IAR Embedded Workbench®

IAR Assembler™ Reference Guide

for the 8051 Microcontroller Architecture

IAR SYSTEMS

A8051-8

COPYRIGHT NOTICE

DISCLAIMER

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

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

TRADEMARKS

EDITION NOTICE

8th edition: April 2011

Part number: A8051-8

This guide applies to version 8.x of IAR Embedded Workbench® for 8051.

Internal reference: ISUD.

Contents

Tables

Preface

Welcome to the IAR Assembler for 8051 Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use the 8051 IAR Assembler 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 using assembler language for your 8051 microcontroller and need to get detailed reference information on how to use the IAR Assembler for 8051. In addition, you should have working knowledge of the following:

- The architecture and instruction set of your 8051 microcontroller. Refer to the documentation from the chip manufacturer for information about your 8051 microcontroller
- 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 for 8051, you should read the *Introduction to the IAR Assembler* chapter in this reference guide.

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

If you are new to using the IAR toolkit, we recommend that you first read the initial chapters of the *IDE Project Management and Building Guide*. They give product overviews, as well as tutorials that can help you get started.

What this guide contains

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

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

Other documentation

User documentation is available as hypertext PDFs and as a context-sensitive online help system in HTML format. You can access the documentation from the Information Center or from the **Help** menu in the IAR Embedded Workbench IDE. The online help system is also available via the F1 key.

USER AND REFERENCE GUIDES

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

- System requirements and information about how to install and register the IAR Systems products, refer to the booklet *Quick Reference* (available in the product box) and the *Installation and Licensing Guide*.
- Getting started using IAR Embedded Workbench and the tools it provides, see the guide *Getting Started with IAR Embedded Workbench®* .
- Using the IDE for project management and building, see the *IDE Project Management and Building Guide*
- Using the IAR C-SPY® Debugger, see the *C-SPY® Debugging Guide for 8051*
- Programming for the IAR C/C++ Compiler for 8051, see the *IAR C/EC++ Compiler Reference Guide for 8051*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, see the *IAR Linker and Library Tools Reference Guide.*

- Using the IAR DLIB Library, see the *DLIB Library Reference information*, available in the online help system.
- Using the IAR CLIB Library, see the *IAR C Library Functions Reference Guide*, available in the online help system.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for 8051, see the Information Center for a list of migration guides.
- Developing safety-critical applications using the MISRA C guidelines, see the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.

Note: Additional documentation might be available depending on your product installation.

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Comprehensive information about debugging using the IAR C-SPY® Debugger
- Reference information about the menus, windows, and dialog boxes in the IDE
- Compiler reference information
- Keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1. Note that if you select a function name in the editor window and press F1 while using the CLIB library, you will get reference information for the DLIB library.

WEB SITES

Recommended web sites:

- The website of your chip manufacturer.
- The IAR Systems web site, **www.iar.com**, holds application notes and other product information.

Finally, the Embedded C++ Technical Committee web site, **www.caravan.net/ec2plus**, contains information about the Embedded C++ standard.

Document conventions

When, in this text, 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 `8051\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.`*`n`*`\8051\doc`.

TYPOGRAPHIC CONVENTIONS

This guide 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 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

NAMING CONVENTIONS

The following naming conventions are used for the products and tools from IAR Systems® referred to in this guide:

Brand name	Generic term
IAR Embedded Workbench® for 8051	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for 8051	the IDE

Table 2: Naming conventions used in this guide

Brand name	Generic term
IAR C-SPY® Debugger for 8051	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for 8051	the compiler
IAR Assembler™ for 8051	the assembler
IAR XLINK Linker™	XLINK, the linker
IAR XAR Library Builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Library™	the DLIB library
IAR CLIB Library™	the CLIB library

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

Introduction to the IAR Assembler

This chapter describes the source code format for the 8051 IAR Assembler and provides programming hints.

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

Source format

The format of an assembler source line is as follows:

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

where the components are as follows:

label A label, which is assigned the value and type of the current program location counter (PLC). The `:` (colon) is optional if the label starts in the first column.

operation An assembler instruction or directive. This must not start in the first column.

operands An assembler instruction can have zero, one, or more operands.

The data definition directives, for example DB and DC8, can have any number of operands. For reference information about the data definition directives, see *Data definition or allocation directives*, page 87.

Other assembler directives can have one, two, or three operands, separated by commas.

comment Comment, preceded by a `;` (semicolon).

The fields can be separated by spaces or tabs.

A source line may not exceed 2047 characters.

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

The 8051 IAR Assembler uses the default filename extensions s51, asm, and msa for source files.

Assembler expressions

Expressions consist of operands and operators.

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

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators; see also *Precedence of operators*, page 37.

The following operands are valid in an expression:

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

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

The valid operators are described in the chapter *Assembler operators*, page 37.

TRUE AND FALSE

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

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

USING SYMBOLS IN RELOCATABLE EXPRESSIONS

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

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

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

```
       NAME       prog1
       PUBLIC     first
       PUBLIC     second
       RSEG       DATA
first  DB         5
second DB         3
       ENDMOD
```

```
MODULE     prog2
EXTERN     first
EXTERN     second
RSEG       CODE
MOV  A,first
MOV  A,first+1
MOV  A,1+first
MOV  A,first/second
ENDMOD
```

SYMBOLS

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

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

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. See page 33 for additional information.

Notice that symbols and labels are byte addresses. For additional information, see *Generating lookup table*, page 88.

LABELS

Symbols used for memory locations are referred to as labels.

Program location counter (PLC)

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

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

```
SJMP   $      ; Loop forever
```

INTEGER CONSTANTS

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

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

Commas and decimal points are not permitted.

The following types of number representation are supported:

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

Table 3: Integer constant formats

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

ASCII CHARACTER CONSTANTS

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

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

Table 4: ASCII character constant formats

FLOATING-POINT CONSTANTS

The 8051 IAR Assembler will accept floating-point values as constants and convert them into IEEE single-precision (signed 32-bit) floating-point format or fractional format.

Floating-point numbers can be written in the format:

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

The following table shows some valid examples:

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

Table 5: Floating-point constants

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

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

PREDEFINED SYMBOLS

The 8051 IAR Assembler defines a set of symbols for use in assembler source files. The symbols provide information about the current assembly, allowing you to test them in preprocessor directives or include them in the assembled code. The strings returned by the assembler are enclosed in double quotes.

The following predefined symbols are available:

Symbol	Value
__DATE__	Current date in dd/Mmm/yyyy format (string).
__FILE__	Current source filename (string).
__IAR_SYSTEMS_ASM__	IAR assembler identifier (number).
__LINE__	Current source line number (number).
__TID__	Target identity, consisting of two bytes (number). The high byte is the target identity, which is 32 (0x20) for A8051. The low byte is the processor option *16. The following values are therefore possible: -v0 0x2000 -v1 0x2010 -v2 0x2020
__TIME__	Current time in hh:mm:ss format (string).
__VER__	Version number in integer format; for example, version 4.17 is returned as 417 (number).

Table 6: Predefined symbols

Notice that `__TID__` is related to the predefined symbol `__TID__` in the 8051 IAR C/EC++ Compiler. It is described in the *IAR C/EC++ Compiler Reference Guide for 8051*.

Including symbol values in code

There are several data definition directives provided to 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:

```
      RSEG    DATA
td    DB      __TIME__,",",__DATE__,0 ; time and date

      RSEG    CODE
      EXTERN  printstring
main
      MOV      A,td          ; load address of string
      MOV      R1,A
      LCALL    printstring   ; routine to print string
      RET
```

Testing symbols for conditional assembly

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

For example, in a source file written for use on any one of the 8051 family members, you may want to assemble appropriate code for a specific processor. You could do this using the `__TID__` symbol as follows:

```
#define TARGET ((__TID__& 0x0F00)>>4)
#if (TARGET==0x02)
...
#else
...
#endif
```

See *Conditional assembly directives*, page 69.

Register symbols

This table shows the existing predefined register symbols:

Register symbol	Addressing	Description
`R0–R7`	8-bit	Data registers

Table 7: Register symbols

Register symbol	Addressing	Description
A	8-bit	Data register
B	8-bit	Data register or SFR address of register B
ACC	8-bit	SFR address of register A
DPL	8-bit	SFR address of the low part of register DPTR
DPH	8-bit	SFR address of the high part of register DPTR
PSW	8-bit	SFR address of register PSW (program status word)

Table 7: Register symbols (Continued)

Programming hints

This section gives hints on how to write efficient code for the 8051 IAR Assembler. For information about projects including both assembler and C or Embedded C++ source files, see the *IAR C/EC++ Compiler Reference Guide for 8051*.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of 8051 derivatives are included in the IAR product package, in the \8051\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 8051 IAR C/EC++ Compiler, and they are suitable to use as templates when creating new header files for other 8051 derivatives.

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

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

USING C-STYLE PREPROCESSOR DIRECTIVES

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

USING JMP AND CALL

JMP is a pseudo mnemonic which is expanded to the smallest possible of the instructions SJMP, AJMP, or LJMP. If the expression is unresolved, the assembler expands JMP to LJMP, because that instruction can reach the entire address space. Likewise, CALL is a pseudo mnemonic which is expanded to the smallest possible of the instructions ACALL or LCALL. If the expression is unresolved, the assembler expands CALL to LCALL, because that instruction can reach the entire address space.

For this reason, we recommend that you decide which instruction that you need, and do not use JMP or CALL unnecessarily.

Upgrading from previous versions of the assembler

The current version of the IAR C/EC++ Compiler for 8051 has been completely rewritten to achieve a substantial increase in code efficiency. Because of this, the assembler interface to C functions has been changed and is incompatible with version 5 and earlier in object code.

However, the new assembler is source code compatible with previous versions. Reassembled source code can be used together with version 6 or later of the IAR Assembler for 8051. Note, however, that the byte order has been changed from big-endian to little-endian.

Assembler options

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

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

Setting command line options

To set assembler options from the command line, you include them on the command line, after the `a8051` command:

```
a8051 [options] [sourcefile] [options]
```

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

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

For example, when assembling the source file `power2.s51`, use the following command to generate a list file to the default filename (`power2.lst`):

```
a8051 power2 -L
```

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

```
a8051 power2 -l list.lst
```

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

```
a8051 power2 -Llist\
```

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

EXTENDED COMMAND LINE FILE

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

By default, extended command line files have the extension `xcl`, and can be specified using the `-f` command line option. For example, to read the command line options from `extend.xcl` when assembling the file `source.s51`, enter:

```
a8051 source.s51 -f extend.xcl
```

ERROR RETURN CODES

When using the 8051 IAR Assembler from within a batch file, you may need to determine whether the assembly was successful in order to decide what step to take next. For this reason, the assembler returns the following error return codes:

Return code	Description
`0`	Assembly successful, warnings may appear
`1`	There were warnings (only if the `-ws` option is used)
`2`	There were errors

Table 8: Assembler error return codes

ASSEMBLER ENVIRONMENT VARIABLES

Options can also be specified using the `ASM8051` environment variable. The assembler appends the value of this variable to every command line, so it provides a convenient method of specifying options that are required for every assembly.

The following environment variables can be used with the 8051 IAR Assembler:

Environment variable	Description
`ASM8051`	Specifies command line options; for example: `set ASM8051=-L -ws`
`A8051_INC`	Specifies directories to search for include files; for example: `set A8051_INC=c:\myinc\`

Table 9: Assembler environment variables

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

```
ASM8051=-l temp.lst
```

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

Summary of assembler options

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

Command line option	Description
`-B`	Macro execution information
`-b`	Makes a library module
`-c{SDMEAO}`	Conditional list
`-D`*`symbol`*`[=`*`value`*`]`	Defines a symbol
`-d`	Disable `#ifdef/#endif` matching
`-E`*`number`*	Maximum number of errors
`-f` *`filename`*	Extends the command line
`-G`	Opens standard input as source
`-I`*`prefix`*	Includes paths
`-i`	Lists #included text
`-L[`*`prefix`*`]`	Lists to prefixed source name
`-l` *`filename`*	Lists to named file
`-M`*`ab`*	Macro quote characters
`-N`	Omit header from assembler listing
`-n`	Enables support for multibyte characters
`-O`*`prefix`*	Sets object filename prefix
`-o` *`filename`*	Sets object filename
`-p`*`lines`*	Lines/page
`-r`	Generates debug information
`-S`	Sets silent operation
`-s{+|-}`	Case sensitive user symbols
`-T`	Active lines only
`-t`*`n`*	Tab spacing
`-U`*`symbol`*	Undefines a symbol
`-v[0|1|2]`	Processor configuration
`-w[`*`string`*`][s]`	Disables warnings
`-X`	Unreferenced externals in object file
`-x{DI2}`	Includes cross-references

Table 10: Assembler options summary

Descriptions of assembler options

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

-B

```
-B
```

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

- The name of the macro
- The definition of the macro
- The arguments to the macro
- The expanded text of the macro.

This option is mainly used in conjunction with the list file options `-L` or `-l`; for additional information, see page 30.

This option is identical to the **Macro execution info** option on the **List** page in the **A8051** category in the IAR Embedded Workbench.

-b

```
-b
```

This option causes the object file to be a library module rather than a program module. A program module is always included during linking. A library module will only be included if it is referenced in your application.

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

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

This option is identical to the **Make library module** option on the **Output** page in the **A8051** category in the IAR Embedded Workbench.

-c

```
-c{SDMEAO}
```

Use this option to control the contents of the assembler list file. This option is mainly used in conjunction with the list file options `-L` and `-l`; see page 30 for additional information.

The following table shows the available parameters:

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

Table 11: Conditional list (-c)

This option is related to the **Output list file** option on the **List** page in the **A8051** category in the IAR Embedded Workbench.

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

Use this option to define a preprocessor symbol with the name *`symbol`* and the value *`value`*. If no value is specified, 1 is used.

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

Example

For example, you could arrange your source to produce either the test or production version of your program dependent on whether the symbol `TESTVER` was defined. To do this, use include sections such as:

```
#ifdef  TESTVER
...    ; additional code lines for test version only
#endif
```

Then select the version required in the command line as follows:

Production version:	`a8051 prog`
Test version:	`a8051 prog -DTESTVER`

Alternatively, your source might use a variable that you need to change often. You can then leave the variable undefined in the source, and use `-D` to specify the value on the command line; for example:

```
a8051 prog -DFRAMERATE=3
```

This option is identical to the **Defined symbols** option on the **Preprocessor** page in the **A8051** category in the IAR Embedded Workbench.

-d

-d

Allows unmatched #ifdef … #endif statements to be used without causing an error.

The checks for #ifdef … #endif matching are performed for each module, and a #endif outside modules will therefore normally generate an error message. Use this option to turn checking off.

Example

This allows you to write constructs such as:

```
#ifdef Version1
  MODULE M1
  NOP
  ENDMOD
#endif
  MODULE M2
  .
  .
  .
  etc
```

This option is identical to the **Disable #ifdef/#endif matching** option on the **Language** page in the **A8051** category in the IAR Embedded Workbench.

-E

-E*number*

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

By default, the maximum number is 100. The -E option allows you to decrease or increase this number to see more or fewer errors in a single assembly.

-f

-f *filename*

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

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

Example

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

```
a8051 prog -f extend.xcl
```

-G

`-G`

This option causes the assembler to read the source from the standard input stream, rather than from a specified source file.

When `-G` is used, no source filename may be specified.

-I

`-I`*`prefix`*

Use this option to specify paths to be used by the preprocessor by adding the `#include` file search prefix *`prefix`*.

By default, the assembler searches for `#include` files only in the current working directory and in the paths specified in the `A8051_INC` environment variable. The `-I` option allows you to give the assembler the names of directories where it will also search if it fails to find the file in the current working directory.

Example

Using the options:

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

and then writing:

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

in the source, will make the assembler search first in the current directory, then in the directory `c:\global\`, and finally in the directory `c:\thisproj\headers\`.

You can also specify the include path with the `A8051_INC` environment variable, see *Assembler environment variables*, page 24.

This option is related to the **Include paths** option on the **Preprocessor** page in the **A8051** category in the IAR Embedded Workbench.

-i

`-i`

Includes `#include` files in the list file.

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

This option is related to the **Include paths** option on the **Preprocessor** page in the **A8051** category in the IAR Embedded Workbench.

-L

`-L[prefix]`

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

To simply generate a listing, use the `-L` option without a prefix. The listing is sent to the file with the same name as the source, but the extension will be `lst`.

The `-L` option lets you specify a prefix, for example to direct the list file to a subdirectory. Notice that you cannot include a space before the prefix.

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

Example

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

```
a8051 prog -Llist\
```

This option is related to the options on the **List** page in the **A8051** category in the IAR Embedded Workbench.

-l

`-l` *filename*

Use this option to make the assembler generate a listing and send it to the file *filename*. If no extension is specified, `lst` is used. Notice that you must include a space before the filename.

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

This option is related to the options on the **List** page in the **A8051** category in the IAR Embedded Workbench.

-M

`-M`*ab*

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

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

Example

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

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

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

a8051 *filename* -M'<>'

This option is identical to the **Macro quote characters** option on the **Language** page in the **A8051** category in the IAR Embedded Workbench.

-N

```
-N
```

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

This option is useful in conjunction with the list file options `-L` or `-l`; see page 30 for additional information.

This option is related to the options on the **List** page in the **A8051** category in the IAR Embedded Workbench.

-n

```
-n
```

By default, multibyte characters cannot be used in assembler source code. If you use this option, multibyte characters in the source code are interpreted according to the host computer's default setting for multibyte support.

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

This option is identical to the **Enable multibyte support** option on the **Language** page in the **A8051** category in the IAR Embedded Workbench.

-O

-O*prefix*

Use this option to set the prefix to be used on the name of the object file. Notice that you cannot include a space before the prefix.

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

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

Example

To send the object code to the file `obj\prog.r51` rather than to the default file `prog.r51`:

```
a8051 prog -Oobj\
```

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

-o

`-o` *`filename`*

This option sets the filename to be used for the object file. Notice that you must include a space before the filename. If no extension is specified, `r51` is used.

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

Example

For example, the following command puts the object code to the file `obj.r51` instead of the default `prog.r51`:

```
a8051 prog -o obj
```

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

-p

`-p`*`lines`*

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

This option is used in conjunction with the list options `-L` or `-l`; see page 30 for additional information.

This option is identical to the **Lines/page** option on the **List** page in the **A8051** category in the IAR Embedded Workbench.

-r

`-r`

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

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

This option is identical to the **Generate debug information** option on the **Output** page in the **A8051** category in the IAR Embedded Workbench.

-S

`-S`

By default, the assembler sends various informational messages via the standard output stream. Use the `-S` option to prevent this.

Error and warning messages are sent to the error output stream, so they are displayed regardless of this setting.

-s

`-s{+|-}`

Use the `-s` option to control whether the assembler is sensitive to the case of user symbols:

Command line option	Description
`-s+`	Case sensitive user symbols
`-s-`	Case insensitive user symbols

Table 12: Controlling case sensitivity in user symbols (-s)

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

This option is identical to the **User symbols are case sensitive** option on the **Language** page in the **A8051** category in the IAR Embedded Workbench.

-T

`-T`

Includes only active lines in listings, for example not those in false `#if` blocks. By default, all lines are listed.

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

This option is identical to the **Active lines only** option on the **List** page in the **A8051** category in the IAR Embedded Workbench.

-t -t*n*

By default the assembler sets 8 character positions per tab stop. The `-t` option allows you to specify a tab spacing to *n*, which must be in the range 2 to 9.

This option is useful in conjunction with the list options `-L` or `-l`; see page 30 for additional information.

This option is identical to the **Tab spacing** option on the **List** page in the **A8051** category in the IAR Embedded Workbench.

-U -U*symbol*

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

By default, the assembler provides certain predefined symbols; see *Predefined symbols*, page 19. The `-U` option allows you to undefine such a predefined symbol to make its name available for your own use through a subsequent `-D` option or source definition.

Example

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

```
a8051 prog -U__TIME__
```

-v -v[0|1|2]

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

The following table shows how the `-v` options are mapped to the 8051 derivatives:

Option	Description	Derivative
-v0	Supports derivatives that use a standard 8051 core, with a maximum of 64 Kbytes of code memory. This option corresponds to the compiler option `--cpu=plain`.	8051
-v1	Supports derivatives with a maximum of 2 Kbytes of code memory. Using this processor option, no long jump (`LJMP`) instructions will be generated, only the shorter `AJMP` instructions. This option corresponds to the compiler option `--cpu=tiny`.	80751

Table 13: Specifying the processor configuration (-v)

Option	Description	Derivative
`-v2`	Supports derivatives that use cores similar to the extended core of the Dallas DS80C390/DS80C400 processors. Using this processor option, 3-byte addresses will be generated when appropriate. This option corresponds to the compiler option `--cpu=extended1`.	Dallas DS80C390/ DS80C400

Table 13: Specifying the processor configuration (-v) (Continued)

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

The `-v` option is identical to the **CPU core** option on the **Target** page in the **General** category in the IAR Embedded Workbench.

`-w` `-w[+|-][[,]range][,range,...][s]`

By default, the assembler displays a warning message when it detects an element of the source which is legal in a syntactical sense, but may contain a programming error; see *Assembler diagnostics*, page 107, for details.

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

Command line option	Description
`-w+`	Enables all warnings
`-w-`	Disables all warnings
`-w+n`	Enables just warning *n*
`-w-n`	Disables just warning *n*
`-w+m-n`	Enables warnings *m* to *n*
`-w-m-n`	Disables warnings *m* to *n*
`-w+,-m-n`	Enables all warnings except *m* to *n*
`-w-,+m-n`	Disables all warnings except *m* to *n*
`-w+,-m-n,-o-p`	Enables all warnings except *m* to *n* and *o* to *p*
`-w-,+m-n,+o-p`	Disables all warnings except *m* to *n* and *o* to *p*

Table 14: Disabling assembler warnings (-w)

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

By default, the assembler generates exit code 0 for warnings. Use `-ws` to generate exit code 1 if a warning message is produced.

Example

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

```
a8051 prog -w-0
```

To disable warnings 0 to 8 and 14-15, use the following command:

```
a8051 prog -w-0-8,-14-15
```

This option is related to the options on the **Diagnostics** page in the **A8051** category in the IAR Embedded Workbench.

-X

`-X`

Use this option to force all unreferenced externally declared symbols to be included in the object file.

-x

`-x{DI2}`

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

This option is useful in conjunction with the list options `-L` or `-l`; see page 30 for additional information.

The following parameters are available:

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

Table 15: Including cross-references in assembler list file (-x)

This option is identical to the **Include cross-reference** option on the **List** page in the **A8051** category in the IAR Embedded Workbench.

Assembler operators

This chapter first describes the precedence of the assembler operators, and then summarizes the operators, classified according to their precedence. Finally, this chapter provides reference information about each operator, presented in alphabetical order.

Precedence of operators

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

The following rules determine how expressions are evaluated:

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

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

Summary of assembler operators

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

UNARY OPERATORS – 1

`+`	Unary plus.
`–`	Unary minus.
`!`, `NOT`	Logical NOT.
`~`, `BITNOT`	Bitwise NOT.
`LOW`	Low byte.
`HIGH`	High byte.
`BYTE2`	Second byte.

BYTE3	Third byte.
BYTE4	Fourth byte
LWRD	Low word.
HWRD	High word.
DATE	Current time/date.
LOC	Local variable reference.
PRM	Parameter reference
SFB	Segment begin.
SFE	Segment end.
SIZEOF	Segment size.

MULTIPLICATIVE ARITHMETIC OPERATORS – 2

*	Multiplication.
/	Division.
%	Modulo.

ADDITIVE ARITHMETIC OPERATORS – 3

+	Addition.
–	Subtraction.

SHIFT OPERATORS – 4

>>, SHR	Logical shift right.
<<, SHL	Logical shift left.

AND OPERATORS – 5

&&, AND	Logical AND.
&, BITAND	Bitwise AND.

OR OPERATORS – 6

||, OR	Logical OR.
|, BITOR	Bitwise OR.

XOR	Logical exclusive OR.
^, BITXOR	Bitwise exclusive OR.

COMPARISON OPERATORS – 7

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

Description of operators

The following sections give detailed descriptions of each assembler operator. See *Assembler expressions*, page 16, for related information. The number within parentheses specifies the priority of the operator.

* Multiplication (2).

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

Example

```
2*2 → 4
-2*2 → -4
```

+ Unary plus (1).

Unary plus operator.

Example

```
+3 → 3
3*+2 → 6
```

+ Addition (3).

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

Example

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

- Unary minus (1).

The unary minus operator performs arithmetic negation on its operand.

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

Example

```
-3 → -3
3*-2 → -6
4--5 → 9
```

- Subtraction (3).

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

Example

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

/ Division (2).

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

Example

```
9/2 → 4
-12/3 → -4
9/2*6 → 24
```

<, LT Less than (7).

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

Example

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

<=, LE Less than or equal (7)

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

Example

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

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

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

Example

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

=, ==, EQ Equal (7).

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

Example

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

`>, GT` Greater than (7).

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

Example

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

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

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

Example

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

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

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

Example

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

`&, BITAND` Bitwise AND (5).

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

Example

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

~, BITNOT

Bitwise NOT (1).

Use ~ to perform bitwise NOT on its operand.

Example

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

|, BITOR

Bitwise OR (6).

Use | to perform bitwise OR on its operands.

Example

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

^, BITXOR

Bitwise exclusive OR (6).

Use ^ to perform bitwise XOR on its operands.

Example

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

%, MOD

Modulo (2).

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

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

Example

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

!, NOT

Logical NOT (1).

Use `!` to negate a logical argument.

Example

```
! B'0101 → 0
! B'0000 → 1
```

||, OR Logical OR (6).

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

Example

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

BYTE2 Second byte (1).

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

Example

```
BYTE2 0x12345678 → 0x56
```

BYTE3 Third byte (1).

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

Example

```
BYTE3 0x12345678 → 0x34
```

BYTE4 Fourth byte (1).

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 (1).

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

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

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

Example

To assemble the date of assembly:

```
today: DC8 DATE 6, DATE 5, DATE 4
```

HIGH High byte (1).

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

Example

```
HIGH 0xABCD → 0xAB
```

HWRD High word (1).

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

Example

```
HWRD 0x12345678 → 0x1234
```

LOC Local variable reference (2)

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

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

See also the *IAR C/EC++ Compiler Reference Guide for 8051* for information about the assembler language interface.

Syntax

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

Parameters

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

Example

```
        MOV     R0,#LOC(func,IOVERLAY,0)
```

This will load the address of the first local variable of `func` into the `R0` register. The `IOVERLAY` memory segment is used for storing static overlay frames.

`LOW` Low byte (1).

`LOW` takes a single operand, which is interpreted as an unsigned, 16-bit integer value. The result is the unsigned, 8-bit integer value of the lower order byte of the operand.

Example

`LOW 0xABCD` → `0xCD`

`LWRD` Low word (1).

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

Example

`LWRD 0x12345678` → `0x5678`

`PRM` Parameter reference (2).

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

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

See also the *IAR C/EC++ Compiler Reference Guide for 8051* for information about the assembler language interface.

Syntax

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

Parameters

function The name of the function.

segment The name of a memory segment, which must be defined before PRM is used.

offset An offset from the start address.

Example

```
        MOV     R0,#PRM(func,IOVERLAY,0)
```

This will load the address of the first parameter of func into the R0 register. The IOVERLAY memory segment is used for storing static overlay frames.

SFB Segment begin (1).

Syntax

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

Parameters

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

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

Description

SFB accepts a single operand to its right. The operand must be the name of a relocatable segment.

The operator evaluates to the absolute address of the first byte of that segment. This evaluation takes place at linking time.

Example

```
       NAME  demo
       RSEG  CODE
start: DC16  SFB(CODE)
```

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

`SFE` Segment end (1).

Syntax

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

Parameters

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

Description

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

Example

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

Even if the above code is linked with many other modules, end will still be set to the address of the last byte of the segment.

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

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

`<<, SHL` Logical shift left (4).

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

Example

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

`>>, SHR` Logical shift right (4).

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

Example

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

`SIZEOF` Segment size (1).

Syntax

`SIZEOF` *`segment`*

Parameters

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

Description

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

Example

```
      NAME   demo
      RSEG   CODE
size: DC16   SIZEOF CODE
```

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

UGT Unsigned greater than (7).

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

Example

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

ULT Unsigned less than (7).

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

Example

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

XOR Logical exclusive OR (6).

Use XOR to perform logical XOR on its two operands.

Example

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

Assembler directives

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

Summary of assembler directives

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

Directive	Description	Section
`$`	Includes a file.	Assembler control
`#define`	Assigns a value to a label.	C-style preprocessor
`#elif`	Introduces a new condition in a `#if…#endif` block.	C-style preprocessor
`#else`	Assembles instructions if a condition is false.	C-style preprocessor
`#endif`	Ends a `#if`, `#ifdef`, or `#ifndef` block.	C-style preprocessor
`#error`	Generates an error.	C-style preprocessor
`#if`	Assembles instructions if a condition is true.	C-style preprocessor
`#ifdef`	Assembles instructions if a symbol is defined.	C-style preprocessor
`#ifndef`	Assembles instructions if a symbol is undefined.	C-style preprocessor
`#include`	Includes a file.	C-style preprocessor
`#message`	Generates a message on standard output.	C-style preprocessor
`#undef`	Undefines a label.	C-style preprocessor
`/*comment*/`	C-style comment delimiter.	Assembler control
`//`	C++ style comment delimiter.	Assembler control
`=`	Assigns a permanent value local to a module.	Value assignment
`ALIAS`	Assigns a permanent value local to a module.	Value assignment
`ALIGN`	Aligns the location counter by inserting zero-filled bytes.	Segment control
`ALIGNRAM`	Aligns the program counter.	Segment control
`ARGFRAME`	Defines a function's arguments.	Function control
`ASEG`	Begins an absolute segment.	Segment control
`ASEGN`	Begins a named absolute segment.	Segment control

Table 16: Assembler directives summary

Directive	Description	Section
ASSIGN	Assigns a temporary value.	Value assignment
CASEOFF	Disables case sensitivity.	Assembler control
CASEON	Enables case sensitivity.	Assembler control
CFI	Specifies call frame information.	Call frame information
COL	Sets the number of columns per page.	Listing control
COMMON	Begins a common segment.	Segment control
DB	Generates 8-bit byte constants, including strings.	Data definition or allocation
DC8	Generates 8-bit byte constants, including strings.	Data definition or allocation
DC16	Generates 16-bit word constants.	Data definition or allocation
DC24	Generates 24-bit word constants.	Data definition or allocation
DC32	Generates 32-bit long word constants.	Data definition or allocation
DD	Generates 32-bit long word constants.	Data definition or allocation
DEFINE	Defines a file-wide value.	Value assignment
DS	Allocates space for 8-bit bytes.	Data definition or allocation
DS16	Allocates space for 16-bit words.	Data definition or allocation
DS24	Allocates space for 24-bit words.	Data definition or allocation
DS32	Allocates space for 32-bit words.	Data definition or allocation
DS8	Allocates space for 8-bit bytes.	Data definition or allocation
DT	Generates 24-bit word constants.	Data definition or allocation
DW	Generates 16-bit word constants, including strings.	Data definition or allocation
ELSE	Assembles instructions if a condition is false.	Conditional assembly

Table 16: Assembler directives summary (Continued)

Directive	Description	Section
ELSEIF	Specifies a new condition in an IF…ENDIF block.	Conditional assembly
END	Terminates the assembly of the last module in a file.	Module control
ENDIF	Ends an IF block.	Conditional assembly
ENDM	Ends a macro definition.	Macro processing
ENDMAC	Ends a macro definition.	Macro processing
ENDMOD	Terminates the assembly of the current module.	Module control
ENDR	Ends a REPT, REPTC or REPTI structure.	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment
EVEN	Aligns the program counter to an even address.	Segment control
EXITM	Exits prematurely from a macro.	Macro processing
EXPORT	Exports symbols to other modules.	Symbol control
EXTERN	Imports an external symbol.	Symbol control
EXTRN	Imports an external symbol.	Symbol control
FUNCALL	Defines function call information.	Function control
FUNCTION	Defines a function.	Function control
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
LOCFRAME	Defines a function's local variables.	Function control
LSTCND	Controls conditional assembler listing.	Listing control
LSTCOD	Controls multi-line code listing.	Listing control
LSTEXP	Controls the listing of macro generated lines.	Listing control
LSTMAC	Controls the listing of macro definitions.	Listing control
LSTOUT	Controls assembler-listing output.	Listing control
LSTPAG	Controls the formatting of output into pages.	Listing control
LSTREP	Controls the listing of lines generated by repeat directives.	Listing control
LSTSAS	Controls structured assembler listing.	Listing control
LSTXRF	Generates a cross-reference table.	Listing control

Table 16: Assembler directives summary (Continued)

Directive	Description	Section
`MACRO`	Defines a macro.	Macro processing
`MODULE`	Begins a library module.	Module control
`NAME`	Begins a program module.	Module control
`ODD`	Aligns the program counter to an odd address.	Segment control
`ORG`	Sets the location counter.	Segment control
`PAGE`	Generates a new page.	Listing control
`PAGSIZ`	Sets the number of lines per page.	Listing control
`PROGRAM`	Begins a program module.	Module control
`PUBLIC`	Exports symbols to other modules.	Symbol control
`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`	Repeats subsequent instructions until a condition is true.	Symbol control
`RSEG`	Begins a relocatable segment.	Segment control
`RTMODEL`	Declares runtime model attributes.	Module control
`SET`	Assigns a temporary value.	Value assignment
`sfr`	Creates byte-access SFR labels.	Value assignment
`SFRTYPE`	Specifies SFR attributes.	Value assignment
`STACK`	Begins a stack segment.	Segment control

Table 16: Assembler directives summary (Continued)

Syntax conventions

In the syntax definitions the following conventions are used:

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

  ORG *expr*

  expr represents an arbitrary expression.

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

  ```
  END [expr]
  ```

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

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

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

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

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

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

LABELS AND COMMENTS

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

```
label SET expr
```

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

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

PARAMETERS

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

Parameter	**What it consists of**
expr	An expression; see *Assembler expressions*, page 16.
label	A symbolic label.
symbol	An assembler symbol.

Table 17: Assembler directive parameters

Module control directives

Module control directives are used for marking the beginning and end of source program modules, and for assigning names and types to them.

Directive	Description
`END`	Terminates the assembly of the last module in a file.
`ENDMOD`	Terminates the assembly of the current module.
`LIBRARY`	Begins a library module.
`MODULE`	Begins a library module.
`NAME`	Begins a program module.
`PROGRAM`	Begins a program module.
`RTMODEL`	Declares runtime model attributes.

Table 18: Module control directives

SYNTAX

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

PARAMETERS

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

DESCRIPTION

Beginning a program module

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

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

Beginning a library module

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

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

Terminating a module

Use ENDMOD to define the end of a module.

Terminating the last module

Use END to indicate the end of the source file. Any lines after the END directive are ignored.

Assembling multi-module files

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

The following rules apply when assembling multi-module files:

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

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

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

Declaring runtime model attributes

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

A module can have several runtime model definitions.

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

If you are writing assembler routines for use with C code, and you want to control the module consistency, refer to the *IAR C/EC++ Compiler Reference Guide for 8051.*

Examples

The following example defines three modules where:

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

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

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

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

Symbol control directives

These directives control how symbols are shared between modules.

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

Table 19: Symbol control directives

SYNTAX

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

PARAMETERS

symbol	Symbol to be imported or exported.

DESCRIPTION

Exporting symbols to other modules

Use `PUBLIC` to make one or more symbols available to other modules. Symbols declared `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 are no restrictions on the number of `PUBLIC`-declared symbols in a module.

Exporting symbols with multiple definitions to other modules

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

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

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

Importing symbols

Use EXTERN to import an untyped external symbol.

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

EXAMPLES

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

```
     NAME   error
     EXTERN print
     PUBLIC err

err  CALL   print
     DB     "** Error **"
     RET
     END    err
```

Segment control directives

The segment directives control how code and data are generated.

Directive	Description
ALIGN	Aligns the location counter by inserting zero-filled bytes.
ALIGNRAM	Aligns the program counter.
ASEG	Begins an absolute segment.
ASEGN	Begins a named absolute segment.
COMMON	Begins a common segment.

Table 20: Segment control directives

Directive	Description
EVEN	Aligns the program counter to an even address.
ODD	Aligns the program counter to an odd address.
ORG	Sets the location counter.
RSEG	Begins a relocatable segment.
STACK	Begins a stack segment.

Table 20: Segment control directives

SYNTAX

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

PARAMETERS

address — Address where this segment part will be placed.

align — Exponent of the value to which the address should be aligned, in the range 0 to 30.

expr — Address to set the location counter to.

flag — NOROOT, ROOT

NOROOT means that the segment part may be discarded by the linker if no symbols in this segment part are referred to. Normally all segment parts except startup code and interrupt vectors should set this flag. The default mode is ROOT which indicates that the segment part must not be discarded.

REORDER, NOREORDER

REORDER allows the linker to reorder segment parts. For a given segment, all segment parts must specify the same state for this flag. The default mode is NOREORDER which indicates that the segment parts must remain in order.

SORT, NOSORT
SORT means that the linker will sort the segment parts in decreasing alignment order. For a given segment, all segment parts must specify the same state for this flag. The default mode is NOSORT which indicates that the segment parts will not be sorted.

segment The name of the segment.

start A start address that has the same effect as using an ORG directive at the beginning of the absolute segment.

type The memory type, typically CODE, or DATA. In addition, any of the types supported by the IAR XLINK Linker.

value Byte value used for padding, default is zero.

DESCRIPTION

Beginning an absolute segment

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

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

Beginning a named absolute segment

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

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

Beginning a relocatable segment

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

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

Beginning a stack segment

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

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

Beginning a common segment

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

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

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

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

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

Setting the program location counter (PLC)

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

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

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

Aligning a segment

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

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

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

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

EXAMPLES

Beginning an absolute segment

The following example assembles interrupt routine entry addresses in the appropriate 8051 interrupt vectors using an absolute segment:

```
EXTERN      iesrv,t0srv

            ASEG
            ORG   0
            JMP   main        ; Power on

            ORG   3
            JMP   iesrv       ; External interrupt

            ORG   0BH
            JMP   t0srv       ; Timer interrupt

            ORG   30H
main:       MOV   A,#1

            END
```

Beginning a relocatable segment

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

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

```
       EXTERN     divrtn,mulrtn

       RSEG       table
       DW         divrtn,mulrtn

       ORG        $+6
       DW         subrtn

       RSEG       code
subrtn MOV        A,R7
       SUBB       A,#20
       MOV        R7,A
       END
```

Beginning a stack segment

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

```
       STACK      rpnstack
parms  DS         100
opers  DS         100
       END
```

The data is allocated from high to low addresses.

Beginning a common segment

The following example defines two common segments containing variables:

```
       NAME       common1
       COMMON     data
count  DD         1
       ENDMOD

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

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

Value assignment directives

These directives are used for assigning values to symbols.

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

Table 21: Value assignment directives

Directive	Description
sfr	Creates byte-access SFR labels.
SFRTYPE	Specifies SFR attributes.

Table 21: Value assignment directives

SYNTAX

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

PARAMETERS

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

DESCRIPTION

Defining a temporary value

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

Defining a permanent local value

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

Use EQU to create a local symbol that denotes a number or offset.

The symbol is only valid in the module in which it was defined, but can be made available to other modules with a PUBLIC directive.

Use EXTERN to import symbols from other modules.

Defining a permanent global value

Use DEFINE to define symbols that should be known to all modules in the source file.

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

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

Defining special function registers

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

Checking symbol values

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

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

EXAMPLES

Redefining a symbol

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

```
            NAME        table
cons        SET         1
```

```
buildit     MACRO       times
            DW          cons
cons        SET         cons * 3
            IF          times > 1
            buildit     times - 1
            ENDIF
            ENDM
main        buildit     4
            END
```

It generates the following code:

```
  1    000000                  NAME    table
  2    000001          cons    SET     1
 10    000000          main    buildit    4
 10    000000          main    buildit    4
 10.1  000000 0001             DW      cons
 10.2  000003          cons    SET     cons * 3
 10.3  000002                  IF      4 > 1
 10.4  000002                  buildit    4 - 1
 10.5  000002 0003             DW      cons
 10.6  000009          cons    SET     cons * 3
 10.7  000004                  IF      4 - 1 > 1
 10.8  000004                  buildit    4 - 1 - 1
 10.9  000004 0009             DW      cons
 10.10 00001B          cons    SET     cons * 3
 10.11 000006                  IF      4 - 1 - 1 > 1
 10.12 000006                  buildit    4 - 1 - 1 - 1
 10.13 000006 001B             DW      cons
 10.14 000051          cons    SET     cons * 3
 10.15 000008                  IF      4 - 1 - 1 - 1 > 1
 10.16 000008                  buildit    4 - 1 - 1 - 1 - 1
 10.17 000008                  ENDIF
 10.18 000008                  ENDM
 10.19 000008                  ENDIF
 10.20 000008                  ENDM
 10.21 000008                  ENDIF
 10.22 000008                  ENDM
 10.23 000008                  ENDIF
 10.24 000008                  ENDM
 11    000008                  END
```

Using local and global symbols

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

```
            NAME        add1
locn        DEFINE      020H
value       EQU         77
            MOV         R1,locn
            MOV         A,value
            ADD         A,R1
            MOV         R1,A
            RET
            ENDMOD

            NAME        add2
value       EQU         77
            MOV         R1,locn
            MOV         A,value
            ADD         A,R1
            MOV         R1,A
            RET
            END
```

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

Using the LIMIT directive

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

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

Conditional assembly directives

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

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

Table 22: Conditional assembly directives

Directive	Description
`IF`	Assembles instructions if a condition is true.

Table 22: Conditional assembly directives (Continued)

SYNTAX

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

PARAMETERS

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

DESCRIPTION

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

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

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

EXAMPLES

The following macro subtracts a constant from the register r.

```
sub  MACRO  r,c
     IF     c=1
     DEC    r
     ELSEIF c=2
     DEC    r
     DEC    r
     ELSE
     XCH    A,r
     SUBB   A,#c
     XCH    A,r
     ENDIF
     ENDM
```

If the argument to the macro is less than 2, it generates DEC instructions to save instruction cycles and code size; otherwise it generates a SUBB instruction.

It could be tested with the following program:

```
main MOV    R6,#7
     sub    R6,2
     MOV    R7,#22
     sub    R7,1
     RET

     END
```

Macro processing directives

These directives allow user macros to be defined.

Directive	Description
ENDM	Ends a macro definition.
ENDMAC	Ends a macro definition.
ENDR	Ends a repeat structure.
EXITM	Exits prematurely from a macro.
LOCAL	Creates symbols local to a macro.
MACRO	Defines a macro.
REPT	Assembles instructions a specified number of times.
REPTC	Repeats and substitutes characters.

Table 23: Macro processing directives

Directive	Description
REPTI	Repeats and substitutes strings.

Table 23: Macro processing directives

SYNTAX

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

PARAMETERS

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

DESCRIPTION

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

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

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

Defining a macro

You define a macro with the statement:

```
macroname MACRO [,arg] [,arg] …
```

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

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

```
errmac  MACRO   text
        CALL    abort
        DB      text,0
        ENDM
```

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

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

The assembler will expand this to:

```
        CALL    abort
        DB      'Disk not ready',0
```

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

The previous example could therefore be written as follows:

```
errmac  MACRO
        CALL    abort
        DB      \1,0
        ENDM
```

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

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

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

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

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

Passing special characters

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

For example:

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

The macro can be called using the macro quote characters:

```
         macld  <R6,#3>
         END
```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 30.

Predefined macro symbols

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

```
        MODULE MAN

do_op   MACRO
        IF _args == 2
         ADD \1,\2
        ELSE
          INC \1
        ENDIF
        ENDM

        RSEG CODE

        do_op A
        do_op A,#1

        END
```

The following listing is generated:

```
     1    000000                  MODULE MAN
     2    000000
    10    000000
    11    000000                  RSEG CODE
    12    000000
    13    000000                  do_op A
    13.1  000000                  IF _args == 2
    13.2  000000                   ADD A,
    13.3  000000                  ELSE
    13.4  000000 04                 INC A
    13.5  000001                  ENDIF
    13.6  000001                  ENDM
    14    000001                  do_op A,#1
    14.1  000001                  IF _args == 2
    14.2  000001 2401              ADD A,#1
    14.3  000003                  ELSE
    14.4  000003                    INC A
    14.5  000003                  ENDIF
```

```
14.6  000003                  ENDM
15    000003
16    000003                  END
```

How macros are processed

There are three distinct phases in the macro process:

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

Repeating statements

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

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

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

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

EXAMPLES

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

Coding 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.

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

```
                NAME            play
                RSEG            XDATA
buffer          DS              256

                RSEG            CODE
play            MOV             DPTR,#LWRD(buffer)
                MOV             R5,#255
loop            MOVX            A,@DPTR
                MOV             P1,A
                INC             DPTR
                DJNZ            R5,loop
                RET
                END
```

The main program calls this routine as follows:

```
doplay          CALL            play
```

For efficiency we can recode this as the following macro:

```
                NAME            play
                PUBLIC          main

                RSEG            XDATA
buffer          DS              256

play            MACRO
                LOCAL           loop
                MOV             DPTR,#LWRD(buffer)
                MOV             R5,#255
loop            MOVX            A,@DPTR
                MOV             P1,A
                INC             DPTR
                DJNZ            R5,loop
                RET
                ENDM

                RSEG            CODE
main:           play
                END
```

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

Using REPTC and REPTI

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

```
          NAME   reptc

          EXTERN plotc

banner    REPTC  chr, "Welcome"
          MOV    R6,’chr’
          CALL   plotc
          ENDR

          END
```

This produces the following code:

```
1    000000                   NAME      reptc
2    000000
3    000000                   EXTERN    plotc
4    000000          banner   REPTC     chr,"Welcome"
5    000000                   MOV       R6,'chr'
6    000000                   CALL      plotc
7    000000                   ENDR
7.1  000000 AE57              MOV       R6,'W'
7.2  000002 12....            CALL      plotc
7.3  000005 AE65              MOV       R6,'e'
7.4  000007 12....            CALL      plotc
7.5  00000A AE6C              MOV       R6,'l'
7.6  00000C 12....            CALL      plotc
7.7  00000F AE63              MOV       R6,'c'
7.8  000011 12....            CALL      plotc
7.9  000014 AE6F              MOV       R6,'o'
7.10 000016 12....            CALL      plotc
7.11 000019 AE6D              MOV       R6,'m'
7.12 00001B 12....            CALL      plotc
7.13 00001E AE65              MOV       R6,'e'
7.14 000020 12....            CALL      plotc
8    000023
9    000023                   END
```

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

```
          NAME   repti

          EXTERN base, count, init, func
```

```
banner   REPTI  adds, base, count, init
         MOV    R0,LOW(adds)
         MOV    R1,HIGH(adds)
         CALL   func
         ENDR

         END
```

This produces the following code:

```
  1    000000                  NAME    repti
  2    000000
  3    000000                  EXTERN  base,count,init,func
  4    000000
  5    000000          banner  REPTI   adds,base,count,init
  6    000000                  MOV     R0,LOW(adds)
  7    000000                  MOV     R1,HIGH(adds)
  8    000000                  CALL    func
  9    000000                  ENDR
  9.1  000000 A8..             MOV     R0,LOW(base)
  9.2  000002 A9..             MOV     R1,HIGH(base)
  9.3  000004 12....           CALL    func
  9.4  000007 A8..             MOV     R0,LOW(count)
  9.5  000009 A9..             MOV     R1,HIGH(count)
  9.6  00000B 12....           CALL    func
  9.7  00000E A8..             MOV     R0,LOW(init)
  9.8  000010 A9..             MOV     R1,HIGH(init)
  9.9  000012 12....           CALL    func
 10    000015
 11    000015                  END
```

Listing control directives

These directives provide control over the assembler list file.

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

Table 24: Listing control directives

Directive	Description
LSTREP	Controls the listing of lines generated by repeat directives.
LSTXRF	Generates a cross-reference table.
PAGE	Generates a new page.
PAGSIZ	Sets the number of lines per page.

Table 24: Listing control directives (Continued)

SYNTAX

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

PARAMETERS

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

DESCRIPTION

Turning the listing on or off

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

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

Listing conditional code and strings

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

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

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

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

Controlling the listing of macros

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

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

Controlling the listing of generated lines

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

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

Generating a cross-reference table

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

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

Specifying the list file format

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

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

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

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

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

EXAMPLES

Turning the listing on or off

To disable the listing of a debugged section of program:

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

Listing conditional code and strings

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

```
        NAME    lstcndtst
        EXTERN  print

        RSEG    prom

debug   SET     0
begin   IF      debug
        CALL    print
        ENDIF

        LSTCND+
begin2  IF      debug
        CALL    print
        ENDIF

        END
```

This will generate the following listing:

```
     1    00000000                    NAME    lstcndtst
     2    00000000                    EXTERN  print
     3    00000000
     4    00000000                    RSEG    CODE
     5    00000000
     6    00000000            debug   SET     0
     7    00000000            begin   IF      debug
     8    00000000                    CALL    print
     9    00000000                    ENDIF
    10    00000000
    11    00000000                    LSTCND+
    12    00000000            begin2  IF      debug
    14    00000000                    ENDIF
    15    00000000
    16    00000000                    END
```

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

```
     1    000000                  NAME    lstcodtst
     2    000000 0001000A         DW      1,10,100,100,10000
     3    00000A
     4    00000A                  LSTCOD+
     5    00000A 0001000A         DW      1,10,100,1000,10000
```

```
                006403E8
                2710
      6    000014                  END
```

Controlling the listing of macros

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

```
dec2    MACRO  arg
        DEC    arg
        DEC    arg
        ENDM

        LSTMAC+
inc2    MACRO  arg
        INC    arg
        INC    arg
        ENDM

begin:
        dec2   R6

        LSTEXP-
        inc2   R7
        RET
        END    begin
```

This will produce the following output:

```
      5    000000
      6    000000                  LSTMAC+
      7    000000          inc2    MACRO   arg
      8    000000                  INC     arg
      9    000000                  INC     arg
     10    000000                  ENDM
     11    000000
     12    000000          begin:
     13    000000                  dec2    R6
     13.1  000000 1E               DEC     R6
     13.2  000001 1E               DEC     R6
     13.3  000002                  ENDM
     14    000002
     15    000002                  LSTEXP-
     16    000002                  inc2    R7
     17    000004 22               RET
     18    000005                  END     begin
```

Formatting listed output

The following example formats the output into pages of 66 lines each with 132 columns. The LSTPAG directive organizes the listing into pages, starting each module on a new page. The PAGE directive inserts additional page breaks.

```
PAGSIZ 66  ; Page size
COL 132
LSTPAG+
...
ENDMOD
MODULE
...
PAGE
...
```

C-style preprocessor directives

The following C-language preprocessor directives are available:

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

Table 25: C-style preprocessor directives

SYNTAX

```
#define label text
#elif condition
#else
#endif
#error "message"
#if condition
```

```
#ifdef label
#ifndef label
#include {"filename" | <filename>}
#message "message"
#undef label
```

PARAMETERS

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

DESCRIPTION

Defining and undefining labels

Use `#define` to define a temporary label.

```
#define label value
```

is similar to:

```
label SET value
```

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

Conditional directives

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

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

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

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

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

Including source files

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

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

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

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

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

Displaying errors

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

Defining comments

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

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

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

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

```
#define five 5          ; comment

        MOV     five+addr,R7    ; syntax error!
                ; Expands to "5          ; comment+addr,R7"
```

EXAMPLES

Using conditional directives

The following example defines the labels tweak and adjust. If adjust is defined, then register R6 is decremented by an amount that depends on adjust, in this case 30.

```
#define tweak 1
#define adjust 3

#ifdef  tweak
        MOV     A,R6
        CLR     C
#if     adjust=1
        SUBB    A,#4
#elif   adjust=2
        SUBB    A,#20
#elif   adjust=3
        SUBB    A,#30
#endif
        MOV     R6,A
#endif                          /* ifdef tweak */
```

Including a source file

The following example uses #include to include a file defining macros into the source file. For example, the following macros could be defined in Macros.s51:

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

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

```
        NAME   include

; standard macro definitions
#include "macros.s51"
```

```
; program
main:   xch     DPL,DPH
        RET
        END main
```

Data definition or allocation directives

These directives define values or reserve memory:

Directive	Description	Expression restrictions
DB	Generates 8-bit byte constants, including strings.	
DC8	Generates 8-bit byte constants, including strings.	
DC16	Generates 16-bit word constants.	
DC24	Generates 24-bit constants.	
DC32	Generates 32-bit constants.	
DD	Generates 32-bit double word constants.	
DS	Allocates space for 8-bit values.	No external references Absolute
DS8	Allocates space for 8-bit integers.	No external references Absolute
DS16	Allocates space for 16-bit integers.	No external references Absolute
DS24	Allocates space for 24-bit integers.	No external references Absolute
DS32	Allocates space for 32-bit integers.	No external references Absolute
DT	Generates 24-bit word constants.	
DW	Generates 16-bit word constants.	

Table 26: Data definition or allocation directives

SYNTAX

```
DB expr1[,expr1] ...
DC8 expr1 [,expr1] ...
DC16 expr1 [,expr1] ...
DC24 expr1 [,expr1] ...
DC32 expr1 [,expr1] ...
DD expr1[,expr1] ...
DS expr2
```

```
DS8 expr2
DS16 expr2
DS24 expr2
DS32 expr2
DT expr1[,expr1] ...
DW expr1[,expr1] ...
```

PARAMETERS

expr1	A valid absolute, relocatable, or external expression, or an ASCII string. ASCII strings will be zero filled to a multiple of the data size implied by the directive. Double-quoted strings will be zero-terminated.
expr2	A constant value that specifies the number of data blocks of a given size to be created.

DESCRIPTIONS

Use DB, DC8, DC16, DC24, DC32, DD, DP, or DW to reserve and initialize memory space.

Use DS, DS8, DS16, DS24, or DS32 to reserve uninitialized memory space.

EXAMPLES

Generating lookup table

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

```
        NAME   table

table   DB     addsubr,subsubr,clrsubr

addsubr ADD    A,R7
        RET

subsubr SUBB   A,R7
        RET

clrsubr CLR    A
        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 `0xA` bytes:

```
table   DS8   0xA
```

Assembler control directives

These directives provide control over the operation of the assembler.

Directive	Description
`$`	Includes a file.
`/*comment*/`	C-style comment delimiter.
`//`	C++ style comment delimiter.
`CASEOFF`	Disables case sensitivity.
`CASEON`	Enables case sensitivity.
`RADIX`	Sets the default base on all numeric values.

Table 27: Assembler control directives

SYNTAX

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

PARAMETERS

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

DESCRIPTION

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

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

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

Use `RADIX` to set the default base for 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.

EXAMPLES

Including a source file

The following example uses `$` to include a file defining macros into the source file. For example, the following macros could be defined in `Mymacros.s51`:

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

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

```
        NAME    include

; standard macro definitions

$mymacros.s51

; program
main
        xch     DPL,DPH
        RET
        END   main
```

Defining comments

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

```
/*
Program to read serial input.
Version 6: 19.6.03
Author: mjp
*/
```

Changing the base

To set the default base to 16:

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

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

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

```
RADIX   0x0A
```

Controlling case sensitivity

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

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

The following will generate a duplicate label error:

```
        CASEOFF

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

        END
```

Function directives

The function directives are generated by the IAR C/C++ Compiler for 8051 to pass information about functions and function calls to the IAR XLINK Linker. These directives can be seen if you create an assembler list file by using the compiler option **Output assembler file>Include compiler runtime information** (`-lA`).

Note: These directives are primarily intended to support static overlay, a feature which is useful in smaller microcontrollers.

SYNTAX

```
ARGFRAME segment, size, type
FUNCALL caller, callee
FUNCTION label,value
LOCFRAME segment, size, type
```

PARAMETERS

callee	The called function.
caller	The caller to a function.
label	A label to be declared as function.
segment	The segment in which argument frame or local frame is to be stored.
size	The size of the argument frame or the local frame.
type	The type of argument or local frame; either `STACK` or `STATIC`.
value	Function information.

DESCRIPTIONS

`FUNCTION` declares the *`label`* name to be a function. *`value`* encodes extra information about the function.

`FUNCALL` declares that the function *`caller`* calls the function *`callee`*. *`callee`* can be omitted to indicate an indirect function call.

`ARGFRAME` and `LOCFRAME` declare how much space the frame of the function uses in different memories. `ARGFRAME` declares the space used for the arguments to the function, `LOCFRAME` the space for locals. *`segment`* is the segment in which the space resides. *`size`* is the number of bytes used. *`type`* is either `STACK` or `STATIC`, for stack-based allocation and static overlay allocation, respectively.

`ARGFRAME` and `LOCFRAME` always occur immediately after a `FUNCTION` or `FUNCALL` directive.

After a `FUNCTION` directive for an external function, there can only be `ARGFRAME` directives, which indicate the maximum argument frame usage of any call to that function. After a `FUNCTION` directive for a defined function, there can be both `ARGFRAME` and `LOCFRAME` directives.

After a `FUNCALL` directive, there will first be `LOCFRAME` directives declaring frame usage in the calling function at the point of call, and then `ARGFRAME` directives declaring argument frame usage of the called function.

Call frame information directives

These directives allow backtrace information to be defined in the assembler source code.

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

Table 28: Call frame information directives

SYNTAX

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

Names block directives

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

Extended names block directives

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

Common block directives

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

Extended common block directives

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

Data block directives

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

PARAMETERS

bits The size of the resource in bits.

cell The name of a frame cell.

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

cfiexpr A CFI expression (see *CFI expressions*, page 102).

codealignfactor The smallest 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 will shrink the produced backtrace information in size. The possible range is 1–256.

commonblock The name of a previously defined common block.

constant A constant value or an assembler expression that can be evaluated to a constant value.

dataalignfactor The smallest factor of all frame sizes. If the stack grows towards higher addresses, the factor is negative; if it grows towards lower addresses, the factor is positive. 1 is the default, but a larger value will shrink the produced backtrace information in size. The possible ranges are -256 – -1 and 1 – 256.

label A function label.

name The name of the block.

namesblock The name of a previously defined names block.

offset The offset relative the CFA. An integer with an optional sign.

part A part of a composite resource. The name of a previously declared resource.

resource The name of a resource.

segment The name of a segment.

size The size of the frame cell in bytes.

type The memory type, such as `CODE`, `CONST` or `DATA`. In addition, any of the memory types supported by the IAR XLINK Linker. It is used solely for the purpose of denoting an address space.

DESCRIPTIONS

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

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

Backtrace rows and columns

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

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

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

Defining a names block

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

Start and end a names block with the directives:

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

where *name* is the name of the block.

Only one names block can be open at a time.

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

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

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

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

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

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

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

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

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

  ```
  CFI STACKFRAME cfa resource type
  ```

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

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

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

  ```
  CFI STATICOVERLAYFRAME cfa segment
  ```

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

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

  ```
  CFI BASEADDRESS cfa type
  ```

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

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

Extending a names block

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

Extend an existing names block with the directive:

CFI NAMES *name* EXTENDS *namesblock*

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

CFI ENDNAMES *name*

Defining a common block

The *common block* is used 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 segment type. You have to declare the return address column for the common block.

End a common block with the directive:

CFI ENDCOMMON *name*

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

Inside a common block you can declare the initial value of a CFA or a resource by using the directives listed last in *Common block directives*, page 94. For more information on these directives, see *Simple rules*, page 100, and *CFI expressions*, page 102.

Extending a common block

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

Extend an existing common block with the directive:

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

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

```
CFI ENDCOMMON name
```

Defining a data block

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

Start a data block with the directive:

```
CFI BLOCK name USING commonblock
```

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

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

```
CFI FUNCTION label
```

where *label* is the code label starting the function.

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

```
CFI NOFUNCTION
```

End a data block with the directive:

```
CFI ENDBLOCK name
```

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

Inside a data block you may manipulate the values of the columns by using the directives listed last in *Data block directives*, page 94. For more information on these directives, see *Simple rules*, page 100, and *CFI expressions*, page 102.

SIMPLE RULES

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

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

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

In those rare cases where the descriptive power of the simple rules are not enough, a full CFI expression can be used to describe the information (see *CFI expressions*, page 102). However, whenever possible, you should always use a simple rule instead of a CFI expression.

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

Simple rules for resources

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

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

```
CFI REG SAMEVALUE
```

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

```
CFI REG UNDEFINED
```

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

```
CFI REG1 REG2
```

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

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

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

```
CFI RET CONCAT
```

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

Simple rules for CFAs

In contrast with the rules for resources, the rules for CFAs describe the address of the beginning of the call frame. The call frame often includes the return address pushed by the subroutine calling instruction. The CFA rules describe how to compute the address to the beginning of the current call frame. There are two different forms of CFAs, stack frames and static overlay frames, each declared in the associated names block. See *Names block directives*, page 94.

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

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

```
CFI CFA_SP NOTUSED
```

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

```
CFI CFA_SP SP + 4
```

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

CFI EXPRESSIONS

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

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

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

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

Unary operators

Overall syntax: *OPERATOR*(*operand*)

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

Table 29: Unary operators in CFI expressions

Binary operators

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

Operator	**Operands**	**Description**
ADD	*cfiexpr*,*cfiexpr*	Addition
SUB	*cfiexpr*,*cfiexpr*	Subtraction
MUL	*cfiexpr*,*cfiexpr*	Multiplication
DIV	*cfiexpr*,*cfiexpr*	Division
MOD	*cfiexpr*,*cfiexpr*	Modulo
AND	*cfiexpr*,*cfiexpr*	Bitwise AND
OR	*cfiexpr*,*cfiexpr*	Bitwise OR

Table 30: Binary operators in CFI expressions

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

Table 30: Binary operators in CFI expressions (Continued)

Ternary operators

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

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

Table 31: Ternary operators in CFI expressions

Operator	Operands	Description
`LOAD`	*`size,type,addr`*	Get value from memory. The operands are: sizeA constant expression denoting a size in bytes. typeA memory type. addrA CFA expression denoting a memory address. Gets the value at address *`addr`* in segment type *`type`* of size *`size`*.

Table 31: Ternary operators in CFI expressions (Continued)

EXAMPLE

The following is a generic example and not an example specific to the 8051 microcontroller. This will simplify the example and clarify the usage of the CFI directives. A target-specific example can be obtained by generating assembler output when compiling a C source file.

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

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

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

Table 32: Code sample with backtrace rows and columns

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

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

Defining the names block

The names block for the small example above would be:

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

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

Defining the common block

The common block for the simple example above would be:

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

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

Defining the data block

Continuing the simple example, the data block would be:

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

```
    CFI    CFA SP + 2
    MOV    R1,R0
    CFI    R1 SAMEVALUE
    RET
    CFI ENDBLOCK func1block
```

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

Assembler diagnostics

This chapter lists the error and warning messages for the 8051 IAR Assembler.

Severity levels

The diagnostic messages produced by the 8051 IAR Assembler reflect problems or errors that are found in the source code or occur at assembly time.

ASSEMBLY WARNING MESSAGES

Assembly warning messages are produced when the assembler has found a construct which is probably the result of a programming error or omission. These messages are listed in the section *Warning messages*, page 116.

COMMAND LINE ERROR MESSAGES

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

ASSEMBLY ERROR MESSAGES

Assembly error messages are produced when the assembler has found a construct which violates the language rules. These messages are listed in the section *Error messages*, page 108.

ASSEMBLY FATAL ERROR MESSAGES

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

ASSEMBLER INTERNAL ERROR MESSAGES

During assembly a number of internal consistency checks are performed and if any of these checks fail, the assembler will terminate after giving a short description of the problem. Such errors should normally not occur. However, if you should encounter an error of this type, please report it to your software distributor or to IAR Technical Support. Please include information enough to reproduce the problem.

This would typically include:

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

Error messages

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

All errors are issued as complete, self-explanatory messages. The error message consists of the 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 will be preceded by the source line number and the name of the *current* file:

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

GENERAL ERROR MESSAGES

The following section lists the general error messages.

0 **Invalid syntax**
The assembler could not decode the expression.

1 **Too deep #include nesting (max. is 10)**
The assembler limit for nesting of `#include` files was exceeded. A recursive `#include` could be the reason.

2 **Failed to open #include file name**
Could not open a `#include` file. The file does not exist in the specified directories. Check the `-I` prefixes.

3 **Invalid #include file name**
A `#include` file name must be written `<file>` or `"file"`.

4 **Unexpected end of file encountered**
End of file encountered within a conditional assembly, the repeat directive, or during macro expansion. The probable cause is a missing end of conditional assembly etc.

5 **Too long source line (max. is 2048 characters) truncated**
The source line length exceeds the assembler limit.

6 **Bad constant**
A character that is not a legal digit was encountered.

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

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

9 **Too many errors encountered (>100).**

10 **Space or tab expected**

11 **Too deep block nesting (max is 50)**
The preprocessor directives are nested too deep.

12 **String too long (max is 2045)**
The assembler string length limit was exceeded.

13 **Missing delimiter in literal or character constant**
No closing delimiter ' or " was found in character or literal constant.

14 **Missing #endif**
A #if, #ifdef, or #ifndef was found but had no matching #endif.

15 **Invalid character encountered: char; ignored**

16 **Identifier expected**
A name of a label or symbol was expected.

17 **')' expected**

18 **No such pre-processor command: command**
# was followed by an unknown identifier.

19 **Unexpected token found in pre-processor line**
The preprocessor line was not empty after the argument part was read.

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

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

22 **Macro parameter *parameter* redefined**
A #define symbol's formal parameter was repeated.

23 **',' or ')' expected**

24 **Unmatched #else, #endif or #elif**
Fatal. Missing #if, #ifdef, or #ifndef.

25 ***#error*** **error**
Printout via the `#error` directive.

26 **'(' expected**

27 **Too many active macro parameters (max is 256)**
Fatal. Preprocessor limit exceeded.

28 **Too many nested parameterized macros (max is 50)**
Fatal. Preprocessor limit exceeded.

29 **Too deep macro nesting (max is 100)**
Fatal. Preprocessor limit exceeded.

30 **Actual macro parameter too long (max is 512)**
A single macro (in `#define`) argument may not exceed the length of a source line.

31 **Macro macro called with too many parameters**
The number of parameters used was greater than the number in the macro declaration.

32 **Macro macro called with too few parameters**
The number of parameters used was less than the number in the macro declaration (`#define`).

33 **Too many MACRO arguments**
The number of assembler macros exceeds 32.

34 **May not be redefined**
Assembler macros may not be redefined.

35 **No name on macro**
An assembler macro definition without a label was encountered.

36 **Illegal formal parameter in macro**
A parameter that was not an identifier was found.

37 **ENDM or EXITM not in macro**
An `ENDM` directive or `EXITM` directive encountered outside a macro.

38 **'>' expected but found end-of-line**
A < was found but no matching >.

39 **END before start of module**
The end-of-module directive has no matching `MODULE` directive.

40 **Bad instruction**
The mnemonic/directive does not exist.

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

42 **Duplicate label**
The label has already appeared in the label field or has been declared as `EXTERN`.

43 **Illegal effective address**
The addressing mode (operands) is not allowed for this mnemonic.

44 **',' expected**
A comma was expected but not found.

45 **Name duplicated**
The name of `RSEG`, `STACK`, or `COMMON` segments is already used but for something else.

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

47 **Segment name expected**
The `RSEG`, `STACK`, and `COMMON` directives need a name.

48 **Value out of *`range`* range**
The value exceeds its limits.

49 **Alignment already set**
`RSEG`, `STACK`, and `COMMON` segments do not allow alignment to be set more than once. Use `ALIGN`, `EVEN`, or `ODD` instead.

50 **Undefined symbol: symbol**
The symbol did not appear in label field or in an `EXTERN` or `sfr` declaration.

51 **Can't be both PUBLIC and EXTERN**
Symbols can be declared as either `PUBLIC` or `EXTERN`.

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

53 **Expression must be absolute**
The expression cannot involve relocatable or external symbols.

54 **Expression can not be forward**
The assembler must be able to solve the expression the first time this expression is encountered.

55 **Illegal size**
The maximum size for expressions is 32 bits.

56 **Too many digits**
The value exceeds the size of the destination.

57 **Unbalanced conditional assembly directives**
Missing conditional assembly `IF` or `ENDIF`.

58 **ELSE without IF**
Missing conditional assembly `IF`.

59 **ENDIF without IF**
Missing conditional assembly `IF`.

60 **Unbalanced structured assembly directives**
Missing structured assembly `IF` or `ENDIF`.

61 **'+' or '-' expected**
A plus or minus sign is missing.

62 **Illegal operation on extern or public symbol**
An illegal operation has been used on a public or external symbol, e.g. `VAR`.

63 **Illegal operation on non-constant label**
It is illegal to make a non-constant symbol `PUBLIC` or `EXTERN`.

64 **Extern or unsolved expression**
The expression must be solved at assembly time, i.e. not include external references.

65 **'=' expected**
Equals sign was missing.

66 **Segment too long (max is max)**
The length of `ASEG`, `RSEG`, `STACK`, or `COMMON` segments is larger than the addressable length.

67 **Public did not appear in label field**
A symbol was declared `PUBLIC` but no label with the same name was found in the source file.

68 **End of block-repeat without start**
The repeat directive `REPT` was not found although the `ENDR` directive was.

69 **Segment must be relocatable**
The operation is not allowed on `ASEG`.

70 **Limit exceeded: error text, value is: value (decimal)**
The value exceeded the limits set with the LIMIT directive. The error text is set by the user in the LIMIT directive.

71 **Symbol symbol has already been declared EXTERN**
An attempt to redeclare an EXTERN as EXTERN was made.

72 **Symbol symbol has already been declared PUBLIC**
An attempt to redeclare a PUBLIC as PUBLIC was made.

73 **End-of-module missing**
A PROGRAM or MODULE directive was encountered before ENDMOD was found.

74 **Expression must yield non-negative result**
The expression was evaluated to a negative number, whereas a positive number was required.

75 **Repeat directive unbalanced**
This error is caused by a REPT directive without a matching ENDR, or a an ENDR directive without a matching REPT.

76 **End of repeat directive is missing**
A REPT directive without a closing ENDR was encountered.

77 **LOCALs not allowed in this context, (symbol)**
Local symbols must be declared within macro definitions.

78 **End of macro expected**
An assembler macro is being defined but there was no end-of-macro.

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

80 **End of conditional assembly expected**
Conditional assembly is active but there was no end of if.

81 **End of structured assembly expected**
One of the directives for structured assembly is active but has no matching END.

82 **Misplaced end of structured assembly**
A directive that terminates one of the structured assembly directives was found but no matching START directive is active.

83 **Error in SFR attribute definition**
The SFRTYPE directive was used with unknown attributes.

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

85 **Wrong number of arguments**
Expected a different number of arguments.

86 **Number expected**
Characters other than digits were encountered.

87 **Label must be public or extern**
The label must be declared with `PUBLIC` or `EXTERN`.

88 **Label not defined with DEFFN**
The label has to be defined via `DEFFN` before used in this context.

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

90 **Different parts of ASEG have overlapping code**

91 **Internal error**

92 **Empty macro stack overflow**

93 **Macro stack overflow**

94 **Attempt to access out-of-stack value**

95 **Invalid macro operator**

96 **No such macro argument**

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

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

99 **#include within macro not supported**

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

8051-SPECIFIC ERROR MESSAGES

In addition to the general error messages, the 8051 IAR Assembler may generate the following error messages:

401 **Too many operands**

402 **:8 or :16 expected**

403 **There is no error message with this number**

404 **The register name is not allowed here**

405 **There is no error message with this number**

406 **Illegal suffix**

407 **Illegal value value**

408 **Illegal size specifier specifier**

409 **C-comment has no end**

410 **Could not solve step**

411 **Nothing to BREAK out of**

412 **CASE after DEFAULT**
`DEFAULT` is a catch-all case and is not allowed to have a `CASE` after it.

413 **CASE outside SWITCH**

414 **COMMA expected**

415 **Nothing to CONTINUE to**
`CONTINUE` needs something to continue.

416 **Cannot solve break**
The break count must be solvable.count value

417 **DEFAULT outside SWITCH**

418 **ELSE used more than once**
It is not allowed to have multiple `ELSE` directives for an `IF`.

419 **ELSE without matching IF**

420 **ELSEIF cannot be used after ELSE**

421 **ELSEIF with no matching IF**

422 **ENDF without matching FOR**

423 **ENDIF without matching IF**

424 **ENDS without matching SWITCH**

425 **ENDW without matching WHILE**

426 **THEN without matching IF**

427 **Negative step value**

428 **Zero step value**

429 **UNTIL without matching REPEAT**

430 **Break argument must be 1,2, or 3**

431 **Multiple DEFAULT**
It is not allowed to have more than one `DEFAULT` inside a `SWITCH`.

432 **Can't assign register to register**

433 **Illegal constant prefix specifier**

434 **Illegal prefix specifier**

435 **Illegal bit suffix specifier**

Warning messages

GENERAL

The following section lists the general warning messages.

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

1 **Nested comment**
A C-type comment, /* ... */, was nested.

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

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

4 **Macro or define expected**

5 **Floating point value out-of-range**
Floating point value is too large to be represented by the floating-point system of the target.

6 **Floating point division by zero**

7 **Wrong usage of string operator ('#' or '##'); ignored.**
The current implementation restricts usage of the `#` and `##` operators to the token field of parameterized macros. In addition, the `#` operator must precede a formal parameter.

8 **Macro parameter(s) not used**

9 **Macro redefined**

10 **Unknown macro**

11 **Empty macro argument**

12 **Recursive macro**

13 **Redefinition of Special Function Register**
The special function register (`SFR`) has already been defined.

14 **Division by zero**
Division by 0 in constant expression.

15 **Constant truncated**
The constant was longer than the size of the destination.

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

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

18 **End of program while in include file**
The program ended while a file was being included.

19 **Symbol symbol duplicated**

20 **Bit symbol cannot be used as operand**
A symbol was declared using the bit directive, but since the bit address is not calculated the symbol should not be used.

21 **Label did not appear in label field**

22 **Set segment alignment the same value or larger**
When the alignment set by `ALIGN` is larger than the segment alignment it may be lost at link time.

8051-SPECIFIC WARNING MESSAGES

In addition to the general warning messages, the 8051 IAR Assembler may generate the following warning messages:

400 **Number out of range**
The value does not fit the instruction/directive and is truncated.

401 **SFR neither defined as READ nor WRITE**
The `SFRTYPE` directive was used in such a way that the special function register is inaccessible.

402 **More than one SFR size attribute defined using default (byte)**
The `SFRTYPE` directive was used with multiple size definitions. The assembler will use default byte size.

403 No SFR size attribute defined using default (byte)

The SFRTYPE directive was used with no size definition. The assembler will use default byte size.

404 Displacement out of bounds

The offset in a JMP or CALL instruction does not fit, the destination label is to far off.

405 Accessing SFR incorrectly, check read/write flags

An attempt such as to write to a read-only SFR has been made.

406 Accessing SFR using incorrect size

An attempt such as to write to a read-only SFR has been made.

407 Address may not be reachable

408 SFR address might not be bit addressable

409 Bit address used as regular dir8 address