISTER_MODEL_REG16__` and for the Medium and Large data models, it is equal to `__REGISTER_MODEL_REG20__`. These symbolic names can be used when testing the `__REGISTER_MODEL__` symbol.
`__ROPI__`	The integer `1` when the `--ropi` command line option is used, and undefined otherwise.
`__TID__`	Target identity, consisting of two bytes (number). The high byte is the target identity, which is `43` for `a430`.
`__SUBVERSION__`	An integer that identifies the subversion number of the assembler version number, for example 3 in 1.2.3.4.
`__TIME__`	The current time in `hh:mm:ss` format (string).
`__VER__`	The version number in integer format; for example, version 4.17 is returned as 417 (number).

Table 9: Predefined symbols (Continued)

Note: The symbol `__TID__` is related to the predefined symbol `__TID__` in the IAR C/C++ Compiler for MSP430. It is described in the *IAR C/C++ Compiler Reference Guide for MSP430*.

Including symbol values in code

Several data definition directives make it possible to include a symbol value in the code. These directives define values or reserve memory. To include a symbol value in the code, use the symbol in the appropriate data definition directive.

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

```
            name    timeOfAssembly
            extern  printStr
            rseg    CODE:CODE

printTime   mov.w   #time, r12      ; Load address of time string
                                    ; in r12
            call    #printStr       ; Call string output routine.
            ret
            rseg    DATA16_C:DATA
time:       dc8     __TIME__        ; String representing the
                                    ; time of assembly.
            end
```

Testing symbols for conditional assembly

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

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

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

For more information, see Conditional assembly directives, page 91.

ABSOLUTE AND RELOCATABLE EXPRESSIONS

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

Expressions that include symbols in relocatable segments cannot be resolved at assembly time, because they depend on the location of segments. These are referred to as relocatable expressions.

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 absolute and relocatable expressions as follows:

```
            name    simpleExpressions
            rseg    CONST:CONST
            extern  size
first       dc8     5                 ; A relocatable label.
second      equ     10 + 5            ; An absolute expression.

            dc8     first             ; Examples of some legal
            dc8     first + 1         ; relocatable expressions.
            dc8     first + second
            dc8     first + 8 * size
            end
```

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

EXPRESSION RESTRICTIONS

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

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

No forward

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

No external

No external references in the expression are allowed.

Absolute

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

Fixed

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

List file format

The format of an assembler list file is as follows:

HEADER

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

BODY

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

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

SUMMARY

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

SYMBOL AND CROSS-REFERENCE TABLE

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

This information is provided for each symbol in the table:

Information	Description
Symbol	The symbol's user-defined name.
Mode	ABS (Absolute), or REL (Relocatable).
Segments	The name of the segment that this symbol is defined relative to.

Table 10: Symbol and cross-reference table

Information	Description
Value/Offset	The value (address) of the symbol within the current module, relative to the beginning of the current segment part.

Table 10: Symbol and cross-reference table (Continued)

Programming hints

This section gives hints on how to write efficient code for the IAR Assembler. For information about projects including both assembler and C or C++ source files, see the *IAR C/C++ Compiler Reference Guide for MSP430*.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several MSP430 devices are included in the IAR Systems product package, in the `430\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 IAR C/C++ Compiler for MSP430.

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

```
#ifdef __IAR_SYSTEMS_ASM__
  ; Add your assembler-specific defines here.
#endif
```

USING C-STYLE PREPROCESSOR DIRECTIVES

The C-style preprocessor directives are processed before other assembler directives. Therefore, do not use preprocessor directives in macros and do not mix them with assembler-style comments. For more information about comments, see Assembler control directives, page 114.

C-style preprocessor directives like `#define` are valid in the remainder of the source code file, while assembler directives like `EQU` only are valid in the current module.

Tracking call frame usage

In this section, these topics are described:

- *Call frame information overview*, page 27
- *Call frame information in more detail*, page 28

These tasks are described:

- *Defining a names block*, page 28

For reference information, see:

CALL FRAME INFORMATION OVERVIEW

Call frame information (CFI) is information about the *call frames*. Typically, a call frame contains a return address, function arguments, saved register values, compiler temporaries, and local variables. Call frame information holds enough information about call frames to support two important features:

- C-SPY can use call frame information to reconstruct the entire call chain from the current `PC` (program counter) and show the values of local variables in each function in the call chain.
- Call frame information can be used, together with information about possible calls for calculating the total stack usage in the application. Note that this feature might not be supported by the product you are using.

The compiler automatically generates call frame information for all C and C++ source code. Call frame information is also typically provided for each assembler routine in the system library. However, if you have other assembler routines and want to enable C-SPY to show the call stack when executing these routines, you must add the required call frame information annotations to your assembler source code. Stack usage can also be handled this way (by adding the required annotations for each function call), but you can also specify stack usage information for any routines in a *stack usage control file* (see the *IAR C/C++ Compiler Reference Guide for MSP430*), which is typically easier.

CALL FRAME INFORMATION IN MORE DETAIL

You can add call frame information to assembler files by using `cfi` directives. You can use these to specify:

- The *start address* of the call frame, which is referred to as the *canonical frame address* (CFA). There are two different types of call frames:
  - On a stack—*stack frames*. For stack frames the CFA is typically the value of the stack pointer after the return from the routine.
  - In static memory, as used in a static overlay system—*static overlay frames*. This type of call frame is not required by the MSP430 microcontroller and is thus not supported.
- How to find the return address.
- How to restore various resources, like registers, when returning from the routine.

When adding the call frame information for each assembler module, you must:

1. Provide a *names block* where you describe the resources to be tracked.
2. Provide a *common block* where you define the resources to be tracked and specify their default values. This information must correspond to the calling convention used by the compiler.
3. Annotate the resources used in your source code, which in practice means that you describe the changes performed on the call frame. Typically, this includes information about when the stack pointer is changed, and when permanent registers are stored or restored on the stack.

   To do this you must define a *data block* that encloses a continuous piece of source code where you specify *rules* for each resource to be tracked. When the descriptive power of the rules is not enough, you can instead use *CFI expressions*.

A full description of the calling convention might require extensive call frame information. In many cases, a more limited approach will suffice. The recommended way to create an assembler language routine that handles call frame information correctly is to start with a C skeleton function that you compile to generate assembler output. For an example, see the *IAR C/C++ Compiler Reference Guide for MSP430.*

DEFINING A NAMES BLOCK

A *names block* is used for declaring 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 can appear: a resource declaration, a stack frame declaration, a static overlay frame declaration, and 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.

  To declare more than one resource, separate them with commas.

  A resource can 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 memory type (to get the address space). To declare more than one stack frame CFA, separate 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.

DEFINING A COMMON BLOCK

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

Start a common block with the directive:

```
CFI COMMON name USING namesblock
```

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

Declare the return address column with the directive:

```
CFI RETURNADDRESS resource type
```

where *resource* is a resource defined in *namesblock* and *type* is the memory in which the calling function resides. You must declare the return address column for the common block.

Inside a common block, you can declare the initial value of a CFA or a resource by using the directives available for common blocks, see Call frame information directives for common blocks, page 119. For more information about how to use these directives, see *Specifying rules for tracking resources and the stack depth*, page 31 and *Using CFI expressions for tracking complex cases*, page 33.

End a common block with the directive:

```
CFI ENDCOMMON name
```

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

ANNOTATING YOUR SOURCE CODE WITHIN A DATA BLOCK

The *data block* contains the actual tracking information for one continuous piece of code.

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 for the current data block 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 for the current data block 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 can manipulate the values of the resources by using the directives available for data blocks, see Call frame information directives for data blocks, page 120. For more information on how to use these directives, see *Specifying rules for tracking resources and the stack depth*, page 31, and *Using CFI expressions for tracking complex cases*, page 33.

SPECIFYING RULES FOR TRACKING RESOURCES AND THE STACK DEPTH

To describe the tracking information for individual resources, two sets of simple rules with specialized syntax can be used:

- Rules for tracking resources

  ```
  CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
  CFI resource { resource | FRAME(cfa, offset) }
  ```

- Rules for tracking the stack depth (CFAs)

  ```
  CFI cfa { NOTUSED | USED }
  CFI cfa { resource | resource + constant | resource - constant }
  ```

You can use these rules 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, you can use a full *CFI expression* with dedicated *operators* to describe the information, see *Using CFI expressions for tracking complex cases*, page 33. However, whenever possible, you should always use a rule instead of a CFI expression.

Rules for tracking 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, in other words, already correctly located, use `SAMEVALUE` as the location. Conceptually, this declares that the resource does not have to be restored because it already contains the correct value. For example, to declare that a register `R11` is restored to the same value, use the directive:

```
CFI R11 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) because 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 `R11` is a scratch register and does not have to be restored, use the directive:

```
CFI R11 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 R11 is temporarily located in a register R12 (and should be restored from that register), use the directive:

```
CFI R11 R12
```

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 R11 is located at offset –4 counting from the frame pointer CFA_SP, use the directive:

```
CFI R11 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.

Rules for tracking the stack depth (CFAs)

In contrast to 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 assembler call instruction. The CFA rules describe how to compute the address of the beginning of the current stack frame.

Each stack frame CFA is associated with a stack pointer. When going back one call frame, the associated stack pointer is restored to the current CFA. For stack frame CFAs there are two possible 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 stack pointer 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 stack pointer 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
```

USING CFI EXPRESSIONS FOR TRACKING COMPLEX CASES

You can use *call frame information expressions* (CFI expressions) when the descriptive power of the rules for resources and CFAs is not enough. However, you should always use a simple rule if there is one.

CFI expressions consist of operands and operators. Three sets of operators are allowed in a CFI expression:

- Unary operators
- Binary operators
- Ternary operators

In most cases, they have an equivalent operator in the regular assembler expressions.

In this example, `R12` is restored to its original value. However, instead of saving it, the effect of the two post increments is undone by the subtract instruction.

```
AddTwo:
        cfi block addTwoBlock using myCommon
        cfi function addTwo
        cfi nocalls
        cfi r12 samevalue
        add @r12+, r13
        cfi r12 sub(r12, 2)
        add @r12+, r13
        cfi r12 sub(r12, 4)
        sub #4, r12
        cfi r12 samevalue
        ret
        cfi endblock addTwoBlock
```

For more information about the syntax for using the operators in CFI expressions, see Call frame information directives for tracking resources and CFAs, page 121.

STACK USAGE ANALYSIS DIRECTIVES

The stack usage analysis directives (`CFI FUNCALL`, `CFI TAILCALL`, `CFI INDIRECTCALL`, and `CFI NOCALLS`) are used for building a call graph which is needed for stack usage analysis. These directives can be used only in data blocks. When the data block is a function block (in other words, when the `CFI FUNCTION` directive has been used in the data block), you should not specify a *`caller`* parameter. When a stack usage analysis directive is used in code that is shared between functions, you must use the *`caller`* parameter to specify which of the possible functions the information applies to.

The `CFI FUNCALL`, `CFI TAILCALL`, and `CFI INDIRECTCALL` directives must be placed immediately before the instruction that performs the call. The `CFI NOCALLS` directive can be placed anywhere in the data block.

EXAMPLES OF USING CFI DIRECTIVES

The following is a generic example of how to add and use the required CFI directives. The example is not specific to the MSP430 microcontroller. To obtain an example specific to the microcontroller you are using, generate assembler output when you compile a C source file.

Consider a generic processor with a stack pointer `SP`, and two registers `R0` and `R1`. Register `R0` is used as a scratch register (the register may be destroyed by a function call), whereas register `R1` must be restored after the function call. To simplify, all instructions, registers, and addresses are assumed to have a width of 16 bits.

Consider the following short code example with the corresponding call frame information. At entry, assume that the stack contains a 16-bit return address. The stack grows from high addresses toward zero. The CFA denotes the top of the call frame, in other words, the value of the stack pointer after returning from the function.

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

Table 11: Code sample with call frame information

Each 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 row at address `0000` is the initial row and the result of the calling convention used for the function.

The RET column is the return address column—that is, the location of the return address. The value of `R0` is undefined because it 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

            ; Offset -2 from top of frame.
            cfi  RET frame(CFA,-2)
            cfi     endcommon trivialCommon
```

Note: `SP` cannot be changed using a `CFI` directive as it is the resource associated with `CFA`.

Annotating your source code within a data block

You should place the `CFI` directives at the point where the call frame information has changed, in other words, immediately *after* the instruction that changes the call frame information.

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
```

Assembler options

- Using command line assembler options
- Summary of assembler options
- Description of assembler options

Using command line assembler options

Assembler options are parameters you can specify to change the default behavior of the assembler. You can specify options from the command line—which is described in more detail in this section—and from within the IAR Embedded Workbench® IDE.

The IAR Embedded Workbench® IDE User Guide for MSP430 describes how to set assembler options in the IDE, and gives reference information about the available options.

SPECIFYING OPTIONS AND THEIR PARAMETERS

To set assembler options from the command line, include them after the a430 command:

```
a430 [options] [sourcefile] [options]
```

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

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

For example, when assembling the source file `power2.s43`, use this command to generate a list file to the default filename (`power2.lst`):

```
a430 power2.s43 -L
```

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

```
a430 power2.s43 -l list.lst
```

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

```
a430 power2.s43 -Llist\
```

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

EXTENDED COMMAND LINE FILE

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

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

```
a430 -f extend.xcl
```

Summary of assembler options

This table summarizes the assembler options available from the command line:

Command line option	Description
`-B`	Macro execution information
`-c`	Conditional list
`--code_model`	Specifies the code model to use
`-D`	Defines preprocessor symbols
`--data_model`	Specifies the data model to use
`-E`	Maximum number of errors
`-f`	Extends the command line
`-G`	Opens standard input as source
`-g`	Disables the automatic search for system include files
`-h`	Enables workaround for hardware issue CPU6
`--hw_workaround`	Enables workarounds for various hardware issues
`-I`	Adds a search path for a header file
`-i`	Lists #included text
`-L`	Generates a list file to path
`-l`	Generates a list file
`-M`	Macro quote characters
`--macro_positions_in_diagnostics`	Obtains positions inside macros in diagnostic messages
`-N`	Omits header from the assembler listing
`-n`	Enables support for multibyte characters

Table 12: Assembler options summary

Command line option	Description
`--no_path_in_file_macros`	Removes the path from the return value of the symbols `__FILE__` and `__BASE_FILE__`
`--no_ubrof_messages`	Suppresses UBROF error messages in object files
`-O`	Sets the object filename to path
`-o`	Sets the object filename
`-p`	Sets the number of lines per page in the list file
`-r`	Generates debug information.
`--ropi`	Specifies position-independent code and read-only data
`-S`	Sets silent operation
`-s`	Case-sensitive user symbols
`--system_include_dir`	Specifies the path for system include files
`-t`	Tab spacing
`-U`	Undefines a symbol
`-v`	Selects the processor core
`-w`	Disables warnings
`-x`	Includes cross-references

Table 12: Assembler options summary (Continued)

Description of assembler options

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

Note that if you use the page **Extra Options** to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

-B

Syntax `-B`

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

- The name of the macro
- The definition of the macro
- The arguments to the macro

- The expanded text of the macro.

This option is mainly used in conjunction with the list file options `-L` or `-l`.

See also

-L, page 45.

Project>Options>Assembler >List>Macro execution info

-c

Syntax

`-c{D|M|E|A|O}`

Parameters

`D`	Disables list file
`M`	Includes macro definitions
`E`	Excludes macro expansions
`A`	Includes assembled lines only
`O`	Includes multiline code

Description

Use this option to control the contents of the assembler list file.

This option is mainly used in conjunction with the list file options `-L` or `-l`.

See also

-L, page 45.

To set related options, select:

Project>Options>Assembler >List

--code_model

Syntax

`--code_model{small|large}`

Parameters

`small`	Specifies the Small code model.
`large`	Specifies the Large code model.

Description

Use this option to specify the code model to use. Effectively, this option defines the predefined preprocessor symbol `__CODE_MODEL__`.

See also

Predefined symbols, page 21

To set this option, use **Project>Options>Assembler>Extra Options**.

-D

Syntax

`-Dsymbol[=value]`

Parameters

symbol	The name of the symbol you want to define.
value	The value of the symbol. If no value is specified, `1` is used.

Description

Use this option to define a symbol to be used by the preprocessor.

Example

You might want to arrange your source code to produce either the test version or the production version of your application, depending 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 on the command line as follows:

Production version:	`a430 prog`
Test version:	`a430 prog -DTESTVER`

Alternatively, your source might use a variable that you must 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:

```
a430 prog -DFRAMERATE=3
```

Project>Options>Assembler>Preprocessor>Defined symbols

--data_model

Syntax

`--data_model{small|medium|large}`

Parameters

`small`	Specifies the Small data model.
`medium`	Specifies the Medium data model.

`large`	Specifies the Large data model.

Description

Use this option to specify the data model to use. Effectively, this option defines the predefined preprocessor symbol `__DATA_MODEL__`.

See also

Predefined symbols, page 21

To set this option, use **Project>Options>Assembler>Extra Options**.

-E

Syntax

`-E`*`number`*

Parameters

`number`	The number of errors before the assembler stops the assembly. *`number`* must be a positive integer; `0` indicates no limit.

Description

Use this option to specify the maximum number of errors that the assembler reports. By default, the maximum number is 100.

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

-f

Syntax

`-f` *`filename`*

Parameters

`filename`	The commands that you want to extend the command line with are read from the specified file. Notice that there must be a space between the option itself and the filename.

For information about specifying a filename, see *Using command line assembler options*, page 37.

Description

Use this option to extend the command line with text read from the specified file.

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

Example

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

```
a430 prog -f extend.xcl
```

To set this option, use:

Project>Options>Assembler>Extra Options

-G

Syntax

```
-G
```

Description

Use this option to make the assembler read the source from the standard input stream, rather than from a specified source file.

When `-G` is used, you cannot specify a source filename.

This option is not available in the IDE.

-g

Syntax

```
-g
```

Description

By default, the assembler automatically locates the system include files. Use this option to disable the automatic search for system include files. In this case, you might need to set up the search path by using the `-I` assembler option.

Project>Options>Assembler>Preprocessor>Ignore standard include directories

-h

Syntax

```
-h
```

Description

Use this option to enable an assembler workaround for the hardware issue CPU6. When enabled, the assembler will issue an error message if it detects an operand that could trigger the hardware issue CPU6.

Note: This option is not enabled automatically by the IAR Embedded Workbench IDE.

See also

For more information about the available workarounds for different hardware issues, see the release notes.

To set this option, **use Project>Options>Assembler>Extra Options**.

--hw_workaround

Syntax

```
--hw_workaround=nop_after_lpm
```

Parameters

`nop_after_lpm`	Workaround for hardware issues CPU18, CPU19, CPU24, CPU25, CPU27, and CPU 29

Description

Use this option to enable assembler workarounds for various hardware issue. Typically, the assembler will issue a warning message if it detects a code sequence that could trigger a hardware issue.

See also

For more information about the available workarounds for different hardware issues, see the release notes.

When you select a device in the IAR Embedded Workbench IDE, the relevant hardware workarounds are enabled automatically.

-I

Syntax

`-I`*`path`*

Parameters

`path`	The search path for `#include` files.

Description

Use this option to specify paths to be used by the preprocessor. This option can be used more than once on the command line.

By default, the assembler searches for `#include` files in the current working directory, in the system header directories, and in the paths specified in the `IASM430_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 code, 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 `ASM430_INC` environment variable, provided that this variable is set, and in the system header directories.

Project>Options>Assembler>Preprocessor>Additional include directories

-i

Syntax

`-i`

Description

Use this option to list `#include` files in the list file.

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

Project>Options>Assembler >List>#included text

-L

Syntax

`-L[`*`path`*`]`

Parameters

No parameter	Generates a listing with the same name as the source file, but with the filename extension `lst`.
`path`	The path to the destination of the list file. Note that you must not include a space before the path.

Description

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

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

Example

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

```
a430 prog -Llist\
```

To set related options, select:

Project>Options>Assembler >List

-l

Syntax `-l` *`filename`*

Parameters

`filename`	The output is stored in the specified file. Note that you must include a space before the filename. If no extension is specified, `lst` is used.

For information about specifying a filename, see *Using command line assembler options*, page 37.

Description Use this option to make the assembler generate a listing and send it to the file *`filename`*. By default, the assembler does not generate a list file.

To generate a list file with the default filename, use the `-L` option instead.

To set related options, select:

Project>Options>Assembler >List

-M

Syntax `-M`*`ab`*

Parameters

`ab`	The characters to be used as left and right quotes of each macro argument, respectively.

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

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

Example For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

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

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

```
a430 filename -M'<>'
```

Project>Options>Assembler >Language>Macro quote characters

--macro_positions_in_diagnostics

Syntax `--macro_positions_in_diagnostics`

Description Use this option to obtain position references inside macros in diagnostic messages. This is useful for detecting incorrect source code constructs in macros.

To set this option, use **Project>Options>Assembler>Extra Options**.

-N

Syntax `-N`

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

This option is useful in conjunction with the list file options `-L` or `-l`.

See also *-L*, page 45.

Project>Options>Assembler >List>Include header

-n

Syntax `-n`

Description By default, multibyte characters cannot be used in assembler source code. Use this option to interpret multibyte characters in the source code according to the host computer's default setting for multibyte support.

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

Project>Options>Assembler >Language>Enable multibyte support

--no_path_in_file_macros

Syntax `--no_path_in_file_macros`

Description Use this option to exclude the path from the return value of the predefined preprocessor symbols `__FILE__` and `__BASE_FILE__`.

This option is not available in the IDE.

--no_ubrof_messages

Syntax `--no_ubrof_messages`

Description Use this option to minimize the size of your application object file by excluding messages from the UBROF files. The file size can decrease by up to 60%. Note that the XLINK diagnostic messages will, however, be less useful when you use this option.

To set this option, use **Project>Options>Assembler>Extra Options**.

-O

Syntax `-O[`*`path`*`]`

Parameters

path	The path to the destination of the object file. Note that you must not include a space before the path.

Description Use this option to set the path to be used on the name of the object file.

By default, the path is null, so the object filename corresponds to the source filename. The `-O` option lets you specify a path, for example, to direct the object file to a subdirectory.

Note that `-O` cannot be used at the same time as `-o`.

Example

To send the object code to the file `obj\prog.r43` rather than to the default file `prog.r43`:

```
a430 prog -Oobj\
```

Project>Options>General Options>Output>Output directories>Object files

-o

Syntax

`-o {`*`filename`*`|`*`directory`*`}`

Parameters

filename	The object code is stored in the specified file.
directory	The object code is stored in a file (filename extension `o`) which is stored in the specified directory.

For information about specifying a filename or directory, see *Using command line assembler options*, page 37.

Description

By default, the object code produced by the assembler is located in a file with the same name as the source file, but with the extension `o`. Use this option to specify a different output filename for the object code.

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

Project>Options>General Options>Output>Output directories>Object files

-p

Syntax

`-p`*`lines`*

Parameters

lines	The number of lines per page, which must be in the range `10` to `150`.

Description

Use this option to set the number of lines per page explicitly.

This option is used in conjunction with the list options `-L` or `-l`.

See also *-L*, page 45.

Project>Options>Assembler>List>Lines/page

-r

Syntax `-r`

Description Use this option to make the assembler generate debug information, which means the generated output can be used in a symbolic debugger such as IAR C-SPY® Debugger.

Project>Options>Assembler >Output>Generate debug information

--ropi

Syntax `--ropi`

Description Use this option to specify that the code is intended for position-independent code and read-only data. Effectively, when this option is specified, the predefined preprocessor symbol `__ROPI__` is defined to `1`.

See also *Predefined symbols*, page 21

To set this option, use **Project>Options>Assembler>Extra Options**.

-S

Syntax `-S`

Description By default, the assembler sends various minor messages via the standard output stream. Use this option to make the assembler operate without sending any messages to the standard output stream.

The assembler sends error and warning messages to the error output stream, so they are displayed regardless of this setting.

This option is not available in the IDE.

-s

Syntax `-s{+|-}`

Parameters

`+`	Case-sensitive user symbols.
`-`	Case-insensitive user symbols.

Description Use this option to control whether the assembler is sensitive to the case of user symbols. By default, case sensitivity is on.

Example By default, for example `LABEL` and `label` refer to different symbols. When `-s-` is used, `LABEL` and `label` instead refer to the same symbol.

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

--system_include_dir

Syntax `--system_include_dir` *`path`*

Parameters

`path`	The path to the system include files.

Description By default, the assembler automatically locates the system include files. Use this option to explicitly specify a different path to the system include files. This might be useful if you have not installed IAR Embedded Workbench in the default location.

This option is not available in the IDE.

-t

Syntax `-t`*`n`*

Parameters

`n`	The tab spacing; must be in the range `2` to `9`.

Description By default, the assembler sets 8 character positions per tab stop. Use this option to specify a different tab spacing.

This option is useful in conjunction with the list options `-L` or `-l`.

See also

-L, page 45.

Project>Options>Assembler>List>Tab spacing

-U

Syntax

`-Usymbol`

Parameters

symbol	The predefined symbol to be undefined.

Description

By default, the assembler provides certain predefined symbols.

Use this option to undefine such a predefined symbol to make its name available for your own use through a subsequent `-D` option or source definition.

Example

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

```
a430 prog -U__TIME__
```

See also

Predefined symbols, page 21.

This option is not available in the IDE.

-v

Syntax

`-v[0|1]`

Parameters

`0`	Specifies devices based on the MSP430 architecture.
`1`	Specifies devices based on the MSP430X architecture.

Description

Use this option to select the architecture for which the code is to be generated. If no processor core option is specified, the assembler uses the `-v0` option by default.

Project>Options>General Options>Target>Device

-w

Syntax

`-w[+|-|+n|-n|+m-n|-m-n][s]`

Parameters

No parameter	Disables all warnings.
+	Enables all warnings.
-	Disables all warnings.
+n	Enables just warning *n*.
-n	Disables just warning *n*.
+m-n	Enables warnings *m* to *n*.
-m-n	Disables warnings *m* to *n*.
`s`	Generates the exit code `1` if a warning message is produced. By default, warnings generate exit code `0`.

Description

By default, the assembler displays a warning message when it detects an element of the source code which is legal in a syntactical sense, but might contain a programming error.

Use this option to disable all warnings, a single warning, or a range of warnings.

Note that the `-w` option can only be used once on the command line.

Example

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

```
a430 prog -w-0
```

To disable warnings 0 to 8, use this command:

```
a430 prog -w-0-8
```

See also

Assembler diagnostics, page 125.

To set related options, select:

Project>Options>Assembler>Diagnostics

-x

Syntax `-x{D|I|2}`

Parameters

`D`	Includes preprocessor #defines.
`I`	Includes internal symbols.
`2`	Includes dual-line spacing.

Description

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

This option is useful in conjunction with the list options `-L` or `-l`.

See also

-L, page 45.

Project>Options>Assembler>List>Include cross reference

Assembler operators

- Precedence of assembler operators
- Summary of assembler operators
- Description of assembler operators

Precedence of assembler operators

Each operator has a precedence number assigned to it that determines the order in which the operator and its operands are evaluated. The precedence numbers range from 1 (the highest precedence, that is, first evaluated) to 7 (the lowest precedence, that is, last evaluated).

These 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, this expression evaluates to 1:

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

Summary of assembler operators

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

PARENTHESIS OPERATOR

Precedence: 1

`()`	Parenthesis.

UNARY OPERATORS

Precedence: 1

`+`	Unary plus.
`-`	Unary minus.
`!`, `NOT`	Logical NOT.
`~`, `BITNOT`	Bitwise NOT.
`LOW`	Low byte.
`HIGH`	High byte.
`LWRD`	Low word.
`HWRD`	High word.
`DATE`	Current time/date.
`SFB`	Segment begin.
`SFE`	Segment end.
`SIZEOF`	Segment size.

MULTIPLICATIVE ARITHMETIC OPERATORS

Precedence: 2

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

ADDITIVE ARITHMETIC OPERATORS

Precedence: 3

`+`	Addition.
`-`	Subtraction.

SHIFT OPERATORS

Precedence: 4

`>>`, `SHR` — Logical shift right.

`<<`, `SHL` — Logical shift left.

AND OPERATORS

Precedence: 5

`&&`, `AND` — Logical AND.

`&`, `BITAND` — Bitwise AND.

OR OPERATORS

Precedence: 6

`||`, `OR` — Logical OR.

`|`, `BITOR` — Bitwise OR.

`XOR` — Logical exclusive OR.

`^`, `BITXOR` — Bitwise exclusive OR.

COMPARISON OPERATORS

Precedence: 7

`=`, `==`, `EQ` — Equal.

`<>`, `!=`, `NE` — Not equal.

`>`, `GT` — Greater than.

`<`, `LT` — Less than.

`UGT` — Unsigned greater than.

`ULT` — Unsigned less than.

`>=`, `GE` — Greater than or equal.

`<=`, `LE` — Less than or equal.

Description of assembler operators

This section gives detailed descriptions of each assembler operator.

See also *Expressions, operands, and operators*, page 18.

()Parenthesis

Precedence

1

Description

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

Example

```
1+2*3 -> 7
(1+2)*3 -> 9
```

* Multiplication

Precedence

2

Description

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

Example

```
2*2 -> 4
-2*2 -> -4
```

+ Unary plus

Precedence

1

Description

Unary plus operator.

Example

```
+3 -> 3
3*+2 -> 6
```

+ Addition

Precedence

3

Description

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

Example

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

– Unary minus

Precedence 1

Description The unary minus operator performs arithmetic negation on its operand.

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

Example

```
-3 -> -3
3*-2 -> -6
4--5 -> 9
```

– Subtraction

Precedence 3

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

Example

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

/ Division

Precedence 2

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

Example

```
9/2 -> 4
-12/3 -> -4
9/2*6 -> 24
```

< Less than

Precedence

7

Description

< or LT evaluates to 1 (true) if the left operand has a lower numeric value than the right operand, otherwise it is 0 (false).

Example

```
-1 < 2 -> 1
2 < 1 -> 0
2 < 2 -> 0
```

<= Less than or equal

Precedence

7

Description

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

Example

```
1 <= 2 -> 1
2 <= 1 -> 0
1 <= 1 -> 1
```

<>, != Not equal

Precedence

7

Description

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

Example

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

=, == Equal

Precedence

7

Description

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

Example

```
1 = 2 -> 0
2 == 2 -> 1
'ABC' = 'ABCD' -> 0
```

> Greater than

Precedence 7

Description

`>` or `GT` evaluates to 1 (true) if the left operand has a higher numeric value than the right operand, otherwise it is 0 (false).

Example

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

>= Greater than or equal

Precedence 7

Description

`>=` or `GE` evaluates to 1 (true) if the left operand is equal to or has a higher numeric value than the right operand, otherwise it is 0 (false).

`>=` evaluates to 1 (true) if the left operand is equal to or has a higher numeric value than the right operand, otherwise it is 0 (false).

Example

```
1 >= 2 -> 0
2 >= 1 -> 1
1 >= 1 -> 1
```

&& Logical AND

Precedence 5

Description

Use `&&` or `AND` to perform logical AND between its two integer operands. If both operands are non-zero the result is 1 (true), otherwise it is 0 (false).

Example

```
1010B && 0011B -> 1
1010B && 0101B -> 1
1010B && 0000B -> 0
```

& Bitwise AND

Precedence 5

Description Use& or BITAND to perform bitwise AND between the integer operands. Each bit in the 32-bit result is the logical AND of the corresponding bits in the operands.

Example

```
1010B & 0011B -> 0010B
1010B & 0101B -> 0000B
1010B & 0000B -> 0000B
```

~ Bitwise NOT)

Precedence 1

Description Use ~ or BITNOT to perform bitwise NOT on its operand. Each bit in the 32-bit result is the complement of the corresponding bit in the operand.

Example

```
~ 1010B -> 11111111111111111111111111110101B
```

| Bitwise OR

Precedence 6

Description Use | or BITOR to perform bitwise OR on its operands. Each bit in the 32-bit result is the inclusive OR of the corresponding bits in the operands.

Example

```
1010B | 0101B -> 1111B
1010B | 0000B -> 1010B
```

^ Bitwise exclusive OR

Precedence 6

Description Use ^ or BITXOR to perform bitwise XOR on its operands. Each bit in the 32-bit result is the exclusive OR of the corresponding bits in the operands.

Example

```
1010B ^ 0101B -> 1111B
1010B ^ 0011B -> 1001B
```

% Modulo

Precedence 2

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

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

Example

```
2 % 2 -> 0
12 % 7 -> 5
3 % 2 -> 1
```

! Logical NOT

Precedence 1

Description Use `!` or `NOT` to negate a logical argument.

Example

```
! 0101B -> 0
! 0000B -> 1
```

|| Logical OR

Precedence 6

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

Example

```
1010B || 0000B -> 1
0000B || 0000B -> 0
```

<< Logical shift left

Precedence 4

Description Use `<<` or `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 << 3 -> 11100000B
0000011111111111B << 5 -> 1111111111100000B
14 << 1 -> 28
```

>> Logical shift right

Precedence

4

Description

Use `>>` or `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 >> 3 -> 00001110B
1111111111111111B >> 20 -> 0
14 >> 1 -> 7
```

BYTE1 First byte

Precedence

1

Description

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

Example

```
BYTE1 0xABCD -> 0xCD
```

BYTE2 Second byte

Precedence

1

Description

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

Example

```
BYTE2 0x12345678 Õ 0x56
```

BYTE3 Third byte ()

Precedence 1

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

Example `BYTE3 0x12345678 -> 0x34`

BYTE4 Fourth byte

Precedence 1

Description `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 time/date

Precedence 1

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

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

`DATE 1`	Current second (0–59).
`DATE 2`	Current minute (0–59).
`DATE 3`	Current hour (0–23).
`DATE 4`	Current day (1–31).
`DATE 5`	Current month (1–12).
`DATE 6`	Current year `MOD 100` (1998 Õ98, 2000 Õ00, 2002 Õ02).

Example To assemble the date of assembly:

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

HIGH High byte

Precedence 1

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

Example `HIGH 0xABCD -> 0xAB`

HWRD High word ()

Precedence 1

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

Example `HWRD 0x12345678 -> 0x1234`

LOW Low byte

Precedence 1

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

Example `LOW 0xABCD -> 0xCD`

LWRD Low word

Precedence 1

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

Example `LWRD 0x12345678 -> 0x5678`

SFB segment begin

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

Precedence 1

Parameters

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

offset An optional offset from the start address. The parentheses are optional if *offset* is omitted.

Description `SFB` accepts a single operand to its right. The operator evaluates to the absolute address of the first byte of that segment. This evaluation occurs at linking time.

Example

```
            name    segmentBegin
            rseg    MYCODE:CODE  ; Forward declaration of MYCODE.
            rseg    SEGTAB:CONST
start       dc16    sfb(MYCODE)
            end
```

Even if this code is linked with many other modules, `start` is still set to the address of the first byte of the segment.

SFE segment end ()

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

Precedence 1

Parameters

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

offset An optional offset from the start address. The parentheses are optional if `offset` is omitted.

Description `SFE` accepts a single operand to its right. The operator evaluates to the address of the first byte after the segment end. This evaluation occurs at linking time.

Example

```
               name    segmentEnd
               rseg    MYCODE:CODE  ; Forward declaration of MYCODE.
               rseg    SEGTAB:CONST
end            dc16    sfe(MYCODE)
               end
```

Even if this code is linked with many other modules, `end` is still set to the first byte after the segment `MYCODE`.

The size of the segment `MYCODE` can be achieved by using the `SIZEOF` operator or calculated as:

```
SFE(MYCODE)-SFB(MYCODE)
```

SIZEOF segment size ()

Syntax

`SIZEOF` *`segment`*

Precedence

1

Parameters

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

Description

`SIZEOF` generates `SFE-SFB` for its argument. That is, it calculates the size in bytes of a segment. This is done when modules are linked together.

Example

This code sets `size` to the size of the segment `MYCODE`:

```
               module  table
               rseg    MYCODE:CODE  ; Forward declaration of MYCODE.
               rseg    SEGTAB:CONST
size           dc32    sizeof(MYCODE)
               endmod

               module  application
               rseg    MYCODE:CODE
               nop                  ; Placeholder for application.
               end
```

UGT Unsigned greater than

Precedence: 7

Description: `UGT` evaluates to 1 (true) if the left operand has a larger value than the right operand, otherwise it is 0 (false). The operation treats the operands as unsigned values.

Example:

```
2 UGT 1 -> 1
-1 UGT 1 -> 1
```

ULT Unsigned less than

Precedence: 7

Description: `ULT` evaluates to 1 (true) if the left operand has a smaller value than the right operand, otherwise it is 0 (false). The operation treats the operands as unsigned values.

Example:

```
1 ULT 2 -> 1
-1 ULT 2 -> 0
```

XOR Logical exclusive OR

Precedence: 6

Description: `XOR` evaluates to 1 (true) if either the left operand or the right operand is non-zero, but to 0 (false) if both operands are zero or both are non-zero. Use `XOR` to perform logical XOR on its two operands.

Example:

```
0101B XOR 1010B -> 0
0101B XOR 0000B -> 1
```

Assembler directives

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

Summary of assembler directives

The assembler directives are classified into these groups according to their function:

- *Module control directives*, page 76
- *Symbol control directives*, page 79
- *segment control directives*, page 82
- *Value assignment directives*, page 88
- *Conditional assembly directives*, page 91
- *Macro processing directives*, page 93
- *Listing control directives*, page 101
- *C-style preprocessor directives*, page 106
- *Data definition or allocation directives*, page 111
- *Assembler control directives*, page 114
- *Function directives*, page 117
- *Call frame information directives for names blocks*, page 117.
- *Call frame information directives for common blocks*, page 119
- *Call frame information directives for data blocks*, page 120
- *Call frame information directives for tracking resources and CFAs*, page 121
- *Call frame information directives for stack usage analysis*, page 124

This table gives a summary of all the assembler directives:

Directive	Description	Section
`_args`	Is set to number of arguments passed to macro.	Macro processing
`$`	Includes a file.	Assembler control
`#define`	Assigns a value to a label.	C-style preprocessor
`#elif`	Introduces a new condition in an `#if...#endif` block.	C-style preprocessor
`#else`	Assembles instructions if a condition is false.	C-style preprocessor

Table 13: Assembler directives summary

Directive	Description	Section
#endif	Ends an #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
#line	Changes the line numbers.	C-style preprocessor
#message	Generates a message on standard output.	C-style preprocessor
#pragma	Recognized but ignored.	C-style preprocessor
#undef	Undefines a label.	C-style preprocessor
/*comment*/	C-style comment delimiter.	Assembler control
//	C++ style comment delimiter.	Assembler control
=	Assigns a permanent value local to a module.	Value assignment
ALIAS	Assigns a permanent value local to a module.	Value assignment
ALIGN	Aligns the 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
BLOCK	Specifies the block number for an alias created by the SYMBOL directive.	Symbol control
CASEOFF	Disables case sensitivity.	Assembler control
CASEON	Enables case sensitivity.	Assembler control
CFI	Specifies call frame information.	Call frame information
COL	Sets the number of columns per page. Retained for backward compatibility reasons; recognized but ignored.	Listing control
COMMON	Begins a common segment.	Segment control
DB	Generates 8-bit constants, including strings.	Data definition or allocation

Table 13: Assembler directives summary (Continued)

Directive	Description	Section
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 32-bit floating-point constants.	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
.double	Generates 32-bit values in Texas Instruments' floating-point format.	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
DS64	Allocates space for 64-bit integers.	Data definition or allocation
DW	Generates 16-bit constants.	Data definition or allocation

Table 13: Assembler directives summary (Continued)

Directive	Description	Section
ELSE	Assembles instructions if a condition is false.	Conditional assembly
ELSEIF	Specifies a new condition in an IF…ENDIF block.	Conditional assembly
END	Ends 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	Ends the assembly of the current module.	Module control
ENDR	Ends a repeat structure.	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment
EVEN	Aligns the program counter to an even address.	Segment control
EXITM	Exits prematurely from a macro.	Macro processing
EXTERN	Imports an external symbol.	Symbol control
.float	Generates 48-bit values in Texas Instruments' floating-point format.	Data definition or allocation
FUNCTION	Declares a label name to be a function.	Function
IF	Assembles instructions if a condition is true.	Conditional assembly
IMPORT	Imports an external symbol.	Symbol control
LIBRARY	Begins a library module.	Module control
LIMIT	Checks a value against limits.	Value assignment
LOCAL	Creates symbols local to a macro.	Macro processing
LSTCND	Controls conditional assembler listing.	Listing control
LSTCOD	Controls multi-line code listing.	Listing control
LSTEXP	Controls the listing of macro generated lines.	Listing control
LSTMAC	Controls the listing of macro definitions.	Listing control
LSTOUT	Controls assembler-listing output.	Listing control
LSTPAG	Retained for backward compatibility reasons. Recognized but ignored.	Listing control
LSTREP	Controls the listing of lines generated by repeat directives.	Listing control
LSTXRF	Generates a cross-reference table.	Listing control

Table 13: Assembler directives summary (Continued)

Directive	Description	Section
MACRO	Defines a macro.	Macro processing
MODULE	Begins a library module.	Module control
MULTWEAK	Exports symbols to other modules; multiple definitions allowed.	Symbol control
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
OVERLAY	Recognized but ignored.	Symbol control
PAGE	Retained for backward compatibility reasons.	Listing control
PAGSIZ	Retained for backward compatibility reasons.	Listing control
PROGRAM	Begins a program module.	Module control
PUBLIC	Exports symbols to other modules.	Symbol control
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.	Symbol control
RADIX	Sets the default base.	Assembler control
REPT	Assembles instructions a specified number of times.	Macro processing
REPTC	Repeats and substitutes characters.	Macro processing
REPTI	Repeats and substitutes strings.	Macro processing
REQUIRE	Forces a symbol to be referenced.	Symbol control
RSEG	Begins a relocatable segment.	Segment control
RTMODEL	Declares runtime model attributes.	Module control
SET	Assigns a temporary value.	Value assignment
SFRB	Creates byte-access SFR labels.	Value assignment
SFRL	Creates 4-byte-access SFR labels.	Value assignment
SFRTYPE	Specifies SFR attributes.	Value assignment
SFRW	Creates word-access SFR labels.	Value assignment
STACK	Begins a stack segment.	Segment control
SYMBOL	Creates an alias that can be used for referring to a C/C++ symbol.	Symbol control
VAR	Assigns a temporary value.	Value assignment

Table 13: Assembler directives summary (Continued)

Description of assembler directives

The following pages give reference information about the assembler directives.

Module control directives

Syntax

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

Parameters

address	An expression (label plus offset) that ca be resolved at assembly time. It is output in the object code as a program entry address.
expr	An optional expression used by the assembler to encode the runtime options. It must be within the range 0-255 and evaluate to a constant value. The expression is only meaningful if you are assembling source code that originates as assembler output from the compiler.
key	A text string specifying the key.
symbol	Name assigned to module, used by XLINK, XAR, and XLIB when processing object files.
value	A text string specifying the value.

Description

Module control directives are used for marking the beginning and end of source program modules, and for assigning names and types to them. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 24.

Directive	Description	Expression restrictions
`END`	Ends the assembly of the last module in a file.	Locally defined symbols plus offset or integer constants

Table 14: Module control directives

Directive	Description	Expression restrictions
ENDMOD	Ends the assembly of the current module.	Locally defined symbols plus offset or integer constants
LIBRARY	Begins a library module.	No external references Absolute
MODULE	Begins a library module.	No external references Absolute
NAME	Begins a program module.	Absolute
PROGRAM	Begins a program module.	No external references Absolute
RTMODEL	Declares runtime model attributes.	Not applicable

Table 14: Module control directives (Continued)

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, the IAR XAR Library Builder, 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 several 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.

Beginning a module

Use any of the directives NAME or PROGRAM to begin an ELF module, and to assign a name.

A module is included in the linked application, even if other modules do not reference them. For more information about how modules are included in the linked application, read about the linking process in the *IAR C/C++ Compiler Reference Guide for MSP430*.

Note: There can be only one module in a file.

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 ends the last module in the file, if this is not done explicitly with an ENDMOD directive.

Defining a program entry

Program entries must be either relocatable or absolute and cannot be external. The defined program entry for the application will show up in the XLINK map file, and in some of the XLINK output formats.

Assembling multi-module files

These 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 the next module (beginning with MODULE, LIBRARY, NAME, or PROGRAM).

If any of the directives NAME, MODULE, LIBRARY, or PROGRAM is missing, the module is assigned the name of the source file and the attribute program.

Declaring runtime model attributes

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

A module can have several runtime model definitions.

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

If you are writing assembler routines for use with C or C++ code, and you want to control the module consistency, refer to the *IAR C/C++ Compiler Reference Guide for MSP430.*

The following example defines three modules where:

- `MOD_1` and `MOD_2` cannot be linked together since they have different values for runtime model `CAN`.
- `MOD_1` and `MOD_3` can be linked together since they have the same definition of runtime model `RTOS` and no conflict in the definition of `CAN`.
- `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 "CAN",      "ISO11519"
rtmodel "Platform", "M7"
; ...
endmod

module  mod_2
rtmodel "CAN",      "ISO11898"
rtmodel "Platform", "*"
; ...
endmod

module  mod_3
rtmodel "Platform", "M7"
; ...
end
```

Symbol control directives

Syntax

`label BLOCK` *`old_label, block_number`*

`EXTERN` *`symbol`* `[,`*`symbol`*`]` …

`MULTWEAK` *`symbol`* `[,`*`symbol`*`]` …

`IMPORT` *`symbol`* `[,`*`symbol`*`]` …

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

`PUBWEAK` *`symbol`* `[,`*`symbol`*`]` …

`REQUIRE` *`symbol`*

`label` `SYMBOL "`*`C/C++_symbol`*`" [,`*`old_label`*`]`

Parameters

`block_number`	Block number of the alias created by the `SYMBOL` directive.

C/C++_symbol	C/C++ symbol to create an alias for.
label	Label to be used as an alias for a C/C++ symbol.
old_label	Alias created earlier by a SYMBOL directive.
symbol	Symbol to be imported or exported.

Description

These directives control how symbols are shared between modules:

Directive	Description
BLOCK	Specifies the block number for an alias created by the SYMBOL directive.
EXTERN, IMPORT	Imports an external symbol.
MULTWEAK	Exports symbols to other modules; multiple definitions allowed.
OVERLAY	Recognized but ignored.
PUBLIC	Exports symbols to other modules.
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.
REQUIRE	Forces a symbol to be referenced.
SYMBOL	Creates an alias for a C/C++ symbol.

Table 15: Symbol control directives

Exporting symbols to other modules

Use PUBLIC to make one or more symbols available to other modules. Symbols defined PUBLIC can be relocatable or absolute, and can also be used in expressions (with the same rules as for other symbols).

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

There can be any number of PUBLIC-defined symbols in a module.

Exporting symbols with multiple definitions to other modules

PUBWEAK is similar to PUBLIC except that it allows the same symbol to be defined in more than one module. Only one of those definitions is 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 uses the PUBLIC definition.

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

Note: Library modules are only linked if a reference to a symbol in that module is made, and that symbol was not already 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 or IMPORT to import an untyped external symbol.

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

Referring to scoped C/C++ symbols

Use the SYMBOL directive to create an alias for a C/C++ symbol. You can use the alias to refer to the C/C++ symbol. The symbol and the alias must be located within the same scope.

Use the BLOCK directive to provide the block scope for the alias.

Typically, the SYMBOL and the BLOCK directives are for compiler internal use only, for example, when referring to objects inside classes or namespaces. For detailed information about how to use these directives, declare and define your C/C++ symbol, compile, and view the assembler listfile output.

Example

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.

Because 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 is resolved at link time.

```
            name    errorMessage
            extern  print
            public  err
            rseg    CODE:CODE

err         call    print
            dc8     "** Error **"
            ret

            end
```

Mode control directives

Syntax

```
CODE
DATA
DATA8
DATA16
DATA24
DATA32
DATA64
```

Description

These directives provide control over the assembly mode:

Directive	Description
CODE	Subsequent instructions are assembled, linked, and disassembled as code.
DATA, DATA8	Subsequent instructions are assembled, linked, and disassembled as 8-bit data.
DATA16	Subsequent instructions are assembled, linked, and disassembled as 16-bit data.
DATA24	Subsequent instructions are assembled, linked, and disassembled as 24-bit data.
DATA32	Subsequent instructions are assembled, linked, and disassembled as 32-bit data.
DATA64	Subsequent instructions are assembled, linked, and disassembled as 64-bit data.

Table 16: Mode control directives

The CODE and DATA directives set the assembly mode for code and data sections. This information is used by C-SPY.

Note: The CODE or DATA directives are required for big-endian applications, but they improve the disassembly for all applications.

You can use the CODE or DATA directives to:

- Start a code/data segment part (RSEG) that generates bytes that end up in the image, either code or data
- Change the assembly mode in the middle of a segment part. You do not need the CODE or DATA directives for declaring segments, extern labels etc, nor when you declare RAM space.

segment control directives

Syntax

ALIGN *align* [,*value*]

ALIGNRAM *align*

```
ASEG [start]

ASEGN segment [:type] [:flag] [,address]

COMMON segment [:type] [:flag] [(align)]

EVEN [value]

ODD [value]

ORG expr

RSEG segment [:type] [:flag] [(align)]

STACK segment [:type] [:flag] [(align)]
```

Parameters

address	Address where this segment part is placed.
align	The power of two to which the address should be aligned. The permitted range is 0 to 8. The default align value is 0, except for code segments where the default is 1.
expr	Address to set the location counter to.

flag	`ROOT`, `NOROOT`
	
	`ROOT` (the default mode) indicates that the segment part must not be discarded.
	
	`NOROOT` means that the segment part is discarded by the linker if no symbols in this segment part are referred to. Normally, all segment parts except startup code and interrupt vectors should set this flag.
	
	`REORDER`, `NOREORDER`
	
	`NOREORDER` (the default mode) indicates that the segment parts must remain in order.
	
	`REORDER` allows the linker to reorder segment parts. For a given segment, all segment parts must specify the same state for this flag.
	
	`SORT`, `NOSORT`
	
	`NOSORT` (the default mode) indicates that the segment parts are not sorted.
	
	`SORT` means that the linker sorts the segment parts in decreasing alignment order. For a given segment, all segment parts must specify the same state for this flag.

segment	The name of the segment. The segment name is a user-defined symbol that follows the rules described in *Symbols*, page 20.
start	A start address that has the same effect as using an ORG directive at the beginning of the absolute segment.
type	The memory type, typically CODE or DATA. In addition, any of the types supported by the IAR XLINK Linker.
value	Byte value used for padding, default is zero.

Description

The segment directives control how code and data are located. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 24.

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 program location counter (PLC).	No external references Absolute (see below)
RSEG	Begins a relocatable segment.	No external references Absolute
STACK	Begins a stack segment.	

Table 17: Segment control directives

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.

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

```
            module  resetVector
            extern  main

            aseg
            org     0xfffe          ; Start the segment at the
                                    ; reset vector address.
reset       dc16    main            ; Point the reset vector to
                                    ; the externally defined main
                                    ; label.
            end
```

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 start a new segment. The assembler maintains separate location counters (initially set to zero) for all segments, which makes it possible to switch segments and mode anytime without having to save the current program location counter.

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

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`:

```
            module  calculate
            extern  operator
            extern  addOperator, subOperator

            rseg    TABLE:CONST(8)
operatorTable:
            dc8     addOperator, subOperator
```

```
              rseg    CODE:CODE
calculate     lda     operator
              ldhx    #operatorTable
              cbeq    ,X+,add
              cbeq    ,X+,sub
              ;...
              rts

add           ;...
              rts
sub           ;...
              rts

              end
```

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 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 several different routines to share a reusable, common area of memory for data.

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

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

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

This example defines two common segments containing variables:

```
              name    common1
              common  MYDATA
count         dc32    1
              endmod

              name    common2
              common  MYDATA
up            ds8     1
              ds8     2
down          ds8     1
              end
```

Because the common segments have the same name, MYDATA, the variables up and count refer to the same location in memory.

Setting the program location counter (PLC)

Use ORG to set the program location counter of the current segment to the value of an expression. 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 can be either absolute or relative (and the value is interpreted as an offset relative to the segment start in both cases).

The program location counter is 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. You do this by specifying an expression for the power of two to which the program counter should be aligned. That is, a value of 1 aligns to an even address and a value of 2 aligns to an address evenly divisibly by 4.

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 value used for padding bytes must be within the range 0 to 255.

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

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.

```
            name    alignment
            rseg    DATA        ; Start a relocatable data segment.
            even                ; Ensure it is on an even boundary.
target      dc16    1           ; target and best will be on an
best        dc16    1           ; even boundary.
            align   6           ; Now, align to a 64-byte boundary,
results     ds8     64          ; and create a 64-byte table.
            end
```

Value assignment directives

Syntax

```
label = expr
label ALIAS expr
label ASSIGN expr
label DEFINE const_expr
label EQU expr
LIMIT expr, min, max, message
[const] SFRB register = value
[const] SFRL register = value
[const] SFRTYPE register attribute [,attribute] = value
[const] SFRW register = value
label SET expr
label VAR expr
```

Parameters

attribute	One or more of these:
	`BYTE`: The SFR must be accessed as a byte.
	`READ`: You can read from this SFR.
	`WORD`: The SFR must be accessed as a word.
	`WRITE`: You can write to this SFR.
const_expr	Constant value assigned to symbol.
expr	Value assigned to symbol or value to be tested.
label	Symbol to be defined.
message	A text message that is printed when *expr* is out of range.
min, max	The minimum and maximum values allowed for *expr*.
register	The special function register.
value	The SFR port address.

Description

These directives are used for assigning values to symbols:

Directive	Description
=, EQU	Assigns a permanent value local to a module.
ALIAS	Assigns a permanent value local to a module.
ASSIGN, SET, VAR	Assigns a temporary value.
DEFINE	Defines a file-wide value.
LIMIT	Checks a value against limits.
SFRB	Creates byte-access SFR labels.
SFRL	Creates 4-byte-access SFR labels.
SFRTYPE	Specifies SFR attributes.
SFRW	Creates word-access SFR labels.

Table 18: Value assignment directives

Defining a temporary value

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

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

```
            name    table
cons        set     1

; Generate table of powers of 3.
cr_tabl     macro   times
            dc32    cons
cons        set     cons * 3
            if      times > 1
            cr_tabl times - 1
            endif
            endm

            rseg    CODE:CODE
table       cr_tabl 4
            end
```

Defining a permanent local value

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 (but not with a PUBWEAK 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 was 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. Also, the expression assigned to the defined symbol must be constant.

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
gVal        define  0x20           ; Definition of a permanent
                                   ; global value.
lVal        equ     12             ; Definition of a local value.

            rseg    CODE:CODE
add12       mov     #gVal, r8
            addc     #lVal, r8
            ret
            endmod

            name    add2
            public  add20
lVal        equ     20             ; Redefinition of local value.

            rseg    CODE:CODE
add20       mov     #gVal, r8
            addc     #lVal, r8
            ret
            end
```

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

Defining special function registers

Use SFRB to create special function register labels with the attributes READ, WRITE, and BYTE turned on. Use SFRW to create special function register labels with the attributes

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

Prefix the directive with const to disable the WRITE attribute assigned to the SFR. You will then get an error or warning message when trying to write to the SFR. The const keyword must be placed on the same line as the directive.

In this example several SFR variables are declared with a variety of access capabilities:

```
            name    sfrs
            rseg    CODE:CODE

            sfrb    portd = 0x12    ; Byte read/write access.
            sfrw    ocr1 = 0x2A     ; Word read/write access.
const       sfrb    pind = 0x10     ; Byte read only access.
            sfrtype portb write, byte = 0x18 ; Byte write only
                                             ; access.
            end
```

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 appears.

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

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.

```
            module    setLimit
speed       set       23
            limit     speed,10,30,"Speed is out of range!"
            end
```

Conditional assembly directives

Syntax

ELSE

ELSEIF *condition*

ENDIF

IF *condition*

Parameters

condition One of these:

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

Description

Use the `IF`, `ELSE`, and `ENDIF` directives to control the assembly process at assembly time. If the condition following the `IF` directive is not true, the subsequent instructions do not generate any code (that is, it is not 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 can 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 can 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 can be nested to any level.

Example

This example uses a macro to add a constant to a direct page memory location:

```
; If the second argument to the addMem macro is 1, 2, or 3,
; it generates the equivalent number of INC instructions. For any
; other non-zero value of the second argument, it generates a
; mov.w instruction.

addMem      macro   loc,val         ; loc is a direct page memory
                                    ; location, and val is an
                                    ; 8-bit value to add to that
                                    ; location.
            if      val = 0
                                    ; Do nothing.
            elseif  val = 1
            inc     loc
            elseif  val = 2
            inc     loc
            inc     loc
            elseif  val = 3
            inc     loc
            inc     loc
            inc     loc
            else
            add     #val, loc
            endif
            endm


            module  addWithMacro
            rseg    CODE:CODE

addSome     addMem  0xa0,0          ; Add 0 to memory loc. 0xa0.
            addMem  0xa0,1          ; Add 1 to the same address.
            addMem  0xa0,2          ; Add 2 to the same address.
            addMem  0xa0,3          ; Add 3 to the same address.
            addMem  0xa0,47         ; Add 47 to the same address.
            ret
            end
```

Macro processing directives

Syntax

_args

ENDM

ENDR

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

Parameters

actual	Strings to be substituted.
argument	Symbolic argument names.
expr	An expression.
formal	An argument into which each character of *actual* (REPTC) or each string of *actual* (REPTI) is substituted.
name	The name of the macro.
symbol	Symbols to be local to the macro.

Description

These directives allow user macros to be defined. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 24.

Directive	Description	Expression restrictions
_args	Is set to number of arguments passed to macro.	
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 19: Macro processing directives

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.

The macro process consists of three distinct phases:

1 The assembler scans and saves macro definitions. The text between `MACRO` and `ENDM` is saved but not syntax checked. Include-file references `$`*`file`* are recorded and included during macro expansion.

2 A macro call forces the assembler to invoke the macro processor (expander). The macro expander switches (if not already in a macro) the assembler input stream from a source file to the output from the macro expander. The macro expander takes its input from the requested macro definition.

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

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

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:

```
            name    errMacro
errMac      macro   text
            extern  abort
            call    abort
            dc8     text,0
            endm
            end
```

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

```
errMac  'Disk not ready'
```

The assembler expands this to:

```
call   abort
dc8    'Disk not ready',0
even
```

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:

```
            name    errMacro
errMac      macro   text
            extern  abort
            call    abort
            dc8     \1,0
            endm
            end
```

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:

```
            name    ldaMacro
ldaMac      macro   op
            add     op
            endm
            end
```

The macro can be called using the macro quote characters:

```
ldaMac  <R4,R5>
```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 46.

Predefined macro symbols

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

```
fill        macro
            if      _args == 2
            rept    \2
            dc8     \1
            endr
            else
            dc8     \1
            endif
            endm

            module  filler
            rseg    CODE:CODE
            fill    3
            fill    4, 3
            end
```

It generates this code:

```
10    000000
11    000000                          module  filler
12    000000                          rseg    CODE:CODE
13    000000                          fill    3
13.1  000000                          if      _args == 2
13.2  000000                          rept
13.3  000000                          dc8     3
13.4  000000                          endr
13.5  000000                          else
13.6  000000 03                       dc8     3
13.7  000001                          endif
13.8  000001                          endm
14    000001                          fill    4, 3
14.1  000001                          if      _args == 2
14.2  000001                          rept    3
14.3  000001                          dc8     4
14.4  000001                          endr
14.5  000001 04                       dc8     4
14.6  000004                          else
14.7  000004                          dc8     4
14.8  000004                          endif
14.9  000004                          endm
15    000004                          end
```

Repeating statements

Use the `REPT...ENDR` structure to assemble the same block of instructions several times. If *`expr`* evaluates to 0 nothing is 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.

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

```
            name    reptc
            extern  plotc
            rseg    CODE:CODE

banner      reptc   chr, "Welcome"
            mov     'chr', r8
            call    plotc
            endr
            end
```

This produces this code:

```
1     000000                NAME    reptc
2     000000                extern  plotc
3     000000                rseg    CODE:CODE
4     000000
5     000000                banner  reptc  chr, 'Welcome'
6     000000                        mov    'chr', r8
7     000000                        call   plotc
8     000000                        endr
8.1   000000 18405500               mov    'W', r8
8.2   000004 9012....               call   plotc
8.3   000008 18405B00               mov    'e', r8
8.4   00000C 9012....               call   plotc
8.5   000010 18405A00               mov    'l', r8
8.6   000014 9012....               call   plotc
8.7   000018 18404900               mov    'c', r8
8.8   00001C 9012...                call   plotc
8.9   000020 18484D00               mov    'o', r8
8.10  000024 9012....               call   plotc
8.11  000028 18404300               mov    'm', r8
8.12  00002C 9012....               call   plotc
8.13  000030 18403300               mov    'e', r8
8.14  000034 9012....               call   plotc
9     000038                        end
```

This example uses `REPTI` to clear several memory locations:

```
            name    repti
            extern  base, count, init
            rseg    CODE:CODE

banner      repti   adds, base, count, init
            clr     adds
            endr

            end
```

This produces this code:

```
1     000000              name    repti
2     000000              extern  base, count, init
3     000000              rseg    CODE:CODE
4     000000
5     000000    banner    repti   adds, base, count, init
6     000000              clr     adds
7     000000              endr
7.1   000000 8043....     clr     base
7.2   000004 8043....     clr     count
7.3   000008 8043....     clr     init
8     00000C
9     00000C              end
```

Coding inline for efficiency

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

This example outputs bytes from a buffer to a port:

```
            extern  port
            rseg    RAM
buffer      db      25
            rseg    PROM
;Plays 256  bytes from buffer to port
play        mov     #buffer, r4
            mov     #256, r5
loop        mov     @r4+,&port
            inc     r4
            dec     r5
            jne     loop
            ret
            end
```

For efficiency we can recode this using a macro:

```
            play    macro
            local   loop
            mov     #buffer,r4
            mov     #64,r5
loop        mov     @r4+,&port
            mov     @r4+,&port
            mov     @r4+,&port
            mov     @r4+,&port
            dec     dec r5
            jne     loop
            endm
```

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

Listing control directives

Syntax

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

Parameters

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

Description

These directives provide control over the assembler list file:

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

Table 20: Listing control directives

Directive	Description
LSTXRF	Generates a cross-reference table.
PAGE	Generates a new page.
PAGSIZ	Sets the number of lines per page.

Table 20: Listing control directives (Continued)

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).

To disable the listing of a debugged section of program:

```
lstout-
; This section has already been debugged.
lstout+
; This section is currently being debugged.
end
```

Listing conditional code and strings

Use LSTCND+ to force the assembler to list source code only for the parts of the assembly that are not disabled by previous conditional IF statements.

The default setting is LSTCND-, which lists all source lines.

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

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

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

```
            name    lstcndTest
            extern  print
            rseg    FLASH:CODE

debug       set     0
begin       if      debug
            call    print
            endif

            lstcnd+
begin2      if      debug
            call    print
            endif

            end
```

This generates the following listing:

```
1     000000                  name    lstcndTest
2     000000                  extern  print
3     000000                  rseg    FLASH:CODE
4     000000
5     000000        debug     set     0
6     000000        begin     if      debug
7     000000                  call    print
8     000000                  endif
9     000000
10    000000                  lstcnd+
11    000000        begin2    if      debug
13    000000                  endif
14    000000
15    000000                  end
```

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.

This example shows the effect of `LSTMAC` and `LSTEXP`:

```
            name    lstmacTest
            extern  memLoc
            rseg    FLASH:CODE

dec2        macro   arg
            dec     arg
            dec     arg
            endm

            lstmac+
inc2        macro   arg
            inc     arg
            inc     arg
            endm

begin       dec2    memLoc
            lstexp-
            inc2    memLoc
            ret

; Restore default values for
; listing control directives.

            lstmac-
            lstexp+

            end     begin
```

This produces the following output:

```
  9     000000                          name    lstmacTest
 10     000000                          extern  memLoc
 11     000000                          rseg    FLASH:CODE
 12     000000
 17     000000
 18     000000                          lstmac+
 19     000000                inc2      macro   arg
 20     000000                          inc     arg
 21     000000                          inc     arg
 22     000000                          endm
 23     000000
 24     000000                begin     dec2    memLoc
 24.1   000000 9083...                  dec     memLoc
 24.2   000004 9083...                  dec     memLoc
 24.3   000008                          endm
 25     000008                          lstexp-
 26     000008                          inc2    memLoc
 27     000010 3041                     ret
 28     000012
 29     000012                  ; Restore default values for
 30     000012                  ; listing control directives.
 31     000012
 32     000012                          lstmac-
 33     000012                          lstexp+
 34     000012
 35     000012                          end     begin
```

Controlling the listing of generated lines

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

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

Generating a cross-reference table

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

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

Specifying the list file format

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

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

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

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

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

C-style preprocessor directives

Syntax

```
#define symbol text

#elif condition

#else

#endif

#error "message"

#if condition

#ifdef symbol

#ifndef symbol

#include {"filename" | <filename>}

#line line-no {"filename"}

#message "message"

#undef symbol
```

Parameters

condition	An absolute expression The expression must not contain any assembler labels or symbols, and any non-zero value is considered as true. The C preprocessor operator `defined` can be used.
filename	Name of file to be included or referred.
line-no	Source line number.
message	Text to be displayed.
symbol	Preprocessor symbol to be defined, undefined, or tested.
text	Value to be assigned.

Description

The assembler has a C-style preprocessor that is similar to the C89 standard.

These C-language preprocessor directives are available:

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

Table 21: C-style preprocessor directives

You must not mix assembler language and C-style preprocessor directives. Conceptually, they are different languages and mixing them might lead to unexpected behavior because an assembler directive is not necessarily accepted as a part of the C preprocessor language.

Note that the preprocessor directives are processed before other directives. As an example avoid constructs like:

```
redef       macro                   ; Avoid the following!
#define \1 \2
            endm
```

because the `\1` and `\2` macro arguments are not available during the preprocessing phase.

Defining and undefining preprocessor symbols

Use `#define` to define a value of a preprocessor symbol.

`#define` *`symbol value`*

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

Conditional preprocessor 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 (that is, it will not be assembled or syntax checked) until an #endif or #else directive is found.

All assembler directives (except for END) and file inclusion can 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 can 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.

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

```
            module  calibrate
            extern  calibrationConstant
            rseg    CODE:CODE

#define     tweak  1
#define     adjust 3

calibrate   mov     calibrationConstant, r8
#ifdef      tweak
#if         adjust==1
            sub     #4, r8
#elif       adjust==2
            sub     #20, r8
#elif       adjust==3
            sub     #30, r8
#endif
#endif      /* ifdef tweak */
            mov     r8, calibrationConstant
            ret

            end
```

Including source files

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

`#include "filename"` and `#include <filename>` search these directories in the specified order:

1 The source file directory. (This step is only valid for `#include "filename"`.)

2 The directories specified by the `-I` option, or options. The directories are searched in the same order as specified on the command line, followed by the ones specified by environment variables.

3 The current directory, which is the same as where the assembler executable file is located.

4 The automatically set up library system include directories. See *-g*, page 43.

This example uses `#include` to include a file defining macros into the source file. For example, these macros could be defined in `Macros.inc`:

```
; Exchange registers a and b.
; Use the stack for temporary storage.

xch         macro   a,b
            push    a
            mov     a,b
            pop     b
            endm
```

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

```
            program includeFile
            rseg    CODE:CODE

; Standard macro definitions.
#include "Macros.inc"

xchRegs     xch     r8, r9
            ret

            end
```

Displaying errors

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

Ignoring #pragma

A `#pragma` line is ignored by the assembler, making it easier to have header files common to C and assembler.

Changing the source line numbers

Use the `#line` directive to change the source line numbers and the source filename used in the debug information. `#line` operates on the lines following the `#line` directive.

Comments in C-style preprocessor directives

If you make a comment within a define statement, use:

- the C comment delimiters `/* ... */` to comment sections
- the C++ comment delimiter `//` to mark the rest of the line as comment.

Do not use assembler comments within a define statement as it leads to unexpected behavior.

This expression evaluates to 3 because the comment character is preserved by `#define`:

```
#define x 3    ; This is a misplaced comment.

            module  misplacedComment1
expression  equ     x * 8 + 5
            ;...
            end
```

This example illustrates some problems that might occur when assembler comments are used in the C-style preprocessor:

```
#define five  5     ; This comment is not OK.
#define six   6     // This comment is OK.
#define seven 7     /* This comment is OK. */

            module  misplacedComment2
            rseg    CONST:CONST(2)

            DC32    five, 11, 12
; The previous line expands to:
;          "DC32    5     ; This comment is not OK., 11, 12"

            DC32    six + seven, 11, 12
; The previous line expands to:
;          "DC32    6 + 7, 11, 12"

            end
```

Data definition or allocation directives

Syntax

```
DB expr [,expr] ...
DC8 expr [,expr] ...
DC16 expr [,expr] ...
DC24 expr [,expr] ...
DC32 expr [,expr] ...
DC64 expr [,expr] ...
DF value [,value] ...
DF32 value [,value] ...
DF64 value [,value] ...
DL expr [,expr] ...
.double value [,value] ...
DS count
DS8 count
DS16 count
DS24 count
DS32 count
DS64 count
.float value [,value] ...
```

Parameters

count	A valid absolute expression specifying the number of elements to be reserved.
expr	A valid absolute, relocatable, or external expression, or an ASCII string. ASCII strings are zero filled to a multiple of the data size implied by the directive. Double-quoted strings are zero-terminated.
value	A valid absolute expression or floating-point constant.

Description

These directives define values or reserve memory.

Use `DC8`, `DC16`, `DC24`, `DC32`, `DC64`, `DF32`, or `DF64` to create a constant, which means an area of bytes is reserved big enough for the constant.

Use `DS`, `DS8`, `DS16`, `DS24`, `DS32`, or `DS64` to reserve a number of uninitialized bytes.

For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 24.

The column *Alias* in the following table shows the Texas Instruments directive that corresponds to the IAR Systems directive.

Directive	Alias	Description
`DC8`	`DB`	Generates 8-bit constants, including strings.

Table 22: Data definition or allocation directives

Directive	Alias	Description
DC16	DW	Generates 16-bit constants.
DC24		Generates 24-bit constants.
DC32		Generates 32-bit constants.
DC64		Generates 64-bit constants
DF32	DF	Generates 32-bit floating-point constants.
DF64		Generates 64-bit floating-point constants.
.double		Generates 32-bit values in Texas Instruments' floating-point format.
DS8	DS	Allocates space for 8-bit integers.
DS16	DS 2	Allocates space for 16-bit integers.
DS24		Allocates space for 24-bit integers.
DS32	DS 4	Allocates space for 32-bit integers.
DS64	DS 8	Allocates space for 64-bit integers.
.float		Generates 48-bit values in Texas Instruments' floating-point format.

Table 22: Data definition or allocation directives (Continued)

Generating a lookup table

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

```
            module  sumTableAndIndex
            rseg    DATA16_C:CONST

table       dc8     12
            dc8     15
            dc8     17
            dc8     16
            dc8     14
            dc8     11
            dc8     9

            rseg    CODE:CODE
count       set     0

addTable    mov     #0, r8

            rept    7
            if      count == 7
            exitm
            endif
            addc    table + count. r8
count       set     count + 1
            endr

            ret

            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 `10` bytes:

```
table   DS8   10
```

Assembler control directives

Syntax

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

Parameters

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

Description

These directives provide control over the operation of the assembler. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 24.

Directive	Description	Expression restrictions
`$`	Includes a file.	
`/*comment*/`	C-style comment delimiter.	
`//`	C++ style comment delimiter.	
`CASEOFF`	Disables case sensitivity.	
`CASEON`	Enables case sensitivity.	
`RADIX`	Sets the default base on all numeric values.	No forward references No external references Absolute Fixed

Table 23: Assembler control directives

Use `$` to insert the contents of a file into the source file at a specified point. This is an alias for `#include`, see *C-style preprocessor directives*, page 107.

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

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

Use `RADIX` to set the default base for constants. The default base is 10.

Controlling case sensitivity

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

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

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

```
            module  caseSensitivity1
            rseg    CODE:CODE

            caseoff
label       nop                     ; Stored as "LABEL".
            bra     LABEL
            end
```

The following will generate a duplicate label error:

```
            module  caseSensitivity2
            rseg    CODE:CODE
            caseoff
label       nop                     ; Stored as "LABEL".
LABEL       nop                     ; Error, "LABEL" already defined.
            end
```

Including a source file

This example uses `$` to include a file defining macros into the source file. For example, these macros could be defined in `Macros.inc`:

```
xch         macro   a,b
            push    a
            mov     a,b
            pop     b
            endm
```

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

```
            program includeFile
            rseg    CODE:CODE

; Standard macro definitions.
$Macros.inc

xchRegs     xch     r8,r9
            ret
            end     xchRegs
```

Defining comments

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

```
/*
Program to read serial input.
Version 1: 19.2.11
Author: mjp
*/
```

See also *C-style preprocessor directives*, page 107.

Changing the base

To set the default base to 16:

```
            module  radix
            rseg    CODE:CODE

            radix   16              ; With the default base set
            mov     12, r8          ; to 16, the immediate value
            ;...                    ; of the load instruction is
                                    ; interpreted as 0x12.

; To reset the base from 16 to 10 again, the argument must be
; written in hexadecimal format.

            radix   0x0a            ; Reset the default base to 10.
            mov     12, r8          ; Now, the immediate value of
            ;...                    ; the load instruction is
                                    ; interpreted as 0x0c.
            end
```

Function directives

Syntax

`CALL_GRAPH_ROOT` *`function`* `[,`*`category`*`]`

Parameters

`function`	The function, a symbol.
`category`	An optional call graph root category, a string.

Description

Use this directive to specify that, for stack usage analysis purposes, the function *`function`* is a call graph root. You can also specify an optional category, a quoted string.

The compiler will generate this directive in assembler list files, when needed.

Example

```
CALL_GRAPH_ROOT my_interrupt, "interrupt"
```

See also

Call frame information directives for stack usage analysis, page 124, for information about `CFI` directives required for stack usage analysis.

IAR C/C++ Compiler Reference Guide for MSP430 for information about how to enable and use stack usage analysis.

Call frame information directives for names blocks

Syntax

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 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`*`] …`

Parameters

bits	The size of the resource in bits.
cell	The name of a frame cell.
cfa	The name of a CFA (canonical frame address).
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 segment memory type, such as `CODE`, `CONST` or `DATA`. In addition, any of the memory types supported by the IAR XLINK Linker. It is only used for denoting an address space.

Description

Use these directives to define a names block:

Directive	Description
`CFI BASEADDRESS`	Declares a base address CFA (Canonical Frame Address).
`CFI ENDNAMES`	Ends a names block.
`CFI FRAMECELL`	Creates a reference into the caller's frame.
`CFI NAMES`	Starts a names block.
`CFI RESOURCE`	Declares a resource.
`CFI RESOURCEPARTS`	Declares a composite resource.
`CFI STACKFRAME`	Declares a stack frame CFA.
`CFI VIRTUALRESOURCE`	Declares a virtual resource.

Table 24: Call frame information directives names block

Example

Examples of using CFI directives, page 34

See also

Tracking call frame usage, page 26

Call frame information directives for common blocks

Syntax

Common block directives:

```
CFI COMMON name USING namesblock
CFI ENDCOMMON name
CFI CODEALIGN codealignfactor
CFI DATAALIGN dataalignfactor
CFI RETURNADDRESS resource type
```

Extended common block directives:

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

Parameters

codealignfactor	The smallest common factor of all instruction sizes. Each CFI directive for a data block must be placed according to this alignment. 1 is the default and can always be used, but a larger value reduces the produced call frame information in size. The possible range is 1–256.
commonblock	The name of a previously defined common block.
dataalignfactor	The smallest common factor of all frame sizes. If the stack grows toward higher addresses, the factor is negative; if it grows toward lower addresses, the factor is positive. 1 is the default, but a larger value reduces the produced call frame information in size. The possible ranges are –256 to –1 and 1 to 256.
name	The name of the block.
namesblock	The name of a previously defined names block.
resource	The name of a resource.
type	The memory type, such as `CODE`, `CONST` or `DATA`. In addition, any of the segment memory types supported by the IAR XLINK Linker. It is only used for denoting an address space.

Description

Use these directives to define a common block:

Directive	Description
`CFI CODEALIGN`	Declares code alignment.

Table 25: Call frame information directives common block

Directive	Description
CFI COMMON	Starts or extends a common block.
CFI DATAALIGN	Declares data alignment.
CFI ENDCOMMON	Ends a common block.
CFI RETURNADDRESS	Declares a return address column.

Table 25: Call frame information directives common block (Continued)

In addition to these directives you might also need the call frame information directives for specifying rules or CFI expressions for resources and CFAs, see *Call frame information directives for tracking resources and CFAs*, page 121.

Example

Examples of using CFI directives, page 34

See also

Tracking call frame usage, page 26

Call frame information directives for data blocks

Syntax

```
CFI BLOCK name USING commonblock
CFI ENDBLOCK name
CFI { NOFUNCTION | FUNCTION label }
CFI { INVALID | VALID }
CFI { REMEMBERSTATE | RESTORESTATE }
CFI PICKER
CFI CONDITIONAL label [, label] …
```

Parameters

commonblock	The name of a previously defined common block.
label	A function label.
name	The name of the block.

Description

These directives allow call frame information to be defined in the assembler source code:

Directive	Description
CFI BLOCK	Starts a data block.
CFI CONDITIONAL	Declares a data block to be a conditional thread.

Table 26: Call frame information directives for data blocks

Directive	Description
`CFI ENDBLOCK`	Ends a data block.
`CFI FUNCTION`	Declares a function associated with a data block.
`CFI INVALID`	Starts a range of invalid call frame information.
`CFI NOFUNCTION`	Declares a data block to not be associated with a function.
`CFI PICKER`	Declares a data block to be a picker thread. Used by the compiler for keeping track of execution paths when code is shared within or between functions.
`CFI REMEMBERSTATE`	Remembers the call frame information state.
`CFI RESTORESTATE`	Restores the saved call frame information state.
`CFI VALID`	Ends a range of invalid call frame information.

Table 26: Call frame information directives for data blocks (Continued)

In addition to these directives you might also need the call frame information directives for specifying rules or CFI expressions for resources and CFAs, see *Call frame information directives for tracking resources and CFAs*, page 121.

Example

Examples of using CFI directives, page 34

See also

Tracking call frame usage, page 26

Call frame information directives for tracking resources and CFAs

Syntax

```
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

cfa — The name of a CFA (canonical frame address).

cfiexpr — A CFI expression, which can be one of these:

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

constant	A constant value or an assembler expression that can be evaluated to a constant value.
offset	The offset relative the CFA. An integer with an optional sign.
resource	The name of a resource.

Unary operators

Overall syntax: `OPERATOR(operand)`

CFI operator	Operand	Description
COMPLEMENT	*cfiexpr*	Performs a bitwise NOT on a CFI expression.
LITERAL	*expr*	Get the value of the assembler expression. This can insert the value of a regular assembler expression into a CFI expression.
NOT	*cfiexpr*	Negates a logical CFI expression.
UMINUS	*cfiexpr*	Performs arithmetic negation on a CFI expression.

Table 27: Unary operators in CFI expressions

Binary operators

Overall syntax: `OPERATOR(operand1,operand2)`

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

Table 28: Binary operators in CFI expressions

CFI operator	Operands	Description
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 is preserved when shifting.
RSHIFTL	*cfiexpr*,*cfiexpr*	Logical shift right of the left operand. The number of bits to shift is specified by the right operand. The sign bit will not be preserved when shifting.
SUB	*cfiexpr*,*cfiexpr*	Subtraction
XOR	*cfiexpr*,*cfiexpr*	Bitwise XOR

Table 28: Binary operators in CFI expressions (Continued)

Ternary operators

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

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

Table 29: Ternary operators in CFI expressions

Description

Use these directives to track resources and CFAs in common blocks and data blocks:

Directive	Description
CFI *cfa*	Declares the value of a CFA.
CFI *resource*	Declares the value of a resource.

Table 30: Call frame information directives for tracking resources and CFAs

Example — *Examples of using CFI directives*, page 34

See also — *Tracking call frame usage*, page 26

Call frame information directives for stack usage analysis

Syntax

```
CFI FUNCALL { caller } callee
CFI INDIRECTCALL { caller }
CFI NOCALLS { caller }
CFI TAILCALL { callee }
```

Description

These directives allow call frame information to be defined in the assembler source code:

Directive	Description
`CFI FUNCALL`	Declares function calls for stack usage analysis.
`CFI INDIRECTCALL`	Declares indirect calls for stack usage analysis.
`CFI NOCALLS`	Declares absence of calls for stack usage analysis.
`CFI TAILCALL`	Declares tail calls for stack usage analysis.

Table 31: Call frame information directives for stack usage analysis

See also — *Tracking call frame usage*, page 26

The *IAR C/C++ Compiler Reference Guide for MSP430* for information about stack usage analysis.

Assembler diagnostics

The following pages describe the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.

Message format

All diagnostic messages are displayed on the screen, and printed in the optional list file.

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

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

In addition, you can find all messages specific to the IAR Assembler for MSP430 in the release note `a430_msg.htm`.

Severity levels

The diagnostic messages produced by the IAR Assembler for MSP430 reflect problems or errors that are found in the source code or occur at assembly time.

OPTIONS FOR DIAGNOSTICS

There are two assembler options for diagnostics. You can:

- Disable or enable all warnings, ranges of warnings, or individual warnings, see *-w*, page 53
- Set the number of maximum errors before the compilation stops, see *-E*, page 42.

ASSEMBLY WARNING MESSAGES

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

COMMAND LINE ERROR MESSAGES

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

ASSEMBLY ERROR MESSAGES

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

ASSEMBLY FATAL ERROR MESSAGES

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

ASSEMBLER INTERNAL ERROR MESSAGES

An internal error is a diagnostic message that signals that there was a serious and unexpected failure due to a fault in the assembler.

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

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

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Symbols