8051 IAR Assembler

Reference Guide

for the

8051 Family of Microcontrollers

COPYRIGHT NOTICE

DISCLAIMER

TRADEMARKS

EDITION NOTICE

This edition replaces previous editions of this guide.

Sixth edition: January 2001

Part number: A8051-6

Contents

Tables

Preface

Welcome to the 8051 IAR Assembler 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 best suit your application requirements.

Who should read this guide

You should read this guide if you plan to develop embedded applications or modules for the 8051 microcontroller using assembly language. In addition, you should have working knowledge of the following:

- General assembly language programming
- The architecture and instruction set of the 8051 microcontroller (refer to the chip manufacturer's documentation for information about assembler instructions, 8051 architecture, and instruction set)
- Windows 95/98/2000 or Windows NT, depending on your operating system

How to use this guide

If you are new to using the IAR toolkit, we recommend that you read the initial chapters of the *IAR Embedded Workbench™ User Guide*. It includes comprehensive information about the installation of all IAR tools and product overviews, as well as tutorials that can help you get started.

If you are an intermediate or advanced user, take advantage of the reference information in all of the chapters, which provide details about, for example, options, environments, and diagnostics.

What this guide contains

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

- *Introduction to the 8051 IAR Assembler* describes the formats of the source code and assembler listings and provides programming hints for the 8051 IAR Assembler.
- *Assembler options* explains how to set assembler options from the command line and provides the syntax and a description of each of the options.
- *Assembler operators* describes the precedence and provides a summary and examples of assembler operators.

- *Assembler directives* gives an alphabetical summary of the assembler directives, describes the syntax conventions, and provides detailed reference information about directives according to usage.
- *Assembler diagnostics* lists the error and warning messages specific to the 8051 IAR Assembler.

Document conventions

This guide uses the following typographic conventions:

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

Table 1: Typographical conventions used in this guide

Introduction to the 8051 IAR Assembler

This chapter describes the source code format for the 8051 IAR Assembler. It provides programming hints for the assembler and shows the format of assembler list files.

Refer to 8051 hardware documentation for syntax descriptions of the instruction mnemonics.

Source format

The format of an assembler source line is as follows:

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

where the components are as follows:

label	A label, which is assigned the value and type of the current program location counter (PLC). The `:` (colon) is optional if the label starts in the first column.
operation	An assembler instruction or directive. This must not start in the first column.
operands	An assembler instruction can have zero, one, or two operands that are 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.

Assembler expressions

Expressions can consist of operands and operators.

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

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators. For more information, see *Precedence of operators*, page 25.

The following operands are valid in an expression:

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

These are described in greater detail in the following sections.

The valid operators are described in the chapter *Assembler operators*, page 25.

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
       EXTERN    third
       RSEG      DATA
first  DB        5
second DB        3
       ENDMOD

       MODULE    prog2
       EXTERN    first
       EXTERN    second
       EXTERN    third
       RSEG      CODE
       MOV  R7,first
       MOV  R7,first+1
       MOV  R7,1+first
       MOV  R7,(first/second)*third
```

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

SYMBOLS

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

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

For built-in symbols like instructions, registers, operators, and directives case is insignificant. For user-defined symbols case is by default significant but can be turned on and off using the **Case sensitive user symbols** (`-s`) assembler option. See page 21 for additional information.

LABELS

Symbols used for memory locations are referred to as labels.

Program location counter (PLC)

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

```
        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 2: Integer constant formats

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

ASCII CHARACTER CONSTANTS

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

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

Table 3: ASCII character constant formats

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.

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 with the following contents:: Bit `0-7` Always 0. Bit `8-14` Target Id, which is `14(0EH)` for 8051 Bit `15` Intrinsic support
__TIME__	Current time in `hh:mm:ss` format (string).
__VER__	Version number in integer format; for example, version 4.17 is returned as 417 (number).

Table 4: Predefined symbols

Notice that `__TID__` is related to the predefined symbol `__TID__` in the 8051 IAR Compiler. It is described in the chapter *Predefined symbols reference* in the *8051 IAR C Compiler Reference Guide*.

Including symbol values in code

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

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

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

      RSEG    CODE
      EXTERN  printstring
main
      MOV      R4,td        ; load address of string
      LCALL    printstring  ; routine to print string
      RET
```

Testing symbols for conditional assembly

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

For example, you may want to assemble appropriate code for a specific processor such as the 8051 microprocessor. You could do this using the `__TID__` symbol as follows:

```
#define TARGET ((__TID__& 0x0F00)>>8)
#if (TARGET==0x0E)
...
#else
...
#endif
```

Register symbols

Definitions of the symbols for registers—including standard SFRs—for the different processor variants, are supplied in the `sfrnnn.inc` files in the `\inc` directory.

Programming hints

This section gives hints on how to write efficient code for the 8051 IAR Assembler.

SPECIAL FUNCTION REGISTERS

Specific header files for a number of 8051 derivatives are included in the IAR product package. The header files are named `sfrnnn.inc`, for example `sfr515a.inc`, and define the processor-specific special function registers (SFRs).

Since the 8051 IAR Assembler has predefined SFR declarations, you should not declare those SFRs for the application program. For information about which SFRs are predefined, see the `A8051.htm` file.

The header files are also suitable to use as templates when creating new header files for other 8051 derivatives.

Example

The SFR timer 2 controll register `T2CON` is located at address `0xC8`. The definition for this is:

```
sfr  T2CON = 0xC8
```

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.

List file format

This section shows how the assembly code is represented in the assembler list file. The following code example is used:

```
     NAME dio

; define the ports
     ASEG DATA

; define the macros
strobe MACRO
     MOV A,P1
     ORL 1,#128
     MOV P1,A
     ANL 1,#127
     MOV P1,A
     ENDM

outdat MACRO val
     MOV P3,val
     ENDM

; vector table
     ASEG CODE
     ORG 0
     SJMP main ; Reset vector

; main code
     ORG 0x001C
main
     outdat #23
     strobe
     outdat #40
     strobe
done
     JMP done
     END
```

The following section shows the format of the 8051 IAR Assembler list file.

HEADER

The header section shows the selected command line options:

```
###########################################################################
  IAR Systems 8051 Assembler VN.nnx/XXX nn/Mmm/yyyy  hh:mm:ss
  Copyright 1999 IAR Systems. All rights reserved.

  Source file   =  c:\iar\ew23\8051\tutor\dio.s03
  List file     =  c:\iar\ew23\8051\projects\debug\list\dio.lst
  Object file   =  c:\iar\ew23\8051\projects\debug\obj\dio.r03
  Command line  =  -s+ -M<> -w+
                   -LC:\IAR\EW23\8051\Projects\Debug\List\ -t8
                   -IC:\IAR\EW23\8051\inc\
                   -OC:\IAR\EW23\8051\Projects\Debug\Obj\ -r
                    C:\IAR\EW23\8051\tutor\dio.s03
###########################################################################
```

BODY

The body of the list file shows the assembler-generated code:

```
 1    000000               NAME dio
 2    000000
 3    000000          ; define the ports
 4    000000               ASEG DATA
 5    000000
 6    000000          ; define the macros
14    000000
18    000000
19    000000          ; vector table
20    000000               ASEG CODE
21    000000               ORG 0
22    000000 801A          SJMP main ; Reset vector
23    000002
24    000002          ; main code
25    00001C               ORG 0x001C
26    00001C          main
27    00001C               outdat #23
27.1  00001C 75B017        MOV P3,#23
27.2  00001F               ENDM
28    00001F               strobe
28.1  00001F E590          MOV A,P1
28.2  000021 430180        ORL 1,#128
28.3  000024 F590          MOV P1,A
28.4  000026 53017F        ANL 1,#127
28.5  000029 F590          MOV P1,A
28.6  00002B               ENDM
29    00002B               outdat #40
29.1  00002B 75B028        MOV P3,#40
29.2  00002E               ENDM
30    00002E               strobe
30.1  00002E E590          MOV A,P1
30.2  000030 430180        ORL 1,#128
30.3  000033 F590          MOV P1,A
30.4  000035 53017F        ANL 1,#127
30.5  000038 F590          MOV P1,A
30.6  00003A               ENDM
31    00003A          done
32    00003A 80FE          JMP done
33    00003C               END
```

Lines generated by macros will, if listed, have a `.` (period) in the source line number field:

```
27.1  00001C 75B017        MOV P3,#23
27.2  00001F               ENDM
```

For information about assembler macros, see *Macro processing directives*, page 58.

CRC

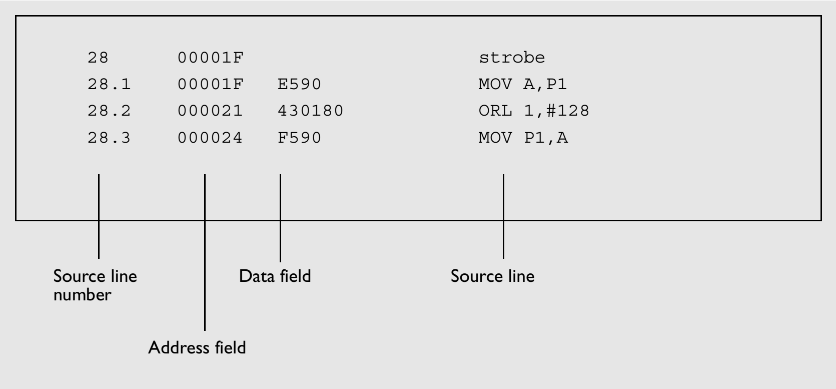
The CRC section contains the assembler report where the CRC checksum value can be used for verifying the integrity of the assembled code:

```
##############################
#          CRC:125B          #
#        Errors:   0         #
#        Warnings: 0         #
#         Bytes: 34          #
##############################
```

LIST FIELDS

The assembly list contains the following fields of information:

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

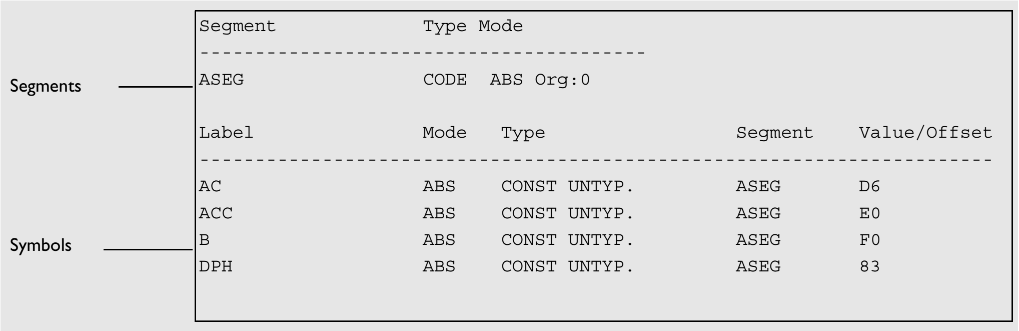


```
28      00001F                    strobe
28.1    00001F   E590             MOV A,P1
28.2    000021   430180           ORL 1,#128
28.3    000024   F590             MOV P1,A
```

Figure 1: Assembler list fields

SYMBOL AND CROSS-REFERENCE TABLE

If the `LSTXRF+` directive has been included, or the option `-x` has been specified, the following symbol and cross-reference table is produced:


Segments
Symbols

```
Segment             Type Mode
----------------------------------------
ASEG                CODE  ABS Org:0

Label               Mode   Type                 Segment    Value/Offset
-----------------------------------------------------------------------
AC                  ABS    CONST UNTYP.         ASEG       D6
ACC                 ABS    CONST UNTYP.         ASEG       E0
B                   ABS    CONST UNTYP.         ASEG       F0
DPH                 ABS    CONST UNTYP.         ASEG       83
```

Figure 2: Symbol and cross-reference table in assembler list file

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

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

Table 5: Symbol and cross-reference table

Output formats

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

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

Assembler options

This chapter explains how to set assembler options from the command line and provides the syntax and a description of each of the options.

Refer to the *IAR Embedded Workbench™ User Guide* for information about the assembler options available in the IAR Embedded Workbench and how to set them.

Setting assembler 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.s03`, 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`, enter:

```
a8051 -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 6: 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 7: Asssembler environment variables

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

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

Summary of assembler options

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

Command line option	**Description**
`-B`	Macro execution information
`-b`	Make a library module
`-c{DSEAOMC}`	Conditional list
`-D`*symb*`[=`*value*`]`	Define symbol
`-d`	Disable `#ifdef/#endif` matching
`-f extend.xcl`	Extend the command line
`-G`	Open standard input as source
`-I`*prefix*	Include paths
`-i`	#included text
`-L[`*prefix*`]`	List to prefixed source name
`-l` *filename*	List to named file
`-M`*ab*	Macro quote characters
`-N`	No header
`-O`*prefix*	Set object filename prefix
`-o` *filename*	Set object filename
`-p`*lines*	Lines/page
`-r[en]`	Generate debug information
`-S`	Set silent operation
`-s{+|-}`	Case sensitive user symbols
`-T`	Active lines only
`-t`*n*	Tab spacing
`-U`*symb*	Undefine symbol
`-u`	Use A8051 V2.xx operators
`-v[0|1|2|3|4|5|6]`	Processor configuration
`-w[`*string*`][s]`	Disable warnings
`-x{DI2}`	Include cross-reference

Table 8: 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 17.

This option is identical to the **Macro execution info** option on the **List** page of 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.

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

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

This option is identical to the **Make a LIBRARY module** option on the **Code generation** page in the **A8051** category in the IAR Embedded Workbench.

-c

`-c{DSEAOMC}`

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 17 for additional information.

The following table shows the available parameters:

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

Table 9: Conditional list (-c)

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

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

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

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

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 **#define** option 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 **Code generation** page in the **A8051** category in the IAR Embedded Workbench.

-f

```
-f extend.xcl
```

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

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

Example

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

```
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` `-Iprefix`

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\` provided that the `A8051_INC` environment variable is set.

This option is related to the **#include** option 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** option in the **A8051** category in the IAR Embedded Workbench.

`-L` `-L[prefix]`

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

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

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

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

Example

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

```
a8051 prog -Llist\
```

This option is related to the **List** options 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 **List** options 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.

This option is identical to the **Macro quote chars** option on the **Code generation** page for 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 17 for additional information.

This option is related to the **List** options 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 must not include a space before the prefix.

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

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

Example

To send the object code to the file `obj\prog.r03` rather than to the default file `prog.r03`:

```
a8051 prog -Oobj\
```

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

`-o` `-o` *`filename`*

This option sets the filename to be used for the object file. Notice that you must include a space before the filename. If no extension is specified, `r03` 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.r03` instead of the default `prog.r03`:

```
a8051 prog -o obj
```

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

This option is related to the filename and directory that you specify when creating a new source file or project in the IAR Embedded Workbench.

-p

`-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 17 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[en]`

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

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

The following table shows the available parameters:

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

Table 10: Generating debug information (-r)

This option is identical to the **Debug** option in the **A8051** category in the IAR Embedded Workbench.

-S

`-S`

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

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

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

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

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

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

Table 11: Controlling case sensitivity in user symbols (-s)

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

This option is identical to the **Case sensitive user symbols** option on the **Code generation** page in the **A8051** category in the IAR Embedded Workbench.

`-T` `-T`

Includes only active lines, 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 17 for additional information.

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

`-U` `-U`*`symb`*

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

By default, the assembler provides certain predefined symbols; see *Predefined symbols*, page 4. The `-U` option allows you to undefine such a predefined symbol to make its name available for your own use through a subsequent `-D` option or source definition.

Example

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

```
a8051 prog -U __TIME__
```

This option is identical to the **#undef** option in the **A8051** category in the IAR Embedded Workbench.

-u

`-u`

Causes the assembler to use the A8051 V2.xx operators.

-w

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

By default, the assembler displays a warning message when it detects an element of the source which is legal in a syntactical sense, but may contain a programming error; see *Assembler diagnostics*, page 87, for details.

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

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

Table 12: Disabling assembler warnings (-w)

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

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

Example

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

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

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

```
a8051 prog -w-0-8
```

This option is identical to the **Warnings** option on the **Code generation** page for the **A8051** category in the IAR Embedded Workbench.

`-x` `-x{DI2}`

Use this option to make the assembler include a cross-reference table at the end of the list file; see the chapter *Introduction to the 8051 IAR Assembler*, for an example.

This option is useful in conjunction with the list options `-L` or `-l`; see page 17 for additional information.

The following parameters are available:

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

Table 13: Including cross-references in assembler list file (-x)

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

Assembler operators

This chapter describes the order of precedence for the assembler operators and defines them. Furthermore, examples and a detailed description are given for each assembler operator.

Precedence of operators

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

The following rules determine how expressions are evaluated:

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

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

Summary of assembler operators

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

UNARY OPERATORS – 1

`+`	Unary plus
`-`	Unary minus
`NOT (!)`	Logical NOT
`LOW`	Low byte
`HIGH`	High byte
`BYTE2`	Second byte
`BYTE3`	Third byte

`LWRD`	Low word
`HWRD`	High word
`DATE`	Current date/time
`SFB`	Segment begin
`SFE`	Segment end
`SIZEOF`	Segment size
`BITNOT (~)`	Bitwise NOT

MULTIPLICATIVE ARITHMETIC AND SHIFT OPERATORS –2

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

ADDITIVE ARITHMETIC OPERATORS – 3

`+`	Addition
`–`	Subtraction

SHIFT OPERATORS – 4

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

AND OPERATORS – 5

`AND (&&)`	Logical AND
`BITAND (&)`	Bitwise AND

OR OPERATORS – 6

`OR (||)`	Logical OR
`XOR`	Logical exclusive OR
`BITOR (|)`	Bitwise OR
`BITXOR (^)`	Bitwise exclusive OR

COMPARISON OPERATORS – 7

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

Descriptions of assembler operators

The following sections give detailed descriptions of each assembler operator. See *Assembler expressions*, page 1, for related information.

`*` 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.

Examples

```
2*2 → 4
-2*2 → -4
```

`+` Unary plus (1).

Unary plus operator.

Examples

```
+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.

Examples

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

`-` Unary minus (1).

The unary minus operator performs arithmetic negation on its operand.

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

`-` Subtraction (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.

Examples

```
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.

Examples

```
9/2 → 4
-12/3 → -4
9/2*6 → 24
```

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

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

Examples

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

`BITAND(&)` Bitwise AND (5).

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

Examples

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

`BITNOT (~)` Bitwise NOT (1).

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

Example

```
BITNOT B’1010 → B’11111111111111111111111111110101
```

`BITOR (|)` Bitwise OR (6).

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

Examples

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

`BITXOR (^)` Bitwise exclusive OR (6).

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

Examples

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

`BYTE2` Second byte (1).

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

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
```

`DATE` Current date/time (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 5, DATE 4, DATE 3
```

`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.

Examples

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

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

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

Examples

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

`GT, >` Greater than (7).

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

Examples

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

`HIGH` Second 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
```

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

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

Examples

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

`LOW` Low byte (1).

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

Example

```
LOW 0xABCD → 0xCD
```

`LT, <` Less than (7).

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

Examples

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

`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
```

`MOD (%)` Modulo (2).

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

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

Examples

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

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

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

Examples

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

`NOT (!)` Logical NOT (1).

Use `NOT` to negate a logical argument.

Examples

```
NOT B’0101 → 0
NOT B’0000 → 1
```

`OR (||)` Logical OR (6).

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

Examples

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

`SFB` Segment begin (1).

Syntax

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

Parameters

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

Description

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

Examples

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

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

`SFE` Segment end (1).

Syntax

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

Parameters

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

Description

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

Examples

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

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

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

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

Examples

```
B’00011100 SHL 3 → B’11100000
B’00000111111111111 SHL 5 → B’11111111111100000
14 SHL 1 → 28
```

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

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

Examples

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

`SIZEOF` Segment size (1).

Syntax

`SIZEOF` *`segment`*

Parameters

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

Description

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

Examples

The following example sets `size` to the size of segment `CODE`.

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

`UGT` Unsigned greater than (7).

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

Examples

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

`ULT` Unsigned less than (7).

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

Examples

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

`XOR` Logical exclusive OR (6).

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

Examples

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

Assembler directives

This chapter gives an alphabetical summary of the assembler directives, describes the syntax conventions, and provides complete reference information about directives for module control, symbol control, segment control, value assignment, conditional assembly, macro processing, listing control, C-style preprocessor, data definition or allocation, and assembler control.

Summary of directives

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

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

Table 14: Assembler directives summary

Directive	Description	Section
`BREAK`	Exits prematurely from a loop or switch construct	Structured assembly
`CASE`	Case in `SWITCH` block.	Structured assembly
`CASEOFF`	Disables case sensitivity.	Assembler control
`CASEON`	Enables case sensitivity.	Assembler control
`COL`	Sets the number of columns per page.	Listing control
`COMMON`	Begins a common segment.	Segment control
`CONTINUE`	Continues execution of a loop or switch construct	Structured assembly
`CYCLEMAX`	Selects the greater of two possible cyclecount values	Listing controls
`CYCLEMEAN`	Selects the mean value	Listing controls
`CYCLEMIN`	Selects the lower of two possible cyclecount values	Listing controls
`CYCLES`	Sets the listed cycle count	Listing control
`DB`	Generates 8-bit byte constants, including strings.	Data definition or allocation
`DC16`	Generates 16-bit word constants, including strings.	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
`DC8`	Generates 8-bit byte constants, including strings.	Data definition or allocation
`DD`	Generates 32-bit long word constants.	Data definition or allocation
`DEFAULT`	Default case in `SWITCH` block	Structured assembly
`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

Table 14: Assembler directives summary (continued)

Directive	Description	Section
`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
`ELSEIF`	Specifies a new condition in an `IF`…`ENDIF` block.	Conditional assembly
`ELSEIFS`	Specifies a new condition in an `IF...ENDIF` block.	Structured assembly
`ELSES`	Specifies instructions to be executed if a condition is false.	Structured assembly
`END`	Terminates the assembly of the last module in a file.	Module control
`ENDF`	Ends a `FOR` loop	Structured assembly
`ENDIF`	Ends an `IF` block.	Conditional assembly
`ENDIFS`	Ends an `IFS` block.	Structured assembly
`ENDM`	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
`ENDS`	Ends a SWITCH block.	Structured assembly
`ENDW`	Ends a WHILE loop.	Structured assembly
`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
`FOR`	Repeats subsequent instructions a specified number of times.	Structured assembly
`IF`	Assembles instructions if a condition is true.	Conditional assembly

Table 14: Assembler directives summary (continued)

Directive	Description	Section
`IFS`	Specifies instructions to be executed if a condition is true	Structured assembly
`IMPORT`	Imports an external symbol.	Symbol control
`LIBRARY`	Begins a library module.	Module control
`LIMIT`	Checks a value against limits.	Value assignment
`LOCAL`	Creates symbols local to a macro.	Macro processing
`LSTCND`	Controls conditional assembly listing.	Listing control
`LSTCOD`	Controls multi-line code listing.	Listing control
`LSTCYC`	Controls the listing of cycle counts.	Listing control
`LSTEXP`	Controls the listing of macro generated lines.	Listing control
`LSTMAC`	Controls the listing of macro definitions.	Listing control
`LSTOUT`	Controls assembly-listing output.	Listing control
`LSTPAG`	Controls the formatting of output into pages.	Listing control
`LSTREP`	Controls the listing of lines generated by repeat directives.	Listing control
`LSTSAS`	Controls structured assembly listing	Listing control
`LSTXRF`	Generates a cross-reference table.	Listing control
`MACRO`	Defines a macro.	Macro processing
`MODULE`	Begins a library module.	Module control
`NAME`	Begins a program module.	Module control
`ODD`	Aligns the program counter to an odd address.	Segment control
`ORG`	Sets the location counter.	Segment control
`PAGE`	Generates a new page.	Listing control
`PAGSIZ`	Sets the number of lines per page.	Listing control
`PROGRAM`	Begins a program module.	Module control
`PUBLIC`	Exports symbols to other modules.	Symbol control
`RADIX`	Sets the default base.	Assembler control
`REPEAT`	Repeats subsequent instructions until a condition is true.	Structured assembly
`REPT`	Assembles instructions a specified number of times.	Macro processing
`REPTC`	Repeats and substitutes characters.	Macro processing
`REPTI`	Repeats and substitutes strings	Macro processing
`RSEG`	Begins a relocatable segment.	Segment control
`SET`	Assigns a temporary value	Value assignment

Table 14: Assembler directives summary (continued)

Directive	Description	Section
`sfr`	Creates byte-access SFR labels.	Value assignment
`SFRTYPE`	Specifies SFR attributes.	Value assignment
`STACK`	Begins a stack segment.	Segment control
`SWITCH`	Multiple case switch	Structured assembly
`UNTIL`	Ends a REPEAT loop.	Structured assembly
`WHILE`	Repeats subsequent instructions until a condition is true.	Structured assembly

Table 14: Assembler directives summary (continued)

Syntax conventions

In the syntax definitions the following conventions are used:

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

`ORG` *`expr`*

`expr` represents an arbitrary expression.

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

`END [`*`expr`*`]`

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

`LOCAL` *`symbol`* `[,`*`symbol`*`]` …

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

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

`LSTOUT{+|-}`

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

LABELS AND COMMENTS

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

`label` `VAR` *`expr`*

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

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

PARAMETERS

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

Parameter	What it consists of
expr	An expression; see *Assembler expressions*, page 1.
label	A symbolic label.
symbol	An assembler symbol.

Table 15: Assembler directive parameters

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

Module control directives

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

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

Table 16: Module control directives

SYNTAX

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

PARAMETERS

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

DESCRIPTION

Beginning a program module

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

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

Beginning a library module

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

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

Terminating a module

Use `ENDMOD` to define the end of a module.

Terminating the last module

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

Assembling multi-module files

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

The following rules apply when assembling multi-module files:

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

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

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

Symbol control directives

These directives control how symbols are shared between modules.

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

Table 17: Symbol control directives

SYNTAX

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

PARAMETERS

`symbol`	Symbol to be imported or exported.

DESCRIPTION

Exporting symbols to other modules

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

The `PUBLIC` directive always exports full 32-bit values, which makes it feasible to use global 32-bit constants also in assemblers for 8-bit and 16-bit processors. With the `LOW`, `HIGH`, `BYTE2`, and `BYTE3` operators, any part of such a constant can be loaded in an 8-bit or 16-bit register or word.

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

Importing symbols

Use `EXTERN` to import an untyped external symbol.

EXAMPLES

The following example defines 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.

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

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

```
        NAME   error
        EXTERN print
        PUBLIC err

err     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.
`ASEG`	Begins an absolute segment.
`COMMON`	Begins a common segment.
`EVEN`	Aligns the program counter to an even address.
`ODD`	Aligns the program counter to an odd address.
`ORG`	Sets the location counter.
`RSEG`	Begins a relocatable segment.
`STACK`	Begins a stack segment.

Table 18: Segment control directives

SYNTAX

```
ALIGN align [,value]
ASEG [start [(align)]]
COMMON segment [:type] [(align)]
EVEN [value]
ODD [value]
ORG expr
RSEG segment [:type] [flag] [(align)]
RSEG segment [:type], address
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. For example, `align 1` results in word alignment `2`.
expr	Address to set the location counter to.

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

segment	The name of the segment.
start	A start address that has the same effect as using an `ORG` directive at the beginning of the absolute segment.
type	The memory type; one of: `UNTYPED` (the default), `CODE`, or `DATA`. In addition, the following types are provided for compatibility with the IAR C Compilers: `XDATA`, `IDATA`, `BIT`, `REGISTER`, and `CONST`.
value	Byte value used for padding, default is zero.

DESCRIPTION

Beginning an absolute segment

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

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

Beginning a relocatable segment

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

Up to 256 unique, relocatable segments may be defined in a single module.

Beginning a stack segment

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

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

Beginning a common segment

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

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

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

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

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

Setting the program location counter (PLC)

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

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

All program location counters are set to zero at the beginning of an 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 `EVEN` directive aligns the program counter to an even address.

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       R6,R7
       SUBI      R6,20
       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`.

Aligning a segment

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

```
        RSEG   data ; Start a relocatable data segment

        EVEN        ; Ensure it's on an even boundary
target  DW     1    ; Put target and best on even boundary
best    DW     1
        ALIGN  6    ; Now align to a 64 byte boundary
results DS     64   ; And create a 64 byte table
        END
```

Value assignment directives

These directives are used for assigning values to symbols.

Directive	Description
`=`	Assigns a permanent value local to a module.
`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 (ASSIGN)`	Assigns a temporary value.
`sfr`	Creates byte-access SFR labels.
`SFRTYPE`	Specifies SFR attributes.

Table 19: Value assignment directives

SYNTAX

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

```
label SET expr
label EQU expr
label = expr
label DEFINE expr
LIMIT label,min,max,message
[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 `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.

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
           builditttimes - 1
           ENDIF
           ENDM
main       buildit4
           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
           CLR        R7
           MOV        R6,locn
           MOV        R4,A
           MOV        R5,value
           ADD        R6,R7
           RET
           ENDMOD
           NAME       add2
value      EQU        77
           CLR        R7
           MOV        R6,locn
           MOV        R4,A
           MOV        R5,value
           ADD        R6,R7
           RET
           END
```

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

Using special function registers

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

```
sfrb portd= 0x12/*byte read/write access*/
const sfrb pind= 0x10/*byte read only access*/

SFRTYPE portb write, byte= 0x18/*byte write only access*/
```

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

Table 20: Conditional assembly directives

SYNTAX

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

PARAMETERS

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

DESCRIPTION

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

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

All assembler directives (except `END`) as well as the inclusion of files may be disabled by the conditional directives. Each `IFxx` 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=2
     DEC    r
     ELSE
     XCH    A,r
     SUBB   A,c
     XCH    A,r
     ENDIF
     ENDM
```

If the argument to the macro is 2, it generates an `SUBI` instruction to save instruction cycles; otherwise it generates a `DEC` 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.
`ENDR`	Ends a repeat structure.
`EXITM`	Exits prematurely from a macro.
`LOCAL`	Creates symbols local to a macro.
`MACRO`	Defines a macro.
`REPT`	Assembles instructions a specified number of times.
`REPTC`	Repeats and substitutes characters.
`REPTI`	Repeats and substitutes strings.

Table 21: Macro processing directives

SYNTAX

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

PARAMETERS

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

DESCRIPTION

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

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

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

For an example where macro directives are used, see *List file format*, page 6.

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:

```
         macld  <R6, 1>
         END
```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 18.

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.

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

EXAMPLES

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

Coding in-line for efficiency

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

The following 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
```

Structured assembly directives

The structured assembly directives allow loops and control structures to be implemented at assembly level.

Directive	Description
`BREAK`	Exits prematurely from a loop or switch construct.
`CASE`	Case in `S_SWITCH` block.
`CONTINUE`	Continues execution of a loop or switch construct.
`DEFAULT`	Default case in `S_SWITCH` block.
`ELSES`	Specifies instructions to be executed if a condition is false.
`ELSEIFS`	Specifies a new condition in an `S_IF...S_ENDIF` block.
`ENDF`	Ends an `S_FOR` loop.
`ENDIFS`	Ends an `S_IF` block.
`ENDS`	Ends an `S_SWITCH` block.
`ENDW`	Ends an `S_WHILE` loop.
`FOR`	Repeats subsequent instructions a specified number of times.

Table 22: Structured assembly directives

Directive	**Description**
`IFS`	Specifies instructions to be executed if a condition is true.
`REPEAT`	Repeats subsequent instructions until a condition is true.
`SWITCH`	Multiple case switch.
`UNTIL`	Ends an `S_REPEAT` loop.
`WHILE`	Repeats subsequent instructions until a condition is true.

Table 22: Structured assembly directives (continued)

SYNTAX

```
S_IF{condition | expression}
S_ELSE
S_ELSEIF{condition | expression}
S_ENDIF
S_WHILE{condition | expression}
S_ENDW
S_REPEAT
S_UNTIL{condition | expression}
S_FOR reg = start {TO | DOWNTO} end {BY | STEP} step
S_ENDF
S_SWITCH
S_CASE op
S_CASE op1..op2
S_DEFAULT
S_ENDS
S_BREAK levels
S_CONTINUE
```

PARAMETERS

`condition` One of the following conditions:

`<CC>` Carry clear

`<CS>` Carry set

`<EQ>` Equal

`<NE>` Not equal

`<VC>` Overflow clear

`<VS>` Overflow set.

`expression`	An expression of the form: `reg rel op`
`reg`	One of the following registers: `R0`...`R31`, `ZERO`, `HP`, `SP`, `GP`, `TP`, `EP`, `LP`
`rel`	One of the following relations: `>=`, `<=`, `!=`, `<>`, `==`, `=`, `>` or `<`
`op`, *`op1`*, *`op2`*	An intermediate or memory operand.
`start`, *`end`*, *`step`*	An intermediate or memory operand. If `step` is omitted it defaults to `#1` or `#-1` if `DOWNTO` is specified. The increment or decrement in this structure is implemented with `ADD/SUB`.
`levels`	Number of levels to break, from 1 to 3.

DESCRIPTION

The 8051 IAR Assembler includes a versatile range of directives for structured assembly, to make it easier to implement loops and control structures at assembly level.

The advantage of using the structured assembly directives is that the resulting programs are clearer, and their logic is easier to understand.

The directives are designed to generate simple, predictable code so that the resulting program is as efficient as if it were programmed by hand.

Conditional constructs

Use `S_IF...S_ENDIF` to generate assembler source code for comparison and jump instructions. The generated code is assembled like ordinary code, and is similar to macros. This should not be confused with conditional assembly.

`S_IF` blocks can be nested to any level.

Use `S_ELSE` after an `S_IF` directive to introduce instructions to be executed if the `S_IF` condition is false.

Use `S_ELSEIF` to introduce a new condition after an `S_IF` directive.

Loop directives

Use `S_WHILE...S_ENDW` to create a loop which is executed as long as the expression is `TRUE`. If the expression is false at the beginning of the loop the body will not be executed.

Use the `S_REPEAT...S_UNTIL` construct to create a loop with a body that is executed at least once, and as long as the expression is `FALSE`.

You can use `S_BREAK` to exit prematurely from an `S_WHILE...S_ENDW` or `S_REPEAT...S_UNTIL` loop, or `S_CONTINUE` to continue with the next iteration of the loop.

The directives generate the same statements as the `S_IF` directive.

Iteration construct

Use `S_FOR...S_ENDF` to assemble instructions to repeat a block of instructions for a specified sequence of values.

`S_BREAK` can be used to exit prematurely from an `S_FOR` loop, and continue execution following the `S_ENDF`.

`S_CONTINUE` can be used to continue with the next iteration of the loop.

Switch construct

Use the `S_SWITCH...S_ENDS` block to execute one of a number of sets of statements, depending on the value of test.

`S_CASE` defines each of the tests, and `S_DEFAULT` introduces an `S_CASE` which is always true.

Note that `S_CASE` falls through by default similar to switch statements in the C language.

`S_BREAK` can be used to exit from a `S_SWITCH...S_ENDS` block.

EXAMPLES

Using conditional constructs

The following program tests the `A` register and plots `'N'`, `'Z'`, or `'P'`, depending on whether it is less than zero, zero, or greater than zero:

```
        NAME    else
        EXTERN  plot

main    IFS     A < 0
        MOV     A,'N'
        ELSEIFS A == 0
        MOV     A,'Z'
        ELSES
        MOV     A,'p'
        ENDIFS
        CALL    plot
        RET
        END
```

This generates the following code:

```
     1    000000                    NAME    else
     2    000000                    EXTERN  plot
     3    000000
     4    000000          main      IFS     A < 0
     4.1  000000 C0E0               PUSH    ACC
     4.2  000002 C3                 CLR     CY
     4.3  000003 9500               SUBB    A,0
     4.4  000005 D0E0               POP     ACC
     4.5  000007 5004               JNC     _?0
     5    000009 E54E               MOV     A,'N'
     6    00000B                    ELSEIFS A == 0
     6.1  00000B 8016               JMP     _?1
     6.2  00000D          _?0
     6.3  00000D C0E0               PUSH    ACC
     6.4  00000F D2D1               SETB    PSW.1
     6.5  000011 C3                 CLR     CY
     6.6  000012 9500               SUBB    A,0
     6.7  000014 6002               JZ      $+4
     6.8  000016 C2D1               CLR     PSW.1
     6.9  000018 D0E0               POP     ACC
     6.10 00001A 30D104             JNB     PSW.1,_?2
     7    00001D E55A               MOV     A,'Z'
     8    00001F                    ELSES
     8.1  00001F 8002               JMP     _?1
     8.2  000021          _?2
     9    000021 E570               MOV     A,'p'
    10    000023                    ENDIFS
    10.1  000023          _?1
    11    000023 12....             CALL    plot
    12    000026 22                 RET
    13    000027                    END
```

Using loop constructs

The following example uses an `REPEAT ... UNTIL` loop to reverse the order of bits in register `B` and put the result in register `A`:

```
        NAME   repeat
reverse REPEAT
        XCH    A,0xF0
        RRC    A
        XCH    A,0xF0
        RLC    A
        UNTIL  A<> #0
        RET

        END
```

This generates the following code:

```
 1    000000                 NAME   repeat
 2    000000          reverse REPEAT
 2.1  000000          _?0
 3    000000 C5F0            XCH    A,0xF0
 4    000002 13              RRC    A
 5    000003 C5F0            XCH    A,0xF0
 6    000005 33              RLC    A
 7    000006                 UNTIL  A<> #0
 7.1  000006 C0E0            PUSH    ACC
 7.2  000008 D2D1            SETB    PSW.1
 7.3  00000A C3              CLR     CY
 7.4  00000B 9400            SUBB    A,#0
 7.5  00000D 6002            JZ      $+4
 7.6  00000F C2D1            CLR     PSW.1
 7.7  000011 D0E0            POP     ACC
 7.8  000013 20D1EA          JB      PSW.1,_?0
 7.9  000016          _?1
 8    000016 22              RET
 9    000017
10    000017                 END
```

Using iteration constructs

The following example uses an `FOR ... ENDF` block to send a 501 even number to a part named `port1`:

```
        NAME     for_loop
        EXTERN   port1
play    FOR      A = #0 TO #100 BY #2
        MOV      port1,A
        ENDF
        RET

        END
```

This generates the following code:

```
 1    000000                 NAME     for_loop
 2    000000                 EXTERN   port1
 3    000000          play    FOR      A = #0 TO #100 BY #2
 3.1  000000 7400            MOV     A,#0
```

```
3.2  000002 8004            JMP     _?1
3.3  000004         _?0
4    000004 F5..            MOV      port1,A
5    000006                 ENDF
5.1  000006 2402    _?2     ADD     A,#2
5.2  000008 C0E0    _?1     PUSH    ACC
5.3  00000A C3              CLR     CY
5.4  00000B 9464            SUBB    A,#100
5.5  00000D D0E0            POP     ACC
5.6  00000F 40F3            JC      _?0
5.7  000011         _?3
6    000011 22              RET
7    000012
8    000012                 END
```

Using switch constructs

The following example uses an `SWITCH...ENDS` block to print `Zero`, `Positive`, or `Negative` depending on the value of the `A` register. It uses an external `print` routine to print an immediate string:

```
pos     DB      "Positive"
neg     DB      "Negative"
zer     DB      "Zero"

        NAME    switch
        EXTERN  print

test    SWITCH  A

        CASE    #0
        MOV     R3,#LOW(zer)
        MOV     R4,#HIGH(zer)
        CALL    print
        BREAK

        CASE    #0x80 .. #0xFF
        MOV     R3,#LOW(neg)
        MOV     R4,#HIGH(neg)
        CALL    print
        BREAK

        DEFAULT
        MOV     R3,#LOW(pos)
        MOV     R4,#HIGH(pos)
        CALL    print
        BREAK
        ENDS
        END
```

This generates the following code:

```
 1    000000 506F7369*pos     DB      "Positive"
 2    000009 4E656761*neg     DB      "Negative"
 3    000012 5A65726F*zer     DB      "Zero"
 4    000017
 5    000017                  NAME    switch
 6    000000                  EXTERN  print
 7    000017
 8    000017          test    SWITCH  A
 9    000017
10    000017                  CASE    #0
10.1  000017 C0E0             PUSH    ACC
10.2  000019 D2D1             SETB    PSW.1
10.3  00001B C3               CLR     CY
10.4  00001C 9400             SUBB    A,#0
10.5  00001E 6002             JZ      $+4
10.6  000020 C2D1             CLR     PSW.1
10.7  000022 D0E0             POP     ACC
10.8  000024 30D109           JNB     PSW.1,_?1
11    000027 7B12             MOV     R3,#LOW(zer)
12    000029 7C00             MOV     R4,#HIGH(zer)
13    00002B 12....           CALL    print
14    00002E                  BREAK
14.1  00002E 802D             JMP     _?0
15    000030
16    000030                  CASE    #0x80 .. #0xFF
16.1  000030 C0E0     _?1     PUSH    ACC
16.2  000032 C3               CLR     CY
16.3  000033 9480             SUBB    A,#0x80
16.4  000035 D0E0             POP     ACC
16.5  000037 401B             JC      _?2
16.6  000039 C0E0             PUSH    ACC
16.7  00003B D2D1             SETB    PSW.1
16.8  00003D C3               CLR     CY
16.9  00003E 94FF             SUBB    A,#0xFF
16.10 000040 6002             JZ      $+4
16.11 000042 C2D1             CLR     PSW.1
16.12 000044 D0E0             POP     ACC
16.13 000046 4003             JC      $+5
16.14 000048 30D109           JNB     PSW.1,_?2
17    00004B 7B09             MOV     R3,#LOW(neg)
18    00004D 7C00             MOV     R4,#HIGH(neg)
19    00004F 12....           CALL    print
20    000052                  BREAK
```

```
20.1  000052 8009             JMP     _?0
21    000054
22    000054                  DEFAULT
22.1  000054          _?2
23    000054 7B00             MOV     R3,#LOW(pos)
24    000056 7C00             MOV     R4,#HIGH(pos)
25    000058 12....           CALL    print
26    00005B                  BREAK
26.1  00005B 8000             JMP     _?0
27    00005D                  ENDS
27.1  00005D          _?0
28    00005D
29    00005D                  END
```

Listing control directives

These directives provide control over the assembler list file.

Directive	Description
`COL`	Sets the number of columns per page.
`CYCLES`	Sets the listed cycle count.
`CYCLEMAX`	Selects the greater of two possible cycle count values.
`CYCLEMIN`	Selects the lower of two possible cycle count values.
`CYCLEMEAN`	Selects the mean value.
`LSTCND`	Controls conditional assembly listing.
`LSTCOD`	Controls multi-line code listing.
`LSTCYC`	Controls the listing of cycle counts.
`LSTEXP`	Controls the listing of macro-generated lines.
`LSTMAC`	Controls the listing of macro definitions.
`LSTOUT`	Controls assembly-listing output.
`LSTPAG`	Controls the formatting of output into pages.
`LSTREP`	Controls the listing of lines generated by repeat directives.
`LSTSAS`	Controls structured assembly listing.
`LSTXRF`	Generates a cross-reference table.
`PAGE`	Generates a new page.
`PAGSIZ`	Sets the number of lines per page.

Table 23: Listing control directives

SYNTAX

```
COL columns
LSTCND{+ | -}
LSTCOD{+ | -}
LSTCYC{+ | -}
LSTEXP{+ | -}
LSTMAC{+ | -}
LSTOUT{+ | -}
LSTPAG{+ | -}
LSTREP{+ | -}
LSTSAS{+ | -}
LSTXRF{+ | -}
COL columns
CYCLES expr
CYCLEMAX
CYCLEMIN
CYCLEMEAN
PAGE
PAGSIZ lines
```

PARAMETERS

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

DESCRIPTION

Turning the listing on or off

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

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

Listing conditional code and strings

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

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

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

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

Controlling the listing of macros

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

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

Controlling the listing of generated lines

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

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

Generating a cross-reference table

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

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

Specifying the list file format

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

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

Use `LSTPAG+` to format the assembly output list into pages. The default is `LSTPAG-`, which gives a continuous listing.

Use `PAGE` to generate a new page in the assembly 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    prom
     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-
11    000000
12    000000          begin    dec2   R6
12    000000          begin    dec2   R6
12.1  000000 A51E             DEC     R6
12.2  000002 A51E             DEC     R6
12.3  000004                  ENDM
13    000004
14    000004                  LSTEXP+
15    000004                  inc2    R7
15.1  000004 A50F             INC     R7
15.2  000006 A50F             INC     R7
15.3  000008                  ENDM
16    000008 22               RET
17    000009
18    000009                  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 24: 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 VAR value
```

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

Conditional directives

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

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

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

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

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

Including source files

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

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

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

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

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

Displaying errors

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

Defining comments

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

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

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

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

```
#define five 5 ; comment

  STS  five+addr,R17   ;syntax error!
  ; Expands to "STS 5 ; comment+addr,R17"

  LDS  R16,five + addr; incorrect code!
  ; Expanded to "LDS R16,5 ; comment + addr"
```

EXAMPLES

Using conditional directives

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

```
#definetweek 1
#defineadjust  3

#ifdef tweek
#if    adjust=1
       SUB       R6,4
#elif   adjust=2
       SUB       R6,20
#elif  adjust=3
       SUB       R6,30
#endif
#endif  /* ifdef tweek*/
```

Including a source file

The following example uses `#include` to include a file defining macros into the source file. For example, the following macros could be defined in `macros.s03`:

```
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.s03"

; Program
main   xch       R6,R7
       RET
       END       main
```

Data definition or allocation directives

These directives define temporary values or reserve memory.

Directive	Description
`DB`	Generates 8-bit byte constants, including strings.
`DC16`	Generates 16-bit word constants, including strings.
`DC24`	Generates 24-bit word constants.
`DC32`	Generates 32-bit double word constants.
`DC8`	Generates 8-bit byte constants, including strings.
`DD`	Generates 32-bit double word constants.
`DS`	Allocates space for 8-bit bytes.
`DS16`	Allocates space for 16-bit words.
`DS24`	Allocates space for 24-bit words.
`DS32`	Allocates space for 32-bit words.
`DS8`	Allocates space for 8-bit bytes.
`DT`	Generates 24-bit word constants.
`DW`	Generates 16-bit word constants, including strings.

Table 25: Data definition or allocation directives

SYNTAX

```
DB expr[,expr] ...
DC16 expr [,expr] ...
DC24 expr [,expr] ...
DC32 expr [,expr] ...
DC8 expr [,expr] ...
DD expr[,expr] ...
DS expr[,expr] ...
DS16 expr [,expr] ...
DS24 expr [,expr] ...
DS32 expr [,expr] ...
DS8 expr [,expr] ...
DT expr[,expr] ...
DW expr[,expr] ...
```

PARAMETERS

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

DESCRIPTION

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

Use `DS`, `DW`, `DT`, `DP` `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   DW     addsubr,subsubr,clrsubr

addsubr ADD    R6,R7
        RET

subsubr SUB    R6,R7
        RET

clrsubr CLR    R6
        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.

Table 26: Assembler control directives

SYNTAX

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

PARAMETERS

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

DESCRIPTION

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

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

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

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

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

```
RADIX  0x0A
```

Controlling case sensitivity

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

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

EXAMPLES

Including a source file

The following example uses `$` to include a file defining macros into the source file. For example, the following macros could be defined in `mymacros.s03`:

```
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
$macros.s03

; Program
main    xch       R6,R7
        RET
        END       main
```

Defining comments

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

```
/*
Program to read serial input.
Version 3: 19.9.00
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`.

Controlling case sensitivity

By default `CASEON` is active, so the following example will generate an error:

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

However, the `CASEOFF` directive will remove the error in the example above:

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

Assembler diagnostics

When the 8051 IAR Assembler performs a diagnostic check, it may detect errors in your application and give a diagnostic message. This chapter lists the different error and warning messages that can appear.

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 96.

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 88.

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. To display it at sign-on, run the assembler, `a8051`, without parameters.

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

404 **Illegal suffix**

405 **Illegal value value**

406 **Illegal size specifier specifier**

407 **C-comment has no end**

408	**Could not solve step**
409	**Nothing to BREAK out of**
410	**CASE after DEFAULT** DEFAULT is a catch-all case and is not allowed to have a CASE after it.
411	**CASE outside SWITCH**
412	**COMMA expected**
413	**Nothing to CONTINUE to** CONTINUE needs something to continue.
414	**Cannot solve break** The break count must be solvable.count value
415	**DEFAULT outside SWITCH**
416	**ELSE used more than once** It is not allowed to have multiple ELSE directives for an IF.
417	**ELSE without matching IF**
418	**ELSEIF cannot be used after ELSE**
419	**ELSEIF with no matching IF**
420	**ENDF without matching FOR**
421	**ENDIF without matching IF**
422	**ENDS without matching SWITCH**
423	**ENDW without matching WHILE**
424	**THEN without matching IF**
425	**Negative step value**
426	**Zero step value**
427	**UNTIL without matching REPEAT**
428	**Break argument must be 1,2, or 3**
429	**Multiple DEFAULT** It is not allowed to have more than one DEFAULT inside a SWITCH.
430	**Can’t assign register to register**

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 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 flaggs**
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**