AVR® IAR ASSEMBLER, IAR XLINK LINKER™, AND IAR XLIB LIBRARIAN™

Reference Guide

for Atmel® Corporation's

AVR® Microcontroller

COPYRIGHT NOTICE

© Copyright 2000 IAR Systems. All rights reserved.

No part of this document may be reproduced without the prior written consent of IAR Systems. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

DISCLAIMER

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

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

TRADEMARKS

IAR and C-SPY are registered trademarks of IAR Systems. IAR Embedded Workbench, IAR XLINK Linker, and IAR XLIB Librarian are trademarks of IAR Systems. Atmel and AVR are registered trademarks of Atmel Corporation. Microsoft is a registered trademark, and Windows is a trademark of Microsoft Corporation.

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

First edition: March 2000

Part number: AAVR-1

WELCOME

Welcome to the AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide.

This guide provides reference information about the IAR Systems Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ for the Atmel AVR microcontroller.

Before reading this guide we recommend you to read the initial chapters of the *AVR IAR Embedded Workbench™ User Guide*, where you will find information about installing the IAR Systems development tools, product overviews, and tutorials that will help you get started. The *AVR IAR Embedded Workbench™ User Guide* also contains complete reference information about the IAR Embedded Workbench and the IAR C-SPY® Debugger.

For information about programming with the AVR IAR Compiler, refer to the *AVR IAR Compiler Reference Guide*.

Refer to the chip manufacturer's documentation for information about the AVR architecture and instruction set.

If you want to know more about IAR Systems, visit the website **www.iar.com** where your will find company information, product news, technical support, and much more.

ABOUT THIS GUIDE

This guide consists of the following parts:

◆ ***Part 1: The AVR IAR Assembler***

Introduction to the AVR IAR Assembler provides programming information. It also describes the source code format, and the format of assembler listings.

Assembler options first explains how to set the assembler options and how to use environment variables. It then gives an alphabetical summary of the assembler options, and contains complete reference information about each option.

Assembler operators gives a summary of the assembler operators, arranged in order of precedence, and provides complete reference information about each operator.

Assembler directives gives an alphabetical summary of the assembler directives, and provides complete reference information about each of the directives, classified into groups according to their function.

Assembler diagnostics provides a list of error and warning messages specific to the AVR IAR Assembler.

- ***Part 2: The IAR XLINK Linker***

  Introduction to the IAR XLINK Linker describes the IAR XLINK Linker, and gives examples of how it can be used. It also explains the XLINK listing format.

  XLINK options describes how to set the XLINK options, gives an alphabetical summary of the options, and provides detailed information about each option.

  XLINK output formats summarizes the output formats available from XLINK.

  XLINK environment variables gives reference information about the IAR XLINK Linker environment variables.

  XLINK diagnostics describes the error and warning messages produced by the IAR XLINK Linker.

- ***Part 3: The IAR XLIB Librarian***

  Introduction to the IAR XLIB Librarian describes the IAR XLIB Librarian, which is designed to allow you to create and maintain relocatable libraries of routines.

  XLIB options gives a summary of the XLIB commands, and complete reference information about each command.

  XLIB environment variables gives reference information about the IAR XLIB Librarian environment variables.

  XLIB diagnostics describes the error and warning messages produced by the IAR XLIB Librarian.

ASSUMPTIONS AND CONVENTIONS

ASSUMPTIONS

This guide assumes that you already have a working knowledge of the following:

- General assembly language programming.
- The architecture of the AVR microcontroller.
- The instruction set of the AVR microcontroller.

  Refer to the chip manufacturer's documentation for information about the assembler instructions.
- Windows 95/98 or Windows NT, depending on your host system.

CONVENTIONS

This guide uses the following typographical conventions:

Style	*Used for*
`computer`	Text that you type in, or that appears on the screen.
`parameter`	A label representing the actual value you should type 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	Cross-references to another part of this guide, or to another guide.
🛠	Identifies instructions specific to the versions of the IAR Systems tools for the IAR Embedded Workbench interface.

CONTENTS

PART 1: THE AVR IAR ASSEMBLER

This part of the AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide includes the following chapters:

- *Introduction to the AVR IAR Assembler*
- *Assembler options*
- *Assembler operators*
- *Assembler directives*
- *Assembler diagnostics*.

INTRODUCTION TO THE AVR IAR ASSEMBLER

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

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

SOURCE FORMAT

The format of an assembler source line is as follows:

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

where the components are as follows:

`label`	A label, which is assigned the value and type of the current program location counter (PLC). The `:` (colon) is optional if the label starts in the first column.
`operation`	An assembler instruction or directive. This must not start in the first column.
`operands`	An assembler instruction can have zero, one, or 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 92. Other assembler directives can have one, two, or three operands, separated by commas.
`comment`	Comment, preceded by a `;` (semicolon).

The fields can be separated by spaces or tabs.

A source line may not exceed 2047 characters.

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

ASSEMBLER EXPRESSIONS

Expressions can consist of operands and operators.

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

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

The following operands are valid in an expression:

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

These are described in greater detail in the following sections.

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

TRUE AND FALSE

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

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

USING SYMBOLS IN RELOCATABLE EXPRESSIONS

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

Such expressions are evaluated and resolved at link time, by the IAR XLINK Linker™. There are no restrictions on the expression; any operator can be used on symbols from any segment, or any combination of segments. For example, a program could define the segments `DATA` and `CODE` as follows:

```
        NAME    prog1
        EXTERN  third
        RSEG    DATA
first:  DB      5
second: DB      3
```

```
        ENDMOD
        MODULE  prog2
        RSEG    CODE
start   …
```

Then in the segment `CODE` the following instructions 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 page 32 for additional information.

Notice that symbols and labels are byte addresses. For additional information, see *Generating lookup table*, page 94.

LABELS

Symbols used for memory locations are referred to as labels.

Program location counter (PLC)

The program location counter is called `$`. 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 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’`

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

ASCII CHARACTER CONSTANTS

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

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

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.

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:
`__TIME__`	Current time in `hh:mm:ss` format (string).
`__VER__`	Version number in integer format; for example, version 4.17 is returned as 417 (number).

Values of `__TID__`:

Processor option	*Value*
`-v0`	`0x5A00`
`-v1`	`0x5A10`
`-v2`	`0x5A20`
`-v3`	`0x5A30`
`-v4`	`0x5A40`
`-v5`	`0x5A50`
`-v6`	`0x5A60`

Notice that `__TID__` is related to the predefined symbol `__TID__` in the AVR IAR Compiler. It is described in the chapter *Predefined symbols reference* in the *AVR IAR Compiler Reference Guide*. For detailed information about the `-v` processor option, see the chapter *Configuration* in the *AVR IAR Compiler Reference Guide*.

Including symbol values in code

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

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

```
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 use one of the conditional assembly directives.

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
```

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

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.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of AVR derivatives are included in the IAR product package. The header files are named `io`*`chip`*`.h`, for example `io2313.h`, and define the processor-specific special function registers (SFRs).

Since the header files are also intended to be used with the AVR C Compiler, `ICCAVR`, the SFR declarations are made with macros. The macros that convert the declaration to assembler or compiler syntax are defined in the `iomacro.h` file.

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

Example

The EEPROM address register at I/O address `0x1E` of the AT90mega103 microcontroller derivative is defined in the `iom103.h` file as:

```
SFR_W(EEAR,    0x1E)     /* EEPROM Address register */
```

The declaration is converted by macros defined in the file `iomacro.h` to:

```
sfrw   EEAR  = 0x1E
sfrb   EEARL = 0x1E
sfrb   EEARH = 0x1F
```

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 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 11, 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 segment name:CODE: segment flags	1
.DB *data1*,*data2*,*data3*	DB *data1*,*data2*,*data3*	

Atmel AVR Assembler format	*AVR IAR Assembler format*	*Comments*
`.DEF` *`name = value`*	`#define` *`name value`*	2
`.DSEG`	`RSEG` segment name:`DATA`:segment flags	1
`.DW` *`data1,data2,data3`*	`DW` *`data1,data2,data3`*	
`.ENDMACRO`	`ENDM`	
`.EQU` *`label = expression`*	*`label`* `EQU` *`expression`*	
`.ESEG`	`RSEG` segment name:`XDATA`:segment flags	1
`.INCLUDE` *`file`*	`#include` *`file`*	2
`.LIST`	`LSTOUT+`	
`.LISTMAC`	`LSTEXP+`	
`.MACRO` *`macroname`*	*`macroname`* `MACRO` *`arguments…`*	3
`.NOLIST`	`LSTOUT-`	
`.ORG` *`expression`*	`ORG` *`expression`*	
`.SET` *`label = expression`*	*`label`* `VAR` *`expression`*	

Comments:

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 is being assembled. Refer to the *AVR IAR 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
```

Notice that, in the Atmel 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.

LIST FILE FORMAT

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

```
          MODULE         AAVR_MAN

OFFSET1   EQU       5
#define   OFFSET2   16

FETCH     MACRO
          LDI       R26,LOW(\1)
          LDI       R27,\1 >> 8
          ADD       R26,R28
          ADC       R27,R29
          LD        R16,X+
          LD        R17,X+
          ENDM

          RSEG      CODE
          PUBLIC    start
          EXTERN    int1,int2

start:
          FETCH     OFFSET1
          STS       int1,R16
          STS       int1+1,R17
          FETCH     OFFSET2
          RET

          END
```

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

HEADER

The header section shows the selected command line options:

```
############################################################################
#                                                                          #
#    IAR Systems AVR Assembler Vx.x dd/Mmm/yyyy  hh:mm:ss                  #
#    Copyright 1999 IAR Systems. All rights reserved.                      #
#                                                                          #
#         Target option =  Relative jumps reach entire addr space          #
#         Source file   =  atest1.s90                                      #
#         List file     =  atest1.lst                                      #
#         Object file   =  atest1.r90                                      #
#         Command line  =  -l atest1.lst atest1.s90                        #
#                                         (c) Copyright IAR Systems 1999   #
############################################################################
```

BODY

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

```
 1    00000000                      MODULE  AAVR_MAN
 2    00000000
 3    00000005              OFFSET1 EQU     5
 4    00000000              #define OFFSET2 16
 5    00000000
14    00000000
15    00000000                      RSEG    CODE
16    00000000                      PUBLIC  start
17    00000000                      EXTERN  int1,int2
18    00000000
19    00000000              start:
20    00000000                      FETCH   OFFSET1
20.1  00000000 E0A5                 LDI     R26,LOW(OFFSET1)
20.2  00000002 E0B0                 LDI     R27,OFFSET1 >> 8
20.3  00000004 0FAC                 ADD     R26,R28
20.4  00000006 1FBD                 ADC     R27,R29
20.5  00000008 910D                 LD      R16,X+
20.6  0000000A 911D                 LD      R17,X+
20.7  0000000C                      ENDM
21    0000000C 9300....             STS     int1,R16
22    00000010 9310....             STS     int1+1,R17
23    00000014                      FETCH   OFFSET2
23.1  00000014 E1A0                 LDI     R26,LOW(OFFSET2)
```

```
23.2  00000016 E0B0                LDI     R27,OFFSET2 >> 8
23.3  00000018 0FAC                ADD     R26,R28
23.4  0000001A 1FBD                ADC     R27,R29
23.5  0000001C 910D                LD      R16,X+
23.6  0000001E 911D                LD      R17,X+
23.7  00000020                     ENDM
24    00000020 9508                RET
25    00000022
26    00000022                     END
```

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

```
20.1  00000000 E0A5                LDI     R26,LOW(OFFSET1)
20.2  00000002 E0B0                LDI     R27,OFFSET1 >> 8
20.3  00000004 0FAC                ADD     R26,R28
```

For information about assembler macros, see *Macro processing directives*, page 74.

CRC

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

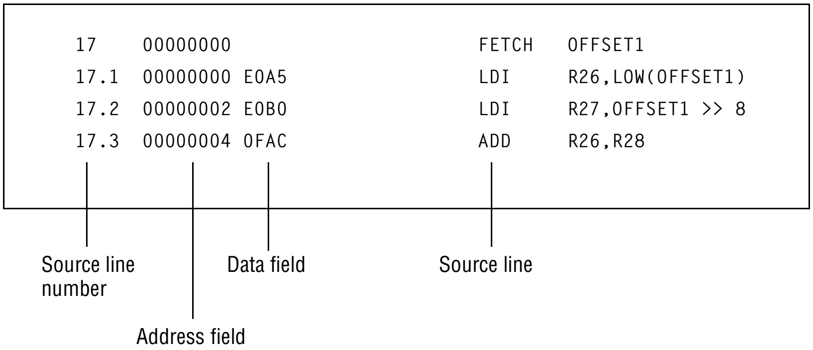
```
##############################
#          CRC:241D          #
#        Errors:   0         #
#        Warnings: 0         #
#         Bytes: 34          #
##############################
```

LIST FIELDS

The assembly list contains the following fields of information:

- The line number in the source file. Lines generated by macros will, if listed, have a `.` (period) in the source line number field.
- The address field shows the location in memory, which can be absolute or relative depending on the type of segment. The notation is hexadecimal.

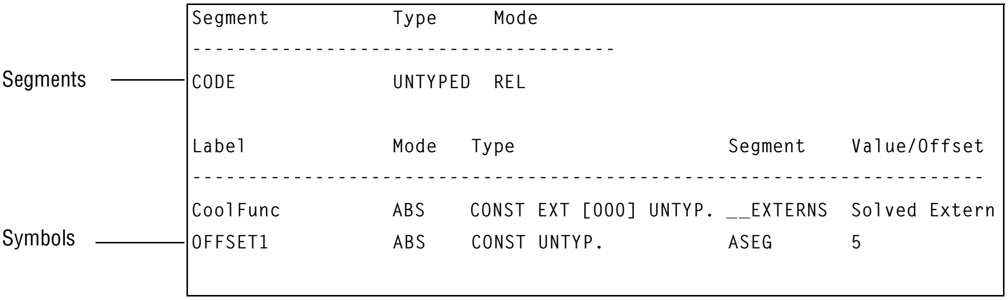
- The data field shows the data generated by the source line. The notation is hexadecimal. Unsolved values are represented by ..... (periods) in the list file, where two periods signify one byte. These unsolved values will be solved during the linking process.
- The assembler source line.



```
17     00000000                          FETCH   OFFSET1
17.1   00000000 E0A5                     LDI     R26,LOW(OFFSET1)
17.2   00000002 E0B0                     LDI     R27,OFFSET1 >> 8
17.3   00000004 0FAC                     ADD     R26,R28
```

SYMBOL AND CROSS-REFERENCE TABLE

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



```
Segment             Type      Mode
--------------------------------------
CODE                UNTYPED   REL

Label               Mode   Type                     Segment    Value/Offset
------------------------------------------------------------------------
CoolFunc            ABS    CONST EXT [000] UNTYP. __EXTERNS  Solved Extern
OFFSET1             ABS    CONST UNTYP.             ASEG       5
```

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).

Information	*Description*
Type	The label's type.
Segment	The name of the segment to which this label is defined relative.
Value/Offset	The value (address) of the label within the current module, relative to the beginning of the current segment part.

OUTPUT FORMATS

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

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

ASSEMBLER OPTIONS

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

SETTING ASSEMBLER OPTIONS

To set assembler options from the command line, 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

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\`

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 *XLINK environment variables*, page 171, and *XLIB environment variables*, page 225.

OPTIONS SUMMARY

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

Command line option	*Description*
`-B`	Macro execution information
`-b`	Make a library module
`-c{DMEAO}`	Conditional list
`-D`*`symb`*`[=`*`value`*`]`	Define symbol
`-E`*`number`*	Maximum number of errors
`-f extend.xcl`	Extend the command line
`-G`	Open standard input as source
`-I`*`prefix`*	Include paths
`-i`	#included text
`-L[`*`prefix`*`]`	List to prefixed source name
`-l` *`filename`*	List to named file
`-M`*`ab`*	Macro quote characters
`-N`	No header
`-O`*`prefix`*	Set object filename prefix
`-o` *`filename`*	Set object filename
`-p`*`lines`*	Lines/page
`-r[en]`	Generate debug information
`-S`	Set silent operation

Command line option	*Description*
`-s{+|-}`	Case sensitive user symbols
`-tn`	Tab spacing
`-u_enhancedCore`	Enable AVR-specific extended instructions
`-Usymb`	Undefine symbol
`-v[0|1|2|3|4|5|6]`	Processor configuration
`-w[string][s]`	Disable warnings
`-x{DI2}`	Include cross-reference

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

-B

Prints macro execution information. This option is mainly used in conjunction with the list file options `-L` or `-l`; for additional information, see page 27.

SYNTAX

`-B`

DESCRIPTION

Causes the assembler to print macro execution information to the standard output stream on every call of a macro. The information consists of:

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

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

-b

Makes a library module to be used with the IAR XLIB Librarian.

SYNTAX

`-b`

DESCRIPTION

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

Conditional list. This option is mainly used in conjunction with the list file options `-L` and `-l`; see page 27 for additional information.

SYNTAX

`-c{DMEAO}`

DESCRIPTION

Sets one or more of the following:

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

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

-D

Defines a symbol to be used by the preprocessor.

SYNTAX

```
Dsymb[=value]
```

DESCRIPTION

Defines a symbol with the name `symb` and the value `value`. If no value is specified, 1 is used.

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

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

Sets maximum number of errors to be reported.

SYNTAX

`-E`*`number`*

DESCRIPTION

Sets the maximum number of errors the assembler reports.

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

Extends the command line.

SYNTAX

`-f extend.xcl`

DESCRIPTION

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

The `-f` option is particularly useful where there is a large number of options which are more conveniently placed in a file than on the command line itself. For example, to run the assembler with further options taken from the file `extend.xcl`, use:

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

-G

Opens standard input as source.

SYNTAX

`-G`

DESCRIPTION

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

Includes paths to be used by the preprocessor.

SYNTAX

`-I`*`prefix`*

DESCRIPTION

Adds the `#include` file search prefix *`prefix`*.

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

For example, using the options:

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

and then writing:

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

in the source, will make the assembler search first in the current directory, then in the directory `c:\global\`, and finally in the directory `c:\thisproj\headers\` provided that the `AAVR_INC` environment variable is set.

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

-i

Includes `#include` text to be used by the preprocessor.

SYNTAX

```
-i
```

DESCRIPTION

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.

-L

Generates a list file with the prefixed source file name.

SYNTAX

```
-L[prefix]
```

DESCRIPTION

Causes the assembler to generate a listing and send it to the file *`prefixsourcename`*`.lst`. Notice that you must not include a space before the prefix.

By default, the assembler does not generate a list file. To simply generate a listing, you use the `-L` option without a prefix. The listing is sent to the file with the same name as the source, but extension `lst`.

The `-L` option lets you specify a prefix, for example to direct the list file to a subdirectory:

```
aavr prog -Llist\
```

This sends the list file to `list\prog.lst` rather than the default `prog.lst`.

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

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

-l

Generates a list file with the specified filename.

SYNTAX

```
-l filename
```

DESCRIPTION

Causes the assembler to generate a listing and send it to the named file. 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

Specifies quote characters for macro arguments.

SYNTAX

```
-Mab
```

DESCRIPTION

Sets the characters used for the left and right quotes of each macro argument to *`a`* and *`b`* respectively.

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

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

Omits the header from assembler list file. This option is useful in conjunction with the list file options `-L` or `-l`; see page 27 for additional information.

SYNTAX

`-N`

DESCRIPTION

By default the assembler list file contains a header section. Use this option to omit the header section that is normally printed in the beginning of the list file.

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

-O

Sets the object filename prefix.

SYNTAX

`-O`*`prefix`*

DESCRIPTION

Set the prefix to be used on the filename of the object file. Notice that you must not include a space before the prefix.

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

```
aavr prog -Oobj\
```

This sends the object to `obj\prog.r90` rather than to the default file `prog.r90`.

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

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

-o

Sets the object filename.

SYNTAX

```
-o filename
```

DESCRIPTION

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.

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.

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

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

Sets number of lines per page. This option is used in conjunction with the list options `-L` or `-l`; see page 27 for additional information.

SYNTAX

```
-plines
```

DESCRIPTION

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

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

-r

Generates debug information to be used with C-SPY.

SYNTAX

```
-r[en]
```

DESCRIPTION

The `-r` option makes the assembler include 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.

- Using the `e` modifier includes the full source file into the object file.
- Using the `n` modifier will generate an object file without source information; symbol information will be available.

This option is identical to the **Generate debug information** option in the **AAVR** category in the IAR Embedded Workbench.

-S

Specifies silent operation.

SYNTAX

```
-S
```

DESCRIPTION

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. You can 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

Makes user symbols case sensitive.

SYNTAX

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

DESCRIPTION

The `-s` option determines 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

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

Specifies the tab spacing. This option is useful in conjunction with the list options `-L` or `-l`; see page 27 for additional information.

SYNTAX

```
-tn
```

DESCRIPTION

The `-t` option sets the number of character positions per tab stop to *n*, which must be in the range 2 to 9.

By default, the assembler sets eight character positions per tab stop.

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

-u_enhancedCore

Enables AVR-specific extended instructions. This option is only useful in conjunction with the `-v3` option.

SYNTAX

```
-u_enhancedCore
```

DESCRIPTION

The `-u_enhancedCore` option enables the extended instructions that are available in, for example, the AT90mega161 microcontroller derivative that you specify by using the `-v3` option.

-U

Undefines a predefined symbol.

SYNTAX

```
-Usymb
```

DESCRIPTION

The `-U` option undefines the symbol *`symb`*.

By default, the assembler provides certain predefined symbols; see *Predefined symbols*, page 25. 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.

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.

-v

Specifies the processor configuration.

SYNTAX

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

DESCRIPTION

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, i.e. `RJMP` and `RCALL` can reach the entire address space.	AT90S2313 AT90S2323 AT90S2333 AT90S2343 AT90S4433
`-v1`	≤ 8 Kbytes code. `RJMP` wraparound is possible, i.e. `RJMP` and `RCALL` can reach the entire address space.	AT90S4414 AT90S4434 AT90S8515 AT90S8534 AT90S8535
`-v2`	≤ 128 Kbytes code. `RJMP` wraparound is not possible, i.e. `RJMP` and `RCALL` cannot reach the entire address space. `CALL` and `JMP` available.	Currently no derivative available using this model.
`-v3`	≤ 128 Kbytes code. `RJMP` wraparound is not possible, i.e. `RJMP` and `RCALL` cannot reach the entire address space. `CALL` and `JMP` available.	AT90mega603 AT90mega103 AT90mega161
`-v4`	≤ 128 Kbytes code. `RJMP` wraparound is not possible, i.e. `RJMP` and `RCALL` cannot reach the entire address space. `CALL` and `JMP` available.	Currently no derivative available using this model.

Option	*Description*	*Derivative*
`-v5`	≤ 8 Mbytes code. `RJMP` wraparound is not possible, i.e. `RJMP` and `RCALL` cannot reach the entire address space. `CALL` and `JMP` available.	Currently no derivative available using this model.
`-v6`	≤ 8 Mbytes code. `RJMP` wraparound is not possible, i.e. `RJMP` and `RCALL` cannot reach the entire address space. `CALL` and `JMP` available.	Currently no derivative available using this model.

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

Disables warnings.

SYNTAX

`-w[string][s]`

DESCRIPTION

By default, the assembler displays a warning message when it detects an element of the source which is legal in a syntactical sense, but may contain a programming error; see *Assembler diagnostics*, page 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.
`-w+n`	Enables just warning `n`.
`-w-n`	Disables just warning `n`.
`-w+m-n`	Enables warnings `m` to `n`.

Command line option	*Description*
`-w-`*`m`*`-`*`n`*	Disables warnings *`m`* to *`n`*.

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.

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
```

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

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

-x

Includes cross-references in the assembler list file. This option is useful in conjunction with the list options `-L` or `-l`; see page 27 for additional information.

SYNTAX

```
-x{DI2}
```

DESCRIPTION

Causes the assembler to generate a cross-reference list at the end of the list file. See the chapter *Assembler file formats*, page 21, for details.

The following options are available:

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

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 complete reference information about each operator, presented in alphabetical order.

PRECEDENCE OF OPERATORS

Each operator has a precedence number assigned to it that determines the order in which the operator and its operands are evaluated. The precedence numbers range from 1 (the highest precedence, i.e. first evaluated) to 7 (the lowest precedence, i.e. last evaluated). Notice that level 2 does not exist. The available levels are 1 and 3–7.

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))
  ```

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

SUMMARY OF ASSEMBLER OPERATORS

UNARY OPERATORS – 1

`+`	Unary plus.
`BITNOT (~)`	Bitwise NOT.
`BYTE2`	Second byte.
`BYTE3`	Third byte.
`DATE`	Current date/time.
`HIGH`	High byte.
`HWRD`	High word.
`LOW`	Low byte.
`LWRD`	Low word.
`NOT (!)`	Logical NOT.
`SFB`	Segment begin.
`SFE`	Segment end.
`SIZEOF`	Segment size.
`-`	Unary minus.

MULTIPLICATIVE ARITHMETIC AND SHIFT OPERATORS – 3

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

ADDITIVE ARITHMETIC OPERATORS – 4

`+`	Addition.
`-`	Subtraction.

AND OPERATORS – 5

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

OR OPERATORS – 6

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

COMPARISON OPERATORS – 7

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

The following sections give full descriptions of each assembler operator.

*****

Multiplication (3).

DESCRIPTION

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

EXAMPLES

```
2*2 → 4
-2*2 → -4
```

+

Unary plus (1).

DESCRIPTION

Unary plus operator.

EXAMPLES

```
+3 → 3
3*+2 → 6
```

+

Addition (4).

DESCRIPTION

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

EXAMPLES

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

–

Unary minus (1).

DESCRIPTION

The unary minus operator performs arithmetic negation on its operand.

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

–

Subtraction (4).

DESCRIPTION

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

EXAMPLES

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

/

Division (3).

DESCRIPTION

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

EXAMPLES

```
9/2 → 4
-12/3 → -4
9/2*6 → 24
```

AND (&&)

Logical AND (5).

DESCRIPTION

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

EXAMPLES

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

BITAND(&)

Bitwise AND (5).

DESCRIPTION

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

EXAMPLES

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

BITNOT (~)

Bitwise NOT (1).

DESCRIPTION

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

EXAMPLE

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

BITOR (|)

Bitwise OR (6).

DESCRIPTION

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

EXAMPLES

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

BITXOR (^)

Bitwise exclusive OR (6).

DESCRIPTION

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

EXAMPLES

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

BYTE2

Second byte (1).

DESCRIPTION

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

EXAMPLE

```
BYTE2 0x12345678 → 0x56
```

BYTE3

Third byte (1).

DESCRIPTION

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

EXAMPLE

`BYTE3 0x12345678 → 0x34`

DATE

Current date/time (1).

DESCRIPTION

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

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

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

EXAMPLE

To assemble the date of assembly:

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

EQ, =, ==

Equal (7).

DESCRIPTION

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

EXAMPLES

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

GE, > =

Greater than or equal (7).

DESCRIPTION

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

EXAMPLES

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

GT, >

Greater than (7).

DESCRIPTION

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

EXAMPLES

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

HIGH

Second byte (1).

DESCRIPTION

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

EXAMPLE

```
HIGH 0xABCD → 0xAB
```

HWRD

High word (1).

DESCRIPTION

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

EXAMPLE

```
HWRD 0x12345678 → 0x1234
```

LE, < =

Less than or equal (7).

DESCRIPTION

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

EXAMPLES

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

LOW

Low byte (1).

DESCRIPTION

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

EXAMPLE

```
LOW 0xABCD → 0xCD
```

LT, <

Less than (7).

DESCRIPTION

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

EXAMPLES

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

LWRD

Low word (1).

DESCRIPTION

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

EXAMPLE

```
LWRD 0x12345678 → 0x5678
```

MOD (%)

Modulo (3).

DESCRIPTION

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

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

EXAMPLES

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

NE, < > , ! =

Not equal (7).

DESCRIPTION

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

EXAMPLES

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

NOT (!)

Logical NOT (1).

DESCRIPTION

Use `NOT` to negate a logical argument.

EXAMPLES

```
NOT B'0101 → 0
NOT B'0000 → 1
```

OR (||)

Logical OR (6).

DESCRIPTION

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

EXAMPLES

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

SFB

Segment begin (1).

SYNTAX

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

PARAMETERS

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

DESCRIPTION

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

EXAMPLES

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

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

SFE

Segment end (1).

SYNTAX

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

PARAMETERS

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

DESCRIPTION

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

EXAMPLES

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

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

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

```
SFE(NEAR_I)-SFB(NEAR_I)
```

SHL (< <)

Logical shift left (3).

DESCRIPTION

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

EXAMPLES

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

SHR (> >)

Logical shift right (3).

DESCRIPTION

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

EXAMPLES

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

SIZEOF

Segment size (1).

SYNTAX

`SIZEOF` *`segment`*

PARAMETERS

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

DESCRIPTION

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

EXAMPLES

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

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

UGT

Unsigned greater than (7).

DESCRIPTION

`UGT` evaluates to 1 (true) if the left operand has a larger absolute value than the right operand.

EXAMPLES

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

ULT

Unsigned less than (7).

DESCRIPTION

`ULT` evaluates to 1 (true) if the left operand has a smaller absolute value than the right operand.

EXAMPLES

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

XOR

Logical exclusive OR (6).

DESCRIPTION

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

EXAMPLES

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

ASSEMBLER DIRECTIVES

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

Module control directives, page 58
Symbol control directives, page 61
Segment control directives, page 62
Value assignment directives, page 68
Conditional assembly directives, page 73
Macro processing directives, page 74
Listing control directives, page 83
C-style preprocessor directives, page 88
Data definition or allocation directives, page 92
Assembler control directives, page 95.

SUMMARY OF DIRECTIVES

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

Directive	*Description*	*Section*
`$`	Includes a file.	Assembler control
`#define`	Assigns a value to a label.	C-style preprocessor
`#elif`	Introduces a new condition in a `#if`…`#endif` block.	C-style preprocessor
`#else`	Assembles instructions if a condition is false.	C-style preprocessor
`#endif`	Ends a `#if`, `#ifdef`, or `#ifndef` block.	C-style preprocessor
`#error`	Generates an error.	C-style preprocessor
`#if`	Assembles instructions if a condition is true.	C-style preprocessor
`#ifdef`	Assembles instructions if a symbol is defined.	C-style preprocessor
`#ifndef`	Assembles instructions if a symbol is undefined.	C-style preprocessor
`#include`	Includes a file.	C-style preprocessor

Directive	*Description*	*Section*
`#message`	Generates a message on standard output.	C-style preprocessor
`#undef`	Undefines a label.	C-style preprocessor
`/*comment*/`	C-style comment delimiter.	Assembler control
`//`	C++ style comment delimiter.	Assembler control
`=`	Assigns a permanent value local to a module.	Value assignment
`ALIAS`	Assigns a permanent value local to a module.	Value assignment
`ALIGN`	Aligns the location counter by inserting zero-filled bytes.	Segment control
`ASEG`	Begins an absolute segment.	Segment control
`ASSIGN`	Assigns a temporary value.	Value assignment
`CASEOFF`	Disables case sensitivity.	Assembler control
`CASEON`	Enables case sensitivity.	Assembler control
`COL`	Sets the number of columns per page.	Listing control
`COMMON`	Begins a common segment.	Segment control
`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

Directive	*Description*	*Section*
`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
`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
`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

Directive	*Description*	*Section*
`LIBRARY`	Begins a library module.	Module control
`LIMIT`	Checks a value against limits.	Value assignment
`LOCAL`	Creates symbols local to a macro.	Macro processing
`LSTCND`	Controls conditional assembly listing.	Listing control
`LSTCOD`	Controls multi-line code listing.	Listing control
`LSTEXP`	Controls the listing of macro generated lines.	Listing control
`LSTMAC`	Controls the listing of macro definitions.	Listing control
`LSTOUT`	Controls assembly-listing output.	Listing control
`LSTPAG`	Controls the formatting of output into pages.	Listing control
`LSTREP`	Controls the listing of lines generated by repeat directives.	Listing control
`LSTXRF`	Generates a cross-reference table.	Listing control
`MACRO`	Defines a macro.	Macro processing
`MODULE`	Begins a library module.	Module control
`NAME`	Begins a program module.	Module control
`ODD`	Aligns the program counter to an odd address.	Segment control
`ORG`	Sets the location counter.	Segment control
`PAGE`	Generates a new page.	Listing control
`PAGSIZ`	Sets the number of lines per page.	Listing control
`PROGRAM`	Begins a program module.	Module control
`PUBLIC`	Exports symbols to other modules.	Symbol control
`RADIX`	Sets the default base.	Assembler control
`REPT`	Assembles instructions a specified number of times.	Macro processing
`REPTC`	Repeats and substitutes characters.	Macro processing
`RSEG`	Begins a relocatable segment.	Segment control
`RTMODEL`	Declares run-time model attributes.	Module control
`sfrb`	Creates byte-access SFR labels.	Value assignment

Directive	*Description*	*Section*
`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

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 *Expressions and operators*, page 21.
`label`	A symbolic label.
`symbol`	An assembler symbol.

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

MODULE CONTROL DIRECTIVES

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

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

SYNTAX

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

PARAMETERS

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

DESCRIPTION

Beginning a program module

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

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

Beginning a library module

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

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

Terminating a module

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

Terminating the last module

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

Assembling multi-module files

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

The following rules apply when assembling multi-module files:

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

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

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

Declaring run-time model attributes
Use `RTMODEL` to enforce consistency between modules. All modules that are linked together and define the same run-time 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 run-time model.

A module can have several run-time model definitions.

Note: The compiler run-time 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 *Configuration* chapter in the *AVR IAR 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 run-time model `"foo"`.
- `MOD_1` and `MOD_3` *can* be linked together since they have the same definition of run-time model `"bar"` and no conflict in the definition of `"foo"`.
- `MOD_2` and `MOD_3` *can* be linked together since they have no run-time model conflicts. The value `"*"` matches any run-time 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.

SYNTAX

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

PARAMETERS

`symbol`	Symbol to be imported or exported.

DESCRIPTION

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

The `PUBLIC` directive always exports full 32-bit values, which makes it feasible to use global 32-bit constants also in assemblers for 8-bit and 16-bit processors. With the `LOW`, `HIGH`, `>>`, 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.

Importing symbols
Use `EXTERN` to import an untyped external symbol.

EXAMPLES

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

```
     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.
`COMMON`	Begins a common segment.
`EVEN`	Aligns the program counter to an even address.
`ODD`	Aligns the program counter to an odd address.
`ORG`	Sets the location counter.

Directive	*Description*
`RSEG`	Begins a relocatable segment.
`STACK`	Begins a stack segment.

SYNTAX

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

PARAMETERS

address	Address where this segment part will be placed.
align	Exponent of the value to which the address should be aligned, in the range 0 to 30. For example, `align 1` results in word alignment 2.
expr	Address to set the location counter to.

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

	`SORT` The linker will sort the segment parts in decreasing alignment order. For a given segment, all segment parts must specify the same state for this flag. The default mode is `NOSORT` which indicates that the segment parts will not be sorted.
segment	The name of the segment.
start	A start address that has the same effect as using an `ORG` directive at the beginning of the absolute segment.
type	The memory type; one of `UNTYPED` (the default), `CODE`, or `DATA`. In addition, the following types are generated for compatibility with the AVR IAR Compiler: `CODE`, `FARCODE`, `DATA`, `FARDATA`, `XDATA`, `CONST`, and `FARCONST`. `HUGEDATA` and `HUGECONST` are equivalent to `DATA` and `CONST`, respectively. The compiler uses `XDATA` for `EEPROM` variables and we recommend this usage also in assembler source code since it will make debugging in C-SPY easier. (`IDATA`, `BIT`, and `REGISTER` are available but should not be used; these segment types are included for compatibility with other IAR assemblers but using them in the AVR IAR Assembler could result in an undefined behavior.)
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 *Segment control*, page 123.

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.

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 `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.

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,”fred 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.

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.

Directive	*Description*
`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.

SYNTAX

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

PARAMETERS

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

DESCRIPTION

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

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

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

For an example where macro directives are used, see *List file format*, page 14.

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 28.

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:

- The assembler performs scanning and saving of macro definitions. The text between `MACRO` and `ENDM` is saved but not syntax checked. Include-file references `$file` are recorded and will be included during macro *expansion*.

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

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

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

Repeating statements

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

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

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

EXAMPLES

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

Coding in-line for efficiency

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

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

```
        NAME    play

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
```

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

Using REPTC and REPTI

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

```
          NAME   reptc

          EXTERN plotc

banner    REPTC  chr, "Welcome"
          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 assembly-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.

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, `ELSE`, or `END`.

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

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

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

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

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

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

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

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

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

Specifying the list file format

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

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

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

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

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

EXAMPLES

Turning the listing on or off

To disable the listing of a debugged section of program:

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

Listing conditional code and strings

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

```
        NAME    lstcndtst
        EXTERN  print

        RSEG    prom

debug   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.
`#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.

SYNTAX

```
#define label text
#elif condition
```

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

PARAMETERS

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

DESCRIPTION

Defining and undefining labels

Use `#define` to define a temporary label.

`#define` *`label value`*

is similar to:

`label` `VAR` *`value`*

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

Conditional directives

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

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

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

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

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

Including source files

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

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

1 The source file directory.

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

3 The current directory.

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

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

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

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

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

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

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

```
#define    five 5 ; comment

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

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

EXAMPLES

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

```
#define tweek 1
#define adjust 3

#ifdef  tweek
#if     adjust=1
        SUBI    R16,4
#elif   adjust=2
```

```
        SUBI    R16,20
#elif   adjust=3
        SUBI    R16,30
#endif
#endif                              /* ifdef tweek */
```

Including a source file

The following example uses `#include` to include a file defining macros into the source file. For example, the following macros could be defined in `macros.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 temporary values or reserve memory.

Directive	*Description*
`DB`	Generates 8-bit byte constants, including strings.
`DC16`	Generates 16-bit word constants, including strings.
`DC24`	Generates 24-bit word constants.
`DC32`	Generates 32-bit double word constants.
`DC8`	Generates 8-bit byte constants, including strings.

Directive	*Description*
`DD`	Generates 32-bit double word constants.
`DP`	Generates 24-bit word constants.
`DS`	Allocates space for 8-bit bytes.
`DS16`	Allocates space for 16-bit words.
`DS24`	Allocates space for 24-bit words.
`DS32`	Allocates space for 32-bit words.
`DS8`	Allocates space for 8-bit bytes.
`DW`	Generates 16-bit word constants, including strings.

SYNTAX

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

PARAMETERS

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

DESCRIPTION

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

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

EXAMPLES

Generating lookup table

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

```
           NAME      table
           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.

SYNTAX

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

PARAMETERS

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

DESCRIPTION

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

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

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

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

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

```
RADIX  0x0A
```

Controlling case sensitivity

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

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

EXAMPLES

Including a source file

The following example uses `$` to include a file defining macros into the source file. For example, the following macros could be defined in `mymacros.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 2: 19.9.99
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`.

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
```

ASSEMBLER DIAGNOSTICS

This chapter lists the error and warning messages for the AVR Assembler. For details of the IAR XLINK Linker™ and IAR XLIB Librarian™ diagnostic messages, see the chapters *XLINK diagnostics* and *XLIB diagnostics*.

INTRODUCTION

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

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

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

The error messages produced by the assembler fall into the following categories:

- Command line error messages.
- Assembly warning messages.
- Assembly error messages.
- Assembly fatal error messages.
- Assembler internal error messages.

COMMAND LINE ERROR MESSAGES

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

ASSEMBLY ERROR MESSAGES

Assembly error messages are produced when the assembler has found a construct which violates the language rules. These messages are listed in the section *Error messages*, page 100.

ASSEMBLY WARNING MESSAGES

Assembly warning messages are produced when the assembler has found a construct which is probably the result of a programming error or omission. These messages are listed in the section *Warning messages*, page 112.

ASSEMBLY FATAL ERROR MESSAGES

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

ASSEMBLER INTERNAL ERROR MESSAGES

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

- The exact internal error message text.
- The source file of the program that generated the internal error.
- A list of the options that were used when the internal error occurred.

ERROR MESSAGES

GENERAL

The following section lists the general error messages.

0 Invalid syntax

The assembler could not decode the expression.

1 Too deep #include nesting (max. is 10)

The assembler limit for nesting of `#include` files was exceeded. A recursive `#include` could be the reason.

2 Failed to open #include file `name`

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

3 Invalid #include file name

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

4 Unexpected end of file encounted

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

5 Too long source line (max. is 2048 characters) truncated

The source line length exceeds the assembler limit.

6 Bad constant

A character that is not a legal digit was encountered.

7 Hexadecimal constant without digits

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

8 Invalid floating point constant

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

9 Too many errors encountered (> 100).

10 Space or tab expected

11 Too deep block nesting (max is 50)

The preprocessor directives are nested too deep.

12 String too long (max is 2045)

The assembler string length limit was exceeded.

13 Missing delimiter in literal or character constant

No closing delimiter `' or "` was found in character or literal constant.

14 Missing #endif

A `#if`, `#ifdef`, or `#ifndef` was found but had no matching `#endif`.

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

16 Identifier expected

A name of a label or symbol was expected.

17 ')' expected

18 No such pre-processor command: *`command`*

`#` was followed by an unknown identifier.

19 Unexpected token found in pre-processor line

The preprocessor line was not empty after the argument part was read.

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

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

22 Macro parameter *`parameter`* redefined

A `#define` symbol's formal parameter was repeated.

23 ’,’ or ’)’ expected

24 Unmatched #else, #endif or #elif

Fatal. Missing `#if`, `#ifdef`, or `#ifndef`.

25 #error ***`error`*****.**

Printout via the `#error` directive.

26 ’(’ expected

27 Too many active macro parameters (max is 256)

Fatal. Preprocessor limit exceeded.

28 Too many nested parameterized macros (max is 50)

Fatal. Preprocessor limit exceeded.

29 Too deep macro nesting (max is 100)

Fatal. Preprocessor limit exceeded.

30 Actual macro parameter too long (max is 512)

A single macro (in `#define`) argument may not exceed the length of a source line.

31 Macro *`macro`* **called with too many parameters**

The number of parameters used was greater than the number in the macro declaration.

32 Macro *`macro`* **called with too few parameters**

The number of parameters used was less than the number in the macro declaration (`#define`).

33 Too many MACRO arguments

The number of assembler macros exceeds 32.

34 May not be redefined

Assembler macros may not be redefined.

35 No name on macro

An assembler macro definition without a label was encountered.

36 Illegal formal parameter in macro

A parameter that was not an identifier was found.

37 ENDM or EXITM not in macro

An `ENDM` directive or `EXITM` directive encountered outside a macro.

38 ' > ' expected but found end-of-line

A `<` was found but no matching `>`.

39 END before start of module

The end-of-module directive has no matching `MODULE` directive.

40 Bad instruction

The mnemonic/directive does not exist.

41 Bad label

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

42 Duplicate label

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

43 Illegal effective address

The addressing mode (operands) is not allowed for this mnemonic.

44 ’,’ expected

A comma was expected but not found.

45 Name duplicated

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

46 Segment type expected

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

47 Segment name expected

The `RSEG`, `STACK`, and `COMMON` directives need a name.

48 Value out of range *`range`*

The value exceeds its limits.

49 Alignment already set

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

50 Undefined symbol: *`symbol`*

The symbol did not appear in label field or in an `EXTERN` or `sfr` declaration.

51 Can’t be both PUBLIC and EXTERN

Symbols can be declared as either `PUBLIC` or `EXTERN`.

52 EXTERN not allowed

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

53 Expression must be absolute

The expression cannot involve relocatable or external symbols.

54 Expression can not be forward

The assembler must be able to solve the expression the first time this expression is encountered.

55 Illegal size

The maximum size for expressions is 32 bits.

56 Too many digits

The value exceeds the size of the destination.

57 Unbalanced conditional assembly directives

Missing conditional assembly `IF` or `ENDIF`.

58 ELSE without IF

Missing conditional assembly `IF`.

59 ENDIF without IF

Missing conditional assembly `IF`.

60 Unbalanced structured assembly directives

Missing structured assembly `IF` or `ENDIF`.

61 ’ + ’ or ’-’ expected

A plus or minus sign is missing.

62 Illegal operation on extern or public symbol

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

63 Illegal operation on non-constant label

It is illegal to make a non-constant symbol `PUBLIC` or `EXTERN`.

64 Extern or unsolved expression

The expression must be solved at assembly time, i.e. not include external references.

65 ' = ' expected

Equals sign was missing.

66 Segment too long (max is *`max`*)

The length of `ASEG`, `RSEG`, `STACK`, or `COMMON` segments is larger than the addressable length.

67 Public did not appear in label field

A symbol was declared `PUBLIC` but no label with the same name was found in the source file.

68 End of block-repeat without start

The repeat directive `REPT` was not found although the `ENDR` directive was.

69 Segment must be relocatable

The operation is not allowed on `ASEG`.

70 Limit exceeded: *`error text`*, value is: *`value`*(decimal)

The value exceeded the limits set with the `LIMIT` directive. The error text is set by the user in the `LIMIT` directive.

71 Symbol *`symbol`* has already been declared EXTERN

An attempt to redeclare an `EXTERN` as `EXTERN` was made.

72 Symbol *`symbol`* has already been declared PUBLIC

An attempt to redeclare a `PUBLIC` as `PUBLIC` was made.

73 End-of-module missing

A `PROGRAM` or `MODULE` directive was encountered before `ENDMOD` was found.

74 Expression must yield non-negative result

The expression was evaluated to a negative number, whereas a positive number was required.

75 Repeat directive unbalanced

This error is caused by a `REPT` directive without a matching `ENDR`, or a an `ENDR` directive without a matching `REPT`.

76 End of repeat directive is missing

A `REPT` directive without a closing `ENDR` was encountered.

77 LOCALs not allowed in this context, (*`symbol`*)

Local symbols must be declared within macro definitions.

78 End of macro expected

An assembler macro is being defined but there was no end-of-macro.

79 End of repeat expected

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

80 End of conditional assembly expected

Conditional assembly is active but there was no end of if.

81 End of structured assembly expected

One of the directives for structured assembly is active but has no matching `END`.

82 Misplaced end of structured assembly

A directive that terminates one of the structured assembly directives was found but no matching `START` directive is active.

83 Error in SFR attribute definition

The `SFRTYPE` directive was used with unknown attributes.

84 Illegal symbol type in symbol

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

85 Wrong number of arguments

Expected a different number of arguments.

86 Number expected

Characters other than digits were encountered.

87 Label must be public or extern

The label must be declared with `PUBLIC` or `EXTERN`.

88 Label not defined with DEFFN

The label has to be defined via `DEFFN` before used in this context.

89 Sorry DEMO version, bytecount exceeded (max bytes)

90 Different parts of ASEG have overlapping code

91 Internal error

92 Empty macro stack overflow

93 Macro stack overflow

94 Attempt to access out-of-stack value

95 Invalid macro operator

96 No such macro argument

97 Sorry Lite version, bytecount exceeded (max bytes)

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

99 #include within macro not supported

100 Duplicate segment definitions

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

AVR-SPECIFIC ERROR MESSAGES

In addition to the general errors, the AVR IAR Assembler may generate the following errors:

400 Absolute operand is not possible here.

401 Accessing SFR incorrectly, check read/write flags.

402 Accessing SFR using incorrect size.

403 Number out of range. Valid range is -128 (-0x80) to 255 (0xFF).

404 Bit-number out of range. Valid range is 0 to 7 (0x07).

405 Address cannot be negative.

406 Register not valid. Use register R16–R31 here.

407 Register not valid. Use register Y or Z.

408 Port address out of range. Valid range is 0 to 63 (0x3F).

409 Register displacement out of range. Valid range is 0 to 63 (0x3F).

410 Address out of range. Valid range is 0 to 8388606 (0x7FFFFE).

411 Address must be even.

412 PC offset out of range. Valid range is -128 (-0x80) to 126 (0x7E).

413 PC offset must be even.

414 Address out of range. Valid range is 0 to 8190 (0x1FFE).

415 PC offset out of range. Valid range is -4096 (-0x1000) to 4094 (0x0FFE).

416 Port address out of range. Valid range is 0 to 31 (0x1F).

417 Number out of range. Valid range is -32 (-0x20) to 63 (0x3F).

418 Register not valid. Use any of registers R24, R26, R28, or R30 here.

419 Address out of range. Valid range is 0 to 65535 (0xFFFF).

420 Instructions must be at an even address. Insert directive 'ALIGN 1' here.

421 Register not valid. Use register R16 - R23 here.

422 Register not even. Use even register R0 - R30 here.

423 Not a good register pair. Use register R1:R0 - R31:R30 here.

424 This register pair syntax is only available for MOVW.

425 Register pair not valid. Use register pair R25:R24 - R31:R30 here.

426 This register pair syntax is only available for ADIW and SBIW.

WARNING MESSAGES

GENERAL

The following section lists the general warning messages.

0 Unreferenced label

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

1 Nested comment

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

2 Unknown escape sequence

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

3 Non-printable character

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

4 Macro or define expected

5 Floating point value out-of-range

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

6 Floating point division by zero

7 Wrong usage of string operator ('#' or '##'); ignored.

The current implementation restricts usage of the # and ## operators to the token field of parameterized macros. In addition, the # operator must precede a formal parameter.

8 Macro parameter(s) not used

9 Macro redefined

10 Unknown macro

11 Empty macro argument

12 Recursive macro

13 Redefinition of Special Function Register

The special function register (`SFR`) has already been defined.

14 Division by zero

Division by 0 in constant expression.

15 Constant truncated

The constant was longer than the size of the destination.

16 Suspicious sfr expression

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

17 Empty module *`module`*, module skipped

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

18 End of program while in include file

The program ended while a file was being included.

19 Symbol *`symbol`* duplicated

20 Bit symbol cannot be used as operand

A symbol was declared using the bit directive, but since the bit address is not calculated the symbol should not be used.

21 Label did not appear in label field

22 Set segment alignment the same *`value`* or larger

When the alignment set by `ALIGN` is larger than the segment alignment it may be lost at link time.

AVR-SPECIFIC WARNING MESSAGES

In addition to the general warnings, the AVR Assembler may generate the following warnings:

400 SFR neither defined as READ nor WRITE

401 More than one SFR size attribute defined, using default (byte)

402 No SFR size attribute defined, using default (byte)

PART 2: THE IAR XLINK LINKER

This part of the AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide contains the following chapters:

- *Introduction to the IAR XLINK Linker*
- *XLINK options*
- *XLINK output formats*
- *XLINK environment variables*
- *XLINK diagnostics*.

INTRODUCTION TO THE IAR XLINK LINKER

The following chapter describes the IAR XLINK Linker™, and gives examples of how it can be used.

Note: The IAR XLINK Linker is a general tool. Therefore, some of the options described in the following chapters may not be relevant for your product.

KEY FEATURES

The IAR XLINK Linker converts one or more relocatable object files produced by the IAR Systems assembler or compiler to machine code for a specified target processor. It supports a wide range of industry-standard loader formats, in addition to the IAR Systems debug format used by the IAR C-SPY Debugger.

The IAR XLINK Linker supports user libraries, and will load only those modules that are actually needed by the program you are linking.

The final output produced by the IAR XLINK Linker is an absolute, target-executable object file that can be programmed into an EPROM, downloaded to a hardware emulator, or run directly on the host using the IAR C-SPY Debugger.

The IAR XLINK Linker offers the following important features:

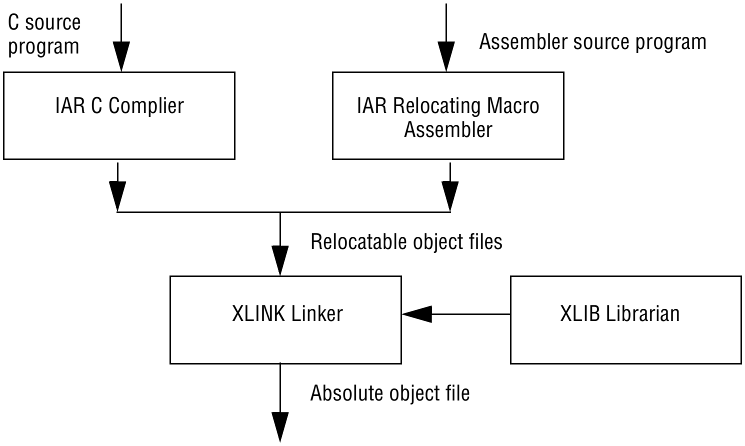
- Unlimited number of input files.
- Searches user-defined library files and loads only those modules needed by the application.
- Symbols may be up to 255 characters long with all characters being significant. Both uppercase and lowercase may be used.
- Global symbols can be defined at link time.
- Flexible segment commands allow full control of the locations of relocatable code and data in memory.
- Support for over 30 output formats.

THE LINKING PROCESS

The IAR XLINK Linker is a powerful, flexible software tool for use in the development of embedded-controller applications. XLINK reads one or more relocatable object files produced by the IAR Systems assembler or compiler and produces absolute, machine-code programs as output.

It is equally well suited for linking small, single-file, absolute assembler programs as it is for linking large, relocatable, multi-module, C/Embedded C++, or mixed C/Embedded C++ and assembler programs.

The following diagram illustrates the linking process:



OBJECT FORMAT

The object files produced by the IAR Systems assembler and compiler use a proprietary format called UBROF, which stands for Universal Binary Relocatable Object Format. An application can be made up of any number of UBROF relocatable files, in any combination of assembler and C/Embedded C++ programs.

XLINK FUNCTIONS

The IAR XLINK Linker performs three distinct functions when you link a program:

- It loads modules containing executable code or data from the input file(s).

- It links the various modules together by resolving all global (i.e. non-local, program-wide) symbols that could not be resolved by the assembler or compiler.
- It loads modules needed by the program from user-defined or IAR-supplied libraries.
- It locates each segment of code or data at a user-specified address.

LIBRARIES

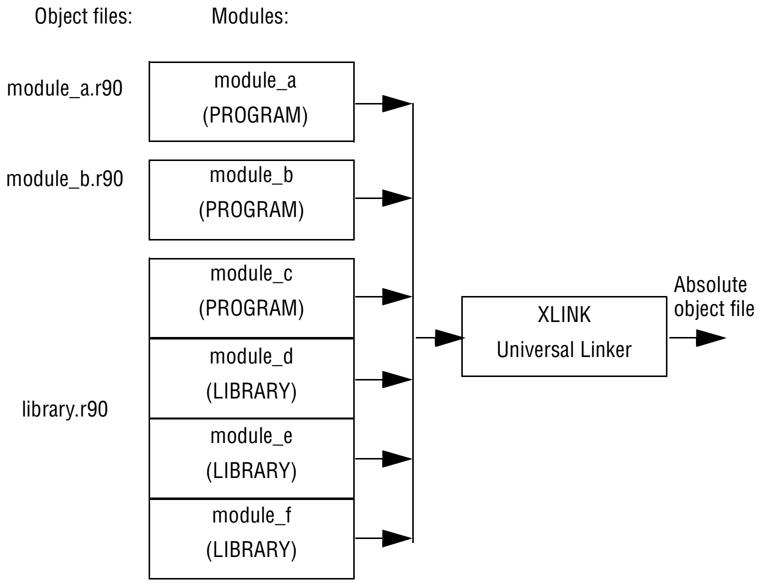
When the IAR XLINK Linker reads a library file (which can contain multiple C/Embedded C++ or assembler modules) it will only load those modules which are actually needed by the program you are linking. The IAR XLIB Librarian is used for managing these library files.

OUTPUT FORMAT

The final output produced by the IAR XLINK Linker is an absolute, executable object file that can be put into an EPROM, downloaded to a hardware emulator, or executed on the PC using the IAR C-SPY® Debugger. The default output format is `MOTOROLA`.

INPUT FILES AND MODULES

The following diagram shows how the IAR XLINK Linker processes input files and load modules for a typical assembler or C/Embedded C++ program:



The main program has been assembled from two source files, `module_a.s90` and `module_b.s90`, to produce two relocatable files. Each of these files consists of a single module `module_a` and `module_b`. By default, the assembler assigns the `PROGRAM` attribute to both `module_a` and `module_b`. This means that they will always be loaded and linked whenever the files they are contained in are processed by the IAR XLINK Linker.

The code and data from a single C/Embedded C++ source file ends up as a single module in the file produced by the compiler. In other words, there is a one-to-one relationship between C/Embedded C++ source files and C/Embedded C++ modules. By default, the compiler gives this module the same name as the original C/Embedded C++ source file. Libraries of multiple C/Embedded C++ modules can only be created using the IAR XLIB Librarian™.

Assembler programs can be constructed so that a single source file contains multiple modules, each of which can be a program module or a library module.

LIBRARIES

In the previous diagram, the file `library.r90` consists of multiple modules, each of which could have been produced by the assembler or the compiler.

The module `module_c`, which has the `PROGRAM` attribute will *always* be loaded whenever the `library.r90` file is listed among the input files for the linker. In the run-time libraries, the startup module `cstartup` (which is a required module in all C/Embedded C++ programs) has the `PROGRAM` attribute so that it will always get included when you link a C/Embedded C++ project.

The other modules in the `library.r90` file have the `LIBRARY` attribute. Library modules are only loaded if they contain an entry (a function, variable, or other symbol declared as `PUBLIC`) that is referenced in some way by another module that is loaded. This way, the IAR XLINK Linker only gets the modules from the library file that it needs to build the program. For example, if the entries in `module_e` are not referenced by any loaded module, `module_e` will not be loaded.

This works as follows:

If `module_a` makes a reference to an external symbol, the IAR XLINK Linker will search the other input files for a module containing that symbol as a `PUBLIC` entry; i.e. a module where the entry itself is located. If it finds the symbol declared as `PUBLIC` in `module_c`, it will then load that module (if it has not already been loaded). This procedure is iterative, so if `module_c` makes a reference to an external symbol the same thing happens.

It is important to understand that a library file is just like any other relocatable object file. There is really no distinct type of file called a library (modules have a `LIBRARY` or `PROGRAM` attribute). What makes a file a library is what it contains and how it is used. Put simply, a library is an `r90` file that contains a group of related, often-used modules, most of which have a `LIBRARY` attribute so that they can be loaded on a demand-only basis.

Creating libraries

You can create your own libraries, or add to existing libraries, using C/Embedded C++ or assembler modules.

The compiler option `--library_module` can be used for making a C/Embedded C++ module have a `LIBRARY` attribute instead of the default `PROGRAM` attribute.

In assembler programs, the `MODULE` directive is used for giving a module the `LIBRARY` attribute, and the `NAME` directive is used for giving a module the `PROGRAM` attribute.

The IAR XLIB Librarian is used for creating and managing libraries. Among other tasks, it can be used for altering the attribute (`PROGRAM` or `LIBRARY`) of any other module after it has been compiled or assembled.

SEGMENTS

Once the IAR XLINK Linker has identified the modules to be loaded for a program, one of its most important functions is to assign load addresses to the various code and data segments that are being used by the program.

In assembly language programs the programmer is responsible for declaring and naming relocatable segments and determining how they are used. In C/Embedded C++ programs the compiler creates and uses a set of predefined code and data segments, and the programmer has only limited control over segment naming and usage.

Each module contains a number of segment parts. Each segment part belongs to a segment, and contains either bytes of code or data, or reserves space in RAM. Using the XLINK segment control command line options (`-Z`, `-P`, and `-b`), you can cause load addresses to be assigned to segments and segment parts.

After module linking is completed, XLINK removes the segment parts that were not required. It accomplishes this by first including all `ROOT` segment parts in loaded modules, and then adding enough other segment parts to satisfy all dependencies. Dependencies are either references to external symbols defined in other modules or segment part references within a module. The `ROOT` segment parts normally consists of the root of the C run-time boot process and any interrupt vector elements.

Compilers and assemblers that produce UBROF 7 or later can put individual functions and variables into separate segment parts, and can represent all dependencies between segment parts in the object file. This enables XLINK to exclude functions and variables that are not required in the build process.

SEGMENT CONTROL

The following options control the allocation of segments.

`-K`*`segs=inc,count`*	Duplicate code.
`-P`*`pack_def`*	Define packed segments.
`-Z`*`seg_def`*	Define segments.
`-b`*`bank_def`*	Define banked segments.
`-M`*`range_def`*	Map logical addresses to physical addresses.

For detailed information about the options, see the chapter *XLINK options*.

Segment placement using `-Z` and `-P` is performed one placement command at a time, taking previous placement commands into account. As each placement command is processed, any part of the ranges given for that placement command that is already in use is removed from the considered ranges. Memory ranges can be in use either by segments placed by earlier segment placement commands, by segment duplication, or by objects placed at absolute addresses in the input fields.

For example, if there are two data segments (`Z1`, `Z2`) that must be placed in the zero page (`0-FF`) and three (`A1`, `A2`, `A3`) that can be placed anywhere in available RAM, they can be placed like this:

```
-Z(DATA)Z1,Z2=0-FF
-Z(DATA)A1,A2,A3=0-1FFF
```

This will place `Z1` and `Z2` from 0 and up, giving an error if they do not fit into the range given, and then place `A1`, `A2`, and `A3` from the first address not used by `Z1` and `Z2`.

The `-P` option differs from `-Z` in that it does not necessarily place the segments (or segment parts) sequentially. See page 152 for more information about the `-P` option. With `-P` it is possible to put segment parts into holes left by earlier placements.

Use the `-Z` option when you need to keep a segment in one consecutive chunk, when you need to preserve the order of segment parts in a segment, or, more unlikely, when you need to put segments in a specific order. There can be several reasons for doing this, but most of them are fairly obscure.

The most important is to keep variables and their initializers in the same order and in one block. Compilers using UBROF 7 or later output attributes that direct the linker to keep segment parts together, so for these compilers `-Z` is no longer required for variable initialization segments.

Use `-P` when you need to put things into several ranges, for instance when banking.

When possible, use the `-P` option instead of `-b`, since `-P` is generally more powerful and more convenient. The `-b` option is supported mainly for backward compatibility reasons, but also because there are still some things it can do that are not supported when using the `-P` option.

Bit segments are always placed first, regardless of where their placement commands are given.

ADDRESS TRANSLATION

XLINK can do logical to physical address translation on output for some output formats. Logical addresses are the addresses as seen by the program, and these are the addresses used in all other XLINK command line options. Normally these addresses are also used in the output object files, but by using the `-M` option a mapping from the logical addresses to physical addresses as used in the output object file is established.

ALLOCATION SEGMENT TYPES

The following table lists the different types of segments that can be processed by XLINK:

Segment type	*Description*
`STACK`	Allocated from high to low addresses by default. The aligned segment size is subtracted from the load address before allocation, and successive segments are placed below the preceding segment.
`RELATIVE`	Allocated from low to high addresses by default.

Segment type	*Description*
`COMMON`	All segment parts are located at the same address.

If stack segments are mixed with relative or common segments in a segment definition, the linker will produce a warning message but will allocate the segments according to the default allocation set by the first segment in the segment list.

Common segments have a size equal to the largest declaration found for the particular segment. That is, if module `A` declares a common segment `COMSEG` with size 4, while module `B` declares this segment with size 5, the latter size will be allocated for the segment.

Be careful not to overlay common segments containing code or initializers.

Relative and stack segments have a size equal to the sum of the different (aligned) declarations.

MEMORY SEGMENT TYPES

The optional *`type`* parameter is used for assigning a type to all of the segments in the list. The *`type`* parameter affects how XLINK processes the segment overlaps. Additionally, it generates information in some of the output formats that are used by some hardware emulators and by C-SPY.

Segment type	*Description*
`BIT`	Bit memory.*
`CODE`	Code memory.
`CONST`	Constant memory.
`DATA`	Data memory.
`FAR`	Data in `FAR` memory. XLINK will not check access to it, and a part of a segment straddling a 64 Kbyte boundary will be moved upwards to start at the boundary.
`FARC, FARCONST`	Constant in `FAR` memory (behaves as above).
`FARCODE`	Code in `FAR` memory.
`HUGE`	Data in `HUGE` memory. No straddling problems.

Segment type	*Description*
`HUGEC`, `HUGECONST`	Constant in `HUGE` memory.
`HUGECODE`	Code in `HUGE` memory.
`IDATA`	Internal data memory.
`NEAR`	Data in `NEAR` memory. Accessed using 16-bit addressing, this segment can be located anywhere in the 32-bit address space.
`NEARC`, `NEARCONST`	Constant in `NEAR` memory.
`UNTYPED`	Default type.
`XDATA`	External data memory.
`ZPAGE`	Zero-page data memory.

* The address of a `BIT` segment is specified in bits, not in bytes. `BIT` memory is allocated first.

RANGE ERRORS

If the ranges specified in the `-Z` option are too short, it will cause either error `24 Segment` *`segment`* `overlaps segment` *`segment`*, if any segment overlaps another, or error `26 Segment` *`segment`* `is too long`, if the ranges are too small.

By default, XLINK checks to be sure that the various segments that have been defined (by the segment placement option and absolute segments) do not overlap in memory.

EXAMPLES

To locate `SEGA` at address `0`, followed immediately by `SEGB`:

```
-Z(CODE)SEGA,SEGB=0
```

To allocate `SEGA` downwards from `FFFH`, followed by `SEGB` below it:

```
-Z(CODE)SEGA,SEGB#FFF
```

To allocate specific areas of memory to `SEGA` and `SEGB`:

```
-Z(CODE)SEGA,SEGB=100-1FF,400-6FF,1000
```

In this example `SEGA` will be placed between address `100` and `1FF`, if it fits in that amount of space. If it does not, XLINK will try the range `400-6FF`. If none of these ranges are large enough to hold `SEGA`, it will start at `1000`.

`SEGB` will be placed, according to the same rules, after segment `SEGA`. If `SEGA` fits the `100–1FF` range then XLINK will try to put `SEGB` there as well (following `SEGA`). Otherwise, `SEGB` will go into the `400` to `6FF` range if it is not too large, or else it will start at `1000`.

```
-Z(NEAR)SEGA,SEGB=19000-1FFFF
```

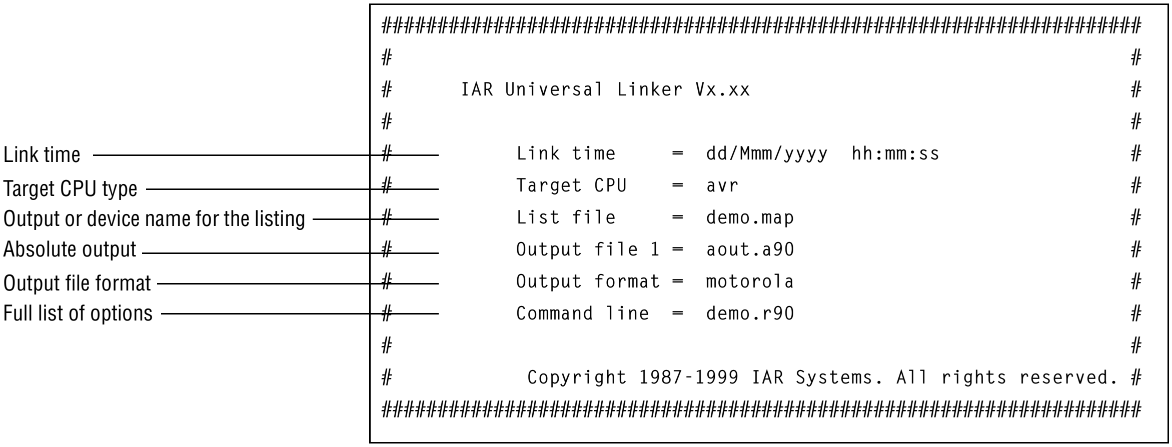
Segments `SEGA` and `SEGB` will be dumped at addresses `19000` to `1FFFF` but the default 16-bit addressing mode will be used for accessing the data (i.e. `9000` to `FFFF`).

LISTING FORMAT

The default XLINK listing consists of the following sections:

HEADER

Shows the command line options selected for the XLINK command:



```
#####################################################################
#                                                                   #
#      IAR Universal Linker Vx.xx                                   #
#                                                                   #
#           Link time     =  dd/Mmm/yyyy  hh:mm:ss                  #
#           Target CPU    =  avr                                    #
#           List file     =  demo.map                               #
#           Output file 1 =  aout.a90                               #
#           Output format =  motorola                               #
#           Command line  =  demo.r90                               #
#                                                                   #
#            Copyright 1987-1999 IAR Systems. All rights reserved.  #
#####################################################################
```

Link time
Target CPU type
Output or device name for the listing
Absolute output
Output file format
Full list of options

The full list of options shows the options specified on the command line. Options in command files specified with the `-f` option are also shown, in brackets.

CROSS-REFERENCE

The cross-reference consists of the entry list, module map and/or the segment map. It includes the program entry point, used in some output formats for hardware emulator support; see the assembler `END` directive in *Module control directives*, page 58.

Module map (-xm)

The module map consists of a subsection for each module that was loaded as part of the program.

Each subsection shows the following information:

List of public symbols

List of segments

Segment name

Segment type and adress

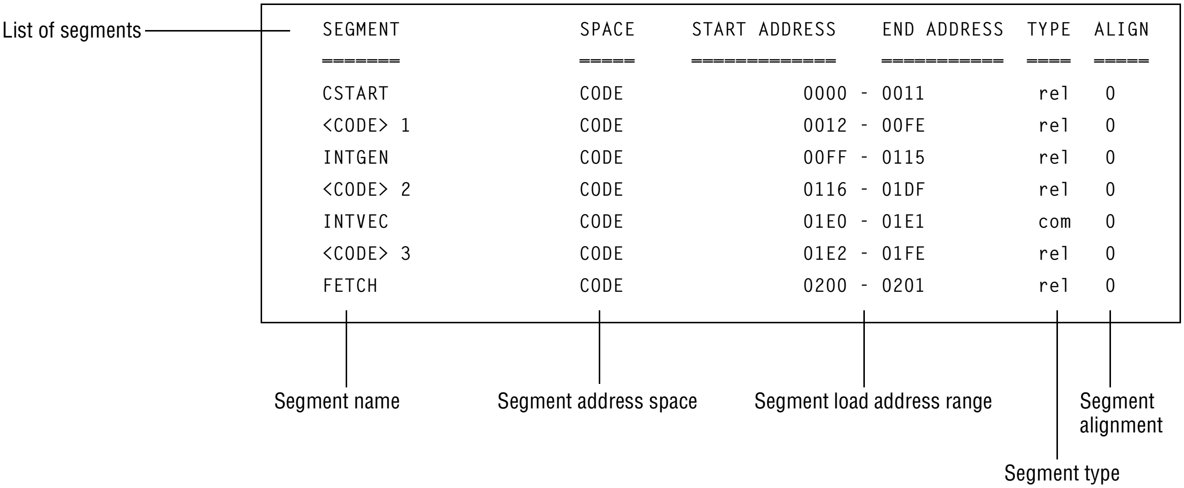
List of local symbols

```
                ****************************************
                *              MODULE MAP              *
                ****************************************


  DEFINED ABSOLUTE ENTRIES
  PROGRAM MODULE, NAME : ?ABS_ENTRY_MOD

Absolute parts
           ENTRY                   ADDRESS         REF BY
           =====                   =======         ======
           _HEAP_SIZE              00000010
           _RSTACK_SIZE            00000010
           _CSTACK_SIZE            00000040

**************************************************************************

  FILE NAME : c:\iar\ew23\avr\projects\debug\obj\common.r90
  PROGRAM MODULE, NAME : common

  SEGMENTS IN THE MODULE
  ======================
TINY_Z
  Relative segment, address: DATA 00000079 - 0000008C (14 bytes)
  Segment part 3.             Intra module refs:   ?<Segment init: TINY_Z>
                                                   init_fibonacci()
                                                   get_fibonacci(char)
           LOCAL                   ADDRESS
           =====                   =======
           fibonacci               00000079

--------------------------------------------------------------------------
CODE
  Relative segment, address: CODE 0000000A - 00000069 (60 bytes)
           ENTRY                   ADDRESS         REF BY
           =====                   =======         ======
           init_fibonacci()        0000000A        main (tutor)
               non_banked function
```

If the module contains any non-relocatable parts, they are listed before the segments.

Segment map (-xs)

The segment list gives the segments in increasing address order:



```
SEGMENT                 SPACE    START ADDRESS    END ADDRESS  TYPE  ALIGN
=======                 =====    =============    ===========  ====  =====
CSTART                  CODE              0000 - 0011           rel    0
<CODE> 1                CODE              0012 - 00FE           rel    0
INTGEN                  CODE              00FF - 0115           rel    0
<CODE> 2                CODE              0116 - 01DF           rel    0
INTVEC                  CODE              01E0 - 01E1           com    0
<CODE> 3                CODE              01E2 - 01FE           rel    0
FETCH                   CODE              0200 - 0201           rel    0
```

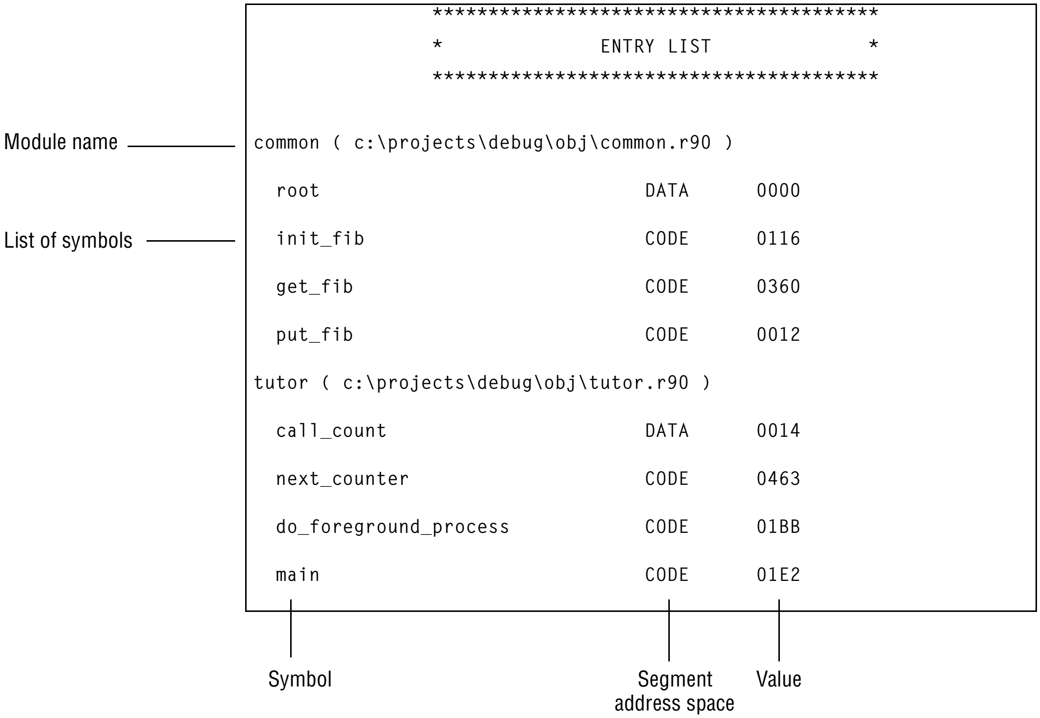
This lists the start and end address for each segment, and the following parameters:

Parameter	*Description*
`TYPE`	The type of segment: `rel` Relative `stc` Stack. `bnk` Banked. `com` Common. `dse` Defined but not used.
`ORG`	The origin; the type of segment start address: `stc` Absolute, for `ASEG` segments. `flt` Floating, for `RSEG`, `COMMON`, or `STACK` segments.
`P/N`	Positive/Negative; how the segment is allocated: `pos` Upwards, for `ASEG`, `RSEG`, or `COMMON` segments. `neg` Downwards, for `STACK` segment.

Parameter	*Description*
`ALIGN`	The segment is aligned to the next `2^ALIGN` address boundary.

Symbol listing (-xe)

The symbol listing shows the entry name and address for each module and filename.



```
                ****************************************
                *              ENTRY LIST              *
                ****************************************

common ( c:\projects\debug\obj\common.r90 )

  root                              DATA      0000

  init_fib                          CODE      0116

  get_fib                           CODE      0360

  put_fib                           CODE      0012

tutor ( c:\projects\debug\obj\tutor.r90 )

  call_count                        DATA      0014

  next_counter                      CODE      0463

  do_foreground_process             CODE      01BB

  main                              CODE      01E2
```

Module name
List of symbols
Symbol
Segment address space
Value

CHECKSUMMED AREAS AND MEMORY USAGE

If the **Generate checksum** (`-J`) and **Fill unused code memory** (`-H`) options have been specified, the listing includes a list of the checksummed areas, in order:

```
          *********************************
             CHECKSUMMED AREAS, IN ORDER
          *********************************


   00000000   -   00007FFF    in CODE memory
   0000D414   -   0000D41F    in CODE memory
Checksum = 32e19
          *********************************
                END OF CROSS REFERENCE
          *********************************
   2068 bytes of CODE memory   (30700 range fill)
   2064 bytes of DATA memory   (12 range fill)
Errors: none
Warnings: none
```

This information is followed, irrespective of the options selected, by the memory usage and the number of errors and warnings.

XLINK OPTIONS

The XLINK options allow you to control the operation of the IAR XLINK Linker™.

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

SETTING XLINK OPTIONS

To set options from the command line, either:

- Specify the options on the command line, after the `xlink` command.
- Specify the options in the `XLINK_ENVPAR` environment variable; see the chapter *XLINK environment variables*.
- Specify the options in an extended linker command line (`xcl`) file, and include this on the command line with the `-f` *`file`* command.

  Note: You can include C-style `/*...*/` or `//` comments in linker command files.

SUMMARY OF OPTIONS

The following table summarizes the XLINK command line options:

Command line option	*Description*
`-!`	Comment delimeter
`-A` *`file,…`*	Load as program
`-a`	Disable static overlay
`-B`	Always generate output
`-b`*`bank_def`*	Define banked segments
`-C` *`file, …`*	Load as library
`-c`*`cpu`*	Processor type
`-D`*`symbol=value`*	Define symbol
`-d`	Disable code generation

Command line option	*Description*
-E *file*,…	Inherent, no object code
-e*new*=*old*[,*old*] …	Rename external symbols
-F*format*	Output format
-f *file*	XCL filename
-G	Disable global type checking
-H*hexstring*	Fill unused code memory
-h[(*seg_type*)]{*range*}	Fill ranges.
-I*pathname*	Include paths
-J*size*,*method*[,*comp*]	Generate checksum
-K*segs*=*inc*,*count*	Duplicate code
-L*directory*	List to directory
-l *file*	List to named file
-M*range_def*	Map logical addresses to physical addresses.
-n[c]	Ignore local symbols
-o *file*	Output file
-P*pack_def*	Define packed segments
-p*lines*	Lines/page
-Q	Scatter loading
-R[w]	Disable range check
-r	Debug info
-rt	Debug info with terminal I/O
-S	Silent operation
-w[*n*|*s*|*t*|*ID*[=*severity*]]	Diagnostics control
-x[e][m][s]	Cross-reference
-Y[*char*]	Format variant
-y[*chars*]	Format variant

Command line option	*Description*
`-Zseg_def`	Define segments
`-z`	Segment overlap warnings

The following sections describe each of the XLINK command line options.

-!

Delimits a comment in the linker command file.

SYNTAX

```
-! comment -!
```

DESCRIPTION

A `-!` can be used for bracketing off comments in an extended linker command file. Unless the `-!` is at the beginning of a line, it must be preceded by a space or tab.

Note: You can include C-style and C++ style comments in your files; the use of these is recommended since they are less error-prone than `-!`.

-A

Loads modules as program modules.

SYNTAX

```
-A file,…
```

DESCRIPTION

Use `-A` to temporarily force all of the modules within the specified input files to be loaded as if they were all program modules, even if some of the modules have the `LIBRARY` attribute.

This option is particularly suited for testing library modules before they are installed in a library file, since the `-A` option will override an existing library module with the same entries. In other words, XLINK will load the module from the *input file* specified in the `-A` argument instead of one with an entry with the same name in a library module.

This option is identical to the **Load as PROGRAM** option in the **XLINK** category in the IAR Embedded Workbench.

-a

Disables static overlay.

SYNTAX

`-a{i|w}[function-list]`

DESCRIPTION

Use `-a` to control the static memory allocation of variables. The options are as follows:

Option	*Description*
`-a`	Disables overlaying totally, for debugging purposes.
`-ai`	Disables indirect tree overlaying.
`-aw`	Disables warning 16, `Function is called from two function trees`. Do this only if you are sure the code is correct.

In addition, the `-a` option can specify one or more function lists, to specify additional options for specified functions. Each function list can have the following form, where *`function`* specifies a public function or a *`module:function`* combination:

Function list	*Description*
`(`*`function,function…`*`)`	Function trees will not be overlayed with another function.
`[`*`function,function…`*`]`	Function trees will not be allocated unless they are called by another function.
`{`*`function,function…`*`}`	Indicates that the specified functions are interrupt functions.

Several `-a` options may be specified, and each `-a` option may include several suboptions, in any order.

-B

Always generate output.

SYNTAX

```
-B
```

DESCRIPTION

Use `-B` to generate an output file even if a non-fatal error was encountered during the linking process, such as a missing global entry or a duplicate declaration. Normally, XLINK will not generate an output file if an error is encountered.

Note: XLINK always aborts on fatal errors, even with `-B`.

The `-B` option allows missing entries to be patched in later in the absolute output image.

This option is identical to the **Always generate output** option in the **XLINK** category in the IAR Embedded Workbench.

-b

Defines banked segments.

SYNTAX

`-b [`*`addrtype`*`] [(`*`type`*`)]` *`segments=first,length,increment`*`[,` *`count`*`]`

where the parameters are as follows:

`addrtype`	The type of load addresses used when dumping the code:	
	`omitted`	Logical addresses with bank number.
	`#`	Linear physical addresses.
	`@`	64180-type physical addresses.
`type`	Specifies the memory type for all segments if applicable for the target microcontroller. If omitted it defaults to `UNTYPED`.	

`segments`	The list of banked segments to be linked. The delimiter between segments in the list determines how they are packed: `:` (colon) The next segment will be placed in a new bank. `,` (comma) The next segment will be placed in the same bank as the previous one.
`first`	The start address of the first segment in the banked segment list. This is a 32-bit value: the high-order 16 bits represent the starting bank number while the low-order 16 bits represent the start address for the banks in the logical address area.
`length`	The length of each bank, in bytes. This is a 16-bit value.
`increment`	The incremental factor between banks, ie the number that will be added to *`first`* to get to the next bank. This is a 32-bit value: the high-order 16 bits are the bank increment, and the low-order 16 bits are the increment from the start address in the logical address area.
`count`	Number of banks available, in decimal.

DESCRIPTION

Use `-b` to allocate banked segments for a program that is designed for bank-switched operation. It also enables the banking mode of linker operation.

There can be more than one `-b` definition.

Logical addresses are the addresses as seen by the program. In most bank-switching schemes this means that a logical address contains a bank number in the most significant 16 bits and an offset in the least significant 16 bits.

Linear physical addresses are calculated by taking the bank number (the most significant 16 bits of the address) times the bank length and adding the offset (the least significant 16 bits of the address). Specifying linear physical addresses affects the load addresses of bytes output by XLINK, not the addresses seen by the program.

64180-type physical addresses are calculated by taking the least significant 8 bits of the bank number, shifting it left 12 bits and then adding the offset.

Using either of these simple translations is only useful for some rather simple memory layouts. Linear physical addressing as calculated by XLINK is useful for a bank memory at the very end of the address space. Anything more complicated will need some post-processing of XLINK output, either by a PROM programmer or a special program. See the `simple` subdirectory for source code for the start of such a program.

For example, to specify that the three code segments `BSEG1`, `BSEG2`, and `BSEG3` should be linked into banks starting at 8000, each with a length of 4000, with an increment between banks of 10000:

```
-b(CODE)BSEG1,BSEG2,BSEG3=8000,4000,10000
```

For more information see, *Segment control*, page 123.

Note: This option is included for backward compatibility reasons. We recommend that you instead use `-P` to define packed segments; see page 152.

-C

Loads modules as library modules.

SYNTAX

```
-C file,…
```

DESCRIPTION

Use `-C` to temporarily cause all of the modules within the specified input files to be treated as if they were all library modules, even if some of the modules have the `PROGRAM` attribute. This means that the modules in the input files will be loaded only if they contain an entry that is referenced by another loaded module.

This option is identical to the **Load as LIBRARY** option in the **XLINK** category in the IAR Embedded Workbench.

-c

Specifies the target processor.

SYNTAX

`-c`*`processor`*

DESCRIPTION

Use `-c` to specify the target processor, for example `avr`.

The environment variable `XLINK_CPU` can be set to install a default for the `-c` option so that it does not have to be specified on the command line; see the chapter *XLINK environment variables*.

This option is related to the **Target** options in the **General** category in the IAR Embedded Workbench.

-D

Defines a symbol.

SYNTAX

`-D`*`symbol`*`=`*`value`*

DESCRIPTION

where *`symbol`* is any external (`EXTERN`) symbol in the program that is not defined elsewhere, and *`value`* the value to be assigned to *`symbol`*.

Use `-D` to define absolute symbols at link time. This is especially useful for configuration purposes. Any number of symbols can be defined in a linker command file. The symbol(s) defined in this manner will belong to a special module generated by the linker called `?ABS_ENTRY_MOD`.

XLINK will display an error message if you attempt to redefine an existing symbol.

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

-d

Disables code generation.

SYNTAX

```
-d
```

DESCRIPTION

Use `-d` to disable the generation of output code from XLINK. This option is useful for the trial linking of programs; for example, checking for syntax errors, missing symbol definitions, etc. XLINK will run slightly faster for large programs when this option is used.

-E

Inherent, no object code.

SYNTAX

```
-E file,…
```

DESCRIPTION

Use `-E` to empty load specified input files; they will be processed normally in all regards by the linker but output code will not be generated for these files.

One potential use for this feature is in creating separate output files for programming multiple EPROMs. This is done by empty loading all input files except the ones you want to appear in the output file.

In the following example a project consists of four files, `file1` to `file4`, but we only want object code generated for `file4` to be put into an EPROM:

```
-E file1,file2,file3
file4
-o project.hex
```

To read object files from `v:\general\lib` and `c:\project\lib`:

```
-Iv:\general\lib;c:\project\lib
```

This option is related to the **Input** options in the **XLINK** category in the IAR Embedded Workbench.

-e

Rename external symbols.

SYNTAX

`-enew=old [,old] …`

DESCRIPTION

Use `-e` to configure a program at link time by redirecting a function call from one function to another.

This can also be used for creating stub functions; i.e. when a system is not yet complete, undefined function calls can be directed to a dummy routine until the real function has been written.

-F

Output format.

SYNTAX

`-Fformat`

DESCRIPTION

Use `-F` to specify the output format.

The environment variable `XLINK_FORMAT` can be set to install an alternate default format on your system; see the chapter *XLINK environment variables*.

The parameter should be one of the supported XLINK output formats; for details of the formats see the chapter *XLINK output formats*.

If not specified, the default `MOTOROLA` format will be used.

Note: Specifying the `-F` option as `DEBUG` does not include C-SPY debug support. Use the `-r` option instead.

This option is related to the **Output** options in the **XLINK** category in the IAR Embedded Workbench.

-f

Specifies the linker command file.

SYNTAX

```
-f file
```

DESCRIPTION

Use `-f` to extend the XLINK command line by reading arguments from a command file, just as if they were typed in on the command line. If not specified an extension of `xcl` is assumed.

Arguments are entered into the linker command file with a text editor using the same syntax as on the command line. However, in addition to spaces and tabs, the Enter key provides a valid delimiter between arguments. A command line may be extended by entering a backslash, `\`, at the end of line.

Note: You can include C-style `/*...*/` or `//` comments in linker command files.

This option is related to the **Include** options in the **XLINK** category in the IAR Embedded Workbench.

-G

Disables the global type checking.

SYNTAX

```
-G
```

DESCRIPTION

Use `-G` to disable type checking at link time. While a well-written program should not need this option, there may be occasions where it is helpful.

By default, XLINK performs link-time type checking between modules by comparing the external references to an entry with the `PUBLIC` entry (if the information exists in the object modules involved). A warning is generated if there are mismatches.

This option is identical to the **No global type checking** option in the **XLINK** category in the IAR Embedded Workbench.

-H

Fills unused code memory.

SYNTAX

```
-Hhexstring
```

DESCRIPTION

Use `-H` to fill all gaps between segment parts introduced by the linker with the repeated *`hexstring`*.

The linker can introduce gaps because of alignment restrictions, or to fill ranges given in segment placement options. The normal behavior, when no `-H` option is given, is that these gaps are not given a value in the output file.

The following example will fill all the gaps with the value `0xbeef`:

```
-HBEEF
```

Even bytes will get the value `0xbe`, and odd bytes will get the value `0xef`.

This option corresponds to the **Fill unused code memory** option in the **XLINK** category in the IAR Embedded Workbench.

-h

Fill ranges.

SYNTAX

```
-h[(seg_type)]{range}
```

DESCRIPTION

Use `-h` to specify the ranges to fill. Normally, all ranges given in segment-placement commands (`-Z` and `-P`) into which any actual content (code or constant data) is placed, are filled. For example:

```
-Z(CODE)INTVEC=0-FF
-Z(CODE)RCODE,CODE,CDATA0=0-7FFF,F800-FFFF
-Z(DATA)IDATA0,UDATA0=8000-8FFF
```

If `INTVEC` contains anything, the range `0-FF` will be filled. If `RCODE`, `CODE` or `CDATA0` contains anything, the ranges `0-7FFF` and `F800-FFFF` will be filled. `IDATA0` and `UDATA0` are normally only place holders for variables, which means that the range `8000-8FFF` will not be filled.

Using `-h` you can explicitly specify which ranges to fill. The syntax allows you to use an optional segment type (which can be used for specifying address space for architectures with multiple address spaces) and one or more address ranges. For example:

```
-h(CODE)0-FFFF
```

or, equivalently, as segment type `CODE` is the default,

```
-h0-FFFF
```

This will cause the range `0-FFFF` to be filled, regardless of what ranges are specified in segment-placement commands. Often `-h` will not be needed.

The `-h` option can be specified more than once, in order to specify fill ranges for more than one address space. It does not restrict the ranges used for calculating checksums.

-I

Specifies include paths.

SYNTAX

```
-Ipathname
```

DESCRIPTION

Specifies a pathname to be searched for object files.

By default, XLINK searches for object files only in the current working directory. The `-I` option allows you to specify the names of the directories which it will also search if it fails to find the file in the current working directory.

This is equivalent to the `XLINK_DFLTDIR` environment variable; see the chapter *XLINK environment variables*.

This option is related to the **Include** option in the **XLINK** category in the IAR Embedded Workbench.

-J

Generates a checksum.

SYNTAX

`-J`*`size`*`,`*`method`*`[,`*`comp`*`]`

DESCRIPTION

Use `-J` to checksum all generated raw data bytes. This option can only be used if the `-H` option has been specified.

`size` specifies the number of bytes in the checksum, and can be 1, 2, or 4.

`method` specifies the algorithm used, and can be one of the following:

Method	*Description*
`sum`	Simple arithmetic sum.
`crc16`	CRC16 (generating polynomial `0x11021`).
`crc32`	CRC32 (generating polynomial `0x104C11DB7`).
`crc=`*`n`*	CRC with a generating polynomial of *`n`*.

`comp` can be 1 to specify one's complement, or 2 to specify two's complement.

In all cases it is the least significant 1, 2, or 4 bytes of the result that will be output, in the natural byte order for the processor. The CRC checksum is calculated as if the following code was called for each bit in the input, starting with a CRC of 0:

```
unsigned long
crc(int bit, unsigned long oldcrc)
{
   unsigned long newcrc = (oldcrc << 1) ^ bit;
   if (oldcrc & 0x80000000)
      newcrc ^= POLY;
   return newcrc;
}
```

`POLY` is the generating polynomial. The checksum is the result of the final call to this routine. If *`comp`* is specified, the checksum is the one's or two's compliment of the result.

The linker will place the checksum byte(s) at the label `__checksum` in the segment `CHECKSUM`. This segment must be placed using the segment placement options like any other segment.

For example, to calculate a 4-byte checksum using the generating polynomial `0x104C11DB7` and output the one's complement of the calculated value, specify:

```
-J4,crc32,1
```

This option corresponds to the **Generate checksum** option in the **XLINK** category in the IAR Embedded Workbench.

-K

Duplicates code.

SYNTAX

```
-Ksegs=inc,count
```

DESCRIPTION

Use `-K` to duplicate any raw data bytes from the segments in *`segs`* *`count`* times, adding *`inc`* to the addresses each time. This will typically be used for segments mentioned in a `-Z` option.

This can be used for making part of a PROM be non-banked even though the entire PROM is physically banked. Use the `-b` or `-P` option to place the banked segments into the rest of the PROM.

For example, to copy the contents of the `RCODE0` and `RCODE1` segments four times, using addresses `0x20000` higher each time, specify:

```
-KRCODE0,RCODE1=20000,4
```

This will place 5 instances of the bytes from the segments into the output file, at the addresses `x`, `x+0x20000`, `x+0x40000`, `x+0x60000`, and `x+0x80000`.

For more information, see *Segment control*, page 123.

-L

Generates a list file, and specifies a directory.

SYNTAX

`-L[`*`directory`*`]`

DESCRIPTION

Causes the linker to generate a listing and send it to the file *`directory`*`\`*`outputname`*`.lst`. Notice that you must not include a space before the prefix.

By default, the linker does not generate a listing. To simply generate a listing, you use the `-L` option without specifying a directory. The listing is sent to the file with the same name as the output file, but extension `lst`.

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

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

-l

Generates a listing and specifies its filename.

SYNTAX

`-l` *`file`*

DESCRIPTION

Causes the linker to generate a listing and send it to the named file. If no extension is specified, `lst` is used by default. However, an extension of `map` is recommended to avoid confusing linker list files with assembler or compiler list files.

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

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

-M

Maps logical addresses to physical addresses.

SYNTAX

`-M[(type)]logical_range=physical_range`

where the parameters are as follows:

`type`		Specifies the memory type for all segments if applicable for the target processor. If omitted it defaults to `UNTYPED`.
`range`	`start-end`	The range starting at `start` and ending at `end`.
	`[start-end]*count+offset`	Specifies `count` ranges, where the first is from `start` to `end`, the next is from `start+offset` to `end+offset`, and so on. The `+offset` part is optional, and defaults to the length of the range.
	`[start-end]/pagesize`	Specifies the entire `range` from `start` to `end`, divided into pages of size and alignment `pagesize`. *Note*: The `start` and `end` of the range do not have to coincide with a page boundary.

DESCRIPTION

XLINK can do logical to physical address translation on output for some output formats. Logical addresses are the addresses as seen by the program, and these are the addresses used in all other XLINK command line options. Normally these addresses are also used in the output object files, but by using the `-M` option, a mapping from the logical addresses to physical addresses, as used in the output object file, is established.

Each occurrence of `-M` defines a linear mapping from a list of logical address ranges to a list of physical address ranges, in the order given, byte by byte.

For example, the command:

```
-M0-FF,200-3FF=1000-11FF,1400-14FF
```

will define the following mapping:

Logical address	*Physical address*
`0x00-0xFF`	`0x1000-0x10FF`
`0x200-0x2FF`	`0x1100-0x11FF`
`0x300-0x3FF`	`0x1400-0x14FF`

Several `-M` command line options can be given to establish a more complex mapping.

Address translation can be useful in banked systems. The following example assumes a code bank at address `0x8000` of size `0x4000`, replicated 4 times, occupying a single physical ROM. To define all the banks using physically contiguous addresses in the output file, the following command is used:

```
-P(CODE)BANKED=[8000-BFFF]*4+10000 // Place banked code
-M(CODE)[8000-BFFF]*4+10000=10000  // Single ROM at 0x10000
```

The `-M` option only supports some output formats, primarily the simple formats with no debug information. The following list shows the currently supported formats:

- `aomf80196`
- `aomf8051`
- `aomf8096`
- `ashling`
- `ashling-6301`
- `ashling-64180`
- `ashling-6801`
- `ashling-8080`
- `ashling-8085`
- `ashling-z80`
- `extended-tekhex`
- `hp-code`
- `intel-extended`
- `intel-standard`
- `millenium`
- `motorola`
- `mpds-symb`
- `pentica-a`
- `pentica-b`
- `pentica-c`
- `pentica-d`
- `rca`
- `symbolic`
- `ti7000`
- `typed`
- `zax`

-n

Ignores local symbols.

SYNTAX

`-n[c]`

DESCRIPTION

Use `-n` to ignore all local (non-public) symbols in the input modules. This option speeds up the linking process and can also reduce the amount of host memory needed to complete a link. If `-n` is used, locals will not appear in the list file cross-reference and will not be passed on to the output file.

Use `-nc` to ignore just compiler-generated local symbols, such as jump or constant labels. These are usually only of interest when debugging at assembler level.

Note: Local symbols are only included in files if they were compiled or assembled with the appropriate option to specify this.

This option is related to the **Output** options in the **XLINK** category in the IAR Embedded Workbench.

-o

Specifies the name of the output file.

SYNTAX

`-o` *`file`*

DESCRIPTION

Use `-o` to specify the name of the XLINK output file. If a name is not specified the linker will use the name `aout.hex`. If a name is supplied without a file type, the default file type for the selected output format will be used; see *-F*, page 142, for additional information.

If a format is selected that generates two output files, the user-specified file type will only affect the primary output file (first format).

This option is related to the **Output** options in the **XLINK** category in the IAR Embedded Workbench.

-P

Defines packed segments.

SYNTAX

`-P [(type)]segments=range[,range] …`

where the parameters are as follows:

type		Specifies the memory type for all segments if applicable for the target processor. If omitted it defaults to `UNTYPED`.
segments		A list of one or more segments to be linked, separated by commas.
range	*start-end*	The range starting at *start* and ending at *end*.
	[*start-end*]**count*+*offset*	Specifies *count* ranges, where the first is from *start* to *end*, the next is from *start*+*offset* to *end*+*offset*, and so on. The +*offset* part is optional, and defaults to the length of the range.
	[*start-end*]/*pagesize*	Specifies the entire range from *start* to *end*, divided into pages of size and alignment *pagesize*. *Note*: The *start* and *end* of the range do not have to coincide with a page boundary.

DESCRIPTION

Use `-P` to pack the segment parts from the specified segments into the specified ranges, where a segment part is defined as that part of a segment that originates from one module.

The linker splits each segment into its segment parts and forms new segments for each of the ranges. All the ranges must be closed; i.e. both *`start`* and *`end`* must be specified. The segment parts will not be placed in any specific order into the ranges.

The following examples show the address range syntax:

`0-9F,100-1FF`	Two ranges, one from zero to `9F`, one from `100` to `1FF`.
`[1000-1FFF]*3+2000`	Three ranges: `1000-1FFF`,`3000-3FFF`,`5000-5FFF`.
`[1000-1FFF]*3`	Three ranges: `1000-1FFF`,`2000-2FFF`,`3000-3FFF`.
`[50-77F]/200`	Five ranges: `50-1FF`,`200-3FF`,`400-5FF`,`600-77F`.

All numbers in segment placement command line options are interpreted as hexadecimal unless they are preceded by a `.` (period). That is, the numbers written as `10` and `.16` are both interpreted as sixteen.

For more information see *Segment control*, page 123.

-p

Specifies the number of lines per page in the XLINK list file.

SYNTAX

`-p`*`lines`*

DESCRIPTION

Sets the number of lines per page for the XLINK list files to *`lines`*, which must be in the range 10 to 150.

The environment variable `XLINK_PAGE` can be set to install a default page length on your system; see the chapter *XLINK environment variables.*

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

-Q

Specifies scatter loading.

SYNTAX

```
-Qsegment=initializer_segment
```

DESCRIPTION

Use `-Q` to do automatic setup for copy initialization of segments (scatter loading). This will cause the linker to generate a new segment (*`initializer_segment`*) into which it will place all data content of the segment *`segment`*. Everything else, e.g. symbols and debugging information, will still be associated with the segment *`segment`*. Code in the application must at runtime copy the contents of *`initializer_segment`* (in ROM) to *`segment`* (in RAM).

This is very similar to what compilers do for initialized variables and is useful for code that needs to be in RAM memory.

The segment *`initializer_segment`* must be placed like any other segment using the segment placement commands.

Assume for example that the code in the segment `RAMCODE` should be executed in RAM. `-Q` can be used for making the linker transfer the contents of the segment `RAMCODE` (which will reside in RAM) into the (new) segment `ROMCODE` (which will reside in ROM), like this:

```
-QRAMCODE=ROMCODE
```

Then `RAMCODE` and `ROMCODE` need to be placed, using the usual segment placement commands. `RAMCODE` needs to be placed in the relevant part of RAM, and `ROMCODE` in ROM. Here is an example:

```
-Z(DATA)RAM segments,RAMCODE,Other RAM=0-1FFF
-Z(CODE)ROM segments,ROMCODE,Other ROM segments=4000-7FFF
```

This will reserve room for the code in `RAMCODE` somewhere between address `0` and address `0x1FFF`, the exact address depending on the size of other segments placed before it. Similarly, `ROMCODE` (which now contains all the original contents of `RAMCODE`) will be placed somewhere between `0x4000` and `0x7FFF`, depending on what else is being placed into ROM.

At some time before executing the first code in `RAMCODE`, the contents of `ROMCODE` will need to be copied into it. This can be done as part of the startup code (in `CSTARTUP`) or in some other part of the code.

-R

Disables range check.

SYNTAX

```
-R[w]
```

DESCRIPTION

Use `-R` to specify the address range check.

If an address is relocated out of the target CPU's address range (code, external data, or internal data address) an error message is generated. This usually indicates an error in an assembly language module or in the segment placement.

The following table shows how the modifiers are mapped:

Option	*Description*
(default)	An error message is generated.
`-Rw`	Range errors are treated as warnings
`-R`	Disables the address range checking

This option is related to the **Range checks** options in the **XLINK** category in the IAR Embedded Workbench.

-r

Generates debug information.

SYNTAX

```
-r
```

DESCRIPTION

Use `-r` to output a file in DEBUG (UBROF) format, with a `d90` extension, to be used with the IAR C-SPY® Debugger. For emulators that support the IAR Systems DEBUG format, use `-F ubrof`.

Specifying `-r` overrides any `-F` option.

This option is related to the **Output** options in the **XLINK** category in the IAR Embedded Workbench.

-rt

Generates debug information with terminal I/O.

SYNTAX

```
-rt
```

DESCRIPTION

Use `-rt` to use the output file with the C-SPY debugger and emulate terminal I/O.

This option is related to the **Output** options in the **XLINK** category in the IAR Embedded Workbench.

-S

Sets silent operation.

SYNTAX

```
-S
```

DESCRIPTION

Use `-S` to turn off the XLINK sign-on message and final statistics report so that nothing appears on your screen during execution. However, this option does not disable error and warning messages or the list file output.

-w

Diagnostics control.

SYNTAX

```
-w[n|s|t|ID[=severity]]
```

DESCRIPTION

Use just `-w` without an argument to suppress warning messages.

The optional argument *n* specifies which warning to disable; for example, to disable warnings 3 and 7:

```
-w3 -w7
```

Specifying `-ws` changes the return status of XLINK as follows:

Condition	*Default*	*-ws*
No errors or warnings	0	0
Warnings but no errors	0	1
One or more errors	2	2

Specifying `-wt` suppresses the detailed type information given for warnings 6 (type conflict) and 35 (multiple structs with the same tag).

Specifying `-wID` changes the severity of a particular diagnostic message. `ID` is the identity of a diagnostic message, which is either the letter `e` followed by an error number, the letter `w` followed by a warning number, or just a warning number.

The optional argument `severity` can be either `i`, `w`, or `e`. If omitted it defaults to `i`.

Severity	*Description*
`i`	Ignore this diagnostic message. No diagnostic output.
`w`	Report this diagnostic message as a warning.
`e`	Report this diagnostic message as an error.

`-w` can be used several times in order to change the severity of more than one diagnostic.

Fatal errors are not affected by this option.

Some examples:

```
-w26
-ww26
-ww26=i
```

These three are equivalent and turn off warning 26.

```
-we106=w
```

This causes error 106 to be reported as a warning.

If the argument is omitted, all warnings are disabled.

As the severity of diagnostic messages can be changed, the identity of a particular diagnostic message includes its original severity as well as its number. That is, diagnostic messages will typically be output as:

`Warning[w6]`: Type conflict for external/entry ...

`Error[e1]`: Undefined external ...

This option is related to the **Diagnostics** options in the **XLINK** category in the IAR Embedded Workbench.

-x

Generates cross-reference information. This option is used with the list options `-L` or `-l`; see page 148 for additional information.

SYNTAX

```
-x[e][m][s]
```

DESCRIPTION

Use `-x` to include a segment map in the XLINK list file.

The following modifiers are available:

Modifier	*Description*
`s`	A list of all the segments in dump order.
`e`	An abbreviated list of every entry (global symbol) in every module. This entry map is useful for quickly finding the address of a routine or data element.
`m`	A list of all segments, local symbols, and entries (public symbols) for every module in the program.

When the `-x` option is specified without any of the optional parameters, a default cross-reference list file will be generated which is equivalent to `-xms`. This includes:

- A header section with basic program information.
- A module load map with symbol cross-reference information.
- A segment load map in dump order.

Cross-reference information is listed to the screen if neither of the `-l` or `-L` options has been specified.

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

-Y

Specifies a format variant.

SYNTAX

```
-Y[char]
```

DESCRIPTION

Use `-Y` to select enhancements available for some output formats. For more information, see the chapter *XLINK output formats*.

This option is related to the **Output** options in the **XLINK** category in the IAR Embedded Workbench.

-y

Specifies a format variant.

SYNTAX

```
-y[chars]
```

DESCRIPTION

Use `-y` to specify output format variants for some formats. A sequence of flag characters can be specified after the option `-y`. The affected formats are `IEEE695` and `XCOFF78K`.

For more information, see the chapter *XLINK output formats*.

This option is related to the **Output** options in the **XLINK** category in the IAR Embedded Workbench.

-Z

Defines segments.

SYNTAX

```
-Z [(type)]segments[=|#]range[,range] …
```

The parameters are as follows:

type		Specifies the memory type for all segments if applicable for the target processor. If omitted it defaults to `UNTYPED`.
segments		A list of one or more segments to be linked, separated by commas. The segments are allocated in memory in the same order as they are listed. Appending +*nnnn* to a segment name increases the amount of memory that XLINK will allocate for that segment by *nnnn* bytes.
= or #	Specifies how segments are allocated:	
	=	Allocates the segments so they begin at the start of the specified range (upward allocation).
	#	Allocates the segment so they finish at the end of the specified range (downward allocation).
	If an allocation operator (and range) is not specified, the segments will be allocated upwards from the last segment that was linked, or from address 0 if no segments have been linked.	
range	*start-end*	The range starting at *start* and ending at *end*.

`[start-end]*count+offset`	Specifies *`count`* ranges, where the first is from *`start`* to *`end`*, the next is from *`start+offset`* to *`end+offset`*, and so on. The *`+offset`* part is optional, and defaults to the length of the range.
`[start-end]/pagesize`	Specifies the entire *`range`* from *`start`* to *`end`*, divided into pages of size and alignment *`pagesize`*. *Note*: The *`start`* and *`end`* of the range do not have to coincide with a page boundary.

DESCRIPTION

Use `-Z` to specify how and where segments will be allocated in the memory map.

If the linker finds a segment in an input file that is not defined either with `-Z`, `-b`, or `-P`, an error is reported. There can be more than one `-Z` definition.

Placement into far memory (the `FAR`, `FARCODE`, `FARCONST` segment types) is treated separately. Using the `-Z` option for far memory, places the segments that fit entirely into the first page and range sequentially, and then places the rest using a special variant of sequential placement that can move an individual segment part into the next range if it did not fit. This means, as before, that far segments can be split into several memory ranges, but it is guaranteed that a far segment has a well-defined start and end.

The following examples show the address range syntax:

`0-9F,100-1FF`	Two ranges, one from zero to `9F`, one from `100` to `1FF`.
`[1000-1FFF]*3+2000`	Three ranges: `1000-1FFF,3000-3FFF,5000-5FFF`.

`[1000-1FFF]*3`	Three ranges: `1000-1FFF`,`2000-2FFF`,`3000-3FFF`.
`[50-77F]/200`	Five ranges: `50-1FF`,`200-3FF`,`400-5FF`,`600-77F`.

All numbers in segment placement command line options are interpreted as hexadecimal unless they are preceded by a `.` (period). That is, the numbers written as `10` and `.16` are both interpreted as sixteen.

For more information see *Segment control*, page 123.

-z

Reduces segment overlap errors.

SYNTAX

```
-z
```

DESCRIPTION

Use `-z` to reduce segment overlap errors to warnings, making it possible to produce cross-reference maps, etc.

This option is related to the **Diagnostics** options in the **XLINK** category in the IAR Embedded Workbench.

XLINK OUTPUT FORMATS

This chapter gives a summary of the IAR XLINK Linker™ output formats.

SINGLE OUTPUT FILE

The following formats result in the generation of a single output file:

Format	*Type*	*Extension*	*Address type*
`AOMF8051†`	binary	from CPU	N
`AOMF8096†`	binary	from CPU	N
`AOMF80196†`	binary	from CPU	N
`AOMF80251`	binary	from CPU	N
`ASHLING`	binary	none	N
`ASHLING-6301†`	binary	from CPU	N
`ASHLING-64180†`	binary	from CPU	NS
`ASHLING-6801†`	binary	from CPU	N
`ASHLING-8080†`	binary	from CPU	NS
`ASHLING-8085†`	binary	from CPU	NS
`ASHLING-Z80†`	binary	from CPU	NS
`DEBUG (UBROF)†§`	binary	`dbg`	NL
`ELF`	binary	`elf`	NL
`EXTENDED-TEKHEX†`	ASCII	from CPU	NLPS
`HP-CODE`	binary	`x`	NLPS
`HP-SYMB`	binary	`l`	NLPS
`IEEE695†**`	binary	`695`	NL
`INTEL-EXTENDED`	ASCII	from CPU	NLPS
`INTEL-STANDARD`	ASCII	from CPU	N
`MILLENIUM (Tektronix)`	ASCII	from CPU	N
`MOTOROLA`	ASCII	from CPU	NLPS

Format	*Type*	*Extension*	*Address type*
`MPDS-CODE`	binary	`tsk`	N
`MPDS-SYMB`	binary	`sym`	NLPS
`MSD`	ASCII	`sym`	N
`MSP430_TXT`	ASCII	`txt`	NLPS
`NEC-SYMBOLIC†`	ASCII	`sym`	N
`NEC2-SYMBOLIC†`	ASCII	`sym`	N
`NEC78K-SYMBOLIC†`	ASCII	`sym`	N
`PENTICA-A`	ASCII	`sym`	NLPS
`PENTICA-B`	ASCII	`sym`	NLPS
`PENTICA-C`	ASCII	`sym`	NLPS
`PENTICA-D`	ASCII	`sym`	NLPS
`RCA`	ASCII	from CPU	N
`SIMPLE`	binary	`raw`	NLPS
`SYMBOLIC`	ASCII	from CPU	NLPS
`SYSROF†`	binary	`abs`	NLPS
`TEKTRONIX (Millenium)`	ASCII	`hex`	N
`TI7000 (TMS7000)`	ASCII	from CPU	N
`TYPED`	ASCII	from CPU	NLPS
`UBROF†`	binary	`dbg`	NL
`UBROF5†`	binary	`dbg`	NL
`UBROF6†`	binary	`dbg`	NL
`UBROF7†`	binary	`dbg`	NL
`XCOFF78k`	binary	`lnk`	NL
`ZAX`	ASCII	from CPU	NLPS

† The format depends on the typing of the segments. This indicates that the *`type`* field specified in the XLINK `-Z` option is important.

** The format is supported only for certain combinations of CPU and debugger; see `xlink.txt` and `xman.txt` for more information.

§ Using `-FUBROF` (or `-FDEBUG`) will generate UBROF output matching the latest UBROF format version in the input. Using `-FUBROF5` (or `-FUBROF6`) will force output of the specified version of the format, irrespective of the input.

Address type
The address type is one of the following:

N = Non-banked address.
L = Banked logical address.
P = Banked physical address.
S = Banked 64180 physical address.

TWO OUTPUT FILES

The following formats result in the generation of two output files:

Format	*Code format*	*Ext.*	*Symbolic format*	*Ext.*
`DEBUG-MOTOROLA`	`DEBUG`	`a90`	`MOTOROLA`	`obj`
`DEBUG-INTEL-EXT`	`DEBUG`	`a90`	`INTEL-EXT`	`hex`
`DEBUG-INTEL-STD`	`DEBUG`	`a90`	`INTEL-STD`	`hex`
`HP`	`HP-CODE`	`x`	`HP-SYMB`	`l`
`MPDS`	`MPDS-CODE`	`tsk`	`MPDS-SYMB`	`sym`
`MPDS-I`	`INTEL-STANDARD`	`hex`	`MPDS-SYMB`	`sym`
`MPDS-M`	`Motorola`	`s19`	`MPDS-SYMB`	`sym`
`MSD-I`	`INTEL-STANDARD`	`hex`	`MSD`	`sym`
`MSD-M`	`Motorola`	`hex`	`MSD`	`sym`
`MSD-T`	`MILLENIUM`	`hex`	`MSD`	`sym`
`NEC`	`INTEL-STANDARD`	`hex`	`NEC-SYMB`	`sym`
`NEC2`	`INTEL-STANDARD`	`hex`	`NEC2-SYMB`	`sym`
`NEC78K`	`INTEL-STANDARD`	`hex`	`NEC2-SYMB`	`sym`
`PENTICA-AI`	`INTEL-STANDARD`	`obj`	`Pentica-a`	`sym`
`PENTICA-AM`	`Motorola`	`obj`	`Pentica-a`	`sym`

Format	*Code format*	*Ext.*	*Symbolic format*	*Ext.*
`PENTICA-BI`	`INTEL-STANDARD`	`obj`	`Pentica-b`	`sym`
`PENTICA-BM`	`Motorola`	`obj`	`Pentica-b`	`sym`
`PENTICA-CI`	`INTEL-STANDARD`	`obj`	`Pentica-c`	`sym`
`PENTICA-CM`	`Motorola`	`obj`	`Pentica-c`	`sym`
`PENTICA-DI`	`INTEL-STANDARD`	`obj`	`Pentica-d`	`sym`
`PENTICA-DM`	`Motorola`	`obj`	`Pentica-d`	`sym`
`ZAX-I`	`INTEL-STANDARD`	`hex`	`ZAX`	`sym`
`ZAX-M`	`Motorola`	`hex`	`ZAX`	`sym`

OUTPUT FORMAT VARIANTS

The following enhancements can be selected for the specified output formats, using the **Format variant** (`-Y`) option:

Output format	*Option*	*Description*
`PENTICA-A,B,C,D` and `MPDS-SYMB`	`Y0`	Symbols as *`module`*`:`*`symbolname`*.
	`Y1`	Labels and lines as *`module:symbolname`*.
	`Y2`	Lines as *`module:symbolname`*.
`AOMF8051`	`Y0`	Extra type of information for Hitex.
`INTEL-STANDARD`	`Y0`	End only with `:00000001FF`.
	`Y1`	End with `PGMENTRY`, else `:0000001FF`.
`MPDS-CODE`	`Y0`	Fill with `0xFF` instead.
`DEBUG, -r`	`Y#`	Old UBROF version.
`INTEL-EXTENDED`	`Y0`	Segmented variant.
	`Y1`	32-bit linear variant.

Refer to the file `xlink.txt` for information about additional options that may have become available since this guide was published.

Use **Format variant** (`-y`) to specify output format variants for some formats. A sequence of flag characters can be specified after the option `-y`. The affected formats are `IEEE695` (see page 167), `ELF` (see page 168), and `XCOFF78K` (see page 169).

IEEE695

For `IEEE695` the available format modifier flags are:

Modifier	*Description*
No #define constants (`-yd`)	Do not emit any #define constant records. This can sometimes drastically reduce the size of the output file.
Output global types globally (`-yg`)	Output globally visible types in a `BB2` block at the beginning of the output file.
Output global types in each module (`-yl`)	Output the globally visible types in a BB1 block at the beginning of each module in the output file.
Treat bit sections as byte sections (`-yb`)	XLINK supports the use of IEEE-695 *based variables* to represent bit variables, and the use of bit addresses for bit-addressable sections. Turning on this modifier makes XLINK treat these as if they were byte variables or sections.
Adjust output for the Mitsubishi PDB30 debugger (`-ym`)	Turning on this modifier adjusts the output in some particular ways for the Mitsubishi PDB30 debugger. *Note*: You will need to use the `l` and `b` modifiers as well (`-ylbm`).
No block-local constants (`-ye`)	Using this modifier will cause XLINK to not emit any block-local constant in the output file. One way these can occur is if an `enum` is declared in a block.
Handle variable life times (`-yv`)	Use the *variable life time* support in IEEE-695 to output more accurate debug information for variables whose location vary.
Output stack adjust records (`-ys`)	Output IEEE-695 *stack adjust* records to indicate the offset from the stack pointer of a virual frame pointer.
Output module locals in `BB10` block (`-ya`)	Output information about module local symbols in `BB10` (assembler level) blocks as well as in the `BB3` (high level) blocks, if any.

Modifier	*Description*
Last return refers to end of function (`-yr`)	Change the source line information for the last return statement in a function to refer to the last line of the function instead of the line where it is located.

The following table shows the recommended format variant modifiers for specific debuggers:

Debugger	*Format variant modifier*
6812 Noral debugger	`-ygvs`
68HC16 Microtek debugger	`-ylb`
740 Mitsubishi PD38	`-ylbma`
7700 HP RTC debugger	`-ygbr`
7700 Mitsubishi PD77	`-ylbm`
H8300 HP RTC debugger	`-ygbr`
H8300H HP RTC debugger	`-ygbr`
H8S HP RTC debugger	`-ygbr`
M16C HP RTC debugger	`-ygbr`
M16C Mitsubishi PD30/PDB30/KDB30	`-ylbm`
T900 Toshiba RTE900 m25	`-ygbe`

ELF

For `ELF` the available format modifier flags are:

Modifier	*Description*
Suppress DWARF debug output (`-yn`)	Output an ELF file without debug information.
Multiple ELF program sections (`-yp`)	Output one ELF program section for each segment, instead of one section for all segments combined.

The XLINK ELF/DWARF format output includes module-local symbols. The command line option `-n` can be used for suppressing module-local symbols in any output format.

The XLINK output conforms to ELF as described in *Executable and Linkable Format (ELF)* and to DWARF version 2, as described in *DWARF Debugging Information Format,* revision 2.0.0 (July 27, 1993); both are parts of the Tools Interface Standard Portable Formats Specification, version 1.1.

Note: The ELF format is currently supported for the 68HC11, 68HC12, 68HC16, SH and V850 products.

XCOFF78K

For `XCOFF78K` the available format modifier flags are:

Modifier	*Description*
`-ys`	Truncates names longer than 31 characters to 31 characters. Irrespective of the setting of this modifier, section names longer than 7 characters are always truncated to 7 characters and module names are truncated to 31 characters.
`-yp`	Strips source file paths, if there are any, from source file references, leaving only the file name and extension.
`-ye`	Includes module enums. Normally XLINK does not output module-local constants in the `XCOFF78K` file. The way IAR compilers currently work these include all `#define` constants as well as all `SFR`s. Use this modifier to have them included.
`-yl`	Hobbles line number info. When outputting debug information, use this modifier to ignore any source file line number references that are not in a strictly increasing order within a function.

If you want to specify more than one flag, all flags must be specified after the same `-y` option; for example, `-ysp`.

XLINK ENVIRONMENT VARIABLES

The IAR XLINK Linker™ supports a number of environment variables. These can be used for creating defaults for various XLINK options so that they do not have to be specified on the command line.

Except for the `XLINK_ENVPAR` environment variable, the default values can be overruled by the corresponding command line option. For example, the `-FMPDS` command line argument will supersede the default format selected with the `XLINK_FORMAT` environment variable.

SUMMARY OF XLINK ENVIRONMENT VARIABLES

The following environment variables can be used by the IAR XLINK Linker:

Environment variable	*Description*
`XLINK_COLUMNS`	Sets the number of columns per line.
`XLINK_CPU`	Sets the target CPU type.
`XLINK_DFLTDIR`	Sets a path to a default directory for object files.
`XLINK_ENVPAR`	Creates a default XLINK command line.
`XLINK_FORMAT`	Sets the output format.
`XLINK_PAGE`	Sets the number of lines per page.

XLINK_COLUMNS

Sets the number of columns per line.

DESCRIPTION

Use `XLINK_COLUMNS` to set the number of columns in the list file. The default is 80 columns.

EXAMPLE

To set the number of columns to 132:

```
set XLINK_COLUMNS=132
```

XLINK_CPU

Sets the target processor.

DESCRIPTION

Use `XLINK_CPU` to set a default for the `-c` option so that it does not have to be specified on the command line.

EXAMPLE

To set the target processor to `avr`:

```
set XLINK_CPU=avr
```

RELATED COMMANDS

This is equivalent to the XLINK `-c` option; see *-c*, page 140.

XLINK_DFLTDIR

Sets a path to a default directory for object files.

DESCRIPTION

Use `XLINK_DFLTDIR` to specify a path for object files. The specified path, which should end with `\`, is prefixed to the object filename.

EXAMPLE

To specify the path for object files as `c:\iar\lib`:

```
set XLINK_DFLTDIR=c:\iar\lib\
```

XLINK_ENVPAR

Creates a default XLINK command line.

DESCRIPTION

Use `XLINK_ENVPAR` to specify XLINK commands that you want to execute each time you run XLINK.

EXAMPLE

To create a default XLINK command line:

```
set XLINK_ENVPAR=-FMOTOROLA
```

RELATED COMMANDS

For more information about reading linker commands from a file, see *-f*, page 143.

XLINK_FORMAT

Sets the output format.

DESCRIPTION

Use `XLINK_FORMAT` to set the format for linker output. For a list of the available output formats, see the chapter *XLINK output formats*.

EXAMPLE

To set the output format to Motorola:

```
set XLINK_FORMAT=MOTOROLA
```

RELATED COMMANDS

This is equivalent to the XLINK `-F` option; see *-F*, page 142.

XLINK_PAGE

Sets the number of lines per page.

DESCRIPTION

Use `XLINK_PAGE` to set the number of lines per page (20–150). The default is a list file without page breaks.

EXAMPLES

To set the number of lines per page to 64:

```
set XLINK_PAGE=64
```

RELATED COMMANDS

This is equivalent to the XLINK `-p` option; see *-p*, page 153.

XLINK DIAGNOSTICS

This chapter describes the errors and warnings produced by the IAR XLINK Linker™.

INTRODUCTION

The error messages produced by the IAR XLINK Linker fall into the following categories:

- XLINK warning messages.
- XLINK error messages.
- XLINK fatal error messages.
- XLINK internal error messages.

XLINK WARNING MESSAGES

XLINK warning messages will appear when the linker detects something that may be wrong. The code that is generated may still be correct.

XLINK ERROR MESSAGES

XLINK error messages are produced when the linker detects that something is incorrect. The linking process will be aborted unless the **Always generate output** (`-B`) option is specified. The code produced is almost certainly faulty.

XLINK FATAL ERROR MESSAGES

XLINK fatal error messages abort the linking process. They occur when continued linking is useless, i.e. the fault is irrecoverable.

XLINK INTERNAL ERRORS

During linking, a number of internal consistency checks are performed. If any of these checks fail, the linker will terminate after giving a short description of the problem. These errors will normally not occur, but if they do you should report them to the IAR Systems Technical Support group. Please include information enough to reproduce the problem from both souce and object code. This would typically include:

- The exact internal error message text.

- The object code files, as well as the corresponding source code files, of the program that generated the internal error.

  If the file size total is very large, please contact IAR Technical Support before sending the files.

- A list of the compiler/assembler and linker options that were used when the internal error occurred, including the linker command file.

  If you are using the IAR Embedded Workbench, these settings are stored in the `prj` and `dtp` files of your project.

- Product names and version numbers of the IAR Systems development tools that were used.

ERROR MESSAGES

If you get a message that indicates a corrupt object file, reassemble or recompile the faulty file since an interrupted assembly or compilation may produce an invalid object file.

The following table lists the IAR XLINK Linker error messages:

0 Format chosen cannot support banking

Format unable to support banking.

1 Corrupt file. Unexpected end of file in module *`module`* **(***`file`***) encountered**

Linker aborts immediately. Recompile or reassemble, or check the compatibility between the linker and C compiler.

2 Too many errors encountered (> 100)

Linker aborts immediately.

3 Corrupt file. Checksum failed in module *`module`* **(***`file`***). Linker checksum is** *`linkcheck`***, module checksum is** *`modcheck`*

Linker aborts immediately. Recompile or reassemble.

4 **Corrupt file. Zero length identifier encountered in module** *`module`* **(***`file`***)**

Linker aborts immediately. Recompile or reassemble.

5 **Address type for CPU incorrect. Error encountered in module** *`module`* **(***`file`***)**

Linker aborts immediately. Check that you are using the right files and libraries.

6 **Program module** *`module`* **redeclared in file** *`file`***. Ignoring second module**

XLINK will not produce code unless the **Always generate output** (`-B`) option (forced dump) is used.

7 **Corrupt file. Unexpected UBROF – format end of file encountered in module** *`module`* **(***`file`***)**

Linker aborts immediately. Recompile or reassemble.

8 **Corrupt file. Unknown or misplaced tag encountered in module** *`module`* **(***`file`***). Tag** *`tag`*

Linker aborts immediately. Recompile or reassemble.

9 **Corrupt file. Module** *`module`* **start unexpected in file** *`file`*

Linker aborts immediately. Recompile or reassemble.

10 **Corrupt file. Segment no.** *`segno`* **declared twice in module** *`module`* **(***`file`***)**

Linker aborts immediately. Recompile or reassemble.

11 **Corrupt file. External no.** *`ext no`* **declared twice in module** *`module`* **(***`file`***)**

Linker aborts immediately. Recompile or reassemble.

12 Unable to open file `file`

Linker aborts immediately. If you are using the command line, check the environment variable `XLINK_DFLTDIR`.

13 Corrupt file. Error tag encountered in module `module` **(**`file`**)**

A UBROF error tag was encountered. Linker aborts immediately. Recompile or reassemble.

14 Corrupt file. Local `local` **defined twice in module** `module` **(**`file`**)**

Linker aborts immediately. Recompile or reassemble.

15 This error message has been deleted.

16 Segment `segment` **is too long for segment definition**

The segment defined does not fit into the memory area reserved for it. Linker aborts immediately.

17 Segment `segment` **is defined twice in segment definition -Zsegdef**

Linker aborts immediately.

18 Range error in module `module` **(**`file`**), segment** `segment` **at address** `address`**. Value** `value`**, in tag** `tag`**, is out of bounds**

The address is out of the CPU address range. Locate the cause of the problem using the information given in the error message.

The check can be suppressed by the `-R` option.

19 Corrupt file. Undefined segment referenced in module `module` **(**`file`**)**

Linker aborts immediately. Recompile or reassemble.

20 Undefined external referenced in module `module` **(**`file`**)**

Linker aborts immediately. Recompile or reassemble.

21 Segment *`segment`* **in module** *`module`* **does not fit bank**

The segment is too long. Linker aborts immediately.

22 Paragraph no. is not applicable for the wanted CPU. Tag encountered in module *`module`* **(***`file`***)**

Linker aborts immediately. Delete the paragraph number declaration in the `xcl` file.

23 Corrupt file. T_REL_FI_8 or T_EXT_FI_8 is corrupt in module *`module`* **(***`file`***)**

The tag `T_REL_FI_8` or `T_EXT_FI_8` is faulty. Linker aborts immediately. Recompile or reassemble.

24 Segment *`segment`* **overlaps segment** *`segment`*

The segments overlap each other; i.e. both include the same address.

25 Corrupt file. Unable to find module *`module`* **(***`file`***)**

A module is missing. Linker aborts immediately.

26 Segment *`segment`* **is too long**

This error should never occur unless the program is extremely large. Linker aborts immediately.

27 Entry entry in module *`module`* **(***`file`***) redefined in module** *`module`* **(***`file`***)**

There are two or more entries with the same name. Linker aborts immediately.

28 File *`file`* **is too long**

The program is too large. Split the file. Linker aborts immediately.

29 No object file specified in command-line

There is nothing to link. Linker aborts immediately.

30 Option `option` also requires the `option` option

Linker aborts immediately.

31 Option `option` cannot be combined with the `option` option

Linker aborts immediately.

32 Option `option` cannot be combined with the `option` option and the `option` option

Linker aborts immediately.

33 Faulty value `value`, (range is 10-150)

Faulty page setting. Linker aborts immediately.

34 Filename too long

The filename is more than 255 characters long. Linker aborts immediately.

35 Unknown flag `flag` in cross reference option `option`

Linker aborts immediately.

36 Option `option` does not exist

Linker aborts immediately.

37 - not succeeded by character

The `-` (dash) marks the beginning of an option, and must be followed by a character. Linker aborts immediately.

38 Option `option` must not be defined more than once

Linker aborts immediately.

39 Illegal character specified in option `option`

Linker aborts immediately.

40 Argument expected after option *`option`*

This option must be succeeded by an argument. Linker aborts immediately.

41 Unexpected '-' in option *`option`*

Linker aborts immediately.

42 Faulty symbol definition -D*`symbol definition`*

Incorrect syntax. Linker aborts immediately.

43 Symbol in symbol definition too long

The symbol name is more than 255 characters. Linker aborts immediately.

44 Faulty value *`value`***, (range 80-300)**

Faulty column setting. Linker aborts immediately.

45 Unknown CPU *`CPU`* **encountered in** *`context`*

Linker aborts immediately. Make sure that the argument to `-c` is valid. If you are using the command line you can get a list of CPUs by typing `xlink -c?`.

46 Undefined external *`external`* **referred in** *`module`* **(***`file`***)**

Entry to *`external`* is missing.

47 Unknown format *`format`* **encountered in** *`context`*

Linker aborts immediately.

48 This error message has been deleted.

49 This error message has been deleted.

50 **Paragraph no. not allowed for this CPU, encountered in option** *`option`*

Linker aborts immediately. Do not use paragraph numbers in declarations.

51 *`Input base`* **value expected in option** *`option`*

Linker aborts immediately.

52 **Overflow on value in option** *`option`*

Linker aborts immediately.

53 **Parameter exceeded 255 characters in extended command line file** *`file`*

Linker aborts immediately.

54 **Extended command line file** *`file`* **is empty**

Linker aborts immediately.

55 **Extended command line variable XLINK_ENVPAR is empty**

Linker aborts immediately.

56 **Non-increasing range in segment definition** *`segment def`*

Linker aborts immediately.

57 **No CPU defined**

No CPU defined, either in the command line or in `XLINK_CPU`. Linker aborts immediately.

58 No format defined

No format defined, either in the command line or in `XLINK_FORMAT`. Linker aborts immediately.

59 Revision no. for file is imcompatible with XLINK revision no.

Linker aborts immediately.

If this error occurs after recompilation or reassembly, the wrong version of XLINK is being used. Check with your supplier.

60 Segment *`segment`* defined in bank definition and segment definition.

Linker aborts immediately.

61 This error message has been deleted.

62 Input file *`file`* cannot be loaded more than once

Linker aborts immediately.

63 Trying to pop an empty stack in module *`module`* (*`file`*)

Linker aborts immediately. Recompile or reassemble.

64 Module *`module`* (*`file`*) has not the same debug type as the other modules

Linker aborts immediately.

65 Faulty replacement definition -e *`replacement`* definition

Incorrect syntax. Linker aborts immediately.

66 Function with F-index *`index`* has not been defined before indirect reference in module *`module`* (*`file`*)

Indirect call to an undefined in module. Probably caused by an omitted function declaration.

67 Function `name` **has same F-index as** `function-name`**, defined in module** `module` **(**`file`**)**

Probably a corrupt file. Recompile file.

68 External function `name` **in module** `module` **(**`file`**) has no global definition**

If no other errors have been encountered, this error is generated by an assembly-language call from C where the required declaration using the `$DEFFN` assembly-language support directive is missing. The declaration is necessary to inform the linker of the memory requirements of the function.

69 Indirect or recursive function `name` **in module** `module` **(file) has parameters or auto variables in nondefault memory**

The recursively or indirectly called function name is using extended language memory specifiers (bit, data, idata, etc) to point to non-default memory, memory which is not allowed.

Function parameters to indirectly called functions must be in the default memory area for the memory model in use, and for recursive functions, both local variables and parameters must be in default memory.

70 This error message has been deleted.

71 Segment `segment` **is incorrectly defined (in a bank definition, has wrong segment type or mixed with other segment types)**

This is usually due to misuse of a predefined segment; see the explanation of `segment` in the *AVR IAR Compiler Reference Guide*. It may be caused by changing the predefined linker control file.

72 Segment `name` **must be defined in a segment option definition (-Z, -b, or -P)**

This is caused either by the omission of a segment in the linker (usually a segment needed by the C system control) file or by a spelling error (segment names are case sensitive).

73 Label ?ARG_MOVE not found (recursive function needs it)

In the library there should be a module containing this label. If it has been removed it must be restored.

74 There was an error when writing to file `file`

Either the linker or your host system is corrupt, or the two are incompatible.

75 SFR address in module `module` **(**`file`**), segment** `segment` **at address** `address`**, value** `value` **is out of bounds**

A special function register (SFR) has been defined to an incorrect address. Change the definition.

76 Absolute segments overlap in module `module`

The linker has found two or more absolute segments in `module` overlapping each other.

77 Absolute segments in module `module` **(**`file`**) overlaps absolute segment in module** `module` **(**`file`**)**

The linker has found two or more absolute segments in `module` (`file`) and `module` (`file`) overlapping each other.

78 Absolute segment in module `module` **(**`file`**) overlaps segment** `segment`

The linker has found an absolute segment in `module` (`file`) overlapping a relocatable segment.

79 Faulty allocation definition -a `definition`

The linker has discovered an error in an overlay control definition.

80 Symbol in allocation definition (-a) too long

A symbol in the `-a` command is too long.

81 Unknown flag in extended format option *`option`*

Make sure that the flags are valid.

82 Conflict in segment *`name`***. Mixing overlayable and not overlayable segment parts.**

These errors only occur with the 8051 and converted PL/M code.

83 The overlayable segment *`name`* **may not be banked.**

These errors only occur with the 8051 and converted PL/M code.

84 The overlayable segment *`name`* **must be of relative type.**

These errors only occur with the 8051 and converted PL/M code.

85 The far/farc segment *`name`* **in module** *`module`* **(***`file`***) is larger than** *`size`*

The segment *`name`* is too large to be a `far` segment.

86 This error message has been deleted.

87 Function with F-index *`i`* **has not been defined before tiny_func referenced in module** *`module`* **(***`file`***)**

Check that all tiny functions are defined before they are used in a module.

88 Wrong library used (compiler version or memory model mismatch). Problem found in *`module`* **(***`file`***). Correct library tag is** *`tag`*

Code from this compiler needs a matching library. A library belonging to a later or earlier version of the compiler may have been used.

92 Cannot use this format with this cpu

Some formats need CPU-specific information and are only supported for some CPUs.

93 Non-existant warning number `number`**, (valid numbers are 0-**`max`**)**

An attempt to suppress a warning that does not exist gives this error.

94 Unknown flag `x` **in local symbols option -n**`x`

The character `x` is not a valid flag in the local symbols option.

95 Module `module` **(**`file`**) uses source file references, which are not available in UBROF 5 output**

This feature cannot be filtered out by the linker when producing UBROF 5 output.

96 Unmatched -! comment in extended command file

An odd number of `-!` (comment) options were seen in a linker command file.

97 Unmatched -! comment in extended command line variable XLINK_ENVPAR

As above, but for the environment variable `XLINK_ENVPAR`.

98 Unmatched /* comment in extended command file

No matching `*/` was found in the linker command file.

99 Syntax error in segment definition: `option`

There was a syntax error in the option.

100 Segment name too long: `segment` **in** `option`

The segment name exceeds the maximum length (255 characters).

101 Segment already defined: `segment` **in** `option`

The segment has already been mentioned in a segment definition option.

102 No such segment type: *`option`*

The specified segment type is not valid.

103 Ranges must be closed in *`option`*

The `-P` option requires all memory ranges to have an end.

104 Failed to fit all segments into specified ranges. Problem discovered in segment *`segment`***.**

The packing algorithm used in the linker did not manage to fit all the segments.

105 Recursion not allowed for this system. Check module map for recursive functions

The run-time model used does not support recursion. Each function determined by the linker to be recursive is marked as such in the module map part of the linker list file.

106 Syntax error or bad argument in *`option`*

There was an error when parsing the command line argument given.

107 Banked segments do not fit into the number of banks specified

The linker did not manage to fit all of the contents of the banked segments into the banks given.

108 Cannot find function *`function`* **mentioned in -a#**

All the functions specified in an indirect call option must exist in the linked program.

109 Function *`function`* **mentioned as callee in -a# is not indirectly called**

Only functions that actually can be called indirectly can be specified to do so in an indirect call option.

110 Function `function` mentioned as caller in -a# does not make indirect calls

Only functions that actually make indirect calls can be specified to do so in an indirect call option.

111 The file `file` is not a UBROF file

The contents of the file are not in a format that XLINK can read.

112 The module `module` is for an unknown cpu (tid = `tid`). Either the file is corrupt or you need a later version of XLINK

The version of XLINK used has no knowledge of the CPU that the file was compiled/assembled for.

113 Corrupt input file: `symptom` in module `module` (`file`)

The input file indicated appears to be corrupt. This can occur either because the file has for some reason been corrupted after it was created, or because of a problem in the compiler/assembler used to create it. If the latter appears to be the case, please contact IAR Technical Support.

114 This message does not exist.

115 Unmatched '"' in extended command file or XLINK_ENVPAR

When parsing an extended command file or the environment variable `XLINK_ENVPAR`, XLINK found an unmatched quote character.

For filenames with quote characters you need to put a backslash before the quote character. For example, writing

```
c:\iar\"A file called \"file\""
```

will cause XLINK to look for a file called

```
A file called "file"
```

in the `c:\iar` directory.

116 Definition of *`symbol`* in module *`module1`* is not compatible with definition of *`symbol`* in module *`module2`*

The symbol *`symbol`* has been tentatively defined in one or both of the modules. Tentative definitions must match other definitions.

117 Incompatible runtime modules. Module *`module1`* specifies that *`attribute`* must be *`value1`*, but module *`module2`* has the value *`value2`*

These modules cannot be linked together. They were compiled with settings that resulted in incompatible run-time modules.

118 Incompatible runtime modules. Module *`module1`* specifies that *`attribute`* must be *`value`*, but module *`module2`* specifies no value for this attribute.

These modules cannot be linked together. They were compiled with settings that resulted in incompatible run-time modules.

119 Cannot handle C++ identifiers in this output format

The selected output format does not support the use of C++ identifiers (block-scoped names or names of C++ functions).

120 Overlapping address ranges for address translation. *`address type`* address *`address`* is in more than one range

The address *`address`* (of logical or physical type) is the source or target of more han one address translation command.

If, for example, both `-M0-2FFF=1000` and `-M2000-3FFF=8000` are given, this error may be given for any of the logical addresses in the range `2000-2FFF`, for which to separate translation commands have been given.

121 Segment part or absolute content at logical addresses *`start - end`* **would be translated into more than one physical address range**

The current implementation of address translation does not allow logical addresses from one segment part (or the corresponding range for absolute parts from assembler code) to end up in more than one physical address range.

If, for example, `-M0-1FFF=10000` and `-M2000-2FFF=20000` are used, a single segment part is not allowed to straddle the boundary at address `2000`.

122 The address *`address`* **is too large to be represented in the output format** *`format`*

The selected output format *`format`* cannot represent the address *`address`*. For example, the output format `INTEL-STANDARD` can only represent addresses in the range `0-FFFF`.

123 The output format *`format`* **does not support address translation (`-M`, `-b#`, or `-b@`)**

Address translation is not supported for all output formats.

124 Segment conflict for segment *`segment`*. **In module** *`module1`* **there is a segment part that is of type** *`type1`*, **while in module** *`module2`* **there is a segment part that is of type** *`type2`*

All segment parts for a given segment must be of the same type. One reason for this conflict can be that a `COMMON` segment is mistakenly declared `RSEG` (relocatable) in one module.

125 This message does not exist.

126 Runtime model attribute "__cpu" not found. Please enter at least one line in your assembly code that contains the following statement: RTMODEL "__cpu", "16C61". Replace 16C61 with your chosen CPU. The CPU must be in uppercase.

The `__cpu` runtime model attribute is needed when producing COFF output. The compiler always supplies this attribute, so this error can only occur for programs consisting entirely of assembler modules.

At least one of the assembler modules must supply this attribute.

127 Segment placement command "*`command`*" provides no address range, but the last address range(s) given is the wrong kind (bit addresses versus byte addresses).

This error will occur if something like this is entered:

```
-Z(DATA)SEG=1000-1FFF
-Z(BIT)BITVARS=
```

Note that the first uses byte addresses and the second needs bit addresses. To avoid this, provide address ranges for both.

128 Segments cannot be mentioned more than once in a copy init command: "`-Q`*`args`*"

Each segment must be either the source or the target of a copy init command.

WARNING MESSAGES

The following section lists the linker warning messages:

0 Too many warnings

Too many warnings encountered.

1 Error tag encountered in module *`module`* (*`file`*)

A UBROF error tag was encountered when loading file *`file`*. This indicates a corrupt file and will generate an error in the linking phase.

2 **Symbol `symbol` is redefined in command-line**

A symbol has been redefined.

3 **Type conflict. Segment `segment`, in module `module`, is incompatible with earlier segment(s) of the same name**

Segments of the same name should have the same type.

4 **Close/open conflict. Segment `segment`, in module `module`, is incompatible with earlier segment of the same name**

Segments of the same name should be either open or closed.

5 **Segment `segment` cannot be combined with previous segment**

The segments will not be combined.

6 **Type conflict for external/entry `entry`, in module `module`, against external/entry in module `module`**

Entries and their corresponding externals should have the same type.

7 **Module `module` declared twice, once as program and once as library. Redeclared in file `file`, ignoring library module**

The program module is linked.

8 This warning message has been deleted.

9 **Ignoring redeclared program entry in module `module` (`file`), using entry from module `module1`**

Only the program entry found first is chosen.

10 **No modules to link**

The linker has no modules to link.

11 Module *`module`* declared twice as library. Redeclared in file *`file`*, ignoring second module

The module found first is linked.

12 Using SFB in banked segment *`segment`* in module *`module`* (*`file`*)

The `SFB` assembler directive may not work in a banked segment.

13 Using SFE in banked segment *`segment`* in module *`module`* (*`file`*)

The `SFE` assembler directive may not work in a banked segment.

14 Entry *`entry`* duplicated. Module *`module`* (*`file`*) loaded, module *`module`* (*`file`*) discarded

Duplicated entries exist in conditionally loaded modules; i.e. library modules or conditionally loaded program modules (with the `-C` option).

15 Predefined type sizing mismatch between modules *`module`* (*`file`*) and *`module`* (*`file`*)

The modules have been compiled with different options for predefined types, such as different sizes of basic C types (e.g. `integer`, `double`).

16 Function *`name`* in module *`module`* (*`file`*) is called from two function trees (with roots *`name1`* and *`name2`*)

The probable cause is *`module`* interrupt function calls another function that also could be executed by a foreground program, and this could lead to execution errors.

17 Segment name is too large or placed at wrong address

This error occurs if a given segment overruns the available address space in the named memory area. To find out the extent of the overrun do a dummy link, moving the start address of the named segment to the lowest address, and look at the linker map file. Then relink with the correct address specification.

18 Segment *`segment`* **overlaps segment** *`segment`*

The linker has found two relocatable segments overlapping each other. Check the segment placement option parameters.

19 Absolute segments overlaps in module *`module`* **(***`file`***)**

The linker has found two or more absolute segments in module *`module`* overlapping each other.

20 Absolute segment in module *`module`* **(***`file`***) overlaps absolute segment in module** *`module`* **(***`file`***)**

The linker has found two or more absolute segments in module *`module`* (*`file`*) and module *`module`* (*`file`*) overlapping each other. Change the `ORG` directives.

21 Absolute segment in module *`module`* **(***`file`***) overlaps segment** *`segment`*

The linker has found an absolute segment in module *`module`* (*`file`*) overlapping a relocatable segment. Change either the `ORG` directive or the `-Z` relocation command.

22 Interrupt function *`name`* **in module** *`module`* **(***`file`***) is called from other functions**

Interrupt functions may not be called.

23 *`limitation-specific warning`*

Due to some limitation in the chosen output format, or in the information available, XLINK cannot produce the correct output. Only one warning for each specific limitation is given.

24 *`num`* **counts of** *`warning`* **total**

For each warning of type 23 emitted, a summary is provided at the end.

25 Using -Y# discards and distorts debug information. Use with care. If possible find an updated debugger that can read modern UBROF

Using the UBROF format modifer `-Y#` is not recommended.

26 No reset vector found

Failed in determining the `LOCATION` setting for XCOFF output format for the 78400 processor, because no reset vector was found.

27 No code at the start address

Failed in determining the `LOCATION` setting for XCOFF output format for the 78400 processor, because no code was found at the address specified in the reset vector.

28 Parts of segment *`name`* are initialized, parts not

This is not useful if the result linking is to be promable.

29 Parts of segment *`name`* are initialized, even though it is of type *`type`* (and thus not promable)

Initing `DATA` memory is not useful if the result of linking is to be promable.

30 Module *`name`* is compiled with tools for *`cpu1`* expected *`cpu2`*

You are building an executable for CPU *`cpu2`*, but module name is compiled for CPU *`cpu1`*.

31 Modules have been compiled with possibly incompatible settings: *`more information`*

According to the contents of the modules, they are not compatible.

32 Format option set more than once. Using format *`format`*

The format option can only be given once. The linker uses the format *`format`*.

33 Using -r overrides format option. Using UBROF

The `-r` option specifies UBROF format and C-SPY® library modules. It overrides any `-F` (format) option.

34 The 20 bit segmented variant of the INTEL EXTENDED format cannot represent the addresses specified. Consider using -Y1 (32 bit linear addressing).

The program uses addresses higher than `0xFFFFF`, and the segmented variant of the chosen format cannot handle this. The linear addressing variant can handle full 32 bit addresses.

35 There is more than one definition for the struct/union type with tag *`tag`*

Two or more different structure/union types with the same tag exist in the program. If this is not intentional, it is likely that the declarations differ slightly. It is very likely that there will also be one or more warnings about type conflicts (warning 6). If this is intentional, consider turning this warning off.

36 There are indirectly called functions doing indirect calls. This can make the static overlay system unreliable

XLINK does not know what functions can call what functions in this case, which means that it cannot make sure static overlays are safe.

37 More than one interrupt function makes indirect calls. This can make the static overlay system unreliable. Using -ai will avoid this

If a function is called from an interrupt while it is already running its params and locals will be overwritten.

38 There are indirect calls both from interrupts and from the main program. This can make the static overlay system unreliable. Using -ai will avoid this

If a function is called from an interrupt while it is already running its params and locals will be overwritten.

39 The function *`function`* in module *`module`* (*`file`*) does not appear to be called. No static overlay area will be allocated for its params and locals

As far as XLINK can tell, there are no callers for the function, so no space is needed for its params and locals. To make XLINK allocate space anyway use `-a(`*`function`*`)`.

40 The module *`module`* contains obsolete type information that will not be checked by the linker

This kind of type information is no longer used.

41 The function *`function`* in module *`module`* (*`file`*) makes indirect calls but is not mentioned in the left part of any -a# declaration

If any `-a#` indirect call options are given they must, taken together, specify the complete picture.

42 This warning message does not exist.

43 The function *`function`* in module *`module`* (*`file`*) is indirectly called but is not mentioned in the right part of any -a# declaration

If any `-a#` indirect call options are given they must, taken together, specify the complete picture.

44 C library routine localtime failed. Timestamps will be wrong

XLINK is unable to determine the correct time. This primarily affects the dates in the list file. This problem has been observed on one host platform if the date is after the year 2038.

45 Memory attribute info mismatch between modules *`module1`* (*`file1`*) and *`module2`* (*`file2`*)

The UBROF 7 memory attribute information in the given modules is not the same.

46 External function *`function`* **in module** *`module`* **(***`file`***) has no global definition**

This warning replaces error 68.

47 Range error in module *`module`* **(***`file`***), segment** *`segment`* **at address** *`address`***. Value** *`value`***, in tag** *`tag`***, is out of bounds** *`bounds`*

This replaces error 18 when `-Rw` is specified.

48 Corrupt input file: *`symptom`* **in module** *`module`* **(***`file`***)**

The input file indicated appears to be corrupt. This warning is used in preference to Error 113 when the problem is not serious, and is unlikely to cause trouble.

49 Using SFB/SFE in module *`module`* **(***`file`***) for segment** *`segment`***, which has no included segment parts**

SFB/SFE (assembler directives for getting the start or end of a segment) has been used on a segment for which no segment parts were included.

50 There was a problem when trying to embed the source file *`source`* **in the object file**

This warning is given if the file *`source`* could not be found or if there was an error reading from it. XLINK searches for source files in the same places as it searches for object files, so including the directory where the source file is located in the XLINK **Include** (`-I`) option could solve the problem.

51 Some source reference debug info was lost when translating to UBROF 5 (example: statements in *`function`* **in module** *`module`*

UBROF 6 file references can handle source code in more than one source file for a module. This is not possible in UBROF 5 embedded source, so any references to files not included have been removed.

52 More than one definition for the byte at address `address` **in common segment** `segment`

The most probable cause is that more than one module defines the same interrupt vector.

53 Some untranslated addresses overlap translation ranges. Example: Address `addr1` **(untranslated) conflicts with logical address** `addr2` **(translated to** `addr1`**)**

This can be caused by something like this:

```
-Z(CODE)SEG1=1000-1FFF
-Z(CODE)SEG2=2000-2FFF
-M(CODE)1000=2000
```

This will place `SEG1` at logical address `1000` and `SEG2` at logical address `2000`. However, the translation of logical address `1000` to physical address `2000` and the absence of any translation for logical address `1000` will mean that in the output file, both `SEG1` and `SEG2` will appear at physical address `1000`.

PART 3: THE IAR XLIB LIBRARIAN

This part of the AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide contains the following chapters:

- *Introduction to the IAR XLIB Librarian*
- *XLIB options*
- *XLIB environment variables*
- *XLIB diagnostics*.

INTRODUCTION TO THE IAR XLIB LIBRARIAN

This chapter describes the IAR XLIB Librarian™, which enables you to manipulate the relocatable object files produced by the IAR Systems Assembler and Compiler.

Like the IAR XLINK Linker™, the IAR XLIB Librarian uses the UBROF standard object format (Universal Binary Relocatable Object Format).

LIBRARIES

A library is a single file that contains a number of relocatable object modules, each of which can be loaded independently from other modules in the file as it is needed.

Normally, the modules in a library file all have the `LIBRARY` attribute, which means that they will only be loaded by the linker if they are actually needed in the program. This is referred to as *demand loading* of modules.

On the other hand, a module with the `PROGRAM` attribute is *always* loaded when the file in which it is contained is processed by the linker.

A library file is no different from any other relocatable object file produced by the assembler or compiler, except that it includes a number of modules of the `LIBRARY` type.

USING LIBRARIES WITH C/EMBEDDED C++ PROGRAMS

All C/Embedded C++ programs make use of libraries, and the IAR Systems compilers are supplied with a number of standard library files.

Most C/Embedded C++ programmers will use the IAR XLIB Librarian at some point, for one of the following reasons:

- To replace or modify a module in one of the standard libraries. For example, the librarian can be used for replacing the distribution versions of the `CSTARTUP` and/or `putchar` modules with ones that you have customized.

- To add C/Embedded C++ or assembler modules to the standard library file so they will always be available whenever a C/Embedded C++ program is linked.
- To create custom library files that can be linked into their programs, as needed, along with the standard C/Embedded C++ library.

USING LIBRARIES WITH ASSEMBLER PROGRAMS

If you are only using assembler there is no need to use libraries. However, libraries provide the following advantages, especially when writing medium- and large-sized assembler applications:

- They allow you to combine utility modules used in more than one project into a simple library file. This simplifies the linking process by eliminating the need to include a list of input files for all the modules you need. Only the library module(s) needed for the program will be included in the output file.
- They simplify program maintenance by allowing multiple modules to be placed in a single assembler source file. Each of the modules can be loaded independently as a library module.
- They reduce the number of object files that make up an application, maintenance, and documentation.

You can create your assembly language library files using one of two basic methods:

- A library file can be created by assembling a single assembler source file which contains multiple library-type modules. The resulting library file can then be modified using XLIB.
- A library file can be produced by using XLIB to merge any number of existing modules together to form a user-created library.

The `NAME` and `MODULE` assembler directives are used for declaring modules as being of `PROGRAM` or `LIBRARY` type, respectively.

For additional information, see *Libraries*, page 121.

XLIB OPTIONS

This chapter summarizes the IAR XLIB Librarian™ options, classified according to their function, and gives a full syntactic and functional description of each librarian option.

SUMMARY OF XLIB OPTIONS

LIBRARY LISTING OPTIONS

The following table shows the library listing options:

Option	*Description*
`LIST-ALL-SYMBOLS`	Lists every symbol in modules.
`LIST-CRC`	Lists CRC values of modules.
`LIST-DATE-STAMPS`	Lists dates of modules.
`LIST-ENTRIES`	Lists `PUBLIC` symbols in modules.
`LIST-EXTERNALS`	Lists `EXTERN` symbols in modules.
`LIST-MODULES`	Lists modules.
`LIST-OBJECT-CODE`	Lists low-level relocatable code.
`LIST-SEGMENTS`	Lists segments in modules.

LIBRARY EDITING OPTIONS

The following table shows the library editing options:

Option	*Description*
`DELETE-MODULES`	Removes modules from a library.
`FETCH-MODULES`	Adds modules to a library.
`INSERT-MODULES`	Moves modules in a library.
`MAKE-LIBRARY`	Changes a module to library type.
`MAKE-PROGRAM`	Changes a module to program type.
`RENAME-ENTRY`	Renames `PUBLIC` symbols.
`RENAME-EXTERNAL`	Renames `EXTERN` symbols.

Option	*Description*
`RENAME-GLOBAL`	Renames `EXTERN` and `PUBLIC` symbols.
`RENAME-MODULE`	Renames one or more modules.
`RENAME-SEGMENT`	Renames one or more segments.
`REPLACE-MODULES`	Updates executable code.

MISCELLANEOUS LIBRARY OPTIONS

The following table shows miscellaneous library options:

Option	*Description*
`COMPACT-FILE`	Shrinks library file size.
`DEFINE-CPU`	Specifies CPU type.
`DIRECTORY`	Displays available object files.
`DISPLAY-OPTIONS`	Displays XLIB options.
`ECHO-INPUT`	Command file diagnostic tool.
`EXIT`	Returns to operating system.
`HELP`	Displays help information.
`ON-ERROR-EXIT`	Quits on a batch error.
`QUIT`	Returns to operating system.
`REMARK`	Comment in command file.

USING XLIB OPTIONS

The individual words of an identifier can be abbreviated to the limit of ambiguity. For example, `LIST-MODULES` can be abbreviated to `L-M`.

When running XLIB you can press Enter at any time to prompt for information, or display a list of the possible options.

Giving XLIB options from the command line

The `-c` command line option allows you to run XLIB options from the command line. Each argument specified after the `-c` option is treated as one XLIB option.

For example, specifying:

```
xlib -c "LIST-MOD math.r90" "LIST-MOD mod.r90 m.txt"
```

is equivalent to entering the following options in XLIB:

```
*LIST-MOD math.r90
*LIST-MOD mod.r90 m.txt
*QUIT
```

Note that each command line argument must be enclosed in double quotes if it includes spaces.

XLIB BATCH FILES

Running XLIB with a single command-line parameter specifying a file causes XLIB to read options from that file instead of from the console.

PARAMETERS

The following parameters are common to many of the XLIB options.

Parameter	*What it means*
objectfile	File containing object modules.
start, end	The first and last modules to be processed, in one of the following forms: *n* The *n*th module. $ The last module. *name* Module *name*. *name*+*n* The module *n* modules after *name*. $-*n* The module *n* modules before the last.
listfile	File to which a listing will be sent.
source	A file from which modules will be read.
destination	The file to which modules will be sent.

MODULE EXPRESSIONS

In most of the XLIB options you can or must specify a source module (like *`oldname`* in `RENAME-MODULE`), or a range of modules (*`startmodule, endmodule`*).

Internally in all XLIB operations modules are numbered from 1 in ascending order. Modules may be referred to by the actual name of the module, by the name plus or minus a relative expression, or by an absolute number. The latter is very useful when a module name is very long, unknown, or contains unusual characters such as space or comma.

The following table shows the available variations on module expressions:

Name	*Description*
`3`	The third module.
`$`	The last module.
`name``+4`	The module 4 modules after *`name`*.
`name``-12`	The module 12 modules before *`name`*.
`$-2`	The module 2 modules before the last module.

The option `LIST-MOD FILE,,$-2` will thus list the three last modules in `FILE` on the terminal.

LIST FORMAT

The `LIST` options give a list of symbols, where each symbol has one of the following prefixes:

Prefix	*Description*
`nn``.Pgm`	A program module with relative number *`nn`*.
`nn``.Lib`	A library module with relative number *`nn`*.
`Ext`	An external in the current module.
`Ent`	An entry in the current module.
`Loc`	A local in the current module.
`Rel`	A standard segment in the current module.
`Stk`	A stack segment in the current module.

Prefix	*Description*
`Com`	A common segment in the current module.

The following sections give full reference information for each XLIB option.

COMPACT-FILE

Shrinks library file size.

SYNTAX

`COMPACT-FILE` *`objectfile`*

DESCRIPTION

Use `COMPACT-FILE` to concatenate short, absolute records into longer records of variable length. This will decrease the size of a library file by about 5 %, in order to give library files which take up less time during the loader/linker process.

EXAMPLE

The following option compacts the file `maxmin.r90`:

```
COMPACT-FILE maxmin
```

This displays:

```
20 byte(s) deleted
```

DEFINE-CPU

Specifies CPU type.

SYNTAX

`DEFINE-CPU` *`cpu`*

PARAMETERS

`cpu`	The target processor.

DESCRIPTION

This option must be issued before any operations on object files can be done.

EXAMPLES

The following option defines the CPU as AVR:

```
DEF-CPU AVR
```

DELETE-MODULES

Removes modules from a library.

SYNTAX

`DELETE-MODULES` *`objectfile start end`*

DESCRIPTION

Use `DELETE-MODULES` to delete the specified modules.

EXAMPLES

The following option deletes module `2` from the file `math.r90`:

```
DEL-MOD math 2 2
```

DIRECTORY

Displays available object files.

SYNTAX

`DIRECTORY [`*`specifier`*`]`

DESCRIPTION

Use `DIRECTORY` to display on the terminal all files of the type that applies to the target processor. If no *`specifier`* is given, the current directory is listed.

EXAMPLES

The following option lists object files in the current directory:

```
DIR
```

It displays:

```
general      770
math         502
maxmin       375
```

DISPLAY-OPTIONS

Displays XLIB options.

SYNTAX

```
DISPLAY-OPTIONS [listfile]
```

DESCRIPTION

Use `DISPLAY-OPTIONS` to list on the *`listfile`* the names of all the CPUs which are recognized by this version of the IAR XLIB Librarian. The default file types of object files for the different CPUs are also listed. After that a list of all UBROF tags is output.

EXAMPLES

To list the options to the file `opts.lst`:

```
DISPLAY-OPTIONS opts
```

ECHO-INPUT

Command file diagnostic tool.

SYNTAX

```
ECHO-INPUT
```

DESCRIPTION

`ECHO-INPUT` is useful when debugging command files in batch mode because it makes all command input visible on the terminal. In the interactive mode it has no effect.

EXAMPLES

In a batch file

```
ECHO-INPUT
```

echoes all subsequent XLIB options.

EXIT

Returns to the operating system.

SYNTAX

```
EXIT
```

DESCRIPTION

Use `EXIT` to exit from XLIB after an interactive session.

EXAMPLES

To exit from XLIB:

```
EXIT
```

EXTENSION

Sets the default extension.

SYNTAX

```
EXTENSION
```

DESCRIPTION

Use `EXTENSION` to set the default extension.

FETCH-MODULES

Adds modules to a library.

SYNTAX

`FETCH-MODULES` *`source destination`* `[`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `FETCH-MODULES` to append the specified modules to the *`destination`* file. If *`destination`* already exists, it must be empty or contain valid object modules; otherwise it will be created.

EXAMPLES

The following option copies the module `mean` from `math.r90` to `general.r90`:

```
FETCH-MOD math general mean
```

HELP

Displays help information.

SYNTAX

```
HELP [option] [listfile]
```

PARAMETERS

option	Option for which help is displayed.

DESCRIPTION

If the `HELP` option is given with no parameters, a list of the available options will be displayed on the terminal. If a parameter is specified, all options which match the parameter will be displayed with a brief explanation of their syntax and function. A * matches all options. `HELP` output can be directed to any file.

EXAMPLES

For example, the option:

```
HELP LIST-MOD
```

displays:

```
LIST-MODULES <Object file> [<List file>] [<Start module>]
[<End module>]
    List the module names from [<Start module>] to
    [<End module>].
```

INSERT-MODULES

Inserts modules in a library.

SYNTAX

```
INSERT-MODULES objectfile start end {BEFORE | AFTER} dest
```

DESCRIPTION

Use `INSERT-MODULES` to move the specified modules before or after the `dest`.

EXAMPLES

The following option moves the module `mean` before the module `min` in the file `math.r90`:

```
INSERT-MOD math mean mean BEFORE min
```

LIST-ALL-SYMBOLS

Lists every symbol in modules.

SYNTAX

`LIST-ALL-SYMBOLS` *`objectfile`* `[`*`listfile`*`] [`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `LIST-ALL-SYMBOLS` to list all symbols (module names, segments, externals, entries, and locals) for the specified modules in the *`objectfile`*. They are listed to the *`listfile`*.

Each symbol is identified with a prefix; see *List Format*, page 208.

EXAMPLES

The following option lists all the symbols in `math.r90`:

```
LIST-ALL-SYMBOLS math
```

This displays:

```
1.  Lib  max
      Rel   CODE
      Ent   max
      Loc   A
      Loc   B
      Loc   C
      Loc   ncarry
2.  Lib  mean
      Rel   DATA
      Rel   CODE
      Ext   max
      Loc   A
      Loc   B
      Loc   C
      Loc   main
      Loc   start
```

```
3.  Lib  min
      Rel   CODE
      Ent   min
      Loc   carry
```

LIST-CRC

Lists CRC values of modules.

SYNTAX

`LIST-CRC` *`objectfile`* `[`*`listfile`*`] [`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `LIST-CRC` to list the module names and their associated CRCs for the specified modules.

Each symbol is identified with a prefix; see *List Format*, page 208.

EXAMPLES

The following option lists the CRCs for all modules in `math.r90`:

```
LIST-CRC math
```

This displays:

```
EC41            1.  Lib  max
ED72            2.  Lib  mean
9A73            3.  Lib  min
```

LIST-DATE-STAMPS

Lists dates of modules.

SYNTAX

`LIST-DATE-STAMPS` *`objectfile`* `[`*`listfile`*`] [`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `LIST-DATE-STAMPS` to list the module names and their associated generation dates for the specified modules.

Each symbol is identified with a prefix; see *List Format*, page 208.

EXAMPLES

The following option lists the date stamps for all the modules in `math.r90`:

```
LIST-DATE-STAMPS math
```

This displays:

```
15/Feb/98       1.  Lib  max
15/Feb/98       2.  Lib  mean
15/Feb/98       3.  Lib  min
```

LIST-ENTRIES

Lists `PUBLIC` symbols in modules.

SYNTAX

`LIST-ENTRIES` *`objectfile`* `[`*`listfile`*`] [`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `LIST-ENTRIES` to list the names and associated entries for the specified modules.

Each symbol is identified with a prefix; see *List Format*, page 208.

EXAMPLES

The following option lists the entries for all the modules in `math.r90`:

```
LIST-ENTRIES math
```

This displays:

```
1.  Lib  max
      Ent   max
2.  Lib  mean
3.  Lib  min
      Ent   min
```

LIST-EXTERNALS

Lists `EXTERN` symbols in modules.

SYNTAX

`LIST-EXTERNALS` *`objectfile`* `[`*`listfile`*`] [`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `LIST-EXTERNALS` to list the module names and associated externals for the specified modules.

Each symbol is identified with a prefix; see *List Format*, page 208.

EXAMPLES

The following option lists the externals for all the modules in `math.r90`:

```
LIST-EXT math
```

This displays:

```
1.  Lib  max
2.  Lib  mean
      Ext   max
3.  Lib  min
```

LIST-MODULES

Lists modules.

SYNTAX

`LIST-MODULES` *`objectfile`* `[`*`listfile`*`] [`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `LIST-MODULES` to list the module names for the specified modules.

Each symbol is identified with a prefix; see *List Format*, page 208.

EXAMPLES

The following option lists all the modules in `math.r90`:

```
LIST-MOD math
```

It produces the following output:

```
1.  Lib  max
2.  Lib  min
3.  Lib  mean
```

LIST-OBJECT-CODE

Lists low-level relocatable code.

SYNTAX

`LIST-OBJECT-CODE` *`objectfile`* `[`*`listfile`*`]`

DESCRIPTION

Use `LIST-OBJECT-CODE` to list the contents of the object file on the list file in ASCII format.

Each symbol is identified with a prefix; see *List Format*, page 208.

EXAMPLES

The following option lists the object code of `math.r90` to `object.lst`:

```
LIST-OBJECT-CODE math object
```

LIST-SEGMENTS

Lists segments in modules.

SYNTAX

`LIST-SEGMENTS` *`objectfile`* `[`*`listfile`*`] [`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `LIST-SEGMENTS` to list the module names and associated segments for the specified modules.

Each symbol is identified with a prefix; see *List Format*, page 208.

EXAMPLES

The following option lists the segments in the module `mean` in the file `math.r90`:

```
LIST-SEG math,,mean mean
```

Notice the use of two commas to skip the *`listfile`* parameter.

This produces the following output:

```
          2.  Lib  mean
                Rel   DATA
                Rel   CODE
```

MAKE-LIBRARY

Changes a module to library type.

SYNTAX

`MAKE-LIBRARY` *`objectfile`* `[`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `MAKE-LIBRARY` to change the module header attributes to conditionally loaded for the specified modules.

EXAMPLES

The following option converts all the modules in `main.r90` to library modules:

```
MAKE-LIB main
```

MAKE-PROGRAM

Changes a module to program type.

SYNTAX

`MAKE-PROGRAM` *`objectfile`* `[`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `MAKE-PROGRAM` to change the module header attributes to unconditionally loaded for the specified modules.

EXAMPLES

The following option converts module `start` in `main.r90` into a program module:

```
MAKE-PROG main start
```

ON-ERROR-EXIT

Quits on a batch error.

SYNTAX

```
ON-ERROR-EXIT
```

DESCRIPTION

Use `ON-ERROR-EXIT` to make the librarian abort if an error is found. It is suited for use in batch mode.

EXAMPLES

The following batch file aborts if the `FETCH-MODULES` option fails:

```
ON-ERROR-EXIT
FETCH-MODULES math new
```

QUIT

Returns to the operating system.

SYNTAX

```
QUIT
```

DESCRIPTION

Use `QUIT` to exit and return to the operating system.

EXAMPLES

To quit from XLIB:

```
QUIT
```

REMARK

Comment in command file.

SYNTAX

```
REMARK text
```

DESCRIPTION

Use `REMARK` to include a comment.

EXAMPLES

The following example illustrates the use of a comment in an XLIB command file:

```
REM Now compact file
COMPACT-FILE math
```

RENAME-ENTRY

Renames `PUBLIC` symbols.

SYNTAX

`RENAME-ENTRY` *`objectfile old new`* `[`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `RENAME-ENTRY` to rename all occurrences of an entry from *`old`* to *`new`* in the specified modules.

EXAMPLES

The following option renames the entry for modules `2` to `4` in `math.r90` from `mean` to `average`:

```
RENAME-ENTRY math mean average 2 4
```

RENAME-EXTERNAL

Renames `EXTERN` symbols.

SYNTAX

`RENAME-EXTERN` *`objectfile old new`* `[`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `RENAME-EXTERN` to rename all occurrences of an external symbol from *`old`* to *`new`* in the specified modules.

EXAMPLES

The following option renames all external symbols in `math.r90` from `error` to `err`:

```
RENAME-EXT math error err
```

RENAME-GLOBAL

Renames `EXTERN` and `PUBLIC` symbols.

SYNTAX

```
RENAME-GLOBAL objectfile old new [start] [end]
```

DESCRIPTION

Use `RENAME-GLOBAL` to rename all occurrences of an external symbol or entry from *`old`* to *`new`* in the specified modules.

EXAMPLES

The following option renames all occurrences of `mean` to `average` in `math.r90`:

```
RENAME-GLOBAL math mean average
```

RENAME-MODULE

Renames one or more modules.

SYNTAX

```
RENAME-MODULE objectfile old new
```

DESCRIPTION

Use `RENAME-MODULE` to rename a module. Notice that if there is more than one module with the name *`old`*, only the first one encountered is changed.

EXAMPLES

The following example renames the module `average` to `mean` in the file `math.r90:`

```
RENAME-MOD math average mean
```

RENAME-SEGMENT

Renames one or more segments.

SYNTAX

```
RENAME-SEGMENT objectfile old new [start] [end]
```

DESCRIPTION

Use `RENAME-SEGMENT` to rename all occurrences of a segment from the name *`old`* to *`new`* in the specified modules.

EXAMPLES

The following example renames all `CODE` segments to `ROM` in the file `math.r90`:

```
RENAME-SEG math CODE ROM
```

REPLACE-MODULES

Updates executable code.

SYNTAX

`REPLACE-MODULES` *`source destination`*

DESCRIPTION

Use `REPLACE-MODULES` to replace modules with the same name from *`source`* to *`destination`*. All replacements are logged on the terminal. The main application for this option is to update large run-time libraries etc.

EXAMPLES

The following example replaces modules in `math.r90` with modules from `newmath.r90`:

```
REPLACE-MOD newmath math
```

This displays:

```
Replacing module 'max'
Replacing module 'mean'
Replacing module 'min'
```

XLIB ENVIRONMENT VARIABLES

The IAR XLIB Librarian™ supports a number of environment variables. These can be used for creating defaults for various XLIB options so that they do not have to be specified on the command line.

SUMMARY OF XLIB ENVIRONMENT VARIABLES

The following environment variables can be used by XLIB:

Environment variable	*Description*
`XLIB_COLUMNS`	Sets the number of columns.
`XLIB_CPU`	Sets the CPU type.
`XLIB_PAGE`	Sets the number of lines per page.
`XLIB_SCROLL_BREAK`	Sets the scroll pause in number of lines.

XLIB_COLUMNS

Sets the number of columns.

DESCRIPTION

Use `XLIB_COLUMNS` to set the number of columns for listings (80–132). The default is 80 columns.

EXAMPLE

To set the number of columns to 132:

```
set XLIB_COLUMNS=132
```

XLIB_CPU

Sets the CPU type.

DESCRIPTION

Use `XLIB_CPU` to set the CPU type so that the `DEFINE-CPU` option does not need to be entered at the beginning of an XLIB session.

EXAMPLE

To set the CPU type to AVR:

```
set XLIB_CPU=avr
```

XLIB_PAGE

Sets the number of lines per page.

DESCRIPTION

Use `XLIB_PAGE` to set the number of lines per page (10–100) for the list file. The default is a listing without page breaks.

EXAMPLE

To set the number of lines per page to 66:

```
set XLIB_PAGE=66
```

XLIB_SCROLL_BREAK

Sets the scroll pause.

DESCRIPTION

Use `XLIB_SCROLL_BREAK` to make the XLIB output pause and wait for the Enter key to be pressed after the specified number of lines (16–100) on the screen have scrolled by.

EXAMPLE

To pause every 22 lines:

```
set XLIB_SCROLL_BREAK=22
```

XLIB DIAGNOSTICS

This chapter lists the messages produced by the IAR XLIB Librarian™.

XLIB MESSAGES

The following section lists the XLIB messages. Options flagged as erroneous never alter object files.

1 Bad object file, EOF encountered

Bad or empty object file, which could be the result of an aborted assembly or compilation.

2 Unexpected EOF in batch file

The last command in a command file must be `EXIT`.

3 Unable to open file *`file`*

Could not open the command file or, if `ON-ERROR-EXIT` has been specified, this message is issued on any failure to open a file.

4 Variable length record out of bounds

Bad object module, could be the result of an aborted assembly.

5 Missing or non-default parameter

A parameter was missing in the direct mode.

6 No such CPU

A list with the possible choices is displayed when this error is found.

7 CPU undefined

`DEFINE-CPU` must be issued before object file operations can begin. A list with the possible choices is displayed when this error is found.

8 Ambiguous CPU type

A list with the possible choices is displayed when this error is found.

9 No such command

Use the `HELP` option.

10 Ambiguous command

Use the `HELP` option.

11 Invalid parameter(s)

Too many parameters or a misspelled parameter.

12 Module out of sequence

Bad object module, could be the result of an aborted assembly.

13 Incompatible object, consult distributor!

Bad object module, could be the result of an aborted assembly, or that the assembler/compiler revision used is incompatible with the version of XLIB used.

14 Unknown tag: hh

Bad object module, could be the result of an aborted assembly.

15 Too many errors

More than 32 errors will make XLIB abort.

16 Assembly/compilation error?

The `T_ERROR` tag was found. Edit and re-assemble/re-compile your program.

17 Bad CRC, hhhh expected

Bad object module; could be the result of an aborted assembly.

18 Can’t find module: xxxxx

Check the available modules with `LIST-MOD` file.

19 Module expression out of range

Module expression is less than one or greater than `$`.

20 Bad syntax in module expression: xxxxx

The syntax is invalid.

21 Illegal insert sequence

The specified destination in the `INSERT-MODULES` option must not be within the *`start-end`* sequence.

22 < End module > found before < Start module > !

Source module range must be from low to high order.

23 Before or after!

Bad `BEFORE/AFTER` specifier in the `INSERT-MODULES` option.

24 Corrupt file, error occurred in *`tag`*

A fault is detected in the object file *`tag`*. Reassembly or recompilation may help. Otherwise contact your supplier.

25 *`File`* **is write protected**

The file *`file`* is write protected and cannot be written to.

26 Non-matching replacement module name found in source file

In the source file, a module *`name`* with no corresponding entry in the destination file was found.