AVR IAR Assembler

Reference Guide

for Atmel Corporation's

AVR Microcontroller

EDITION NOTICE

Second edition: June 2003

Part number: AAVR-2

Contents

Tables

Preface

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

Who should read this guide

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

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

How to use this guide

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

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

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

What this guide contains

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

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

Other documentation

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

- Using the IAR Embedded Workbench™ and the IAR C-SPY™ Debugger, refer to the *AVR IAR Embedded Workbench™ IDE User Guide*
- Programming for the AVR IAR C/EC++ Compiler, refer to the *AVR IAR C/EC++ Compiler Reference Guide*
- Using the IAR XLINK Linker™, the IAR XLIB Librarian™, and the IAR XAR Library Builder™, refer to the *IAR Linker and Library Tools Reference Guide*.
- Using the IAR C Library, refer to the *IAR C Library Functions Reference Guide*, available from the IAR Embedded Workbench IDE **Help** menu.
- Using the Embedded C++ Library, refer to the *EC++ Library Reference*, available from the IAR Embedded Workbench IDE **Help** menu.

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

Document conventions

This guide uses the following typographic conventions:

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

Table 1: Typographic conventions used in this guide

Introduction to the AVR IAR Assembler

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

Refer to Atmel Corporation's hardware documentation for syntax descriptions of the instruction mnemonics.

Source format

The format of an assembler source line is as follows:

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

where the components are as follows:

label	A label, which is assigned the value and type of the current program location counter (PLC). The `:` (colon) is optional if the label starts in the first column. A directive starting in the first column will be handled as a directive. Use `-j_no_directives_at_linebeg` to have it handled as a label.
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. The data definition directives, for example DB and DC8, can have any number of operands. For reference information about the data definition directives, see *Data definition or allocation directives*, page 79. Other assembler directives can have one, two, or three operands, separated by commas.
comment	Comment, preceded by a `;` (semicolon) Use /* … */ to comment sections Use // to mark the rest of the line as comment.

The fields can be separated by spaces or tabs.

A source line may not exceed 2047 characters.

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

The AVR IAR Assembler uses the default filename extensions `s90`, `asm`, and `msa` for source files.

List file format

The format of an assembler list file is as follows:

HEADER

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

BODY

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

- The line number in the source file. Lines generated by macros 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. Unresolved values are represented by ..... (periods), where two periods signify one byte. These unresolved values will be resolved during the linking process.
- The assembler source line.

SUMMARY

The *end* of the file contains a summary of errors and warnings that were generated, and a checksum (CRC).

Note: The CRC number depends on the date when the source file was assembled.

SYMBOL AND CROSS-REFERENCE TABLE

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

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 type.
Segment	The name of the segment that this label is defined relative to.
Value/Offset	The value (address) of the label within the current module, relative to the beginning of the current segment part.

Table 2: Symbol and cross-reference table

Assembler expressions

Expressions consist of operands and operators.

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

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators; see also *Precedence of operators*, page 29.

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 29.

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
        RSEG    CODE
start   …
```

Then in the segment CODE the following relocatable expressions are legal:

```
        LDI     R27,first
        LDI     R27,first+1
        LDI     R27,1+first
        LDI     R27,(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 -s on page 23 for additional information.

Note that symbols and labels are byte addresses. For additional information, see *Generating lookup table*, page 81.

LABELS

Symbols used for memory locations are referred to as labels.

Program location counter (PLC)

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

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

```
        RJMP    $       ; 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 of them to indicate a negative number.

Commas and decimal points are not permitted.

The following types of number representation are supported:

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

Table 3: Integer constant formats

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

ASCII CHARACTER CONSTANTS

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

Format	Value
'ABCD'	ABCD (four characters).
"ABCD"	ABCD'\0' (five characters the last ASCII null).
'A''B'	A'B
'A'''	A'

Table 4: ASCII character constant formats

Format	Value
' ' ' ' (4 quotes)	'
' ' (2 quotes)	Empty string (no value).
""	Empty string (an ASCII null character).
\'	'
\\	\

Table 4: ASCII character constant formats (Continued)

PREDEFINED SYMBOLS

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

The following predefined symbols are available:

Symbol	Value
__DATE__	Current date in dd/Mmm/yyyy format (string).
__FILE__	Current source filename (string).
__IAR_SYSTEMS_ASM__	IAR assembler identifier (number).
__LINE__	Current source line number (number).

__TID__	Target identity, consisting of two bytes (number). The high byte is the target identity, which is `90` for `AAVR`. The low byte is the processor option *16.
	The following values are therefore possible:
	`-v0` `0x5A00`
	`-v1` `0x5A10`
	`-v2` `0x5A20`
	`-v3` `0x5A30`
	`-v4` `0x5A40`
	`-v5` `0x5A50`
	`-v6` `0x5A60`

__TIME__	Current time in `hh:mm:ss` format (string).

Table 5: Predefined symbols

Symbol	Value
__VER__	Version number in integer format; for example, version 4.17 is returned as 417 (number).

Table 5: Predefined symbols (Continued)

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

Including symbol values in code

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

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

```
tim     DC8    __TIME__        ; Time string
        ...
        LD     R16,LOW(tim)  ;  Load low byte of address of
                                    ; string in R16
        LD     R17,tim>>8      ; Load high byte of address
                               ; of string in R16
                               ; Don't use HIGH() since
                               ; this would prevent XLINK
                               ; from making a proper
                               ; range check
        RCALL  printstr        ; Call string output
                               ; routine
```

Testing symbols for conditional assembly

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

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

```
#define TARGET ((__TID__& 0x0F0)>>4)
#if (TARGET==1)
...
...
#else
...
...
#endif
```

See *Conditional assembly directives*, page 61.

Register symbols

The following table shows the existing predefined register symbols:

Name	Address size	Description
`R0–R31`	8 bits	General purpose registers
`X`	16 bits	R27 and R26 combined
`Y`	16 bits	R29 and R28 combined
`Z`	16 bits	R31 and R30 combined

Table 6: Predefined register symbols

To specify a *register pair*, use `:` (colon), as in the following example:

```
R17:R16
```

Notice that only consecutive registers can be specified in register pairs. The upper odd register should be entered to the left of the colon, and the lower even register to the right.

Programming hints

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

ACCESSING SPECIAL FUNCTION REGISTERS

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

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

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

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

USING C-STYLE PREPROCESSOR DIRECTIVES

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

MIGRATING ASSEMBLER SOURCE CODE FROM THE ATMEL AVR ASSEMBLER TO THE AVR IAR ASSEMBLER

Although the Atmel AVR Assembler and the AVR IAR Assembler use the same mnemonics for the instructions they do not use the same assembler directives. Neither do they treat labels in code space in the same way. This section gives guidelines on how to migrate code from the Atmel AVR Assembler to the AVR IAR Assembler.

Directives

The AVR IAR Assembler directly supports all, except two, of the Atmel AVR Assembler directives. The difference lies in the formatting of the directives. The two unsupported directives are: `.DEVICE` and `EXIT`. See *Handling the unsupported directives*, page 10, for information on how to migrate these directives. The table below shows how to translate the Atmel directives into IAR directives. Text written in italics represents data fields that match between the two formats, underlined text represents features only available in one format.

Atmel AVR Assembler format	AVR IAR Assembler format	Comments
label: .BYTE *size*	*label:* DS8 *size*	
.CSEG	RSEG <u>segment name:</u>CODE: <u>segment flags</u>	1
.DB *data1*,*data2*,*data3*	DB *data1*,*data2*,*data3*	
.DEF *name* = *value*	#define *name value*	2
.DSEG	RSEG <u>segment name:</u>DATA<u>:segment flags</u>	1
.DW *data1*,*data2*,*data3*	DW *data1*,*data2*,*data3*	
.ENDMACRO	ENDM	
.EQU *label* = *expression*	*label* EQU *expression*	
.ESEG	RSEG <u>segment name:</u>XDATA<u>:segment flags</u>	1
.INCLUDE *file*	#include *file*	2
.LIST	LSTOUT+	
.LISTMAC	LSTEXP+	
.MACRO *macroname*	*macroname* MACRO *arguments…*	3

Table 7: Migrating from Atmel AVR Assembler to AVR IAR Assembler

Atmel AVR Assembler format	AVR IAR Assembler format	Comments
`.NOLIST`	`LSTOUT-`	
`.ORG` *`expression`*	`ORG` *`expression`*	
`.SET` *`label`* `=` *`expression`*	*`label`* `VAR` *`expression`*	

Table 7: Migrating from Atmel AVR Assembler to AVR IAR Assembler (Continued)

1: If no segment name or type (`CODE`, `DATA`, or `XDATA`) is specified, an unnamed segment of type `UNTYPED` is created.
2: The C-style preprocessor of the AVR IAR Assembler is used instead of the assembler macro processor.
3: The names of the macro parameters are `\1`, `\2`, ... in the AVR IAR Assembler instead of `@0`, `@1`, ... in the Atmel AVR Assembler.

Handling the unsupported directives

The `.DEVICE` directive is not required in the AVR IAR Assembler where you instead use the `-v` command line option to specify for what kind of microcontroller the assembler source code is being assembled. Refer to the *AVR IAR C/EC++ Compiler Reference Guide* for a translation table between derivative names and processor options.

The `.EXIT` directive does not exist in the AVR IAR Assembler. You can replace this directive by enclosing the text after the `.EXIT` directive with the `#if 0` and `#endif` preprocessor directives. It is not possible to implement the `.EXIT` directive within a macro.

Linking

The AVR IAR Assembler does not produce an output file that can be used directly for downloading code into the AVR microcontroller; the object file must first be linked, using the IAR XLINK Linker. This applies also to projects consisting of only one assembler source file.

Modules and segments

A single assembler source file may consist of several modules, and each module can consist of one or more segments. Each segment can consist of multiple segment parts. When the IAR XLINK Linker links the project, it will remove all segment parts that are not referenced by another module. It is therefore important to remember to have at least one program module in each project.

Labels

Both the Atmel AVR Assembler and the AVR IAR Assembler treat all labels, except labels in code segments, as byte addresses. Code that works with labels in data segments does not have to be altered. Notice however that the Atmel AVR Assembler treats labels in code segments as *word* addresses whereas the AVR IAR Assembler treats them as *byte* addresses. It is therefore important to remember to alter the code to reflect this; see the example below.

Also notice that labels are local to one module. To access a label in another module, export it, using the `PUBLIC` directive, from the module where it is declared. Then import it, using the `EXTERN` directive, into the module where it is used.

Atmel AVR Assembler example:

```
.CSEG
start: LDI        R30,low(2*code_pointer)
       LDI        R31,high(2*code_pointer)
       LPM
       MOV        R16,R0
       ADIW       R30,1
       LPM
       MOV        R31,R0
       MOV        R30,R16
       ICALL
       RJMP       start
func:  LDI        R16,0
       RET
code_pointer:
       DW         func
```

AVR IAR Assembler example:

```
       MODULE Example

       RSEG       SEGMENT_NAME:CODE

start: LDI        R30,low(code_pointer)
       LDI        R31,high(code_pointer)
       LPM
       MOV        R16,R0
       ADIW       R30,1
       LPM
       MOV        R31,R0
       MOV        R30,R16
       ICALL
       RJMP       start
```

```
        RSEG        SEGMENT_NAME:CODE

func:   LDI         R16,0
        RET

        RSEG        SEGMENT_NAME:CODE

code_pointer:
        DW          func / 2

        END
```

Note that, in the Atmel AVR Assembler case, the first reference to a label in a code segment is multiplied by two. This is necessary since the `LPM` instruction uses *byte* addressing of the flash memory whereas labels in code segments are *word* addresses. In the AVR IAR Assembler case there is no need to multiply the label by two since all labels are byte addresses.

In the AVR IAR Assembler case, notice that the address of the function label is divided by two in the declaration of `code_pointer`. This is necessary since `ICALL` uses *word* addresses and all labels in the AVR IAR Assembler are *byte* labels.

Assembler options

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

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

Setting command line options

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

```
aavr [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.s90`, use the following command to generate a list file to the default filename (`power2.lst`):

```
aavr 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`:

```
aavr 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`:

```
aavr 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:

```
aavr -f extend.xcl
```

ERROR RETURN CODES

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

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

Table 8: Assembler error return codes

ASSEMBLER ENVIRONMENT VARIABLES

Options can also be specified using the `ASMAVR` 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 AVR IAR Assembler:

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

Table 9: Asssembler environment variables

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

```
ASMAVR=-l temp.lst
```

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

Summary of assembler options

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

Command line option	Description
`-B`	Macro execution information
`-b`	Makes a library module
`-c{DMEAO}`	Conditional list
`-D`*symbol*`[=`*value*`]`	Defines a symbol
`-E`*number*	Maximum number of errors
`-f` *filename*	Extends the command line
`-G`	Opens standard input as source
`-I`*prefix*	Includes paths
`-i`	Lists #included text
`-j_no_directives_at_linebeg`	Treats assembler directives starting in the first column as labels
`-L[`*prefix*`]`	Lists to prefixed source name
`-l` *filename*	Lists to named file
`-M`*ab*	Macro quote characters
`-N`	Omit header from assembler listing
`-n`	Enables support for multibyte characters
`-O`*prefix*	Sets object filename prefix
`-o` *filename*	Sets object filename
`-p`*lines*	Lines/page
`-r`	Generates debug information
`-S`	Sets silent operation
`-s{+|-}`	Case-sensitive user symbols
`-t`*n*	Tab spacing
`-U`*symbol*	Undefines a symbol
`-u_enhancedCore`	Enables AVR-specific enhanced instructions
`-v[0|1|2|3|4|5|6]`	Processor configuration
`-w[`*string*`][s]`	Disables warnings
`-x{DI2}`	Includes cross-references

Table 10: Assembler options summary

Descriptions of assembler options

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

-B

`-B`

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

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

This option is mainly used in conjunction with the list file options `-L` or `-l`; for additional information, see page 19.

This option is identical to the **Macro execution info** option in the **AAVR** 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 in the **AAVR** category in the IAR Embedded Workbench.

-c

`-c{DMEAO}`

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 19 for additional information.

The following table shows the available parameters:

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

Table 11: Conditional list (-c)

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

`-D` `-Dsymbol[=value]`

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

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

Example

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

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

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

Production version:	`aavr prog`
Test version:	`aavr 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:

```
aavr prog -DFRAMERATE=3
```

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

-E

`-Enumber`

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

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

This option is identical to the **Max number of errors** option in the **AAVR** category in the IAR Embedded Workbench.

-f

`-f filename`

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

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

Example

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

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

Example

Using the options:

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

and then writing:

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

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

You can also specify the include path with the `AAVR_INC` environment variable, see *Assembler environment variables*, page 14.

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

`-j_no_directives_at_linebeg`

`-j_no_directives_at_linebeg`

The default behavior of the assembler is to treat assembler directives starting in the first column as directives, not labels.

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

`-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 the extension will be `lst`.

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

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

Example

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

```
aavr prog -Llist\
```

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

`-M` `-M`*`ab`*

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

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

Example

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

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

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

```
aavr filename -M'<>'
```

This option is identical to the **Macro quote chars** option in the **AAVR** 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 19 for additional information.

This option is related to the **List file** option in the **AAVR** category in the IAR Embedded Workbench.

`-n` `-n`

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

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

This option is identical to the **Enable multibyte support** option in the **AAVR** category in the IAR Embedded Workbench.

`-O` `-O`*`prefix`*

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

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

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

Example

To send the object code to the file `obj\prog.r90` rather than to the default file `prog.r90`:

```
aavr 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, `r90` 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.r90` instead of the default `prog.r90`:

```
aavr 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` `-plines`

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 19 for additional information.

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

`-r` `-r`

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

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

This option is identical to the **Generate debug information** option in the **AAVR** 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 12: Controlling case sensitivity in user symbols (-s)

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

This option is identical to the **Case sensitive user symbols** option in the **AAVR** 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 19 for additional information.

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

-U `-Usymbol`

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

By default, the assembler provides certain predefined symbols; see *Predefined symbols*, page 6. 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:

```
aavr prog -U __TIME__
```

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

-u_enhancedCore `-u_enhancedCore`

Use this option to allow the assembler to generate instructions from the enhanced instruction set that is available in some AVR derivatives, for example AT90mega161.

The enhanced instruction set consists of the following instructions:

```
MUL
MOVW
MULS
MULSU
FMUL
FMULS
FMULSU
LPM Rd,Z
LPM Rd,Z+
ELPM Rd,Z
ELPM Rd,Z+
SPM
```

This option corresponds to the **Enhanced core** option in the **General** category in the IAR Embedded Workbench.

-v `-v[0|1|2|3|4|5|6]`

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

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

- In the options `-v0` and `-v1`, relative jumps reach the entire address space.
- In the options `-v2`, `-v3`, and `-v4`, jumps do not wrap. The `ELPM` instruction is supported.
- The `-v5` and `-v6` options have the same characteristics as `-v3`. In addition, they support the `EICALL` and `EIJMP` instructions.

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

Option	Description	Derivative
`-v0`	≤ 8 Kbytes code. `RJMP` wraparound is possible, that is `RJMP` and `RCALL` can reach the entire address space.	ATtiny10 ATtiny11 ATtiny12 ATtiny15 ATtiny22 ATtiny26 ATtiny28 AT90S1200 AT90S2313 AT90S2323 AT90S2333 AT90S2343 AT90S4433
`-v1`	≤ 8 Kbytes code. `RJMP` wraparound is possible, that is `RJMP` and `RCALL` can reach the entire address space.	ATmega8 ATmega8515 ATmega8535 AT90S4414 AT90S4434 AT90S8515 AT90S8534 AT90S8535
`-v2`	≤ 128 Kbytes code. `RJMP` wraparound is not possible, that is `RJMP` and `RCALL` cannot reach the entire address space. `CALL` and `JMP` are available.	Currently no derivative available using this model.

Table 13: Specifying the processor configuration (-v)

Option	Description	Derivative
`-v3`	≤ 128 Kbytes code. `RJMP` wraparound is not possible, that is `RJMP` and `RCALL` cannot reach the entire address space. `CALL` and `JMP` are available.	ATmega16 ATmega32 ATmega64 ATmega83 ATmega103 ATmega128 ATmega161 ATmega162 ATmega163 ATmega169 ATmega323 FpSLic (at94k)
`-v4`	≤ 128 Kbytes code. `RJMP` wraparound is not possible, that is `RJMP` and `RCALL` cannot reach the entire address space. `CALL` and `JMP` are available.	Currently no derivative available using this model.
`-v5`	≤ 8 Mbytes code. `RJMP` wraparound is not possible, that is `RJMP` and `RCALL` cannot reach the entire address space. `CALL` and `JMP` are available.	Currently no derivative available using this model.
`-v6`	≤ 8 Mbytes code. `RJMP` wraparound is not possible, that is `RJMP` and `RCALL` cannot reach the entire address space. `CALL` and `JMP` are available.	Currently no derivative available using this model.

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

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

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

`-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 *Diagnostics*, page 99, 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

Table 14: Disabling assembler warnings (-w)

Command line option	Description
`-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 14: Disabling assembler warnings (-w) (Continued)

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:

```
aavr prog -w-0
```

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

```
aavr prog -w-0-8
```

This option is identical to the **Warnings** option in the **AAVR** 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.

This option is useful in conjunction with the list options `-L` or `-l`; see page 19 for additional information.

The following parameters are available:

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

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

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

Assembler operators

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

Precedence of operators

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

The following rules determine how expressions are evaluated:

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

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

Summary of assembler operators

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

UNARY OPERATORS – 1

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

`BYTE3`	Third byte.
`LWRD`	Low word.
`HWRD`	High word.
`DATE`	Current time/date.
`SFB`	Segment begin.
`SFE`	Segment end.
`SIZEOF`	Segment size.

MULTIPLICATIVE ARITHMETIC AND SHIFT OPERATORS – 3

`*`	Multiplication.
`/`	Division.
`MOD, %`	Modulo.
`SHR, >>`	Logical shift right.
`SHL, <<`	Logical shift left.

ADDITIVE ARITHMETIC OPERATORS – 4

`+`	Addition.
`-`	Subtraction.

AND OPERATORS – 5

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

OR OPERATORS – 6

`OR, ||`	Logical OR.
`BITOR, |`	Bitwise OR.
`XOR`	Logical exclusive OR.
`BITXOR, ^`	Bitwise exclusive OR.

COMPARISON OPERATORS – 7

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

Description of operators

The following sections give detailed descriptions of each assembler operator. See *Assembler expressions*, page 3, for related information. The number within parentheses specifies the priority of the operator.

`*` Multiplication (3).

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

Example

```
2*2 → 4
-2*2 → -4
```

`+` Unary plus (1).

Unary plus operator.

Example

```
+3 → 3
3*+2 → 6
```

`+` Addition (4).

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

Example

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

– Unary minus (1).

The unary minus operator performs arithmetic negation on its operand.

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

Example

```
-3 → -3
3*-2 → -6
4--5 → 9
```

– Subtraction (4).

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

Example

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

/ Division (3).

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

Example

```
9/2 → 4
-12/3 → -4
9/2*6 → 24
```

AND, &&

Logical AND (5).

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

Example

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

BITAND, &

Bitwise AND (5).

Use & to perform bitwise AND between the integer operands.

Example

```
B'1010 & B'0011 → B'0010
B'1010 & B'0101 → B'0000
B'1010 & B'0000 → B'0OOO
```

BITNOT, ~

Bitwise NOT (1).

Use ~ to perform bitwise NOT on its operand.

Example

```
~ B'1010 → B'11111111111111111111111111110101
```

BITOR, |

Bitwise OR (6).

Use | to perform bitwise OR on its operands.

Example

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

BITXOR, ^

Bitwise exclusive OR (6).

Use ^ to perform bitwise XOR on its operands.

Example

```
B'1010 ^ B'0101 → B'1111
```

`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 time/date (1).

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

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

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

Example

To assemble the date of assembly:

```
today: DC8 DATE 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.

Example

```
1 = 2 → 0
2 == 2 → 1
'ABC' = 'ABCD' → 0
B’1010 ^ B’0011 → B’1001
```

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

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

Example

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

`GT, >` Greater than (7).

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

Example

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

`HIGH` High byte (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 numeric value that is lower than or equal to the right operand.

Example

```
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.

Example

```
-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 (3).

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

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

Example

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

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

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

Example

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

`NOT, !` Logical NOT (1).

Use `!` to negate a logical argument.

Example

```
! B’0101 → 0
! B’0000 → 1
```

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

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

Example

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

SFB Segment begin (1).

Syntax

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

Parameters

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

Description

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

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

Example

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

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

SFE Segment end (1).

Syntax

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

Parameters

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

Description

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

Example

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

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

The size of the segment MY_SEGMENT can be calculated as:

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

SHL, << Logical shift left (3).

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

Example

```
B'00011100 << 3 → B'11100000
B'00000111111111111 << 5 → B'11111111111100000
14 << 1 → 28
```

SHR, >> Logical shift right (3).

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

Example

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

`SIZEOF` Segment size (1).

Syntax

SIZEOF *segment*

Parameters

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

Description

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

Example

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

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

`UGT` Unsigned greater than (7).

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

Example

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

`ULT` Unsigned less than (7).

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

Example

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

`XOR` Logical exclusive OR (6).

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

Example

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

Assembler directives

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

Summary of assembler directives

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

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

Table 16: Assembler directives summary

Directive	Description	Section
CASEON	Enables case sensitivity.	Assembler control
CFI	Specifies call frame information.	Call frame information
COL	Sets the number of columns per page.	Listing control
COMMON	Begins a common segment.	Segment control
DB	Generates 8-bit byte constants, including strings.	Data definition or allocation
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
DEFINE	Defines a file-wide value.	Value assignment
DP	Generates 24-bit word constants.	Data definition or allocation
DS	Allocates space for 8-bit bytes.	Data definition or allocation
DS16	Allocates space for 16-bit words.	Data definition or allocation
DS24	Allocates space for 24-bit words.	Data definition or allocation
DS32	Allocates space for 32-bit words.	Data definition or allocation
DS8	Allocates space for 8-bit bytes.	Data definition or allocation
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

Table 16: Assembler directives summary (Continued)

Directive	Description	Section
END	Terminates the assembly of the last module in a file.	Module control
ENDIF	Ends an IF block.	Conditional assembly
ENDM	Ends a macro definition.	Macro processing
ENDMOD	Terminates the assembly of the current module.	Module control
ENDR	Ends a repeat structure	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment
EVEN	Aligns the program counter to an even address.	Segment control
EXITM	Exits prematurely from a macro.	Macro processing
EXPORT	Exports symbols to other modules.	Symbol control
EXTERN	Imports an external symbol.	Symbol control
EXTRN	Imports an external symbol.	Symbol control
IF	Assembles instructions if a condition is true.	Conditional assembly
IMPORT	Imports an external symbol.	Symbol control
LIBRARY	Begins a library module.	Module control
LIMIT	Checks a value against limits.	Value assignment
LOCAL	Creates symbols local to a macro.	Macro processing
LSTCND	Controls conditional assembler listing.	Listing control
LSTCOD	Controls multi-line code listing.	Listing control
LSTEXP	Controls the listing of macro generated lines.	Listing control
LSTMAC	Controls the listing of macro definitions.	Listing control
LSTOUT	Controls assembler-listing output.	Listing control
LSTPAG	Controls the formatting of output into pages.	Listing control
LSTREP	Controls the listing of lines generated by repeat directives.	Listing control
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

Table 16: Assembler directives summary (Continued)

Directive	Description	Section
PAGSIZ	Sets the number of lines per page.	Listing control
PROGRAM	Begins a program module.	Module control
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.	Symbol control
PUBLIC	Exports symbols to other modules.	Symbol control
RADIX	Sets the default base.	Assembler control
REPT	Assembles instructions a specified number of times.	Macro processing
REPTC	Repeats and substitutes characters.	Macro processing
REPTI	Repeats and substitutes strings.	Macro processing
REQUIRE	Forces a symbol to be referenced.	Symbol control
RSEG	Begins a relocatable segment.	Segment control
RTMODEL	Declares runtime model attributes.	Module control
SFRB	Creates byte-access SFR labels.	Value assignment
SFRTYPE	Specifies SFR attributes.	Value assignment
SFRW	Creates word-access SFR labels.	Value assignment
STACK	Begins a stack segment.	Segment control
VAR	Assigns a temporary value.	Value assignment

Table 16: Assembler directives summary (Continued)

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

Syntax conventions

In the syntax definitions the following conventions are used:

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

  `ORG` *`expr`*

  `expr` represents an arbitrary expression.

- Optional parameters are shown in square brackets. So, for example, in:
  `END [expr]`

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

  `PUBLIC symbol [,symbol] …`

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

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

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

LABELS AND COMMENTS

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

`label 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 3.
label	A symbolic label.
symbol	An assembler symbol.

Table 17: Assembler directive parameters

Module control directives

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

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

Table 18: Module control directives

SYNTAX

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

PARAMETERS

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

DESCRIPTION

Beginning a program module

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

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

Beginning a library module

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

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

Terminating a module

Use ENDMOD to define the end of a module.

Terminating the last module

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

Assembling multi-module files

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

The following rules apply when assembling multi-module files:

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

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

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

Declaring runtime model attributes

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

A module can have several runtime model definitions.

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

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

Examples

The following example defines three modules where:

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

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

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

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

Symbol control directives

These directives control how symbols are shared between modules.

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

Table 19: Symbol control directives

SYNTAX

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

PARAMETERS

symbol	Symbol to be imported or exported.

DESCRIPTION

Exporting symbols to other modules

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

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

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

Exporting symbols with multiple definitions to other modules

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

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

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

Importing symbols

Use EXTERN to import an untyped external symbol.

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

EXAMPLES

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

```
     NAME   error
     EXTERN print
     PUBLIC err

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

     END
```

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.
ASEGN	Begins a named absolute segment.
COMMON	Begins a common segment.
EVEN	Aligns the program counter to an even address.

Table 20: Segment control directives

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

Table 20: Segment control directives (Continued)

SYNTAX

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

PARAMETERS

address	Address where this segment part will be placed.
align	Exponent of the value to which the address should be aligned, in the range 0 to 30.
expr	Address to set the location counter to.
flag	NOROOT, ROOT NOROOT means that the segment part may be discarded by the linker if no symbols in this segment part are referred to. Normally all segment parts except startup code and interrupt vectors should set this flag. The default mode is ROOT which indicates that the segment part must not be discarded. REORDER 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 that no reordering is performed. SORT SORT means that the linker will sort the segment parts in decreasing alignment order. For a given segment, all segment parts must specify the same state for this flag. The default mode is that no sorting is performed.

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

DESCRIPTION

Beginning an absolute segment

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

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

Beginning a relocatable segment

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

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

Beginning a stack segment

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

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

Beginning a common segment

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

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

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

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

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

Setting the program location counter (PLC)

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

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

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

Aligning a segment

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

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

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

EXAMPLES

Beginning an absolute segment

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

```
EXTERN  EINT1, EINT2, RESET

ASEG    INTVEC
ORG     0h

RJMP    RESET
RJMP    EINT1
RJMP    EINT2
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 for creating a gap of six bytes in the table.

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

```
   EXTERN    Table1,Table2

   RSEG TABLES
   DC16 Table1, Table2

   ORG  $+6
   DC16 Table3

   RSEG CONST

Table3 DC8  1,2,4,8,16,32
   END
```

Beginning a stack segment

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

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

The data is allocated from high to low addresses.

Beginning a common segment

The following example defines two common segments containing variables:

```
         NAME    common1
         COMMON  data
count    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  DC16    1        ; target and best will be on
                         ; an even boundary
best    DC16    1
        ALIGN   6        ; Now align to a 64 byte boundary
results DS8     64       ; And create a 64 byte table
        END
```

Value assignment directives

These directives are used for assigning values to symbols.

Directive	Description
`=`	Assigns a permanent value local to a module.
`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.
`SFRB`	Creates byte-access SFR labels.
`SFRTYPE`	Specifies SFR attributes.
`SFRW`	Creates word-access SFR labels.
`VAR`	Assigns a temporary value.

Table 21: Value assignment directives

SYNTAX

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

```
[const] SFRB register = value
[const] SFRTYPE register attribute [,attribute] = value
[const] SFRW register = value
label VAR expr
```

PARAMETERS

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

DESCRIPTION

Defining a temporary value

Use either of `ASSIGN` and `VAR` to define a symbol that may be redefined, such as for use with macro variables. Symbols defined with `VAR` 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 SFRB to create special function register labels with attributes READ, WRITE, and BYTE turned on. Use SFRW to create special function register labels with attributes READ, WRITE, or WORD turned on. Use SFRTYPE to create special function register labels with specified attributes.

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

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 VAR to redefine the symbol cons in a REPT loop to generate a table of the first 8 powers of 3:

```
         NAME     table
cons     VAR      1
buildit  MACRO    times
         DC16     cons
cons     VAR      cons*3
         IF       times>1
         buildit  times-1
         ENDIF
         ENDM
main     buildit 4
         END
```

It generates the following code:

```
 1    00000000                      NAME    table
 2    00000001              cons    VAR     1
10    00000000              main    buildit 4
10.1  00000000 0001                 DC16    cons
10.2  00000003              cons    VAR     cons*3
10.3  00000002                      IF      4>1
10    00000002                      buildit 4-1
10.1  00000002 0003                 DC16    cons
10.2  00000009              cons    VAR     cons*3
10.3  00000004                      IF      4-1>1
10    00000004                      buildit 4-1-1
10.1  00000004 0009                 DC16    cons
10.2  0000001B              cons    VAR     cons*3
10.3  00000006                      IF      4-1-1>1
10    00000006                      buildit 4-1-1-1
10.1  00000006 001B                 DC16    cons
10.2  00000051              cons    VAR     cons*3
10.3  00000008                      IF      4-1-1-1>1
10.4  00000008                      buildit 4-1-1-1-1
10.5  00000008                      ENDIF
10.6  00000008                      ENDM
10.7  00000008                      ENDIF
10.8  00000008                      ENDM
10.9  00000008                      ENDIF
10.10 00000008                      ENDM
10.11 00000008                      ENDIF
10.12 00000008                      ENDM
11    00000008                      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    R27
         LDI    R26,locn
         LD     R16,X
         LDI    R17,value
         ADD    R16,R17
         RET
         ENDMOD
```

```
        NAME   add2
value   EQU    88
        CLR    R27
        LDI    R26,locn
        LD     R16,X
        LDI    R17,value
        ADD    R16,R17
        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 */
        SFRW ocr1                 = 0x2A    /* word 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   VAR       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 22: 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 `IF` directive must be terminated by an `ENDIF` directive. The `ELSE` directive is optional, and if used, it must be inside an `IF...ENDIF` block. `IF...ENDIF` and `IF...ELSE...ENDIF` blocks may be nested to any level.

EXAMPLES

The following macro subtracts a constant from the register pair `R25:R24`.

```
subW MACRO  c
     IF     c<64
     SBIW   R25:R24,c
     ELSE
     SUBI   R24,LOW(c)
```

```
     SBCI   R25,c >> 8
     ENDIF
     ENDM
```

If the argument to the macro is less than 64, it is possible to use the `SBIW` instruction to save two bytes of code memory.

It could be tested with the following program:

```
main LDI    R24,0
     LDI    R25,0
     subW   16
     LDI    R24,0
     LDI    R25,0
     subW   75
     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 23: 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.

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
        EVEN
        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
                EVEN
```

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

The previous example could therefore be written as follows:

```
errmac  MACRO
        CALL    abort
        DB      \1,0
        EVEN
        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
         LDI    op
         ENDM
```

The macro can be called using the macro quote characters:

```
         macld  <R16, 1>
         END
```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 20.

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  AAVR_MAN

DO_LPM  MACRO
     IF _args == 2
       LPM  \1,\2
     ELSE
       LPM
     ENDIF
     ENDM

     RSEG      CODE

     DO_LPM
     DO_LPM  R16,Z+

END
```

The following listing is generated:

```
   1    00000000                      MODULE  AAVR_MAN
   2    00000000
  10    00000000
  11    00000000                      RSEG    CODE
  12    00000000
  13    00000000                      DO_LPM
  13.1  00000000                      IF _args == 2
  13.2  00000000                      LPM     ,
  13.3  00000000                      ELSE
  13.4  00000000 95C8                 LPM
  13.5  00000002                      ENDIF
  13.6  00000002                      ENDM
  14    00000002                      DO_LPM  R16,Z+
  14.1  00000002                      IF _args == 2
  14.2  00000002 9105                 LPM     R16,Z+
  14.3  00000004                      ELSE
  14.4  00000004                      LPM
  14.5  00000004                      ENDIF
  14.6  00000004                      ENDM
  15    00000004
  16    00000004                      END
```

How macros are processed

There are three distinct phases in the macro process:

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

   The macro expander has no knowledge of assembler symbols since it only deals with text substitutions at source level. Before a line from the called macro definition is handed over to the assembler, the expander scans the line for all occurrences of symbolic macro arguments, and replaces them with their expansion arguments.
3. The expanded line is then processed as any other assembler source line. The input stream to the assembler will continue to be the output from the macro processor, until all lines of the current macro definition have been read.

Repeating statements

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

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

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

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

EXAMPLES

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

Coding in-line for efficiency

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

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

```
        NAME    play

portb   VAR     0x18
        RSEG    DATA
buffer  DS      256

        RSEG    CODE
play    LDI     R27,HIGH(buffer)
        LDI     R26,LOW(buffer)
        LDI     R25,255
loop    LD      R0,X+
        OUT     portb,R0
        DEC     R25
        BRNE    loop
        RET
        END
```

The main program calls this routine as follows:

```
doplay  CALL    play
```

For efficiency we can recode this using a macro:

```
        NAME    play

portb   VAR     0x18
        RSEG    DATA
buffer  DS      256

play    MACRO
        LOCAL   loop
        LDI     R27,HIGH(buffer)
        LDI     R26,LOW(buffer)
        LDI     R25,255
loop    LD      R0,X+
        OUT     portb,R0
        DEC     R25
        BRNE    loop
        ENDM

        RSEG    CODE
        play
        END
```

Note 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"
        LDI     R16,'chr'
        CALL    plotc
        ENDR

        END
```

This produces the following code:

```
  1    00000000                        NAME    reptc
  2    00000000
  3    00000000                        EXTERN  plotc
  4    00000000
  5    00000000                 banner REPTC   chr, "Welcome"
  6    00000000                        LDI     R16,'chr'
  7    00000000                        RCALL   plotc
  8    00000000                        ENDR
  8.1  00000000 E507                   LDI     R16,'W'
  8.2  00000002 ....                   RCALL   plotc
  8.3  00000004 E605                   LDI     R16,'e'
  8.4  00000006 ....                   RCALL   plotc
  8.5  00000008 E60C                   LDI     R16,'l'
  8.6  0000000A ....                   RCALL   plotc
  8.7  0000000C E603                   LDI     R16,'c'
  8.8  0000000E ....                   RCALL   plotc
  8.9  00000010 E60F                   LDI     R16,'o'
  8.10 00000012 ....                   RCALL   plotc
  8.11 00000014 E60D                   LDI     R16,'m'
  8.12 00000016 ....                   RCALL   plotc
  8.13 00000018 E605                   LDI     R16,'e'
  8.14 0000001A ....                   RCALL   plotc
  9    0000001C
 10    0000001C                        END
```

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

```
        NAME    repti

        EXTERN base, count, init
```

```
banner   REPTI  adds, base, count, init
         LDI    R30,LOW(adds)
         LDI    R31,HIGH(adds)
         LDI    R16,0
         STD    Z+0,R16
         ENDR

         END
```

This produces the following code:

```
   1    00000000                      NAME    reptc
   2    00000000
   3    00000000                      EXTERN  adds, base, count, init
   4    00000000
   5    00000000              banner  REPTI   adds, base, count, init
   6    00000000                      LDI     R30,LOW(adds)
   7    00000000                      LDI     R31,adds >> 8
   8    00000000                      LDI     R16,0
   9    00000000                      ST      Z,R16
  10    00000000                      STD     Z+1,R16
  11    00000000                      ENDR
  11.1  00000000 ....                 LDI     R30,LOW( base)
  11.2  00000002 ....                 LDI     R31, base >> 8
  11.3  00000004 E000                 LDI     R16,0
  11.4  00000006 8300                 ST      Z,R16
  11.5  00000008 8301                 STD     Z+1,R16
  11.6  0000000A ....                 LDI     R30,LOW( count)
  11.7  0000000C ....                 LDI     R31, count >> 8
  11.8  0000000E E000                 LDI     R16,0
  11.9  00000010 8300                 ST      Z,R16
  11.10 00000012 8301                 STD     Z+1,R16
  11.11 00000014 ....                 LDI     R30,LOW( init)
  11.12 00000016 ....                 LDI     R31, init >> 8
  11.13 00000018 E000                 LDI     R16,0
  11.14 0000001A 8300                 ST      Z,R16
  11.15 0000001C 8301                 STD     Z+1,R16
  12    0000001E
  13    0000001E                      END
```

Listing control directives

These directives provide control over the assembler list file.

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

Table 24: Listing control directives

SYNTAX

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

PARAMETERS

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

DESCRIPTION

Turning the listing on or off

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

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

Listing conditional code and strings

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

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

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

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

Controlling the listing of macros

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

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

Controlling the listing of generated lines

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

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

Generating a cross-reference table

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

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

Specifying the list file format

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

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

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

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

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

EXAMPLES

Turning the listing on or off

To disable the listing of a debugged section of program:

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

Listing conditional code and strings

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

```
        NAME    lstcndtst
        EXTERN  print

        RSEG    prom

debug   VAR     0
begin   IF      debug
        CALL    print
        ENDIF

        LSTCND+
begin2  IF      debug
        CALL    print
        ENDIF

        END
```

This will generate the following listing:

```
 1   00000000                     NAME    lstcndtst
 2   00000000                     EXTERN  print
 3   00000000
 4   00000000                     RSEG    CODE
 5   00000000
 6   00000000             debug   VAR     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    00000000                      NAME    lstcodtst
  2    00000000                      EXTERN  print
  3    00000000
  4    00000000                      RSEG    CONST
  5    00000000
  6    00000000 00010000000A*table1: DD      1,10,100,1000,10000
  7    00000014
  8    00000014                      LSTCOD+
  9    00000014 00010000000A table2: DD      1,10,100,1000,10000
                000000640000
                03E800002710
                0000
 10    00000028
 11    00000028                      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   R16

        LSTEXP-
        inc2   R17
        RET
        END    begin
```

This will produce the following output:

```
5     00000000
6     00000000                      LSTMAC+
7     00000000              inc2    MACRO   arg
8     00000000                      INC     arg
9     00000000                      INC     arg
10    00000000                      ENDM
11    00000000
12    00000000              begin:
13    00000000                      dec2    R16
13.1  00000000 950A                 DEC     R16
13.2  00000002 950A                 DEC     R16
13.3  00000004                      ENDM
14    00000004
15    00000004                      LSTEXP-
16    00000004                      inc2    R17
17    00000008 9508                 RET
18    0000000A
19    0000000A                      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.

Table 25: C-style preprocessor directives

Directive	Description
`#endif`	Ends a `#if`, `#ifdef`, or `#ifndef` block.
`#error`	Generates an error.
`#if`	Assembles instructions if a condition is true.
`#ifdef`	Assembles instructions if a symbol is defined.
`#ifndef`	Assembles instructions if a symbol is undefined.
`#include`	Includes a file.
`#message`	Generates a message on standard output.
`#undef`	Undefines a label.

Table 25: C-style preprocessor directives

SYNTAX

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

PARAMETERS

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

`message`	Text to be displayed.
`text`	Value to be assigned.

DESCRIPTION

Defining and undefining labels

Use `#define` to define a temporary label.

`#define` *`label value`*

is similar to:

`label` `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.

1 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:

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

```
#define five 5 ; this comment is not ok
#define six 6 // this comment is ok
#define seven 7 /* this comment is ok */

LDS     five,R5         ; syntax error!
; expands to "LDS     5 ; this comment is not ok,R5"

LDS     R16,five + 2        ; incorrect code!
; expands to "LDS     R16,5 ; this comment is not ok + 2"

STS     six+seven,R5 ; ok
; expands to "STS     6+7,R5"
```

EXAMPLES

Using conditional directives

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

```
#define tweak 1
#define adjust 3
```

```
#ifdef  tweak
#if     adjust=1
        SUBI    R16,4
#elif   adjust=2
        SUBI    R16,20
#elif   adjust=3
        SUBI    R16,30
#endif
#endif                            /* ifdef tweak */
```

Including a source file

The following example uses `#include` to include a file defining macros into the source file. For example, the following macros could be defined in `Macros.s90`:

```
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.s90"

; program
main:   xch     R16,R17
        RET
        END main
```

Data definition or allocation directives

These directives define values or reserve memory:

Directive	Description	Expression restrictions
`DC8, DB`	Generates 8-bit constants, including strings.	
`DC16, DW`	Generates 16-bit constants.	
`DC24, DP`	Generates 24-bit constants.	
`DC32, DD`	Generates 32-bit constants.	
`DS8, DS`	Allocates space for 8-bit integers.	No external references; Absolute

Table 26: Data definition or allocation directives

Directive	Description	Expression restrictions
`DS16`	Allocates space for 16-bit integers.	No external references; Absolute
`DS24`	Allocates space for 24-bit integers.	No external references; Absolute
`DS32`	Allocates space for 32-bit integers.	No external references; Absolute

Table 26: Data definition or allocation directives (Continued)

SYNTAX

```
DB expr
DC8 expr [,expr] ...
DC16 expr [,expr] ...
DC24 expr [,expr] ...
DC32 expr [,expr] ...
DD expr [,expr]
DP expr [,expr]
DS expr [,expr]
DS8 expr [,expr] ...
DS16 expr [,expr] ...
DS24 expr [,expr] ...
DS32 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 data size implied by the directive. Double-quoted strings will be zero-terminated.

DESCRIPTIONS

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

Size	Reserve and initialize memory	Reserve unitialized memory
8-bit integers	DC8, DB	DS8, DS
16-bit integers	DC16, DW	DS16
24-bit integers	DC24, DP	DS24
32-bit integers	DC32, DD	DS32

Table 27: Using data definition or allocation directives

EXAMPLES

Generating lookup table

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

```
        NAME    table
        RSEG    CONST
table   DW      addsubr/2, subsubr/2, clrsubr/2
        RSEG    CODE
addsubr ADD     R16,R17
        RET
subsubr SUB     R16,R17
        RET
clrsubr CLR     R16
        RET

        END
```

Note: In the AVR architecture, code addresses are word addresses and in the AVR IAR Assembler, labels are byte addresses. This implies that a function pointer must be divided by two before it is issued to `ICALL`, `EICALL`, `IJMP`, or `EIJMP`. This can be done either in the table or with instructions before the jump/call instruction.

Defining strings

To define a string:

```
mymsg   DC8 'Please enter your name'
```

To define a string which includes a trailing zero:

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

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

```
errmsg  DC8 'Don''t understand!'
```

Reserving space

To reserve space for `0xA` bytes:

```
table   DS8   0xA
```

Assembler control directives

These directives provide control over the operation of the assembler.

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

Table 28: Assembler control directives

SYNTAX

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

PARAMETERS

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

DESCRIPTION

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

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

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

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

Controlling case sensitivity

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

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

EXAMPLES

Including a source file

The following example uses $ to include a file defining macros into the source file. For example, the following macros could be defined in Mymacros.s90:

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

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

```
        NAME    include

; standard macro definitions

$mymacros.s90

; program
main
        xch     R16,R17
        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: 19.12.01
Author: mjp
*/
```

Changing the base

To set the default base to 16:

```
        RADIX  D'16
        LDI    R16,12
```

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

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

```
RADIX  0x0A
```

Controlling case sensitivity

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

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

The following will generate a duplicate label error:

```
        CASEOFF

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

        END
```

Call frame information directives

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

Directive	Description
CFI BASEADDRESS	Declares a base address CFA (Canonical Frame Address).
CFI BLOCK	Starts a data block.
CFI CODEALIGN	Declares code alignment.
CFI COMMON	Starts or extends a common block.
CFI CONDITIONAL	Declares data block to be a conditional thread.
CFI DATAALIGN	Declares data alignment.
CFI ENDBLOCK	Ends a data block.
CFI ENDCOMMON	Ends a common block.
CFI ENDNAMES	Ends a names block.
CFI FRAMECELL	Creates a reference into the caller's frame.
CFI FUNCTION	Declares a function associated with data block.
CFI INVALID	Starts range of invalid backtrace information.
CFI NAMES	Starts a names block.
CFI NOFUNCTION	Declares data block to not be associated with a function.
CFI PICKER	Declares data block to be a picker thread.

Table 29: Call frame information directives

Directive	Description
`CFI REMEMBERSTATE`	Remembers the backtrace information state.
`CFI RESOURCE`	Declares a resource.
`CFI RESOURCEPARTS`	Declares a composite resource.
`CFI RESTORESTATE`	Restores the saved backtrace information state.
`CFI RETURNADDRESS`	Declares a return address column.
`CFI STACKFRAME`	Declares a stack frame CFA.
`CFI STATICOVERLAYFRAME`	Declares a static overlay frame CFA.
`CFI VALID`	Ends range of invalid backtrace information.
`CFI VIRTUALRESOURCE`	Declares a virtual resource.
`CFI` *`cfa`*	Declares the value of a CFA.
`CFI` *`resource`*	Declares the value of a resource.

Table 29: Call frame information directives (Continued)

SYNTAX

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

Names block directives

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

Extended names block directives

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

Common block directives

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

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

Extended common block directives

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

Data block directives

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

PARAMETERS

bits	The size of the resource in bits.
cell	The name of a frame cell.
cfa	The name of a CFA (canonical frame address).
cfiexpr	A CFI expression (see *CFI expressions*, page 93).
codealignfactor	The smallest factor of all instruction sizes. Each CFI directive for a data block must be placed according to this alignment. 1 is the default and can always be used, but a larger value will shrink the produced backtrace information in size. The possible range is 1–256.
commonblock	The name of a previously defined common block.
constant	A constant value or an assembler expression that can be evaluated to a constant value.

dataalignfactor	The smallest factor of all frame sizes. If the stack grows towards higher addresses, the factor is negative; if it grows towards lower addresses, the factor is positive. 1 is the default, but a larger value will shrink the produced backtrace information in size. The possible ranges are -256 – -1 and 1 – 256.
label	A function label.
name	The name of the block.
namesblock	The name of a previously defined names block.
offset	The offset relative the CFA. An integer with an optional sign.
part	A part of a composite resource. The name of a previously declared resource.
resource	The name of a resource.
segment	The name of a segment.
size	The size of the frame cell in bytes.
type	The memory type, such as `CODE`, `CONST` or `DATA`. In addition, any of the memory types supported by the IAR XLINK Linker. It is used solely for the purpose of denoting an address space.

DESCRIPTIONS

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

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

Backtrace rows and columns

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

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

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

Defining a names block

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

Start and end a names block with the directives:

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

where *name* is the name of the block.

Only one names block can be open at a time.

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

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

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

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

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

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

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

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

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

  `CFI STACKFRAME` *`cfa resource type`*

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

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

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

  `CFI STATICOVERLAYFRAME` *`cfa segment`*

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

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

  `CFI BASEADDRESS` *`cfa type`*

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

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

Extending a names block

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

Extend an existing names block with the directive:

`CFI NAMES` *`name`* `EXTENDS` *`namesblock`*

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

`CFI ENDNAMES` *`name`*

Defining a common block

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

Start a common block with the directive:

```
CFI COMMON name USING namesblock
```

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

Declare the return address column with the directive:

```
CFI RETURNADDRESS resource type
```

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

End a common block with the directive:

```
CFI ENDCOMMON name
```

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

Inside a common block you can declare the initial value of a CFA or a resource by using the directives listed last in *Common block directives*, page 85. For more information on these directives, see *Simple rules*, page 91, and *CFI expressions*, page 93.

Extending a common block

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

Extend an existing common block with the directive:

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

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

```
CFI ENDCOMMON name
```

Defining a data block

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

Start a data block with the directive:

```
CFI BLOCK name USING commonblock
```

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

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

```
CFI FUNCTION label
```

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

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

```
CFI NOFUNCTION
```

End a data block with the directive:

```
CFI ENDBLOCK name
```

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

Inside a data block you may manipulate the values of the columns by using the directives listed last in *Data block directives*, page 86. For more information on these directives, see *Simple rules*, page 91, and *CFI expressions*, page 93.

SIMPLE RULES

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

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

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

In those rare cases where the descriptive power of the simple rules are not enough, a full CFI expression can be used to describe the information (see *CFI expressions*, page 93). However, whenever possible, you should always use a simple rule instead of a CFI expression.

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

Simple rules for resources

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

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

```
CFI REG SAMEVALUE
```

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

```
CFI REG UNDEFINED
```

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

```
CFI REG1 REG2
```

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

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

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

```
CFI RET CONCAT
```

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

Simple rules for CFAs

In contrast with the rules for resources, the rules for CFAs describe the address of the beginning of the call frame. The call frame often includes the return address pushed by the subroutine calling instruction. The CFA rules describe how to compute the address to the beginning of the current call frame. There are two different forms of CFAs, stack frames and static overlay frames, each declared in the associated names block. See *Names block directives*, page 85.

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

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

```
CFI CFA_SP NOTUSED
```

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

```
CFI CFA_SP SP + 4
```

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

CFI EXPRESSIONS

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

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

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

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

Unary operators

Overall syntax: `OPERATOR(operand)`

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

Table 30: Unary operators in CFI expressions

Binary operators

Overall syntax: `OPERATOR(operand1,operand2)`

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

Table 31: Binary operators in CFI expressions

Operator	Operands	Description
RSHIFTA	*cfiexpr,cfiexpr*	Arithmetic shift right of the left operand. The number of bits to shift is specified by the right operand. In contrast with RSHIFTL the sign bit will be preserved when shifting.

Table 31: Binary operators in CFI expressions (Continued)

Ternary operators

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

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

Table 32: Ternary operators in CFI expressions

EXAMPLE

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

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

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

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

Table 33: Code sample with backtrace rows and columns

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

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

Defining the names block

The names block for the small example above would be:

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

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

Defining the common block

The common block for the simple example above would be:

```
CFI COMMON trivialCommon USING trivialNames
CFI RETURNADDRESS RET DATA
```

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

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

Defining the data block

Continuing the simple example, the data block would be:

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

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

Diagnostics

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

Message format

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

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

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

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

Severity levels

The diagnostics are divided into different levels of severity:

Warning

A diagnostic message that is produced when the assembler finds a programming error or omission which is of concern but not so severe as to prevent the completion of compilation. Warnings can be disabled by use of the command-line option `-w`, see page 21.

Error

A diagnostic message that is produced when the assembler has found a construct which clearly violates the language rules, such that code cannot be produced.

Fatal error

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

INTERNAL ERROR

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

`Internal error: message`

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

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

A

B

C

D

E

F

M

N

O

P

R

S