IAR Assemblers

Reference Guide

for the Renesas
78K0/78K0S and 78K0R Microcontroller Subfamilies

A78K-3

IAR SYSTEMS

EDITION NOTICE

Third edition: May 2010

Part number: A78K-3

This guide applies to version 4.7 of IAR Embedded Workbench® for Renesas Electronics' 78K0/78K0S and 78K0R Microcontroller Subfamilies.

Internal reference: AFE1/AFE2, R13, IMAE.

Contents

Tables

Preface

Welcome to the IAR Assemblers Reference Guide for 78K. The purpose of this guide is to provide you with detailed reference information that can help you to use the IAR Assemblers for 78K to develop your application according to your requirements.

Because of important differences in architecture between the 78K0 and 78K0S microcontroller cores and the 78K0R microcontroller core, IAR Embedded Workbench for Renesas Electronics' 78K Microcontroller Subfamilies includes two separate assemblers: the 78K0/78K0S Assembler and the 78K0R Assembler. In those cases where the assemblers behave the same way, both assemblers will be referred to together as the IAR Assemblers for 78K in this guide.

Who should read this guide

You should read this guide if you plan to develop an application, or part of an application, using assembler language for the 78K microcontrollers and need to get detailed reference information on how to use the IAR Assemblers. In addition, you should have working knowledge of the following:

- The architecture and instruction set of the 78K microcontrollers. Refer to the documentation from Renesas for information about the 78K microcontrollers
- General assembler language programming
- Application development for embedded systems
- The operating system of your host computer.

How to use this guide

When you first begin using the IAR Assemblers, you should read the chapter *Introduction to the IAR Assemblers for 78K* in this reference guide.

If you are an intermediate or advanced user, you can focus more on the reference chapters that follow the introduction. Note that *Part 1. Common reference* applies to both the 78K0/78K0S Assembler and the 78K0R Assembler, but that information which is specific to either of the assemblers is described separately in *Part 2. 78K0/78K0S Assembler reference* and *Part 3. 78K0R Assembler reference*, respectively.

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

What this guide contains

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

Part 1. Common reference

This section provides a general overview of the IAR Assemblers and gives reference information about functionality that applies to both the 78K0/78K0S Assembler and the 78K0R Assembler.

- *Introduction to the IAR Assemblers for 78K* provides programming information. It also describes the source code format, and the format of assembler listings.
- *Assembler directives* gives an alphabetical summary of the assembler directives, and provides detailed reference information about each of the directives, classified into groups according to their function.
- *Diagnostics* contains information about the formats and severity levels of diagnostic messages.

Part 2. 78K0/78K0S Assembler reference

This section gives reference information specific to 78K0/78K0S Assembler:

- *78K0/78K0S Assembler options* first explains how to set the 78K0/78K0S Assembler options from the command line and how to use environment variables. It then gives an alphabetical summary of the assembler options, and contains detailed reference information about each option.
- *78K0/78K0S Assembler operators* gives a summary of the 78K0/78K0S Assembler operators, arranged in order of precedence, and provides detailed reference information about each operator.

Part 3. 78K0R Assembler reference

This section gives reference information specific to 78K0R Assembler:

- *78K0R Assembler options* first explains how to set the 78K0R Assembler options from the command line and how to use environment variables. It then gives an alphabetical summary of the assembler options, and contains detailed reference information about each option.

- *78K0R Assembler operators* gives a summary of the 78K0R Assembler operators, arranged in order of precedence, and provides detailed reference information about each operator.
- *78K0R pragma directives* describes the pragma directives available in the 78K0R Assembler.

Other documentation

The complete set of IAR Systems development tools for the 78K microcontrollers is described in a series of guides. For information about:

- Using the IAR Embedded Workbench® IDE with the IAR C-SPY® Debugger, refer to the *IAR Embedded Workbench® IDE User Guide*
- Programming for the IAR C/C++ Compilers for 78K, refer to the *IAR C/C++ Compilers Reference Guide for 78K*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR DLIB Library, refer to the online help system
- Using the IAR CLIB Library, refer to the *IAR C Library Functions Reference Guide*, available from the online help system
- Debugging your applications using one of the hardware debugger systems, refer to the *IAR C-SPY® Hardware Debugger Systems User Guide for 78K*
- Developing safety-critical applications using the MISRA C guidelines, refer to the *IAR Embedded Workbench® MISRA C Reference Guide*
- Porting application code and projects created with a previous IAR Embedded Workbench IDE for 78K, refer to the *78K IAR Embedded Workbench® Migration Guide*.

All of these guides are delivered in hypertext PDF or HTML format on the installation media.

Document conventions

This guide uses the following typographic conventions:

Style	Used for
`computer`	Text that you enter or that appears on the screen.
`parameter`	A label representing the actual value you should enter as part of a command.
`{option}`	A mandatory part of a command.
`[option]`	An optional part of a command.

Table 1: Typographic conventions used in this guide

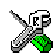
Style	Used for
`a|b|c`	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
reference	A cross-reference within this guide or to another guide.
…	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench interface.
	Identifies instructions specific to the command line interface.

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

Part 1. Common reference

This part of the guide is common to both the assembler for the 78K0 and 78K0S cores and the assembler for the 78K0R core. It includes the following chapters:

- Introduction to the IAR Assemblers for 78K
- Assembler directives
- Diagnostics.

Introduction to the IAR Assemblers for 78K

This chapter contains the following sections:

- Supported 78K derivatives
- Introduction to assembler programming
- Modular programming
- Assembling
- Source format
- Assembler instructions
- Expressions, operands, and operators
- List file format
- Programming hints.

Supported 78K derivatives

The IAR Assemblers for 78K support all derivatives based on the standard Renesas 78K0, 78K0R, and 78K0S microcontroller cores. The support has been implemented in the form of separate assembler executable files for the 78K0 and 78K0S cores on the one hand and the 78K0R core on the other. The two assemblers behave identically in most respects but differ on some points, most notably in that pragma directives are only available for the 78K0R Assembler.

Note: Except for those cases where the assemblers behave differently, both assemblers will be referred to as the IAR Assemblers for 78K in this guide.

Introduction to assembler programming

Even if you do not intend to write a complete application in assembler language, there may be situations where you will find it necessary to write parts of the code in assembler, for example when using mechanisms in the 78K microcontroller that require precise timing and special instruction sequences.

To write efficient assembler applications, you should be familiar with the architecture and instruction set of the 78K microcontroller. Refer to Renesas' hardware documentation for syntax descriptions of the instruction mnemonics.

GETTING STARTED

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

- Work through the tutorials—especially the one about mixing C and assembler modules—that you find in the *IAR Embedded Workbench® IDE User Guide*
- Read about the assembler language interface—also useful when mixing C and assembler modules—in the *IAR C/C++ Compilers Reference Guide for 78K*
- In the IAR Embedded Workbench IDE, you can base a new project on a *template* for an assembler project.

Modular programming

Typically, you write your assembler code in assembler source files. In each source file, you define one or several assembler *modules* by using the module control directives. By structuring your code in small modules—in contrast to one single monolithic module—you can organize your application code in a logical structure, which makes the code easier to understand, and which benefits:

- an efficient program development
- reuse of modules
- maintenance.

Each module has a name and a type, where the type can be either `PROGRAM` or `LIBRARY`. The linker will always include a `PROGRAM` module, whereas a `LIBRARY` module is only included in the linked code if other modules reference a public symbol in the module. A module consists of one or more segments.

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

Assembling

In the command line interface, the following line assembles the source file `myfile.s26` into the object file `myfile.r26` using the default settings:

The 78K0/78K0S Assembler:	`a78k myfile.s26`
The 78K0R Assembler:	`a78k0r myfile.s26`

Source format

The format of an assembler source line is as follows:

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

where the components are as follows:

label — A definition of a label, which is a symbol that represents an address. If the label starts in the first column—that is, to the leftmost on the line—the : (colon) is optional.

operation — An assembler instruction or directive. This must not start in the first column—there must be some whitespace to the left of it.

operands — An assembler instruction or directive can have zero, one, or more operands. The operands are separated by commas. An operand can be:
- a constant representing a numeric value or an address
- a symbolic name representing a numeric value or an address (where the latter also is referred to as a label)
- a register
- a predefined symbol
- the program location counter (PLC)
- an expression.

comment — Comment, preceded by a ; (semicolon)
C or C++ comments are also allowed.

The components are 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; that is, to columns 8, 16, 24 etc.

The IAR Assemblers for 78K use the default filename extensions `s26`, `asm`, and `msa` for source files.

Assembler instructions

The IAR Assemblers for 78K support the syntax for assembler instructions as described in the chip manufacturer's hardware documentation, with the following exceptions:

The instruction operators `$` and `$!` are not permitted before a PC-relative address, and `!` and `!!` are not permitted before an absolute address.

Note: See also *Operand modifiers for 78K0/78K0S*, page 32, and *Operand modifiers for 78K0R*, page 32.

Expressions, operands, and operators

Expressions consist of expression 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. Range checking is performed if a value is used for generating code.

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators; see *Precedence of assembler operators*, page 89 (78K0/78K0S) or *Precedence of assembler operators*, page 123 (78K0R). The valid operators are described in the chapters *78K0/78K0S Assembler operators* and *78K0R Assembler operators*.

The following operands are valid in an expression:

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

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

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`, `01234`
Decimal	`1234`, `-1`, `d'1234`, `1234d`
Hexadecimal	`0FFFFh`, `0xFFFF`, `h'FFFF`

Table 2: Integer constant formats

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

ASCII CHARACTER CONSTANTS

ASCII constants can consist of any number of 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).
"" (2 double quotes)	Empty string (an ASCII null character).
\'	', for quote within a string, as in 'I\'d love to'
\\	\, for \ within a string
\"	", for double quote within a string

Table 3: ASCII character constant formats

FLOATING-POINT CONSTANTS

The IAR Assemblers for 78K will accept floating-point values as constants and convert them into IEEE single-precision (78K0/78K0S: signed 32-bit; 78K0R: signed 64-bit) floating-point format or fractional format.

Floating-point numbers can be written in the format:

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

The following table shows some valid examples:

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

Table 4: Floating-point constants

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

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

TRUE AND FALSE

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

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

SYMBOLS

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

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

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

Case is insignificant for built-in symbols like instructions, registers, operators, and directives. For user-defined symbols case is by default significant, but can be turned on and off using the **Case sensitive user symbols** (78K0/78K0S: `-s`; 78K0R: `--case_insensitive`) assembler option. See *-s*, page 86 or *--case_insensitive*, page 109, respectively, for additional information.

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

LABELS

Symbols used for memory locations are referred to as labels.

Program location counter (PLC)

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

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

```
         BR      $      ; Loop forever
```

BIT VARIABLES AS OPERANDS

In the 78K0/78K0S Assembler—but not in the 78K0R Assembler—it is possible to define and use relocatable bit variables as shown in the following example:

```
         NAME    MODZERO ; file 0
         PUBLIC  bitzero
         PUBLIC  bitone
         PUBLIC  biteight

         RSEG    bits
         SADDR
bitzero  DS      1
bitone   DS      1
         DS      6
biteight DS      1
         RSEG    code
startz
         MOV1    CY,bitzero
         END

         NAME    modone  ; file 1
         EXTERN  bitzero
         EXTERN  bitone
         EXTERN  biteight
         RSEG    code
start
         MOV1    CY,bitzero
         MOV1    CY,bitone
         MOV1    CY,biteight
         END
```

Link this with the following command:

```
XLINK -c78000 -Z(BIT)bits=0 -Z(CODE)code=3000 modone modzero
```

The symbol `bitzero` will now refer to bit 0 on address `FE20h`, and `biteight` will refer to bit 0 at address `FE21h`. To put the relocatable bits anywhere else in the short addressed area, use the following link command:

```
-Z(BIT)bits=(shortaddress - FE20h)*8 + bitposition
```

REGISTER SYMBOLS

The following table shows the existing predefined register symbols:

Name	Register size	Description
`R0–R7`	8 bits	Byte registers
`RP0-RP3`	16 bits	Word registers
`PC`	16 bits (78K0/78K0S) 20 bits (78K0R)	Program counter
`SP`	16 bits	Stack pointer
`PSW`	8 bits	Processor status
`CS`	8 bits	Code segment registers (78K0R only)
`ES`	8 bits	Data segment registers (78K0R only)

Table 5: Predefined register symbols

PREDEFINED SYMBOLS

The IAR Assemblers for 78K define a set of symbols for use in assembler source files. The symbols provide information about the current assembly, allowing you to test them in preprocessor directives or include them in the assembled code. The strings returned by the assembler are enclosed in double quotes.

The following predefined symbols are available:

Symbol	Value
`__A78K__`	Target identity.
`__BUILD_NUMBER__`	A unique integer that identifies the build number of the assembler currently in use. The build number does not necessarily increase with an assembler that is released later.
`__CORE__`	Processor core: 0 / `__78K0_BASIC__` 1 / `__78K0__` 2 / `__78K0S__` 3 / `__78K0R__`
`__DATE__`	The current date in Mmm dd yyyy format (string).

Table 6: Predefined symbols

Symbol	Value
`__FILE__`	The name of the current source file (string).
`__IAR_SYSTEMS_ASM__`	IAR assembler identifier (number).
`__LINE__`	The current source line number (number).
`__TID__`	Target identity, consisting of two bytes (number). The value for the IAR Assemblers for 78K is `0x1E30`. The high byte is the target identity, `30` for the IAR Assemblers for 78K. The low byte is the processor option *16. The use of `__TID__` is not recommended. We recommend that you use the symbols `__A78K__` and `__CORE__` instead.
`__SUBVERSION__`	An integer that identifies the version letter of the version number, for example the C in 4.21C, as an ASCII character.
`__TIME__`	The current time in `hh:mm:ss` format (string).
`__VER__`	The version number in integer format; for example, version 4.17 is returned as 417 (number).

Table 6: Predefined symbols (Continued)

Including symbol values in code

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

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

```
timdat DC8    __TIME__,",",__DATE__,0 ; time and date
       ...
       MOVW   RP0,#timdat              ;  load address of string
       CALL   printstring              ;  routine to print string
```

Testing symbols for conditional assembly

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

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

```
#if (__CORE__==__78K0__)
...
...
#else
...
```

```
…
#endif
```

See *Conditional assembly directives*, page 37.

ABSOLUTE AND RELOCATABLE EXPRESSIONS

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

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:  DC8     5
second: DC8     3
        ENDMOD
        MODULE  prog2
        RSEG    CODE
start   …
```

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

```
        DC8     first
        DC8     first+1
        DC8     1+first
        DC8     (first/second)*third
```

Note: At the time of assembly, 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.

EXPRESSION RESTRICTIONS

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

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

No forward

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

No external

No external references in the expression are allowed.

Absolute

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

Fixed

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

List file format

The format of an assembler list file is as follows:

HEADER

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

BODY

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

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

SUMMARY

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

SYMBOL AND CROSS-REFERENCE TABLE

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

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

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

Table 7: Symbol and cross-reference table

Programming hints

This section gives hints on how to write efficient code for the IAR Assemblers for 78K. For information about projects including both assembler and C or C++ source files, see the *IAR C/C++ Compilers Reference Guide for 78K*.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of 78K derivatives are included in the IAR Systems product package, in the `\78k\inc` directory. These header files define the processor-specific special function registers (SFRs) and interrupt vector numbers.

The header files are intended to be used also with the IAR C/C++ Compiler for 78K, and they are suitable to use as templates when creating new header files for other 78K derivatives.

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

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

USING C-STYLE PREPROCESSOR DIRECTIVES

The C-style preprocessor directives are processed before other assembler directives. Therefore, do not use preprocessor directives in macros and do not mix them with assembler-style comments. For more information about comments, see *Assembler control directives*, page 56.

Assembler directives

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

Summary of assembler directives

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

Directive	Description	Section
`_args`	Is set to number of arguments passed to macro.	Macro processing
`$`	Includes a file. 78K0/78K0S only.	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
`#line`	Changes the line numbers.	C-style preprocessor
`#message`	Generates a message on standard output. 78K0/78K0S only.	C-style preprocessor
`#pragma`	Controls extension features. 78K0R only.	C-style preprocessor
`#undef`	Undefines a label.	C-style preprocessor
`/*comment*/`	C-style comment delimiter.	Assembler control
`//`	C++ style comment delimiter.	Assembler control
`=`	Assigns a permanent value local to a module.	Value assignment
`ALIAS`	Assigns a permanent value local to a module.	Value assignment
`ALIGN`	Aligns the program location counter by inserting zero-filled bytes.	Segment control
`ALIGNRAM`	Aligns the program location counter.	Segment control

Table 8: Assembler directives summary

Directive	Description	Section
ARGFRAME	Declares the space used for the arguments to a function.	Function
ASEG	Begins an absolute segment.	Segment control
ASEGN	Begins a named absolute segment.	Segment control
ASSIGN	Assigns a temporary value.	Value assignment
BLOCK	Defines type information for a symbol.	Recognized for future compatibility
CASEOFF	Disables case sensitivity.	Assembler control
CASEON	Enables case sensitivity.	Assembler control
CFI	Specifies call frame information.	Call frame information
COL	Sets the number of columns per page (78K0/78K0S). Retained in 78K0R for backward compatibility reasons.	Listing control
COMMON	Begins a common segment.	Segment control
CONST	Specifies an SFR label as read-only. 78K0/78K0S only.	Value assignment
DB	Generates 8-bit constants, including strings.	Data definition or allocation
DC8	Generates 8-bit constants, including strings.	Data definition or allocation
DC16	Generates 16-bit constants.	Data definition or allocation
DC24	Generates 24-bit constants.	Data definition or allocation
DC32	Generates 32-bit constants.	Data definition or allocation
DC64	Generates 64-bit constants. 78K0R only.	Data definition or allocation
DD	Generates 32-bit constants.	Data definition or allocation
DEFINE	Defines a file-wide value.	Value assignment
DF32	Generates 32-bit floating-point constants. 78K0R only.	Data definition or allocation

Table 8: Assembler directives summary (Continued)

Directive	Description	Section
DF64	Generates 64-bit floating-point constants. 78K0R only.	Data definition or allocation
DP	Generates 24-bit constants.	Data definition or allocation
DS	Allocates space for 8-bit integers.	Data definition or allocation
DS8	Allocates space for 8-bit integers.	Data definition or allocation
DS16	Allocates space for 16-bit integers. 78K0R only.	Data definition or allocation
DS24	Allocates space for 24-bit integers. 78K0R only.	Data definition or allocation
DS32	Allocates space for 32-bit integers. 78K0R only.	Data definition or allocation
DS64	Allocates space for 64-bit integers. 78K0R only.	Data definition or allocation
DW	Generates 16-bit constants.	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
ENDMAC	Exits prematurely from a macro.	Macro processing
ENDMOD	Terminates the assembly of the current module.	Module control
ENDR	Ends a repeat structure	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment
EVEN	Aligns the program counter to an even address.	Segment control
EXITM	Exits prematurely from a macro.	Macro processing
EXPORT	Exports symbols to other modules.	Symbol control
EXTERN	Imports an external symbol.	Symbol control

Table 8: Assembler directives summary (Continued)

Directive	Description	Section
FUNCALL	Declares that the function *caller* calls the function *callee*.	Function
FUNCTION	Declares a label name to be a function.	Function
IF	Assembles instructions if a condition is true.	Conditional assembly
IMPORT	Imports an external symbol.	Symbol control
LIBRARY	Begins a library module.	Module control
LIMIT	Checks a value against limits.	Value assignment
LOCAL	Creates symbols local to a macro.	Macro processing
LOCFRAME	Declares the space used for the locals in a function.	Function
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 assembler-listing output.	Listing control
LSTPAG	Controls the formatting of output into pages (78K0/78K0S). Retained in 78K0R for backward compatibility reasons.	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 location counter to an odd address.	Segment control
ORG	Sets the program location counter.	Segment control
OVERLOAD	Overloaded class name.	Recognized for future compatibility
PAGE	Generates a new page (78K0/78K0S). Retained in 78K0R for backward compatibility reasons.	Listing control
PAGSIZ	Sets the number of lines per page (78K0/78K0S). Retained in 78K0R for backward compatibility reasons.	Listing control

Table 8: Assembler directives summary (Continued)

Directive	Description	Section
PROGRAM	Begins a program module.	Module control
PUBLIC	Exports symbols to other modules.	Symbol control
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.	Symbol control
RADIX	Sets the default base.	Assembler control
REPT	Assembles instructions a specified number of times.	Macro processing
REPTC	Repeats and substitutes characters.	Macro processing
REPTI	Repeats and substitutes strings.	Macro processing
REQUIRE	Forces a symbol to be referenced.	Symbol control
RSEG	Begins a relocatable segment.	Segment control
RTMODEL	Declares runtime model attributes.	Module control
SADDR	Begins a short address relocatable segment. 78K0/78K0S only.	Segment control
SET	Assigns a temporary value.	Value assignment
sfr	Creates byte-access SFR labels. 78K0/78K0S only.	Value assignment
sfrp	Creates word-access SFR labels. 78K0/78K0S only.	Value assignment
SFRTYPE	Specifies SFR attributes. 78K0/78K0S only.	Value assignment
SHORTAD	Begins a short address relocatable segment. 78K0/78K0S only.	Segment control
STACK	Begins a stack segment. 78K0/78K0S only.	Segment control
SYMBOL	Defines part of a class name.	Recognized for future compatibility
VAR	Assigns a temporary value.	Value assignment

Table 8: Assembler directives summary (Continued)

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. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
`END`	Terminates the assembly of the last module in a file.	Only locally defined labels or integer constants
`ENDMOD`	Terminates the assembly of the current module.	Only locally defined labels or integer constants
`LIBRARY`	Begins a library module.	No external references Absolute
`MODULE`	Begins a library module.	No external references Absolute
`NAME`	Begins a program module.	No external references Absolute
`PROGRAM`	Begins a program module.	No external references Absolute
`RTMODEL`	Declares runtime model attributes.	Not applicable

Table 9: Module control directives

SYNTAX

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

PARAMETERS

address — An optional expression that determines the start address of the program. It can take any positive integer value.

expr — An optional expression used by the compiler to encode the runtime options. It must be within the range 0-255 and evaluate to a constant value. The expression is only meaningful if you are assembling source code that originates as assembler output from the compiler.

key	A text string specifying the key.
symbol	Name assigned to module, used by XLINK, XAR, and XLIB when processing object files.
value	A text string specifying the value.

DESCRIPTIONS

Beginning a program module

Use `NAME` or `PROGRAM` to begin a program module, and to assign a name for future reference by the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian.

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

Beginning a library module

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

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

Terminating a module

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

Terminating the source file

Use `END` to indicate the end of the source file. Any lines after the `END` directive are ignored. The `END` directive also terminates the last module in the file, if this is not done explicitly with an `ENDMOD` directive.

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 placed after the *last* module, and there must not be any source lines (except for comments and listing control directives) between an ENDMOD and a MODULE directive.

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

Declaring runtime model attributes

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

A module can have several runtime model definitions.

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

If you are writing assembler routines for use with C or C++ code, and you want to control the module consistency, refer to the *IAR C/C++ Compilers Reference Guide for 78K*.

Examples

The following example defines three modules where:

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

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

MODULE MOD_2
```

```
    RTMODEL    "foo", "2"
    RTMODEL    "bar", "*"
    ...
  ENDMOD

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

Symbol control directives

These directives control how symbols are shared between modules.

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

Table 10: Symbol control directives

SYNTAX

EXPORT *symbol* [,*symbol*] …
EXTERN *symbol*[:SADDR] [,*symbol*[:SADDR]] …
IMPORT *symbol* [,*symbol*] …
PUBLIC *symbol* [,*symbol*] …
PUBWEAK *symbol* [,*symbol*] …
REQUIRE *symbol*

PARAMETERS

symbol	Symbol to be imported or exported.

DESCRIPTIONS

Exporting symbols to other modules

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

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

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

Exporting symbols with multiple definitions to other modules

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

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

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

Importing symbols

Use EXTERN to import an untyped external symbol. Add the :SADDR suffix to the symbol to indicate that it resides in the short address area—78K0/78K0S only.

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

EXAMPLES

The following example defines a subroutine to print an error message, and exports the entry address err so that it can be called from other modules.

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

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

```
        NAME        error
        EXTERN      print
        PUBLIC      err
```

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

     END
```

Segment control directives

The segment directives control how code and data are located. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
ALIGN	Aligns the program location counter by inserting zero-filled bytes.	No external references Absolute
ALIGNRAM	Aligns the program location counter.	No external references Absolute
ASEG	Begins an absolute segment.	No external references Absolute
ASEGN	Begins a named absolute segment.	No external references Absolute
COMMON	Begins a common segment.	No external references Absolute
EVEN	Aligns the program counter to an even address.	No external references Absolute
ODD	Aligns the program counter to an odd address.	No external references Absolute
ORG	Sets the location counter.	No external references Absolute (see below)
RSEG	Begins a relocatable segment.	No external references Absolute
SADDR	Begins a short address relocatable segment. 78K0/78K0S only.	No external references Absolute
SHORTAD	Equivalent to SADDR (provided for backward compatibility). 78K0/78K0S only.	No external references Absolute
STACK	Begins a stack segment. 78K0/78K0S only.	No external references Absolute

Table 11: Segment control directives

SYNTAX

```
ALIGN align [,value]
ALIGNRAM align
ASEG [start]
ASEGN segment [:type], address
COMMON segment [:type] [flag] [(align)]
EVEN [value]
ODD [value]
ORG expr
RSEG segment [:type] [flag] [(align)]
SADDR
SHORTAD
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`, `ROOT`
	`NOROOT` means that the segment part is discarded by the linker if no symbols in this segment part are referred to. Normally all segment parts except startup code and interrupt vectors should set this flag. The default mode is `ROOT` which indicates that the segment part must not be discarded.
	
	`REORDER`, `NOREORDER`
	`REORDER` allows the linker to reorder segment parts. For a given segment, all segment parts must specify the same state for this flag. The default mode is `NOREORDER` which indicates that the segment parts must remain in order.
	
	`SORT`, `NOSORT`
	`SORT` means that the linker will sort the segment parts in decreasing alignment order. For a given segment, all segment parts must specify the same state for this flag. The default mode is `NOSORT` which indicates that the segment parts will not be sorted.

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

type	The memory type, typically CODE or DATA. In addition, any of the types supported by the IAR XLINK Linker.
value	Byte value used for padding, default is zero.

DESCRIPTIONS

Beginning an absolute segment

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

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

Beginning a named absolute segment

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

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

Beginning a relocatable segment

Use RSEG to set the current mode of the assembly to relocatable assembly mode. The assembler maintains separate 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 (78K0/78K0S only)

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

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

Beginning a common segment

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

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

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

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

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

Setting the program location counter (PLC)

Use ORG to set the program location counter of the current segment to the value of an expression. The optional parameter will assume the value and type of the new location counter. When ORG is used in an absolute segment (ASEG), the parameter expression must be absolute. However, when ORG is used in a relative segment (RSEG), the expression may be either absolute or relative (and the value is interpreted as an offset relative to the segment start in both cases).

The program location counter is set to zero at the beginning of an assembler module.

Aligning a segment

Use ALIGN to align the program location counter to a specified address boundary. The expression gives the power of two to which the program counter should be aligned and the permitted range is 0 to 8.

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

ALIGN aligns by inserting zero/filled bytes, up to a maximum of 255. The EVEN directive aligns the program counter to an even address (which is equivalent to ALIGN 1) and the ODD directive aligns the program location counter to an odd address. The byte value for padding must be within the range 0 to 255.

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

EVEN, ODD, and ALIGN can only be used for code segments. Use ALIGNRAM for data segments.

Defining short labels (78K0/78K0S only)

The SADDR directive specifies that labels defined in a relocatable segment will belong to the short address area. The SADDR directive is in effect until the next segment control directive.

EXAMPLES

Beginning an absolute segment

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

```
     EXTERN  intnmi,intwdt,intp0,intp1

     ASEG
     ORG     0x0
     DC16    main      ; RESET_vect
int0 DC16    intnmi
int1 DC16    intwdt
int2 DC16    intp0
int3 DC16    intp1
     ORG     0x2100
main MOVW    AX,[SP+2] ; Start of code

     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  divrtn,mulrtn

        RSEG    table
        DC16    divrtn,mulrtn
        ORG     $+6
        DC16    subrtn

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

Beginning a stack segment (78K0/78K0S only)

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    DC32    1
         ENDMOD

         NAME    common2
         COMMON  data
up       DC8     1
         ORG     $+2
down     DC8     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.

```
         NAME     align
         RSEG     code(1) ; Start a relocatable code segment

         EVEN             ; Ensure it's on an even boundary
target   DC16     1       ; Target is on an even boundary

         ALIGN    8       ; Zero-fill to a 64-byte boundary
results  DC8      64      ; Create a 64-byte table

         RSEG     data(1) ; Start a relocatable data segment
         ALIGNRAM 3       ; Align to an 8-byte boundary
ages     DS8      64      ; Create another 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.
`CONST`	Specifies an SFR label as read-only. 78K0/78K0S only.
`DEFINE`	Defines a file-wide value.
`EQU`	Assigns a permanent value local to a module.
`LIMIT`	Checks a value against limits.
`SET`	Assigns a temporary value.
`sfr`	Creates byte-access SFR labels. 78K0/78K0S only.
`sfrp`	Creates word-access SFR labels. 78K0/78K0S only.
`SFRTYPE`	Specifies SFR attributes. 78K0/78K0S only.
`VAR`	Assigns a temporary value.

Table 12: Value assignment directives

SYNTAX

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

OPERAND MODIFIERS FOR 78K0/78K0S

These prefixes can be used to modify 78K0/78K0S operands:

Modifier	Description
`S:`	Can prefix an operand symbol to force short addressing. In the case of `sfr` and `sfrp`, it applies to the overlapping area `0xFF00` to `0xFF1F`.
`N:`	1) Can prefix an operand symbol to force 16-bit addressing. 2) Can prefix a label in a branch instruction, `BR`, to force a long branch.

Table 13: Operand modifiers—78K0/78K0S

OPERAND MODIFIERS FOR 78K0R

These prefixes can be used to modify 78K0R operands:

Prefix	Usage	Description
no prefix	source/destination	The assembler uses SFR or 16-bit (near) addressing
`S:`	source/destination	Forces the assembler to use short addressing (saddr)
`N:`	source/destination	Forces the assembler to use 16-bit (near) addressing
`F:`, `ES:`	source/destination	Forces the assembler to use `ES:16`-bit (far) addressing

Table 14: Operand modifiers—78K0R

Prefix	Usage	Description
`$:`, `S:`, `S:$`	branch	Forces the assembler to use 8-bit relative addressing
no prefix, `R:`, `R:$`	branch	The assembler uses 16-bit relative addressing
`N:`	branch	Forces the assembler to use 16-bit absolute addressing
`F:`, `ES:`	branch	Forces the assembler to use 20-bit absolute addressing
`R:`, `R:$`	call	Forces the assembler to use 16-bit relative addressing
no prefix, `N:`	call	The assembler uses 16-bit absolute addressing
`F:`, `ES:`	call	Forces the assembler to use 20-bit absolute addressing

Table 14: Operand modifiers—78K0R (Continued)

Also note the following:

- 20-bit constant addresses are treated as SFR or near addresses, depending on whether they are inside the SFR area or not
- 16-bit constant addresses are treated as near addresses
- 16- or 20-bit constant addresses prefixed with `S:` are treated as short address (saddr) area addresses.

DESCRIPTIONS

Defining a temporary value

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

Defining a permanent local value

Use `EQU`, `ALIAS`, or `=` to assign a value to a symbol.

Use `EQU`, `ALIAS`, or `=` to create a local symbol that denotes a number or offset. The symbol is only valid in the module in which it was defined, but can be made available to other modules with a `PUBLIC` directive (but not with a `PUBWEAK` directive).

Use `EXTERN` to import symbols from other modules.

Defining a permanent global value

Use `DEFINE` to define symbols that should be known to the module containing the directive and all modules following that module in the same source file. If a `DEFINE` directive is placed outside of a module, the symbol will be known to all modules following the directive in the same source file.

A symbol which 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 (78K0/78K0S only)

Use `sfr` to create special function register labels with attributes `READ`, `WRITE`, and `BYTE` turned on. Use `sfrp` to create special function register labels with the attributes `READ`, `WRITE`, and `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/warning when trying to write to the SFR.

If a special function register located in the common short address area `0xFFF00–0xFF1F` is used with an instruction that accepts both short addressing and SFR addressing, SFR addressing will be used. The short address form can be forced by using the `S:` prefix. Note that symbols or expressions in the common address area without `sfr`/`sfrp` or `SFRTYPE` attributes are, by default, treated as short addresses.

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.

EXAMPLES

Defining a permanent global value

```
globvalue  DEFINE  12
```

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
; Generate table of powers of 3

cons     SET      1
loop     REPT     4
cons     SET      cons * 3
         DC16     cons
         ENDR

         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 to declare `locn` for use anywhere in the file:

```
        NAME    add1
locn    DEFINE  020h
value   EQU     77
        MOV     R0,locn
        ADD     R0,#value
        RET
        ENDMOD

        NAME    add2
value   EQU     88
        MOV     R0,locn
        ADD     R0,#value
        RET
        END
```

Using bit equates (78K0/78K0S only)

It is possible to define bit equates in the 78K0/78K0S Assembler according to the following example. Note that only the bit number that makes up a bit address can be an external. However, it is not possible to use bit equates that are externals.

```
      1    000000                  NAME  bits
      2    000000          sfr     P0=0xFF00
      3    000000          sfr     P1=0xFF01
      4    000000
      5    000707          strobe  EQU     P0.7
      6    000700          ready   EQU     P0.0
```

```
 7    000000
 8    000000 F400     main    MOV     A,P0
 9    000002 710400   nodata  MOV1    CY,ready
10    000005 9DFB             BNC     nodata
11    000007 AF               RET
12    000008
13    000008                  END
```

Using special function registers (78K0/78K0S only)

The following example defines three SFRs:

```
sfrp    IF0    = 0xFFE0  ; Interrupt flag register 0
sfr     IF0L   = 0xFFEO  ; Low byte of interrupt flag
                           register 0
sfr     IF0H   = 0xFFE1  ; High byte of interrupt flag
                           register 0
```

The following shows how attributes can be combined in any manner to express how the SFR can be used:

```
SFRTYPE IF0     word,read,write = 0xFFE0
SFRTYPE IF0L    byte,read,write = 0xFFEO
SFRTYPE IF0H    byte,read,write = 0xFFE1
```

However, the following instruction is illegal since BYTE access is not used, and the assembler will give a warning:

```
MOV     A,IF0
```

Using the LIMIT directive

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

```
          speed
VAR       23
LIMIT     speed,10,30,"...speed out of range..."
```

Conditional assembly directives

These directives provide logical control over the selective assembly of source code. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
ELSE	Assembles instructions if a condition is false.	
ELSEIF	Specifies a new condition in an IF…ENDIF block.	No forward references No external references Absolute Fixed
ENDIF	Ends an IF block.	
IF	Assembles instructions if a condition is true.	No forward references No external references Absolute Fixed

Table 15: Conditional assembly directives

SYNTAX

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

PARAMETERS

condition One of the following:

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

DESCRIPTIONS

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 assembly directives may be used anywhere in an assembly, but have their greatest use in conjunction with macro processing.

All assembler directives (except for END) as well as the inclusion of files 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 adds a constant to a register:

```
addm MACRO   a,b
     IF      'b'='1'
     INC     a
     ELSE
     ADD     a,#b
     ENDIF
     ENDM
```

If the argument to the macro is 1 it generates an INC instruction to save instruction cycles; otherwise it generates an ADD instruction.

It could be tested with the following program:

```
main MOV     R1,#17
     addm    R1,2
     MOV     R1,#22
     addm    R1,1
     RET

     END
```

Macro processing directives

These directives allow user macros to be defined. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
`_args`	Is set to number of arguments passed to macro.	
`ENDM`	Ends a macro definition.	
`ENDMAC`	Ends a macro definition.	
`ENDR`	Ends a repeat structure.	
`EXITM`	Exits prematurely from a macro.	
`LOCAL`	Creates symbols local to a macro.	
`MACRO`	Defines a macro.	
`REPT`	Assembles instructions a specified number of times.	No forward references No external references Absolute Fixed
`REPTC`	Repeats and substitutes characters.	
`REPTI`	Repeats and substitutes text.	

Table 16: Macro processing directives

SYNTAX

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

PARAMETERS

actual	A string to be substituted.
argument	A symbolic argument name.
expr	An expression.

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

DESCRIPTIONS

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

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

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

Defining a macro

You define a macro with the statement:

name MACRO [*argument*] [,*argument*] …

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

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

```
errmac  MACRO   text
        CALL    abort
        DC8     text,0
        ENDM
```

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

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

The assembler will expand this to:

```
        CALL    abort
        DC8     'Disk not ready',0
```

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

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

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

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

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

Passing special characters

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

For example:

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

The macro can be called using the macro quote characters:

```
         macld  <A, #1>
         END
```

You can redefine the macro quote characters with the -M command line option; see *-M*, page 83.

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  A78K_MAN

        EXTERN  sub1

DO_SUB1 MACRO   p1 ,p2
        IF _args == 2
            CMP   p1 ,p2
            BZ    nocall
            CALL  sub1
nocall:
          ELSE
            CALL  sub1
          ENDIF
        ENDM

        RSEG  CODE

        DO_SUB1
```

```
        DO_SUB1  A, #2

        END
```

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

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

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

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

It is illegal to *redefine* a macro.

Passing special characters

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

For example:

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

It could be called using:

```
        macld   <A, #1>
        END
```

You can redefine the macro quote characters with the -M command line option.

How macros are processed

There are three distinct phases in the macro process:

1. The assembler performs scanning and saving of macro definitions. The text between MACRO and ENDM is saved but not syntax checked.
2. A macro call forces the assembler to invoke the macro processor (expander). The macro expander switches (if not already in a macro) the assembler input stream from a source file to the output from the macro expander. The macro expander takes its input from the requested macro definition.

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

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

Repeating statements

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

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

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

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

EXAMPLES

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

Coding inline for efficiency

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

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

```
        NAME    play
P0      DEFINE  0xFF00

        RSEG    DATA
buffer  DS8     25
watch   DC8     0xFF

        RSEG    CODE
play    MOVW    AX,#buffer
        MOVW    HL,AX
        MOV     B,#0
        MOV     A,[HL+B]
loop    INC     B
        MOV     P0, A
        MOV     A,[HL+B]
        CMP     A,watch
        BNZ     loop
```

```
        RET

        END
```

The main program calls this routine as follows:

```
doplay  CALL    play
```

For efficiency we can recode this as the following macro:

```
        NAME    play
        ORG     0
        DC16    main

play    MACRO
        LOCAL   loop
        MOVW    AX,#buffer
        MOVW    HL,AX
        MOV     B,#0
        MOV     A,[HL+B]
loop    INC     B
        MOV     P0, A
        MOV     A,[HL+B]
        CMP     A,watch
        BNZ     loop
        ENDM

P0      DEFINE  0xFF00

        RSEG    DATA
buffer  DS8      25
watch   DC8      0xFF

        RSEG CODE
main    play
        play
        RET

        END
```

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

Using REPTC and REPTI

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

```
         NAME   reptc

         EXTERN plotc
P0       DEFINE  0xFF00
banner   REPTC  chr,"Welcome"
         MOV    P0,#'chr'
         CALL   plotc
         ENDR

         END
```

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

```
         NAME   repti

         EXTERN base,count,init

         MOV    A, #0
banner   REPTI  adds,base,count,init
         MOV    adds, A
         ENDR

         END
```

Listing control directives

These directives provide control over the assembler list file.

Directive	Description
`COL`	Sets the number of columns per page (78K0/78K0S). Retained in 78K0R for backward compatibility reasons.
`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 (78K0/78K0S). Retained in 78K0R for backward compatibility reasons.

Table 17: Listing control directives

Directive	Description
`LSTREP`	Controls the listing of lines generated by repeat directives.
`LSTXRF`	Generates a cross-reference table.
`PAGE`	Generates a new page (78K0/78K0S). Retained in 78K0R for backward compatibility reasons.
`PAGSIZ`	Sets the number of lines per page (78K0/78K0S). Retained in 78K0R for backward compatibility reasons.

Table 17: Listing control directives (Continued)

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, by default 80 is used
lines	An absolute expression in the range 10 to 150, by default 44 is used

DESCRIPTIONS

Turning the listing on or off

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

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

Listing conditional code and strings

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

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

Use LSTCOD+ to list more than one line of code for a source line, if needed; that is, long ASCII strings will produce several lines of output.

The default setting is LSTCOD-, which restricts the listing of output code to just the first line of code for a source line.

Using the LSTCND and LSTCOD directives does *not* affect code generation.

Controlling the listing of macros

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

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

Controlling the listing of generated lines

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

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

Generating a cross-reference table

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

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

Specifying the list file format (78K0/78K0S only)

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

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

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

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

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

EXAMPLES

Turning the listing on or off

To disable the listing of a debugged section of program:

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

Listing conditional code and strings

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

```
        NAME    lstcndtst
        EXTERN  print

        RSEG    prom

debug   VAR     0
begin   IF      debug
        CALL    print
        ENDIF

        LSTCND+
begin2  IF      debug
        CALL    print
        ENDIF

        END
```

Controlling the listing of macros

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

```
dec2    MACRO  arg
        DEC    arg
        DEC    arg
        ENDM

        LSTMAC-

inc2    MACRO  arg
        INC    arg
        INC    arg
        ENDM
```

```
begin   dec2   R6

        LSTEXP-
        inc2   R7
        RET

        END    begin
```

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 80
        LSTPAG+
        ...
        ENDMOD
        MODULE
        ...
        PAGE
        ...
```

C-style preprocessor directives

The following C-language preprocessor directives are available:

Directive	Description
#define	Assigns a value to a preprocessor symbol.
#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 preprocessor symbol is defined.
#ifndef	Assembles instructions if a preprocessor symbol is undefined.
#include	Includes a file.
#line	Changes the line numbers of the source code lines immediately following the #line directive, or the filename of the file being compiled.

Table 18: C-style preprocessor directives

Directive	Description
`#message`	Generates a message on standard output. 78K0/78K0S only.
`#pragma`	Controls extension features. Pragma directives can only be used in code written for the 78K0R Assembler and they are described in the chapter *78K0R pragma directives*. In the 78K0/78K0S Assembler, they are recognized but ignored.
`#undef`	Undefines a preprocessor symbol.

Table 18: C-style preprocessor directives (Continued)

SYNTAX

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

PARAMETERS

condition	An absolute expression	The expression must not contain any assembler labels or symbols, and any non-zero value is considered as true.
filename	Name of file to be included.	
message	Text to be displayed.	
symbol	Preprocessor symbol to be defined, undefined, or tested.	
text	Value to be assigned.	

DESCRIPTIONS

The preprocessor directives are processed before other directives. As an example avoid constructs like:

```
redef macro     ; avoid the following
#define \1 \2
```

```
        endm
```

since the \1 and \2 macro arguments will not be available during the preprocess.

Also be careful with comments; the preprocessor understands /* */ and //. The following expression will evaluate to 3 since the comment character will be preserved by #define:

```
#define x 3    ; comment
exp EQU x*8+5
```

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

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

```
#define   five 5 ; comment

     MOV   five,  #3       ; syntax error
     ; Expands to "MOV 0x05 ; comment, #3"

     MOV    A,  #five + addr ; incorrect code
     ; Expands to "MOV A, 0x05 ; comment + addr"
```

Defining and undefining preprocessor symbols

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

#define *symbol value*

is similar to:

symbol SET *value*

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

Conditional preprocessor directives

Use the #if...#else...#endif directives to control the assembly process at assembly time. If the condition following the #if directive is not true, the subsequent instructions will not generate any code (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. The filename can be specified within double quotes or within angle brackets.

Following is the full description of the assembler's `#include` file search procedure:

- If the name of the `#include` file is an absolute path, that file is opened.
- When the assembler encounters the name of an `#include` file in angle brackets such as:
  `#include <io`*`derivative`*`.h>`

  it searches the following directories for the file to include:

  1 The directories specified with the `-I` option, in the order that they were specified.

  2 The directories specified using the `A78K_INC` environment variable, if any.

- When the assembler encounters the name of an `#include` file in double quotes such as:
  `#include "vars.h"`

  it searches the directory of the source file in which the `#include` statement occurs, and then performs the same sequence as for angle-bracketed filenames.

  If there are nested `#include` files, the assembler starts searching the directory of the file that was last included, iterating upwards for each included file, searching the source file directory last.

Use angle brackets for header files provided with the IAR Assemblers for 78K, and double quotes for header files that are part of your application.

Displaying errors

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

EXAMPLES

Using conditional preprocessor directives

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

```
#define tweak 1
#define adjust 3

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

Including a source file

The following example uses `#include` to include a file defining macros into the source file. For example, the following macros could be defined in `macros.s26`:

```
xchrp   MACRO   a,b
        PUSH    a
        PUSH    b
        POP     a
        POP     b
        ENDM
```

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

```
        NAME   include

; Standard macro definitions
#include c:\iar\asm\inc\macros.s26"

; Program
main:   xchrp   RP2,RP3
        RET
        END     main
```

Data definition or allocation directives

These directives define values or reserve memory. The column *Alias* in the following table shows the Renesas directive that corresponds to the IAR Systems directive. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

Directive	Alias	Description
DC8	DB	Generates 8-bit constants, including strings.
DC16	DW	Generates 16-bit constants.
DC24	DP	Generates 24-bit constants.
DC32	DD	Generates 32-bit constants.
DC64		Generates 64-bit constants. 78K0R only.
DF32		Generates 32-bit floating-point constants. 78K0R only.
DF64		Generates 64-bit floating-point constants. 78K0R only.
DS8	DS	Allocates space for 8-bit integers.
DS16		Allocates space for 16-bit integers. 78K0R only.
DS24		Allocates space for 24-bit integers. 78K0R only.
DS32		Allocates space for 32-bit integers. 78K0R only.
DS64		Allocates space for 64-bit integers. 78K0R only.

Table 19: Data definition or allocation directives

SYNTAX

```
DB expr [,expr] …
DC8 expr [,expr] ...
DC16 expr [,expr] ...
DC24 expr [,expr] ...
DC32 expr [,expr] ...
DC64 expr [,expr] …
DD expr [,expr] …
DF32 value [,value] …
DF64 value [,value] …
DP expr [,expr] …
DS count
DS8 count
DS16 count
DS24 count
DS32 count
DS64 count
DW expr [,expr] …
```

PARAMETERS

count A valid absolute expression specifying the number of elements to be reserved.

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

value A valid absolute expression or floating-point constant.

DESCRIPTIONS

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

Size	Reserve and initialize memory	Reserve uninitialized memory
8-bit integers	DC8, DB	DS8, DS
16-bit integers	DC16, DW	DS16
24-bit integers	DC24, DP	DS24
32-bit integers	DC32, DD	DS32
64-bit integers	DC64	DS64
32-bit floats	DF32	DS32
64-bit floats	DF64	DS64

Table 20: Using data definition or allocation directives

EXAMPLES

Generating a lookup table

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

```
        NAME    table
        RSEG    CONST
table   DC16    addsubr, subsubr, clrsubr
        RSEG    CODE
addsubr ADD     A,C
        RET
subsubr SUB     A,C
        RET
clrsubr MOV     A,#0
        RET

        END
```

Defining strings

To define a string:

```
myMsg   DC8 'Please enter your name'
```

To define a string which includes a trailing zero:

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

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

```
errMsg  DC8 'Don''t understand!'
```

Reserving space

To reserve space for `0xA` bytes:

```
table   DS8   0xA
```

Assembler control directives

These directives provide control over the operation of the assembler. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
`$`	Includes a file. 78K0/78K0S only.	
`/*comment*/`	C-style comment delimiter.	
`//`	C++ style comment delimiter.	
`CASEOFF`	Disables case sensitivity.	
`CASEON`	Enables case sensitivity.	
`RADIX`	Sets the default base on all numeric values.	No forward references No external references Absolute Fixed

Table 21: Assembler control directives

SYNTAX

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

PARAMETERS

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

DESCRIPTIONS

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

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

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

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

Controlling case sensitivity

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

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

Defining comments

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

```
/*
Program to read serial input.
Version 1: 19.2.02
Author: mjp
*/
```

Changing the base

To set the default base to 16:

```
        RADIX  D'16
        MOV    R1,#12
```

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

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

```
RADIX   0x0A
```

Controlling case sensitivity

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

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

The following will generate a duplicate label error:

```
        CASEOFF

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

        END
```

Function directives

The function directives are generated by the IAR C/C++ Compiler for 78K to pass information about functions and function calls to the IAR XLINK Linker. These directives can be seen if you create an assembler list file by using the compiler option **Output assembler file>Include compiler runtime information** (`-lA`).

Note: These directives are primarily intended to support static overlay, a feature which is useful in smaller microcontrollers. The IAR C/C++ Compiler for 78K does not use static overlay, as it has no use for it.

SYNTAX

```
ARGFRAME &lt;segment&gt;, <size>, <type>
FUNCALL <caller>, <callee>
FUNCTION <label>,<value>
LOCFRAME &lt;segment&gt;, <size>, <type>
```

PARAMETERS

label	A label to be declared as function.
value	Function information.
segment	The segment in which argument frame or local frame is to be stored.
size	The size of the argument frame or the local frame.

type	The type of argument or local frame; either `STACK` or `STATIC`.
caller	The caller to a function.
callee	The called function.

DESCRIPTIONS

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

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

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

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

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

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

Call frame information directives

These directives allow backtrace information to be defined in the assembler source code. The benefit is that you can view the call frame stack when you debug your assembler code.

Directive	Description
`CFI BASEADDRESS`	Declares a base address CFA (Canonical Frame Address).
`CFI BLOCK`	Starts a data block.
`CFI CODEALIGN`	Declares code alignment.
`CFI COMMON`	Starts or extends a common block.
`CFI CONDITIONAL`	Declares data block to be a conditional thread.

Table 22: Call frame information directives

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

Table 22: Call frame information directives (Continued)

SYNTAX

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

Names block directives

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

Extended names block directives

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

Common block directives

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

Extended common block directives

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

Data block directives

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

PARAMETERS

bits	The size of the resource in bits.
cell	The name of a frame cell.
cfa	The name of a CFA (canonical frame address).

cfiexpr	A CFI expression (see *CFI expressions*, page 68).
codealignfactor	The smallest factor of all instruction sizes. Each CFI directive for a data block must be placed according to this alignment. 1 is the default and can always be used, but a larger value will shrink the produced backtrace information in size. The possible range is 1–256.
commonblock	The name of a previously defined common block.
constant	A constant value or an assembler expression that can be evaluated to a constant value.
dataalignfactor	The smallest factor of all frame sizes. If the stack grows towards higher addresses, the factor is negative; if it grows towards lower addresses, the factor is positive. 1 is the default, but a larger value will shrink the produced backtrace information in size. The possible ranges are -256 – -1 and 1 – 256.
label	A function label.
name	The name of the block.
namesblock	The name of a previously defined names block.
offset	The offset relative the CFA. An integer with an optional sign.
part	A part of a composite resource. The name of a previously declared resource.
resource	The name of a resource.
segment	The name of a segment.
size	The size of the frame cell in bytes.
type	The memory type, such as `CODE`, `CONST` or `DATA`. In addition, any of the memory types supported by the IAR XLINK Linker. It is used solely for the purpose of denoting an address space.

DESCRIPTIONS

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

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

Backtrace rows and columns

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

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

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

Defining a names block

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

Start and end a names block with the directives:

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

where `name` is the name of the block.

Only one names block can be open at a time.

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

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

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

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

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

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

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

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

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

  ```
  CFI STACKFRAME cfa resource type
  ```

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

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

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

  ```
  CFI STATICOVERLAYFRAME cfa segment
  ```

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

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

  ```
  CFI BASEADDRESS cfa type
  ```

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

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

Extending a names block

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

Extend an existing names block with the directive:

```
CFI NAMES name EXTENDS namesblock
```

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

```
CFI ENDNAMES name
```

Defining a common block

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

Start a common block with the directive:

```
CFI COMMON name USING namesblock
```

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

Declare the return address column with the directive:

```
CFI RETURNADDRESS resource type
```

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

End a common block with the directive:

```
CFI ENDCOMMON name
```

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

Inside a common block you can declare the initial value of a CFA or a resource by using the directives listed last in *Common block directives*, page 61. For more information on these directives, see *Simple rules*, page 66, and *CFI expressions*, page 68.

Extending a common block

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

Extend an existing common block with the directive:

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

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

```
CFI ENDCOMMON name
```

Defining a data block

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

Start a data block with the directive:

```
CFI BLOCK name USING commonblock
```

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

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

```
CFI FUNCTION label
```

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

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

```
CFI NOFUNCTION
```

End a data block with the directive:

```
CFI ENDBLOCK name
```

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

Inside a data block you may manipulate the values of the columns by using the directives listed last in *Data block directives*, page 61. For more information on these directives, see *Simple rules*, page 66, and *CFI expressions*, page 68.

SIMPLE RULES

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

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

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

In those rare cases where the descriptive power of the simple rules are not enough, a full CFI expression can be used to describe the information (see *CFI expressions*, page 68). However, whenever possible, you should always use a simple rule instead of a CFI expression.

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

Simple rules for resources

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

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

```
CFI REG SAMEVALUE
```

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

```
CFI REG UNDEFINED
```

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

```
CFI REG1 REG2
```

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

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

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

```
CFI RET CONCAT
```

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

Simple rules for CFAs

In contrast with the rules for resources, the rules for CFAs describe the address of the beginning of the call frame. The call frame often includes the return address pushed by the subroutine calling instruction. The CFA rules describe how to compute the address to the beginning of the current call frame. There are two different forms of CFAs, stack frames and static overlay frames, each declared in the associated names block. See *Names block directives*, page 60.

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

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

```
CFI CFA_SP NOTUSED
```

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

```
CFI CFA_SP SP + 4
```

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

CFI EXPRESSIONS

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

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

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

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

Unary operators

Overall syntax: `OPERATOR(operand)`

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

Table 23: Unary operators in CFI expressions

Binary operators

Overall syntax: `OPERATOR(operand1,operand2)`

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

Table 24: Binary operators in CFI expressions

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

Table 24: Binary operators in CFI expressions (Continued)

Ternary operators

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

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

Table 25: Ternary operators in CFI expressions

EXAMPLE

The following is a generic example and not an example specific to the 78K microcontroller. This will simplify the example and clarify the usage of the CFI directives. A target-specific example can be obtained by generating assembler output when compiling a C source file.

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

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

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

Table 26: Code sample with backtrace rows and columns

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

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

Defining the names block

The names block for the small example above would be:

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

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

Defining the common block

The common block for the simple example above would be:

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

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

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

Defining the data block

Continuing the simple example, the data block would be:

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

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

Diagnostics

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

Message format

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

`filename,linenumber level[tag]: message`

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

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

Severity levels

The diagnostics are divided into different levels of severity:

Remark (78K0R only)

A diagnostic message that is produced when the assembler finds a source code construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued but can be enabled, see *--remarks*, page 120.

Warning

A diagnostic message that is produced when the assembler finds a programming error or omission which is of concern but not so severe as to prevent the completion of compilation. In the 78K0/78K0S Assembler, warnings can be disabled by use of the command-line option `-w`, see *-w*, page 87. In the 78K0R Assembler, warnings can be disabled by use of the command line option `--no_warnings`, see *--no_warnings*, page 118.

Error

A diagnostic message that is produced when the assembler has found a construct which clearly violates the language rules, such that code cannot be produced. An error will produce a non-zero exit code.

Fatal error

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

SETTING THE SEVERITY LEVEL

The diagnostic messages can be suppressed or the severity level can be changed for all types of diagnostics except for fatal errors and some of the regular errors.

See *Summary of 78K0/78K0S Assembler options*, page 78, or *Summary of 78K0R Assembler options*, page 108, for a description of the assembler options that are available for setting severity levels.

See the chapter *78K0R pragma directives*, for a description of the pragma directives that are available in the 78K0R Assembler for setting severity levels.

INTERNAL ERROR

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

```
Internal error: message
```

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

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

Part 2. 78K0/78K0S Assembler reference

This part of the IAR Assembler Reference Guide for 78K describes the assembler for the 78K0 and 78K0S cores and includes the following chapters:

- 78K0/78K0S Assembler options
- 78K0/78K0S Assembler operators.

78K0/78K0S Assembler options

This chapter first explains how to set the options for the 78K0/78K0S Assembler from the command line, and gives an alphabetical summary of the assembler options. It then provides detailed reference information for each assembler option.

The *IAR Embedded Workbench® IDE User Guide* describes how to set assembler options in the IAR Embedded Workbench.

For information about setting options for the 78K0R Assembler, see the chapter *78K0R Assembler options*.

Setting command line options

To set assembler options from the command line, you include them on the command line, after the `a78k` command:

```
a78k [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.s26`, use the following command to generate a list file to the default filename (`power2.lst`):

```
a78k power2.s26 -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`:

```
a78k power2.s26 -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`:

```
a78k power2.s26 -Llist\
```

Note: The subdirectory you specify must already exist. The trailing backslash is required because the parameter is added to the beginning of 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:

```
a78k -f extend.xcl
```

ERROR RETURN CODES

When using the IAR Assembler for 78K0/78K0S from within a batch file, you may need to determine whether the assembly was successful in order to decide what step to take next. For this reason, the assembler returns the following error return codes:

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

Table 27: Assembler error return codes

ASSEMBLER ENVIRONMENT VARIABLES

Options can also be specified using the `ASM78K` environment variables, see *Environment variables*, page 107.

Summary of 78K0/78K0S Assembler options

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

Command line option	Description
`-B`	Macro execution information
`-c`	Conditional list
`-D`	Defines a symbol
`-d`	Disables matching
`-E`	Maximum number of errors

Table 28: 78K0/78K0S Assembler options summary

Command line option	Description
`-f`	Extends the command line
`-G`	Opens standard input as source
`-I`	Specifies an include path
`-i`	#included text
`-L`	Lists to prefixed source name
`-l`	Lists to named file
`-M`	Macro quote characters
`-N`	Omits header from assembler listing
`-n`	Enables support for multibyte characters
`-O`	Sets object filename prefix
`-o`	Sets object filename
`-p`	Lines/page
`-r`	Generates debug information
`-S`	Sets silent operation
`-s`	Case-sensitive user symbols
`-t`	Tab spacing
`-U`	Undefines a symbol
`-v`	Processor core
`-w`	Disables warnings
`-X`	Includes unreferenced external symbols
`-x`	Includes cross-references

Table 28: 78K0/78K0S Assembler options summary (Continued)

Descriptions of assembler options

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

-B `-B`

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

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

This option is mainly used in conjunction with the list file options `-L` or `-l`; for additional information, see page 82.

Project>Options>Assembler >List>Macro execution info

-c `-c{DMEAO}`

Use this option to control the contents of the assembler list file. This option is mainly used in conjunction with the list file options `-L` and `-l`; see page 82 for additional information.

The following table shows the available parameters:

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

Table 29: Conditional list (-c)

To set related options, select:

Project>Options>Assembler >List

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

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

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

Example

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

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

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

Production version: `a78k prog`

Test version: `a78k 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:

```
a78k prog -DFRAMERATE=3
```

Project>Options>Assembler >Preprocessor>Defined symbols

`-d` `-d`

This option disables `#ifdef`, `#endif` matching.

This option is not available in the IAR Embedded Workbench IDE.

`-E` `-E`*`number`*

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

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

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

`-f` `-f` *`filename`*

Extends the command line with text read from the specified file. 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:

```
a78k prog -f extend.xcl
```

To set related options, select:

Project>Options>Assembler >Extra Options

`-G` `-G`

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

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

This option is not available in the IAR Embedded Workbench IDE.

`-I` `-I`*`prefix`*

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

By default, the assembler searches for `#include` files only in the current working directory and in the paths specified in the `A78K_INC` environment variable. The `-I` option allows you to give the assembler the names of directories where it will also search if it fails to find the file in the current working directory.

Example

Using the options:

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

and then writing:

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

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

Project>Options>Assembler >Preprocessor>Additional include directories

`-i` `-i`

Includes `#include` files in the list file.

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

To set related options, select:

Project>Options>Assembler >List>#included text

`-L` `-L[prefix]`

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

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

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

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

Example

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

```
a78k prog -Llist\
```

To set related options, use:

Project>Options>General Options >Output>Output directories

`-l` `-l` *`filename`*

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

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

To set related options, select:

Project>Options>Assembler >List

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

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

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

Example

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

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

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

```
a78k filename -M'<>'
```

Project>Options>Assembler >Language>Macro quote characters

-N

`-N`

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

This option is useful in conjunction with the list file options `-L` or `-l`; see page 82 for additional information.

To set related options, select:

Project>Options>Assembler >List

-n

`-n`

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

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

Project>Options>Assembler >Language>Enable multibyte support

-O

`-O`*`prefix`*

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

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

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

Example

To send the object code to the file `obj\prog.r26` rather than to the default file `prog.r26`:

```
a78k prog -Oobj\
```

To set related options, use:

Project>Options>General Options >Output>Output directories

-o `-o filename`

This option sets the filename to be used for the object file. Notice that you must include a space before the filename. If no extension is specified, `r26` is used.

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

Example

For example, the following command puts the object code to the file `obj.r26` instead of the default `prog.r26`:

```
a78k prog -o obj
```

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

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

-p `-plines`

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

This option is used in conjunction with the list options `-L` or `-l`; see page 82 for additional information.

Project>Options>Assembler >List>Lines/page

-r `-r`

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

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

Project>Options>Assembler >Output>Generate debug information

-S `-S`

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

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

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

This option is not available in the IAR Embedded Workbench IDE.

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

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

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

Table 30: Controlling case sensitivity in user symbols (-s)

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

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

---

`-t` `-t`*`n`*

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

This option is useful in conjunction with the list options `-L` or `-l`; see page 82 for additional information.

Project>Options>Assembler >List>Tab spacing

`-U` `-U`*`symbol`*

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

By default, the assembler provides certain predefined symbols; see *Predefined symbols*, page 10. The `-U` option allows you to undefine such a predefined symbol to make its name available for your own use through a subsequent `-D` option or source definition.

Example

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

```
a78k prog -U __TIME__
```

This option is not available in the IAR Embedded Workbench IDE.

-v `-v[0|1|2]`

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

The following table shows how the `-v` options are mapped to the 78K derivatives:

Option	Description
`-v0`	78K0 (without DIV/MUL instructions)
`-v1`	78K0 (with DIV/MUL instructions)
`-v2`	78K0S

Table 31: Specifying the processor configuration (-v)

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

To set related options, use:

Project>Options>General Options >Target>Device

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

By default, the assembler displays a warning message when it detects an element of the source which is legal in a syntactical sense, but may contain a programming error; see *Diagnostics*, page 73, for details.

Use this option to disable warnings.

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

Table 32: Disabling assembler warnings (-w)

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

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

Example

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

```
a78k prog -w-0
```

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

```
a78k prog -w-0-8
```

To set related options, select:

Project>Options>Assembler >Diagnostics

-X -X

This option includes unreferenced external symbols in the output.

This option is not available in the IAR Embedded Workbench IDE.

-x -x{DI2}

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

This option is useful in conjunction with the list options `-L` or `-l`; see page 82 for additional information.

The following parameters are available:

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

Table 33: Including cross-references in assembler list file (-x)

Project>Options>Assembler >List>Include cross reference

78K0/78K0S Assembler operators

This chapter first describes the precedence of the 78K0/78K0S Assembler operators, and then summarizes the operators, classified according to their precedence. Finally, this chapter provides reference information about each operator, presented in alphabetical order.

For information about the operators for the 78K0R Assembler, see the chapter *78K0R Assembler operators*.

Precedence of assembler operators

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

The following rules determine how expressions are evaluated:

- The highest precedence operators are evaluated first, then the second highest precedence operators, and so on until the lowest precedence operators are evaluated.
- Operators of equal precedence are evaluated from left to right in the expression.
- Parentheses (and) can be used for grouping operators and operands and for controlling the order in which the expressions are evaluated. For example, the following expression evaluates to 1:
  `7/(1+(2*3))`

Summary of assembler operators

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

UNARY OPERATORS – 1

`()`	Parenthesis.
`+`	Unary plus.

`–`	Unary minus.
`NOT, !`	Logical NOT.
`BINNOT, ~`	Bitwise NOT.
`LOW`	Low byte.
`HIGH`	High byte.
`BYTE2`	Second byte.
`BYTE3`	Third byte.
`LWRD`	Low word.
`HWRD`	High word.
`DATE`	Current time/date.
`SFB`	Segment begin.
`SFE`	Segment end.
`SIZEOF`	Segment size.

MULTIPLICATIVE AND SHIFT ARITHMETIC OPERATORS – 3

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

ADDITIVE ARITHMETIC OPERATORS – 4

`+`	Addition.
`–`	Subtraction.

AND OPERATORS – 5

`AND, &&`	Logical AND.
`BINAND, &`	Bitwise AND.

OR OPERATORS – 6

`OR, ||`	Logical OR.
`BINOR, |`	Bitwise OR.
`XOR`	Logical exclusive OR.
`BINXOR, ^`	Bitwise exclusive OR.

COMPARISON OPERATORS – 7

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

Descriptions of assembler operators

The following sections give detailed descriptions of each assembler operator. See *Expressions, operands, and operators*, page 6, for related information.

`()` Parenthesis (1).

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

Example

```
1+2*3 → 7
(1+2)*3 → 9
```

`*` Multiplication (3).

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

Example

```
2*2 → 4
-2*2 → -4
```

\+ Unary plus (1).

Unary plus operator.

Example

```
+3 → 3
3*+2 → 6
```

\+ Addition (4).

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

Example

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

– Unary minus (1).

The unary minus operator performs arithmetic negation on its operand.

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

– Subtraction (4).

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

Example

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

`/` Division (3).

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

Example

```
9/2 → 4
-12/3 → -4
9/2*6 → 24
```

`<, LT` Less than (7).

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

Example

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

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

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

Example

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

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

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

Example

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

`=, ==, EQ` Equal (7).

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

Example

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

`>, GT` Greater than (7).

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

Example

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

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

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

Example

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

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

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

Example

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

`&`, `BINAND` Bitwise AND (5).

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

Example

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

`~`, `BINNOT` Bitwise NOT (1).

Use ~ to perform bitwise NOT on its operand.

Example

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

`|`, `BINOR` Bitwise OR (6).

Use | to perform bitwise OR on its operands.

Example

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

`^`, `BINXOR` Bitwise exclusive OR (6).

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

Example

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

`%`, `MOD` Modulo (3).

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

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

Example

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

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

Use `!` to negate a logical argument.

Example

```
! B'0101 → 0
! B'0000 → 1
```

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

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

Example

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

`BYTE2` Second byte (1).

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

Example

```
BYTE2 0x12345678 → 0x56
```

`BYTE3` Third byte (1).

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

Example

```
BYTE3 0x12345678 → 0x34
```

`DATE` Current time/date (1).

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

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

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

Example

To assemble the date of assembly:

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

`HIGH` High byte (1).

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

Example

```
HIGH 0xABCD → 0xAB
```

`HWRD` High word (1).

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

Example

```
HWRD 0x12345678 → 0x1234
```

`LOW` Low byte (1).

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

Example

`LOW 0xABCD → 0xCD`

`LWRD` Low word (1).

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

Example

`LWRD 0x12345678 → 0x5678`

`SFB` Segment begin (1).

Syntax

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

Parameters

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

Description

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

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

Example

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

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

SFE Segment end (1).

Syntax

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

Parameters

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

Description

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

Example

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

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

The size of the segment MY_SEGMENT can be calculated as:

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

<<, SHL Logical shift left (3).

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

Example

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

>>, SHR Logical shift right (3).

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

Example

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

SIZEOF Segment size (1).

Syntax

SIZEOF *segment*

Parameters

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

Description

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

Example

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

sets size to the size of segment CODE.

UGT Unsigned greater than (7).

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

Example

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

ULT Unsigned less than (7).

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

Example

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

XOR Logical exclusive OR (6).

Use XOR to perform logical XOR on its two operands.

Example

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

Part 3. 78K0R Assembler reference

This part of the IAR Assembler Reference Guide for 78K describes the assembler for the 78K0R core and includes the following chapters:

- 78K0R Assembler options
- 78K0R Assembler operators
- 78K0R pragma directives.

78K0R Assembler options

This chapter first explains how to set the options for the 78K0R Assembler from the command line, and gives an alphabetical summary of the assembler options. It then provides detailed reference information for each assembler option.

The *IAR Embedded Workbench® IDE User Guide* describes how to set assembler options in the IAR Embedded Workbench.

For information about setting options for the 78K0/78K0S Assembler, see the chapter *78K0/78K0S Assembler options*.

Setting command line options

To set assembler options from the command line, include them on the command line after the `a78k0r` command, either before or after the source filename. For example, when assembling the source `prog.s26`, use the following command to generate an object file with debug information:

```
a78k0r prog.s26 --debug
```

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

```
a78k0r prog.s26 -l prog.lst
```

Some other options accept a string that is not a filename. The string is included after the option letter, but without a space. For example, to define a symbol:

```
a78k0r prog.s26 -DDEBUG=1
```

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

Notice that a 78K0R command line option has a *short* name and/or a *long* name:

- A short option name consists of one character, with or without parameters. You specify it with a single dash, for example `-r`.
- A long name consists of one or several words joined by underscores, and it may have parameters. You specify it with double dashes, for example `--debug`.

SPECIFYING PARAMETERS

When a parameter is needed for an option with a short name, it can be specified either immediately following the option or as the next command line argument.

For instance, an include file path of `\usr\include` can be specified either as:

```
-I\usr\include
```

or as

```
-I \usr\include
```

Note: `/` can be used instead of `\` as directory delimiter. A trailing backslash can be added to the last directory name, but is not required.

Additionally, output file options can take a parameter that is a directory name. The output file will then receive a default name and extension.

When a parameter is needed for an option with a long name, it can be specified either immediately after the equal sign (=) or as the next command line argument, for example:

```
--diag_suppress=Pe0001
```

or

```
--diag_suppress Pe0001
```

Options that accept multiple values may be repeated, and may also have comma-separated values (without space), for example:

```
--diag_warning=Be0001,Be0002
```

The current directory is specified with a period (`.`), for example:

```
a78k0r prog.s26 -l .
```

A file specified by `-` (a single dash) is standard input or output, whichever is appropriate.

Note: When an option takes a parameter, the parameter cannot start with a dash (`-`) followed by another character. Instead you can prefix the parameter with two dashes (`--`). The following example will generate a list on standard output:

```
a78k0r prog.s26 -l ---
```

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:

```
a78k0r -f extend.xcl
```

ENVIRONMENT VARIABLES

Both 78K0/78K0S and 78K0R Assembler options can also be specified in the `ASM78K` environment variable. The assembler automatically 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 IAR Assembler 78K0R:

Environment variable	Description
`A78K_INC`	Specifies directories to search for include files; for example: `A78K_INC=c:\program files\iar systems\embe dded workbench 3.n\78k\inc;c:\headers`
`ASM78K`	Specifies command line options; for example: `ASM78K=-l asm.lst`

Table 34: Environment variables

ERROR RETURN CODES

The IAR Assembler for 78K0R returns status information to the operating system which can be tested in a batch file.

The following command line error codes are supported:

Code	Description
0	Assembly successful, but there may have been warnings.
1	There were warnings, provided that the option --warnings_affect_exit_code was used.
2	There were non-fatal errors or fatal assembly errors (making the assembler abort).
3	There were crashing errors.

Table 35: 78K0R error return codes

Summary of 78K0R Assembler options

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

Command line option	Description
`--case_insensitive`	Case-insensitive user symbols
`--core`	Specifies the microcontroller core.
`-D`	Defines preprocessor symbols
`--debug`	Generates debug information
`--dependencies`	Lists file dependencies
`--diag_error`	Treats these diagnostics as errors
`--diag_remark`	Treats these diagnostics as remarks
`--diag_suppress`	Suppresses these diagnostics
`--diag_warning`	Treats these diagnostics as warnings
`--diagnostics_tables`	Lists all diagnostic messages
`--dir_first`	Allows directives in the first column
`--enable_multibytes`	Enables support for multibyte characters
`--error_limit`	Specifies the allowed number of errors before the assembler stops
`-f`	Extends the command line
`--generate_far_runtime_library_calls`	Generates `__far` runtime library calls
`--header_context`	Lists all referred source files
`-I`	Includes file paths
`-l`	Lists to named file
`-M`	Macro quote characters
`--mnem_first`	Allows mnemonics in the first column
`--no_warnings`	Disables all warnings
`--no_wrap_diagnostics`	Disables wrapping of diagnostic messages
`-o`	Sets object filename

Table 36: 78K0R Assembler options summary

Command line option	Description
--only_stdout	Uses standard output only
--preinclude	Includes an include file before reading the source file
--preprocess	Preprocessor output to file
-r	Generates debug information
--remarks	Enables remarks
--silent	Sets silent operation
--warnings_affect_exit_code	Warnings affect exit code
--warnings_are_errors	Treats all warnings as errors

Table 36: 78K0R Assembler options summary (Continued)

Descriptions of assembler options

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

Note that if you use the page **Extra Options** to specify specific command line options, there is no check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--case_insensitive

`--case_insensitive`

Use this option to make user symbols case insensitive.

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

You can also use the assembler directives `CASEON` and `CASEOFF` to control case sensitivity for user-defined symbols. See *Assembler control directives*, page 56, for more information.

Note: The `--case_insensitive` option does not affect preprocessor symbols. Preprocessor symbols are always case sensitive, regardless of whether they are defined in the IAR Embedded Workbench IDE or on the command line. See *Defining and undefining preprocessor symbols*, page 51.

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

`--core` `--core=78k0r`

This option is designed for selecting the processor core for which the code is to be generated. Because the IAR Assembler for 78K0R currently only supports the 78K0R processor core, the only available parameter is the default parameter:

```
--core=78k0r
```

Note: This option has been introduced for reasons of forwards compatibility. Currently it has no effect.

To set related options, use:

Project>Options>General Options>Target>Device

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

Defines a symbol to be used by the preprocessor with the name *`symbol`* and the value *`value`*. If no value is specified, 1 is used.

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

Example

You may want to 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 on the command line as follows:

Production version:	`a78k0r prog`
Test version:	`a78k0r 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:

```
a78k0r prog -DFRAMERATE=3
```

Project>Options>Assembler>Preprocessor>Defined symbols

`--debug, -r` `--debug`

```
-r
```

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

In order to reduce the size and link time of the object file, the assembler does not generate debug information by default.

Project>Options>Assembler >Output>Generate debug information

--dependencies

`--dependencies=[i][m] {`*`filename|directory`*`}`

When you use this option, each source file opened by the assembler is listed in a file. The following modifiers are available:

Option modifier	Description
`i`	Include only the names of files (default)
`m`	Makefile style

Table 37: Generating a list of dependencies (--dependencies)

If a *`filename`* is specified, the assembler stores the output in that file.

If a *`directory`* is specified, the assembler stores the output in that directory, in a file with the extension `i`. The filename will be the same as the name of the assembled source file, unless a different name has been specified with the option `-o`, in which case that name will be used.

To specify the working directory, replace *`directory`* with a period (`.`).

If `--dependencies` or `--dependencies=i` is used, the name of each opened source file, including the full path if available, is output on a separate line. For example:

```
c:\iar\product\include\stdio.h
d:\myproject\include\foo.h
```

If `--dependencies=m` is used, the output uses makefile style. For each source file, one line containing a makefile dependency rule is output. Each line consists of the name of the object file, a colon, a space, and the name of a source file. For example:

```
foo.r26: c:\iar\product\include\stdio.h
foo.r26: d:\myproject\include\foo.h
```

Example 1

To generate a listing of file dependencies to the file `listing.i`, use:

```
a78k0r prog --dependencies=i listing
```

Example 2

To generate a listing of file dependencies to a file called `listing.i` in the `mypath` directory, you would use:

```
a78k0r prog --dependencies \mypath\listing
```

Note: Both `\` and `/` can be used as directory delimiters.

Example 3

An example of using `--dependencies` with gmake:

1 Set up the rule for assembling files to be something like:

```
%.r26 : %.c
      $(ASM) $(ASMFLAGS) $< --dependencies=m $*.d
```

That is, besides producing an object file, the command also produces a dependent file in makefile style (in this example using the extension `.d`).

2 Include all the dependent files in the makefile using for example:

```
-include $(sources:.c=.d)
```

Because of the `-`, it works the first time, when the `.d` files do not yet exist.

This option is not available in the IAR Embedded Workbench IDE.

--diag_error

`--diag_error=`*`tag`*`,`*`tag,...`*

Use this option to classify diagnostic messages as errors.

An error indicates a violation of the assembler language rules, of such severity that object code will not be generated, and the exit code will not be 0.

The following example classifies warning `As001` as an error:

```
--diag_error=As001
```

Project>Options>Assembler >Diagnostics>Treat these as errors

--diag_remark

`--diag_remark=`*`tag`*`,`*`tag,...`*

Use this option to classify diagnostic messages as remarks.

A remark is the least severe type of diagnostic message and indicates a source code construct that may cause strange behavior in the generated code.

The following example classifies the warning `As001` as a remark:

```
--diag_remark=As001
```

Project>Options>Assembler >Diagnostics>Treat these as remarks

--diag_suppress `--diag_suppress=tag,tag,...`

Use this option to suppress diagnostic messages. The following example suppresses the warnings `As001` and `As002`:

```
--diag_suppress=As001,As002
```

Project>Options>Assembler >Diagnostics>Suppress these diagnostics

--diag_warning `--diag_warning=tag,tag,...`

Use this option to classify diagnostic messages as warnings.

A warning indicates an error or omission that is of concern, but which will not cause the assembler to stop before the assembly is completed.

The following example classifies the remark `As028` as a warning:

```
--diag_warning=As028
```

Project>Options>Assembler >Diagnostics>Treat these as warnings

--diagnostics_tables `--diagnostics_tables {filename|directory}`

Use this option to list all possible diagnostic messages in a named file. This can be very convenient, for example, if you have used a `#pragma` directive to suppress or change the severity level of any diagnostic messages, but forgot to document why.

This option cannot be given together with other options.

If a *`filename`* is specified, the assembler stores the output in that file.

If a *`directory`* is specified, the assembler stores the output in that directory, in a file with the name `diagnostics_tables.txt`. To specify the working directory, replace *`directory`* with a period (`.`).

Example 1

To output a list of all possible diagnostic messages to the file `diag.txt`, use:

```
--diagnostics_tables diag
```

Example 2

If you want to generate a table to a file `diagnostics_tables.txt` in the working directory, you could use:

```
--diagnostics_tables .
```

Both `\` and `/` can be used as directory delimiters.

This option is not available in the IAR Embedded Workbench IDE.

`--dir_first` `--dir_first`

The default behavior of the assembler is to treat all identifiers starting in the first column as labels.

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

Project>Options>Assembler >Language>Allow directives in first column

---

`--enable_multibytes` `--enable_multibytes`

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

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

Project>Options>Assembler>Language>Enable multibyte support

`--error_limit` `--error_limit=`*`n`*

Use the `--error_limit` option to specify the number of errors allowed before the assembler stops. By default, 100 errors are allowed. *`n`* must be a positive number; `0` indicates no limit.

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

`-f` `-f` *`filename`*

Extends the command line with text read from the specified file. 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:

```
a78k0r prog -f extend.xcl
```

To set this option, use:

Project>Options>Assembler>Extra Options

`--generate_far_runtime_library_calls`

`--generate_far_runtime_library_calls`

Use this option to generate `__far` runtime library calls. The option makes the assembler define the predefined macro `__USE_FAR_RT_CALLS__` and set the runtime model attribute `__far_rt_calls` to true.

Note: This option should be used together with the corresponding compiler option when building libraries that need to have the runtime assembler support routines placed in `__far` code.

To set this option, use:

Project>Options>Assembler>Extra Options

`--header_context`

`--header_context`

Occasionally, it is necessary to know which header file that was included from what source line, to find the cause of a problem. Use this option to list, for each diagnostic message, not only the source position of the problem, but also the entire include stack at that point.

This option is not available in the IAR Embedded Workbench IDE.

`-I`

`-I`*`prefix`*

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 `A78K_INC` environment variable. The `-I` option allows you to give the assembler the names of directories which it will also search if it fails to find the file in the current working directory.

Example

For example, using the options:

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

and then writing:

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

in the source, will make the assembler search first in the current directory, then in the directory `c:\global\`, and then in the directory `C:\thisproj\headers\`. Finally, the assembler searches the directories specified in the `A78K_INC` environment variable, provided that this variable is set.

Project>Options>Assembler >Preprocessor>Additional include directories

---

-l `-l[a][d][e][m][o][x][N] {`*`filename`*`|`*`directory`*`}`

By default, the assembler does not generate a listing. Use this option to generate a listing to a file.

You can choose to include one or more of the following types of information:

Command line option	Description
`-la`	Assembled lines only
`-ld`	The `LSTOUT` directive controls if lines are written to the list file or not. Using `-ld` turns the start value for this to off.
`-le`	No macro expansions
`-lm`	Macro definitions
`-lo`	Multiline code
`-lx`	Includes cross-references
`-lN`	Do not include diagnostics

Table 38: Conditional list options (-l)

If a *`filename`* is specified, the assembler stores the output in that file.

If a *`directory`* is specified, the assembler stores the output in that directory, in a file with the extension `lst`. The filename will be the same as the name of the assembled source file, unless a different name has been specified with the option `-o`, in which case that name will be used.

To specify the working directory, replace *`directory`* with a period (`.`).

Example 1

To generate a listing to the file `list.lst`, use:

```
a78k0r sourcefile -l list
```

Example 2

If you assemble the file `mysource.s26` and want to generate a listing to a file `mysource.lst` in the working directory, you could use:

```
a78k0r mysource -l .
```

Note: Both `\` and `/` can be used as directory delimiters.

To set related options, select:

Project>Options>Assembler >List

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

Specifies quote characters for macro arguments by setting 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.

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

```
a78k0r filename -M'<>'
```

Example

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

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

Project>Options>Assembler >Language>Macro quote characters

`--mnem_first` `--mnem_first`

The default behavior of the assembler is to treat all identifiers starting in the first column as labels.

Use this option to make mnemonics names (without a trailing colon) starting in the first column to be recognized as mnemonics.

Project>Options>Assembler >Language>Allow mnemonics in first column

--no_warnings `--no_warnings`

By default the assembler issues standard warning messages. Use this option to disable all warning messages.

This option is not available in the IAR Embedded Workbench IDE.

--no_wrap_diagnostics `--no_wrap_diagnostics`

By default, long lines in assembler diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.

This option is not available in the IAR Embedded Workbench IDE.

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

Use the `-o` option to specify an output file.

If a *`filename`* is specified, the assembler stores the object code in that file.

If a *`directory`* is specified, the assembler stores the object code in that directory, in a file with the same name as the name of the assembled source file, but with the extension `r26`. To specify the working directory, replace *`directory`* with a period (`.`).

Example 1

To store the assembler output in a file called `obj.r26` in the `mypath` directory, you would use:

```
a78k0r sourcefile -o \mypath\obj
```

Example 2

If you assemble the file `mysource.s26` and want to store the assembler output in a file `mysource.r26` in the working directory, you could use:

```
a78k0r mysource -o .
```

Note: Both `\` and `/` can be used as directory delimiters. You must include a space between the option itself and the filename.

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

--only_stdout

--only_stdout

Causes the assembler to use stdout also for messages that are normally directed to stderr.

This option is not available in the IAR Embedded Workbench IDE.

--preinclude

--preinclude *includefile*

Use this option to make the compiler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.

To set this option, use:

Project>Options>Assembler>Extra Options

--preprocess

--preprocess=[c][n][l] {*filename*|*directory*}

Use this option to direct preprocessor output to a named file.

The following table shows the mapping of the available preprocessor modifiers:

Command line option	Description
--preprocess=c	Preserve comments that otherwise are removed by the preprocessor, that is, C and C++ style comments. Assembler style comments are always preserved
--preprocess=n	Preprocess only
--preprocess=l	Generate #line directives

Table 39: Directing preprocessor output to file (--preprocess)

If a *filename* is specified, the assembler stores the output in that file.

If a *directory* is specified, the assembler stores the output in that directory, in a file with the extension i. The filename will be the same as the name of the assembled source file, unless a different name has been specified with the option -o, in which case that name will be used.

To specify the working directory, replace *directory* with a period (.).

Example 1

To store the assembler output with preserved comments to the file output.i, use:

```
a78k0r sourcefile --preprocess=c output
```

Example 2

If you assemble the file `mysource.s26` and want to store the assembler output with `#line` directives to a file `mysource.i` in the working directory, you could use:

```
a78k0r mysource --preprocess=l .
```

Note: Both `\` and `/` can be used as directory delimiters.

Project>Options>Assembler >Preprocessor>Preprocessor output to file

---

`-r, --debug`

```
-r
```

```
--debug
```

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

In order to reduce the size and link time of the object file, the assembler does not generate debug information by default.

Project>Options>Assembler >Output>Generate debug information

`--remarks`

```
--remarks
```

Use this option to make the assembler generate remarks, which is the least severe type of diagnostic message and which indicates a source code construct that may cause strange behavior in the generated code. By default remarks are not generated.

See *Severity levels*, page 73, for additional information about diagnostic messages.

Project>Options>Assembler >Diagnostics>Enable remarks

`--silent`

```
--silent
```

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

This option is not available in the IAR Embedded Workbench IDE.

--warnings_affect_exit_code

```
--warnings_affect_exit_code
```

By default the exit code is not affected by warnings, only errors produce a non-zero exit code. With this option, warnings will generate a non-zero exit code.

This option is not available in the IAR Embedded Workbench IDE.

--warnings_are_errors

```
--warnings_are_errors
```

Use this option to make the assembler treat all warnings as errors. If the assembler encounters an error, no object code is generated.

If you want to keep some warnings, you can use this option in combination with the option `--diag_warning`. First make all warnings become treated as errors and then reset the ones that should still be treated as warnings, for example:

```
--diag_warning=As001
```

For additional information, see --*diag_warning*, page 113.

Project>Options>Assembler >Diagnostics>Treat all warnings as errors

78K0R Assembler operators

This chapter first describes the precedence of the 78K0R Assembler operators, and then summarizes the operators, classified according to their precedence. Finally, this chapter provides reference information about each operator, presented in alphabetical order.

For information about the operators for the 78K0/78K0S Assembler, see the chapter *78K0/78K0S Assembler operators*.

Precedence of assembler operators

Each operator has a precedence number assigned to it that determines the order in which the operator and its operands are evaluated. The precedence numbers range from 1 (the highest precedence, that is, first evaluated) to 15 (the lowest precedence, that is, last evaluated).

The following rules determine how expressions are evaluated:

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

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

Note: The precedence order in the IAR Assembler for 78K0R closely follows the precedence order of the ANSI C++ standard for operators, where applicable.

Summary of assembler operators

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

PARENTHESIS OPERATOR – 1

`()`	Parenthesis.

FUNCTION OPERATORS – 2

`BYTE1`	First byte.
`BYTE2`	Second byte.
`BYTE3`	Third byte.
`BYTE4`	Fourth byte.
`DATE`	Current date/time.
`HIGH`	High byte.
`HWRD`	High word.
`LOW`	Low byte.
`LWRD`	Low word.
`SFB`	Segment begin.
`SFE`	Segment end.
`SIZEOF`	Segment size.
`UPPER`	Third byte.

UNARY OPERATORS – 3

`+`	Unary plus.
`BINNOT [~]`	Bitwise NOT.
`NOT [!]`	Logical NOT.
`-`	Unary minus.

MULTIPLICATIVE ARITHMETIC OPERATORS – 4

`*`	Multiplication.
`/`	Division.
`MOD [%]`	Modulo.

ADDITIVE ARITHMETIC OPERATORS – 5

`+`	Addition.
`–`	Subtraction.

SHIFT OPERATORS – 6

`SHL [<<]`	Logical shift left.
`SHR [>>]`	Logical shift right.

COMPARISON OPERATORS – 7

`GE [>=]`	Greater than or equal.
`GT [>]`	Greater than.
`LE [<=]`	Less than or equal.
`LT [<]`	Less than.
`UGT`	Unsigned greater than.
`ULT`	Unsigned less than.

EQUIVALENCE OPERATORS – 8

`EQ [=] [==]`	Equal.
`NE [<>] [!=]`	Not equal.

LOGICAL OPERATORS – 9-14

`BINAND [&]`	Bitwise AND (9).
`BINXOR [^]`	Bitwise exclusive OR (10).
`BINOR [|]`	Bitwise OR (11).

`AND [&&]`	Logical AND (12).
`XOR`	Logical exclusive OR (13).
`OR [||]`	Logical OR (14).

CONDITIONAL OPERATOR – 15

`?:`	Conditional operator.

Descriptions of assembler operators

The following sections give full descriptions of each assembler operator. The number within parentheses specifies the priority of the operator

`()` Parenthesis (1).

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

Example

```
1+2*3 → 7
(1+2)*3 → 9
```

`*` Multiplication (4).

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

Example

```
2*2 → 4
-2*2 → -4
```

`+` Unary plus (3).

Unary plus operator.

Example

```
+3 → 3
3*+2 → 6
```

+ Addition (5).

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

Example

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

- Unary minus (3).

The unary minus operator performs arithmetic negation on its operand.

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

Example

```
-3 → -3
3*-2 → -6
4--5 → 9
```

- Subtraction (5).

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

Example

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

/ Division (4).

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

Example

```
9/2 → 4
-12/3 → -4
9/2*6 → 24
```

`?:` Conditional operator (15).

The result of this operator is the first *expr* if *condition* evaluates to true and the second *expr* if *condition* evaluates to false.

Note: The question mark and a following label must be separated by space or a tab, otherwise the `?` will be considered the first character of the label.

Syntax

condition ? *expr* : *expr*

Example

```
5 ? 6 : 7 →6
0 ? 6 : 7 →7
```

`AND [&&]` Logical AND (12).

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

Example

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

`BINAND [&]` Bitwise AND (9).

Use `BINAND` to perform bitwise AND between the integer operands. Each bit in the 32-bit result is the logical AND of the corresponding bits in the operands.

Example

```
1010B BINAND 0011B → 0010B
1010B BINAND 0101B → 0000B
1010B BINAND 0000B → 0000B
```

BINNOT [~] Bitwise NOT (3).

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

Example

```
BINNOT 1010B → 11111111111111111111111111110101B
```

BINOR [|] Bitwise OR (11).

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

Example

```
1010B BINOR 0101B → 1111B
1010B BINOR 0000B → 1010B
```

BINXOR [^] Bitwise exclusive OR (10).

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

Example

```
1010B BINXOR 0101B → 1111B
1010B BINXOR 0011B → 1001B
```

BYTE1 First byte (2).

BYTE1 takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the low byte (bits 7 to 0) of the operand.

Example

```
BYTE1 0x12345678 → 0x78
```

`BYTE2` Second byte (2).

`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 (2).

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

Example

`BYTE3 0x12345678` → `0x34`

`BYTE4` Fourth byte (2).

`BYTE4` takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the high byte (bits 31 to 24) of the operand.

Example

`BYTE4 0x12345678` → `0x12`

`DATE` Current date/time (2).

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 (8).

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

Example

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

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

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

Example

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

`GT [>]` Greater than (7).

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

Example

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

`HIGH` High byte (2).

`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 (2).

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

Example

```
HWRD 0x12345678 → 0x1234
```

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

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

Example

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

`LOW` Low byte (2).

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

Example

```
LOW 0xABCD → 0xCD
```

`LT [<]` Less than (7).

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

Example

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

LWRD Low word (2).

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 (4).

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.

Example

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

NE [<>] [!=] Not equal (8).

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

Example

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

NOT [!] Logical NOT (3).

Use NOT to negate a logical argument.

Example

```
NOT 0101B → 0
NOT 0000B → 1
```

`OR [||]` Logical OR (14).

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

Example

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

`SFB` Segment begin (2).

`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 link time.

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.

Example

```
       NAME   demo
       RSEG   segtab:CONST
start: DC16   SFB(mycode)
```

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 (2).

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

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.

Example

```
      NAME   demo
      RSEG   segtab:CONST
end:  DC16   SFE(mycode)
```

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

The size of the segment MY_SEGMENT can be calculated as:

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

SHL [<<] Logical shift left (6).

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.

Example

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

SHR [>>] Logical shift right (6).

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.

Example

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

SIZEOF Segment size (2).

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

Syntax

SIZEOF (*segment*)

Parameters

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

Example

The following code sets size to the size of the segment mycode.

```
      MODULE  table
      RSEG    mycode:CODE   ;forward declaration of mycode
      RSEG    segtab:CONST
size: DC32    SIZEOF(mycode)
      ENDMOD

      MODULE  application
      RSEG    mycode:CODE
      NOP                   ;placeholder for application code
      ENDMOD
```

UGT Unsigned greater than (7).

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

Example

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

ULT Unsigned less than (7).

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

Example

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

UPPER Third byte (2).

UPPER 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

```
UPPER 0x12345678 → 0x34
```

XOR Logical exclusive OR (13).

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

Example

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

78K0R pragma directives

This chapter describes the pragma directives of the 78K0R Assembler. The 78K0/78K0S Assembler cannot use pragma directives.

The pragma directives control the behavior of the assembler, for example whether it outputs warning messages. The pragma directives are preprocessed, which means that macros are substituted in a pragma directive.

Summary of pragma directives

The following table shows the pragma directives of the assembler:

#pragma directive	Description
`#pragma diag_default`	Changes the severity level of diagnostic messages
`#pragma diag_error`	Changes the severity level of diagnostic messages
`#pragma diag_remark`	Changes the severity level of diagnostic messages
`#pragma diag_suppress`	Suppresses diagnostic messages
`#pragma diag_warning`	Changes the severity level of diagnostic messages
`#pragma message`	Prints a message

Table 40: Pragma directives summary

Descriptions of pragma directives

All pragma directives using = for value assignment should be entered like:

`#pragma` *`pragmaname`*`=`*`pragmavalue`*

or

`#pragma` *`pragmaname`* `=` *`pragmavalue`*

#pragma diag_default

`#pragma diag_default=`*`tag`*`,`*`tag`*`,...`

Changes the severity level back to default or as defined on the command line for the diagnostic messages with the specified tags. For example:

```
#pragma diag_default=Pe117
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma diag_error

`#pragma diag_error=`*`tag`*`,`*`tag`*`,...`

Changes the severity level to `error` for the specified diagnostics. For example:

```
#pragma diag_error=Pe117
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma diag_remark

`#pragma diag_remark=`*`tag`*`,`*`tag`*`,...`

Changes the severity level to `remark` for the specified diagnostics. For example:

```
#pragma diag_remark=Pe177
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma diag_suppress

`#pragma diag_suppress=`*`tag`*`,`*`tag`*`,...`

Suppresses the diagnostic messages with the specified tags. For example:

```
#pragma diag_suppress=Pe117,Pe177
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma diag_warning

`#pragma diag_warning=`*`tag`*`,`*`tag`*`,...`

Changes the severity level to `warning` for the specified diagnostics. For example:

```
#pragma diag_warning=Pe826
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma message

`#pragma message(`*`string`*`)`

Makes the assembler print a message on `stdout` when the file is assembled. For example:

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Symbols

Numerics