

# IAR C/C++ Compilers

Reference Guide

for the Renesas

**78K0/78K0S and 78K0R**

**Microcontroller Subfamilies**



C78K-4

 IAR  
SYSTEMS

## **COPYRIGHT NOTICE**

Copyright © 2000–2010 IAR Systems AB.

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

## **DISCLAIMER**

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

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

## **TRADEMARKS**

IAR Systems, IAR Embedded Workbench, C-SPY, visualSTATE, From Idea To Target, IAR KickStart Kit, IAR PowerPac, IAR YellowSuite, IAR Advanced Development Kit, IAR, and the IAR Systems logotype are trademarks or registered trademarks owned by IAR Systems AB. J-Link is a trademark licensed to IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Renesas is a registered trademark of Renesas Electronics Corporation.

Adobe and Acrobat Reader are registered trademarks of Adobe Systems Incorporated.

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

## **EDITION NOTICE**

Fourth edition: May 2010

Part number: C78K-4

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

Internal reference: R13, Too6.0, IMAE.

# Brief contents

Tables .....	xix
Preface .....	xxi
<b>Part 1. Using the compilers</b> .....	<b>1</b>
Getting started .....	3
Data storage .....	11
Functions .....	21
Placing code and data .....	39
The DLIB runtime environment .....	53
The CLIB runtime environment .....	83
Assembler language interface .....	91
Using C++ .....	109
Efficient coding for embedded applications .....	123
<b>Part 2. Reference information</b> .....	<b>141</b>
External interface details .....	143
Compiler options .....	149
Data representation .....	181
Compiler extensions .....	193
Extended keywords .....	203
Pragma directives .....	217
Intrinsic functions .....	233
The preprocessor .....	237

Library functions .....	243
Segment reference .....	253
Implementation-defined behavior .....	267
Index .....	281

# Contents

Tables .....	xix
Preface .....	xxi
<b>Who should read this guide</b> .....	xxi
<b>How to use this guide</b> .....	xxi
<b>What this guide contains</b> .....	xxii
<b>Other documentation</b> .....	xxiii
Further reading .....	xxiv
<b>Document conventions</b> .....	xxiv
Typographic conventions .....	xxv
Naming conventions .....	xxv
<b>Part I. Using the compilers</b> .....	1
Getting started .....	3
<b>IAR language overview</b> .....	3
<b>Supported 78K devices</b> .....	4
<b>Building applications—an overview</b> .....	4
Compiling .....	5
Linking .....	5
<b>Basic settings for project configuration</b> .....	6
Processor core .....	6
Code model .....	7
Data model (78K0R only) .....	7
Optimization for speed and size .....	7
Runtime environment .....	8
<b>Special support for embedded systems</b> .....	9
Extended keywords .....	10
Pragma directives .....	10
Predefined symbols .....	10
Special function types .....	10
Accessing low-level features .....	10

Data storage .....	11
<b>Introduction</b> .....	11
Different ways to store data .....	11
<b>Data models (78K0R only)</b> .....	12
Specifying a data model .....	12
<b>Memory types</b> .....	13
Short address Area .....	14
Near .....	14
Far (78K0R only) .....	14
SFR .....	14
Using data memory attributes .....	14
Pointers and memory types .....	16
Structures and memory types .....	16
More examples .....	17
<b>C++ and memory types</b> .....	17
<b>Auto variables—on the stack</b> .....	18
The stack .....	18
<b>Dynamic memory on the heap</b> .....	20
Functions .....	21
<b>Function-related extensions</b> .....	21
<b>Code models and memory attributes for function storage</b> .....	21
Using function memory attributes .....	22
<b>Primitives for interrupts, concurrency, and OS-related programming</b> .....	23
Interrupt functions .....	23
Monitor functions .....	24
C++ and special function types .....	27
<b>Banked functions (78K0/78K0S only)</b> .....	27
The 78K0/78K0S banking system .....	28
Using the banked code model .....	28
How banking works .....	29
Writing source code for banked memory .....	31
The bank switching routine .....	34

Downloading to memory .....	36
Debugging banked applications .....	37
<b>Placing code and data</b> .....	39
<b>Segments and memory</b> .....	39
What is a segment? .....	39
<b>Placing segments in memory</b> .....	40
Customizing the linker command file .....	40
<b>Data segments</b> .....	43
Static memory segments .....	43
The stack .....	46
The heap .....	47
Located data .....	49
User-defined segments .....	49
<b>Code segments</b> .....	49
Startup code .....	49
Normal code .....	49
CALLF declared functions (78K0 only) .....	51
Interrupt and callt vectors .....	51
C++ dynamic initialization .....	51
<b>Verifying the linked result of code and data placement</b> .....	51
Segment too long errors and range errors .....	52
Linker map file .....	52
<b>The DLIB runtime environment</b> .....	53
<b>Introduction to the runtime environment</b> .....	53
Runtime environment functionality .....	53
Library selection .....	54
Situations that require library building .....	55
Library configurations .....	55
Debug support in the runtime library .....	56
<b>Using a prebuilt library</b> .....	56
Customizing a prebuilt library without rebuilding .....	58
<b>Choosing formatters for printf and scanf</b> .....	59
Choosing printf formatter .....	59

Choosing scanf formatter .....	60
<b>Overriding library modules .....</b>	<b>61</b>
<b>Building and using a customized library .....</b>	<b>62</b>
Setting up a library project .....	63
Modifying the library functionality .....	63
Using a customized library .....	63
<b>System startup and termination .....</b>	<b>64</b>
System startup .....	64
System termination .....	65
<b>Customizing system initialization .....</b>	<b>66</b>
__low_level_init .....	67
Modifying the file cstartup.s26 .....	67
<b>Standard streams for input and output .....</b>	<b>67</b>
Implementing low-level character input and output .....	68
<b>Configuration symbols for printf and scanf .....</b>	<b>70</b>
Customizing formatting capabilities .....	71
<b>File input and output .....</b>	<b>71</b>
<b>Locale .....</b>	<b>72</b>
Locale support in prebuilt libraries .....	72
Customizing the locale support .....	72
Changing locales at runtime .....	73
<b>Environment interaction .....</b>	<b>74</b>
<b>Signal and raise .....</b>	<b>75</b>
<b>Time .....</b>	<b>75</b>
<b>Strtod .....</b>	<b>75</b>
<b>Assert .....</b>	<b>76</b>
<b>Heaps .....</b>	<b>76</b>
<b>Hardware support .....</b>	<b>76</b>
<b>C-SPY runtime interface .....</b>	<b>77</b>
Low-level debugger runtime interface .....	77
The debugger terminal I/O window .....	78
<b>Checking module consistency .....</b>	<b>78</b>
Runtime model attributes .....	79
Using runtime model attributes .....	79

Predefined runtime attributes .....	80
User-defined runtime model attributes .....	81
The CLIB runtime environment .....	83
<b>Runtime environment</b> .....	83
<b>Input and output</b> .....	85
Character-based I/O .....	85
Formatters used by printf and sprintf .....	85
Formatters used by scanf and sscanf .....	87
<b>System startup and termination</b> .....	87
System startup .....	87
System termination .....	88
<b>Overriding default library modules</b> .....	88
<b>Customizing system initialization</b> .....	88
<b>C-SPY runtime interface</b> .....	88
The debugger terminal I/O window .....	89
Termination .....	89
<b>Checking module consistency</b> .....	89
Assembler language interface .....	91
<b>Mixing C and assembler</b> .....	91
Intrinsic functions .....	91
Mixing C and assembler modules .....	92
Inline assembler .....	93
<b>Calling assembler routines from C</b> .....	94
Creating skeleton code .....	94
Compiling the code .....	95
<b>Calling assembler routines from C++</b> .....	96
<b>Calling convention</b> .....	97
Function declarations .....	98
Using C linkage in C++ source code .....	98
Preserved versus scratch registers .....	99
Function entrance .....	99
Function exit .....	101
Examples .....	102

Function directives .....	103
<b>Call frame information</b> .....	104
CFI directives .....	104
Creating assembler source with CFI support .....	105
Using C++ .....	109
<b>Overview</b> .....	109
Standard Embedded C++ .....	109
Extended Embedded C++ .....	110
Enabling C++ support .....	110
<b>Feature descriptions</b> .....	111
Classes .....	111
Function types .....	114
New and Delete operators (78K0R only) .....	114
Templates .....	116
Variants of casts .....	118
Mutable .....	119
Namespace .....	119
The STD namespace .....	119
Pointer to member functions .....	119
Using interrupts and EC++ destructors .....	119
<b>C++ language extensions</b> .....	120
Efficient coding for embedded applications .....	123
<b>Selecting data types</b> .....	123
Using efficient data types .....	123
Floating-point types .....	124
Alignment of elements in a structure .....	124
Anonymous structs and unions .....	125
<b>Controlling data and function placement in memory</b> .....	126
Data placement at an absolute location .....	127
Data and function placement in segments .....	129
<b>Controlling compiler optimizations</b> .....	130
Scope for performed optimizations .....	131
Optimization levels .....	131

Speed versus size .....	132
Fine-tuning enabled transformations .....	132
<b>Writing efficient code</b> .....	135
Writing optimization-friendly code .....	135
Saving stack space and RAM memory .....	136
Function prototypes .....	136
Integer types and bit negation .....	137
Protecting simultaneously accessed variables .....	137
Accessing special function registers .....	138
Non-initialized variables .....	139
<b>Part 2. Reference information</b> .....	141
External interface details .....	143
<b>Invocation syntax</b> .....	143
Compiler invocation syntax .....	143
Passing options .....	143
Environment variables .....	144
<b>Include file search procedure</b> .....	144
<b>Compiler output</b> .....	145
<b>Diagnostics</b> .....	146
Message format .....	147
Severity levels .....	147
Setting the severity level .....	148
Internal error .....	148
Compiler options .....	149
<b>Options syntax</b> .....	149
Types of options .....	149
Rules for specifying parameters .....	149
<b>Summary of compiler options</b> .....	152
<b>Descriptions of options</b> .....	154
--aggressive_inlining .....	154
--char_is_signed .....	155

--clib .....	155
--code_model .....	155
--code_segment .....	156
--core .....	156
-D .....	157
--data_model .....	157
--debug, -r .....	158
--dependencies .....	158
--diag_error .....	159
--diag_remark .....	160
--diag_suppress .....	160
--diag_warning .....	160
--diagnostics_tables .....	161
--disable_data_alignment .....	161
--discard_unused_publics .....	163
--dlib_config .....	163
-e .....	164
--ec++ .....	164
--eec++ .....	164
--enable_multibytes .....	165
--error_limit .....	165
-f .....	165
--generate_callt_runtime_library_calls .....	166
--generate_far_runtime_library_calls .....	166
--header_context .....	166
-I .....	167
-l .....	167
--library_module .....	168
--mfc .....	168
--migration_preprocessor_extensions .....	169
--module_name .....	169
--near_const_location .....	170
--no_code_motion .....	170
--no_cse .....	170

--no_inline .....	171
--no_path_in_file_macros .....	171
--no_tbaa .....	172
--no_typedefs_in_diagnostics .....	172
--no_unroll .....	173
--no_warnings .....	173
--no_wrap_diagnostics .....	173
-O .....	174
-o, --output .....	174
--omit_types .....	175
--only_stdout .....	175
--output, -o .....	175
--predef_macros .....	176
--preinclude .....	176
--preprocess .....	176
--public_equ .....	177
-r, --debug .....	177
--remarks .....	177
--require_prototypes .....	178
--silent .....	178
--strict_ansi .....	178
--warnings_affect_exit_code .....	179
--warnings_are_errors .....	179
--workseg_area .....	179
<b>Data representation</b> .....	<b>181</b>
<b>Alignment</b> .....	<b>181</b>
Alignment on the 78K microcontroller .....	182
<b>Basic data types</b> .....	<b>182</b>
Integer types .....	182
Floating-point types .....	184
<b>Pointer types</b> .....	<b>185</b>
Function pointers .....	185
Data pointers .....	185

Casting .....	186
<b>Structure types</b> .....	187
Alignment .....	187
General layout .....	187
Packed structure types .....	188
<b>Type qualifiers</b> .....	189
Declaring objects volatile .....	189
Declaring objects const .....	190
<b>Data types in C++</b> .....	191
Compiler extensions .....	193
<b>Compiler extensions overview</b> .....	193
Enabling language extensions .....	194
<b>C language extensions</b> .....	194
Important language extensions .....	194
Useful language extensions .....	196
Minor language extensions .....	199
Extended keywords .....	203
<b>General syntax rules for extended keywords</b> .....	203
Type attributes .....	204
Object attributes .....	206
<b>Summary of extended keywords</b> .....	207
<b>Descriptions of extended keywords</b> .....	208
__banked .....	208
__callf .....	209
__callt .....	209
__far .....	210
__far_func .....	210
__interrupt .....	211
__intrinsic .....	211
__monitor .....	211
__near .....	212
__near_func .....	212
__no_bit_access .....	213

__no_init .....	213
__non_banked .....	213
__noreturn .....	214
__no_save .....	214
__root .....	214
__saddr .....	215
__sfr .....	215
__V2_call .....	216
__V3_call .....	216
<b>Pragma directives .....</b>	<b>217</b>
<b>Summary of pragma directives .....</b>	<b>217</b>
<b>Descriptions of pragma directives .....</b>	<b>218</b>
bank .....	218
basic_template_matching .....	219
bitfields .....	219
constseg .....	220
data_alignment .....	220
dataseg .....	221
diag_default .....	221
diag_error .....	222
diag_remark .....	222
diag_suppress .....	222
diag_warning .....	223
include_alias .....	223
inline .....	224
language .....	224
location .....	225
message .....	225
object_attribute .....	226
optimize .....	226
pack .....	227
__printf_args .....	228
required .....	228

rtmodel .....	229
__scanf_args .....	229
segment .....	230
type_attribute .....	230
unroll .....	231
vector .....	231
<b>Intrinsic functions</b> .....	233
<b>Summary of intrinsic functions</b> .....	233
<b>Descriptions of intrinsic functions</b> .....	234
__break .....	234
__disable_interrupt .....	234
__enable_interrupt .....	234
__get_interrupt_state .....	234
__halt .....	235
__no_operation .....	235
__set_interrupt_state .....	235
__stop .....	235
<b>The preprocessor</b> .....	237
<b>Overview of the preprocessor</b> .....	237
<b>Descriptions of predefined preprocessor symbols</b> .....	238
<b>Descriptions of miscellaneous preprocessor extensions</b> .....	240
NDEBUG .....	240
_Pragma() .....	241
#warning message .....	241
__VA_ARGS__ .....	242
<b>Library functions</b> .....	243
<b>Introduction</b> .....	243
Header files .....	243
Library object files .....	244
Reentrancy .....	244
<b>IAR DLIB Library</b> .....	245
C header files .....	245

C++ header files .....	246
Library functions as intrinsic functions .....	248
Added C functionality .....	248
<b>IAR CLIB Library</b> .....	250
Library definitions summary .....	251
Segment reference .....	253
<b>Summary of segments</b> .....	253
<b>Descriptions of segments</b> .....	254
BCODE .....	255
CHECKSUM .....	255
CLTVEC .....	255
CODE .....	256
CONST .....	256
CSTACK .....	256
DIFUNCT .....	257
FAR_A .....	257
FAR_CONST .....	257
FAR_HEAP .....	258
FAR_I .....	258
FAR_ID .....	258
FAR_N .....	259
FAR_Z .....	259
FCODE .....	259
HEAP .....	260
INTVEC .....	260
NEAR_A .....	260
NEAR_CONST .....	261
NEAR_HEAP .....	261
NEAR_I .....	261
NEAR_ID .....	262
NEAR_N .....	262
NEAR_Z .....	262
RCODE .....	263

SADDR_A .....	263
SADDR_I .....	263
SADDR_ID .....	263
SADDR_N .....	264
SADDR_Z .....	264
SWITCH .....	264
WRKSEG .....	265
XCODE .....	265
<b>Implementation-defined behavior .....</b>	<b>267</b>
<b>Descriptions of implementation-defined behavior .....</b>	<b>267</b>
Translation .....	267
Environment .....	268
Identifiers .....	268
Characters .....	268
Integers .....	270
Floating point .....	270
Arrays and pointers .....	271
Registers .....	271
Structures, unions, enumerations, and bitfields .....	271
Qualifiers .....	272
Declarators .....	272
Statements .....	272
Preprocessing directives .....	273
IAR DLIB Library functions .....	274
IAR CLIB Library functions .....	277
<b>Index .....</b>	<b>281</b>

# Tables

1: Typographic conventions used in this guide .....	xxv
2: Naming conventions used in this guide .....	xxv
3: Processor cores .....	6
4: Command line options for specifying library and dependency files .....	9
5: Data model characteristics .....	13
6: Memory types and their corresponding keywords .....	15
7: Code models .....	21
8: Function memory attributes .....	22
9: Standard bank switching cost in cycles .....	34
10: Fast bank switching cost in cycles .....	35
11: XLINK segment memory types .....	40
12: Memory layout of a target system (example) .....	41
13: Memory types with corresponding segment groups .....	44
14: Segment name suffixes .....	44
15: Library configurations .....	55
16: Levels of debugging support in runtime libraries .....	56
17: Prebuilt libraries .....	57
18: Customizable items .....	58
19: Formatters for printf .....	59
20: Formatters for scanf .....	60
21: Descriptions of printf configuration symbols .....	70
22: Descriptions of scanf configuration symbols .....	70
23: Low-level I/O files .....	71
24: Heaps and memory types .....	76
25: Functions with special meanings when linked with debug info .....	77
26: Example of runtime model attributes .....	79
27: Predefined runtime model attributes .....	80
28: Runtime libraries .....	84
29: Registers used for passing parameters .....	100
30: Registers used for returning values .....	101
31: Call frame information resources defined in a names block .....	104

32: Compiler optimization levels .....	131
33: Compiler environment variables .....	144
34: Error return codes .....	146
35: Compiler options summary .....	152
36: Integer types .....	182
37: Floating-point types .....	184
38: Function pointers .....	185
39: Data pointers .....	185
40: Extended keywords summary .....	207
41: Pragma directives summary .....	217
42: Intrinsic functions summary .....	233
43: Predefined symbols .....	238
44: Traditional standard C header files—DLIB .....	245
45: Embedded C++ header files .....	246
46: Additional Embedded C++ header files—DLIB .....	247
47: Standard template library header files .....	247
48: New standard C header files—DLIB .....	248
49: IAR CLIB Library header files .....	251
50: Segment summary .....	253
51: Message returned by <code>strerror()</code> —IAR DLIB library .....	277
52: Message returned by <code>strerror()</code> —IAR CLIB library .....	280

# Preface

Welcome to the IAR C/C++ Compilers Reference Guide for 78K. The purpose of this guide is to provide you with detailed reference information that can help you to use the compilers to best suit your application requirements. This guide also gives you suggestions on coding techniques so that you can develop applications with maximum efficiency.

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

---

## Who should read this guide

Read this guide if you plan to develop an application using the C or C++ language for the 78K microcontroller and need detailed reference information on how to use the compilers. You should have working knowledge of:

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

---

## How to use this guide

When you start using the IAR C/C++ Compilers for 78K, you should read *Part 1. Using the compilers* in this guide.

When you are familiar with the compiler and have already configured your project, you can focus more on *Part 2. Reference information*.

If you are new to using the IAR Systems build tools, we recommend that you first study the *IAR Embedded Workbench® IDE User Guide*. This guide contains a product

overview, tutorials that can help you get started, conceptual and user information about the IDE and the IAR C-SPY® Debugger, and corresponding reference information.

---

## What this guide contains

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

### **Part 1. Using the compilers**

- *Getting started* gives the information you need to get started using the compiler for efficiently developing your application.
- *Data storage* describes how to store data in memory, focusing on the memory types. For the 78K0R core, the concept of data models is introduced.
- *Functions* gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.
- *Placing code and data* describes the concept of segments, introduces the linker command file, and describes how code and data are placed in memory.
- *The DLIB runtime environment* describes the DLIB runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization introducing the file `cstartup`, how to use modules for locale, and file I/O.
- *The CLIB runtime environment* gives an overview of the CLIB runtime libraries and how to customize them. The chapter also describes system initialization and introduces the file `cstartup`.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

### **Part 2. Reference information**

- *External interface details* provides reference information about how the compiler interacts with its environment—the invocation syntax, methods for passing options to the compiler, environment variables, the include file search procedure, and the different types of compiler output. The chapter also describes how the compiler’s diagnostic system works.
- *Compiler options* explains how to set options, gives a summary of the options, and contains detailed reference information for each compiler option.

- *Data representation* describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.
- *Compiler extensions* gives a brief overview of the compiler extensions to the ISO/ANSI C standard. More specifically the chapter describes the available C language extensions.
- *Extended keywords* gives reference information about each of the 78K-specific keywords that are extensions to the standard C/C++ language.
- *Pragma directives* gives reference information about the pragma directives.
- *Intrinsic functions* gives reference information about functions to use for accessing 78K-specific low-level features.
- *The preprocessor* gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.
- *Library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.
- *Segment reference* gives reference information about the compiler's use of segments.
- *Implementation-defined behavior* describes how the compiler handles the implementation-defined areas of the C language standard.

---

## Other documentation

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

- Using the IDE and the IAR C-SPY Debugger®, refer to the *IAR Embedded Workbench® IDE User Guide*
- Programming for the 78K IAR Assembler, refer to the *IAR Assemblers 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 functions, refer to the online help system
- Using the IAR CLIB Library functions, refer to the *IAR C Library Functions Reference Guide*, available from the online help system.
- Porting application code and projects created with a previous IAR Embedded Workbench for 78K, refer to the *78K IAR Embedded Workbench Migration Guide*
- Using the MISRA-C:1998 rules or the MISRA-C:2004 rules, refer to the *IAR Embedded Workbench® MISRA C:1998 Reference Guide* or the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*, respectively.

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

## FURTHER READING

These books might be of interest to you when using the IAR Systems development tools:

- Barr, Michael, and Andy Oram, ed. *Programming Embedded Systems in C and C++*. O'Reilly & Associates.
- Harbison, Samuel P. and Guy L. Steele (contributor). *C: A Reference Manual*. Prentice Hall.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall. [The later editions describe the ANSI C standard.]
- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C*. R&D Books.
- Lippman, Stanley B. and Josée Lajoie. *C++ Primer*. Addison-Wesley.
- Mann, Bernhard. *C für Mikrocontroller*. Franzis-Verlag. [Written in German.]
- Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley.

We recommend that you visit these web sites:

- The Renesas web site, [www.renesas.com](http://www.renesas.com), contains information and news about the 78K microcontrollers.
- The IAR Systems web site, [www.iar.com](http://www.iar.com), holds application notes and other product information.
- Finally, the Embedded C++ Technical Committee web site, [www.caravan.net/ec2plus](http://www.caravan.net/ec2plus), contains information about the Embedded C++ standard.

---

## Document conventions

When, in this text, we refer to the programming language C, the text also applies to C++, unless otherwise stated.

When referring to a directory in your product installation, for example `78k\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.n\78k\doc`.

## TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

Style	Used for
<code>computer</code>	<ul style="list-style-type: none"> <li>• Source code examples and file paths.</li> <li>• Text on the command line.</li> <li>• Binary, hexadecimal, and octal numbers.</li> </ul>
<code>parameter</code>	A placeholder for an actual value used as a parameter, for example <code>filename.h</code> where <code>filename</code> represents the name of the file.
<code>[option]</code>	An optional part of a command.
<code>a b c</code>	Alternatives in a command.
<code>{a b c}</code>	A mandatory part of a command with alternatives.
<b>bold</b>	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>italic</i>	<ul style="list-style-type: none"> <li>• A cross-reference within this guide or to another guide.</li> <li>• Emphasis.</li> </ul>
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

Table 1: Typographic conventions used in this guide

## NAMING CONVENTIONS

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

Brand name	Generic term
IAR Embedded Workbench® for 78K	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for 78K	the IDE
IAR C-SPY® Debugger for 78K	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for 78K	the compiler

Table 2: Naming conventions used in this guide

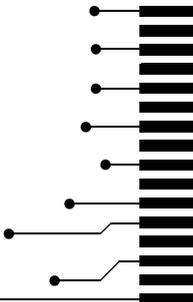
<b>Brand name</b>	<b>Generic term</b>
IAR Assembler™ for 78K	the assembler
IAR XLINK™ Linker	XLINK, the linker
IAR XAR Library builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Library™	the DLIB library
IAR CLIB Library™	the CLIB library

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

# Part I. Using the compilers

This part of the *IAR C/C++ Compilers Reference Guide for 78K* includes these chapters:

- Getting started
- Data storage
- Functions
- Placing code and data
- The DLIB runtime environment
- The CLIB runtime environment
- Assembler language interface
- Using C++
- Efficient coding for embedded applications.





# Getting started

This chapter gives the information you need to get started using the compilers for efficiently developing your application.

First you will get an overview of the supported programming languages, followed by a description of the steps involved for compiling and linking an application.

Next, the compilers are introduced. You will get an overview of the basic settings needed for a project setup, including an overview of the techniques that enable applications to take full advantage of the 78K microcontroller. In the following chapters, these techniques are studied in more detail.

---

## IAR language overview

There are two high-level programming languages you can use with the IAR C/C++ Compiler for 78K:

- C, the most widely used high-level programming language in the embedded systems industry. Using the IAR C/C++ Compiler for 78K, you can build freestanding applications that follow the standard ISO 9899:1990. This standard is commonly known as ANSI C.
- C++, a modern object-oriented programming language with a full-featured library well suited for modular programming. IAR Systems supports two levels of the C++ language:
  - Embedded C++ (EC++), a subset of the C++ programming standard, which is intended for embedded systems programming. It is defined by an industry consortium, the Embedded C++ Technical committee. See the chapter *Using C++*.
  - IAR Extended Embedded C++, with additional features such as full template support, namespace support, the new cast operators, as well as the Standard Template Library (STL).

Each of the supported languages can be used in *strict* or *relaxed* mode, or relaxed with IAR extensions enabled. The strict mode adheres to the standard, whereas the relaxed mode allows some deviations from the standard. For more details, see the chapter *Compiler extensions*.

For information about how the compiler handles the implementation-defined areas of the C language, see the chapter *Implementation-defined behavior*.

It is also possible to implement parts of the application, or the whole application, in assembler language. See the *IAR Assemblers Reference Guide for 78K*.

For more information about the Embedded C++ language and Extended Embedded C++, see the chapter *Using C++*.

---

## Supported 78K devices

The IAR C/C++ Compilers for 78K supports all devices based on the standard Renesas 78K0/78K0S and 78K0R microcontroller cores. The support has been implemented in the form of separate compiler executable files for the 78K0 and 78K0S cores on the one hand and the 78K0R core on the other. This affects a number of compiler features, from the command line needed for compiling, to concepts like data models.

Except for those cases where the compilers behave differently, both compilers will be referred to as the IAR C/C++ Compilers for 78K in this guide. When the compilers behave differently, the denotation 78K0 and 78K0S cores includes the 78K0 core without the `DIV/MUL` instructions, unless otherwise specified.

---

## Building applications—an overview

A typical application is built from several source files and libraries. The source files can be written in C, C++, or assembler language, and can be compiled into object files by the compiler or the assembler.

A library is a collection of object files that are added at link time only if they are needed. A typical example of a library is the compiler library containing the runtime environment and the C/C++ standard library. Libraries can also be built using the IAR XAR Library Builder, the IAR XLIB Librarian, or be provided by external suppliers.

The IAR XLINK Linker is used for building the final application. XLINK normally uses a linker command file, which describes the available resources of the target system.



Below, the process for building an application on the command line is described. For information about how to build an application using the IDE, see the *IAR Embedded Workbench® IDE User Guide*.

## COMPILING

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

The 78K0/78K0S compiler: `icc78k myfile.c`

The 78K0R compiler: `icc78k0r myfile.c`

You must also specify some critical options, see *Basic settings for project configuration*, page 6.

## LINKING

The IAR XLINK Linker is used for building the final application. Normally, XLINK requires the following information as input:

- Several object files and possibly certain libraries
- The standard library containing the runtime environment and the standard language functions
- A program start label
- A linker command file that describes the placement of code and data into the memory of the target system
- Information about the output format.

On the command line, the following line can be used for starting XLINK:

```
xlink myfile.r26 myfile2.r26 -s __program_start -f lnk.xcl
c178ks1.r26 -o aout.hex -FIntel-extended
```

In this example, `myfile.r26` and `myfile2.r26` are object files, `lnk.xcl` is the linker command file, and `c178ks1.r26` is the runtime library. The option `-s` specifies the label where the application starts. The option `-o` specifies the name of the output file, and the option `-r` is used for specifying the output format UBROF, which can be used for debugging in C-SPY®.

The IAR XLINK Linker produces output according to your specifications. Choose the output format that suits your purpose. You might want to load the output to a debugger—which means that you need output with debug information. Alternatively, you might want to load the output to a flash loader or a PROM programmer—in which case you need output without debug information, such as Intel-hex or Motorola S-records. The option `-F` can be used for specifying the output format. (The default output format is `Intel-extended`.)

## Basic settings for project configuration

This section gives an overview of the basic settings for the project setup that are needed to make the compiler generate the best code for the 78K device you are using. You can specify the options either from the command line interface or in the IDE.

The basic settings are:

- Processor core
- Code model
- Data model (78K0R only)
- Optimization settings
- Runtime environment.

In addition to these settings, many other options and settings can fine-tune the result even further. For details about how to set options and for a list of all available options, see the chapters *Compiler options* and the *IAR Embedded Workbench® IDE User Guide*, respectively.

### PROCESSOR CORE

The IAR C/C++ Compilers for 78K support these processor cores of the 78K series:

Processor core	Description
78K0_basic	78K0 core without the DIV/MUL instructions
78K0 (default)	78K0 core with the DIV/MUL instructions
78K0R	78K0R core
78K0S	78K0S core

Table 3: Processor cores



See the Embedded Workbench® IDE online help system for information about specifying the processor core in the IAR Embedded Workbench IDE.



Use the `--core=[78k0_basic|78k0|78k0r|78k0s]` option to specify the processor core for your project; see `--core`, page 156.

## CODE MODEL

The IAR C/C++ Compilers for 78K use two *code models* each, that you can set on file- or function-level to control which function calls are generated by default. Two of these can be used with the 78K0/78K0S Compiler and two with the 78K0R Compiler. The code models are:

### 78K0/78K0S

- The *standard* code model has an upper limit of 64 Kbytes
- The *banked* code model can use the banking mechanism and access a 16-Mbyte address space.

### 78K0R

- The *near* code model has an upper limit of 64 Kbytes
- The *far* code model can access the entire 1 Mbyte memory.

For detailed information about the code models, see the *Functions* chapter.

## DATA MODEL (78K0R ONLY)

The 78K0R IAR C/C++ Compiler use two *data models*, to make it easier to write code for applications with different data requirements. The data models are:

- The *near* data model can access data in the highest 64 Kbytes of data memory
- The *far* data model can address data in the entire 1 Mbytes of data memory.

For detailed information about the data models, see the *Data storage* chapter.

**Note:** The 78K0/78K0S Compiler does *not* use data models.

## OPTIMIZATION FOR SPEED AND SIZE

The compiler is a state-of-the-art compiler with an optimizer that performs, among other things, dead-code elimination, constant propagation, inlining, common sub-expression elimination, and precision reduction. It also performs loop optimizations, such as unrolling and induction variable elimination.

You can decide between several optimization levels and for the highest level you can choose between different optimization goals—*size*, *speed*, or *balanced*. Most optimizations will make the application both smaller and faster. However, when this is not the case, the compiler uses the selected optimization goal to decide how to perform the optimization.

The optimization level and goal can be specified for the entire application, for individual files, and for individual functions. In addition, some individual optimizations, such as function inlining, can be disabled.

For details about compiler optimizations and for more information about efficient coding techniques, see the chapter *Efficient coding for embedded applications*.

## RUNTIME ENVIRONMENT

To create the required runtime environment you should choose a runtime library and set library options. You might also need to override certain library modules with your own customized versions.

Two different sets of runtime libraries are provided:

- The IAR DLIB Library, which supports ISO/ANSI C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, et cetera.
- The IAR CLIB Library is a light-weight library, which is not fully compliant with ISO/ANSI C. Neither does it fully support floating-point numbers in IEEE 754 format or does it support Embedded C++. (This library is used by default for C projects).

The runtime library you choose can be one of the prebuilt libraries, or a library that you customized and built yourself. The IDE provides a library project template for both libraries, that you can use for building your own library version. This gives you full control of the runtime environment. If your project only contains assembler source code, you do not need to choose a runtime library.

For detailed information about the runtime environments, see the chapters *The DLIB runtime environment* and *The CLIB runtime environment*, respectively.

The way you set up a runtime environment and locate all the related files differs depending on which build interface you are using—the IDE or the command line.



### Choosing a runtime library in the IDE

To choose a library, choose **Project>Options**, and click the **Library Configuration** tab in the **General Options** category. Choose the appropriate library from the **Library** drop-down menu.

Note that for the DLIB library there are two different configurations—Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, et cetera. See *Library configurations*, page 55, for more information.

Based on which library configuration you choose and your other project settings, the correct library file is used automatically. For the device-specific include files, a correct include path is set up.



### Choosing runtime environment from the command line

Use the following command line options to specify the library and the dependency files:

Command line	Description
<code>-I 78k\inc</code>	Specifies the include path to device-specific I/O definition files.
<code>-I 78k\inc\{clib dlib}</code>	Specifies the library-specific include path. Use <code>clib</code> or <code>dlib</code> depending on which library you are using.
<code>libraryfile.r26</code>	Specifies the library object file
<code>--clib</code>	Uses CLIB-specific floating-point types (for the IAR CLIB Library only)
<code>--dlib_config</code> <code>C:\...\configfile.h</code>	Specifies the library configuration file (for the IAR DLIB Library only)

Table 4: Command line options for specifying library and dependency files

For a list of all prebuilt library object files for the IAR DLIB Library, see Table 17, *Prebuilt libraries*, page 57. The table also shows how the object files correspond to the dependent project options, and the corresponding configuration files. Make sure to use the object file that matches your other project options.

For a list of all prebuilt object files for the IAR CLIB Library, see Table 28, *Runtime libraries*, page 84. The table also shows how the object files correspond to the dependent project options. Make sure to use the object file that matches your other project options.

### Setting library and runtime environment options

You can set certain options to reduce the library and runtime environment size:

- The formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 59 (DLIB) and *Input and output*, page 85 (CLIB).
- The size of the stack and the heap, see *The stack*, page 46, and *The heap*, page 47, respectively.

---

## Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the 78K microcontroller.

## EXTENDED KEYWORDS

The compiler provides a set of keywords that can be used for configuring how the code is generated. For example, there are keywords for controlling the memory type for individual variables as well as for declaring special function types.



By default, language extensions are enabled in the IDE.

The command line option `-e` makes the extended keywords available, and reserves them so that they cannot be used as variable names. See, `-e`, page 164 for additional information.

For detailed descriptions of the extended keywords, see the chapter *Extended keywords*.

## PRAGMA DIRECTIVES

The pragma directives control the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it issues warning messages.

The pragma directives are always enabled in the compiler. They are consistent with ISO/ANSI C, and are very useful when you want to make sure that the source code is portable.

For detailed descriptions of the pragma directives, see the chapter *Pragma directives*.

## PREDEFINED SYMBOLS

With the predefined preprocessor symbols, you can inspect your compile-time environment, for example the code model and the processor type.

For detailed descriptions of the predefined symbols, see the chapter *The preprocessor*.

## SPECIAL FUNCTION TYPES

The special hardware features of the 78K microcontroller are supported by the compiler's special function types: interrupt and monitor. You can write a complete application without having to write any of these functions in assembler language.

For detailed information, see *Primitives for interrupts, concurrency, and OS-related programming*, page 23.

## ACCESSING LOW-LEVEL FEATURES

For hardware-related parts of your application, accessing low-level features is essential. The compiler supports several ways of doing this: intrinsic functions, mixing C and assembler modules, and inline assembler. For information about the different methods, see *Mixing C and assembler*, page 91.

# Data storage

This chapter gives a brief introduction to the memory layout of the 78K microcontroller and the fundamental ways data can be stored in memory: on the stack, in static (global) memory, or in heap memory. For efficient memory usage, the compilers provide data models and data memory attributes, allowing you to fine-tune the access methods, resulting in smaller code size. The concepts of data models and memory types are described in relation to pointers, structures, Embedded C++ class objects, and non-initialized memory. Finally, detailed information about data storage on the stack and the heap is provided.

---

## Introduction

The 78K microcontroller family can be divided into two main groups, the 78K0 and 78K0S cores and the 78K0R core. The 78K0R core has larger natural addressing capabilities than the other cores.

The 78K0 and 78K0S cores have 64 Kbytes of continuous memory, ranging from 0x0000 to 0xFFFF. The 78K0R core has 1 Mbyte of memory, ranging from 0x00000 to 0xFFFFF.

Different types of physical memory can be placed in the memory range. A typical application will have both read-only memory (ROM) and read-write memory (RAM). In addition, some parts of the memory range contain processor control registers and peripheral units.

All 78K microcontrollers can access memory in different ways. The access methods range from a generic but expensive method that can access the full memory space, to a cheap method that can access the short address memory area.

## DIFFERENT WAYS TO STORE DATA

In a typical application, data can be stored in memory in three different ways:

- Auto variables.  
All variables that are local to a function, except those declared static, are stored on the stack. These variables can be used as long as the function executes. When the function returns to its caller, the memory space is no longer valid.
- Global variables and local variables declared `static`.

In this case, the memory is allocated once and for all. The word *static* in this context means that the amount of memory allocated for this kind of variables does not change while the application is running. For more information, see *Data models (78K0R only)*, page 12 and *Memory types*, page 13.

- Dynamically allocated data.

An application can allocate data on the *heap*, where the data it remains valid until it is explicitly released back to the system by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that there are potential risks connected with using dynamically allocated data in systems with a limited amount of memory, or systems that are expected to run for a long time. For more information, see *Dynamic memory on the heap*, page 20.

---

## Data models (78K0R only)

The IAR C/C++ Compiler for 78K0R supports the concept of data models, to make it easier to write code for applications with different data requirements.

Technically, the data model specifies the default memory type. This means that the data model controls the following:

- The default placement of static and global variables, and constant literals
- Dynamically allocated data, for example data allocated with `malloc`, or, in C++, the operator `new`
- The default pointer type.

The data model only specifies the default memory type. It is possible to override this for individual variables and pointers. For information about how to specify a memory type for individual objects, see *Using data memory attributes*, page 14.

### SPECIFYING A DATA MODEL

Two data models are implemented: Near and Far. These models are controlled by the `--data_model` option. Each model has a default memory type and a default pointer size. If you do not specify a data model option, the compiler will use the Near data model.

Your project can only use one data model at a time, and the same model must be used by all user modules and all library modules. However, you can override the default memory type for individual data objects and pointers by explicitly specifying a memory attribute, see *Using data memory attributes*, page 14.

This table summarizes the different data models:

Data model name	Default memory and pointer attribute	Placement of data
Near (default)	<code>--near</code>	The highest 64 Kbytes
Far	<code>--far</code>	The entire 1 Mbyte of memory

Table 5: Data model characteristics



See the *IAR Embedded Workbench® IDE User Guide* for information about setting options in the IDE.



Use the `--data_model` option to specify the data model for your project; see `--data_model`, page 157.

### The Near data model

The Near data model places data in the highest 64 Kbytes of memory. This memory can be accessed using 2-byte pointers, which means that only 16 bits are needed for pointer storage. The default pointer type passed as a parameter will use one 16-bit register or 2 bytes on the stack.

### The Far data model

The Far data model places data in the first 1 Mbyte of memory. This is the only memory that can be accessed using 3-byte pointers. The default pointer type passed as a parameter will use 4 bytes on the stack.

## Memory types

This section describes the concept of *memory types* used for accessing data by the compiler. It also discusses pointers in the presence of multiple memory types. For each memory type, the capabilities and limitations are discussed.

The compiler uses different memory types to access data that is placed in different areas of the memory. There are different methods for reaching memory areas, and they have different costs when it comes to code space, execution speed, and register usage. The access methods range from generic but expensive methods that can access the full memory space, to cheap methods that can access limited memory areas. Each memory type corresponds to one memory access method. If you map different memories—or part of memories—to memory types, the compiler can generate code that can access data efficiently.

For example, the memory accessible using the near memory access method is called memory of near type, or simply near memory.

In each data model, there is a default memory type that your application will use. However, it is possible to specify—for individual variables or pointers—a different memory type. This makes it possible to create an application that can contain a large amount of data, and at the same time make sure that variables that are used often are placed in memory that can be efficiently accessed.

### SHORT ADDRESS AREA

The short addressable area—saddr memory—consists of 256 bytes of data memory in the address range `0xFE20–0xFF1F` (78K0 and 78K0S) or `0xFFE20–0xFFFF1F` (78K0R).

A saddr object can only be placed in saddr memory. By using objects of this type, the code generated by the compiler to access them is minimized. This means a smaller footprint for the application, and faster execution at run-time.

### NEAR

The near memory consists of 64 Kbytes of data memory. In hexadecimal notation, this is the address range `0x0000–0xFFFF` (78K0 and 78K0S) or `0xF0000–0xFFFFF` (78K0R).

A near object can only be placed in near memory.

### FAR (78K0R ONLY)

Only the 78K0R core has far memory, consisting of the whole 1 Mbytes of data memory. In hexadecimal notation, this is the address range `0x00000–0xFFFFF`.

A far object can only be placed in far memory, and the size of such an object is limited to 64 Kbytes.

### SFR

The special function register (SFR) area is located in the address range `0xFF00–0xFFFF` (78K0 and 78K0S) or `0xFFFF00–0xFFFFF` (78K0R).

Special function registers are defined in the device-specific header files.

### USING DATA MEMORY ATTRIBUTES

The compilers provide *extended keywords* which can be used as *data memory attributes*. These keywords let you override the default memory type for individual data objects and pointers, which means that you can place data objects in other memory areas than the default memory. This also means that you can fine-tune the access method for each individual data object, which results in smaller code size.

This table summarizes the available memory types and their corresponding keywords:

Memory type	Keyword	Address range	Pointer size
Saddr	<code>__saddr</code>	0xFE20-0xFF1F (78K0 and 78K0S) 0xFFE20-0xFFFF1F (78K0R)	16 bits
Near (default)	<code>__near</code>	0x0000-0xFFFF (78K0 and 78K0S) 0xF0000-0xFFFFF (78K0R)	16 bits
Far	<code>__far</code>	0x00000-0xFFFFF (78K0R only)	24 bits
SFR	<code>__sfr</code>	0xFF00-0xFFFF (78K0 and 78K0S) 0xFFFF00-0xFFFFF (78K0R)	n/a

Table 6: Memory types and their corresponding keywords

The keywords are only available if language extensions are enabled in the compiler.



In the IDE, language extensions are enabled by default.



Use the `-e` compiler option to enable language extensions. See *-e*, page 164 for additional information.

For reference information about each keyword, see *Descriptions of extended keywords*, page 208.

## Syntax

The keywords follow the same syntax as the type qualifiers `const` and `volatile`. The memory attributes are *type attributes* and therefore they must be specified both when variables are defined and in the declaration, see *General syntax rules for extended keywords*, page 203.

The following declarations place the variable `i` and `j` in `saddr` memory. The variables `k` and `l` will also be placed in `saddr` memory. The position of the keyword does not have any effect in this case:

```
__saddr int i, j;
int __saddr k, l;
```

Note that the keyword affects both identifiers. If no memory type is specified, the default memory type is used.

The `#pragma type_attribute` directive can also be used for specifying the memory attributes. The advantage of using `pragma` directives for specifying keywords is that it offers you a method to make sure that the source code is portable. Refer to the chapter *Pragma directives* for details about how to use the extended keywords together with `pragma` directives.

## Type definitions

Storage can also be specified using type definitions. These two declarations are equivalent:

```
/* Defines via a typedef */
typedef char __near Byte;
typedef Byte *BytePtr;
Byte AByte;
BytePtr ABytePointer;

/* Defines directly */
__near char AByte;
char __near *ABytePointer;
```

## POINTERS AND MEMORY TYPES

Pointers are used for referring to the location of data. In general, a pointer has a type. For example, a pointer that has the type `int *` points to an integer.

In the compiler, a pointer also points to some type of memory. There are two types of pointers—near and far—that point to data objects located in near and far memory, respectively. However, the far pointer (and far memory) is only available for the 78K0R Compiler. Because `saddr` memory is a part of near memory, near pointers can also manage data objects located in `saddr` memory.

**Note:** Because there is only one pointer type in the 78K0/78K0S Compiler, there is no practical use for explicitly declaring a pointer as near by using the `__near` keyword when writing for the 78K0/78K0S Compiler.

## STRUCTURES AND MEMORY TYPES

For structures, the entire object is placed in the same memory type. It is not possible to place individual structure members in different memory types.

In the example below, the variable `Gamma` is a structure placed in `saddr` memory.

```
struct MyStruct
{
    int mAlpha;
    int mBeta;
};

__saddr struct MyStruct Gamma;
```

This declaration is incorrect:

```
struct MyStruct
{
    int mAlpha;
    __saddr int mBeta; /* Incorrect */
};
```

## MORE EXAMPLES

The following is a series of examples with descriptions. First, some integer variables are defined and then pointer variables are introduced. Finally, a function accepting a pointer to an integer in near memory is declared. The function returns a pointer to an integer in near memory. It makes no difference whether the memory attribute is placed before or after the data type. To read the following examples, start from the left and add one qualifier at each step

<code>int MyA;</code>	A variable defined in default memory.
<code>int __near MyB;</code>	A variable in near memory.
<code>__near int MyC;</code>	A variable in near memory.
<code>int * MyD;</code>	A pointer stored in default memory. The pointer points to an integer in default memory.
<code>int __near * MyE;</code>	A pointer stored in default memory. The pointer points to an integer in near memory.
<code>int __near * __saddr MyF;</code>	A pointer stored in saddr memory pointing to an integer stored in near memory.
<code>int __near * MyFunction(     int __near *);</code>	A declaration of a function that takes a parameter which is a pointer to an integer stored in near memory. The function returns a pointer to an integer stored in near memory.

---

## C++ and memory types

A C++ class object is placed in one memory type, in the same way as for normal C structures. However, the class members that are considered to be part of the object are the non-static member variables. The static member variables can be placed individually in any kind of memory.

Remember, in C++ there is only one instance of each static member variable, regardless of the number of class objects.

All restrictions that apply to the default pointer type also apply to the `this` pointer. This means that it must be possible to convert a pointer to the object to the default pointer type. Also note that for non-static member functions—unless class memory is used, see *Classes*, page 111—the `this` pointer will be of the default data pointer type.

In the Near data model, this means that objects of classes with a member function can only be placed in the default memory type (`__near`).

### Example

In the example below, an object, named `delta`, of the type `MyClass` is defined in near memory. The class contains a static member variable that is stored in `saddr` memory.

```
// A class definition (may be placed in a header file)
class MyClass
{
public:
    int mAlpha;
    int mBeta;

    __saddr static int mGamma;
};

// Needed definitions (should be placed in a source file)
__saddr int MyClass::mGamma;

// An object of class type MyClass
__near MyClass Delta;
```

---

## Auto variables—on the stack

Variables that are defined inside a function—and not declared static—are named *auto variables* by the C standard. A few of these variables are placed in processor registers; the rest are placed on the stack. From a semantic point of view, this is equivalent. The main differences are that accessing registers is faster, and that less memory is required compared to when variables are located on the stack.

Auto variables can only live as long as the function executes; when the function returns, the memory allocated on the stack is released.

### THE STACK

The stack can contain:

- Local variables and parameters not stored in registers

- Temporary results of expressions
- The return value of a function (unless it is passed in registers)
- Processor state during interrupts
- Processor registers that should be restored before the function returns (callee-save registers).

The stack is a fixed block of memory, divided into two parts. The first part contains allocated memory used by the function that called the current function, and the function that called it, etc. The second part contains free memory that can be allocated. The borderline between the two areas is called the *top of stack* and is represented by the stack pointer, which is a dedicated processor register. Memory is allocated on the stack by moving the stack pointer.

A function should never refer to the memory in the area of the stack that contains free memory. The reason is that if an interrupt occurs, the called interrupt function can allocate, modify, and—of course—deallocate memory on the stack.

### Advantages

The main advantage of the stack is that functions in different parts of the program can use the same memory space to store their data. Unlike a heap, a stack will never become fragmented or suffer from memory leaks.

It is possible for a function to call itself—a *recursive function*—and each invocation can store its own data on the stack.

### Potential problems

The way the stack works makes it impossible to store data that is supposed to live after the function returns. The following function demonstrates a common programming mistake. It returns a pointer to the variable `x`, a variable that ceases to exist when the function returns.

```
int *MyFunction()
{
    int x;
    /* Do something here. */
    return &x; /* Incorrect */
}
```

Another problem is the risk of running out of stack. This will happen when one function calls another, which in turn calls a third, etc., and the sum of the stack usage of each function is larger than the size of the stack. The risk is higher if large data objects are stored on the stack, or when recursive functions—functions that call themselves either directly or indirectly—are used.

---

## Dynamic memory on the heap

Memory for objects allocated on the heap will live until the objects are explicitly released. This type of memory storage is very useful for applications where the amount of data is not known until runtime.

In C, memory is allocated using the standard library function `malloc`, or one of the related functions `calloc` and `realloc`. The memory is released again using `free`.

In C++, a special keyword, `new`, allocates memory and runs constructors. Memory allocated with `new` must be released using the keyword `delete`.

The 78K0R Compiler supports heaps in both near memory and far memory. For more information about this, see *The heap*, page 64.

### Potential problems

Applications that are using heap-allocated objects must be designed very carefully, because it is easy to end up in a situation where it is not possible to allocate objects on the heap.

The heap can become exhausted if your application uses too much memory. It can also become full if memory that no longer is in use was not released.

For each allocated memory block, a few bytes of data for administrative purposes is required. For applications that allocate a large number of small blocks, this administrative overhead can be substantial.

There is also the matter of *fragmentation*; this means a heap where small sections of free memory is separated by memory used by allocated objects. It is not possible to allocate a new object if no piece of free memory is large enough for the object, even though the sum of the sizes of the free memory exceeds the size of the object.

Unfortunately, fragmentation tends to increase as memory is allocated and released. For this reason, applications that are designed to run for a long time should try to avoid using memory allocated on the heap.

# Functions

This chapter contains information about functions. It gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.

---

## Function-related extensions

In addition to the ISO/ANSI C standard, the compilers provide several extensions for writing functions in C. Using these, you can:

- Control the storage of functions in memory
- Use primitives for interrupts, concurrency, and OS-related programming
- Facilitate function optimization
- Access hardware features.

The compilers use compiler options, extended keywords, pragma directives, and intrinsic functions to support this.

For more information about optimizations, see *Writing efficient code*, page 135. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

---

## Code models and memory attributes for function storage

By means of *code models*, the compilers control how code is generated for an application.

The compilers support two code models each. If you do not specify a code model, the compiler will use the one of the code models as default. Your project can only use one code model at a time, and the same model must be used by all user modules and all library modules. All code models produce code that is ROMable.

These code models are available:

Code model name	Description
Standard (default)	Non-banked function calls. 78K0/78K0S only.
Banked	Banked function calls. 78K0/78K0S only.
Near (default)	Function calls reach the first 64 Kbytes of memory. 78K0R only.

Table 7: Code models

Code model name	Description
Far	Function calls reach the entire 1 Mbyte memory. 78K0R only.

Table 7: Code models

For further information about the banked code model, see *Banked functions (78K0/78K0S only)*, page 27.



See the *IAR Embedded Workbench® IDE User Guide* for information about specifying a code model in the IDE.



Use the `--code_model` option to specify the code model for your project; see *--code\_model*, page 155.

## USING FUNCTION MEMORY ATTRIBUTES

It is possible to override the default placement for individual functions. Use the appropriate *function memory attribute* to specify this. These attributes are available:

Function memory attribute	Address range	Pointer size	Description
<code>__callt</code>	0–0x0FFFF	2 bytes	The function can be called from any segment.
<code>__callf</code>	0x0800–0x0FFF	2 bytes	The function can be called from any segment. 78K0 only.
<code>__non_banked</code>	0–0xFFFF	2 bytes	The function can be called from any segment. Default in the standard code model. 78K0/78K0S only.
<code>__banked</code>	0–0xFFFFF	3 bytes	The function can be called from any segment. Default in the banked code model. 78K0/78K0S only.
<code>__near_func</code>	0–0xFFFF	2 bytes	The function can be called from any segment. This is the default attribute in the 78K0R Compiler. 78K0R only.
<code>__far_func</code>	0–0xFFFFF	3 bytes	The function can be called from any segment. 78K0R only.

Table 8: Function memory attributes

Pointers with function memory attributes have restrictions in implicit and explicit casts between pointers and between pointers and integer types. For details about the restrictions, see *Casting*, page 186.

For detailed syntax information and for detailed information about each attribute, see the chapter *Extended keywords*.

## Primitives for interrupts, concurrency, and OS-related programming

The IAR C/C++ Compilers for 78K provide the following primitives related to writing interrupt functions, concurrent functions, and OS-related functions:

- The extended keywords `__interrupt` and `__monitor`
- The pragma directive `#pragma vector`
- The intrinsic functions `__enable_interrupt`, `__disable_interrupt`, `__get_interrupt_state`, and `__set_interrupt_state`.

### INTERRUPT FUNCTIONS

In embedded systems, using interrupts is a method for handling external events immediately; for example, detecting that a button was pressed.

In general, when an interrupt occurs in the code, the microcontroller simply stops executing the code it runs, and starts executing an interrupt routine instead. It is extremely important that the environment of the interrupted function is restored after the interrupt is handled; this includes the values of processor registers and the processor status register. This makes it possible to continue the execution of the original code after the code that handled the interrupt was executed.

The 78K microcontroller supports many interrupt sources. For each interrupt source, an interrupt routine can be written. Each interrupt routine is associated with a vector number, which is specified in the 78K microcontroller documentation from the chip manufacturer. The interrupt vector is the offset into the interrupt vector table. If you want to handle several different interrupts using the same interrupt function, you can specify several interrupt vectors. For the 78K microcontroller, the interrupt vector table always starts at the address `0x0`.

The header file `iodevice.h`, where `device` corresponds to the selected device, contains predefined names for the existing exception vectors.

For 78K0 and 78K0R, the `#pragma bank=n` directive (where `n` can be 0–3) can be used for context switching (using another register bank) when executing the interrupt function. Note that this is not possible for 78K0S.

To define an interrupt function, the `__interrupt` keyword and the `#pragma vector` directive can be used. For example:

```
#pragma bank = 2
#pragma vector = 0x14
__interrupt void MyInterruptRoutine(void)
{
    /* Do something */
}
```

**Note:** An interrupt function must have the return type `void`, and it cannot specify any parameters.

If a vector is specified in the definition of an interrupt function, the processor interrupt vector table is populated. It is also possible to define an interrupt function without a vector. This is useful if an application is capable of populating or changing the interrupt vector table at runtime. See the chip manufacturer’s 78K microcontroller documentation for more information about the interrupt vector table.

## MONITOR FUNCTIONS

A monitor function causes interrupts to be disabled during execution of the function. At function entry, the status register is saved and interrupts are disabled. At function exit, the original status register is restored, and thereby the interrupt status that existed before the function call is also restored.

To define a monitor function, you can use the `__monitor` keyword. For reference information, see *\_\_monitor*, page 211.



Avoid using the `__monitor` keyword on large functions, since the interrupt will otherwise be turned off for too long.

## Example of implementing a semaphore in C

In the following example, a binary semaphore—that is, a mutex—is implemented using one static variable and two monitor functions. A monitor function works like a critical region, that is no interrupt can occur and the process itself cannot be swapped out. A semaphore can be locked by one process, and is used for preventing processes from simultaneously using resources that can only be used by one process at a time, for example a USART. The `__monitor` keyword assures that the lock operation is atomic; in other words it cannot be interrupted.

```
/* This is the lock-variable. When non-zero, someone owns it. */
static volatile unsigned int sTheLock = 0;

/* Function to test whether the lock is open, and if so take it.
 * Returns 1 on success and 0 on failure.
 */

__monitor int TryGetLock(void)
{
    if (sTheLock == 0)
    {
        /* Success, nobody has the lock. */

        sTheLock = 1;
    }
}
```

```

        return 1;
    }
    else
    {
        /* Failure, someone else has the lock. */

        return 0;
    }
}

/* Function to unlock the lock.
 * It is only callable by one that has the lock.
 */

__monitor void ReleaseLock(void)
{
    sTheLock = 0;
}

/* Function to take the lock. It will wait until it gets it. */

void GetLock(void)
{
    while (!TryGetLock())
    {
        /* Normally a sleep instruction is used here. */
    }
}

/* An example of using the semaphore. */

void MyProgram(void)
{
    GetLock();

    /* Do something here. */

    ReleaseLock();
}

```

### Example of implementing a semaphore in C++

In C++, it is common to implement small methods with the intention that they should be inlined. However, the compiler does not support inlining of functions and methods that are declared using the `__monitor` keyword.

In the following example in C++, an auto object is used for controlling the monitor block, which uses intrinsic functions instead of the `__monitor` keyword.

```
#include <intrinsics.h>

/* Class for controlling critical blocks. */
class Mutex
{
public:
    Mutex()
    {
        // Get hold of current interrupt state.
        mState = __get_interrupt_state();

        // Disable all interrupts.
        __disable_interrupt();
    }

    ~Mutex()
    {
        // Restore the interrupt state.
        __set_interrupt_state(mState);
    }

private:
    __istate_t mState;
};

class Tick
{
public:
    // Function to read the tick count safely.
    static long GetTick()
    {
        long t;

        // Enter a critical block.
        {
            Mutex m;

            // Get the tick count safely,
            t = smTickCount;
        }
    }
};
```

```

    }
    // and return it.
    return t;
}

private:
    static volatile long smTickCount;
};

volatile long Tick::smTickCount = 0;

extern void DoStuff();

void MyMain()
{
    static long nextStop = 100;

    if (Tick::GetTick() >= nextStop)
    {
        nextStop += 100;
        DoStuff();
    }
}

```

## C++ AND SPECIAL FUNCTION TYPES

C++ member functions can be declared using special function types. However, there is one restriction:

Interrupt member functions must be static. When a non-static member function is called, it must be applied to an object. When an interrupt occurs and the interrupt function is called, there is no object available to apply the member function to.

---

## Banked functions (78K0/78K0S only)

This section introduces the banking technique available for the 78K0/78K0S Compiler. It is important to know when to use it, what it does, and how it works. The section also describes how to write and partition source code and ISRs (interrupt service routines) for banked applications, including the associated compiler and linker command file configuration. Finally, the section also discusses linker output formats suitable for banked addresses, and methods of downloading to multiple PROMs.

**Note:** When you read this section, you should be familiar with the other concepts described in *Part I. Using the compilers* in this book.

## THE 78K0/78K0S BANKING SYSTEM

If you are using a 78K microcontroller with a natural address range of 64 Kbytes of memory, it has a 16-bit addressing capability. *Banking* is a technique for extending the amount of memory that can be accessed by the processor beyond the limit set by the size of the natural addressing scheme of the processor. In other words, more code can be accessed.

Banked memory is used in projects which require such a large amount of executable code that the natural 64 Kbytes address range of the 78K0/78K0S cores is not sufficient to contain it all.

**Note:** Because the 78K0R core has a natural address range of 1 Mbyte of memory and does not need to use banking, the 78K0R compiler has no support for the mechanism.

## USING THE BANKED CODE MODEL

When you create a banked application, you must be aware of the associated compiler and linker command file configuration.

**Note:** These options are not available for the 78K0R Compiler.

### Compiler options for banked mode

To compile your modules for banked mode, use the **Banked** code model option (`--code_model banked`). To bank an individual function when using the non-banked code model, declare the function with the `__banked` attribute.

### Linker options for banked mode

When linking an application, you must place your code segments into banks corresponding to the available physical banks in your hardware. However, the physical bank size and location is dependent upon the size of your root bank which in turn is dependent on your source code.

As a result, you may need to make a few trial passes through the linking process to determine the optimal memory usage.

**Example: banked placement using the -P command**

The `-P` linker command is useful for banked segment placement. For instance, if your banked code uses the ROM memory area `0x8000-0x9FFF`, the linker directives would look like this:

```
// First some defines for the banks
-D_CODEBANK_START=8000
-D_CODEBANK_END=9FFF
-D_CODEBANK_BANKS=4

-P(CODE) MYBANKEDCODESEGMENT=[_CODEBANK_START-_CODEBANK_END]*_CODE
BANK_BANKS+10000
```

This divides the segment into segment parts which are located at the addresses:

```
8000-9FFF //Bank number 0
18000-19FFF //Bank number 1
28000-29FFF //Bank number 2
38000-39FFF //Bank number 3
```

**The bank switching routine**

The bank switching routine is based on an SFR port being used for the bank switching. The SFR port is defined in the linker command file by the line:

```
-D_BANK_REG=PORTADDRESS
```

As long as you use this solution, the only thing you must do is to define the appropriate SFR port address.

**HOW BANKING WORKS**

The banked code model available for the 78K0/78K0S Compiler allows the code memory area to be extended with up to 256 banks of additional ROM memory. Your hardware must provide these additional physical memory banks, as well as the logic required to decode the high address bits which represent the *bank number*.

Because the 78K0 and 78K0S cores cannot handle more than 64 Kbytes of memory at any single time, the extended memory range introduced by banking implies that special care must be taken. Only one bank at a time is visible, and the remaining banks must be brought into the 64 Kbytes address range before they can be used.

This memory map shows an example of a banked code memory area:

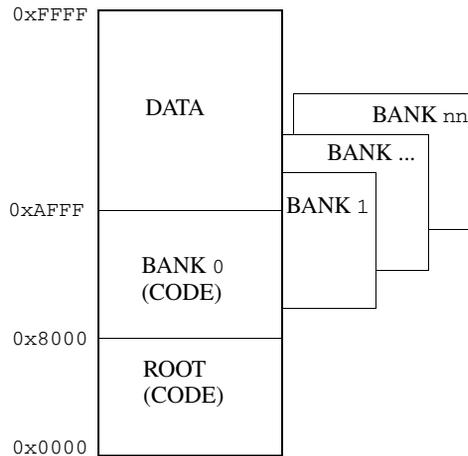


Figure 1: Banked code memory layout (example)

You can place the banked area anywhere in code memory, but there must always be a *root area* for holding the runtime environment code. It is practical to place the root area at address 0 and upwards.

To access code residing in one of the memory banks, the compiler keeps track of the bank number of a banked function by maintaining a 3-byte pointer to it, which has the following form:

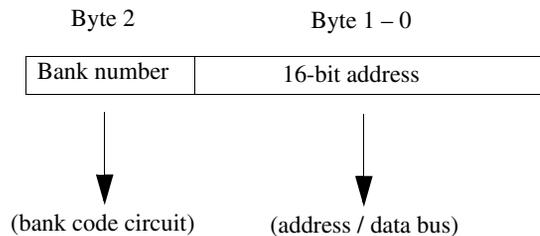


Figure 2: The 3-byte banked address

## Bank switching

For a bank switch, the following actions are performed:

- The current bank number is saved on the stack
- The new bank number—byte 2 of the 3-byte pointer—is written to the SFR port
- Your hardware decodes the bank number to select the appropriate ROM bank
- A call to the address located in byte 0 and 1 of the 3-byte pointer is performed.

For further details, see *The bank switching routine*, page 34.

## Code that cannot be banked

Code banking is achieved by dividing the address space used for program code into two parts: *non-banked* and *banked* code. In Figure 1, *Banked code memory layout (example)*, page 30, the part that contains the non-banked code is referred to as the *root bank*. There must always be a certain amount of code that is non-banked. For example, interrupt service routines must always be available, as interrupts can occur at any time.

The following selected parts must be located in non-banked memory:

- Interrupt and call table vectors
- Interrupt service routines
- Functions declared `__callf` and `__callt`
- The `CSTARTUP` routine
- Segments containing constants and initial values for initialized variables (the segments `CODE`, `CONST`, `DIFUNCT`, `FCODE`, `NEAR_ID`, `RCODE`, `SADDR_ID`, and `SWITCH`)
- Selected parts of the runtime library, such as routines for arithmetic support and `setjmp/longjmp`
- Bank-switching routines, that is, those routines that will perform the mapping from internal memory to banked memory.

Code located in non-banked memory will always be available to the processor, and will always be located at the same internal address.

**Note:** In addition, code compiled for the 78K0R core cannot be banked.

## WRITING SOURCE CODE FOR BANKED MEMORY

Writing code to be used in banked memory is not much different from writing code for standard memory, but there are a few issues to be aware of. These primarily concern partitioning your code into source modules so that they can be most efficiently placed into banks by the linker, and the distinction between banked versus non-banked code.

## C/C++ language considerations

From the C/C++ language standpoint, any arbitrary C/C++ program can be compiled for banked memory. The only restriction is the size of the function, as it cannot be larger than the size of a bank.

### Bank size and code size

Each banked C/C++ source function that you compile will generate a separate *segment part*, and all segment parts generated from banked functions will be located in the `BCODE` linker segment. The code contained in a segment part is an *indivisible unit*, as well as source files which contain several functions. The linker cannot break up a code contained in indivisible units and place part of it in one bank and part of it in another bank. Thus, the code produced from a banked function must always be smaller than the bank size.

This means that you have to consider the size of each C/C++ source file so that the generated code will fit into your banks. If any of your code segments is larger than the specified bank size, the linker will issue an error.

If you need to specify the location of any code individually, you can rename the code segment for each function to a distinct name that will allow you to refer to it individually. To assign a function to a specific segment, use the `@` operator, the `#pragma location` directive, or the `#pragma segment` directive.

For more information about segments, see the chapter *Placing code and data*.

### Banked versus non-banked function calls

Differentiating between the non-banked versus banked function calls for the 78K0/78K0S Compiler is important because non-banked function calls are faster and take up less code space than banked function calls. Therefore, it is useful to be familiar with which types of function declarations that result in non-banked function calls.

In this context, a *function call* is the sequence of machine instructions generated by the compiler whenever a function in your C/C++ source code calls another C/C++ function or library routine. This also includes saving the return address and then sending the new execution address to the hardware.

The 78K0/78K0S Compiler has two function call sequences:

- Non-banked function calls: The return address and new execution address are 16-bit (2 bytes) values. Non-banked function calls are default in the standard code model.
- Banked function calls: The return address and new execution address are 24-bit (3 bytes) values (default in the banked code model)

In the banked code model, banked function calls are generated by default. However, when it is known that both the calling and the called function reside in the same physical bank, it is not necessary to switch bank. This means, a more efficient banked function call, without actual bank switching, can be generated by the compiler. In the following situations, banked function calls without actual bank switching, is generated:

- Both the calling and the called function are defined in the same source file.
- Functions defined `static`. This is because the purpose of the `static` keyword is to prevent the function from being called from outside the module in which it is defined. The exception to this rule is the case in which the address of the function is known by a function in another module, which then calls the `static` function indirectly via that address.

Functions declared using the `__non_banked` memory attribute are located in the non-banked `ROOT` area, which means the functions can be called with non-banked function calls. For functions that are frequently called, it can be recommended to place them in the `ROOT` area.

### ***Example of a non-banked function call in the banked code model***

In this example we assume that `f1()` will call `f2()`. They are defined in separate source modules. Then the definition of `f2()` would look like:

```
__non_banked void f2(void) /* simple void function example */
{
    /* code here...*/
}
```

The function prototype for `f2()` in the module where `f1()` will call `f2()` would be:

```
extern __non_banked void f2(void);
```

The actual call to `f2()` from within `f1()` would be exactly as any other function, for example:

```
void f1(void)
{
    f2();
}
```

### **Interrupt, callf, and callt functions in banked mode**

The code for interrupt service routines, and functions declared using the extended keywords `__callf` and `__callt` are by default non-banked. This means that these functions are located in the `CODE` segment. The compiler handles this automatically.

## Calling banked functions from assembler language

In an assembler language program, calling a C/C++ function located in another bank requires using the same protocol as the compiler. For information about this protocol, see *Calling convention*, page 97. To generate an example of a banked function call, use the technique described in the section *Calling assembler routines from C*, page 94.

## THE BANK SWITCHING ROUTINE

A non-banked function call reserves the register HL if a function pointer is used. A banked call reserves the registers HL and E, which means that there is one 8-bit register fewer to be used as a register parameter. Thus, the cost for parameter passing for banked and non-banked functions cannot be compared in an easy way. However, the cost for different ways of calling a banked function can be compared.

There are two different ways to call a banked function. If the compiler knows that the called banked function is located in the same bank as the calling function, there is no need to switch banks and save the old bank, and as a result—provided some other conditions are also fulfilled—it uses the fast switching routine `?FAR_FAST_CALL_L07` and the corresponding `FAR_FAST_RETURN_L07`. These are faster than if you had to save/restore the old bank and switch to a new bank.

The fast version of the switching routine can only be used if both the calling and the called function are banked and located in the same file (which means that the compiler knows they are located in the same bank). If they are located in different files or if the calling function is non-banked, the compiler must preserve the current bank before switching to the bank of the called function.

This is what a standard bank switch looks like:

Standard bank switching code	Clock cycles
<code>?FAR_CALL_L07:</code>	
<code>xch a, d</code>	2
<code>mov a, BANK_REG</code>	5
<code>xch a, d</code>	2
<code>push de</code>	4
<code>xch a, e</code>	2
<code>mov BANK_REG, a</code>	5
<code>xch a, e</code>	2

Table 9: Standard bank switching cost in cycles

Standard bank switching code		Clock cycles
movw	de, #FAR_RETURN_L07	6
push	de	4
push	hl	4
ret		6
FAR_RETURN_L07:		
pop	hl	4
xch	a, h	2
mov	BANK_REG, a	5
xch	a, h	2
ret		6
Total		61

Table 9: Standard bank switching cost in cycles (Continued)

This is what a fast bank switch looks like:

Fast bank switching code		Clock cycles
?FAR_FAST_CALL_L07:		
xch	a, e	2
mov	d, a	2
xch	a, e	2
push	de	4
movw	de, #FAR_FAST_RETURN_L07	6
push	de	4
push	hl	4
ret		6
FAR_FAST_RETURN_L07:		
pop	hl	4
ret		6
Total		40

Table 10: Fast bank switching cost in cycles

This is the calling overhead for banked functions. It does not take into account the cost of initializing register and stack parameters.

### Modifying the default bank switching routine

The default bank switching code is available in the supplied assembler language source file `L07.s26`, which you can find in the directory `78k\src\lib`.

However, if you only use a small number of banks, you can use only certain bits of the port and use the rest of the port for something else. This is not explicitly supported by the compiler, but you can do it by modifying the `L07.s26` file.

After you have modified the file, reassemble it and replace the object module in the runtime library you are using. Simply include it in your application project and it will be used instead of the standard routine in the library.

If you want to use a completely different solution for bank switching, you must implement your own bank switching routine.

### DOWNLOADING TO MEMORY

There is a wide choice of devices and memories to use, and depending on what your banked mode system looks like, different actions might be needed. For instance, the memory might be a single memory circuit with more than 64 Kbytes capacity, or several smaller memory circuits. The memory circuits can be, for instance, EPROM or flash memory.



By default, the linker generates output in the Intel-extended format, but you can easily change this to use any available format required by the tools you are using. If you are using the IAR Embedded Workbench IDE, the default output format depends on the used build configuration—Release or Debug.

When you download the code into physical memory, special considerations might be needed. For instance, assume a 78K0 or 78K0S banked system with two 16-Kbyte banks of ROM starting at `0x8000`. If the banked code exceeds 16 Kbytes in size, when you link the project the result will be a single output file where the banked code starts at `0x8000` and crosses the upper bank limit. A modern EPROM programmer does not require downloading one file to one EPROM at a time, it handles the download automatically by splitting the file and downloading it. However, older types of programmers do not always support relocation, or are unable to ignore the high byte of the 3-byte address. This means that you have to edit the file manually to set the high bytes of each address to 0 so that the programmer can locate them properly.

## DEBUGGING BANKED APPLICATIONS



The C-SPY® Debugger supports banked mode debugging for the 78K0 and 78K0S cores. To set banked mode debugging options in the IAR Embedded Workbench IDE, choose **Project>Options**, select the **General Options** category, and click the **Target** tab. Type the appropriate values for the following options:

- **Reg. address** specifies the SFR address used as bank register
- **Bank Address** specifies the bank start address
- **No. of banks** specifies the number of banks
- **Bank size (Kbytes)** specifies the size of the banks.

### Banked mode debugging with other debuggers

If your emulator does not support banked mode, one common technique is to isolate a subset of your application's functions that interact (one bank full or less). Then compile, link, and debug these functions as a group using the 78K0/78K0S *standard* code model. Repeat this procedure for various groupings of functions. Then, when you actually test the final banked system on target hardware, many C/C++ programming-related issues will already have been resolved.



# Placing code and data

This chapter describes how the linker handles memory and introduces the concept of segments. It also describes how they correspond to the memory and function types, and how they interact with the runtime environment. The methods for placing segments in memory, which means customizing a linker command file, are described.

The intended readers of this chapter are the system designers that are responsible for mapping the segments of the application to appropriate memory areas of the hardware system.

---

## Segments and memory

In an embedded system, there are many different types of physical memory. Also, it is often critical *where* parts of your code and data are located in the physical memory. For this reason it is important that the development tools meet these requirements.

### WHAT IS A SEGMENT?

A *segment* is a logical entity containing a piece of data or code that should be mapped to a physical location in memory. Each segment consists of many *segment parts*. Normally, each function or variable with static storage duration is placed in a segment part. A segment part is the smallest linkable unit, which allows the linker to include only those units that are referred to. The segment could be placed either in RAM or in ROM. Segments that are placed in RAM do not have any content, they only occupy space.

The compiler has several predefined segments for different purposes. Each segment has a name that describes the contents of the segment, and a *segment memory type* that denotes the type of content. In addition to the predefined segments, you can define your own segments.

At compile time, the compiler assigns each segment its contents. The IAR XLINK Linker is responsible for placing the segments in the physical memory range, in accordance with the rules specified in the linker command file. Ready-made linker command files are provided, but, if necessary, they can be easily modified according to the requirements of your target system and application. It is important to remember that, from the linker's point of view, all segments are equal; they are simply named parts of memory.

For detailed information about individual segments, see the chapter *Segment reference*.

## Segment memory type

XLINK assigns a segment memory type to each of the segments. In some cases, the individual segments have the same name as the segment memory type they belong to, for example `CODE`. Make sure not to confuse the individual segment names with the segment memory types in those cases.

By default, the compiler uses these XLINK segment memory types:

Segment memory type	Description
CODE	For executable code
CONST	For data placed in ROM
DATA	For data placed in RAM

*Table 11: XLINK segment memory types*

XLINK supports several other segment memory types than the ones described above. However, they exist to support other types of microcontrollers.

For more details about segments, see the chapter *Segment reference*.

---

## Placing segments in memory

The placement of segments in memory is performed by the IAR XLINK Linker. It uses a linker command file that contains command line options which specify the locations where the segments can be placed, thereby assuring that your application fits on the target chip. To use the same source code with different derivatives, just rebuild the code with the appropriate linker command file.

In particular, the linker command file specifies:

- The placement of segments in memory
- The maximum stack size
- The maximum heap size.

This section describes the methods for placing the segments in memory, which means that you must customize the linker command file to suit the memory layout of your target system. For showing the methods, fictitious examples are used.

### CUSTOMIZING THE LINKER COMMAND FILE

The `config` directory contains ready-made linker command files for all supported devices (filename extension `.xcl`). The files contain the information required by the linker, and are ready to be used. The only change you will normally have to make to the supplied linker command file is to customize it so it fits the target system memory map.

If, for example, your application uses additional external RAM, you must add details about the external RAM memory area.

As an example, we can assume that the target system has this memory layout:

Range	Type
0x0000–0x7FFF	ROM
0x8000–0x9FFF	ROM (banked code)
0xA000–0xFFFF	RAM

Table 12: Memory layout of a target system (example)

The ROM can be used for storing `CONST` and `CODE` segment memory types. The RAM memory can contain segments of `DATA` type. The main purpose of customizing the linker command file is to verify that your application code and data do not cross the memory range boundaries, which would lead to application failure.

Remember not to change the original file. We recommend that you make a copy in the working directory, and modify the copy instead.

## The contents of the linker command file

Among other things, the linker command file contains three different types of XLINK command line options:

- The CPU used:  

```
-c78000
```

This specifies your target microcontroller.
- Definitions of constants used in the file. These are defined using the XLINK option `-D`.
- The placement directives (the largest part of the linker command file). Segments can be placed using the `-Z` and `-P` options. The former will place the segment parts in the order they are found, while the latter will try to rearrange them to make better use of the memory. The `-P` option is useful when the memory where the segment should be placed is not continuous.

In the linker command file, all numbers are specified in hexadecimal format. However, neither the prefix `0x` nor the suffix `h` is used.

**Note:** The supplied linker command file includes comments explaining the contents.

See the *IAR Linker and Library Tools Reference Guide* for more details.

## Using the `-Z` command for sequential placement

Use the `-Z` command when you must keep a segment in one consecutive chunk, when you must preserve the order of segment parts in a segment, or, more unlikely, when you

must put segments in a specific order. Typical segments that should be located in consecutive chunks are segments that hold the application code.

The following illustrates how to use the `-Z` command to place the segment `MYSEGMENTA` followed by the segment `MYSEGMENTB` in `CONST` memory (that is, ROM) in the memory range `0x0000-0x7FFF`.

```
-Z (CONST) MYSEGMENTA, MYSEGMENTB=0000-7FFF
```

To place two segments of different types consecutively in the same memory area, do not specify a range for the second segment. In the following example, the `MYSEGMENTA` segment is first located in memory. Then, the rest of the memory range could be used by `MYCODE`.

```
-Z (CONST) MYSEGMENTA=0000-7FFF
-Z (CODE) MYCODE
```

Two memory ranges can overlap. This allows segments with different placement requirements to share parts of the memory space; for example:

```
-Z (CONST) MYSMALLSEGMENT=6000-66FF
-Z (CONST) MYLARGESEGMENT=6000-6FFF
```



Even though it is not strictly required, make sure to always specify the end of each memory range. If you do this, the IAR XLINK Linker will alert you if your segments do not fit in the available memory.

### Using the `-P` command for packed placement

The `-P` command differs from `-Z` in that it does not necessarily place the segments (or segment parts) sequentially. With `-P` it is possible to put segment parts into holes left by earlier placements. Typical segments that are suitable for this are segments that do not need to be located in consecutive chunks, such as segments that hold constants, switch tables, et cetera.

The following example illustrates how the XLINK `-P` option can be used for making efficient use of the memory area. This command will place the data segment `MYDATA` in `DATA` memory (that is, in RAM) in a fictitious memory range:

```
-P (DATA) MYDATA=A000-AFFF, D000-DFFF
```

If your application has an additional RAM area in the memory range `0xE000-0xFFFF`, you can simply add that to the original definition:

```
-P (DATA) MYDATA=A000-AFFF, E000-FFFF
```

The linker can then place some parts of the `MYDATA` segment in the first range, and some parts in the second range. If you had used the `-Z` command instead, the linker would have to place all segment parts in the same range.

**Note:** Copy initialization segments—`BASENAME_I` and `BASENAME_ID`—must be placed using `-Z`.

### Using the `-P` command for banked placement (78K0/78K0S only)

For an example of using the `-P` command for banked segment placement, that is, code that should be divided into several different memory banks, see *Example: banked placement using the `-P` command*, page 29.

---

## Data segments

This section contains descriptions of the segments used for storing the different types of data: static, stack, heap, and located.

To get a clear understanding about how the data segments work, you must be familiar with the different memory types (and, for 78K0R, the different data models) available in the compilers. If you need to refresh these details, see the chapter *Data storage*.

### STATIC MEMORY SEGMENTS

Static memory is memory that contains variables that are global or declared static, as described in the chapter *Data storage*. Variables declared static can be divided into these categories:

- Variables that are initialized to a non-zero value
- Variables that are initialized to zero
- Variables that are located by use of the `@` operator or the `#pragma location` directive
- Variables that are declared as `const` and therefore can be stored in ROM
- Variables defined with the `__no_init` keyword, meaning that they should not be initialized at all.

For the static memory segments it is important to be familiar with:

- The segment naming
- How the memory types correspond to segment groups and the segments that are part of the segment groups
- Restrictions for segments holding initialized data
- The placement and size limitation of the segments of each group of static memory segments.

## Segment naming

The names of the segments consist of two parts—the segment group name and a *suffix*—for instance, `NEAR_Z`. There is a segment group for each memory type, where each segment in the group holds different categories of declared data. The names of the segment groups are derived from the memory type and the corresponding keyword, for example `NEAR` and `__near`. The following table summarizes the memory types and the corresponding segment groups:

Memory type	Segment group	Memory range
Near	NEAR	0x0000–0xFFFF (78K0 and 78K0S) 0xF0000–0xFFFFF (78K0R)
Far (78K0R only)	FAR	0x00000–0xFFFFF
Saddr	SADDR	0xFE20–0xFF1F (78K0 and 78K0S) 0xFFE20–0xFFFF1F (78K0R)

Table 13: Memory types with corresponding segment groups

Some of the declared data is placed in non-volatile memory, for example ROM, and some of the data is placed in RAM. For this reason, it is also important to know the XLINK segment memory type of each segment. For more details about segment memory types, see *Segment memory type*, page 40.

This table summarizes the different suffixes, which XLINK segment memory type they are, and which category of declared data they denote:

Categories of declared data	Suffix	Segment memory type
Non-initialized data	N	DATA
Zero-initialized data	Z	DATA
Non-zero initialized data	I	DATA
Initializers for the above	ID	CONST
Constants	C	CONST
Non-initialized absolute addressed data	A	

Table 14: Segment name suffixes

For a list of all supported segments, see *Summary of segments*, page 253.

### Examples

These examples demonstrate how declared data is assigned to specific segments:

<code>__near int j;</code>	The near variables that are to be initialized to zero
<code>__near int i = 0;</code>	when the system starts are placed in the segment
	<code>NEAR_Z</code> .
<code>__no_init __near int j;</code>	The near non-initialized variables are placed in the
	segment <code>NEAR_N</code> .
<code>__near int j = 4;</code>	The near non-zero initialized variables are placed in
	the segment <code>NEAR_I</code> in RAM, and the corresponding
	initializer data in the segment <code>NEAR_ID</code> in ROM.

### Initialized data

When an application is started, the system startup code initializes static and global variables in these steps:

- 1 It clears the memory of the variables that should be initialized to zero.
- 2 It initializes the non-zero variables by copying a block of ROM to the location of the variables in RAM. This means that the data in the ROM segment with the suffix `ID` is copied to the corresponding `I` segment.

This works when both segments are placed in continuous memory. However, if one of the segments is divided into smaller pieces, it is important that:

- The other segment is divided in exactly the same way
- It is legal to read and write the memory that represents the gaps in the sequence.

For example, if the segments are assigned these ranges, the copy will fail:

<code>NEAR_I</code>	<code>0x1000-0x10FF and 0x1200-0x12FF</code>
<code>NEAR_ID</code>	<code>0x4000-0x41FF</code>

However, in the following example, the linker will place the content of the segments in identical order, which means that the copy will work appropriately:

<code>NEAR_I</code>	<code>0x1000-0x10FF and 0x1200-0x12FF</code>
<code>NEAR_ID</code>	<code>0x4000-0x40FF and 0x4200-0x42FF</code>

The `ID` segment can, for all segment groups, be placed anywhere in memory, because it is not accessed using the corresponding access method. Note that the gap between the ranges will also be copied.

- 3 Finally, global C++ objects are constructed, if any.

### Data segments for static memory in the default linker command file

The default linker command file for the 78K0 and 78K0S cores contains these directives to place the static data segments:

```
//First, the segments to be placed in ROM are defined:
-Z (CODE) NEAR_ID, SADDR_ID=1000-7FFF
-P (CODE) CONST, SWITCH=0080-7FFF

//Then, the RAM data segments are placed in memory:
-Z (DATA) NEAR_I, NEAR_Z=FB00-FE1F
-Z (DATA) WRKSEG, SADDR_I, SADDR_Z, SADDR_N=FE20-FEDF
-P (DATA) NEAR_N=FB00-FE1F
```

All the data segments are placed in the area used by on-chip RAM.

## THE STACK

The stack is used by functions to store variables and other information that is used locally by functions, as described in the chapter *Data storage*. It is a continuous block of memory pointed to by the processor stack pointer register *SP*.

The data segment used for holding the stack is called *CSTACK*. The system startup code initializes the stack pointer to the end of the stack segment.

Allocating a memory area for the stack is done differently using the command line interface as compared to when using the IDE.



### Stack size allocation in the IDE

Choose **Project>Options**. In the **General Options** category, click the **Stack/Heap** tab.

Add the required stack size in the **Stack size** text box.



### Stack size allocation from the command line

The size of the *CSTACK* segment is defined in the linker command file.

The default linker file sets up a constant representing the size of the stack, at the beginning of the linker file:

```
-D_CSTACK_SIZE=size
```

**Note:** Normally, this line is prefixed with the comment character *//*. To make the directive take effect, remove the comment character.

Specify an appropriate size for your application. Note that the size is written hexadecimally without the *0x* notation.



## Placement of stack segment

Further down in the linker file, the actual stack segment is defined in the memory area available for the stack:

```
-Z (DATA) CSTACK+_CSTACK_SIZE=FB00-FE1F
```

**Note:** This range does not specify the size of the stack; it specifies the range of the available memory



## Stack size considerations

The compiler uses the internal data stack, `CSTACK`, for a variety of user program operations, and the required stack size depends heavily on the details of these operations. If the given stack size is too large, RAM is wasted. If the given stack size is too small, two things can happen, depending on where in memory you located your stack. Both alternatives are likely to result in application failure. Either program variables will be overwritten, leading to undefined behavior, or the stack will fall outside of the memory area, leading to an abnormal termination of your application.

**Note:** The maximum size of the internal data stack is equal to the size of available near memory, in other words up to 64 Kbytes, irrespective of which code model that you are using.

## THE HEAP

The heap contains dynamic data allocated by the C function `malloc` (or one of its relatives) or the C++ operator `new`.

If your application uses dynamic memory allocation, you should be familiar with:

- Linker segments used for the heap (which differs between the DLIB and the CLIB runtime environment for the 78K0R Compiler)
- Allocating the heap size, which differs depending on which build interface you are using
- Placing the heap segments in memory.

### Heap segments in DLIB for 78K0R

To access a heap in a specific memory when using the DLIB runtime environment with the 78K0R Compiler, use the appropriate memory attribute as a prefix to the standard functions `malloc`, `free`, `calloc`, and `realloc`, for example:

```
__near_malloc
```

If you use any of the standard functions without a prefix, the function will be mapped to the default memory type `near`.

Each heap will reside in a segment with the name `_HEAP` prefixed by a memory attribute. For information about available heaps, see *Heaps*, page 76.

### Heap segments for 78K0/78K0S and in CLIB for 78K0R

When using the CLIB runtime environment with the 78K0R Compiler and always when using the 78K0/78K0S Compiler, the memory allocated to the heap is placed in the segment `HEAP`, which is only included in the application if dynamic memory allocation is actually used.



#### Heap size allocation in the IDE

Choose **Project>Options**. In the **General Options** category, click the **Stack/Heap** tab. Add the required heap size in the **Heap size** text box.



#### Heap size allocation from the command line

The size of the heap segment is defined in the linker command file.

The default linker file sets up a constant, representing the size of the heap, at the beginning of the linker file:

```
-D_NEAR_HEAP_SIZE=size
-D_FAR_HEAP_SIZE=size
```

**Note:** Normally, these lines are prefixed with the comment character `//`. To make the directive take effect, remove the comment character.

Specify the appropriate size for your application.



#### Placement of heap segment

The actual heap segments are allocated in the memory area available for the heaps:

```
-Z (DATA) HEAP+_HEAP_SIZE=A000-FE1F
-Z (DATA) NEAR_HEAP+_NEAR_HEAP_SIZE=F1F00-FFE1F
-Z (DATA) FAR_HEAP+_FAR_HEAP_SIZE=[80000-EFFFF]/10000
```

The `/10000` syntax divides the address range into `0x10000`-byte pages. See the *IAR Linker and Library Tools Reference Guide* for more information.

**Note:** These ranges do not specify the size of the heap; they specify the range of the available memory.



#### Heap size and standard I/O

If your DLIB runtime environment is configured to use `FILE` descriptors, as in the Full configuration, input and output buffers for file handling will be allocated. In that case,

be aware that the size of the input and output buffers is set to 512 bytes in the `stdio` library header file. If the heap is too small, I/O will not be buffered, which is considerably slower than when I/O is buffered. If you execute the application using the simulator driver of the IAR C-SPY® Debugger, you are not likely to notice the speed penalty, but it is quite noticeable when the application runs on an 78K microcontroller. If you use the standard I/O library, you should set the heap size to a value which accommodates the needs of the standard I/O buffer, for example 1 Kbyte.

## LOCATED DATA

A variable that is explicitly placed at an address, for example by using the `#pragma location` directive or the `@` operator, is placed in either the `NEAR_A` or `FAR_A` (78K0R) segment. The individual segment part of the segment knows its location in the memory space, and it does not have to be specified in the linker command file.

## USER-DEFINED SEGMENTS

If you create your own segments using the `#pragma location` directive or the `@` operator, these segments must also be defined in the linker command file using the `-Z` or `-P` segment control directives.

---

## Code segments

This section contains descriptions of the segments used for storing code, and the interrupt vector table. For a complete list of all segments, see *Summary of segments*, page 253.

### STARTUP CODE

The segment `RCODE` contains code used during system setup (`cstartup`) and the runtime library code. The system setup code should be placed at the location where the chip starts executing code after a reset. For the 78K microcontroller, this is at the address `0x0080` (78K0 and 78K0S) or `0x000C0` (78K0R).

In the default linker command file, one of these lines will place the `RCODE` segment:

```
-P (CODE) RCODE=0080-1FFF
-P (CODE) RCODE=000C0-07FFF
```

### NORMAL CODE

Functions declared without a memory type attribute are placed in different segments, depending on which code model you are using.

### Default code location

Standard code—that is, all user-written code when you use the standard (78K0/78K0S) or near (78K0R) code model, as well as functions declared `__callt` and `__interrupt`—is placed in the CODE segment.

In the default linker command file it can look like this:

```
-Z (CODE) CODE=01000-07FFF
```

### Banked code (78K0/78K0S only)

When you use the banked code model, all user-written code is by default located in the BCODE segment. Here, the `-P` linker directive is used for distributing the banked code across the different code banks.

In the linker command file it can look like this:

```
-P (CODE) BCODE=[_CODEBANK_START-_CODEBANK_END] *_CODEBANK_BANKS+
10000
```

Note the use of the linker defines `_CODEBANK_START`, `_CODEBANK_END`, and `_CODEBANK_BANKS` to define the bank start address, bank end address, and the number of banks, respectively. This defines could, for example, look like this:

```
-D_CODEBANK_START=8000
-D_CODEBANK_END=9FFF
-D_CODEBANK_BANKS=4
```

You must also define the SFR register address for code bank switching:

```
-D_CODEBANK_REG=FF00
```

### Far code (78K0R only)

When you use the Far code model, all user-written code is by default located in the XCODE segment.

In the default linker command file it can look like this:

```
-Z (CODE) XCODE=[08000-EFFFF]/10000
```

**Note:** The `/10000` syntax divides the address range into 0x10000-byte pages. See the *IAR Linker and Library Tools Reference Guide* for more information.

## CALLF DECLARED FUNCTIONS (78K0 ONLY)

All functions that you declare using the extended keyword `__callf` are located in the segment `FCODE`. This segment is restricted to the memory area `0x0800-0x0FFF`, which means the linker directive would look like this:

```
-Z (CODE) FCODE=0800-0FFF
```

## INTERRUPT AND CALLT VECTORS

The interrupt vector table contains pointers to interrupt routines, including the reset routine. The table is placed in the segment `INTVEC`. You must place this segment in the address range `0x0-0x003F` (78K0/78K0S) or `0x0-0x0007F` (78K0R). The linker directive for 78K0/78K0S would then look like this:

```
-Z (CODE) INTVEC=0000-003F
```

The vectors for `__callt` declared functions are located in the segment `CLTVEC`. You must place this segment in the address range `0x0040-0x007F` (78K0/78K0S) or `0x00080-0x000BF` (78K0R). The linker directive for a 78K0/78K0S project would then look like this:

```
-Z (CONST) CLTVEC=0040-007F
```

The `__interrupt` and `__callt` declared functions are located in the code segment `CODE`, see *Normal code*, page 49.

## C++ DYNAMIC INITIALIZATION

In C++, all global objects are created before the `main` function is called. The creation of objects can involve the execution of a constructor.

The `DIFUNCT` segment contains a vector of addresses that point to initialization code. All entries in the vector are called when the system is initialized.

For example:

```
-Z (CONST) DIFUNCT=0000-7FFF
```

For additional information, see *DIFUNCT*, page 257.

---

## Verifying the linked result of code and data placement

The linker has several features that help you to manage code and data placement, for example, messages at link time and the linker map file.

## SEGMENT TOO LONG ERRORS AND RANGE ERRORS

All code or data that is placed in relocatable segments will have its absolute addresses resolved at link time. Note that it is not known until link time whether all segments will fit in the reserved memory ranges. If the contents of a segment do not fit in the address range defined in the linker command file, XLINK will issue a *segment too long* error.

Some instructions do not work unless a certain condition holds after linking, for example that a branch must be within a certain distance or that an address must be even. XLINK verifies that the conditions hold when the files are linked. If a condition is not satisfied, XLINK generates a *range error* or warning and prints a description of the error.

For further information about these types of errors, see the *IAR Linker and Library Tools Reference Guide*.

## LINKER MAP FILE

XLINK can produce an extensive cross-reference listing, which can optionally contain the following information:

- A segment map which lists all segments in dump order
- A module map which lists all segments, local symbols, and entries (public symbols) for every module in the program. All symbols not included in the output can also be listed
- A module summary which lists the contribution (in bytes) from each module
- A symbol list which contains every entry (global symbol) in every module.



Use the option **Generate linker listing** in the IDE, or the option `-x` on the command line, and one of their suboptions to generate a linker listing.

Normally, XLINK will not generate an output file if any errors, such as range errors, occur during the linking process. Use the option **Range checks disabled** in the IDE, or the option `-R` on the command line, to generate an output file even if a range error was encountered.

For further information about the listing options and the linker listing, see the *IAR Linker and Library Tools Reference Guide*, and the *IAR Embedded Workbench® IDE User Guide*.

# The DLIB runtime environment

This chapter describes the runtime environment in which an application executes. In particular, the chapter covers the DLIB runtime library and how you can modify it—setting options, overriding default library modules, or building your own library—to optimize it for your application.

The chapter also covers system initialization and termination; how an application can control what happens before the function `main` is called, and how you can customize the initialization.

The chapter then describes how to configure functionality like locale and file I/O, how to get C-SPY® runtime support, and how to prevent incompatible modules from being linked together.

Note that the DLIB runtime environment is the default when you use the C++ language; DLIB can be used with both C and the C++ languages. CLIB on the other hand can only be used with the C language. For information about the CLIB runtime environment, see the chapter *The CLIB runtime environment*.

---

## Introduction to the runtime environment

The runtime environment is the environment in which your application executes. The runtime environment depends on the target hardware, the software environment, and the application code. The IAR DLIB runtime environment can be used as is together with the debugger. However, to be able to run the application on hardware, you must adapt the runtime environment.

This section gives an overview of:

- The runtime environment and its components
- Library selection.

### RUNTIME ENVIRONMENT FUNCTIONALITY

The *runtime environment* supports ISO/ANSI C and C++ including the standard template library. The runtime environment consists of the *runtime library*, which

contains the functions defined by these standards, and include files that define the library interface.

The runtime library is delivered both as prebuilt libraries and (depending on your product package) as source files, and you can find them in the product subdirectories `78k\lib` and `78k\src\lib`, respectively.

The runtime environment also consists of a part with specific support for the target system, which includes:

- Support for hardware features:
  - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for register handling
  - Peripheral unit registers and interrupt definitions in include files.
- Runtime environment support, that is, startup and exit code and low-level interface to some library functions.
- Special compiler support for some functions, for instance functions for floating-point arithmetics.

The runtime environment support and the size of the heaps must be tailored for the specific hardware and application requirements.

For further information about the library, see the chapter *Library functions*.

## LIBRARY SELECTION

To configure the most code-efficient runtime environment, you must determine your application and hardware requirements. The more functionality you need, the larger your code will become.

IAR Embedded Workbench comes with a set of prebuilt runtime libraries. To get the required runtime environment, you can customize it by:

- Setting library options, for example, for choosing `scanf` input and `printf` output formatters, and for specifying the size of the stack and the heap
- Overriding certain library functions, for example `cstartup.s26`, with your own customized versions
- Choosing the level of support for certain standard library functionality, for example, locale, file descriptors, and multibyte characters, by choosing a *library configuration*: normal or full.

You can also make your own library configuration, but that requires that you *rebuild* the library. This allows you to get full control of the runtime environment.

**Note:** Your application project must be able to locate the library, include files, and the library configuration file.

## SITUATIONS THAT REQUIRE LIBRARY BUILDING

Building a customized library is complex. Therefore, consider carefully whether it is really necessary.

You must build your own library when:

- There is no prebuilt library for the required combination of compiler options or hardware support
- You want to define your own library configuration with support for locale, file descriptors, multibyte characters, et cetera.

For information about how to build a customized library, see *Building and using a customized library*, page 62.

## LIBRARY CONFIGURATIONS

It is possible to configure the level of support for, for example, locale, file descriptors, multibyte characters. The runtime library configuration is defined in the *library configuration file*. It contains information about what functionality is part of the runtime environment. The configuration file is used for tailoring a build of a runtime library, and tailoring the system header files used when compiling your application. The less functionality you need in the runtime environment, the smaller it is.

These DLIB library configurations are available:

Library configuration	Description
Normal DLIB	No locale interface, C locale, no file descriptor support, no multibyte characters in <code>printf</code> and <code>scanf</code> , and no hexadecimal floating-point numbers in <code>strtod</code> .
Full DLIB	Full locale interface, C locale, file descriptor support, multibyte characters in <code>printf</code> and <code>scanf</code> , and hexadecimal floating-point numbers in <code>strtod</code> .

Table 15: Library configurations

You can also define your own configurations, which means that you must modify the configuration file. Note that the library configuration file describes how a library was built and thus cannot be changed unless you rebuild the library. For further information, see *Building and using a customized library*, page 62.

The prebuilt libraries are based on the default configurations, see Table 17, *Prebuilt libraries*, page 57. There is also a ready-made library project template that you can use if you want to rebuild the runtime library.

## DEBUG SUPPORT IN THE RUNTIME LIBRARY

You can make the library provide different levels of debugging support—basic, runtime, and I/O debugging.

This table describes the different levels of debugging support:

Debugging support	Linker option in IDE	Linker command line option	Description
Basic debugging	Debug information for C-SPY	-Fubrof	Debug support for C-SPY without any runtime support
Runtime debugging	With runtime control modules	-r	The same as -Fubrof, but also includes debugger support for handling program abort, exit, and assertions.
I/O debugging	With I/O emulation modules	-rt	The same as -r, but also includes debugger support for I/O handling, which means that <code>stdin</code> and <code>stdout</code> are redirected to the C-SPY Terminal I/O window, and that it is possible to access files on the host computer during debugging.

Table 16: Levels of debugging support in runtime libraries

If you build your application project with the XLINK options **With runtime control modules** or **With I/O emulation modules**, certain functions in the library are replaced by functions that communicate with the IAR C-SPY Debugger. For further information, see *C-SPY runtime interface*, page 77.



To set linker options for debug support in the IAR Embedded Workbench IDE, choose **Project>Options** and select the **Linker** category. On the **Output** page, select the appropriate **Format** option.

## Using a prebuilt library

The prebuilt runtime libraries are configured for different combinations of these features:

- Code model
- Data model (78K0R only)
- Processor core
- Library configuration—Normal or Full.

These prebuilt runtime libraries are available:

Library object file	Code model	Data model	Processor core	Library configuration
d178ks0n.r26	Standard	—	78K0_basic	Normal
d178ks0f.r26	Standard	—	78K0_basic	Full
d178ks1n.r26	Standard	—	78K0	Normal
d178ks1f.r26	Standard	—	78K0	Full
d178ks2n.r26	Standard	—	78K0S	Normal
d178ks2f.r26	Standard	—	78K0S	Full
d178kb0n.r26	Banked	—	78K0_basic	Normal
d178kb0f.r26	Banked	—	78K0_basic	Full
d178kb1n.r26	Banked	—	78K0	Normal
d178kb1f.r26	Banked	—	78K0	Full
d178kb2n.r26	Banked	—	78K0S	Normal
d178kb2f.r26	Banked	—	78K0S	Full
d178knn3n.r26	Near	Near	78K0R	Normal
d178knn3f.r26	Near	Near	78K0R	Full
d178knf3n.r26	Near	Far	78K0R	Normal
d178knf3f.r26	Near	Far	78K0R	Full
d178kfn3n.r26	Far	Near	78K0R	Normal
d178kfn3f.r26	Far	Near	78K0R	Full
d178kff3n.r26	Far	Far	78K0R	Normal
d178kff3f.r26	Far	Far	78K0R	Full

Table 17: Prebuilt libraries

The names of the libraries are constructed in this way:

`<library><cpu><code_model><data_model><core><lib_conf>.r26`

where

- `<library>` is `d1` for the IAR DLIB runtime environment
- `<cpu>` is `78k`
- `<code_model>` is one of `s`, `b`, `n`, or `f` for standard, banked, near, and far code, respectively
- `<data_model>` is one of `n` or `f` for the near and far data model, respectively. This applies only to the 78K0R Compiler
- `<core>` is one of 0, 1, 2, or 3, for 78K0\_basic, 78K0, 78K0S, and 78K0R respectively

- `<lib_conf>` is one of `n` or `f` for normal and full, respectively.

**Note:** The library configuration file has the same base name as the library.



The IDE will include the correct library object file and library configuration file based on the options you select. See the *IAR Embedded Workbench® IDE User Guide* for additional information.



If you build your application from the command line, you must specify these items to get the required runtime library:

- Specify which library object file to use on the XLINK command line, for instance:  
`dl78ks1n.r26`
- Specify the include paths for the compiler and assembler:  
`-I 78k\inc\dlib`
- Specify the library configuration file for the compiler:  
`--dlib_config C:\...\dl78ks1n.h`

You can find the library object files and the library configuration files in the subdirectory `78k\lib\dlib`.

## CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

The prebuilt libraries delivered with the compilers can be used as is. However, it is possible to customize parts of a library without rebuilding it. There are two different methods:

- Setting options for:
  - Formatters used by `printf` and `scanf`
  - The sizes of the heap and the stack
- Overriding library modules with your own customized versions.

These items can be customized:

Items that can be customized	Described in
Formatters for <code>printf</code> and <code>scanf</code>	<i>Choosing formatters for printf and scanf</i> , page 59
Startup and termination code	<i>System startup and termination</i> , page 64
Low-level input and output	<i>Standard streams for input and output</i> , page 67
File input and output	<i>File input and output</i> , page 71
Low-level environment functions	<i>Environment interaction</i> , page 74
Low-level signal functions	<i>Signal and raise</i> , page 75
Low-level time functions	<i>Time</i> , page 75

Table 18: Customizable items

Items that can be customized	Described in
Size of heaps, stacks, and segments	Placing code and data, page 39

Table 18: Customizable items (Continued)

For a description about how to override library modules, see *Overriding library modules*, page 61.

## Choosing formatters for printf and scanf

To override the default formatter for all the `printf`- and `scanf`-related functions, except for `wprintf` and `wscanf` variants, you simply set the appropriate library options. This section describes the different options available.

**Note:** If you rebuild the library, it is possible to optimize these functions even further, see *Configuration symbols for printf and scanf*, page 70.

### CHOOSING PRINTF FORMATTER

The `printf` function uses a formatter called `_Printf`. The default version is quite large, and provides facilities not required in many embedded applications. To reduce the memory consumption, three smaller, alternative versions are also provided in the standard C/EC++ library.

This table summarizes the capabilities of the different formatters:

Formatting capabilities	<code>_PrintfFull</code>	<code>_PrintfLarge</code>	<code>_PrintfSmall</code>	<code>_PrintfTiny</code>
Basic specifiers <code>c</code> , <code>d</code> , <code>i</code> , <code>o</code> , <code>p</code> , <code>s</code> , <code>u</code> , <code>X</code> , <code>x</code> , and <code>%</code>	Yes	Yes	Yes	Yes
Multibyte support	†	†	†	No
Floating-point specifiers <code>a</code> , and <code>A</code>	Yes	No	No	No
Floating-point specifiers <code>e</code> , <code>E</code> , <code>f</code> , <code>F</code> , <code>g</code> , and <code>G</code>	Yes	Yes	No	No
Conversion specifier <code>n</code>	Yes	Yes	No	No
Format flag space, <code>+</code> , <code>-</code> , <code>#</code> , and <code>0</code>	Yes	Yes	Yes	No
Length modifiers <code>h</code> , <code>l</code> , <code>L</code> , <code>s</code> , <code>t</code> , and <code>Z</code>	Yes	Yes	Yes	No
Field width and precision, including <code>*</code>	Yes	Yes	Yes	No
<code>long long</code> support	Yes	Yes	No	No

Table 19: Formatters for printf

† Depends on the library configuration that is used.

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 70.



### Specifying the print formatter in the IDE

To use any other formatter than the default (Full), choose **Project>Options** and select the **General Options** category. Select the appropriate option on the **Library options** page.



### Specifying printf formatter from the command line

To use any other formatter than the default (`_PrintfFull`), add one of these lines in the linker command file you are using:

```
-e_PrintfLarge=_Printf
-e_PrintfSmall=_Printf
-e_PrintfTiny=_Printf
```

## CHOOSING SCANF FORMATTER

In a similar way to the `printf` function, `scanf` uses a common formatter, called `_Scanf`. The default version is very large, and provides facilities that are not required in many embedded applications. To reduce the memory consumption, two smaller, alternative versions are also provided in the standard C/C++ library.

This table summarizes the capabilities of the different formatters:

Formatting capabilities	<code>_ScanfFull</code>	<code>_ScanfLarge</code>	<code>_ScanfSmall</code>
Basic specifiers <code>c</code> , <code>d</code> , <code>i</code> , <code>o</code> , <code>p</code> , <code>s</code> , <code>u</code> , <code>X</code> , <code>x</code> , and <code>%</code>	Yes	Yes	Yes
Multibyte support	†	†	†
Floating-point specifiers <code>a</code> , and <code>A</code>	Yes	No	No
Floating-point specifiers <code>e</code> , <code>E</code> , <code>f</code> , <code>F</code> , <code>g</code> , and <code>G</code>	Yes	No	No
Conversion specifier <code>n</code>	Yes	No	No
Scan set [ and ]	Yes	Yes	No
Assignment suppressing <code>*</code>	Yes	Yes	No
<code>long long</code> support	Yes	No	No

Table 20: Formatters for `scanf`

† Depends on the library configuration that is used.

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 70.



### Specifying scanf formatter in the IDE

To use any other formatter than the default (Full), choose **Project>Options** and select the **General Options** category. Select the appropriate option on the **Library options** page.



### Specifying scanf formatter from the command line

To use any other variant than the default (`_ScanfFull`), add one of these lines in the linker command file you are using:

```
-e_ScanfLarge=_Scanf
-e_ScanfSmall=_Scanf
```

---

## Overriding library modules

The library contains modules which you probably need to override with your own customized modules, for example functions for character-based I/O and `cstartup`. This can be done without rebuilding the entire library. This section describes the procedure for including your version of the module in the application project build process. The library files that you can override with your own versions are located in the `78k\src\lib` directory.

**Note:** If you override a default I/O library module with your own module, C-SPY support for the module is turned off. For example, if you replace the module `__write` with your own version, the C-SPY Terminal I/O window will not be supported.



### Overriding library modules using the IDE

This procedure is applicable to any source file in the library, which means that `library_module.c` in this example can be *any* module in the library.

- 1 Copy the appropriate `library_module.c` file to your project directory.
- 2 Make the required additions to the file (or create your own routine, using the default file as a model), and make sure that it has the same *module name* as the original module. The easiest way to achieve this is to save the new file under the same name as the original file.
- 3 Add the customized file to your project.
- 4 Rebuild your project.



### Overriding library modules from the command line

This procedure is applicable to any source file in the library, which means that `library_module.c` in this example can be *any* module in the library.

- 1 Copy the appropriate `library_module.c` to your project directory.
- 2 Make the required additions to the file (or create your own routine, using the default file as a model), and make sure that it has the same *module name* as the original module. The easiest way to achieve this is to save the new file under the same name as the original file.
- 3 Compile the modified file using the same options as for the rest of the project:

```
icc78k library_module.c
```

This creates a replacement object module file named `library_module.r26`.

**Note:** The code model, include paths, and the library configuration file must be the same for `library_module` as for the rest of your code.

- 4 Add `library_module.r26` to the XLINK command line, either directly or by using an extended linker command file, for example:

```
xlink library_module.r26 dl78ks0n.r26
```

Make sure that `library_module.r26` is placed before the library on the command line. This ensures that your module is used instead of the one in the library.

Run XLINK to rebuild your application.

This will use your version of `library_module.r26`, instead of the one in the library. For information about the XLINK options, see the *IAR Linker and Library Tools Reference Guide*.

---

## Building and using a customized library

In some situations, see *Situations that require library building*, page 55, it is necessary to rebuild the library. In those cases you must:

- Set up a library project
- Make the required library modifications
- Build your customized library
- Finally, make sure your application project will use the customized library.

**Note:** To build IAR Embedded Workbench projects from the command line, use the IAR Command Line Build Utility (`iarbuild.exe`). However, no make or batch files for building the library from the command line are provided.

Information about the build process and the IAR Command Line Build Utility, see the *IAR Embedded Workbench® IDE User Guide*.

## SETTING UP A LIBRARY PROJECT

The IDE provides a library project template which can be used for customizing the runtime environment configuration. This library template has Full library configuration, see Table 15, *Library configurations*, page 55.



In the IDE, modify the generic options in the created library project to suit your application, see *Basic settings for project configuration*, page 6.

**Note:** There is one important restriction on setting options. If you set an option on file level (file level override), no options on higher levels that operate on files will affect that file.

## MODIFYING THE LIBRARY FUNCTIONALITY

You must modify the library configuration file and build your own library if you want to modify support for, for example, locale, file descriptors, and multibyte characters. This will include or exclude certain parts of the runtime environment.

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `DLib_defaults.h`. This read-only file describes the configuration possibilities. Your library also has its own library configuration file `d178kcustom.h`, which sets up that specific library with full library configuration. For more information, see Table 18, *Customizable items*, page 58.

The library configuration file is used for tailoring a build of the runtime library, and for tailoring the system header files.

### Modifying the library configuration file

In your library project, open the file `d178kCustom.h` and customize it by setting the values of the configuration symbols according to the application requirements.

When you are finished, build your library project with the appropriate project options.

## USING A CUSTOMIZED LIBRARY

After you build your library, you must make sure to use it in your application project.



In the IDE you must do these steps:

- 1 Choose **Project>Options** and click the **Library Configuration** tab in the **General Options** category.
- 2 Choose **Custom DLIB** from the **Library** drop-down menu.

- 3 In the **Library file** text box, locate your library file.
- 4 In the **Configuration file** text box, locate your library configuration file.

---

## System startup and termination

This section describes the runtime environment actions performed during startup and termination of your application.

The code for handling startup and termination is located in the source files `cstartup.s26`, `cexit_dlib.s26`, and `low_level_init.c` located in the `78k\src\lib` directory.

For information about how to customize the system startup code, see *Customizing system initialization*, page 66.

### SYSTEM STARTUP

During system startup, an initialization sequence is executed before the `main` function is entered. This sequence performs initializations required for the target hardware and the C/C++ environment.

For the hardware initialization, it looks like this:

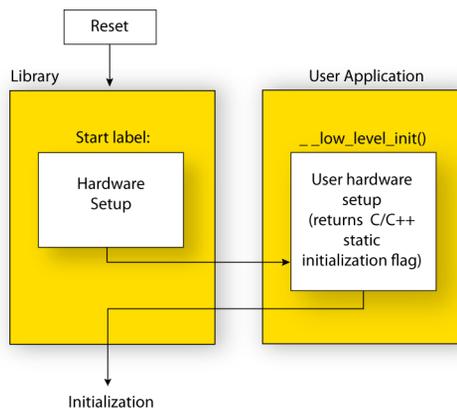


Figure 3: Target hardware initialization phase

- When the CPU is reset it will jump to the program entry label `__program_start` in the system startup code.
- The function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations.

For the C/C++ initialization, it looks like this:

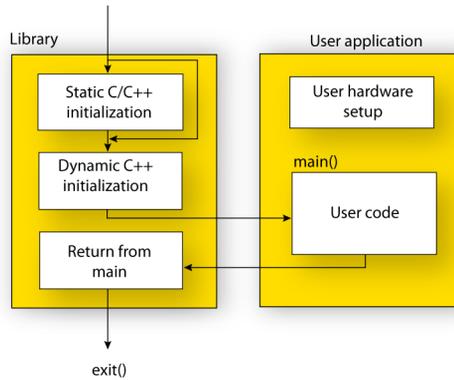


Figure 4: C/C++ initialization phase

- Static and global variables are initialized. That is, zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This step is skipped if `__low_level_init` returns zero. For more details, see *Initialized data*, page 45
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

## SYSTEM TERMINATION

This illustration shows the different ways an embedded application can terminate in a controlled way:

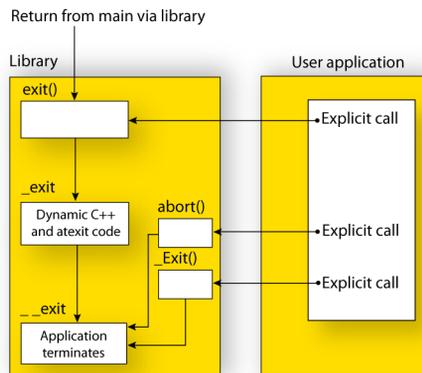


Figure 5: System termination phase

An application can terminate normally in two different ways:

- Return from the `main` function
- Call the `exit` function.

As the ISO/ANSI C standard states that the two methods should be equivalent, the system startup code calls the `exit` function if `main` returns. The parameter passed to the `exit` function is the return value of `main`.

The default `exit` function is written in C. It calls a small assembler function `_exit` that will perform these operations:

- Call functions registered to be executed when the application ends. This includes C++ destructors for static and global variables, and functions registered with the standard C function `atexit`
- Close all open files
- Call `__exit`
- When `__exit` is reached, stop the system.

An application can also exit by calling the `abort` or the `_Exit` function. The `abort` function just calls `__exit` to halt the system, and does not perform any type of cleanup. The `_Exit` function is equivalent to the `abort` function, except for the fact that `_Exit` takes an argument for passing exit status information.

If you want your application to do anything extra at exit, for example resetting the system, you can write your own implementation of the `__exit(int)` function.

### C-SPY interface to system termination

If your project is linked with the XLINK options **With runtime control modules** or **With I/O emulation modules**, the normal `__exit` and `abort` functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see *C-SPY runtime interface*, page 77.

---

## Customizing system initialization

It is likely that you need to customize the code for system initialization. For example, your application might need to initialize memory-mapped special function registers (SFRs), or omit the default initialization of data segments performed by `cstartup`.

You can do this by providing a customized version of the routine `__low_level_init`, which is called from `cstartup.s26` before the data segments are initialized. Modifying the file `cstartup` directly should be avoided.

The code for handling system startup is located in the source files `cstartup.s26` and `low_level_init.c`, located in the `78k\src\lib` directory.

**Note:** Normally, you do not need to customize `cexit.s26`.



If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 62.

**Note:** Regardless of whether you modify the routine `__low_level_init` or the file `cstartup.s26`, you do not have to rebuild the library.

### **\_\_LOW\_LEVEL\_INIT**

A skeleton low-level initialization file is supplied with the product: `low_level_init.c`. Note that static initialized variables cannot be used within the file, because variables have not been initialized at this point.

The value returned by `__low_level_init` determines whether or not data segments should be initialized by the system startup code. If the function returns 0, the data segments will not be initialized.

### **MODIFYING THE FILE CSTARTUP.S26**

As noted earlier, you should not modify the file `cstartup.s26` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s26`, we recommend that you follow the general procedure for creating a modified copy of the file and adding it to your project, see *Overriding library modules*, page 61.

Note that you must make sure that the linker uses the same start label as used in your version of `cstartup.s26`. For information about how to change the start label used by the linker, read about the `-s` option in the *IAR Linker and Library Tools Reference Guide*.

---

## **Standard streams for input and output**

Standard communication channels (streams) are defined in `stdio.h`. If any of these streams are used by your application, for example by the functions `printf` and `scanf`, you must customize the low-level functionality to suit your hardware.

There are primitive I/O functions, which are the fundamental functions through which C and C++ performs all character-based I/O. For any character-based I/O to be available, you must provide definitions for these functions using whatever facilities the hardware environment provides.

## IMPLEMENTING LOW-LEVEL CHARACTER INPUT AND OUTPUT

To implement low-level functionality of the `stdin` and `stdout` streams, you must write the functions `__read` and `__write`, respectively. You can find template source code for these functions in the `78k\src\lib` directory.

If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 62. Note that customizing the low-level routines for input and output does not require you to rebuild the library.

**Note:** If you write your own variants of `__read` or `__write`, special considerations for the C-SPY runtime interface are needed, see *C-SPY runtime interface*, page 77.

### Example of using `__write`

The code in this example uses memory-mapped I/O to write to an LCD display:

```
#include <stddef.h>

__no_init volatile unsigned char lcdIO @ 0x8000;

size_t __write(int handle,
               const unsigned char *buf,
               size_t bufSize)
{
    size_t nChars = 0;

    /* Check for the command to flush all handles */
    if (handle == -1)
    {
        return 0;
    }

    /* Check for stdout and stderr
       (only necessary if FILE descriptors are enabled.) */
    if (handle != 1 && handle != 2)
    {
        return -1;
    }

    for (/* Empty */; bufSize > 0; --bufSize)
    {
        lcdIO = *buf;
        ++buf;
        ++nChars;
    }
}
```

```

    return nChars;
}

```

**Note:** A call to `__write` where `buf` has the value `NULL` is a command to flush the handle.

### Example of using `__read`

The code in this example uses memory-mapped I/O to read from a keyboard:

```

#include <stddef.h>

__no_init volatile unsigned char kbIO @ 0x8000;

size_t __read(int handle,
              unsigned char *buf,
              size_t bufSize)
{
    size_t nChars = 0;

    /* Check for stdin
       (only necessary if FILE descriptors are enabled) */
    if (handle != 0)
    {
        return -1;
    }

    for (/*Empty*/; bufSize > 0; --bufSize)
    {
        unsigned char c = kbIO;
        if (c == 0)
            break;

        *buf++ = c;
        ++nChars;
    }

    return nChars;
}

```

For information about the `@` operator, see *Controlling data and function placement in memory*, page 126.

## Configuration symbols for printf and scanf

When you set up your application project, you typically need to consider what `printf` and `scanf` formatting capabilities your application requires, see *Choosing formatters for printf and scanf*, page 59.

If the provided formatters do not meet your requirements, you can customize the full formatters. However, that means you must rebuild the runtime library.

The default behavior of the `printf` and `scanf` formatters are defined by configuration symbols in the file `DLib_Defaults.h`.

These configuration symbols determine what capabilities the function `printf` should have:

Printf configuration symbols	Includes support for
<code>_DLIB_PRINTF_MULTIBYTE</code>	Multibyte characters
<code>_DLIB_PRINTF_LONG_LONG</code>	Long long (ll qualifier)
<code>_DLIB_PRINTF_SPECIFIER_FLOAT</code>	Floating-point numbers
<code>_DLIB_PRINTF_SPECIFIER_A</code>	Hexadecimal floating-point numbers
<code>_DLIB_PRINTF_SPECIFIER_N</code>	Output count (%n)
<code>_DLIB_PRINTF_QUALIFIERS</code>	Qualifiers h, l, L, v, t, and z
<code>_DLIB_PRINTF_FLAGS</code>	Flags -, +, #, and 0
<code>_DLIB_PRINTF_WIDTH_AND_PRECISION</code>	Width and precision
<code>_DLIB_PRINTF_CHAR_BY_CHAR</code>	Output char by char or buffered

Table 21: Descriptions of printf configuration symbols

When you build a library, these configurations determine what capabilities the function `scanf` should have:

Scanf configuration symbols	Includes support for
<code>_DLIB_SCANF_MULTIBYTE</code>	Multibyte characters
<code>_DLIB_SCANF_LONG_LONG</code>	Long long (ll qualifier)
<code>_DLIB_SCANF_SPECIFIER_FLOAT</code>	Floating-point numbers
<code>_DLIB_SCANF_SPECIFIER_N</code>	Output count (%n)
<code>_DLIB_SCANF_QUALIFIERS</code>	Qualifiers h, j, l, t, z, and L
<code>_DLIB_SCANF_SCANSET</code>	Scanset ([*])
<code>_DLIB_SCANF_WIDTH</code>	Width
<code>_DLIB_SCANF_ASSIGNMENT_SUPPRESSING</code>	Assignment suppressing ([*])

Table 22: Descriptions of scanf configuration symbols

## CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you must set up a library project, see *Building and using a customized library*, page 62. Define the configuration symbols according to your application requirements.

---

## File input and output

The library contains a large number of powerful functions for file I/O operations. If you use any of these functions, you must customize them to suit your hardware. To simplify adaptation to specific hardware, all I/O functions call a small set of primitive functions, each designed to accomplish one particular task; for example, `__open` opens a file, and `__write` outputs characters.

Note that file I/O capability in the library is only supported by libraries with full library configuration, see *Library configurations*, page 55. In other words, file I/O is supported when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is enabled. If not enabled, functions taking a `FILE *` argument cannot be used.

Template code for these I/O files are included in the product:

I/O function	File	Description
<code>__close</code>	<code>close.c</code>	Closes a file.
<code>__lseek</code>	<code>lseek.c</code>	Sets the file position indicator.
<code>__open</code>	<code>open.c</code>	Opens a file.
<code>__read</code>	<code>read.c</code>	Reads a character buffer.
<code>__write</code>	<code>write.c</code>	Writes a character buffer.
<code>remove</code>	<code>remove.c</code>	Removes a file.
<code>rename</code>	<code>rename.c</code>	Renames a file.

Table 23: Low-level I/O files

The primitive functions identify I/O streams, such as an open file, with a file descriptor that is a unique integer. The I/O streams normally associated with `stdin`, `stdout`, and `stderr` have the file descriptors 0, 1, and 2, respectively.

**Note:** If you link your library with I/O debugging support, C-SPY variants of the low-level I/O functions are linked for interaction with C-SPY. For more information, see *Debug support in the runtime library*, page 56.

---

## Locale

*Locale* is a part of the C language that allows language- and country-specific settings for several areas, such as currency symbols, date and time, and multibyte character encoding.

Depending on what runtime library you are using you get different level of locale support. However, the more locale support, the larger your code will get. It is therefore necessary to consider what level of support your application needs.

The DLIB library can be used in two main modes:

- With locale interface, which makes it possible to switch between different locales during runtime
- Without locale interface, where one selected locale is hardwired into the application.

### LOCALE SUPPORT IN PREBUILT LIBRARIES

The level of locale support in the prebuilt libraries depends on the library configuration.

- All prebuilt libraries support the C locale only
- All libraries with *full library configuration* have support for the locale interface. For prebuilt libraries with locale interface, it is by default only supported to switch multibyte character encoding during runtime.
- Libraries with *normal library configuration* do not have support for the locale interface.

If your application requires a different locale support, you must rebuild the library.

### CUSTOMIZING THE LOCALE SUPPORT

If you decide to rebuild the library, you can choose between these locales:

- The standard C locale
- The POSIX locale
- A wide range of European locales.

### Locale configuration symbols

The configuration symbol `_DLIB_FULL_LOCALE_SUPPORT`, which is defined in the library configuration file, determines whether a library has support for a locale interface or not. The locale configuration symbols `_LOCALE_USE_LANG_REGION` and `_ENCODING_USE_ENCODING` define all the supported locales and encodings:

```
#define _DLIB_FULL_LOCALE_SUPPORT 1
#define _LOCALE_USE_C           /* C locale */
```

```
#define _LOCALE_USE_EN_US    /* American English */
#define _LOCALE_USE_EN_GB    /* British English */
#define _LOCALE_USE_SV_SE    /* Swedish in Sweden */
```

See `DLib_Defaults.h` for a list of supported locale and encoding settings.

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Building and using a customized library*, page 62.

**Note:** If you use multibyte characters in your C or assembler source code, make sure that you select the correct locale symbol (the local host locale).

### Building a library without support for locale interface

The locale interface is not included if the configuration symbol `_DLIB_FULL_LOCALE_SUPPORT` is set to 0 (zero). This means that a hardwired locale is used—by default the standard C locale—but you can choose one of the supported locale configuration symbols. The `setlocale` function is not available and can therefore not be used for changing locales at runtime.

### Building a library with support for locale interface

Support for the locale interface is obtained if the configuration symbol `_DLIB_FULL_LOCALE_SUPPORT` is set to 1. By default, the standard C locale is used, but you can define as many configuration symbols as required. Because the `setlocale` function will be available in your application, it will be possible to switch locales at runtime.

## CHANGING LOCALES AT RUNTIME

The standard library function `setlocale` is used for selecting the appropriate portion of the application's locale when the application is running.

The `setlocale` function takes two arguments. The first one is a locale category that is constructed after the pattern `LC_CATEGORY`. The second argument is a string that describes the locale. It can either be a string previously returned by `setlocale`, or it can be a string constructed after the pattern:

```
lang_REGION
```

or

```
lang_REGION.encoding
```

The *lang* part specifies the language code, and the *REGION* part specifies a region qualifier, and *encoding* specifies the multibyte character encoding that should be used.

The `lang_REGION` part matches the `_LOCALE_USE_LANG_REGION` preprocessor symbols that can be specified in the library configuration file.

### Example

This example sets the locale configuration symbols to Swedish to be used in Finland and UTF8 multibyte character encoding:

```
setlocale (LC_ALL, "sv_FI.Utf8");
```

---

## Environment interaction

According to the C standard, your application can interact with the environment using the functions `getenv` and `system`.

**Note:** The `putenv` function is not required by the standard, and the library does not provide an implementation of it.

The `getenv` function searches the string, pointed to by the global variable `__environ`, for the key that was passed as argument. If the key is found, the value of it is returned, otherwise 0 (zero) is returned. By default, the string is empty.

To create or edit keys in the string, you must create a sequence of null terminated strings where each string has the format:

```
key=value\0
```

End the string with an extra null character (if you use a C string, this is added automatically). Assign the created sequence of strings to the `__environ` variable.

For example:

```
const char MyEnv[] = "Key=Value\0Key2=Value2\0";
__environ = MyEnv;
```

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `78k\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 61.

If you need to use the `system` function, you must implement it yourself. The `system` function available in the library simply returns -1.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 62.

**Note:** If you link your application with support for I/O debugging, the functions `getenv` and `system` are replaced by C-SPY variants. For further information, see *Debug support in the runtime library*, page 56.

---

## Signal and raise

Default implementations of the functions `signal` and `raise` are available. If these functions do not provide the functionality that you need, you can implement your own versions.

This does not require that you rebuild the library. You can find source templates in the files `signal.c` and `raise.c` in the `78k\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 61.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 62.

---

## Time

To make the `time` and `date` functions work, you must implement the three functions `clock`, `time`, and `__getzone`.

This does not require that you rebuild the library. You can find source templates in the files `clock.c` and `time.c`, and `getzone.c` in the `78k\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 61.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 62.

The default implementation of `__getzone` specifies UTC as the time zone.

**Note:** If you link your application with support for I/O debugging, the functions `clock` and `time` are replaced by C-SPY variants that return the host clock and time respectively. For further information, see *C-SPY runtime interface*, page 77.

---

## Strtod

The function `strtod` does not accept hexadecimal floating-point strings in libraries with the normal library configuration. To make a library do so, you must rebuild the library, see *Building and using a customized library*, page 62. Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.

---

## Assert

If you linked your application with support for runtime debugging, C-SPY will be notified about failed asserts. If this is not the behavior you require, you must add the source file `xreportassert.c` to your application project. Alternatively, you can rebuild the library. The `__ReportAssert` function generates the assert notification. You can find template code in the `78k\src\lib` directory. For further information, see *Building and using a customized library*, page 62. To turn off assertions, you must define the symbol `NDEBUG`.



In the IDE, this symbol `NDEBUG` is by default defined in a Release project and *not* defined in a Debug project. If you build from the command line, you must explicitly define the symbol according to your needs.

---

## Heaps

The runtime environment supports heaps in these memory types for the 78K0R Compiler:

Memory type	Segment name	Memory attribute	Used by default in data model
Near	NEAR_HEAP	<code>__near</code>	Near
Far	FAR_HEAP	<code>__far</code>	Far

Table 24: Heaps and memory types

See *The heap*, page 47 for information about how to set the size for each heap. To use a specific heap, the prefix in the table is the memory attribute to use in front of `malloc`, `free`, `calloc`, and `realloc`, for example `__near_malloc`. The default functions will use one of the specific heap variants, depending on project settings such as data model. For information about how to use a specific heap in C++, see *New and Delete operators (78K0R only)*, page 114.

---

## Hardware support

Some 78K microcontrollers have a hardware multiplier/divider unit. To include runtime support for this unit, use the sample library code provided in the `78k\src\lib\hw_multiply_division_units` directory



In the IDE, select the **Use Hardware Multiplier/Divider Unit** option on the **Project Options>General Options>Library Configuration** page. If the currently selected 78K device does not have a hardware multiplier/divider unit, this option will not be available.

## C-SPY runtime interface

To include support for runtime and I/O debugging, you must link your application with the XLINK options **With runtime control modules** or **With I/O emulation modules**, see *Debug support in the runtime library*, page 56.

In this case, C-SPY variants of these library functions are linked to the application:

Function	Description
<code>abort</code>	C-SPY notifies that the application has called <code>abort</code> *
<code>clock</code>	Returns the clock on the host computer
<code>__close</code>	Closes the associated host file on the host computer
<code>__exit</code>	C-SPY notifies that the end of the application was reached *
<code>__open</code>	Opens a file on the host computer
<code>__read</code>	<code>stdin</code> , <code>stdout</code> , and <code>stderr</code> will be directed to the Terminal I/O window; all other files will read the associated host file
<code>remove</code>	Writes a message to the Debug Log window and returns -1
<code>rename</code>	Writes a message to the Debug Log window and returns -1
<code>_ReportAssert</code>	Handles failed asserts *
<code>__seek</code>	Seeks in the associated host file on the host computer
<code>system</code>	Writes a message to the Debug Log window and returns -1
<code>time</code>	Returns the time on the host computer
<code>__write</code>	<code>stdin</code> , <code>stdout</code> , and <code>stderr</code> will be directed to the Terminal I/O window, all other files will write to the associated host file

Table 25: Functions with special meanings when linked with debug info

\* The linker option **With I/O emulation modules** is not required for these functions.

## LOW-LEVEL DEBUGGER RUNTIME INTERFACE

The low-level debugger runtime interface is used for communication between the application being debugged and the debugger itself. The debugger provides runtime services to the application via this interface; services that allow capabilities like file and terminal I/O to be performed on the host computer.

These capabilities can be valuable during the early development of an application, for example in an application using file I/O before any flash file system I/O drivers are implemented. Or, if you need to debug constructions in your application that use `stdin` and `stdout` without the actual hardware device for input and output being available. Another debugging purpose can be to produce debug trace printouts.

The mechanism used for implementing this feature works as follows:

The debugger will detect the presence of the function `__DebugBreak`, which will be part of the application if you linked it with the XLINK options for C-SPY runtime interface. In this case, the debugger will automatically set a breakpoint at the `__DebugBreak` function. When the application calls, for example `open`, the `__DebugBreak` function is called, which will cause the application to break and perform the necessary services. The execution will then resume.

## THE DEBUGGER TERMINAL I/O WINDOW

To make the Terminal I/O window available, the application must be linked with support for I/O debugging, see *Debug support in the runtime library*, page 56. This means that when the functions `__read` or `__write` are called to perform I/O operations on the streams `stdin`, `stdout`, or `stderr`, data will be sent to or read from the C-SPY Terminal I/O window.

**Note:** The Terminal I/O window is not opened automatically just because `__read` or `__write` is called; you must open it manually.

See the *IAR Embedded Workbench® IDE User Guide* for more information about the Terminal I/O window.

## Speeding up terminal output

On some systems, terminal output might be slow because the host computer and the target hardware must communicate for each character.

For this reason, a replacement for the `__write` function called `__write_buffered` is included in the DLIB library. This module buffers the output and sends it to the debugger one line at a time, speeding up the output. Note that this function uses about 80 bytes of RAM memory.

To use this feature you can either choose **Project>Options>Linker>Output** and select the option **Buffered terminal output** in the IDE, or add this to the linker command line:

```
-e__write_buffered=__write
```

---

## Checking module consistency

This section introduces the concept of runtime model attributes, a mechanism used by the IAR compilers, assemblers, and linker to ensure that modules are built using compatible settings.

When developing an application, it is important to ensure that incompatible modules are not used together. For example, in the compiler, the floating-point format is incompatible between the CLIB library and the DLIB library. If you write a routine that

only works for the DLIB library, it is possible to check that the routine is not used in an application built using the CLIB library.

The tools provided by IAR Systems use a set of predefined runtime model attributes. You can use these predefined attributes or define your own to perform any type of consistency check.

## RUNTIME MODEL ATTRIBUTES

A runtime attribute is a pair constituted of a named key and its corresponding value. Two modules can only be linked together if they have the same value for each key that they both define.

There is one exception: if the value of an attribute is `*`, then that attribute matches any value. The reason for this is that you can specify this in a module to show that you have considered a consistency property, and this ensures that the module does not rely on that property.

### Example

In this table, the object files could (but do not have to) define the two runtime attributes `color` and `taste`:

Object file	Color	Taste
<code>file1</code>	<code>blue</code>	<code>not defined</code>
<code>file2</code>	<code>red</code>	<code>not defined</code>
<code>file3</code>	<code>red</code>	<code>*</code>
<code>file4</code>	<code>red</code>	<code>spicy</code>
<code>file5</code>	<code>red</code>	<code>lean</code>

Table 26: Example of runtime model attributes

In this case, `file1` cannot be linked with any of the other files, since the runtime attribute `color` does not match. Also, `file4` and `file5` cannot be linked together, because the `taste` runtime attribute does not match.

On the other hand, `file2` and `file3` can be linked with each other, and with either `file4` or `file5`, but not with both.

## USING RUNTIME MODEL ATTRIBUTES

To ensure module consistency with other object files, use the `#pragma rtmodel` directive to specify runtime model attributes in your C/C++ source code. For example:

```
#pragma rtmodel="__rt_version", "1"
```

For detailed syntax information, see *rtmodel*, page 229.

You can also use the `rtmodel` assembler directive to specify runtime model attributes in your assembler source code. For example:

```
rtmodel "color", "red"
```

For detailed syntax information, see the *IAR Assemblers Reference Guide for 78K*.

**Note:** The predefined runtime attributes all start with two underscores. Any attribute names you specify yourself should not contain two initial underscores in the name, to eliminate any risk that they will conflict with future IAR runtime attribute names.

At link time, the IAR XLINK Linker checks module consistency by ensuring that modules with conflicting runtime attributes will not be used together. If conflicts are detected, an error is issued.

## PREDEFINED RUNTIME ATTRIBUTES

The table below shows the predefined runtime model attributes that are available for the compilers. These can be included in assembler code or in mixed C/C++ and assembler code.

Runtime model attribute	Value	Description
<code>__code_model</code>	standard, banked, near, or far	Corresponds to the code model used in the project.
<code>__data_model</code>	near or far	Corresponds to the data model used in the project. 78K0R only.
<code>__rt_version</code>	<i>n</i>	This runtime key is always present in all modules generated by the compiler. If a major change in the runtime characteristics occurs, the value of this key changes.

Table 27: Predefined runtime model attributes

The easiest way to find the proper settings of the `RTMODEL` directive is to compile a C or C++ module to generate an assembler file, and then examine the file.

If you are using assembler routines in the C or C++ code, refer to the chapter *Assembler directives* in the *IAR Assemblers Reference Guide for 78K*.

## Examples

The following assembler source code provides a function that increases the register `E` to count the number of times it was called. The routine assumes that the application does not use `E` for anything else, that is, the register is locked for usage. To ensure this, a runtime module attribute, `__reg_e`, is defined with a value `counter`. This definition will ensure that this specific module can only be linked with either other modules

containing the same definition, or with modules that do not set this attribute. Note that the compiler sets this attribute to `free`, unless the register is locked.

```

        module          myCounter
        public          myCounter
        rseg             CODE:CODE:NOROOT(1)
        rtmodel          "__reg_e", "counter"
myCounter: inc          E
           ret
           endmod
           end

```

If this module is used in an application that contains modules where the register `E` is not locked, the linker issues an error:

```

Error[e117]: Incompatible runtime models. Module myCounter
specifies that '__reg_e' must be 'counter', but module part1 has
the value 'free'

```

## USER-DEFINED RUNTIME MODEL ATTRIBUTES

In cases where the predefined runtime model attributes are not sufficient, you can use the `RTMODEL` assembler directive to define your own attributes. For each property, select a key and a set of values that describe the states of the property that are incompatible. Note that key names that start with two underscores are reserved by the compiler.

For example, if you have a UART that can run in two modes, you can specify a runtime model attribute, for example `uart`. For each mode, specify a value, for example `mode1` and `mode2`. Declare this in each module that assumes that the UART is in a particular mode. This is how it could look like in one of the modules:

```
#pragma rtmodel="uart", "mode1"
```



# The CLIB runtime environment

This chapter describes the runtime environment in which an application executes. In particular, it covers the CLIB runtime library and how you can optimize it for your application.

The standard library uses a small set of low-level input and output routines for character-based I/O. This chapter describes how the low-level routines can be replaced by your own version. The chapter also describes how you can choose printf and scanf formatters.

The chapter then describes system initialization and termination. It presents how an application can control what happens before the start function main is called, and the method for how you can customize the initialization. Finally, the C-SPY® runtime interface is covered.

Note that the legacy CLIB runtime environment is provided for backward compatibility and should not be used for new application projects.

For information about migrating from CLIB to DLIB, see the *78K IAR Embedded Workbench Migration Guide*.

---

## Runtime environment

The CLIB runtime environment includes the C standard library. The linker will include only those routines that are required—directly or indirectly—by your application. For detailed reference information about the runtime libraries, see the chapter *Library functions*.

The prebuilt runtime libraries are configured for different combinations of these features:

- Code model
- Data model
- Processor core.

These prebuilt libraries are available:

Library object file	Code model	Data model	Processor core
c178ks0.r26	Standard	—	78K0_basic
c178ks1.r26	Standard	—	78K0
c178ks2.r26	Standard	—	78K0S
c178kb0.r26	Banked	—	78K0_basic
c178kb1.r26	Banked	—	78K0
c178kb2.r26	Banked	—	78K0S
c178knn3.r26	Near	Near	78K0R
c178knf3.r26	Near	Far	78K0R
c178kfn3.r26	Far	Near	78K0R
c178kff3.r26	Far	Far	78K0R

Table 28: Runtime libraries

The runtime library names are constructed in this way:

```
<type><cpu><code_model><data_model><core>.r26
```

where

- `<type>` c1 for the IAR CLIB Library
- `<cpu>` is 78k
- `<code_model>` is one of s, b, n, or f for standard, banked, near, and far code, respectively
- `<data_model>` is one of n or f for the near and far data model, respectively. This applies only to the 78K0R Compiler
- `<core>` is one of 0, 1, 2, or 3, for 78K0\_basic, 78K0, 78K0S, and 78K0R respectively.



The IDE includes the correct runtime library based on the options you select. See the *IAR Embedded Workbench® IDE User Guide* for additional information.



Specify which runtime library object file to use on the XLINK command line, for instance:

```
c178ks1.r26
```

---

## Input and output

You can customize:

- The functions related to character-based I/O
- The formatters used by `printf/sprintf` and `scanf/sscanf`.

### CHARACTER-BASED I/O

The functions `putchar` and `getchar` are the fundamental C functions for character-based I/O. For any character-based I/O to be available, you must provide definitions for these two functions, using whatever facilities the hardware environment provides.

The creation of new I/O routines is based on these files:

- `putchar.c`, which serves as the low-level part of functions such as `printf`
- `getchar.c`, which serves as the low-level part of functions such as `scanf`.

The code example below shows how memory-mapped I/O could be used to write to a memory-mapped I/O device:

```
__no_init volatile unsigned char devIO @ 0x8000;

int putchar(int outChar)
{
    devIO = outChar;
    return outChar;
}
```

The exact address is a design decision. For example, it can depend on the selected processor variant.

For information about how to include your own modified version of `putchar` and `getchar` in your project build process, see *Overriding library modules*, page 61.

### FORMATTERS USED BY PRINTF AND SPRINTF

The `printf` and `sprintf` functions use a common formatter, called `_formatted_write`. The full version of `_formatted_write` is very large, and provides facilities not required in many embedded applications. To reduce the memory consumption, two smaller, alternative versions are also provided in the standard C library.

### **`_medium_write`**

The `_medium_write` formatter has the same functions as `_formatted_write`, except that floating-point numbers are not supported. Any attempt to use a `%f`, `%g`, `%G`, `%e`, or `%E` specifier will produce a runtime error:

```
FLOATS? wrong formatter installed!
```

`_medium_write` is considerably smaller than `_formatted_write`.

### **`_small_write`**

The `_small_write` formatter works in the same way as `_medium_write`, except that it supports only the `%%`, `%d`, `%o`, `%c`, `%s`, and `%x` specifiers for integer objects, and does not support field width or precision arguments. The size of `_small_write` is 10–15% that of `_formatted_write`.



#### **Specifying the printf formatter in the IDE**

- 1** Choose **Project>Options** and select the **General Options** category. Click the **Library options** tab.
- 2** Select the appropriate **Printf formatter** option, which can be either **Small**, **Medium**, or **Large**.



#### **Specifying the printf formatter from the command line**

To use the `_small_write` or `_medium_write` formatter, add the corresponding line in the linker command file:

```
-e_small_write=_formatted_write
```

or

```
-e_medium_write=_formatted_write
```

To use the full version, remove the line.

#### **Customizing printf**

For many embedded applications, `sprintf` is not required, and even `printf` with `_small_write` provides more facilities than are justified, considering the amount of memory it consumes. Alternatively, a custom output routine might be required to support particular formatting needs or non-standard output devices.

For such applications, a much reduced version of the `printf` function (without `sprintf`) is supplied in source form in the file `intwri.c`. This file can be modified to meet your requirements, and the compiled module inserted into the library in place of the original file; see *Overriding library modules*, page 61.

## FORMATTERS USED BY SCANF AND SSCANF

Similar to the `printf` and `sprintf` functions, `scanf` and `sscanf` use a common formatter, called `_formatted_read`. The full version of `_formatted_read` is very large, and provides facilities that are not required in many embedded applications. To reduce the memory consumption, an alternative smaller version is also provided.

### `_medium_read`

The `_medium_read` formatter has the same functions as the full version, except that floating-point numbers are not supported. `_medium_read` is considerably smaller than the full version.



### Specifying the `scanf` formatter in the IDE

- 1 Choose **Project>Options** and select the **General Options** category. Click the **Library options** tab.
- 2 Select the appropriate **Scanf formatter** option, which can be either **Medium** or **Large**.



### Specifying the read formatter from the command line

To use the `_medium_read` formatter, add this line in the linker command file:

```
-e_medium_read=_formatted_read
```

To use the full version, remove the line.

---

## System startup and termination

This section describes the actions the runtime environment performs during startup and termination of applications.

The code for handling startup and termination is located in the source files `cstartup.s26` and `low_level_init.c`, located in the `78k\src\lib` directory.

### SYSTEM STARTUP

When an application is initialized, several steps are performed:

- The custom function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations
- Static variables are initialized; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the remaining initialized variables
- The `main` function is called, which starts the application.

Note that the system startup code contains code for more steps than described here. The other steps are applicable to the DLIB runtime environment.

## SYSTEM TERMINATION

An application can terminate normally in two different ways:

- Return from the `main` function
- Call the `exit` function.

Because the ISO/ANSI C standard states that the two methods should be equivalent, the `cstartup` code calls the `exit` function if `main` returns. The parameter passed to the `exit` function is the return value of `main`. The default `exit` function is written in assembler.

When the application is built in debug mode, C-SPY stops when it reaches the special code label `?C_EXIT`.

An application can also exit by calling the `abort` function. The default function just calls `__exit` to halt the system, without performing any type of cleanup.

---

## Overriding default library modules

The IAR CLIB Library contains modules which you probably need to override with your own customized modules, for example for character-based I/O, without rebuilding the entire library. For information about how to override default library modules, see *Overriding library modules*, page 61, in the chapter *The DLIB runtime environment*.

---

## Customizing system initialization

For information about how to customize system initialization, see *Customizing system initialization*, page 66.

---

## C-SPY runtime interface

The low-level debugger interface is used for communication between the application being debugged and the debugger itself. The interface is simple: C-SPY will place breakpoints on certain assembler labels in the application. When code located at the special labels is about to be executed, C-SPY will be notified and can perform an action.

## THE DEBUGGER TERMINAL I/O WINDOW

When code at the labels `?C_PUTCHAR` and `?C_GETCHAR` is executed, data will be sent to or read from the debugger window.

For the `?C_PUTCHAR` routine, one character is taken from the output stream and written. If everything goes well, the character itself is returned, otherwise `-1` is returned.

When the label `?C_GETCHAR` is reached, C-SPY returns the next character in the input field. If no input is given, C-SPY waits until the user types some input and presses the Return key.

To make the Terminal I/O window available, the application must be linked with the XLINK option **With I/O emulation modules** selected. See the *IAR Embedded Workbench® IDE User Guide*.

## TERMINATION

The debugger stops executing when it reaches the special label `?C_EXIT`.

---

## Checking module consistency

For information about how to check module consistency, see *Checking module consistency*, page 78.



# Assembler language interface

When you develop an application for an embedded system, there might 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.

This chapter describes the available methods for this and some C alternatives, with their advantages and disadvantages. It also describes how to write functions in assembler language that work together with an application written in C or C++.

Finally, the chapter covers how you can implement support for call frame information in your assembler routines for use in the C-SPY® Call Stack window.

---

## Mixing C and assembler

The IAR C/C++ Compilers for 78K provide several ways to access low-level resources:

- Modules written entirely in assembler
- Intrinsic functions (the C alternative)
- Inline assembler.

It might be tempting to use simple inline assembler. However, you should carefully choose which method to use.

### INTRINSIC FUNCTIONS

The compilers provide a few predefined functions that allow direct access to low-level processor operations without having to use the assembler language. These functions are known as intrinsic functions. They can be very useful in, for example, time-critical routines.

An intrinsic function looks like a normal function call, but it is really a built-in function that the compiler recognizes. The intrinsic functions compile into inline code, either as a single instruction, or as a short sequence of instructions.

The advantage of an intrinsic function compared to using inline assembler is that the compiler has all necessary information to interface the sequence properly with register allocation and variables. The compiler also knows how to optimize functions with such sequences; something the compiler is unable to do with inline assembler sequences. The result is that you get the desired sequence properly integrated in your code, and that the compiler can optimize the result.

For detailed information about the available intrinsic functions, see the chapter *Intrinsic functions*.

## MIXING C AND ASSEMBLER MODULES

It is possible to write parts of your application in assembler and mix them with your C or C++ modules. This gives several benefits compared to using inline assembler:

- The function call mechanism is well-defined
- The code will be easy to read
- The optimizer can work with the C or C++ functions.

This causes some overhead in the form of a `CALL` and a `RET` instruction, and the compiler will regard some registers as scratch registers. However, the compiler will also assume that all scratch registers are destroyed by an inline assembler instruction. In many cases, the overhead of the extra instructions can be removed by the optimizer.

An important advantage is that you will have a well-defined interface between what the compiler produces and what you write in assembler. When using inline assembler, you will not have any guarantees that your inline assembler lines do not interfere with the compiler generated code.

When an application is written partly in assembler language and partly in C or C++, you are faced with several questions:

- How should the assembler code be written so that it can be called from C?
- Where does the assembler code find its parameters, and how is the return value passed back to the caller?
- How should assembler code call functions written in C?
- How are global C variables accessed from code written in assembler language?
- Why does not the debugger display the call stack when assembler code is being debugged?

The first issue is discussed in the section *Calling assembler routines from C*, page 94. The following two are covered in the section *Calling convention*, page 97.

The answer to the final question is that the call stack can be displayed when you run assembler code in the debugger. However, the debugger requires information about the

*call frame*, which must be supplied as annotations in the assembler source file. For more information, see *Call frame information*, page 104.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 94, and *Calling assembler routines from C++*, page 96, respectively.

## INLINE ASSEMBLER

It is possible to insert assembler code directly into a C or C++ function. The `asm` keyword inserts the supplied assembler statement in-line. The following example demonstrates the use of the `asm` keyword. This example also shows the risks of using inline assembler.

```
char sFlag;

void Foo()
{
    while (!Flag)
    {
        asm("MOV A,0xFF30");
        asm("MOV Flag,A");
    }
}
```

In this example, the assignment to the global variable `Flag` is not noticed by the compiler, which means the surrounding code cannot be expected to rely on the inline assembler statement.

The inline assembler instruction will simply be inserted at the given location in the program flow. The consequences or side-effects the insertion might have on the surrounding code are not taken into consideration. If, for example, registers or memory locations are altered, they might have to be restored within the sequence of inline assembler instructions for the rest of the code to work properly.

Inline assembler sequences have no well-defined interface with the surrounding code generated from your C or C++ code. This makes the inline assembler code fragile, and will possibly also become a maintenance problem if you upgrade the compiler in the future. There are also several limitations to using inline assembler:

- The compiler's various optimizations will disregard any effects of the inline sequences, which will not be optimized at all
- In general, assembler directives will cause errors or have no meaning. Data definition directives will however work as expected
- Alignment cannot be controlled; this means, for example, that `DC16` directives might be misaligned

- Auto variables cannot be accessed
- Static variables cannot be accessed
- Labels cannot be declared.

Inline assembler is therefore often best avoided. If no suitable intrinsic function is available, we recommend that you use modules written in assembler language instead of inline assembler, because the function call to an assembler routine normally causes less performance reduction.

---

## Calling assembler routines from C

An assembler routine that will be called from C must:

- Conform to the calling convention
- Have a `PUBLIC` entry-point label
- Be declared as external before any call, to allow type checking and optional promotion of parameters, as in these examples:

```
extern int foo(void);
or
extern int foo(int i, int j);
```

One way of fulfilling these requirements is to create skeleton code in C, compile it, and study the assembler list file.

### CREATING SKELETON CODE

The recommended way to create an assembler language routine with the correct interface is to start with an assembler language source file created by the C compiler. Note that you must create skeleton code for each function prototype.

The following example shows how to create skeleton code to which you can easily add the functional body of the routine. The skeleton source code only needs to declare the variables required and perform simple accesses to them. In this example, the assembler routine takes an `int` and a `char`, and then returns an `int`:

```
extern int gInt;
extern char gChar;
```

```

int Func(int arg1, char arg2)
{
    int locInt = arg1;
    gInt = arg1;
    gChar = arg2;
    return locInt;
}

int main()
{
    int locInt = gInt;
    gInt = Func(locInt, gChar);
    return 0;
}

```

**Note:** In this example we use a low optimization level when compiling the code to show local and global variable access. If a higher level of optimization is used, the required references to local variables could be removed during the optimization. The actual function declaration is not changed by the optimization level.

## COMPILING THE CODE



In the IDE, specify list options on file level. Select the file in the workspace window. Then choose **Project>Options**. In the **C/C++ Compiler** category, select **Override inherited settings**. On the **List** page, deselect **Output list file**, and instead select the **Output assembler file** option and its suboption **Include source**. Also, be sure to specify a low level of optimization.



Use these options to compile the skeleton code:

```
icc78k skeleton.c -lA .
```

The `-lA` option creates an assembler language output file including C or C++ source lines as assembler comments. The `.` (period) specifies that the assembler file should be named in the same way as the C or C++ module (`skeleton`), but with the filename extension `s26`. Also remember to specify the code model and processor core you are using (and for the 78K0R Compiler, the data model), a low level of optimization, and `-e` for enabling language extensions.

The result is the assembler source output file `skeleton.s26`.

**Note:** The `-lA` option creates a list file containing call frame information (CFI) directives, which can be useful if you intend to study these directives and how they are used. If you only want to study the calling convention, you can exclude the CFI directives from the list file. In the IDE, choose **Project>Options>C/C++ Compiler>List** and deselect the suboption **Include call frame information**. On the command line, use the option `-lB` instead of `-lA`. Note that CFI information must be included in the source code to make the C-SPY Call Stack window work.



## The output file

The output file contains the following important information:

- The calling convention
- The return values
- The global variables
- The function parameters
- How to create space on the stack (auto variables)
- Call frame information (CFI).

The `CFI` directives describe the call frame information needed by the Call Stack window in the debugger. For more information, see *Call frame information*, page 104.

---

## Calling assembler routines from C++

The C calling convention does not apply to C++ functions. Most importantly, a function name is not sufficient to identify a C++ function. The scope and the type of the function are also required to guarantee type-safe linkage, and to resolve overloading.

Another difference is that non-static member functions get an extra, hidden argument, the `this` pointer.

However, when using C linkage, the calling convention conforms to the C calling convention. An assembler routine can therefore be called from C++ when declared in this manner:

```
extern "C"
{
    int MyRoutine(int);
}
```

Memory access layout of non-PODs (“plain old data structures”) is not defined, and might change between compiler versions. Therefore, we do not recommend that you access non-PODs from assembler routines.

The following example shows how to achieve the equivalent to a non-static member function, which means that the implicit `this` pointer must be made explicit. It is also possible to “wrap” the call to the assembler routine in a member function. Use an inline member function to remove the overhead of the extra call—this assumes that function inlining is enabled:

```
class MyClass;

extern "C"
{
    void DoIt(MyClass *ptr, int arg);
}

class MyClass
{
public:
    inline void DoIt(int arg)
    {
        ::DoIt(this, arg);
    }
};
```

**Note:** Support for C++ names from assembler code is extremely limited. This means that:

- Assembler list files resulting from compiling C++ files cannot, in general, be passed through the assembler.
- It is not possible to refer to or define C++ functions that do not have C linkage in assembler.

---

## Calling convention

A calling convention is the way a function in a program calls another function. The compiler handles this automatically, but, if a function is written in assembler language, you must know where and how its parameters can be found, how to return to the program location from where it was called, and how to return the resulting value.

It is also important to know which registers an assembler-level routine must preserve. If the program preserves too many registers, the program might be ineffective. If it preserves too few registers, the result would be an incorrect program.

This section describes the calling convention used by the compilers. In the 78K0/78K0S Compiler, the default calling convention behaves slightly differently in the standard

code model compared in to the banked code model. These differences are also pointed out in this section.

These items are examined:

- Function declarations
- C and C++ linkage
- Preserved versus scratch registers
- Function entrance
- Function exit
- Return address handling.

At the end of the section, some examples are shown to describe the calling convention in practice.

## FUNCTION DECLARATIONS

In C, a function must be declared in order for the compiler to know how to call it. A declaration could look as follows:

```
int MyFunction(int first, char * second);
```

This means that the function takes two parameters: an integer and a pointer to a character. The function returns a value, an integer.

In the general case, this is the only knowledge that the compiler has about a function. Therefore, it must be able to deduce the calling convention from this information.

## USING C LINKAGE IN C++ SOURCE CODE

In C++, a function can have either C or C++ linkage. To call assembler routines from C++, it is easiest if you make the C++ function have C linkage.

This is an example of a declaration of a function with C linkage:

```
extern "C"
{
    int F(int);
}
```

It is often practical to share header files between C and C++. This is an example of a declaration that declares a function with C linkage in both C and C++:

```
#ifdef __cplusplus
extern "C"
{
#endif
```

```
int F(int);

#ifdef __cplusplus
}
#endif
```

## PRESERVED VERSUS SCRATCH REGISTERS

The general 78K CPU registers are divided into three separate sets, which are described in this section.

### Scratch registers

Any function is permitted to destroy the contents of a scratch register. If a function needs the register value after a call to another function, it must store it during the call, for example on the stack.

The following registers are used as scratch registers:

- The registers `AX` and `HL`, and, in 78K0R, `CS` and `ES`.
- Registers that are used as register parameters and for returning values by a function
- For banked functions, the register `DE` is also used as scratch register.

### Preserved registers

Preserved registers, on the other hand, are preserved across function calls. The called function can use the register for other purposes, but must save the value before using the register and restore it at the exit of the function.

For non-banked functions, the registers `BC`, and `DE` are preserved registers. For banked functions, the register `BC` is a preserved register.

### Special registers

The stack pointer register must at all times point to or below the last element on the stack. If an interrupt occurs, everything below the point the stack pointer points to will be destroyed.

## FUNCTION ENTRANCE

Parameters can be passed to a function using one of two basic methods: in registers or on the stack. It is much more efficient to use registers than to take a detour via memory, so the calling convention is designed to use registers as much as possible. Only a limited number of registers can be used for passing parameters; when no more registers are

available, the remaining parameters are passed on the stack. The parameters are also passed on the stack in these cases:

- Structure types: `struct`, `union`, and classes, except structs and unions of sizes 1, 2, and 4
- Unnamed parameters to variable length (variadic) functions; in other words, functions declared as `foo(param1, ...)`, for example `printf`.

This means, basically, that any 3-byte object and objects greater than 4 bytes are passed on the stack. For more details, see Table 29, *Registers used for passing parameters*.

### Hidden parameters

In addition to the parameters visible in a function declaration and definition, there can be hidden parameters:

If the function returns a structure, a pointer to the allocated space is passed as a hidden parameter. Hidden parameters are treated as ordinary parameters.

### Register parameters

The registers available for passing parameters are AX, BC, and DE, in that order. The register DE is not available for banked functions.

Parameters	Passed in registers
8-bit values	A, B, C, X, D, E
16-bit values	AX, BC, DE
24-bit values	C : AX (78K0/78K0S) Passed on the stack (78K0R)
32-bit values	BC : AX

Table 29: Registers used for passing parameters

The assignment of registers to parameters is a straightforward process. Traversing the parameters in strict order from left to right, the first parameter is assigned to the available register or registers. Should there be no suitable register available, the parameter is passed on the stack.

### Stack parameters and layout

Stack parameters are stored in the main memory, starting at the location pointed to by the stack pointer. Below the stack pointer (toward low memory) there is free space that the called function can use. The first stack parameter is stored at the location pointed to by the stack pointer. The next one is stored at the next location on the stack that is

divisible by two, etc. Objects on the stack are stored with alignment 2, which means that 1-byte objects (`char`) will occupy 2 bytes.

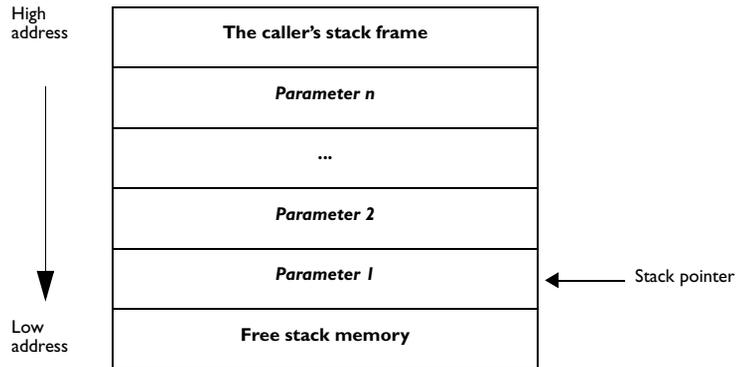


Figure 6: Stack image after the function call

## FUNCTION EXIT

A function can return a value to the function or program that called it, or it can have the return type `void`.

The return value of a function, if any, can be scalar (such as integers and pointers), floating-point, or a structure.

## Registers used for returning values

The registers available for returning values are `A`, `B`, `C`, and `X`.

Return values	Passed in registers
8-bit values	<code>A</code>
16-bit values	<code>AX</code>
24-bit values	<code>C : AX (78K0/78K0S)</code> <code>A : HL (78K0R)</code>
32-bit values	<code>BC : AX</code>

Table 30: Registers used for returning values

## Stack layout at function exit

It is the responsibility of the called function to clean the stack before function exit, in other words, responsible for cleaning the pushed parameters before returning.

There is one exception to this rule; the *caller* cleans the stack if the called function is a ...(*variable argument list*) function, for example

```
int printf (char const *, ...)
```

### Return address handling

A function written in assembler language should, when finished, return to the caller. At a function call, the return address is stored on the stack, for example:

```
CALL    func
```

Typically, a function returns by using the `RET` instruction, for example:

```
RET
```

There are two exceptions to this rule:

- For interrupt functions, the instruction `RETI` is used for a function return
- For the break interrupt, `BRK` instruction, the instruction `RETB` is used for a function return.

### EXAMPLES

The following section shows a series of declaration examples and the corresponding calling conventions. The complexity of the examples increases toward the end.

#### **Example 1**

Assume this function declaration:

```
int add1(int);
```

This function takes one parameter in the register `AX`, and the return value is passed back to its caller in the register `AX`.

This assembler routine is compatible with the declaration; it will return a value that is one number higher than the value of its parameter:

```
name    return
rseg    CODE:CODE
incw    AX
ret
end
```

**Example 2**

This example shows how structures are passed on the stack. Assume these declarations:

```
struct MyStruct
{
    int mA;
};

int MyFunction(struct MyStruct x, int y);
```

The calling function must reserve 8 bytes on the top of the stack and copy the contents of the `struct` to that location. The integer parameter `y` is passed in the register `AX`. The return value is passed back to its caller in the register `AX`.

**Example 3**

The function below will return a structure of type `struct`.

```
struct MyStruct
{
    int mA;
    char mC;
};

struct MyStruct MyFunction(int x);
```

It is the responsibility of the calling function to allocate a memory location for the return value and pass a pointer to it as a hidden first parameter. If you are using the near data model, the pointer to the location where the return value should be stored is passed in `AX`. The caller assumes that these registers remain untouched. The parameter `x` is passed in `BC`.

Assume that the function instead was declared to return a pointer to the structure:

```
struct MyStruct *MyFunction(int x);
```

In this case, the return value is a pointer, so there is no hidden parameter. The parameter `x` is passed in `AX`, and the return value is returned in `AX`.

**FUNCTION DIRECTIVES**

**Note:** This type of directive is primarily intended to support static overlay, a feature which is useful in some smaller microcontrollers. The IAR C/C++ Compiler for 78K does not use static overlay, because it has no use for it.

The function directives `FUNCTION`, `ARGFRAME`, `LOCFRAME`, and `FUNCALL` are generated by the compiler to pass information about functions and function calls to the IAR XLINK Linker. These directives can be seen if you use the compiler option **Assembler file** (`-lA`) to create an assembler list file.

For reference information about the function directives, see the *IAR Assemblers Reference Guide for 78K*.

---

## Call frame information

When you debug an application using C-SPY, you can view the *call stack*, that is, the chain of functions that called the current function. To make this possible, the compiler supplies debug information that describes the layout of the call frame, in particular information about where the return address is stored.

If you want the call stack to be available when debugging a routine written in assembler language, you must supply equivalent debug information in your assembler source using the assembler directive `CFI`. This directive is described in detail in the *IAR Assemblers Reference Guide for 78K*.

### CFI DIRECTIVES

The `CFI` directives provide C-SPY with information about the state of the calling function(s). Most important of this is the return address, and the value of the stack pointer at the entry of the function or assembler routine. Given this information, C-SPY can reconstruct the state for the calling function, and thereby unwind the stack.

A full description about the calling convention might require extensive call frame information. In many cases, a more limited approach will suffice.

When describing the call frame information, the following three components must be present:

- A *names block* describing the available resources to be tracked
- A *common block* corresponding to the calling convention
- A *data block* describing the changes that are performed on the call frame. This typically includes information about when the stack pointer is changed, and when permanent registers are stored or restored on the stack.

This table lists all the resources defined in the names block used by the compiler:

Resource	Description
CFA	The call frames of the regular stack and of the interrupt stack, respectively
A, B, C, D, E, H, L, X	Normal registers
CS_REG	Bit 16–19 in the code address space (78K0R only)
ES_REG	Bit 16–19 in the data address space (78K0R only)

Table 31: Call frame information resources defined in a names block

Resource	Description
SP	The stack pointer for 78K0/78K0S
?SP20	The stack pointer for 78K0R
?RET	For non-banked return addresses, and part of banked return addresses (a 16-bit address for 78K0/78K0S, a 20-bit address for 78K0R)
?RET8	For the bank number in banked return addresses, otherwise undefined (an 8-bit address—78K0/78K0S only)
?RET24	A concatenation of ?RET and ?RET8 used for banked return addresses (a 24-bit address—78K0/78K0S only)
W0-W127	The wrkseg area

Table 31: Call frame information resources defined in a names block (Continued)

## CREATING ASSEMBLER SOURCE WITH CFI SUPPORT

The recommended way to create an assembler language routine that handles call frame information correctly is to start with an assembler language source file created by the compiler.

- 1 Start with suitable C source code, for example:

```
int F(int);
int cfiExample(int i)
{
    return i + F(i);
}
```

- 2 Compile the C source code, and make sure to create a list file that contains call frame information—the CFI directives.



On the command line, use the option `-lA`.



In the IDE, choose **Project>Options>C/C++ Compiler>List** and make sure the suboption **Include call frame information** is selected.

For the source code in this example, the list file looks like this:

```
NAME Cfi

PUBLIC cfiExample
FUNCTION cfiExample,021203H
ARGFRAME CSTACK, 0, STACK
LOCFRAME CSTACK, 2, STACK

CFI Names cfiNames0
CFI StackFrame CFA SP DATA
```

```

CFI Resource A:8, X:8, B:8, C:8, D:8, E:8, H:8, L:8
CFI VirtualResource ?RET24:24
CFI Resource ?RET8:8, SP:16
CFI EndNames cfiNames0

CFI Common cfiCommon0 Using cfiNames0
CFI CodeAlign 1
CFI DataAlign 1
CFI ReturnAddress ?RET24 CODE
CFI CFA SP+2
CFI A Undefined
CFI X Undefined
CFI B SameValue
CFI C SameValue
CFI D SameValue
CFI E SameValue
CFI H Undefined
CFI L Undefined
CFI ?RET24 add(load(2, DATA, sub(CFA, 2)), mul(65536,
?RET8))
CFI ?RET8 SameValue
CFI EndCommon cfiCommon0

EXTERN F
FUNCTION F,0202H
ARGFRAME CSTACK, 2, STACK

RSEG CODE:CODE:NOROOT(0)
cfiExample:
CFI Block cfiBlock0 Using cfiCommon0
CFI Function cfiExample
FUNCALL cfiExample, F
LOCFRAME CSTACK, 2, STACK
ARGFRAME CSTACK, 2, STACK
PUSH      DE
CFI E Frame(CFA, -4)
CFI D Frame(CFA, -3)
CFI CFA SP+4
MOVW      DE,AX
MOVW      AX,DE
CALL      F
XCH       A,X
ADD       A,E
XCH       A,X
ADDC      A,D
POP       DE
CFI E SameValue

```

```
CFI D SameValue
CFI CFA SP+2
RET
CFI EndBlock cfiBlock0

END
```



# Using C++

IAR Systems supports two levels of the C++ language: The industry-standard Embedded C++ and IAR Extended Embedded C++. They are described in this chapter.

---

## Overview

Embedded C++ is a subset of the C++ programming language which is intended for embedded systems programming. It was defined by an industry consortium, the Embedded C++ Technical Committee. Performance and portability are particularly important in embedded systems development, which was considered when defining the language.

### STANDARD EMBEDDED C++

The following C++ features are supported:

- Classes, which are user-defined types that incorporate both data structure and behavior; the essential feature of inheritance allows data structure and behavior to be shared among classes
- Polymorphism, which means that an operation can behave differently on different classes, is provided by virtual functions
- Overloading of operators and function names, which allows several operators or functions with the same name, provided that their argument lists are sufficiently different
- Type-safe memory management using the operators `new` and `delete`
- Inline functions, which are indicated as particularly suitable for inline expansion.

C++ features that are excluded are those that introduce overhead in execution time or code size that are beyond the control of the programmer. Also excluded are late additions to the ISO/ANSI C++ standard. This is because they represent potential portability problems, due to that few development tools support the standard. Embedded C++ thus offers a subset of C++ which is efficient and fully supported by existing development tools.

Standard Embedded C++ lacks these features of C++:

- Templates
- Multiple and virtual inheritance
- Exception handling

- Runtime type information
- New cast syntax (the operators `dynamic_cast`, `static_cast`, `reinterpret_cast`, and `const_cast`)
- Namespaces
- The `mutable` attribute.

The exclusion of these language features makes the runtime library significantly more efficient. The Embedded C++ library furthermore differs from the full C++ library in that:

- The standard template library (STL) is excluded
- Streams, strings, and complex numbers are supported without the use of templates
- Library features which relate to exception handling and runtime type information (the headers `except`, `stdexcept`, and `typeinfo`) are excluded.

**Note:** The library is not in the `std` namespace, because Embedded C++ does not support namespaces.

## EXTENDED EMBEDDED C++

IAR Systems' Extended EC++ is a slightly larger subset of C++ which adds these features to the standard EC++:

- Full template support
- Namespace support
- The `mutable` attribute
- The cast operators `static_cast`, `const_cast`, and `reinterpret_cast`.

All these added features conform to the C++ standard.

To support Extended EC++, this product includes a version of the standard template library (STL), in other words, the C++ standard chapters utilities, containers, iterators, algorithms, and some numerics. This STL is tailored for use with the Extended EC++ language, which means no exceptions, and no support for runtime type information (`rtti`). Moreover, the library is not in the `std` namespace.

**Note:** A module compiled with Extended EC++ enabled is fully link-compatible with a module compiled without Extended EC++ enabled.

## ENABLING C++ SUPPORT



In the compiler, the default language is C. To be able to compile files written in Embedded C++, you must use the `--ec++` compiler option. See `--ec++`, page 164. You must also use the IAR DLIB runtime library.

To take advantage of *Extended* Embedded C++ features in your source code, you must use the `--eec++` compiler option. See `--eec++`, page 164.



To set the equivalent option in the IDE, choose **Project>Options>C/C++ Compiler>Language**.

---

## Feature descriptions

When you write C++ source code for the IAR C/C++ Compilers for 78K, you must be aware of some benefits and some possible quirks when mixing C++ features—such as classes, and class members—with IAR language extensions, such as IAR-specific attributes.

### CLASSES

A class type `class` and `struct` in C++ can have static and non-static data members, and static and non-static function members. The non-static function members can be further divided into virtual function members, non-virtual function members, constructors, and destructors. For the static data members, static function members, and non-static non-virtual function members the same rules apply as for statically linked symbols outside of a class. In other words, they can have any applicable IAR-specific type, memory, and object attribute.

The non-static virtual function members can have any applicable IAR-specific type, memory, and object attribute as long as a pointer to the member function can be implicitly converted to the default function pointer type. The constructors, destructors, and non-static data members cannot have any IAR attributes.

The location operator `@` can be used on static data members and on any type of function members.

For further information about attributes, see *Type qualifiers*, page 189.

#### Example

```
class MyClass
{
public:
    // Locate a static variable in saddr memory at address FFD0
    static __saddr __no_init int mI @ FFD0;

    // Locate a static function in far memory
    static __far void F();

    // Locate a function in far memory
    __far void G();
```

```

// Locate a virtual function in far memory
virtual __far void H();

// Locate a virtual function into SPECIAL
virtual void M() const volatile @ "SPECIAL";
};

```

### The this pointer (78K0R only)

The `this` pointer used for referring to a class object or calling a member function of a class object will by default have the data memory attribute for the default data pointer type. This means that such a class object can only be defined to reside in memory from which pointers can be implicitly converted to a default data pointer. This restriction might also apply to objects residing on a stack, for example temporary objects and auto objects.

### Class memory (78K0R only)

To compensate for this limitation, a class can be associated with a *class memory type*. The class memory type changes:

- the `this` pointer type in all member functions, constructors, and destructors into a pointer to class memory
- the default memory for static storage duration variables—that is, not auto variables—of the class type, into the specified class memory
- the pointer type used for pointing to objects of the class type, into a pointer to class memory.

### Example

```

class __far C
{
public:
    void MyF();           // Has a this pointer of type C __far *
    void MyF() const;    // Has a this pointer of type
                        // C __far const *
    C();                 // Has a this pointer pointing into far
                        // memory
    C(C const &);        // Takes a parameter of type C __far
                        // const & (also true of generated copy
                        // constructor)

    int mI;
};

```

```

C Ca;                // Resides in far memory instead of the
                    // default memory
C __near Cb;        // Resides in near memory, the 'this'
                    // pointer still points into far memory

void MyH()
{
    C cd;            // Resides on the stack
}

C *Cp1;             // Creates a pointer to far memory
C __near *Cp2;     // Creates a pointer to near memory

```

Whenever a class type associated with a class memory type, like `C`, must be declared, the class memory type must be mentioned as well:

```
class __far C;
```

Also note that class types associated with different class memories are not compatible types.

A built-in operator returns the class memory type associated with a class, `__memory_of(class)`. For instance, `__memory_of(C)` returns `__far`.

When inheriting, the rule is that it must be possible to convert implicitly a pointer to a subclass into a pointer to its base class. This means that a subclass can have a *more* restrictive class memory than its base class, but not a *less* restrictive class memory.

```

class __far D : public C
{ // OK, same class memory
public:
    void MyG();
    int mJ;
};

class __near E : public C
{ // OK, near memory is inside far
public:
    void MyG() // Has a this pointer pointing into near memory
    {
        MyF(); // Gets a this pointer into far memory
    }
    int mJ;
};

class F : public C
{ // OK, will be associated with same class memory as C
public:
    void MyG();
}

```

```
int mJ;
};
```

A `new` expression on the class will allocate memory in the heap associated with the class memory. A `delete` expression will naturally deallocate the memory back to the same heap. To override the default `new` and `delete` operator for a class, declare

```
void *operator new(size_t);
void operator delete(void *);
```

as member functions, just like in ordinary C++.

For more information about memory types, see *Memory types*, page 13.

## FUNCTION TYPES

A function type with `extern "C"` linkage is compatible with a function that has C++ linkage.

### Example

```
extern "C"
{
    typedef void (*FpC)(void);    // A C function typedef
}

typedef void (*FpCpp)(void);    // A C++ function typedef

FpC F1;
FpCpp F2;
void MyF(FpC);

void MyG()
{
    MyF(F1);                      // Always works
    MyF(F2);                      // FpCpp is compatible with FpC
}
```

## NEW AND DELETE OPERATORS (78K0R ONLY)

For the 78K0R Compiler, there are operators for `new` and `delete` for each memory that can have a heap, that is, near and far memory.

```
// Assumes that there is a heap in both __near and __far memory
void __far *operator new __far(__far_size_t);
void __near *operator new __near (__near_size_t);
void operator delete(void __far *);
void operator delete(void __near *);
```

```
// And correspondingly for array new and delete operators
void __far *operator new[] __far(__far_size_t);
void __near *operator new[] __near (__near_size_t);
void operator delete[](void __far *);
void operator delete[](void __near *);
```

Use this syntax if you want to override both global and class-specific `operator new` and `operator delete` for any data memory.

Note that there is a special syntax to name the `operator new` functions for each memory, while the naming for the `operator delete` functions relies on normal overloading.

### New and delete expressions

A new expression calls the `operator new` function for the memory of the type given. If a class, struct, or union type with a class memory is used, the class memory will determine the `operator new` function called. For example,

```
void MyF()
{
    // Calls operator new __near(__near_size_t)
    int __near *p = new __near int;

    // Calls operator new __near(__near_size_t)
    int __near *q = new int __near;

    // Calls operator new[] __near(__near_size_t)
    int __near *r = new __near int[10];

    // Calls operator new __far(__far_size_t)
    class __far S
    {
    };
    S *s = new S;

    // Calls operator delete(void __near *)
    delete p;
    // Calls operator delete(void __far *)
    delete s;

    int __far *t = new __near int;
    delete t; // Error: Causes a corrupt heap
}
```

Note that the pointer used in a `delete` expression must have the correct type, that is, the same type as that returned by the `new` expression. If you use a pointer to the wrong memory, the result might be a corrupt heap.

## TEMPLATES

*Extended EC++* supports templates according to the C++ standard, except for the support of the `export` keyword. The implementation uses a two-phase lookup which means that the keyword `typename` must be inserted wherever needed. Furthermore, at each use of a template, the definitions of all possible templates must be visible. This means that the definitions of all templates must be in include files or in the actual source file.

### Templates and data memory attributes (78K0R only)

For data memory attributes to work as expected in templates, two elements of the standard C++ template handling were changed—class template partial specialization matching and function template parameter deduction.

In *Extended Embedded C++*, the class template partial specialization matching algorithm works like this:

*When a pointer or reference type is matched against a pointer or reference to a template parameter type, the template parameter type will be the type pointed to, stripped of any data memory attributes, if the resulting pointer or reference type is the same.*

In *Extended Embedded C++*, the function template parameter deduction algorithm works like this:

*When function template matching is performed and an argument is used for the deduction; if that argument is a pointer to a memory that can be implicitly converted to a default pointer, do the parameter deduction as if it was a default pointer.*

*When an argument is matched against a reference, do the deduction as if the argument and the parameter were both pointers.*

For templates that are matched using this modified algorithm, it is impossible to get automatic generation of special code for pointers to *small* memory types. For *large* and “other” memory types (memory that cannot be pointed to by a default pointer) it is possible. To make it possible to write templates that are fully memory-aware—in the rare cases where this is useful—use the `#pragma basic_template_matching` directive in front of the template function declaration. That template function will then match without the modifications described above.

### Example

```
// We assume that __far is the memory type of the default
// pointer.
#pragma basic_template_matching
template<typename T> void fun(T *);
```

```
void MyF()
{
    fun((int __near *) 0); // T = int __near
}
```

### Non-type template parameters

It is allowed to have a reference to a memory type as a template parameter, even if pointers to that memory type are not allowed.

#### Example

```
extern int __sfr X;

template<__sfr int &y>
void Foo()
{
    y = 17;
}

void Bar()
{
    Foo<X>();
}
```

### The standard template library

The STL (standard template library) delivered with the product is tailored for Extended EC++, as described in *Extended Embedded C++*, page 110.

In the 78K0R Compiler, the containers in the STL, like `vector` and `map`, are memory attribute aware. This means that a container can be declared to reside in a specific memory type which has the following consequences:

- The container itself will reside in the chosen memory
- Allocations of elements in the container will use a heap for the chosen memory
- All references inside it use pointers to the chosen memory.

#### Example

```
#include <vector>

vector<int> D; // D placed in default
              // memory, using the default
              // heap, uses default
              // pointers
vector<int __near> __near X; // X placed in near memory,
```

```

// heap allocation from
// near, uses pointers to
// near memory
vector<int __far> __near Y; // Y placed in near memory,
// heap allocation from
// far, uses pointers to
// far memory

```

Note that `map<key, T>`, `multimap<key, T>`, `hash_map<key, T>`, and `hash_multimap<key, T>` all use the memory of `T`. This means that the `value_type` of these collections will be `pair<key, const T> mem` where `mem` is the memory type of `T`. Supplying a key with a memory type is not useful.

### Example

Note that two containers that only differ by the data memory attribute they use cannot be assigned to each other. Instead, the templated `assign` member method must be used.

```

#include <vector>

vector<int __near> X;
vector<int __far> Y;

void MyF()
{
    // The templated assign member method will work
    X.assign(Y.begin(), Y.end());
    Y.assign(X.begin(), X.end());
}

```

### STL and the IAR C-SPY® Debugger

C-SPY has built-in display support for the STL containers. The logical structure of containers is presented in the watch views in a comprehensive way that is easy to understand and follow.



**Note:** To be able to watch STL containers with many elements in a comprehensive way, the **STL container expansion** option—available by choosing **Tools>Options>Debugger**—is set to display only a few items at first.

### VARIANTS OF CASTS

In Extended EC++ these additional C++ cast variants can be used:

```
const_cast<t2>(t), static_cast<t2>(t), reinterpret_cast<t2>(t).
```

## MUTABLE

The `mutable` attribute is supported in Extended EC++. A `mutable` symbol can be changed even though the whole class object is `const`.

## NAMESPACE

The namespace feature is only supported in *Extended* EC++. This means that you can use namespaces to partition your code. Note, however, that the library itself is not placed in the `std` namespace.

## THE STD NAMESPACE

The `std` namespace is not used in either standard EC++ or in Extended EC++. If you have code that refers to symbols in the `std` namespace, simply define `std` as nothing; for example:

```
#define std
```

You must make sure that identifiers in your application do not interfere with identifiers in the runtime library.

## POINTER TO MEMBER FUNCTIONS

A pointer to a member function can only contain a default function pointer, or a function pointer that can implicitly be casted to a default function pointer. To use a pointer to a member function, make sure that all functions that should be pointed to reside in the default memory or a memory contained in the default memory.

### Example

```
class X
{
public:
    __far_func void F();
};

void (__far_func X::*PMF)(void) = &X::F;
```

## USING INTERRUPTS AND EC++ DESTRUCTORS

If interrupts are enabled and the interrupt functions use static class objects that need to be destroyed, there might be problems if the interrupt occurs during or after the application exits. If an interrupt occurs after the static class object was destroyed, the application will not work properly.

To avoid this, make sure that interrupts are disabled when returning from `main` or when calling `exit` or `abort`. To do this, call the intrinsic function `__disable_interrupt`.

---

## C++ language extensions

When you use the compiler in C++ mode and enable IAR language extensions, the following C++ language extensions are available in the compiler:

- In a friend declaration of a class, the `class` keyword can be omitted, for example:

```
class B;
class A
{
    friend B;          //Possible when using IAR language
                    //extensions
    friend class B; //According to standard
};
```

- Constants of a scalar type can be defined within classes, for example:

```
class A
{
    const int mSize = 10; //Possible when using IAR language
                        //extensions
    int mArr[mSize];
};
```

According to the standard, initialized static data members should be used instead.

- In the declaration of a class member, a qualified name can be used, for example:

```
struct A
{
    int A::F(); // Possible when using IAR language extensions
    int G();   // According to standard
};
```

- It is permitted to use an implicit type conversion between a pointer to a function with C linkage (`extern "C"`) and a pointer to a function with C++ linkage (`extern "C++"`), for example:

```
extern "C" void F(); // Function with C linkage
void (*PF)()        // PF points to a function with C++ linkage
    = &F; // Implicit conversion of function pointer.
```

According to the standard, the pointer must be explicitly converted.

- If the second or third operands in a construction that contains the `?` operator are string literals or wide string literals (which in C++ are constants), the operands can be implicitly converted to `char *` or `wchar_t *`, for example:

```
bool X;

char *P1 = X ? "abc" : "def";           //Possible when using IAR
                                         //language extensions
char const *P2 = X ? "abc" : "def";    //According to standard
```

- Default arguments can be specified for function parameters not only in the top-level function declaration, which is according to the standard, but also in `typedef` declarations, in pointer-to-function function declarations, and in pointer-to-member function declarations.
- In a function that contains a non-static local variable and a class that contains a non-evaluated expression (for example a `sizeof` expression), the expression can reference the non-static local variable. However, a warning is issued.

**Note:** If you use any of these constructions without first enabling language extensions, errors are issued.



# Efficient coding for embedded applications

For embedded systems, the size of the generated code and data is very important, because using smaller external memory or on-chip memory can significantly decrease the cost and power consumption of a system.

The topics discussed are:

- Selecting data types
- Controlling data and function placement in memory
- Controlling compiler optimizations
- Writing efficient code.

As a part of this, the chapter also demonstrates some of the more common mistakes and how to avoid them, and gives a catalog of good coding techniques.

---

## Selecting data types

For efficient treatment of data, you should consider the data types used and the most efficient placement of the variables.

### USING EFFICIENT DATA TYPES

The data types you use should be considered carefully, because this can have a large impact on code size and code speed.

- Use small and unsigned data types, (`unsigned char` and `unsigned short`) unless your application really requires signed values.
- Bitfields with sizes other than 1 bit should be avoided because they will result in inefficient code compared to bit operations.
- When using arrays, it is more efficient if the type of the index expression matches the index type of the memory of the array.

- Using floating-point types on a microprocessor without a math co-processor is very inefficient, both in terms of code size and execution speed.
- Declaring a pointer parameter to point to `const` data might open for better optimizations in the calling function.

For details about representation of supported data types, pointers, and structures types, see the chapter *Data representation*.

## FLOATING-POINT TYPES

Using floating-point types on a microprocessor without a math coprocessor is very inefficient, both in terms of code size and execution speed. Thus, you should consider replacing code that uses floating-point operations with code that uses integers, because these are more efficient.

The compilers only support the 32-bit floating-point format. The 64-bit floating-point format is not supported. The `double` type will be treated as a `float` type.

## ALIGNMENT OF ELEMENTS IN A STRUCTURE

The 78K microcontroller requires that data in memory must be aligned. Each element in a structure must be aligned according to its specified type requirements. This means that the compiler might need to insert *pad bytes* to keep the alignment at 2 (unless the option `--disable_data_alignment` is used).

There are two reasons why this can be considered a problem:

- Due to external demands; for example, network communication protocols are usually specified in terms of data types with no padding in between
- You need to save data memory.

For information about alignment requirements, see *Alignment*, page 181.

There are three ways to solve the problem:

- If you are using the CLIB runtime environment, use the **Disable data alignment** (`--disable_data_alignment`) option. This is an easy way to remove the problem with the drawback that each access to an unaligned element in the structure will use more code. It is also a drawback that the option must be used on all files in the project, and that all data types are affected. It cannot be used with the DLIB environment. For more information, see *--disable\_data\_alignment*, page 161.
- Use the `#pragma pack` directive for a tighter layout of the structure. The drawback is that each access to an unaligned element in the structure will use more code.
- Write your own customized functions for *packing* and *unpacking* structures. This is a more portable way, which will not produce any more code apart from your

functions. The drawback is the need for two views on the structure data—packed and unpacked.

For further details about the `#pragma pack` directive, see *pack*, page 227.

## ANONYMOUS STRUCTS AND UNIONS

When a structure or union is declared without a name, it becomes anonymous. The effect is that its members will only be seen in the surrounding scope.

Anonymous structures are part of the C++ language; however, they are not part of the C standard. In the IAR C/C++ Compilers for 78K they can be used in C if language extensions are enabled.



In the IDE, language extensions are enabled by default.



Use the `-e` compiler option to enable language extensions. See *-e*, page 164, for additional information.

### Example

In this example, the members in the anonymous `union` can be accessed, in function `F`, without explicitly specifying the `union` name:

```
struct S
{
    char mTag;
    union
    {
        long mL;
        float mF;
    };
} St;

void F(void)
{
    St.mL = 5;
}
```

The member names must be unique in the surrounding scope. Having an anonymous struct or union at file scope, as a global, external, or static variable is also allowed. This could for instance be used for declaring I/O registers, as in this example:

```
__no_init __sfr volatile
union
{
    unsigned char IOPORT;
    struct
    {
        unsigned char Way: 1;
        unsigned char Out: 1;
    };
} @ 0xFF00;

/* Here the variables are used*/

void Test(void)
{
    IOPORT = 0;
    Way = 1;
    Out = 1;
}
```

This declares an I/O register byte `IOPORT` at address 0. The I/O register has 2 bits declared, `Way` and `Out`. Note that both the inner structure and the outer union are anonymous.

Anonymous structures and unions are implemented in terms of objects named after the first field, with a prefix `_A_` to place the name in the implementation part of the namespace. In this example, the anonymous union will be implemented through an object named `_A_IOPORT`.

---

## Controlling data and function placement in memory

The compilers provide different mechanisms for controlling placement of functions and data objects in memory. To use memory efficiently, you should be familiar with these mechanisms to know which one is best suited for different situations. You can use:

- Code and data models

Use the different compiler options for code and data models, respectively, to take advantage of the different addressing modes available for the microcontroller and thereby also place functions and data objects in different parts of memory. To read more about data and code models, see *Data models (78K0R only)*, page 12, and *Code models and memory attributes for function storage*, page 21, respectively.

- Memory attributes
 

Use memory attributes to override the default addressing mode and placement of individual functions and data objects. To read more about memory attributes for data and functions, see *Using data memory attributes*, page 14, and *Using function memory attributes*, page 22, respectively.
- The @ operator and the #pragma location directive for absolute placement
 

Use the @ operator or the #pragma location directive to place individual global and static variables at absolute addresses. The variables must be declared either \_\_no\_init or const. This is useful for individual data objects that must be located at a fixed address, for example variables with external requirements, or for populating any hardware tables similar to interrupt vector tables. Note that it is not possible to use this notation for absolute placement of individual functions.
- The @ operator and the #pragma location directive for segment placement
 

Use the @ operator or the #pragma location directive to place groups of functions or global and static variables in named segments, without having explicit control of each object. The variables must be declared either \_\_no\_init or const. The segments can, for example, be placed in specific areas of memory, or initialized or copied in controlled ways using the segment begin and end operators. This is also useful if you want an interface between separately linked units, for example an application project and a boot loader project. Use named segments when absolute control over the placement of individual variables is not needed, or not useful.
- The --code\_segment option
 

Use the --code\_segment option to place functions in named segments, which is useful, for example, if you want to direct them to different fast or slow memories. To read more about the --code\_segment option, see *--code\_segment*, page 156 .

At compile time, data and functions are placed in different segments as described in *Data segments*, page 43, and *Code segments*, page 49, respectively. At link time, one of the most important functions of the linker is to assign load addresses to the various segments used by the application. All segments, except for the segments holding absolute located data, are automatically allocated to memory according to the specifications of memory ranges in the linker command file, as described in *Placing segments in memory*, page 40.

## DATA PLACEMENT AT AN ABSOLUTE LOCATION

The @ operator, alternatively the #pragma location directive, can be used for placing global and static variables at absolute addresses. The variables must be declared using one of these combinations of keywords:

- \_\_no\_init
- \_\_no\_init and const (without initializers)

- `const` (with initializers).

To place a variable at an absolute address, the argument to the `@` operator and the `#pragma location` directive should be a literal number, representing the actual address. The absolute location must fulfill the alignment requirement for the variable that should be located.

**Note:** A variable placed in an absolute location should be defined in an include file, to be included in every module that uses the variable. An unused definition in a module will be ignored. A normal `extern` declaration—one that does not use an absolute placement directive—can refer to a variable at an absolute address; however, optimizations based on the knowledge of the absolute address cannot be performed.

### Examples

In this example, a `__no_init` declared variable is placed at an absolute address. This is useful for interfacing between multiple processes, applications, etc:

```
__no_init volatile char alpha @ 0x2000; /* OK */
```

These examples contain two `const` declared objects. The first one is not initialized, and the second one is initialized to a specific value. Both objects are placed in ROM. This is useful for configuration parameters, which are accessible from an external interface. Note that in the second case, the compiler is not obliged to actually read from the variable, because the value is known.

```
#pragma location=0x2002
__no_init const int beta;           /* OK */

const int gamma @ 0x2004 = 3;      /* OK */
```

In the first case, the value is not initialized by the compiler; the value must be set by other means. The typical use is for configurations where the values are loaded to ROM separately, or for special function registers that are read-only.

These examples show incorrect usage:

```
int delta @ 0x2006;                /* Error, neither */
                                   /* "__no_init" nor "const".*/

__no_init int epsilon @ 0x2007;   /* Error, misaligned. */
```

### C++ considerations

In C++, module scoped `const` variables are static (module local), whereas in C they are global. This means that each module that declares a certain `const` variable will contain

a separate variable with this name. If you link an application with several such modules all containing (via a header file), for instance, the declaration:

```
volatile const __no_init int x @ 0x100;          /* Bad in C++ */
```

the linker will report that more than one variable is located at address 0x100.

To avoid this problem and make the process the same in C and C++, you should declare these variables `extern`, for example:

```
/* The extern keyword makes x public. */
extern volatile const __no_init int x @ 0x100;
```

**Note:** C++ static member variables can be placed at an absolute address just like any other static variable.

## DATA AND FUNCTION PLACEMENT IN SEGMENTS

This method can be used for placing data or functions in named segments other than default:

- The `@` operator, alternatively the `#pragma location` directive, can be used for placing individual variables or individual functions in named segments. The named segment can either be a predefined segment, or a user-defined segment. The variables must be declared either `__no_init` or `const`. If declared `const`, they can have initializers.
- The `--code_segment` option can be used for placing functions, which are parts of the whole compilation unit, in named segments.

C++ static member variables can be placed in named segments just like any other static variable.

If you use your own segments, in addition to the predefined segments, the segments must also be defined in the linker command file using the `-Z` or the `-P` segment control directives.

**Note:** Take care when explicitly placing a variable or function in a predefined segment other than the one used by default. This is useful in some situations, but incorrect placement can result in anything from error messages during compilation and linking to a malfunctioning application. Carefully consider the circumstances; there might be strict requirements on the declaration and use of the function or variable.

The location of the segments can be controlled from the linker command file.

For more information about segments, see the chapter *Segment reference*.

### Examples of placing variables in named segments

In the following three examples, a data object is placed in a user-defined segment. The segment will be allocated in default memory depending on the used data model.

```
__no_init int alpha @ "NOINIT";    /* OK */

#pragma location="CONSTANTS"
const int beta;                    /* OK */

const int gamma @ "CONSTANTS" = 3; /* OK */
```

To override the default segment allocation, you can explicitly specify a memory attribute other than the default:

```
__far __no_init int alpha @ "NOINIT"; /* Placed in far*/
```

This example shows incorrect usage:

```
int delta @ "NOINIT";              /* Error, neither */
                                   /* "__no_init" nor "const" */
```

### Examples of placing functions in named segments

```
void f(void) @ "FUNCTIONS";

void g(void) @ "FUNCTIONS"
{
}

#pragma location="FUNCTIONS"
void h(void);
```

To override the default segment allocation, you can explicitly specify a memory attribute other than the default:

```
__far_func void f(void) @ "FUNCTIONS";
```

---

## Controlling compiler optimizations

The compiler performs many transformations on your application to generate the best possible code. Examples of such transformations are storing values in registers instead of memory, removing superfluous code, reordering computations in a more efficient order, and replacing arithmetic operations by cheaper operations.

The linker should also be considered an integral part of the compilation system, because some optimizations are performed by the linker. For instance, all unused functions and variables are removed and not included in the final output.

## SCOPE FOR PERFORMED OPTIMIZATIONS

You can decide whether optimizations should be performed on your whole application or on individual files. By default, the same types of optimizations are used for an entire project, but you should consider using different optimization settings for individual files. For example, put code that must execute very quickly into a separate file and compile it for minimal execution time, and the rest of the code for minimal code size. This will give a small program, which is still fast enough where it matters.

You can also exclude individual functions from the performed optimizations. The `#pragma optimize` directive allows you to either lower the optimization level, or specify another type of optimization to be performed. Refer to *optimize*, page 226, for information about the pragma directive.

### Multi-file compilation units

In addition to applying different optimizations to different source files or even functions, you can also decide what a compilation unit consists of—one or several source code files.

By default, a compilation unit consists of one source file, but you can also use multi-file compilation to make several source files in a compilation unit. The advantage is that interprocedural optimizations such as inlining, cross call, and cross jump have more source code to work on. Ideally, the whole application should be compiled as one compilation unit. However, for large applications this is not practical because of resource restrictions on the host computer. For more information, see *--mfc*, page 168.

If the whole application is compiled as one compilation unit, it is very useful to make the compiler also discard unused public functions and variables before the interprocedural optimizations are performed. Doing this limits the scope of the optimizations to functions and variables that are actually used. For more information, see *--discard\_unused\_publics*, page 163.

## OPTIMIZATION LEVELS

The compiler supports different levels of optimizations. This table lists the optimizations that are performed on each level:

Optimization level	Description
None (Best debug support)	Variables live through their entire scope Dead code elimination Redundant label elimination Redundant branch elimination
Low	Same as above but variables only live for as long as they are needed, not necessarily through their entire scope

Table 32: Compiler optimization levels

Optimization level	Description
Medium	Same as above
	Live-dead analysis and optimization
	Code hoisting
	Register content analysis and optimization
	Instruction scheduling (when optimizing for speed - 78K0R)
High (Balanced)	Common subexpression elimination
	Same as above
	Peephole optimization
	Cross jumping
	Cross call (when optimizing for size)
	Loop unrolling
	Function inlining
	Code motion
Type-based alias analysis	

Table 32: Compiler optimization levels (Continued)

**Note:** Some of the performed optimizations can be individually enabled or disabled. For more information about these, see *Fine-tuning enabled transformations*, page 132.

A high level of optimization might result in increased compile time, and will most likely also make debugging more difficult, because it is less clear how the generated code relates to the source code. For example, at the low, medium, and high optimization levels, variables do not live through their entire scope, which means processor registers used for storing variables can be reused immediately after they were last used. Due to this, the C-SPY Watch window might not be able to display the value of the variable throughout its scope. At any time, if you experience difficulties when debugging your code, try lowering the optimization level.

## SPEED VERSUS SIZE

At the high optimization level, the compiler balances between size and speed optimizations. However, it is possible to fine-tune the optimizations explicitly for either size or speed. They only differ in what thresholds that are used; speed will trade size for speed, whereas size will trade speed for size. Note that one optimization sometimes enables other optimizations to be performed, and an application might in some cases become smaller even when optimizing for speed rather than size.

## FINE-TUNING ENABLED TRANSFORMATIONS

At each optimization level you can disable some of the transformations individually. To disable a transformation, use either the appropriate option, for instance the command

line option `--no_inline`, alternatively its equivalent in the IDE **Function inlining**, or the `#pragma optimize` directive. These transformations can be disabled individually:

- Common subexpression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis

### Common subexpression elimination

Redundant re-evaluation of common subexpressions is by default eliminated at optimization levels **Medium** and **High**. This optimization normally reduces both code size and execution time. However, the resulting code might be difficult to debug.

**Note:** This option has no effect at optimization levels **None** and **Low**.

To read more about the command line option, see `--no_cse`, page 170.

### Loop unrolling

Loop unrolling means that the code body of a loop, whose number of iterations can be determined at compile time, is duplicated. Loop unrolling reduces the loop overhead by amortizing it over several iterations.

This optimization is most effective for smaller loops, where the loop overhead can be a substantial part of the total loop body.

Loop unrolling, which can be performed at optimization level **High**, normally reduces execution time, but increases code size. The resulting code might also be difficult to debug.

The compiler heuristically decides which loops to unroll. Only relatively small loops where the loop overhead reduction is noticeable will be unrolled. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed.

**Note:** This option has no effect at optimization levels **None**, **Low**, and **Medium**.

For information about the related pragma directive, see `unroll`, page 231. To disable loop unrolling, use the command line option `--no_unroll`, see `--no_unroll`, page 173.

### Function inlining

Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call. This optimization, which is performed at optimization level **High**, normally reduces

execution time, but increases code size. The resulting code might also be difficult to debug.

The compiler decides which functions to inline. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed.

**Note:** This option has no effect at optimization levels **None**, **Low**, and **Medium**.

To read more about the command line option, see `--no_inline`, page 171.

### Code motion

Evaluation of loop-invariant expressions and common subexpressions are moved to avoid redundant re-evaluation. This optimization, which is performed at optimization level **High**, normally reduces code size and execution time. The resulting code might however be difficult to debug.

**Note:** This option has no effect at optimization levels **None**, and **Low**.

### Type-based alias analysis

When two or more pointers reference the same memory location, these pointers are said to be *aliases* for each other. The existence of aliases makes optimization more difficult because it is not necessarily known at compile time whether a particular value is being changed.

Type-based alias analysis optimization assumes that all accesses to an object are performed using its declared type or as a `char` type. This assumption lets the compiler detect whether pointers can reference the same memory location or not.

Type-based alias analysis is performed at optimization level **High**. For ISO/ANSI standard-conforming C or C++ application code, this optimization can reduce code size and execution time. However, non-standard-conforming C or C++ code might result in the compiler producing code that leads to unexpected behavior. Therefore, it is possible to turn this optimization off.

**Note:** This option has no effect at optimization levels **None**, **Low**, and **Medium**.

To read more about the command line option, see `--no_tbaa`, page 172.

### Example

```
short F(short *p1, long *p2)
{
    *p2 = 0;
    *p1 = 1;
    return *p2;
}
```

With type-based alias analysis, it is assumed that a write access to the `short` pointed to by `p1` cannot affect the `long` value that `p2` points to. Thus, it is known at compile time that this function returns 0. However, in non-standard-conforming C or C++ code these pointers could overlap each other by being part of the same union. If you use explicit casts, you can also force pointers of different pointer types to point to the same memory location.

---

## Writing efficient code

This section contains general programming hints on how to implement functions to make your applications robust, but at the same time facilitate compiler optimizations.

### WRITING OPTIMIZATION-FRIENDLY CODE

This is a list of programming techniques that will, when followed, enable the compiler to better optimize the application:

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions can modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. This is inefficient for two main reasons. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables. Also avoid taking the address of frequently accessed static variables.
- The compilers are capable of inlining functions. This means that instead of calling a function, the compiler inserts the content of the function at the location where the function was called. The result is a faster, but often larger, application. Also, inlining might enable further optimizations. The compiler often inlines small functions declared static. The use of the `#pragma inline` directive and the C++ keyword `inline` gives you fine-grained control, and it is the preferred method compared to the traditional way of using preprocessor macros. This feature can be disabled using the `--no_inline` command line option; see `--no_inline`, page 171.
- Avoid using inline assembler. Instead, try writing the code in C or C++, use intrinsic functions, or write a separate module in assembler language. For more details, see *Mixing C and assembler*, page 91.

## SAVING STACK SPACE AND RAM MEMORY

This is a list of programming techniques that will, when followed, save memory and stack space:

- If stack space is limited, avoid long call chains and recursive functions.
- Avoid using large non-scalar types, such as structures, as parameters or return type. To save stack space, you should instead pass them as pointers or, in C++, as references.

## FUNCTION PROTOTYPES

It is possible to declare and define functions using one of two different styles:

- Prototyped
- Kernighan & Ritchie C (K&R C)

Both styles are included in the C standard; however, it is recommended to use the prototyped style, since it makes it easier for the compiler to find problems in the code. Using the prototyped style will also make it possible to generate more efficient code, since type promotion (implicit casting) is not needed. The K&R style is only supported for compatibility reasons.

To make the compiler verify that all functions have proper prototypes, use the compiler option **Require prototypes** (`--require_prototypes`).

### Prototyped style

In prototyped function declarations, the type for each parameter must be specified.

```
int Test(char, int); /* Declaration */

int Test(char ch, int i) /* Definition */
{
    return i + ch;
}
```

### Kernighan & Ritchie style

In K&R style—traditional pre-ISO/ANSI C—it is not possible to declare a function prototyped. Instead, an empty parameter list is used in the function declaration. Also, the definition looks different.

For example:

```
int Test();      /* Declaration */

int Test(ch, i) /* Definition */
char ch;
int i;
{
    return i + ch;
}
```

## INTEGER TYPES AND BIT NEGATION

In some situations, the rules for integer types and their conversion lead to possibly confusing behavior. Things to look out for are assignments or conditionals (test expressions) involving types with different size, and logical operations, especially bit negation. Here, *types* also includes types of constants.

In some cases there might be warnings (for example, for constant conditional or pointless comparison), in others just a different result than what is expected. Under certain circumstances the compiler might warn only at higher optimizations, for example, if the compiler relies on optimizations to identify some instances of constant conditionals. In this example an 8-bit character, a 16-bit integer, and two's complement is assumed:

```
void F1(unsigned char c1)
{
    if (c1 == ~0x80)
        ;
}
```

Here, the test is always false. On the right hand side, `0x80` is `0x0080`, and `~0x0080` becomes `0xFF7F`. On the left hand side, `c1` is an 8-bit unsigned character, so it cannot be larger than 255. It also cannot be negative, which means that the integral promoted value can never have the topmost 8 bits set.

## PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

Variables that are accessed asynchronously, for example by interrupt routines or by code executing in separate threads, must be properly marked and have adequate protection. The only exception to this is a variable that is always *read-only*.

To mark a variable properly, use the `volatile` keyword. This informs the compiler, among other things, that the variable can be changed from other threads. The compiler will then avoid optimizing on the variable (for example, keeping track of the variable in registers), will not delay writes to it, and be careful accessing the variable only the

number of times given in the source code. To read more about the `volatile` type qualifier, see *Declaring objects volatile*, page 189.

A sequence that accesses a `volatile` declared variable must also not be interrupted. Use the `__monitor` keyword in interruptible code to ensure this. This must be done for both write *and* read sequences, otherwise you might end up reading a partially updated variable. This is true for all variables of all sizes. Accessing a small-sized variable can be an atomic operation, but this is not guaranteed and you should not rely on it unless you continuously study the compiler output. It is safer to use the `__monitor` keyword to ensure that the sequence is an atomic operation.

## ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several 78K devices are included in the IAR product installation. The header files are named `iodevice.h` and define the processor-specific special function registers (SFRs).

**Note:** Each header file contains one section used by the compiler, and one section used by the assembler.

SFRs with bitfields are declared in the header file. This example is from `io78f0058.h`:

```
//-----
// SFR bits (default names)
//-----
typedef struct
{
    unsigned char no0:1;
    unsigned char no1:1;
    unsigned char no2:1;
    unsigned char no3:1;
    unsigned char no4:1;
    unsigned char no5:1;
    unsigned char no6:1;
    unsigned char no7:1;
} __BITS8;

//-----
// SFR declarations
//-----
extern __sfr __no_init volatile union { unsigned char TOC0;
__BITS8 TOC0_bit; } @ 0xFF4E;

//-----
// SFR bit declarations
//-----
#define TOE0        TOC0_bit.no0
#define LVR0        TOC0_bit.no2
```

```
#define LVS0      TOC0_bit.no3
#define OSPE     TOC0_bit.no5
#define OSPT     TOC0_bit.no6
```

By including the appropriate include file in your code, it is possible to access either the whole register or any individual bit (or bitfields) from C code as follows:

```
/* whole register access */
TOC0 = 0x12;

/* Bitfield accesses */
TOE0 = 1;
LVS0 = 0;
```

You can also use the header files as templates when you create new header files for other 78K devices. For details about the @ operator, see *Located data*, page 49.

## NON-INITIALIZED VARIABLES

Normally, the runtime environment will initialize all global and static variables when the application is started.

The compilers support the declaration of variables that will not be initialized, using the `__no_init` type modifier. They can be specified either as a keyword or using the `#pragma object_attribute` directive. The compiler places such variables in a separate segment, according to the specified memory keyword. See the chapter *Placing code and data* for more information.

For `__no_init`, the `const` keyword implies that an object is read-only, rather than that the object is stored in read-only memory. It is not possible to give a `__no_init` object an initial value.

Variables declared using the `__no_init` keyword could, for example, be large input buffers or mapped to special RAM that keeps its content even when the application is turned off.

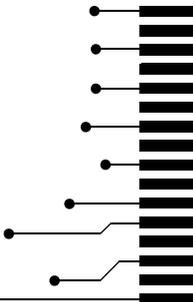
For information about the `__no_init` keyword, see page 213. Note that to use this keyword, language extensions must be enabled; see *-e*, page 164. For information about the `#pragma object_attribute`, see page 226.

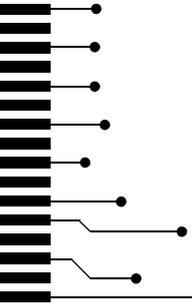


# Part 2. Reference information

This part of the IAR C/C++ Compilers Reference Guide for 78K contains these chapters:

- External interface details
- Compiler options
- Data representation
- Compiler extensions
- Extended keywords
- Pragma directives
- Intrinsic functions
- The preprocessor
- Library functions
- Segment reference
- Implementation-defined behavior.





# External interface details

This chapter provides reference information about how the compilers interact with their environment. The chapter briefly lists and describes the invocation syntax, methods for passing options to the tools, environment variables, the include file search procedure, and finally the different types of compiler output.

---

## Invocation syntax

You can use the compiler either from the IDE or from the command line. Refer to the *IAR Embedded Workbench® IDE User Guide* for information about using the compiler from the IDE.

### COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
icc78k [options] [sourcefile] [options]
```

For example, when compiling the source file `prog.c`, use this command to generate an object file with debug information:

```
icc78k prog.c --debug
```

The source file can be a C or C++ file, typically with the filename extension `c` or `cpp`, respectively. If no filename extension is specified, the file to be compiled must have the extension `c`.

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 that they are specified on the command line.

If you run the compiler from the command line without any arguments, the compiler version number and all available options including brief descriptions are directed to `stdout` and displayed on the screen.

### PASSING OPTIONS

There are three different ways of passing options to the compiler:

- Directly from the command line  
Specify the options on the command line after the `icc78k` command, either before or after the source filename; see *Invocation syntax*, page 143.

- Via environment variables  
The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 144.
- Via a text file, using the `-f` option; see `-f`, page 165.

For general guidelines for the option syntax, an options summary, and a detailed description of each option, see the *Compiler options* chapter.

## ENVIRONMENT VARIABLES

These environment variables can be used with the compilers:

Environment variable	Description
<code>C_INCLUDE</code>	Specifies directories to search for include files; for example: <code>C_INCLUDE=c:\program files\iar systems\embedded workbench 5.n\78k\inc;c:\headers</code>
<code>QCC78K</code>	Specifies command line options; for example: <code>QCC78K=-lA asm.lst</code>

Table 33: Compiler environment variables

## Include file search procedure

This is a detailed description of the `#include` file search procedure of the compilers:

- If the name of the `#include` file is an absolute path, that file is opened.
- If the compiler encounters the name of an `#include` file in angle brackets, such as:

```
#include <stdio.h>
```

it searches these directories for the file to include:

- 1 The directories specified with the `-I` option, in the order that they were specified, see `-I`, page 167.
- 2 The directories specified using the `C_INCLUDE` environment variable, if any, see *Environment variables*, page 144.

- If the compiler encounters the name of an `#include` file in double quotes, for example:

```
#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 compiler starts searching the directory of the file that was last included, iterating upwards for each included file, searching the source file directory last. For example:

```
src.c in directory dir\src
#include "src.h"
...
src.h in directory dir\include
#include "config.h"
...
```

When `dir\exe` is the current directory, use this command for compilation:

```
icc78k ..\src\src.c -I..\include -I..\debugconfig
```

Then the following directories are searched in the order listed below for the file `config.h`, which in this example is located in the `dir\debugconfig` directory:

<code>dir\include</code>	Current file is <code>src.h</code> .
<code>dir\src</code>	File including current file ( <code>src.c</code> ).
<code>dir\include</code>	As specified with the first <code>-I</code> option.
<code>dir\debugconfig</code>	As specified with the second <code>-I</code> option.

Use angle brackets for standard header files, like `stdio.h`, and double quotes for files that are part of your application.

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

---

## Compiler output

The compilers can produce the following output:

- A linkable object file
 

The object files produced by the compiler use a proprietary format called UBROF, which stands for Universal Binary Relocatable Object Format. By default, the object file has the filename extension `r26`.
- Optional list files
 

Various kinds of list files can be specified using the compiler option `-l`, see `-l`, page 167. By default, these files will have the filename extension `lst`.
- Optional preprocessor output files
 

A preprocessor output file is produced when you use the `--preprocess` option; by default, the file will have the filename extension `i`.

- Diagnostic messages

Diagnostic messages are directed to the standard error stream and displayed on the screen, and printed in an optional list file. To read more about diagnostic messages, see *Diagnostics*, page 146.

- Error return codes

These codes provide status information to the operating system which can be tested in a batch file, see *Error return codes*, page 146.

- Size information

Information about the generated amount of bytes for functions and data for each memory is directed to the standard output stream and displayed on the screen. Some of the bytes might be reported as *shared*.

Shared objects are functions or data objects that are shared between modules. If any of these occur in more than one module, only one copy is retained. For example, in some cases inline functions are not inlined, which means that they are marked as shared, because only one instance of each function will be included in the final application. This mechanism is sometimes also used for compiler-generated code or data not directly associated with a particular function or variable, and when only one instance is required in the final application.

### Error return codes

The compiler returns status information to the operating system that can be tested in a batch file.

These command line error codes are supported:

Code	Description
0	Compilation successful, but there might have been warnings.
1	Warnings were produced and the option <code>--warnings_affect_exit_code</code> was used.
2	Errors occurred.
3	Fatal errors occurred, making the compiler abort.
4	Internal errors occurred, making the compiler abort.

Table 34: Error return codes

---

## Diagnostics

This section 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 compiler is produced in the form:

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

with these elements:

<i>filename</i>	The name of the source file in which the issue was encountered
<i>linenumber</i>	The line number at which the compiler detected the issue
<i>level</i>	The level of seriousness of the issue
<i>tag</i>	A unique tag that identifies the diagnostic message
<i>message</i>	An explanation, possibly several lines long

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

Use the option `--diagnostics_tables` to list all possible compiler diagnostic messages.

## SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

### Remark

A diagnostic message that is produced when the compiler finds a source code construction that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see `--remarks`, page 177.

### Warning

A diagnostic message that is produced when the compiler finds a programming error or omission which is of concern, but not so severe as to prevent the completion of compilation. Warnings can be disabled by use of the command line option `--no_warnings`, see page 173.

### Error

A diagnostic message that is produced when the compiler finds a construction which clearly violates the C or C++ 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 compiler finds a condition that not only prevents code generation, but which makes further processing of the source code pointless. After the message is 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 diagnostics messages, except for fatal errors and some of the regular errors.

See *Summary of compiler options*, page 152, for a description of the compiler options that are available for setting severity levels.

See the chapter *Pragma directives*, for a description of the pragma directives that are available for setting severity levels.

### INTERNAL ERROR

An internal error is a diagnostic message that signals that there was a serious and unexpected failure due to a fault in the compiler. It is produced using this form:

Internal error: *message*

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

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

# Compiler options

This chapter describes the syntax of compiler options and the general syntax rules for specifying option parameters, and gives detailed reference information about each option.

---

## Options syntax

Compiler options are parameters you can specify to change the default behavior of the compiler. You can specify options from the command line—which is described in more detail in this section—and from within the IDE.



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

### TYPES OF OPTIONS

There are two *types of names* for command line options, *short* names and *long* names. Some options have both.

- A short option name consists of one character, and it can have parameters. You specify it with a single dash, for example `-e`
- A long option name consists of one or several words joined by underscores, and it can have parameters. You specify it with double dashes, for example `--char_is_signed`.

For information about the different methods for passing options, see *Passing options*, page 143.

### RULES FOR SPECIFYING PARAMETERS

There are some general syntax rules for specifying option parameters. First, the rules depending on whether the parameter is *optional* or *mandatory*, and whether the option has a short or a long name, are described. Then, the rules for specifying filenames and directories are listed. Finally, the remaining rules are listed.

#### Rules for optional parameters

For options with a short name and an optional parameter, any parameter should be specified without a preceding space, for example:

`-O` or `-Oh`

For options with a long name and an optional parameter, any parameter should be specified with a preceding equal sign (=), for example:

```
--misrac2004=n
```

### Rules for mandatory parameters

For options with a short name and a mandatory parameter, the parameter can be specified either with or without a preceding space, for example:

```
-I..\src or -I ..\src\
```

For options with a long name and a mandatory parameter, the parameter can be specified either with a preceding equal sign (=) or with a preceding space, for example:

```
--diagnostics_tables=MyDiagnostics.lst
```

or

```
--diagnostics_tables MyDiagnostics.lst
```

### Rules for options with both optional and mandatory parameters

For options taking both optional and mandatory parameters, the rules for specifying the parameters are:

- For short options, optional parameters are specified without a preceding space
- For long options, optional parameters are specified with a preceding equal sign (=)
- For short and long options, mandatory parameters are specified with a preceding space.

For example, a short option with an optional parameter followed by a mandatory parameter:

```
-lA MyList.lst
```

For example, a long option with an optional parameter followed by a mandatory parameter:

```
--preprocess=n PreprocOutput.lst
```

### Rules for specifying a filename or directory as parameters

These rules apply for options taking a filename or directory as parameters:

- Options that take a filename as a parameter can optionally also take a path. The path can be relative or absolute. For example, to generate a listing to the file `List.lst` in the directory `..\listings\`:

```
icc78k prog.c -l ..\listings\List.lst
```

- For options that take a filename as the destination for output, the parameter can be specified as a path without a specified filename. The compiler stores the output in that directory, in a file with an extension according to the option. The filename will be the same as the name of the compiled source file, unless a different name was specified with the option `-o`, in which case that name is used. For example:

```
icc78k prog.c -l ..\listings\
```

The produced list file will have the default name `..\listings\prog.lst`

- The *current directory* is specified with a period (`.`). For example:
- ```
icc78k prog.c -l .
```
- `/` can be used instead of `\` as the directory delimiter.
  - By specifying `-`, input files and output files can be redirected to the standard input and output stream, respectively. For example:

```
icc78k prog.c -l -
```

### Additional rules

These rules also apply:

- 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; this example will create a list file called `-r`:

```
icc78k prog.c -l ---r
```

- For options that accept multiple arguments of the same type, the arguments can be provided as a comma-separated list (without a space), for example:

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

Alternatively, the option can be repeated for each argument, for example:

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

---

## Summary of compiler options

This table summarizes the compiler command line options:

| Command line option                                 | Description                                                             |
|-----------------------------------------------------|-------------------------------------------------------------------------|
| <code>--aggressive_inlining</code>                  | Inlines larger functions at high speed and balanced optimization levels |
| <code>--char_is_signed</code>                       | Treats <code>char</code> as signed                                      |
| <code>--clib</code>                                 | Uses the legacy CLIB library floating-point types                       |
| <code>--code_model</code>                           | Specifies the code model                                                |
| <code>--code_segment</code>                         | Changes the default segment for functions                               |
| <code>--core</code>                                 | Specifies a CPU core                                                    |
| <code>-D</code>                                     | Defines preprocessor symbols                                            |
| <code>--data_model</code>                           | Specifies the data model (78K0R only)                                   |
| <code>--debug</code>                                | Generates debug information                                             |
| <code>--dependencies</code>                         | Lists file dependencies                                                 |
| <code>--diag_error</code>                           | Treats these as errors                                                  |
| <code>--diag_remark</code>                          | Treats these as remarks                                                 |
| <code>--diag_suppress</code>                        | Suppresses these diagnostics                                            |
| <code>--diag_warning</code>                         | Treats these as warnings                                                |
| <code>--diagnostics_tables</code>                   | Lists all diagnostic messages                                           |
| <code>--disable_data_alignment</code>               | Disables word alignment for data                                        |
| <code>--discard_unused_publics</code>               | Discards unused public symbols                                          |
| <code>--dlib_config</code>                          | Determines the library configuration file                               |
| <code>-e</code>                                     | Enables language extensions                                             |
| <code>--ec++</code>                                 | Enables Embedded C++ syntax                                             |
| <code>--eec++</code>                                | Enables Extended Embedded C++ syntax                                    |
| <code>--enable_multibytes</code>                    | Enables support for multibyte characters in source files                |
| <code>--error_limit</code>                          | Specifies the allowed number of errors before compilation stops         |
| <code>-f</code>                                     | Extends the command line                                                |
| <code>--generate_callt_runtime_library_calls</code> | Generates <code>__callt</code> runtime-library calls                    |

*Table 35: Compiler options summary*

| Command line option                               | Description                                                                                                                                                                           |
|---------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>--generate_far_runtime_library_calls</code> | Generates <code>__far</code> runtime library calls (78K0R only)                                                                                                                       |
| <code>--header_context</code>                     | Lists all referred source files and header files                                                                                                                                      |
| <code>-I</code>                                   | Specifies include file path                                                                                                                                                           |
| <code>-l</code>                                   | Creates a list file                                                                                                                                                                   |
| <code>--library_module</code>                     | Creates a library module                                                                                                                                                              |
| <code>--mfc</code>                                | Enables multi file compilation                                                                                                                                                        |
| <code>--migration_preprocessor_extensions</code>  | Extends the preprocessor                                                                                                                                                              |
| <code>--misrac1998</code>                         | Enables error messages specific to MISRA-C:1998. See the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> .                                                                |
| <code>--misrac2004</code>                         | Enables error messages specific to MISRA-C:2004. See the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> .                                                                |
| <code>--misrac_verbose</code>                     | Enables verbose logging of MISRA C checking. See the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> or the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> . |
| <code>--module_name</code>                        | Sets the object module name                                                                                                                                                           |
| <code>--near_const_location</code>                | Specifies near constants to be writable or mirrored from ROM (78K0R only)                                                                                                             |
| <code>--no_code_motion</code>                     | Disables code motion optimization                                                                                                                                                     |
| <code>--no_cse</code>                             | Disables common subexpression elimination                                                                                                                                             |
| <code>--no_inline</code>                          | Disables function inlining                                                                                                                                                            |
| <code>--no_path_in_file_macros</code>             | Removes the path from the return value of the symbols <code>__FILE__</code> and <code>__BASE_FILE__</code>                                                                            |
| <code>--no_tbaa</code>                            | Disables type-based alias analysis                                                                                                                                                    |
| <code>--no_typedefs_in_diagnostics</code>         | Disables the use of typedef names in diagnostics                                                                                                                                      |
| <code>--no_unroll</code>                          | Disables loop unrolling                                                                                                                                                               |
| <code>--no_warnings</code>                        | Disables all warnings                                                                                                                                                                 |
| <code>--no_wrap_diagnostics</code>                | Disables wrapping of diagnostic messages                                                                                                                                              |
| <code>-O</code>                                   | Sets the optimization level                                                                                                                                                           |

Table 35: Compiler options summary (Continued)

| Command line option         | Description                                                     |
|-----------------------------|-----------------------------------------------------------------|
| -o                          | Sets the object filename                                        |
| --omit_types                | Excludes type information                                       |
| --only_stdout               | Uses standard output only                                       |
| --output                    | Sets the object filename                                        |
| --predef_macros             | Lists the predefined symbols.                                   |
| --preinclude                | Includes an include file before reading the source file         |
| --preprocess                | Generates preprocessor output                                   |
| --public_equ                | Defines a global named assembler label                          |
| -r                          | Generates debug information                                     |
| --remarks                   | Enables remarks                                                 |
| --require_prototypes        | Verifies that functions are declared before they are defined    |
| --silent                    | Sets silent operation                                           |
| --strict_ansi               | Checks for strict compliance with ISO/ANSI C                    |
| --warnings_affect_exit_code | Warnings affects exit code                                      |
| --warnings_are_errors       | Warnings are treated as errors                                  |
| --workseg_area              | Enables a short address work segment area with an optional size |

Table 35: Compiler options summary (Continued)

## Descriptions of options

The following section gives detailed reference information about each compiler option.



Note that if you use the options page **Extra Options** to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

### --aggressive\_inlining

Syntax

--aggressive\_inlining

Description

Use this option to inline larger functions at the optimization levels High balanced and High speed.



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

## --char\_is\_signed

Syntax `--char_is_signed`

Description By default, the compiler interprets the `char` type as unsigned. Use this option to make the compiler interpret the `char` type as signed instead. This can be useful when you, for example, want to maintain compatibility with another compiler.

**Note:** The runtime library is compiled without the `--char_is_signed` option. If you use this option, you might get type mismatch warnings from the linker, because the library uses `unsigned char`.



**Project>Options>C/C++ Compiler>Language>Plain 'char' is**

## --clib

Syntax `--clib`

Description This option enables legacy CLIB library floating-point types. This option must be used if you are using the CLIB library from the command line.



This option is not available in the IDE.

## --code\_model

Syntax `--code_model={standard|s|banked|b|near|n|far|f}`

Parameters

|                                                |                                                                 |
|------------------------------------------------|-----------------------------------------------------------------|
| <code>standard</code> (default for 78K0/78K0S) | Non-banked function calls. 78K0/78K0S only.                     |
| <code>banked</code>                            | Banked function calls. 78K0/78K0S only.                         |
| <code>near</code> (default for 78K0R)          | Function calls reach the first 64 Kbytes of memory. 78K0R only. |
| <code>Far</code>                               | Function calls reach the entire 1 Mbyte memory. 78K0R only.     |

Description Use this option to select the code model for which the code will be generated. If you do not select a code model option, the compiler uses the default code model. Note that all modules of your application must use the same code model.

See also

*Code models and memory attributes for function storage*, page 21.



**Project>Options>General Options>Target>Code model**

## --code\_segment

Syntax

`--code_segment=segment_name`

Description

By default, the compiler places functions into a code segment determined by the code model. Use the `--code_segment` option to by default place all functions in a segment with the name `segment_name`. You can then assign this segment to a fixed address in the linker command file. (The segment name is case-sensitive.)

This is useful if you want to place your code in a different address range and you find the `@` notation, alternatively the `#pragma location` directive, insufficient. Note that any changes to the segment name requires corresponding modifications in the linker command file.

**Note:** This option only applies to functions that are placed in default memory, either explicitly or implicitly. If you use a function memory attribute that explicitly places the function in another segment than the default segment, that function will not be affected by the `--code_segment` option.

See also

For information about the different methods for controlling placement of code, see *Controlling data and function placement in memory*, page 126.



**Project>Options>C/C++ Compiler>Output>Code segment**

## --core

Syntax

`--core={78k0_basic|78k0|78k0s|78k0r}`

Parameters

|                             |                                                   |
|-----------------------------|---------------------------------------------------|
| <code>78K0_basic</code>     | <b>78K0 core without the DIV/MUL instructions</b> |
| <code>78K0 (default)</code> | <b>78K0 core with the DIV/MUL instructions</b>    |
| <code>78K0S</code>          | <b>78K0S core</b>                                 |
| <code>78K0R</code>          | <b>78K0R core</b>                                 |

Description

Use this option to select the processor core for which the code will be generated. If you do not use the option to specify a core, the compiler uses the `78K0` core by default. Note that all modules of your application must use the same core option.

The compiler supports the different 78K microcontroller cores and devices based on these cores.



### Project>Options>General Options>Target>Core

## -D

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                                      |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------|
| Syntax      | <code>-D <i>symbol</i> [=<i>value</i>]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                      |
| Parameters  | <i>symbol</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | The name of the preprocessor symbol  |
|             | <i>value</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | The value of the preprocessor symbol |
| Description | <p>Use this option to define a preprocessor symbol. If no value is specified, 1 is used. This option can be used one or more times on the command line.</p> <p>The option <code>-D</code> has the same effect as a <code>#define</code> statement at the top of the source file:</p> <pre>-D<i>symbol</i></pre> <p>is equivalent to:</p> <pre>#define <i>symbol</i> 1</pre> <p>To get the equivalence of:</p> <pre>#define FOO</pre> <p>specify the = sign but nothing after, for example:</p> <pre>-DFOO=</pre> |                                      |



### Project>Options>C/C++ Compiler>Preprocessor>Defined symbols

## --data\_model

|            |                                                |                                                              |
|------------|------------------------------------------------|--------------------------------------------------------------|
| Syntax     | <code>--data_model={near   n   far   f}</code> |                                                              |
| Parameters | <code>near</code> (default)                    | Data is by default placed in the highest 64 Kbytes of memory |
|            | <code>far</code>                               | Data is by default placed in the entire 1 Mbyte of memory    |

**Description** Use this option to select the data model for which the code will be generated. If you do not select a data model option, the compiler uses the default data model. Note that all modules of your application must use the same data model.

**See also** *Data models (78K0R only)*, page 12.



**Project>Options>General Options>Target>Data model**

## **--debug, -r**

**Syntax** `--debug`  
`-r`

**Description** Use the `--debug` or `-r` option to make the compiler include information in the object modules required by the IAR C-SPY® Debugger and other symbolic debuggers.

**Note:** Including debug information will make the object files larger than otherwise.



**Project>Options>C/C++ Compiler>Output>Generate debug information**

## **--dependencies**

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

**Parameters**

|                          |                               |
|--------------------------|-------------------------------|
| <code>i</code> (default) | Lists only the names of files |
| <code>m</code>           | Lists in makefile style       |

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 150.

**Description** Use this option to make the compiler list all source and header files opened by the compilation into a file with the default filename extension `i`.

**Example** 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 produced. 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
```

An example of using `--dependencies` with a popular make utility, such as `gmake` (GNU make):

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

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

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

- 2 Include all the dependency files in the makefile using, for example:

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

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



This option is not available in the IDE.

## --diag\_error

Syntax

```
--diag_error=tag[, tag, ...]
```

Parameters

*tag*                      The number of a diagnostic message, for example the message number `Pe117`

Description

Use this option to reclassify certain diagnostic messages as errors. An error indicates a violation of the C or C++ language rules, of such severity that object code will not be generated. The exit code will be non-zero. This option may be used more than once on the command line.



**Project>Options>C/C++ Compiler>Diagnostics>Treat these as errors**

## --diag\_remark

|             |                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                       |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| Syntax      | <code>--diag_remark=tag[, tag, ...]</code>                                                                                                                                                                                                                                                                                                                                                                         |                                                                                       |
| Parameters  | <i>tag</i>                                                                                                                                                                                                                                                                                                                                                                                                         | The number of a diagnostic message, for example the message number <code>Pe177</code> |
| Description | <p>Use this option to reclassify certain diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a source code construction that may cause strange behavior in the generated code. This option may be used more than once on the command line.</p> <p><b>Note:</b> By default, remarks are not displayed; use the <code>--remarks</code> option to display them.</p> |                                                                                       |



**Project>Options>C/C++ Compiler>Diagnostics>Treat these as remarks**

## --diag\_suppress

|             |                                                                                                                                                                   |                                                                                       |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| Syntax      | <code>--diag_suppress=tag[, tag, ...]</code>                                                                                                                      |                                                                                       |
| Parameters  | <i>tag</i>                                                                                                                                                        | The number of a diagnostic message, for example the message number <code>Pe117</code> |
| Description | <p>Use this option to suppress certain diagnostic messages. These messages will not be displayed. This option may be used more than once on the command line.</p> |                                                                                       |



**Project>Options>C/C++ Compiler>Diagnostics>Suppress these diagnostics**

## --diag\_warning

|             |                                                                                                                                                                                  |                                                                                       |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| Syntax      | <code>--diag_warning=tag[, tag, ...]</code>                                                                                                                                      |                                                                                       |
| Parameters  | <i>tag</i>                                                                                                                                                                       | The number of a diagnostic message, for example the message number <code>Pe826</code> |
| Description | <p>Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the compiler</p> |                                                                                       |

to stop before compilation is completed. This option may be used more than once on the command line.



**Project>Options>C/C++ Compiler>Diagnostics>Treat these as warnings**

## --diagnostics\_tables

|             |                                                                                                                                                                                                                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--diagnostics_tables {filename directory}</code>                                                                                                                                                                                                                                                              |
| Parameters  | For information about specifying a filename or a directory, see <i>Rules for specifying a filename or directory as parameters</i> , page 150.                                                                                                                                                                       |
| Description | Use this option to list all possible diagnostic messages in a named file. This can be 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.<br><br>This option cannot be given together with other options. |



This option is not available in the IDE.

## --disable\_data\_alignment

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--disable_data_alignment</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Description | This option disables word alignment for data objects. It can only be used together with the CLIB runtime environment.<br><br>Note that accessing an object that is not aligned at its correct alignment requires code that is both larger and slower than the code needed to access the same kind of object when aligned correctly. If there are many accesses to such fields in the program, it is usually better to construct the correct values in a struct that is not packed, and access this instead.<br><br>Note that accessing a misaligned object is slower than accessing a correctly aligned object. If speed is of the highest importance, consider constructing the values in a struct that is not packed, and access this instead. |

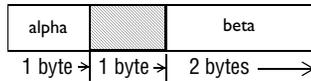
Examples

**Example 1**

This example declares a structure without using the `--disable_data_alignment` option:

```
struct First
{
    char alpha;
    short beta;
};
```

In this example, the structure `First` is not packed and has the following memory layout:



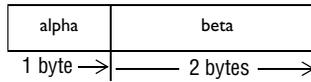
Note that one pad byte has been added.

**Example 2**

This example declares a similar structure using the `--disable_data_alignment` option:

```
struct FirstPacked
{
    char alpha;
    short beta;
};
```

In this example, the structure `FirstPacked` is packed and has the following memory layout:



**Project>Options>C/C++ Compiler>Code>Byte alignment of data.**

## --discard\_unused\_publics

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--discard_unused_publics</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Description | <p>Use this option to discard unused public functions and variables from the compilation unit. This enhances interprocedural optimizations such as inlining, cross call, and cross jump by limiting their scope to public functions and variables that are actually used.</p> <p>This option is only useful when <i>all</i> source files are compiled as one unit, which means that the <code>--mfc</code> compiler option is used.</p> <p><b>Note:</b> Do not use this option only on parts of the application, as necessary symbols might be removed from the generated output.</p> |
| See also    | <code>--mfc</code> , page 168 and <i>Multi-file compilation units</i> , page 131.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |



**Project>Options>C/C++ Compiler>Discard unused publics**

## --dlib\_config

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--dlib_config filename</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Parameters  | For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 150.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Description | <p>Each runtime library has a corresponding library configuration file. Use this option to specify the library configuration file for the compiler. Make sure that you specify a configuration file that corresponds to the library you are using.</p> <p>All prebuilt runtime libraries are delivered with corresponding configuration files. You can find the library object files and the library configuration files in the directory <code>78k\lib</code>. For examples and a list of prebuilt runtime libraries, see <i>Using a prebuilt library</i>, page 56.</p> <p>If you build your own customized runtime library, you should also create a corresponding customized library configuration file, which must be specified to the compiler. For more information, see <i>Building and using a customized library</i>, page 62.</p> <p><b>Note:</b> This option only applies to the IAR DLIB runtime environment.</p> |



To set related options, choose:

**Project>Options>General Options>Library Configuration**

## **-e**

Syntax

-e

Description

In the command line version of the compiler, language extensions are disabled by default. If you use language extensions such as extended keywords and anonymous structs and unions in your source code, you must use this option to enable them.

**Note:** The `-e` option and the `--strict_ansi` option cannot be used at the same time.

See also

The chapter *Compiler extensions*.



**Project>Options>C/C++ Compiler>Language>Allow IAR extensions**

**Note:** By default, this option is enabled in the IDE.

## **--ec++**

Syntax

--ec++

Description

In the compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.



**Project>Options>C/C++ Compiler>Language>Embedded C++**

## **--eec++**

Syntax

--eec++

Description

In the compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++.

See also

*Extended Embedded C++*, page 110.



**Project>Options>C/C++ Compiler>Language>Extended Embedded C++**

## --enable\_multibytes

|             |                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--enable_multibytes</code>                                                                                                                                                                                                                                                                                                                                                                           |
| Description | By default, multibyte characters cannot be used in C or C++ source code. Use this option to make multibyte characters in the source code be interpreted according to the host computer's default setting for multibyte support.<br><br>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>C/C++ Compiler>Language>Enable multibyte support**

## --error\_limit

|             |                                                                                                                                                                  |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--error_limit=n</code>                                                                                                                                     |
| Parameters  | <i>n</i> The number of errors before the compiler stops the compilation. <i>n</i> must be a positive integer; 0 indicates no limit.                              |
| Description | Use the <code>--error_limit</code> option to specify the number of errors allowed before the compiler stops the compilation. By default, 100 errors are allowed. |



This option is not available in the IDE.

## -f

|              |                                                                                                                                                                                                                                                                                                                                                               |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>-f filename</code>                                                                                                                                                                                                                                                                                                                                      |
| Parameters   | For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 150.                                                                                                                                                                                                                                |
| Descriptions | Use this option to make the compiler read command line options from the named file, with the default filename extension <code>.xcl</code> .<br><br>In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character. |

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

## **--generate\_callt\_runtime\_library\_calls**

Syntax

`--generate_callt_runtime_library_calls`

Description

Use this option to generate `__callt` runtime library calls, which means the `CALLT` instruction is used when calling the most frequently used runtime library routines. This will reduce code size.



**Project>Options>C/C++ Compiler>Code>Enable callt runtime library calls.**

## **--generate\_far\_runtime\_library\_calls**

Syntax

`--generate_far_runtime_library_calls`

Description

Use this option to generate `__far` runtime library calls to assembler support routines. This option can be used together with the corresponding assembler option to move the support routines from near to far memory when building a customized library. Note that the startup code will still be located in near memory.

This option sets the runtime model attribute `__far_rt_calls` to true.

**Note:** This option overrides the option

`--generate_callt_runtime_library_calls`.



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

## **--header\_context**

Syntax

`--header_context`

Description

Occasionally, to find the cause of a problem it is necessary to know which header file that was included from which source line. 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 IDE.

**-I**

|             |                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>-I path</code>                                                                                                                     |
| Parameters  | <code>path</code> The search path for <code>#include</code> files                                                                        |
| Description | Use this option to specify the search paths for <code>#include</code> files. This option can be used more than once on the command line. |
| See also    | <i>Include file search procedure</i> , page 144.                                                                                         |



**Project>Options>C/C++ Compiler>Preprocessor>Additional include directories**

**-l**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |   |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |   |                                                                                                                   |   |                        |   |                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|---------------------|---|------------------------------------------------------|---|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|--------------------|-------------|------------------------------------------------------|---|-------------------------------------------------------------------------------------------------------------------|---|------------------------|---|------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>-l[a A b B c C D][N][H] {filename directory}</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |   |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |   |                                                                                                                   |   |                        |   |                                                                                                                                    |
| Parameters  | <table> <tr> <td>a</td> <td>Assembler list file</td> </tr> <tr> <td>A</td> <td>Assembler list file with C or C++ source as comments</td> </tr> <tr> <td>b</td> <td>Basic assembler list file. This file has the same contents as a list file produced with <code>-la</code>, except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *</td> </tr> <tr> <td>B</td> <td>Basic assembler list file. This file has the same contents as a list file produced with <code>-lA</code>, except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *</td> </tr> <tr> <td>c</td> <td>C or C++ list file</td> </tr> <tr> <td>C (default)</td> <td>C or C++ list file with assembler source as comments</td> </tr> <tr> <td>D</td> <td>C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values</td> </tr> <tr> <td>N</td> <td>No diagnostics in file</td> </tr> <tr> <td>H</td> <td>Include source lines from header files in output. Without this option, only source lines from the primary source file are included</td> </tr> </table> | a | Assembler list file | A | Assembler list file with C or C++ source as comments | b | Basic assembler list file. This file has the same contents as a list file produced with <code>-la</code> , except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included * | B | Basic assembler list file. This file has the same contents as a list file produced with <code>-lA</code> , except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included * | c | C or C++ list file | C (default) | C or C++ list file with assembler source as comments | D | C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values | N | No diagnostics in file | H | Include source lines from header files in output. Without this option, only source lines from the primary source file are included |
| a           | Assembler list file                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |   |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |   |                                                                                                                   |   |                        |   |                                                                                                                                    |
| A           | Assembler list file with C or C++ source as comments                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |   |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |   |                                                                                                                   |   |                        |   |                                                                                                                                    |
| b           | Basic assembler list file. This file has the same contents as a list file produced with <code>-la</code> , except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |   |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |   |                                                                                                                   |   |                        |   |                                                                                                                                    |
| B           | Basic assembler list file. This file has the same contents as a list file produced with <code>-lA</code> , except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |   |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |   |                                                                                                                   |   |                        |   |                                                                                                                                    |
| c           | C or C++ list file                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |   |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |   |                                                                                                                   |   |                        |   |                                                                                                                                    |
| C (default) | C or C++ list file with assembler source as comments                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |   |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |   |                                                                                                                   |   |                        |   |                                                                                                                                    |
| D           | C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |   |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |   |                                                                                                                   |   |                        |   |                                                                                                                                    |
| N           | No diagnostics in file                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |   |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |   |                                                                                                                   |   |                        |   |                                                                                                                                    |
| H           | Include source lines from header files in output. Without this option, only source lines from the primary source file are included                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |   |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |   |                                                                                                                   |   |                        |   |                                                                                                                                    |

**\* This makes the list file less useful as input to the assembler, but more useful for reading by a human.**

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 150.

**Description** Use this option to generate an assembler or C/C++ listing to a file. Note that this option can be used one or more times on the command line.



To set related options, choose:

**Project>Options>C/C++ Compiler>List**

## **--library\_module**

**Syntax** `--library_module`

**Description** Use this option to make the compiler generate a library module rather than a program module. A program module is always included during linking. A library module will only be included if it is referenced in your program.



**Project>Options>C/C++ Compiler>Output>Module type>Library Module**

## **--mfc**

**Syntax** `--mfc`

**Description** Use this option to enable *multi-file compilation*. This means that the compiler compiles one or several source files specified on the command line as one unit, which makes interprocedural optimizations such as inlining, cross call, and cross jump possible.

**Note:** The compiler will generate one object file per input source code file, where the first object file contains all relevant data and the other ones are empty. If you want only the first file to be produced, use the `-o` compiler option and specify a certain output file.

**Example** `icc78k myfile1.c myfile2.c myfile3.c --mfc`

**See also** `--discard_unused_publics`, page 163, `-o`, `--output`, page 174, and *Multi-file compilation units*, page 131.



**Project>Options>C/C++ Compiler>Multi-file compilation**

## --migration\_preprocessor\_extensions

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--migration_preprocessor_extensions</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Description | <p>If you need to migrate code from an earlier IAR Systems C or C/C++ compiler, you might want to use this option. Use this option to use the following in preprocessor expressions:</p> <ul style="list-style-type: none"> <li>● Floating-point expressions</li> <li>● Basic type names and <code>sizeof</code></li> <li>● All symbol names (including typedefs and variables).</li> </ul> <p><b>Note:</b> If you use this option, not only will the compiler accept code that does not conform to the ISO/ANSI C standard, but it will also reject some code that <i>does</i> conform to the standard.</p> <p><b>Important!</b> Do not depend on these extensions in newly written code, because support for them might be removed in future compiler versions.</p> |



**Project>Options>C/C++ Compiler>Language>Enable IAR migration preprocessor extensions**

## --module\_name

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--module_name=<i>name</i></code>                                                                                                                                                                                                                                                                                                                                                                                                    |
| Parameters  | <p><i>name</i>                      An explicit object module name</p>                                                                                                                                                                                                                                                                                                                                                                    |
| Description | <p>Normally, the internal name of the object module is the name of the source file, without a directory name or extension. Use this option to specify an object module name explicitly.</p> <p>This option is useful when several modules have the same filename, because the resulting duplicate module name would normally cause a linker error; for example, when the source file is a temporary file generated by a preprocessor.</p> |



**Project>Options>C/C++ Compiler>Output>Object module name**

## --near\_const\_location

|             |                                                                                                                                                                                                                                                                                    |                                                                                                                               |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--near_const_location [RAM ROM0 ROM1]</code>                                                                                                                                                                                                                                 |                                                                                                                               |
| Parameters  | RAM                                                                                                                                                                                                                                                                                | Constants are located in RAM, in the range 0xF0000–0xFFFFF                                                                    |
|             | ROM0                                                                                                                                                                                                                                                                               | Constants are located in ROM, in the range 0x00000–0x0FFFF, and are mirrored by hardware to RAM, in the range 0xF0000–0xFFFFF |
|             | ROM1                                                                                                                                                                                                                                                                               | Constants are located in ROM, in the range 0x10000–0x1FFFF, and are mirrored by hardware to RAM, in the range 0xF0000–0xFFFFF |
| Description | <p>Use this option to specify the location for <code>__near</code> declared constants and strings: in RAM or mirrored from ROM. 78K0R only.</p> <p>For a detailed description of how this works and of the available memory ranges, see the chip manufacturer's documentation.</p> |                                                                                                                               |

## --no\_code\_motion

|             |                                                                                                                                                                                                                                                                                                                                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_code_motion</code>                                                                                                                                                                                                                                                                                                               |
| Description | <p>Use this option to disable code motion optimizations. These optimizations, which are performed at the optimization levels Medium and High, normally reduce code size and execution time. However, the resulting code might be difficult to debug.</p> <p><b>Note:</b> This option has no effect at optimization levels below Medium.</p> |



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Code motion**

## --no\_cse

|             |                                                                                                                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_cse</code>                                                                                                                                                                                                                                                                                              |
| Description | <p>Use this option to disable common subexpression elimination. At the optimization levels Medium and High, the compiler avoids calculating the same expression more than once. This optimization normally reduces both code size and execution time. However, the resulting code might be difficult to debug.</p> |

**Note:** This option has no effect at optimization levels below Medium.



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination**

## **--no\_inline**

Syntax `--no_inline`

Description Use this option to disable function inlining. Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call.

This optimization, which is performed at optimization level High, normally reduces execution time and increases code size. The resulting code might also be difficult to debug.

The compiler heuristically decides which functions to inline. Different heuristics are used when optimizing for speed than for size.

**Note:** This option has no effect at optimization levels below High.



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Function inlining**

## **--no\_path\_in\_file\_macros**

Syntax `--no_path_in_file_macros`

Description Use this option to exclude the path from the return value of the predefined preprocessor symbols `__FILE__` and `__BASE_FILE__`.

See also *Descriptions of predefined preprocessor symbols*, page 238.



This option is not available in the IDE.

## --no\_tbaa

Syntax `--no_tbaa`

Description Use this option to disable type-based alias analysis. When this options is not used, the compiler is free to assume that objects are only accessed through the declared type or through `unsigned char`.

See also *Type-based alias analysis*, page 134.



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Type-based alias analysis**

## --no\_typedefs\_in\_diagnostics

Syntax `--no_typedefs_in_diagnostics`

Description Use this option to disable the use of typedef names in diagnostics. Normally, when a type is mentioned in a message from the compiler, most commonly in a diagnostic message of some kind, the typedef names that were used in the original declaration are used whenever they make the resulting text shorter.

Example 

```
typedef int (*MyPtr)(char const *);
MyPtr p = "foo";
```

will give an error message like this:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "MyPtr"
```

If the `--no_typedefs_in_diagnostics` option is used, the error message will be like this:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "int (*)(char const *)"
```



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

## --no\_unroll

Syntax `--no_unroll`

Description Use this option to disable loop unrolling. Loop unrolling means that the code body of a small loop, whose number of iterations can be determined at compile time, is duplicated to reduce the loop overhead.

This optimization normally reduces execution time, but increases code size. Thus, when optimizing for speed, you might want to disable unrolling to avoid increased code size.

Also, loop unrolling might make the resulting code difficult to debug.

**Note:** This option has no effect at optimization levels below **High**.

See also *Loop unrolling*, page 133.



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling**

## --no\_warnings

Syntax `--no_warnings`

Description By default, the compiler issues warning messages. Use this option to disable all warning messages.



This option is not available in the IDE.

## --no\_wrap\_diagnostics

Syntax `--no_wrap_diagnostics`

Description By default, long lines in 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 IDE.

## -O

Syntax `-O[n|l|m|h|hs|hz]`

### Parameters

|             |                            |
|-------------|----------------------------|
| n           | None* (Best debug support) |
| l (default) | Low*                       |
| m           | Medium                     |
| h           | High, balanced             |
| hs          | High, favoring speed       |
| hz          | High, favoring size        |

**\*The most important difference between None and Low is that at None, all non-static variables will live during their entire scope.**

### Description

Use this option to set the optimization level to be used by the compiler when optimizing the code. If no optimization option is specified, the optimization level Low is used by default. If only `-O` is used without any parameter, the optimization level High balanced is used.

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.

### See also

*Controlling compiler optimizations*, page 130.



**Project>Options>C/C++ Compiler>Optimizations**

## -o, --output

Syntax `-o {filename|directory}`  
`--output {filename|directory}`

### Parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 150.

### Description

By default, the object code output produced by the compiler is located in a file with the same name as the source file, but with the extension `.r26`. Use this option to explicitly specify a different output filename for the object code output.



This option is not available in the IDE.

## --omit\_types

Syntax `--omit_types`

Description By default, the compiler includes type information about variables and functions in the object output. Use this option if you do not want the compiler to include this type information in the output, which is useful when you build a library that should not contain type information. The object file will then only contain type information that is a part of a symbol's name. This means that the linker cannot check symbol references for type correctness.



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

## --only\_stdout

Syntax `--only_stdout`

Description Use this option to make the compiler use the standard output stream (`stdout`) also for messages that are normally directed to the error output stream (`stderr`).



This option is not available in the IDE.

## --output, -o

Syntax `--output {filename|directory}`  
`-o {filename|directory}`

Parameters For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 150.

Description By default, the object code output produced by the compiler is located in a file with the same name as the source file, but with the extension `r26`. Use this option to explicitly specify a different output filename for the object code output.



This option is not available in the IDE.

## --predef\_macros

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--predef_macros {filename directory}</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Parameters  | For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 150.                                                                                                                                                                                                                                                                                                                                                                                     |
| Description | <p>Use this option to list the predefined symbols. When using this option, make sure to also use the same options as for the rest of your project.</p> <p>If a filename is specified, the compiler stores the output in that file. If a directory is specified, the compiler stores the output in that directory, in a file with the <code>predef</code> filename extension.</p> <p> This option is not available in the IDE.</p> |

## --preinclude

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--preinclude includefile</code>                                                                                                                                                                                                                                                                                                                                                                                                             |
| Parameters  | For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 150.                                                                                                                                                                                                                                                                                                                    |
| Description | <p>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.</p> <p> <b>Project&gt;Options&gt;C/C++ Compiler&gt;Preprocessor&gt;Preinclude file</b></p> |

## --preprocess

|                |                                                                                                                                                                                                                                                                                                                                                                      |                |                   |                |                 |                |                           |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------|-------------------|----------------|-----------------|----------------|---------------------------|
| Syntax         | <code>--preprocess [= [c] [n] [1]] {filename directory}</code>                                                                                                                                                                                                                                                                                                       |                |                   |                |                 |                |                           |
| Parameters     | <table> <tr> <td><code>c</code></td> <td>Preserve comments</td> </tr> <tr> <td><code>n</code></td> <td>Preprocess only</td> </tr> <tr> <td><code>1</code></td> <td>Generate #line directives</td> </tr> </table> <p>For information about specifying a filename or a directory, see <i>Rules for specifying a filename or directory as parameters</i>, page 150.</p> | <code>c</code> | Preserve comments | <code>n</code> | Preprocess only | <code>1</code> | Generate #line directives |
| <code>c</code> | Preserve comments                                                                                                                                                                                                                                                                                                                                                    |                |                   |                |                 |                |                           |
| <code>n</code> | Preprocess only                                                                                                                                                                                                                                                                                                                                                      |                |                   |                |                 |                |                           |
| <code>1</code> | Generate #line directives                                                                                                                                                                                                                                                                                                                                            |                |                   |                |                 |                |                           |

**Description** Use this option to generate preprocessed output to a named file.



**Project>Options>C/C++ Compiler>Preprocessor>Preprocessor output to file**

## --public\_equ

**Syntax** `--public_equ symbol [=value]`

**Parameters**

|               |                                                   |
|---------------|---------------------------------------------------|
| <i>symbol</i> | The name of the assembler symbol to be defined    |
| <i>value</i>  | An optional value of the defined assembler symbol |

**Description** This option is equivalent to defining a label in assembler language using the EQU directive and exporting it using the PUBLIC directive. This option can be used more than once on the command line.



This option is not available in the IDE.

## -r, --debug

**Syntax** `-r`  
`--debug`

**Description** Use the `-r` or the `--debug` option to make the compiler include information in the object modules required by the IAR C-SPY Debugger and other symbolic debuggers.

**Note:** Including debug information will make the object files larger than otherwise.



**Project>Options>C/C++ Compiler>Output>Generate debug information**

## --remarks

**Syntax** `--remarks`

**Description** The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks.

See also

*Severity levels*, page 147.



**Project>Options>C/C++ Compiler>Diagnostics>Enable remarks**

## **--require\_prototypes**

Syntax

`--require_prototypes`

Description

Use this option to force the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:

- A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration
- A function definition of a public function with no previous prototype declaration
- An indirect function call through a function pointer with a type that does not include a prototype.

**Note:** This option only applies to functions in the C standard library.



**Project>Options>C/C++ Compiler>Language>Require prototypes**

## **--silent**

Syntax

`--silent`

Description

By default, the compiler issues introductory messages and a final statistics report. Use this option to make the compiler operate without sending these messages to the standard output stream (normally the screen).

This option does not affect the display of error and warning messages.



This option is not available in the IDE.

## **--strict\_ansi**

Syntax

`--strict_ansi`

Description

By default, the compiler accepts a relaxed superset of ISO/ANSI C/C++, see *Minor language extensions*, page 199. Use this option to ensure that the program conforms to the ISO/ANSI C/C++ standard.

**Note:** The `-e` option and the `--strict_ansi` option cannot be used at the same time.



**Project>Options>C/C++ Compiler>Language>Language conformances>Strict ISO/ANSI**

## **--warnings\_affect\_exit\_code**

|             |                                                                                                                                                                              |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--warnings_affect_exit_code</code>                                                                                                                                     |
| Description | By default, the exit code is not affected by warnings, because only errors produce a non-zero exit code. With this option, warnings will also generate a non-zero exit code. |



This option is not available in the IDE.

## **--warnings\_are\_errors**

|             |                                                                                                                                                                                                              |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--warnings_are_errors</code>                                                                                                                                                                           |
| Description | Use this option to make the compiler treat all warnings as errors. If the compiler encounters an error, no object code is generated. Warnings that have been changed into remarks are not treated as errors. |

**Note:** Any diagnostic messages that have been reclassified as warnings by the option `--diag_warning` or the `#pragma diag_warning` directive will also be treated as errors when `--warnings_are_errors` is used.

See also *diag\_warning*, page 223.



**Project>Options>C/C++ Compiler>Diagnostics>Treat all warnings as errors**

## **--workseg\_area**

|             |                                                                                                                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--workseg_area [=size]</code>                                                                                                                                                                                                      |
| Description | Specifies the space reserved in saddr memory for the <code>WRKSEG</code> segment, which is used as the <i>register</i> variable work area. If the parameter <i>size</i> is omitted, the size is by default 20. The maximum value is 128. |

The most frequently used parameters and auto variables are stored in this area, up to the maximum size specified. There is an overhead involved in saving and restoring the work area at each function entry and exit. This overhead can be omitted if the function is declared with the `__no_save` function attribute.

Avoid using the workseg area for functions that are called between a `setjmp` function call and its corresponding `longjmp` call, as the workseg area will likely become corrupt.



**Project>Options>C/C++ Compiler>Code>Short address work area.**

# Data representation

This chapter describes the data types, pointers, and structure types supported by the compilers.

See the chapter *Efficient coding for embedded applications* for information about which data types and pointers provide the most efficient code for your application.

---

## Alignment

Every C data object has an alignment that controls how the object can be stored in memory. Should an object have an alignment of, for example, 4, it must be stored on an address that is divisible by 4.

The reason for the concept of alignment is that some processors have hardware limitations for how the memory can be accessed.

Assume that a processor can read 4 bytes of memory using one instruction, but only when the memory read is placed on an address divisible by 4. Then, 4-byte objects, such as `long` integers, will have alignment 4.

Another processor might only be able to read 2 bytes at a time; in that environment, the alignment for a 4-byte `long` integer might be 2.

A structure type will have the same alignment as the structure member with the most strict alignment. To decrease the alignment requirements on the structure and its members, use the `#pragma pack` directive.

All data types must have a size that is a multiple of their alignment. Otherwise, only the first element of an array would be guaranteed to be placed in accordance with the alignment requirements. This means that the compiler might add pad bytes at the end of the structure. For more information about pad bytes, see *Packed structure types*, page 188.

Note that with the `#pragma data_alignment` directive you can increase the alignment demands on specific variables.

## ALIGNMENT ON THE 78K MICROCONTROLLER

The 78K microcontroller can access memory using 8- or 16-bit operations. However, when a 16-bit access is performed, the data must be located at an even address. The compiler ensures this by assigning an alignment to every data type, ensuring that the 78K microcontroller can read the data. In this case, the word assembler instruction `MOVW` will be used.

For byte alignment, use the option **Disable alignment of data object** (`--disable_data_alignment`). In this case, the instruction `MOV` will be used.

---

## Basic data types

The compilers support both all ISO/ANSI C basic data types and some additional types.

### INTEGER TYPES

This table gives the size and range of each integer data type:

| Data type                   | Size    | Range                   | Alignment |
|-----------------------------|---------|-------------------------|-----------|
| <code>bool</code>           | 8 bits  | 0 to 1                  | 1         |
| <code>char</code>           | 8 bits  | 0 to 255                | 1         |
| <code>signed char</code>    | 8 bits  | -128 to 127             | 1         |
| <code>unsigned char</code>  | 8 bits  | 0 to 255                | 1         |
| <code>signed short</code>   | 16 bits | -32768 to 32767         | 2         |
| <code>unsigned short</code> | 16 bits | 0 to 65535              | 2         |
| <code>signed int</code>     | 16 bits | -32768 to 32767         | 2         |
| <code>unsigned int</code>   | 16 bits | 0 to 65535              | 2         |
| <code>signed long</code>    | 32 bits | $-2^{31}$ to $2^{31}-1$ | 2         |
| <code>unsigned long</code>  | 32 bits | 0 to $2^{32}-1$         | 2         |

*Table 36: Integer types*

Signed variables are represented using the two's complement form.

### Bool

The `bool` data type is supported by default in the C++ language. If you have enabled language extensions, the `bool` type can also be used in C source code if you include the file `stdbool.h`. This will also enable the boolean values `false` and `true`.

## The enum type

The compilers will use the smallest type required to hold `enum` constants, preferring `signed` rather than `unsigned`.

When IAR Systems language extensions are enabled, and in C++, the `enum` constants and types can also be of the type `long`, `unsigned long`, `long long`, or `unsigned long long`.

To make the compiler use a larger type than it would automatically use, define an `enum` constant with a large enough value. For example,

```
/* Disables usage of the char type for enum */
enum Cards{Spade1, Spade2,
           DontUseChar=257};
```

## The char type

The `char` type is by default `unsigned` in the compilers, but the `--char_is_signed` compiler option allows you to make it `signed`. Note, however, that the library is compiled with the `char` type as `unsigned`.

## The wchar\_t type

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The `wchar_t` data type is supported by default in the C++ language. To use the `wchar_t` type also in C source code, you must include the file `stddef.h` from the runtime library.

**Note:** The IAR CLIB Library has only rudimentary support for `wchar_t`.

## Bitfields

In ISO/ANSI C, `int` and `unsigned int` can be used as the base type for integer bitfields. In the IAR C/C++ Compilers for 78K, any integer type can be used as the base type when language extensions are enabled.

Bitfields in expressions will have the same data type as the integer base type.

By default, the compiler places bitfield members from the least significant to the most significant bit in the container type.

If you use the directive `#pragma bitfields=reversed`, the bitfield members are placed from the most significant to the least significant bit.



even though the number is treated as if the exponent was 1. Unlike normal numbers, denormalized numbers do not have an implicit 1 as the most significant bit (the MSB) of the mantissa. The value of a denormalized number is:

$$(-1)^S * 2^{(1-BIAS)} * 0.Mantissa$$

where *BIAS* is 127 and 1023 for 32-bit and 64-bit floating-point values, respectively.

**Note:** The IAR CLIB Library does not fully support the special cases of floating-point numbers, such as infinity, NaN, and subnormal numbers. A library function which gets one of these special cases of floating-point numbers as an argument might behave unexpectedly.

## Pointer types

The compiler has two basic types of pointers: function pointers and data pointers.

### FUNCTION POINTERS

The size of function pointers is always 16 or 24 bits. These function pointers are available:

| Pointer attribute         | Pointer size                     | Default in code model | Description                                                    |
|---------------------------|----------------------------------|-----------------------|----------------------------------------------------------------|
| <code>__non_banked</code> | 2 bytes                          | Standard              | Can address the entire 64 Kbyte memory space. 78K0/78K0S only. |
| <code>__banked</code>     | 3 bytes (uses 4 bytes of memory) | Banked                | Must be used for banked functions. 78K0/78K0S only.            |
| <code>__near_func</code>  | 2 bytes                          | Near                  | Can address the first 64 Kbytes of memory. 78K0R only.         |
| <code>__far_func</code>   | 3 bytes                          | Far                   | Can address the entire 1 Mbyte memory space. 78K0R only.       |

Table 38: Function pointers

### DATA POINTERS

The size of data pointers is always 16 or 24 bits. These data pointers are available:

| Pointer attribute    | Pointer size | Address range                                         | Description        |
|----------------------|--------------|-------------------------------------------------------|--------------------|
| <code>__saddr</code> | 2 bytes      | 0xFE20–0xFF1F (78K0/78K0S)<br>0xFFE20–0xFFF1F (78K0R) | Short address area |

Table 39: Data pointers

| Pointer attribute   | Pointer size         | Address range                                         | Description                                                                 |
|---------------------|----------------------|-------------------------------------------------------|-----------------------------------------------------------------------------|
| <code>__near</code> | 2 bytes<br>(default) | 0x0000–0xFFFF (78K0/78K0S)<br>0xF0000–0xFFFFF (78K0R) | Entire 64 Kbytes address area (78K0/78K0S) or the highest 64 Kbytes (78K0R) |
| <code>__far</code>  | 3 bytes              | 0x00000–0xFFFFF                                       | Entire 1 Mbyte address area. 78K0R only.                                    |

Table 39: Data pointers (Continued)

**Note:** It is not possible to point to an SFR object.

## CASTING

Casts between pointers have these characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a *value* of an integer type to a pointer of a larger type is performed by zero extension
- Casting a *pointer type* to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed by zero extension
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result
- Casting a `__near` pointer to a `__saddr` pointer is a no-operation.
- Casting a `__near` pointer to a `__far` pointer is performed by extension with 0xF0000.
- Casting a `__far` pointer to a `__near` pointer is an illegal operation.

### size\_t

`size_t` is the unsigned integer type required to hold the maximum size of an object. In the IAR C/C++ Compilers for 78K, the size of `size_t` is 16 bits.

### ptrdiff\_t

`ptrdiff_t` is the type of the signed integer required to hold the difference between two pointers to elements of the same array. In the IAR C/C++ Compilers for 78K, the size of `ptrdiff_t` is 16 bits.

**Note:** Subtracting the start address of an object from the end address can yield a negative value, because the object can be larger than what the `ptrdiff_t` can represent. See this example:

```
char buff[60000];           /* Assuming ptrdiff_t is a 16-bit */
char *p1 = buff;           /* signed integer type. */
char *p2 = buff + 60000;
ptrdiff_t diff = p2 - p1;
```

### **intptr\_t**

`intptr_t` is a signed integer type large enough to contain a `void *`. In the IAR C/C++ Compilers for 78K, the size of `intptr_t` is 32 bits.

### **uintptr\_t**

`uintptr_t` is equivalent to `intptr_t`, with the exception that it is unsigned.

---

## Structure types

The members of a `struct` are stored sequentially in the order in which they are declared: the first member has the lowest memory address.

### **ALIGNMENT**

The `struct` and `union` types have the same alignment as the member with the highest alignment requirement. The size of a `struct` is also adjusted to allow arrays of aligned structure objects.

### **GENERAL LAYOUT**

Members of a `struct` are always allocated in the order specified in the declaration. Each member is placed in the `struct` according to the specified alignment (offsets).

#### **Example**

```
struct First
{
    char c;
    short s;
} s;
```

This diagram shows the layout in memory:

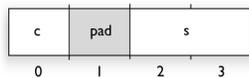


Figure 7: Structure layout

The alignment of the structure is 2 bytes, and a pad byte must be inserted to give `short s` the correct alignment.

## PACKED STRUCTURE TYPES

The `#pragma pack` directive is used for relaxing the alignment requirements of the members of a structure. This changes the layout of the structure. The members are placed in the same order as when declared, but there might be less pad space between members.

Note that accessing an object that is not correctly aligned requires code that is both larger and slower. If such structure members are accessed many times, it is usually better to construct the correct values in a `struct` that is not packed, and access this `struct` instead.

Special care is also needed when creating and using pointers to misaligned members. For direct access to misaligned members in a packed `struct`, the compiler will emit the correct (but slower and larger) code when needed. However, when a misaligned member is accessed through a pointer to the member, the normal (smaller and faster) code is used. In the general case, this will not work.

### Example

This example declares a packed structure:

```
#pragma pack(1)
struct S
{
    char c;
    short s;
};

#pragma pack()
```

In this example, the structure `s` has this memory layout:

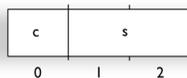


Figure 8: Packed structure layout

This example declares a new non-packed structure, `S2`, that contains the structure `s` declared in the previous example:

```
struct S2
{
    struct S s;
    long l;
};
```

`S2` has this memory layout

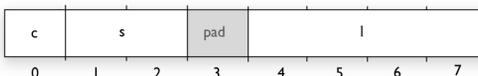


Figure 9: Packed structure layout

The structure `s` will use the memory layout, size, and alignment described in the previous example. The alignment of the member `l` is 2, which means that alignment of the structure `S2` will become 2.

For more information, see *Alignment of elements in a structure*, page 124.

## Type qualifiers

According to the ISO/ANSI C standard, `volatile` and `const` are type qualifiers.

### DECLARING OBJECTS VOLATILE

There are three main reasons for declaring an object `volatile`:

- Shared access; the object is shared between several tasks in a multitasking environment
- Trigger access; as for a memory-mapped SFR where the fact that an access occurs has an effect
- Modified access; where the contents of the object can change in ways not known to the compiler.

### Definition of access to volatile objects

The ISO/ANSI standard defines an abstract machine, which governs the behavior of accesses to `volatile` declared objects. In general and in accordance to the abstract machine, the compiler:

- Considers each read and write access to an object declared `volatile` as an access

- The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:

```
char volatile a;
a = 5; /* A write access */
a += 6; /* First a read then a write access */
```

- An access to a bitfield is treated as an access to the underlying type.

However, these rules are not detailed enough to handle the hardware-related requirements. The rules specific to the IAR C/C++ Compilers for 78K are described below.

### Rules for accesses

In the IAR C/C++ Compilers for 78K, accesses to `volatile` declared objects are subject to these rules:

- All accesses are preserved
- All accesses are complete, that is, the whole object is accessed
- All accesses are performed in the same order as given in the abstract machine
- All accesses are atomic, that is, they cannot be interrupted.

The compilers adhere to these rules for all 8-bit data types. Variables declared `volatile` and `__sfr` can be accessed using bit access unless they are also declared `__nokitaccess`.

For all other object types, only the rule that all accesses are preserved applies.

### DECLARING OBJECTS CONST

The `const` type qualifier is used for indicating that a data object, accessed directly or via a pointer, is non-writable. A pointer to `const` declared data can point to both constant and non-constant objects. It is good programming practice to use `const` declared pointers whenever possible because this improves the compiler's possibilities to optimize the generated code and reduces the risk of application failure due to erroneously modified data.

Static and global objects declared `const` and located in far memory are allocated in ROM. `const` declared `saddr` objects are allocated in RAM and initialized by the runtime system at startup. `const` declared near objects are allocated in ROM (78K0 and 78K0S) or according to the option `--near_const_location` (78K0R).

In C++, objects that require runtime initialization cannot be placed in ROM.

---

## Data types in C++

In C++, all plain C data types are represented in the same way as described earlier in this chapter. However, if any Embedded C++ features are used for a type, no assumptions can be made concerning the data representation. This means, for example, that it is not supported to write assembler code that accesses class members.



# Compiler extensions

This chapter gives a brief overview of the compiler extensions to the ISO/ANSI C standard. All extensions can also be used for the C++ programming language. More specifically the chapter describes the available C language extensions.

---

## Compiler extensions overview

The compilers offer the standard features of ISO/ANSI C and a wide set of extensions, ranging from features specifically tailored for efficient programming in the embedded industry to the relaxation of some minor standards issues.

You can find the extensions available as:

- C/C++ language extensions

For a summary of available language extensions, see *C language extensions*, page 194. For reference information about the extended keywords, see the chapter *Extended keywords*. For information about C++, the two levels of support for the language, and C++ language extensions; see the chapter *Using C++*.

- Pragma directives

The `#pragma` directive is defined by the ISO/ANSI C standard and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

The compilers provide a set of predefined pragma directives, which can be used for controlling the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages.

Most pragma directives are preprocessed, which means that macros are substituted in a pragma directive. The pragma directives are always enabled in the compiler. For several of them there is also a corresponding C/C++ language extension. For a list of available pragma directives, see the chapter *Pragma directives*.

- Preprocessor extensions

The preprocessor of the compiler adheres to the ISO/ANSI standard. The compiler also makes several preprocessor-related extensions available to you. For more information, see the chapter *The preprocessor*.

- Intrinsic functions

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of

instructions. To read more about using intrinsic functions, see *Mixing C and assembler*, page 91. For a list of available functions, see the chapter *Intrinsic functions*.

- Library functions

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. The library also provides some extensions, partly taken from the C99 standard. For more information, see *IAR DLIB Library*, page 245.

**Note:** Any use of these extensions, except for the pragma directives, makes your application inconsistent with the ISO/ANSI C standard.

## ENABLING LANGUAGE EXTENSIONS



In the IDE, language extensions are enabled by default.

For information about how to enable and disable language extensions from the command line, see the compiler options `-e`, page 164, and `--strict_ansi`, page 178.

---

## C language extensions

This section gives a brief overview of the C language extensions available in the compilers. The compilers provide a wide set of extensions, so to help you to find the extensions required by your application, the extensions are grouped according to their expected usefulness. In short, this means:

- Important language extensions—extensions specifically tailored for efficient embedded programming, typically to meet memory restrictions
- Useful language extensions—features considered useful and typically taken from related standards, such as C99 and C++
- Minor language extensions, that is, the relaxation of some minor standards issues and also some useful but minor syntax extensions.

## IMPORTANT LANGUAGE EXTENSIONS

The following language extensions available both in the C and the C++ programming languages are well suited for embedded systems programming:

- Memory attributes, type attributes, and object attributes

For information about the related concepts, the general syntax rules, and for reference information, see the chapter *Extended keywords*.

- Placement at an absolute address or in a named segment

The `@` operator or the directive `#pragma location` can be used for placing global and static variables at absolute addresses, or placing a variable or function in a named

segment. For more information about using these primitives, see *Controlling data and function placement in memory*, page 126, and *location*, page 225.

- Alignment

Each data type has its own alignment, for more details, see *Alignment*, page 181. If you want to change the alignment, the `#pragma pack` and `#pragma data_alignment` directives are available. If you want to use the alignment of an object, use the `__ALIGNOF__()` operator.

The `__ALIGNOF__` operator is used for accessing the alignment of an object. It takes one of two forms:

- `__ALIGNOF__(type)`
- `__ALIGNOF__(expression)`

In the second form, the expression is not evaluated.

- Anonymous structs and unions

C++ includes a feature named anonymous unions. The compilers allow a similar feature for both structs and unions in the C programming language. For more information, see *Anonymous structs and unions*, page 125.

- Bitfields and non-standard types

In ISO/ANSI C, a bitfield must be of type `int` or `unsigned int`. Using IAR Systems language extensions, any integer type or enumeration can be used. The advantage is that the struct will sometimes be smaller. This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*. For more information, see *Bitfields*, page 183.

- Dedicated segment operators `__segment_begin` and `__segment_end`

The syntax for these operators is:

```
void * __segment_begin(segment)
void * __segment_end(segment)
```

These operators return the address of the first byte of the named `segment` and the first byte *after* the named `segment`, respectively. This can be useful if you use the `@` operator or the `#pragma location` directive to place a data object or a function in a user-defined segment.

The named `segment` must be a string literal and `segment` must have been declared earlier with the `#pragma segment` directive. If the segment was declared with a memory attribute `memattr`, the type of the `__segment_begin` operator is a pointer to `memattr void`. Otherwise, the type is a default pointer to `void`. Note that you must enable language extensions to use these operators.

In this example, the type of the `__segment_begin` operator is `void __far *`.

```
#pragma segment="MYSEGMENT" __far
...
segment_start_address = __segment_begin("MYSECTION");
```

See also *segment*, page 230, and *location*, page 225.

## USEFUL LANGUAGE EXTENSIONS

This section lists and briefly describes useful extensions, that is, useful features typically taken from related standards, such as C99 and C++:

- **Inline functions**

The `#pragma inline` directive, alternatively the `inline` keyword, advises the compiler that the function whose declaration follows immediately after the directive should be inlined. This is similar to the C++ keyword `inline`. For more information, see *inline*, page 224.

- **Mixing declarations and statements**

It is possible to mix declarations and statements within the same scope. This feature is part of the C99 standard and C++.

- **Declaration in `for` loops**

It is possible to have a declaration in the initialization expression of a `for` loop, for example:

```
for (int i = 0; i < 10; ++i)
{
    /* Do something here. */
}
```

This feature is part of the C99 standard and C++.

- **The `bool` data type**

To use the `bool` type in C source code, you must include the file `stdbool.h`. This feature is part of the C99 standard and C++. (The `bool` data type is supported by default in C++.)

- **C++ style comments**

C++ style comments are accepted. A C++ style comment starts with the character sequence `//` and continues to the end of the line. For example:

```
// The length of the bar, in centimeters.
int length;
```

This feature is copied from the C99 standard and C++.

### Inline assembler

Inline assembler can be used for inserting assembler instructions in the generated function. This feature is part of the C99 standard and C++.

The `asm` and `__asm` extended keywords both insert an assembler instruction. However, when compiling C source code, the `asm` keyword is not available when the option `--strict_ansi` is used. The `__asm` keyword is always available.

**Note:** Not all assembler directives or operators can be inserted using this keyword.

The syntax is:

```
asm ("string");
```

The string can be a valid assembler instruction or a data definition assembler directive, but not a comment. You can write several consecutive inline assembler instructions, for example:

```
asm ("Label:      nop\n"
     "             jmp Label");
```

where `\n` (new line) separates each new assembler instruction. Note that you can define and use local labels in inline assembler instructions.

For more information about inline assembler, see *Mixing C and assembler*, page 91.

## Compound literals

To create compound literals you can use this syntax:

```
/* Create a pointer to an anonymous array */
int *p = (int []) {1, 2, 3};

/* Create a pointer to an anonymous structX */
structX *px = &(amp;structX) {5, 6, 7};
```

**Note:**

- A compound literal can be modified unless it is declared `const`
- Compound literals are not supported in Embedded C++ and Extended EC++.
- This feature is part of the C99 standard.

## Incomplete arrays at end of structs

The last element of a `struct` can be an incomplete array. This is useful for allocating a chunk of memory that contains both the structure and a fixed number of elements of the array. The number of elements can vary between allocations.

This feature is part of the C99 standard.

**Note:** The array cannot be the only member of the `struct`. If that was the case, then the size of the `struct` would be zero, which is not allowed in ISO/ANSI C.

**Example**

```

struct str
{
    char a;
    unsigned long b[];
};

struct str * GetAStr(int size)
{
    return malloc(sizeof(struct str) +
                  sizeof(unsigned long) * size);
}

void UseStr(struct str * s)
{
    s->b[10] = 0;
}

```

The incomplete array will be aligned in the structure just like any other member of the structure. For more information about structure alignment, see *Structure types*, page 187.

**Hexadecimal floating-point constants**

Floating-point constants can be given in hexadecimal style. The syntax is  $0\times MANTp\{+|- \}EXP$ , where *MANT* is the mantissa in hexadecimal digits, including an optional . (decimal point), and *EXP* is the exponent with decimal digits, representing an exponent of 2. This feature is part of the C99 standard.

**Examples**

```

0x1p0 is 1
0xA.8p2 is 10.5*22

```

**Designated initializers in structures and arrays**

Any initialization of either a structure (struct or union) or an array can have a designation. A designation consists of one or more designators followed by an initializer. A designator for a structure is specified as *.elementname* and for an array [*constant index expression*]. Using designated initializers is not supported in C++.

### Examples

This definition shows a `struct` and its initialization using designators:

```
struct
{
    int i;
    int j;
    int k;
    int l;
    short array[10];
} u =
{
    .l = 6,           /* initialize l to 6 */
    .j = 6,           /* initialize j to 6 */
    8,               /* initialize k to 8 */
    .array[7] = 2,   /* initialize element 7 to 2 */
    .array[3] = 2,   /* initialize element 3 to 2 */
    5,               /* array[4] = 5 */
    .k = 4           /* reinitialize k to 4 */
};
```

Note that a designator specifies the destination element of the initialization. Note also that if one element is initialized more than once, it is the last initialization that will be used.

To initialize an element in a union other than the first, do like this:

```
union
{
    int i;
    float f;
} y = {.f = 5.0};
```

To set the size of an array by initializing the last element, do like this:

```
char array[] = {[10] = 'a'};
```

## MINOR LANGUAGE EXTENSIONS

This section lists and briefly describes minor extensions, that is, the relaxation of some standards issues and also some useful but minor syntax extensions:

- Arrays of incomplete types

An array can have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).

- Forward declaration of `enum` types  
The IAR Systems language extensions allow that you first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.
- Missing semicolon at end of `struct` or `union` specifier  
A warning is issued if the semicolon at the end of a `struct` or `union` specifier is missing.
- Null and `void`  
In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ISO/ANSI C, some operators allow such things, while others do not allow them.
- Casting pointers to integers in static initializers  
In an initializer, a pointer constant value can be cast to an integral type if the integral type is large enough to contain it. For more information about casting pointers, see *Casting*, page 186.
- Taking the address of a register variable  
In ISO/ANSI C, it is illegal to take the address of a variable specified as a register variable. The compiler allows this, but a warning is issued.
- Duplicated size and sign specifiers  
Should the size or sign specifiers be duplicated (for example, `short short` or `unsigned unsigned`), an error is issued.
- `long float` means `double`  
The type `long float` is accepted as a synonym for `double`.
- Repeated `typedef` declarations  
Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.
- Mixing pointer types  
Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.  
Assignment of a string constant to a pointer to any kind of character is allowed, and no warning is issued.
- Non-top level `const`  
Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). Comparing and taking the difference of such pointers is also allowed.

- `Non-lvalue` arrays  
A `non-lvalue` array expression is converted to a pointer to the first element of the array when it is used.
- Comments at the end of preprocessor directives  
This extension, which makes it legal to place text after preprocessor directives, is enabled, unless strict ISO/ANSI mode is used. The purpose of this language extension is to support compilation of legacy code; we do *not* recommend that you write new code in this fashion.
- An extra comma at the end of `enum` lists  
Placing an extra comma is allowed at the end of an `enum` list. In strict ISO/ANSI mode, a warning is issued.
- A label preceding a `}`  
In ISO/ANSI C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. The compiler issues a warning.

**Note:** This also applies to the labels of `switch` statements.

- Empty declarations  
An empty declaration (a semicolon by itself) is allowed, but a remark is issued (provided that remarks are enabled).
- Single-value initialization  
ISO/ANSI C requires that all initializer expressions of static arrays, `structs`, and `unions` are enclosed in braces.  
Single-value initializers are allowed to appear without braces, but a warning is issued. The compiler accepts this expression:

```
struct str
{
    int a;
} x = 10;
```

- Declarations in other scopes

External and static declarations in other scopes are visible. In the following example, the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

```
int test(int x)
{
    if (x)
    {
        extern int y;
        y = 1;
    }

    return y;
}
```

- Expanding function names into strings with the function as context

Use any of the symbols `__func__` or `__FUNCTION__` inside a function body to make the symbol expand into a string, with the function name as context. Use the symbol `__PRETTY_FUNCTION__` to also include the parameter types and return type. The result might, for example, look like this if you use the `__PRETTY_FUNCTION__` symbol:

```
"void func(char) "
```

These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see `-e`, page 164.

# Extended keywords

This chapter describes the extended keywords that support specific features of the 78K microcontrollers and the general syntax rules for the keywords. Finally the chapter gives a detailed description of each keyword.

For information about the address ranges of the different memory areas, see the chapter *Segment reference*.

---

## General syntax rules for extended keywords

To understand the syntax rules for the extended keywords, it is important to be familiar with some related concepts.

The compilers provide a set of attributes that can be used on functions or data objects to support specific features of the 78K microcontroller. There are two types of attributes—*type attributes* and *object attributes*:

- Type attributes affect the *external functionality* of the data object or function
- Object attributes affect the *internal functionality* of the data object or function.

The syntax for the keywords differs slightly depending on whether it is a type attribute or an object attribute, and whether it is applied to a data object or a function.

**Note:** Some of the keywords described in this chapter cannot be used with both the 78K0/78K0S Compiler and the 78K0R Compiler. This is mentioned in the descriptions of the individual keywords.

For information about how to use attributes to modify data, see the chapter *Data storage*. For information about how to use attributes to modify functions, see the chapter *Functions*. For detailed information about each attribute, see *Descriptions of extended keywords*, page 208.

**Note:** The extended keywords are only available when language extensions are enabled in the compiler.



In the IDE, language extensions are enabled by default.



Use the `-e` compiler option to enable language extensions. See `-e`, page 164 for additional information.

## TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that if you use a type attribute, it must be specified both when a function or data object is defined and when it is declared.

You can either place the type attributes directly in your source code, or use the pragma directive `#pragma type_attribute`.

Type attributes can be further divided into *memory type attributes* and *general type attributes*. Memory type attributes are referred to as simply *memory attributes* in the rest of the documentation.

### Memory attributes

A memory attribute corresponds to a certain logical or physical memory in the microcontroller.

- Available *function memory attributes*: `__callf`, `__callt`, `__near_func`, `__far_func`, `__banked`, and `__non_banked`
- Available *data memory attributes*: `__near`, `__far`, `__saddr`, and `__sfr`

Data objects, functions, and destinations of pointers or C++ references always have a memory attribute. If no attribute is explicitly specified in the declaration or by the pragma directive `#pragma type_attribute`, an appropriate default attribute is used. You can specify one memory attribute for each level of pointer indirection.

### General type attributes

These general type attributes are available:

- *Function type attributes* affect how the function should be called: `__interrupt`, `__no_save`, `__V2_call`, and `__V3_call`
- *Data type attributes*: `const`, `__no_bit_access`, and `volatile`

You can specify as many type attributes as required for each level of pointer indirection.

To read more about the type qualifiers `const` and `volatile`, see *Type qualifiers*, page 189.

### Syntax for type attributes used on data objects

In general, type attributes for data objects follow the same syntax as the type qualifiers `const` and `volatile`.

The following declaration assigns the `__far` type attribute to the variables `i` and `j`; in other words, the variable `i` and `j` is placed in far memory. The variables `k` and `l` behave in the same way:

```
__far int i, j;
int __far k, l;
```

Note that the attribute affects both identifiers.

This declaration of `i` and `j` is equivalent with the previous one:

```
#pragma type_attribute=__far
int i, j;
```

The advantage of using pragma directives for specifying keywords is that it offers you a method to make sure that the source code is portable. Note that the pragma directive has no effect if a memory attribute is already explicitly declared.

For more examples of using memory attributes, see *More examples*, page 17.

An easier way of specifying storage is to use type definitions. These two declarations are equivalent:

```
typedef char __far Byte;
typedef Byte *BytePtr;
Byte b;
BytePtr bp;
```

and

```
__far char b;
char __far *bp;
```

Note that `#pragma type_attribute` can be used together with a `typedef` declaration.

### Syntax for type attributes on data pointers

The syntax for declaring pointers using type attributes follows the same syntax as the type qualifiers `const` and `volatile`:

|                             |                                                       |
|-----------------------------|-------------------------------------------------------|
| <code>int __far * p;</code> | The <code>int</code> object is located in far memory. |
| <code>int * __far p;</code> | The pointer is located in far memory.                 |
| <code>__far int * p;</code> | The pointer is located in far memory.                 |

### Syntax for type attributes on functions

The syntax for using type attributes on functions differs slightly from the syntax of type attributes on data objects. For functions, the attribute must be placed either in front of the return type, or in parentheses, for example:

```
__interrupt void my_handler(void);
```

or

```
void (__interrupt my_handler)(void);
```

This declaration of `my_handler` is equivalent with the previous one:

```
#pragma type_attribute=__interrupt
void my_handler(void);
```

### Syntax for type attributes on function pointers

To declare a function pointer, use this syntax:

```
int (__far_func * fp)(double);
```

After this declaration, the function pointer `fp` points to far memory.

An easier way of specifying storage is to use type definitions:

```
typedef __far_func void FUNC_TYPE(int);
typedef FUNC_TYPE *FUNC_PTR_TYPE;
FUNC_TYPE func();
FUNC_PTR_TYPE funcptr;
```

Note that `#pragma type_attribute` can be used together with a `typedef` declaration.

## OBJECT ATTRIBUTES

Object attributes affect the internal functionality of functions and data objects, but not how the function is called or how the data is accessed. This means that an object attribute does not need to be present in the declaration of an object.

These object attributes are available:

- Object attributes that can be used for variables: `__no_init`
- Object attributes that can be used for functions and variables: `location`, `@`, and `__root`
- Object attributes that can be used for functions: `__intrinsic`, `__monitor`, and `__noreturn`.

You can specify as many object attributes as required for a specific function or data object.

For more information about `location` and `@`, see *Controlling data and function placement in memory*, page 126. For more information about `vector`, see *vector*, page 231.

### Syntax for object attributes

The object attribute must be placed in front of the type. For example, to place `myarray` in memory that is not initialized at startup:

```
__no_init int myarray[10];
```

The `#pragma object_attribute` directive can also be used. This declaration is equivalent to the previous one:

```
#pragma object_attribute=__no_init
int myarray[10];
```

**Note:** Object attributes cannot be used in combination with the `typedef` keyword.

---

## Summary of extended keywords

This table summarizes the extended keywords:

| Extended keyword             | Description                                                                       |
|------------------------------|-----------------------------------------------------------------------------------|
| <code>__banked</code>        | Controls the storage of functions. 78K0/78K0S only                                |
| <code>__callf</code>         | Controls the storage of functions. 78K0 only                                      |
| <code>__callt</code>         | Controls the storage of functions                                                 |
| <code>__far</code>           | Controls the storage of data objects. 78K0R only                                  |
| <code>__far_func</code>      | Controls the storage of functions. 78K0R only                                     |
| <code>__interrupt</code>     | Supports interrupt functions                                                      |
| <code>__intrinsic</code>     | Reserved for compiler internal use only                                           |
| <code>__monitor</code>       | Supports atomic execution of a function                                           |
| <code>__near</code>          | Controls the storage of data objects                                              |
| <code>__near_func</code>     | Controls the storage of functions. 78K0R only                                     |
| <code>__no_bit_access</code> | Inhibits the use of bit instructions when accessing individual bits of the object |
| <code>__no_init</code>       | Supports non-volatile memory                                                      |
| <code>__non_banked</code>    | Controls the storage of functions. 78K0/78K0S only                                |
| <code>__noreturn</code>      | Informs the compiler that the function will not return                            |

Table 40: Extended keywords summary

| Extended keyword       | Description                                                                               |
|------------------------|-------------------------------------------------------------------------------------------|
| <code>__no_save</code> | Specifies a function to not save and restore the workseg area that it has used            |
| <code>__root</code>    | Ensures that a function or variable is included in the object code even if unused         |
| <code>__saddr</code>   | Controls the storage of data objects                                                      |
| <code>__sfr</code>     | Controls the storage of data objects                                                      |
| <code>__V2_call</code> | Specifies a calling convention compatible with version 2 of the compiler. 78K0/78K0S only |
| <code>__V3_call</code> | Specifies a calling convention compatible with version 3 of the compiler. 78K0/78K0S only |

Table 40: Extended keywords summary (Continued)

## Descriptions of extended keywords

These sections give detailed information about each extended keyword.

### `__banked`

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on functions, see <i>Type attributes</i> , page 204.                                                                                                                                                                                                                                                                                                                             |
| Description         | <p>The <code>__banked</code> memory attribute overrides the default storage of functions given by the selected code model and places individual functions in banked memory, which means a <code>CALL</code> instruction will be used at function calls. You can also use the <code>__banked</code> attribute to create a pointer explicitly pointing to an object located in the banked memory.</p> <p>The function can be called from any XLINK segment.</p> |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0-0xFFFFF (16 Mbytes)</li> <li>● Maximum size: 16 Mbytes.</li> <li>● Pointer size: 3 bytes</li> </ul> <p><b>Note:</b> This keyword cannot be used with the 78K0R Compiler. If you have 78K0/78K0S code containing this keyword that you need to compile using the 78K0R Compiler, you can compile it if you—as a workaround—define the following:</p> <pre>#define __banked __far_func</pre>          |
| Example             | <code>__banked void myfunction(void);</code>                                                                                                                                                                                                                                                                                                                                                                                                                  |

See also *Banked functions (78K0/78K0S only)*, page 27.

## **\_\_callf**

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on functions, see <i>Type attributes</i> , page 204.                                                                                                                                                                                                                                                                                                                          |
| Description         | The <code>__callf</code> memory attribute overrides the default storage of functions given by the selected code model and specifies that individual functions are called with a <code>CALLF</code> instruction. The function can be placed in any XLINK segment.                                                                                                                                                                                           |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0x0800–0x0FFF (2 Kbytes)</li> <li>● Maximum size: 2048 Kbytes.</li> <li>● Pointer size: 2 bytes</li> </ul> <p><b>Note:</b> This keyword can only be used with the 78K0 core. If you have 78K0 source code containing this keyword that you need to compile for the 78K0R or 78K0S core, you can compile it if you as a workaround define the following:</p> <pre>#define __callf __near_func</pre> |
| Example             | <code>__callf void myfunction(void);</code>                                                                                                                                                                                                                                                                                                                                                                                                                |

## **\_\_callt**

|                     |                                                                                                                                                                                                                                                                           |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on functions, see <i>Type attributes</i> , page 204.                                                                                                                                         |
| Description         | The <code>__callt</code> memory attribute overrides the default storage of functions given by the selected code model and specifies that individual functions are called with a vectored <code>CALLT</code> instruction. The function can be placed in any XLINK segment. |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0–0xFFFF (64 Kbytes)</li> <li>● Maximum size: 65535 bytes.</li> <li>● Pointer size: 2 bytes</li> </ul>                                                                                                            |
| Example             | <code>__callt void myfunction(void);</code>                                                                                                                                                                                                                               |

## **\_\_far**

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 204.                                                                                                                                                                                                                                                                                               |
| Description         | The <code>__far</code> memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in far memory (78K0R only). You can also use the <code>__far</code> attribute to create a pointer explicitly pointing to an object located in far memory.                                                                                                        |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0-FFFFFF (1 Mbyte)</li> <li>● Maximum object size: 65535 bytes. An object cannot cross a 64-Kbyte boundary.</li> <li>● Pointer size: 3 bytes. Arithmetics is only performed on the two lower bytes, except comparison which is always performed on the entire 24-bit address.</li> </ul> <p><b>Note:</b> This keyword can <i>only</i> be used with the 78K0R Compiler.</p> |
| Example             | <pre>__far int x;</pre>                                                                                                                                                                                                                                                                                                                                                                                                            |
| See also            | <i>Memory types</i> , page 13.                                                                                                                                                                                                                                                                                                                                                                                                     |

## **\_\_far\_func**

|                     |                                                                                                                                                                                                                                                                                                                    |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on functions, see <i>Type attributes</i> , page 204.                                                                                                                                                                                  |
| Description         | The <code>__far_func</code> memory attribute overrides the default storage of functions given by the selected code model and places individual functions in farfunc memory. You can also use the <code>__far_func</code> attribute to create a pointer explicitly pointing to an object located in farfunc memory. |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0-0xFFFFF (1 Mbyte)</li> <li>● Maximum size: 65535 bytes. An object cannot cross a 64-Kbyte boundary.</li> <li>● Pointer size: 3 bytes</li> </ul>                                                                                                          |
| Example             | <pre>__far_func void myfunction(void);</pre> <p><b>Note:</b> This keyword can <i>only</i> be used with the 78K0R Compiler.</p>                                                                                                                                                                                     |

## \_\_interrupt

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for type attributes that can be used on functions, see <i>Type attributes</i> , page 204.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Description | <p>The <code>__interrupt</code> keyword specifies interrupt functions. To specify one or several interrupt vectors, use the <code>#pragma vector</code> directive. The range of the interrupt vectors depends on the device used. It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table.</p> <p>An interrupt function must have a <code>void</code> return type and cannot have any parameters.</p> <p>The header file <code>iodevice.h</code>, where <code>device</code> corresponds to the selected device, contains predefined names for the existing interrupt vectors.</p> |
| Example     | <pre>#pragma vector=0x14 __interrupt void my_interrupt_handler(void);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| See also    | <i>Interrupt functions</i> , page 23, <i>vector</i> , page 231, <i>INTVEC</i> , page 260.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |

## \_\_intrinsic

|             |                                                                                  |
|-------------|----------------------------------------------------------------------------------|
| Description | The <code>__intrinsic</code> keyword is reserved for compiler internal use only. |
|-------------|----------------------------------------------------------------------------------|

## \_\_monitor

|             |                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for object attributes that can be used on functions, see <i>Object attributes</i> , page 206.                                                                                                                                                                                                                                       |
| Description | The <code>__monitor</code> keyword causes interrupts to be disabled during execution of the function. This allows atomic operations to be performed, such as operations on semaphores that control access to resources by multiple processes. A function declared with the <code>__monitor</code> keyword is equivalent to any other function in all other respects. |
| Example     | <pre>__monitor int get_lock(void);</pre>                                                                                                                                                                                                                                                                                                                             |
| See also    | <i>Monitor functions</i> , page 24. Read also about the intrinsic functions <code>__disable_interrupt</code> , page 234, <code>__enable_interrupt</code> , page 234, <code>__get_interrupt_state</code> , page 234, and <code>__set_interrupt_state</code> , page 235.                                                                                               |

## **\_\_near**

|                     |                                                                                                                                                                                                                                                                                                                    |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 204.                                                                                                                                                                               |
| Description         | The <code>__near</code> memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in near memory. You can also use the <code>__near</code> attribute to create a pointer explicitly pointing to an object located in near memory. |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0x0000–0xFFFF (78K0 and 78K0S)<br/>0xF0000–0xFFFFF (78K0R)</li> <li>● Maximum object size: 65535 bytes.</li> <li>● Pointer size: 2 bytes.</li> </ul>                                                                                                       |
| Example             | <code>__near int x;</code>                                                                                                                                                                                                                                                                                         |
| See also            | <i>Memory types</i> , page 13.                                                                                                                                                                                                                                                                                     |

## **\_\_near\_func**

|                     |                                                                                                                                                                                                                                                                                                                        |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on functions, see <i>Type attributes</i> , page 204.                                                                                                                                                                                      |
| Description         | The <code>__near_func</code> memory attribute overrides the default storage of functions given by the selected code model and places individual functions in nearfunc memory. You can also use the <code>__near_func</code> attribute to create a pointer explicitly pointing to an object located in nearfunc memory. |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0–0xFFFF (64 Kbytes)</li> <li>● Maximum size: 65535 bytes.</li> <li>● Pointer size: 2 bytes</li> </ul>                                                                                                                                                         |
| Example             | <code>__near_func void myfunction(void);</code>                                                                                                                                                                                                                                                                        |
|                     | <b>Note:</b> This keyword can <i>only</i> be used with the 78K0R Compiler.                                                                                                                                                                                                                                             |

**\_\_no\_bit\_access**

|             |                                                                                                                                                |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for type attributes that can be used on data, see <i>Type attributes</i> , page 204.                          |
| Description | For data objects declared <code>__no_bit_access</code> , the use of bit instructions is suppressed individual bits of the object are accessed. |

**\_\_no\_init**

|             |                                                                                                                                                                                         |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for object attributes, see <i>Object attributes</i> , page 206.                                                                                        |
| Description | Use the <code>__no_init</code> keyword to place a data object in non-volatile memory. This means that the initialization of the variable, for example at system startup, is suppressed. |
| Example     | <pre>__no_init int myarray[10];</pre>                                                                                                                                                   |

**\_\_non\_banked**

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on functions, see <i>Type attributes</i> , page 204.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Description         | <p>The <code>__non_banked</code> memory attribute overrides the default storage of functions given by the selected code model and places individual functions in non-banked memory, which means a standard <code>CALL</code> instruction will be used at function calls. You can also use the <code>__non_banked</code> attribute to create a pointer explicitly pointing to an object located in non-banked memory.</p> <p>Functions in non-banked memory can be called from any <code>XLINK</code> segment. For a banked application, this memory attribute can be useful for functions that must reside in non-banked memory, for instance, interrupt functions.</p> |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0-0xFFFF (64 Kbytes)</li> <li>● Maximum size: 65535 bytes.</li> <li>● Pointer size: 2 bytes</li> </ul> <p><b>Note:</b> This keyword cannot be used with the 78K0R Compiler. If you have 78K0/78K0S source code containing this keyword that you need to compile using the 78K0R Compiler, you can compile it if you as a workaround define the following:</p> <pre>#define __non_banked __near_func</pre>                                                                                                                                                                                                       |
| Example             | <pre>__non_banked void myfunction(void);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |

## **\_\_noreturn**

|             |                                                                                                                                                                                                                                                                                                        |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for object attributes, see <i>Object attributes</i> , page 206.                                                                                                                                                                                                       |
| Description | The <code>__noreturn</code> keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are <code>abort</code> and <code>exit</code> . |
| Example     | <pre>__noreturn void terminate(void);</pre>                                                                                                                                                                                                                                                            |

## **\_\_no\_save**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for type attributes that can be used on functions, see <i>Type attributes</i> , page 204.                                                                                                                                                                                                                                                                                                                             |
| Description | <p>Functions that are declared <code>__no_save</code> will not save and restore the workseg area that it has used.</p> <p>Functions normally save the used area of the <code>WRKSEG</code> segment on the stack. The <code>__no_save</code> keyword cases the function not to save this information when the <code>--workseg_area</code> option is used.</p> <p>This keyword has not effect unless the <code>--workseg_area</code> option is used.</p> |

## **\_\_root**

|             |                                                                                                                                                                                                                                                               |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for object attributes, see <i>Object attributes</i> , page 206.                                                                                                                                                              |
| Description | A function or variable with the <code>__root</code> attribute is kept whether or not it is referenced from the rest of the application, provided its module is included. Program modules are always included and library modules are only included if needed. |
| Example     | <pre>__root int myarray[10];</pre>                                                                                                                                                                                                                            |
| See also    | To read more about modules, segments, and the link process, see the <i>IAR Linker and Library Tools Reference Guide</i> .                                                                                                                                     |

**\_\_saddr**

|                     |                                                                                                                                                                                                                                                                                                                                                  |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 204.                                                                                                                                                                                                             |
| Description         | The <code>__saddr</code> memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in <code>saddr</code> memory. You can also use the <code>__saddr</code> attribute to create a pointer explicitly pointing to an object located in <code>saddr</code> memory. |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: <code>0xFE20-0xFF1F</code> (78K0 and 78K0S)<br/><code>0xFFE20-0xFFFF1F</code> (78K0R)</li> <li>● Maximum object size: 256 bytes.</li> <li>● Pointer size: 2 bytes.</li> </ul>                                                                                                            |
| Example             | <code>__saddr int x;</code>                                                                                                                                                                                                                                                                                                                      |
| See also            | <i>Memory types</i> , page 13.                                                                                                                                                                                                                                                                                                                   |

**\_\_sfr**

|                     |                                                                                                                                                                                                                                                                      |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 204.                                                                                                                                 |
| Description         | The <code>__sfr</code> memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in the <code>sfr</code> area.                                                                      |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: <code>0xFF00-0xFFFF</code> (78K0 and 78K0S)<br/><code>0xFFFF00-0xFFFFF</code> (78K0R)</li> <li>● Maximum object size: Typically, 256 bytes (varies with the device).</li> <li>● Pointer size: n/a</li> </ul> |
| Example             | <code>__saddr int x;</code>                                                                                                                                                                                                                                          |
| See also            | <i>Memory types</i> , page 13.                                                                                                                                                                                                                                       |

## **\_\_V2\_call**

|             |                                                                                                                                                                                                  |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for type attributes that can be used on functions, see <i>Type attributes</i> , page 204.                                                                       |
| Description | Functions that are declared <code>__v2_call</code> uses the same calling convention as used in version 2.x of the compiler.<br><b>Note:</b> This keyword cannot be used with the 78K0R Compiler. |
| See also    | <i>Calling convention</i> , page 97.                                                                                                                                                             |

## **\_\_V3\_call**

|             |                                                                                                                                                                                                  |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for type attributes that can be used on functions, see <i>Type attributes</i> , page 204.                                                                       |
| Description | Functions that are declared <code>__v3_call</code> uses the same calling convention as used in version 3.x of the compiler.<br><b>Note:</b> This keyword cannot be used with the 78K0R Compiler. |
| See also    | <i>Calling convention</i> , page 97.                                                                                                                                                             |

# Pragma directives

This chapter describes the pragma directives of the compilers.

The `#pragma` directive is defined by the ISO/ANSI C standard and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

The pragma directives control the behavior of the compiler, for example how it allocates memory for variables and functions, whether it allows extended keywords, and whether it outputs warning messages.

The pragma directives are always enabled in the compiler.

---

## Summary of pragma directives

This table lists the pragma directives of the compiler that can be used either with the `#pragma` preprocessor directive or the `_Pragma()` preprocessor operator:

| Pragma directive                     | Description                                                                                                    |
|--------------------------------------|----------------------------------------------------------------------------------------------------------------|
| <code>bank</code>                    | Specifies the register bank number of the immediately following interrupt function. 78K0 and 78K0R cores only. |
| <code>basic_template_matching</code> | Makes a template function fully memory-attribute aware                                                         |
| <code>bitfields</code>               | Controls the order of bitfield members                                                                         |
| <code>constseg</code>                | Places constant variables in a named segment                                                                   |
| <code>data_alignment</code>          | Gives a variable a higher (more strict) alignment                                                              |
| <code>dataseg</code>                 | Places variables in a named segment                                                                            |
| <code>diag_default</code>            | Changes the severity level of diagnostic messages                                                              |
| <code>diag_error</code>              | Changes the severity level of diagnostic messages                                                              |
| <code>diag_remark</code>             | Changes the severity level of diagnostic messages                                                              |
| <code>diag_suppress</code>           | Suppresses diagnostic messages                                                                                 |
| <code>diag_warning</code>            | Changes the severity level of diagnostic messages                                                              |
| <code>include_alias</code>           | Specifies an alias for an include file                                                                         |
| <code>inline</code>                  | Inlines a function                                                                                             |
| <code>language</code>                | Controls the IAR Systems language extensions                                                                   |

*Table 41: Pragma directives summary*

| Pragma directive | Description                                                                                                |
|------------------|------------------------------------------------------------------------------------------------------------|
| location         | Specifies the absolute address of a variable, or places groups of functions or variables in named segments |
| message          | Prints a message                                                                                           |
| object_attribute | Changes the definition of a variable or a function                                                         |
| optimize         | Specifies the type and level of an optimization                                                            |
| pack             | Specifies the alignment of structures and union members                                                    |
| __printf_args    | Verifies that a function with a printf-style format string is called with the correct arguments            |
| required         | Ensures that a symbol that is needed by another symbol is included in the linked output                    |
| rtmodel          | Adds a runtime model attribute to the module                                                               |
| __scanf_args     | Verifies that a function with a scanf-style format string is called with the correct arguments             |
| segment          | Declares a segment name to be used by intrinsic functions                                                  |
| type_attribute   | Changes the declaration and definitions of a variable or function                                          |
| unroll           | Unrolls loops                                                                                              |
| vector           | Specifies the vector of an interrupt function                                                              |

Table 41: Pragma directives summary (Continued)

**Note:** For portability reasons, see also *Recognized pragma directives (6.8.6)*, page 273, and the *78K IAR Embedded Workbench Migration Guide*.

## Descriptions of pragma directives

This section gives detailed information about each pragma directive.

### bank

Syntax `#pragma bank=bank_number`

Parameters

*bank\_number* A constant integer expression that denotes one of the register banks 0–3.

|             |                                                                                                                                                                                                                                                                                                                                                                                |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Use this pragma directive to declare an interrupt function that switches to another register bank before starting the execution of the function. The <code>#pragma bank</code> directive controls the register bank number (0–3) of the immediately following interrupt function.<br><br><b>Note:</b> This pragma directive cannot be used in code written for the 78K0S core. |
| Example     | For an example, see <i>Interrupt functions</i> , page 23.                                                                                                                                                                                                                                                                                                                      |
| See also    | <i>Banked functions (78K0/78K0S only)</i> , page 27.                                                                                                                                                                                                                                                                                                                           |

## basic\_template\_matching

|             |                                                                                                                                                                                                                                                                                                                                  |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma basic_template_matching</code>                                                                                                                                                                                                                                                                                     |
| Description | Use this pragma directive in front of a template function declaration to make the function fully memory-attribute aware, in the rare cases where this is useful. That template function will then match the template without the modifications described in <i>Templates and data memory attributes (78K0R only)</i> , page 116. |
| Example     | <pre>#pragma basic_template_matching template&lt;typename T&gt; void fun(T *);  fun((int __near *) 0); /* Template parameter T becomes                        int __near */</pre>                                                                                                                                                |

## bitfields

|                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                       |                                                                                         |                      |                                                                                         |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|-----------------------------------------------------------------------------------------|----------------------|-----------------------------------------------------------------------------------------|
| Syntax                | <code>#pragma bitfields={reversed default}</code>                                                                                                                                                                                                                                                                                                                                                                                                                                |                       |                                                                                         |                      |                                                                                         |
| Parameters            | <table> <tr> <td><code>reversed</code></td> <td>Bitfield members are placed from the most significant bit to the least significant bit.</td> </tr> <tr> <td><code>default</code></td> <td>Bitfield members are placed from the least significant bit to the most significant bit.</td> </tr> </table>                                                                                                                                                                            | <code>reversed</code> | Bitfield members are placed from the most significant bit to the least significant bit. | <code>default</code> | Bitfield members are placed from the least significant bit to the most significant bit. |
| <code>reversed</code> | Bitfield members are placed from the most significant bit to the least significant bit.                                                                                                                                                                                                                                                                                                                                                                                          |                       |                                                                                         |                      |                                                                                         |
| <code>default</code>  | Bitfield members are placed from the least significant bit to the most significant bit.                                                                                                                                                                                                                                                                                                                                                                                          |                       |                                                                                         |                      |                                                                                         |
| Description           | Use this pragma directive to control the order of bitfield members.<br><br>By default, the compiler places bitfield members from the least significant bit to the most significant bit in the container type. Use the <code>#pragma bitfields=reversed</code> directive to place the bitfield members from the most significant to the least significant bit. This setting remains active until you turn it off again with the <code>#pragma bitfields=default</code> directive. |                       |                                                                                         |                      |                                                                                         |

See also *Bitfields*, page 183.

## constseg

|             |                                                                                                                                                                                                                                                                                                                                                                            |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma constseg=[<i>__memoryattribute</i> ]{<i>SEGMENT_NAME</i> default}</code>                                                                                                                                                                                                                                                                                     |
| Parameters  | <p><i>__memoryattribute</i>    An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.</p> <p><i>SEGMENT_NAME</i>        A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.</p> <p>default                Uses the default segment for constants.</p> |
| Description | Use this pragma directive to place constant variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The setting remains active until you turn it off again with the <code>#pragma constseg=default</code> directive.                                                                                        |
| Example     | <pre>#pragma constseg=__saddr MY_CONSTANTS const int factorySettings[] = {42, 15, -128, 0}; #pragma constseg=default</pre>                                                                                                                                                                                                                                                 |

## data\_alignment

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma data_alignment=<i>expression</i></code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Parameters  | <i>expression</i> A constant which must be a power of two (1, 2, 4, etc.).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| Description | <p>Use this pragma directive to give a variable a higher (more strict) alignment of the start address than it would otherwise have. This directive can be used on variables with static and automatic storage duration.</p> <p>When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.</p> <p><b>Note:</b> Normally, the size of a variable is a multiple of its alignment. The <code>data_alignment</code> directive only affects the alignment of the variable's start address, and not its size, and can thus be used for creating situations where the size is not a multiple of the alignment.</p> |

## dataseg

|                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                |                                                                                                                            |                           |                                                                                                      |                      |                           |
|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------|----------------------------------------------------------------------------------------------------------------------------|---------------------------|------------------------------------------------------------------------------------------------------|----------------------|---------------------------|
| Syntax                         | <code>#pragma dataseg=[__memoryattribute ]{SEGMENT_NAME default}</code>                                                                                                                                                                                                                                                                                                                                                                         |                                |                                                                                                                            |                           |                                                                                                      |                      |                           |
| Parameters                     | <table> <tr> <td><code>__memoryattribute</code></td> <td>An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.</td> </tr> <tr> <td><code>SEGMENT_NAME</code></td> <td>A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.</td> </tr> <tr> <td><code>default</code></td> <td>Uses the default segment.</td> </tr> </table> | <code>__memoryattribute</code> | An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used. | <code>SEGMENT_NAME</code> | A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker. | <code>default</code> | Uses the default segment. |
| <code>__memoryattribute</code> | An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.                                                                                                                                                                                                                                                                                                                      |                                |                                                                                                                            |                           |                                                                                                      |                      |                           |
| <code>SEGMENT_NAME</code>      | A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.                                                                                                                                                                                                                                                                                                                                            |                                |                                                                                                                            |                           |                                                                                                      |                      |                           |
| <code>default</code>           | Uses the default segment.                                                                                                                                                                                                                                                                                                                                                                                                                       |                                |                                                                                                                            |                           |                                                                                                      |                      |                           |
| Description                    | Use this pragma directive to place variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The variable will not be initialized at startup, and can for this reason not have an initializer, which means it must be declared <code>__no_init</code> . The setting remains active until you turn it off again with the <code>#pragma constseg=default</code> directive.           |                                |                                                                                                                            |                           |                                                                                                      |                      |                           |
| Example                        | <pre>#pragma dataseg=__saddr MY_SEGMENT __no_init char myBuffer[1000]; #pragma dataseg=default</pre>                                                                                                                                                                                                                                                                                                                                            |                                |                                                                                                                            |                           |                                                                                                      |                      |                           |

## diag\_default

|                  |                                                                                                                                                                                                                                                                                                                                       |                  |                                                                                         |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|-----------------------------------------------------------------------------------------|
| Syntax           | <code>#pragma diag_default=tag[, tag, ...]</code>                                                                                                                                                                                                                                                                                     |                  |                                                                                         |
| Parameters       | <table> <tr> <td><code>tag</code></td> <td>The number of a diagnostic message, for example the message number <code>Pe117</code>.</td> </tr> </table>                                                                                                                                                                                 | <code>tag</code> | The number of a diagnostic message, for example the message number <code>Pe117</code> . |
| <code>tag</code> | The number of a diagnostic message, for example the message number <code>Pe117</code> .                                                                                                                                                                                                                                               |                  |                                                                                         |
| Description      | Use this pragma directive to change the severity level back to the default, or to the severity level defined on the command line by any of the options <code>--diag_error</code> , <code>--diag_remark</code> , <code>--diag_suppress</code> , or <code>--diag_warnings</code> , for the diagnostic messages specified with the tags. |                  |                                                                                         |
| See also         | <i>Diagnostics</i> , page 146.                                                                                                                                                                                                                                                                                                        |                  |                                                                                         |

## diag\_error

|             |                                                                                                                                    |            |                                                                           |
|-------------|------------------------------------------------------------------------------------------------------------------------------------|------------|---------------------------------------------------------------------------|
| Syntax      | <code>#pragma diag_error=tag[, tag, ...]</code>                                                                                    |            |                                                                           |
| Parameters  | <table> <tr> <td><i>tag</i></td> <td>The number of a diagnostic message, for example the message number Pe117.</td> </tr> </table> | <i>tag</i> | The number of a diagnostic message, for example the message number Pe117. |
| <i>tag</i>  | The number of a diagnostic message, for example the message number Pe117.                                                          |            |                                                                           |
| Description | Use this pragma directive to change the severity level to <code>error</code> for the specified diagnostics.                        |            |                                                                           |
| See also    | <i>Diagnostics</i> , page 146.                                                                                                     |            |                                                                           |

## diag\_remark

|             |                                                                                                                                    |            |                                                                           |
|-------------|------------------------------------------------------------------------------------------------------------------------------------|------------|---------------------------------------------------------------------------|
| Syntax      | <code>#pragma diag_remark=tag[, tag, ...]</code>                                                                                   |            |                                                                           |
| Parameters  | <table> <tr> <td><i>tag</i></td> <td>The number of a diagnostic message, for example the message number Pe177.</td> </tr> </table> | <i>tag</i> | The number of a diagnostic message, for example the message number Pe177. |
| <i>tag</i>  | The number of a diagnostic message, for example the message number Pe177.                                                          |            |                                                                           |
| Description | Use this pragma directive to change the severity level to <code>remark</code> for the specified diagnostic messages.               |            |                                                                           |
| See also    | <i>Diagnostics</i> , page 146.                                                                                                     |            |                                                                           |

## diag\_suppress

|             |                                                                                                                                    |            |                                                                           |
|-------------|------------------------------------------------------------------------------------------------------------------------------------|------------|---------------------------------------------------------------------------|
| Syntax      | <code>#pragma diag_suppress=tag[, tag, ...]</code>                                                                                 |            |                                                                           |
| Parameters  | <table> <tr> <td><i>tag</i></td> <td>The number of a diagnostic message, for example the message number Pe117.</td> </tr> </table> | <i>tag</i> | The number of a diagnostic message, for example the message number Pe117. |
| <i>tag</i>  | The number of a diagnostic message, for example the message number Pe117.                                                          |            |                                                                           |
| Description | Use this pragma directive to suppress the specified diagnostic messages.                                                           |            |                                                                           |
| See also    | <i>Diagnostics</i> , page 146.                                                                                                     |            |                                                                           |

## diag\_warning

|             |                                                                                                                       |                                                                                         |
|-------------|-----------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma diag_warning=tag[, tag, ...]</code>                                                                     |                                                                                         |
| Parameters  | <i>tag</i>                                                                                                            | The number of a diagnostic message, for example the message number <code>Pe826</code> . |
| Description | Use this pragma directive to change the severity level to <code>warning</code> for the specified diagnostic messages. |                                                                                         |
| See also    | <i>Diagnostics</i> , page 146.                                                                                        |                                                                                         |

## include\_alias

|             |                                                                                                                                                                                                                                                                                                                                                                                               |                                                                  |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------|
| Syntax      | <code>#pragma include_alias ("orig_header" , "subst_header")</code><br><code>#pragma include_alias (&lt;orig_header&gt; , &lt;subst_header&gt;)</code>                                                                                                                                                                                                                                        |                                                                  |
| Parameters  | <i>orig_header</i>                                                                                                                                                                                                                                                                                                                                                                            | The name of a header file for which you want to create an alias. |
|             | <i>subst_header</i>                                                                                                                                                                                                                                                                                                                                                                           | The alias for the original header file.                          |
| Description | Use this pragma directive to provide an alias for a header file. This is useful for substituting one header file with another, and for specifying an absolute path to a relative file.<br><br>This pragma directive must appear before the corresponding <code>#include</code> directives and <code>subst_header</code> must match its corresponding <code>#include</code> directive exactly. |                                                                  |
| Example     | <code>#pragma include_alias (&lt;stdio.h&gt; , &lt;C:\MyHeaders\stdio.h&gt;)</code><br><code>#include &lt;stdio.h&gt;</code><br><br>This example will substitute the relative file <code>stdio.h</code> with a counterpart located according to the specified path.                                                                                                                           |                                                                  |
| See also    | <i>Include file search procedure</i> , page 144.                                                                                                                                                                                                                                                                                                                                              |                                                                  |

## inline

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                     |                                                         |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|---------------------------------------------------------|
| Syntax              | <code>#pragma inline[=forced]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                     |                                                         |
| Parameters          | <table> <tr> <td><code>forced</code></td> <td>Disables the compiler's heuristics and forces inlining.</td> </tr> </table>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | <code>forced</code> | Disables the compiler's heuristics and forces inlining. |
| <code>forced</code> | Disables the compiler's heuristics and forces inlining.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                     |                                                         |
| Description         | <p>Use this pragma directive to advise the compiler that the function whose declaration follows immediately after the directive should be inlined—that is, expanded into the body of the calling function. Whether the inlining actually occurs is subject to the compiler's heuristics.</p> <p>This is similar to the C++ keyword <code>inline</code>, but has the advantage of being available in C code.</p> <p>Specifying <code>#pragma inline=forced</code> disables the compiler's heuristics and forces inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question (like <code>printf</code>), an error message is emitted.</p> <p><b>Note:</b> Because specifying <code>#pragma inline=forced</code> disables the compiler's heuristics, including the inlining heuristics, the function declared immediately after the directive will not be inlined on optimization levels None or Low. No error or warning message will be emitted.</p> |                     |                                                         |

## language

|                       |                                                                                                                                                                                                                                                                                                |                       |                                                                                                                |                      |                                                           |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|----------------------------------------------------------------------------------------------------------------|----------------------|-----------------------------------------------------------|
| Syntax                | <code>#pragma language={extended default}</code>                                                                                                                                                                                                                                               |                       |                                                                                                                |                      |                                                           |
| Parameters            | <table> <tr> <td><code>extended</code></td> <td>Turns on the IAR Systems language extensions and turns off the <code>--strict_ansi</code> command line option.</td> </tr> <tr> <td><code>default</code></td> <td>Uses the language settings specified by compiler options.</td> </tr> </table> | <code>extended</code> | Turns on the IAR Systems language extensions and turns off the <code>--strict_ansi</code> command line option. | <code>default</code> | Uses the language settings specified by compiler options. |
| <code>extended</code> | Turns on the IAR Systems language extensions and turns off the <code>--strict_ansi</code> command line option.                                                                                                                                                                                 |                       |                                                                                                                |                      |                                                           |
| <code>default</code>  | Uses the language settings specified by compiler options.                                                                                                                                                                                                                                      |                       |                                                                                                                |                      |                                                           |
| Description           | Use this pragma directive to enable the compiler language extensions or for using the language settings specified on the command line.                                                                                                                                                         |                       |                                                                                                                |                      |                                                           |

## location

|             |                                                                                                                                                                                                                                                                                                                                                                                                             |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma location={<i>address</i> <i>NAME</i>}</code>                                                                                                                                                                                                                                                                                                                                                  |
| Parameters  | <p><i>address</i>                    The absolute address of the global or static variable for which you want an absolute location.</p> <p><i>NAME</i>                        A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.</p>                                                                                                                      |
| Description | Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variable must be declared either <code>__no_init</code> or <code>const</code> . Alternatively, the directive can take a string specifying a segment for placing either a variable or a function whose declaration follows the pragma directive. |
| Example     | <pre>#pragma location=0xFF20 __no_init volatile char PORT1; /* PORT1 is located at address                                 0xFF20 */  #pragma location="foo" char PORT1; /* PORT1 is located in segment foo */  /* A better way is to use a corresponding mechanism */ #define FLASH _Pragma("location=\"FLASH\"") ... FLASH int i; /* i is placed in the FLASH segment */</pre>                            |
| See also    | <i>Controlling data and function placement in memory</i> , page 126.                                                                                                                                                                                                                                                                                                                                        |

## message

|             |                                                                                                                         |
|-------------|-------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma message(<i>message</i>)</code>                                                                            |
| Parameters  | <i>message</i> The message that you want to direct to the standard output stream.                                       |
| Description | Use this pragma directive to make the compiler print a message to the standard output stream when the file is compiled. |
| Example:    | <pre>#ifdef TESTING #pragma message("Testing") #endif</pre>                                                             |

## object\_attribute

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma object_attribute=object_attribute[, object_attribute, ...]</code>                                                                                                                                                                                                                                                                                                                                                      |
| Parameters  | For a list of object attributes that can be used with this pragma directive, see <i>Object attributes</i> , page 206.                                                                                                                                                                                                                                                                                                                |
| Description | Use this pragma directive to declare a variable or a function with an object attribute. This directive affects the definition of the identifier that follows immediately after the directive. The object is modified, not its type. Unlike the directive <code>#pragma type_attribute</code> that specifies the storing and accessing of a variable or function, it is not necessary to specify an object attribute in declarations. |
| Example     | <pre>#pragma object_attribute=__no_init char bar;</pre>                                                                                                                                                                                                                                                                                                                                                                              |
| See also    | <i>General syntax rules for extended keywords</i> , page 203.                                                                                                                                                                                                                                                                                                                                                                        |

## optimize

|                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                  |                                                                                       |                                   |                                     |                             |                       |                     |                                            |                        |                             |                      |                                     |                        |                          |
|-----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------|---------------------------------------------------------------------------------------|-----------------------------------|-------------------------------------|-----------------------------|-----------------------|---------------------|--------------------------------------------|------------------------|-----------------------------|----------------------|-------------------------------------|------------------------|--------------------------|
| Syntax                            | <code>#pragma optimize=param[ param...]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                  |                                                                                       |                                   |                                     |                             |                       |                     |                                            |                        |                             |                      |                                     |                        |                          |
| Parameters                        | <table> <tr> <td><code>balanced size speed</code></td> <td>Optimizes balanced between speed and size, optimizes for size, or optimizes for speed</td> </tr> <tr> <td><code>none low medium high</code></td> <td>Specifies the level of optimization</td> </tr> <tr> <td><code>no_code_motion</code></td> <td>Turns off code motion</td> </tr> <tr> <td><code>no_cse</code></td> <td>Turns off common subexpression elimination</td> </tr> <tr> <td><code>no_inline</code></td> <td>Turns off function inlining</td> </tr> <tr> <td><code>no_tbaa</code></td> <td>Turns off type-based alias analysis</td> </tr> <tr> <td><code>no_unroll</code></td> <td>Turns off loop unrolling</td> </tr> </table> | <code>balanced size speed</code> | Optimizes balanced between speed and size, optimizes for size, or optimizes for speed | <code>none low medium high</code> | Specifies the level of optimization | <code>no_code_motion</code> | Turns off code motion | <code>no_cse</code> | Turns off common subexpression elimination | <code>no_inline</code> | Turns off function inlining | <code>no_tbaa</code> | Turns off type-based alias analysis | <code>no_unroll</code> | Turns off loop unrolling |
| <code>balanced size speed</code>  | Optimizes balanced between speed and size, optimizes for size, or optimizes for speed                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                  |                                                                                       |                                   |                                     |                             |                       |                     |                                            |                        |                             |                      |                                     |                        |                          |
| <code>none low medium high</code> | Specifies the level of optimization                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                  |                                                                                       |                                   |                                     |                             |                       |                     |                                            |                        |                             |                      |                                     |                        |                          |
| <code>no_code_motion</code>       | Turns off code motion                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                  |                                                                                       |                                   |                                     |                             |                       |                     |                                            |                        |                             |                      |                                     |                        |                          |
| <code>no_cse</code>               | Turns off common subexpression elimination                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                                  |                                                                                       |                                   |                                     |                             |                       |                     |                                            |                        |                             |                      |                                     |                        |                          |
| <code>no_inline</code>            | Turns off function inlining                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                                  |                                                                                       |                                   |                                     |                             |                       |                     |                                            |                        |                             |                      |                                     |                        |                          |
| <code>no_tbaa</code>              | Turns off type-based alias analysis                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                  |                                                                                       |                                   |                                     |                             |                       |                     |                                            |                        |                             |                      |                                     |                        |                          |
| <code>no_unroll</code>            | Turns off loop unrolling                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                  |                                                                                       |                                   |                                     |                             |                       |                     |                                            |                        |                             |                      |                                     |                        |                          |
| Description                       | <p>Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.</p> <p>The parameters <code>speed</code>, <code>size</code>, and <code>balanced</code> only have effect on the <code>high</code> optimization level and only one of them can be used as it is not possible to optimize for speed and size at the same time. It is also not possible to use preprocessor macros embedded in this pragma directive. Any such macro will not be expanded by the preprocessor.</p>                                                                             |                                  |                                                                                       |                                   |                                     |                             |                       |                     |                                            |                        |                             |                      |                                     |                        |                          |

**Note:** If you use the `#pragma optimize` directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.

**Example**

```
#pragma optimize=speed
int small_and_used_often()
{
    ...
}

#pragma optimize=size no_inline
int big_and_seldom_used()
{
    ...
}
```

**pack****Syntax**

```
#pragma pack(n)
#pragma pack()
#pragma pack({push|pop} [, name] [, n])
```

**Parameters**

|             |                                                                      |
|-------------|----------------------------------------------------------------------|
| <i>n</i>    | Sets an optional structure alignment; one of: 1, 2, 4, 8, or 16      |
| Empty list  | Restores the structure alignment to default                          |
| push        | Sets a temporary structure alignment                                 |
| pop         | Restores the structure alignment from a temporarily pushed alignment |
| <i>name</i> | An optional pushed or popped alignment label                         |

**Description**

Use this pragma directive to specify the maximum alignment of `struct` and `union` members.

The `#pragma pack` directive affects declarations of structures following the pragma directive to the next `#pragma pack` or end of file.

**Note:** This can result in significantly larger and slower code when accessing members of the structure.

See also *Structure types*, page 187.

## **\_\_printf\_args**

|             |                                                                                                                                                                                                                           |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma __printf_args</code>                                                                                                                                                                                        |
| Description | Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example %d) is syntactically correct. |
| Example     | <pre>#pragma __printf_args int printf(char const *,...);  /* Function call */ printf("%d",x); /* Compiler checks that x is a double */</pre>                                                                              |

## **required**

|             |                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma required=symbol</code>                                                                                                                                                                                                                                                                                                                                                              |
| Parameters  | <i>symbol</i> Any statically linked function or variable.                                                                                                                                                                                                                                                                                                                                         |
| Description | <p>Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol.</p> <p>Use the directive if the requirement for a symbol is not otherwise visible in the application, for example if a variable is only referenced indirectly through the segment it resides in.</p> |
| Example     | <pre>const char copyright[] = "Copyright by me";  #pragma required=copyright int main() {     /* Do something here. */ }</pre> <p>Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.</p>                                                                                                                           |

## rtmodel

|                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                    |                                                           |                      |                                                                                                                                                                   |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------|-----------------------------------------------------------|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax               | <code>#pragma rtmodel="key", "value"</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                    |                                                           |                      |                                                                                                                                                                   |
| Parameters           | <table> <tr> <td><code>"key"</code></td> <td>A text string that specifies the runtime model attribute.</td> </tr> <tr> <td><code>"value"</code></td> <td>A text string that specifies the value of the runtime model attribute. Using the special value <code>*</code> is equivalent to not defining the attribute at all.</td> </tr> </table>                                                                                                                                                                                                                                                                                                                                                                                                               | <code>"key"</code> | A text string that specifies the runtime model attribute. | <code>"value"</code> | A text string that specifies the value of the runtime model attribute. Using the special value <code>*</code> is equivalent to not defining the attribute at all. |
| <code>"key"</code>   | A text string that specifies the runtime model attribute.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                    |                                                           |                      |                                                                                                                                                                   |
| <code>"value"</code> | A text string that specifies the value of the runtime model attribute. Using the special value <code>*</code> is equivalent to not defining the attribute at all.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                    |                                                           |                      |                                                                                                                                                                   |
| Description          | <p>Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.</p> <p>This pragma directive is useful for enforcing 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, or the special value <code>*</code>. It can, however, be useful to state explicitly that the module can handle any runtime model.</p> <p>A module can have several runtime model definitions.</p> <p><b>Note:</b> The predefined compiler runtime model attributes start with a double underscore. To avoid confusion, this style must not be used in the user-defined attributes.</p> |                    |                                                           |                      |                                                                                                                                                                   |
| Example              | <pre>#pragma rtmodel="I2C", "ENABLED"</pre> <p>The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                    |                                                           |                      |                                                                                                                                                                   |
| See also             | <i>Checking module consistency</i> , page 78.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                    |                                                           |                      |                                                                                                                                                                   |

## \_\_scanf\_args

|             |                                                                                                                                                                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma __scanf_args</code>                                                                                                                                                                                                      |
| Description | Use this pragma directive on a function with a scanf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example <code>%d</code> ) is syntactically correct. |
| Example     | <pre>#pragma __scanf_args int printf(char const *,...);  /* Function call */ scanf("%d",x); /* Compiler checks that x is a double */</pre>                                                                                             |

## segment

|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |             |                         |                          |                                                                                                                              |              |                                                                                                                   |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|-------------------------|--------------------------|------------------------------------------------------------------------------------------------------------------------------|--------------|-------------------------------------------------------------------------------------------------------------------|
| Syntax                   | <code>#pragma segment="NAME" [__memoryattribute] [align]</code>                                                                                                                                                                                                                                                                                                                                                                                                       |             |                         |                          |                                                                                                                              |              |                                                                                                                   |
| Parameters               | <table> <tr> <td><i>NAME</i></td> <td>The name of the segment</td> </tr> <tr> <td><i>__memoryattribute</i></td> <td>An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used.</td> </tr> <tr> <td><i>align</i></td> <td>Specifies an alignment for the segment part. The value must be a constant integer expression to the power of two.</td> </tr> </table>                                      | <i>NAME</i> | The name of the segment | <i>__memoryattribute</i> | An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used. | <i>align</i> | Specifies an alignment for the segment part. The value must be a constant integer expression to the power of two. |
| <i>NAME</i>              | The name of the segment                                                                                                                                                                                                                                                                                                                                                                                                                                               |             |                         |                          |                                                                                                                              |              |                                                                                                                   |
| <i>__memoryattribute</i> | An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used.                                                                                                                                                                                                                                                                                                                                          |             |                         |                          |                                                                                                                              |              |                                                                                                                   |
| <i>align</i>             | Specifies an alignment for the segment part. The value must be a constant integer expression to the power of two.                                                                                                                                                                                                                                                                                                                                                     |             |                         |                          |                                                                                                                              |              |                                                                                                                   |
| Description              | <p>Use this pragma directive to define a segment name that can be used by the segment operators <code>__segment_begin</code> and <code>__segment_end</code>. All segment declarations for a specific segment must have the same memory type attribute and alignment.</p> <p>If an optional memory attribute is used, the return type of the segment operators <code>__segment_begin</code> and <code>__segment_end</code> is:</p> <pre>void __memoryattribute *</pre> |             |                         |                          |                                                                                                                              |              |                                                                                                                   |
| Example                  | <code>#pragma segment="MYSADDR" __saddr 4</code>                                                                                                                                                                                                                                                                                                                                                                                                                      |             |                         |                          |                                                                                                                              |              |                                                                                                                   |
| See also                 | <i>Important language extensions</i> , page 194. For more information about segments and segment parts, see the chapter <i>Placing code and data</i> .                                                                                                                                                                                                                                                                                                                |             |                         |                          |                                                                                                                              |              |                                                                                                                   |

## type\_attribute

|             |                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma type_attribute=type_attribute[, type_attribute, ...]</code>                                                                                                                                                                                                                                                                                                                        |
| Parameters  | For a list of type attributes that can be used with this pragma directive, see <i>Type attributes</i> , page 204.                                                                                                                                                                                                                                                                                |
| Description | <p>Use this pragma directive to specify IAR-specific <i>type attributes</i>, which are not part of the ISO/ANSI C language standard. Note however, that a given type attribute might not be applicable to all kind of objects.</p> <p>This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.</p> |
| Example     | <p>In this example, an <code>int</code> object with the memory attribute <code>__saddr</code> is defined:</p> <pre>#pragma type_attribute=__saddr int x;</pre>                                                                                                                                                                                                                                   |



**Description**

Use this pragma directive to specify the vector(s) of an interrupt function whose declaration follows the pragma directive. Note that several vectors can be defined for each function.

**Example!**

```
#pragma vector=0x14  
__interrupt void my_handler(void);
```

# Intrinsic functions

This chapter gives reference information about the intrinsic functions, a predefined set of functions available in the compilers.

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions.

---

## Summary of intrinsic functions

To use intrinsic functions in an application, include the header file `intrinsics.h`.

Note that the intrinsic function names start with double underscores, for example:

```
__disable_interrupt
```

This table summarizes the intrinsic functions:

| <b>Intrinsic function</b>          | <b>Description</b>                           |
|------------------------------------|----------------------------------------------|
| <code>__break</code>               | Inserts a BRK instruction                    |
| <code>__disable_interrupt</code>   | Disables interrupts                          |
| <code>__enable_interrupt</code>    | Enables interrupts                           |
| <code>__get_interrupt_state</code> | Returns the interrupt state                  |
| <code>__halt</code>                | Inserts a halt/no-operation instruction pair |
| <code>__no_operation</code>        | Inserts a NOP instruction                    |
| <code>__set_interrupt_state</code> | Restores the interrupt state                 |
| <code>__stop</code>                | Inserts a stop/no-operation instruction pair |

*Table 42: Intrinsic functions summary*

---

## Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.

### **\_\_break**

Syntax

```
void __break(void);
```

Description

Inserts a BRK instruction. This function cannot be used with the 78K0S processor core.

### **\_\_disable\_interrupt**

Syntax

```
void __disable_interrupt(void);
```

Description

Disables interrupts by inserting the DI instruction.

### **\_\_enable\_interrupt**

Syntax

```
void __enable_interrupt(void);
```

Description

Enables interrupts by inserting the EI instruction.

### **\_\_get\_interrupt\_state**

Syntax

```
__istate_t __get_interrupt_state(void);
```

Description

Returns the global interrupt state. The return value can be used as an argument to the `__set_interrupt_state` intrinsic function, which will restore the interrupt state.

Example

```
__istate_t s = __get_interrupt_state();
__disable_interrupt();
```

```
/* Do something here. */
```

```
__set_interrupt_state(s);
```

The advantage of using this sequence of code compared to using `__disable_interrupt` and `__enable_interrupt` is that the code in this example will not enable any interrupts disabled before the call of `__get_interrupt_state`.

## **\_\_halt**

|             |                                               |
|-------------|-----------------------------------------------|
| Syntax      | <code>void __halt(void);</code>               |
| Description | Inserts a halt/no-operation instruction pair. |

## **\_\_no\_operation**

|             |                                         |
|-------------|-----------------------------------------|
| Syntax      | <code>void __no_operation(void);</code> |
| Description | Inserts a NOP instruction.              |

## **\_\_set\_interrupt\_state**

|              |                                                                                                                                                                                                                               |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>void __set_interrupt_state(__istate_t);</code>                                                                                                                                                                          |
| Descriptions | Restores the interrupt state to a value previously returned by the <code>__get_interrupt_state</code> function.<br>For information about the <code>__istate_t</code> type, see <code>__get_interrupt_state</code> , page 234. |

## **\_\_stop**

|             |                                               |
|-------------|-----------------------------------------------|
| Syntax      | <code>void __stop(void);</code>               |
| Description | Inserts a stop/no-operation instruction pair. |



# The preprocessor

This chapter gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.

---

## Overview of the preprocessor

The preprocessor of the IAR C/C++ Compilers for 78K adheres to the ISO/ANSI standard. The compiler also makes these preprocessor-related features available to you:

- **Predefined preprocessor symbols**

These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. For details, see *Descriptions of predefined preprocessor symbols*, page 238.
- **User-defined preprocessor symbols defined using a compiler option**

In addition to defining your own preprocessor symbols using the `#define` directive, you can also use the option `-D`, see *-D*, page 157.
- **Preprocessor extensions**

There are several preprocessor extensions, for example many `pragma` directives; for more information, see the chapter *Pragma directives* in this guide. Read also about the corresponding `_Pragma` operator and the other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 240.
- **Preprocessor output**

Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 176.

Some parts listed by the ISO/ANSI standard are implementation-defined, for example the character set used in the preprocessor directives and inclusion of bracketed and quoted filenames. To read more about this, see *Preprocessing directives*, page 273.

## Descriptions of predefined preprocessor symbols

This table describes the predefined preprocessor symbols:

| Predefined symbol             | Identifies                                                                                                                                                                                                                                                                                                                                                             |
|-------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__BASE_FILE__</code>    | A string that identifies the name of the base source file (that is, not the header file), being compiled. See also <code>__FILE__</code> , page 239, and <code>--no_path_in_file_macros</code> , page 171.                                                                                                                                                             |
| <code>__BUILD_NUMBER__</code> | A unique integer that identifies the build number of the compiler currently in use. The build number does not necessarily increase with a compiler that is released later.                                                                                                                                                                                             |
| <code>__CODE_MODEL__</code>   | An integer that identifies the code model in use. The symbol reflects the <code>--code_model</code> option and is defined to <code>__CODE_MODEL_STANDARD__</code> , <code>__CODE_MODEL_BANKED__</code> , <code>__CODE_MODEL_NEAR__</code> , or <code>__CODE_MODEL_FAR__</code> . These symbolic names can be used when testing the <code>__CODE_MODEL__</code> symbol. |
| <code>__CORE__</code>         | An integer that identifies the chip core in use. The symbol reflects the <code>--core</code> option and is defined to <code>__78K0_BASIC__</code> , <code>__78K0__</code> , <code>__78K0S__</code> , or <code>__78K0R__</code> . These symbolic names can be used when testing the <code>__CORE__</code> symbol.                                                       |
| <code>__cplusplus</code>      | An integer which is defined when the compiler runs in any of the C++ modes, otherwise it is undefined. When defined, its value is 199711L. This symbol can be used with <code>#ifdef</code> to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*                            |
| <code>__DATA_MODEL__</code>   | An integer that identifies the data model in use. The symbol reflects the <code>--data_model</code> option and is defined to <code>__DATA_MODEL_NEAR__</code> or <code>__DATA_MODEL_FAR__</code> . These symbolic names can be used when testing the <code>__DATA_MODEL__</code> symbol.                                                                               |
| <code>__DATE__</code>         | A string that identifies the date of compilation, which is returned in the form "Mmm dd yyyy", for example "Oct 30 2008".*                                                                                                                                                                                                                                             |

Table 43: Predefined symbols

| Predefined symbol                 | Identifies                                                                                                                                                                                                                                                                                                           |
|-----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__embedded_cplusplus</code> | An integer which is defined to 1 when the compiler runs in any of the C++ modes, otherwise the symbol is undefined. This symbol can be used with <code>#ifdef</code> to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.* |
| <code>__FILE__</code>             | A string that identifies the name of the file being compiled, which can be both the base source file and any included header file. See also <code>__BASE_FILE__</code> , page 238, and <code>-no_path_in_file_macros</code> , page 171.*                                                                             |
| <code>__func__</code>             | A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see <code>-e</code> , page 164. See also <code>__PRETTY_FUNCTION__</code> , page 240.                              |
| <code>__FUNCTION__</code>         | A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see <code>-e</code> , page 164. See also <code>__PRETTY_FUNCTION__</code> , page 240.                              |
| <code>__IAR_SYSTEMS_ICC__</code>  | An integer that identifies the IAR compiler platform. The current value is 7. Note that the number could be higher in a future version of the product. This symbol can be tested with <code>#ifdef</code> to detect whether the code was compiled by a compiler from IAR Systems.                                    |
| <code>__ICC78K__</code>           | An integer that is set to 1 when the code is compiled with one of the IAR C/C++ Compilers for 78K, and otherwise to 0.                                                                                                                                                                                               |
| <code>__LINE__</code>             | An integer that identifies the current source line number of the file being compiled, which can be both the base source file and any included header file.*                                                                                                                                                          |
| <code>__LITTLE_ENDIAN__</code>    | An integer that identifies the byte order of the microcontroller. For the 78K microcontroller families, the value of this symbol is defined to 1, which means that the byte order is little-endian.                                                                                                                  |

Table 43: Predefined symbols (Continued)

| Predefined symbol                | Identifies                                                                                                                                                                                                                                                                                                                                                                             |
|----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__PRETTY_FUNCTION__</code> | A string that identifies the function name, including parameter types and return type, of the function in which the symbol is used, for example <code>"void func(char) "</code> . This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see <code>-e</code> , page 164. See also <code>__func__</code> , page 239. |
| <code>__STDC__</code>            | An integer that is set to 1, which means the compiler adheres to the ISO/ANSI C standard. This symbol can be tested with <code>#ifdef</code> to detect whether the compiler in use adheres to ISO/ANSI C.*                                                                                                                                                                             |
| <code>__STDC_VERSION__</code>    | An integer that identifies the version of ISO/ANSI C standard in use. The symbol expands to 199409L. This symbol does not apply in EC++ mode.*                                                                                                                                                                                                                                         |
| <code>__SUBVERSION__</code>      | An integer that identifies the subversion number of the compiler version number, for example 3 in 1.2.3.4.                                                                                                                                                                                                                                                                             |
| <code>__TIME__</code>            | A string that identifies the time of compilation in the form <code>"hh:mm:ss"</code> .*                                                                                                                                                                                                                                                                                                |
| <code>__VER__</code>             | An integer that identifies the version number of the IAR compiler in use. The value of the number is calculated in this way: (100 * the major version number + the minor version number). For example, for compiler version 3.34, 3 is the major version number and 34 is the minor version number. Hence, the value of <code>__VER__</code> is 334.                                   |

Table 43: Predefined symbols (Continued)

\* This symbol is required by the ISO/ANSI standard.

## Descriptions of miscellaneous preprocessor extensions

This section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives, and ISO/ANSI directives.

### NDEBUG

Description

This preprocessor symbol determines whether any assert macros you have written in your application shall be included or not in the built application.

If this symbol is not defined, all assert macros are evaluated. If the symbol is defined, all assert macros are excluded from the compilation. In other words, if the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you write any assert code and build your application, you should define this symbol to exclude the assert code from the final application.

Note that the assert macro is defined in the `assert.h` standard include file.



In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

## **\_Pragma()**

Syntax

```
_Pragma("string")
```

where *string* follows the syntax of the corresponding pragma directive.

Description

This preprocessor operator is part of the C99 standard and can be used, for example, in defines and is equivalent to the `#pragma` directive.

**Note:** The `-e` option—enable language extensions—does not have to be specified.

Example

```
#if NO_OPTIMIZE
    #define NOOPT _Pragma("optimize=none")
#else
    #define NOOPT
#endif
```

See also

See the chapter *Pragma directives*.

## **#warning message**

Syntax

```
#warning message
```

where *message* can be any string.

Description

Use this preprocessor directive to produce messages. Typically, this is useful for assertions and other trace utilities, similar to the way the ISO/ANSI standard `#error` directive is used.

## **\_\_VA\_ARGS\_\_**

**Syntax**

```
#define P(...)      __VA_ARGS__
#define P(x, y, ...)  x + y + __VA_ARGS__
```

`__VA_ARGS__` will contain all variadic arguments concatenated, including the separating commas.

**Description**

Variadic macros are the preprocessor macro equivalents of `printf` style functions. `__VA_ARGS__` is part of the C99 standard.

**Example**

```
#if DEBUG
    #define DEBUG_TRACE(S, ...) printf(S, __VA_ARGS__)
#else
    #define DEBUG_TRACE(S, ...)
#endif
/* Place your own code here */
DEBUG_TRACE("The value is:%d\n",value);
```

will result in:

```
printf("The value is:%d\n",value);
```

# Library functions

This chapter gives an introduction to the C and C++ library functions. It also lists the header files used for accessing library definitions.

For detailed reference information about the library functions, see the online help system.

---

## Introduction

The compilers provide two different libraries:

- IAR DLIB Library is a complete ISO/ANSI C and C++ library. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, et cetera.
- IAR CLIB Library is a light-weight library, which is not fully compliant with ISO/ANSI C. Neither does it fully support floating-point numbers in IEEE 754 format or Embedded C++.

Note that different customization methods are normally needed for these two libraries. For additional information, see the chapter *The DLIB runtime environment* and *The CLIB runtime environment*, respectively.

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For additional information about library functions, see the chapter *Implementation-defined behavior* in this guide.

## HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into several different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

## LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic settings for project configuration*, page 6. The linker will include only those routines that are required—directly or indirectly—by your application.

## REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant.

Most parts of the DLIB library are reentrant, but these functions and parts are not reentrant because they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, and the C++ operators `new` and `delete`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- Multibyte functions—`mbrlen`, `mbrtowc`, `mbsrtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`
- The miscellaneous functions `setlocale`, `rand`, `atexit`, `strerror`, `strtok`
- Functions that use files in some way. This includes `printf`, `scanf`, `getchar`, and `putchar`. The functions `sprintf` and `sscanf` are not included.

For the CLIB library, the `qsort` function and functions that use files in some way are non-reentrant. This includes `printf`, `scanf`, `getchar`, and `putchar`. The functions `sprintf` and `sscanf` are not included.

Some functions also share the same storage for `errno`. These functions are not reentrant, since an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it is read. Among these functions are:

```
exp, exp10, ldexp, log, log10, pow, sqrt, acos, asin, atan2,
cosh, sinh, strtod, strtol, strtoul
```

Remedies for this are:

- Do not use non-reentrant functions in interrupt service routines
- Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

## IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of the ISO/ANSI standard for the programming language C. For additional information, see the chapter *Implementation-defined behavior* in this guide.
- Standard C library definitions, for user programs.
- Embedded C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code. It is described in the chapter *The DLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of 78K features. See the chapter *Intrinsic functions* for more information.

In addition, the IAR DLIB Library includes some added C functionality, partly taken from the C99 standard, see *Added C functionality*, page 248.

### C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter *Compiler extensions*.

The following table lists the C header files:

| Header file             | Usage                                                              |
|-------------------------|--------------------------------------------------------------------|
| <code>assert.h</code>   | Enforcing assertions when functions execute                        |
| <code>ctype.h</code>    | Classifying characters                                             |
| <code>errno.h</code>    | Testing error codes reported by library functions                  |
| <code>float.h</code>    | Testing floating-point type properties                             |
| <code>inttypes.h</code> | Defining formatters for all types defined in <code>stdint.h</code> |
| <code>iso646.h</code>   | Using Amendment 1— <code>iso646.h</code> standard header           |
| <code>limits.h</code>   | Testing integer type properties                                    |
| <code>locale.h</code>   | Adapting to different cultural conventions                         |
| <code>math.h</code>     | Computing common mathematical functions                            |
| <code>setjmp.h</code>   | Executing non-local goto statements                                |
| <code>signal.h</code>   | Controlling various exceptional conditions                         |
| <code>stdarg.h</code>   | Accessing a varying number of arguments                            |

Table 44: Traditional standard C header files—DLIB

| Header file            | Usage                                                  |
|------------------------|--------------------------------------------------------|
| <code>stdbool.h</code> | Adds support for the <code>bool</code> data type in C. |
| <code>stddef.h</code>  | Defining several useful types and macros               |
| <code>stdint.h</code>  | Providing integer characteristics                      |
| <code>stdio.h</code>   | Performing input and output                            |
| <code>stdlib.h</code>  | Performing a variety of operations                     |
| <code>string.h</code>  | Manipulating several kinds of strings                  |
| <code>time.h</code>    | Converting between various time and date formats       |
| <code>wchar.h</code>   | Support for wide characters                            |
| <code>wctype.h</code>  | Classifying wide characters                            |

Table 44: Traditional standard C header files—DLIB (Continued)

## C++ HEADER FILES

This section lists the C++ header files.

### Embedded C++

The following table lists the Embedded C++ header files:

| Header file               | Usage                                                                             |
|---------------------------|-----------------------------------------------------------------------------------|
| <code>complex</code>      | Defining a class that supports complex arithmetic                                 |
| <code>exception</code>    | Defining several functions that control exception handling                        |
| <code>fstream</code>      | Defining several I/O stream classes that manipulate external files                |
| <code>iomanip</code>      | Declaring several I/O stream manipulators that take an argument                   |
| <code>ios</code>          | Defining the class that serves as the base for many I/O streams classes           |
| <code>iosfwd</code>       | Declaring several I/O stream classes before they are necessarily defined          |
| <code>iostream</code>     | Declaring the I/O stream objects that manipulate the standard streams             |
| <code>istream</code>      | Defining the class that performs extractions                                      |
| <code>new</code>          | Declaring several functions that allocate and free storage                        |
| <code>ostream</code>      | Defining the class that performs insertions                                       |
| <code>sstream</code>      | Defining several I/O stream classes that manipulate string containers             |
| <code>stdexcept</code>    | Defining several classes useful for reporting exceptions                          |
| <code>streambuf</code>    | Defining classes that buffer I/O stream operations                                |
| <code>string</code>       | Defining a class that implements a string container                               |
| <code>stringstream</code> | Defining several I/O stream classes that manipulate in-memory character sequences |

Table 45: Embedded C++ header files

The following table lists additional C++ header files:

| Header file             | Usage                                                                 |
|-------------------------|-----------------------------------------------------------------------|
| <code>fstream.h</code>  | Defining several I/O stream classes that manipulate external files    |
| <code>iomanip.h</code>  | Declaring several I/O stream manipulators that take an argument       |
| <code>iostream.h</code> | Declaring the I/O stream objects that manipulate the standard streams |
| <code>new.h</code>      | Declaring several functions that allocate and free storage            |

*Table 46: Additional Embedded C++ header files—DLIB*

### Extended Embedded C++ standard template library

The following table lists the Extended EC++ standard template library (STL) header files:

| Header file             | Description                                            |
|-------------------------|--------------------------------------------------------|
| <code>algorithm</code>  | Defines several common operations on sequences         |
| <code>deque</code>      | A deque sequence container                             |
| <code>functional</code> | Defines several function objects                       |
| <code>hash_map</code>   | A map associative container, based on a hash algorithm |
| <code>hash_set</code>   | A set associative container, based on a hash algorithm |
| <code>iterator</code>   | Defines common iterators, and operations on iterators  |
| <code>list</code>       | A doubly-linked list sequence container                |
| <code>map</code>        | A map associative container                            |
| <code>memory</code>     | Defines facilities for managing memory                 |
| <code>numeric</code>    | Performs generalized numeric operations on sequences   |
| <code>queue</code>      | A queue sequence container                             |
| <code>set</code>        | A set associative container                            |
| <code>slist</code>      | A singly-linked list sequence container                |
| <code>stack</code>      | A stack sequence container                             |
| <code>utility</code>    | Defines several utility components                     |
| <code>vector</code>     | A vector sequence container                            |

*Table 47: Standard template library header files*

### Using standard C libraries in C++

The C++ library works in conjunction with 15 of the header files from the standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, `cassert` and `assert.h`.

The following table shows the new header files:

| Header file            | Usage                                                              |
|------------------------|--------------------------------------------------------------------|
| <code>cassert</code>   | Enforcing assertions when functions execute                        |
| <code>cctype</code>    | Classifying characters                                             |
| <code>cerrno</code>    | Testing error codes reported by library functions                  |
| <code>cfloat</code>    | Testing floating-point type properties                             |
| <code>cinttypes</code> | Defining formatters for all types defined in <code>stdint.h</code> |
| <code>climits</code>   | Testing integer type properties                                    |
| <code>locale</code>    | Adapting to different cultural conventions                         |
| <code>cmath</code>     | Computing common mathematical functions                            |
| <code>csetjmp</code>   | Executing non-local goto statements                                |
| <code>csignal</code>   | Controlling various exceptional conditions                         |
| <code>stdarg</code>    | Accessing a varying number of arguments                            |
| <code>stdbool</code>   | Adds support for the <code>bool</code> data type in C.             |
| <code>stddef</code>    | Defining several useful types and macros                           |
| <code>stdint</code>    | Providing integer characteristics                                  |
| <code>stdio</code>     | Performing input and output                                        |
| <code>stdlib</code>    | Performing a variety of operations                                 |
| <code>string</code>    | Manipulating several kinds of strings                              |
| <code>time</code>      | Converting between various time and date formats                   |
| <code>wchar</code>     | Support for wide characters                                        |
| <code>wctype</code>    | Classifying wide characters                                        |

Table 48: New standard C header files—DLIB

## LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

Certain C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call, for example `memcpy`, `memset`, and `strcat`.

## ADDED C FUNCTIONALITY

The IAR DLIB Library includes some added C functionality, partly taken from the C99 standard.

The following include files provide these features:

- `ctype.h`
- `inttypes.h`

- `math.h`
- `stdbool.h`
- `stdint.h`
- `stdio.h`
- `stdlib.h`
- `wchar.h`
- `wctype.h`

### **cctype.h**

In `cctype.h`, the C99 function `isblank` is defined.

### **inttypes.h**

This include file defines the formatters for all types defined in `stdint.h` to be used by the functions `printf`, `scanf`, and all their variants.

### **math.h**

In `math.h` all functions exist in a `float` variant and a `long double` variant, suffixed by `f` and `l` respectively. For example, `sinf` and `sinl`.

The following C99 macro symbols are defined:

`HUGE_VALF`, `HUGE_VALL`, `INFINITY`, `NAN`, `FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL`, `FP_ZERO`, `MATH_ERRNO`, `MATH_ERREXCEPT`, `math_errhandling`.

The following C99 macro functions are defined:

`fpclassify`, `signbit`, `isfinite`, `isinf`, `isnan`, `isnormal`, `isgreater`, `isless`, `islessequal`, `islessgreater`, `isunordered`.

The following C99 type definitions are added:

`float_t`, `double_t`.

### **stdbool.h**

This include file makes the `bool` type available if the **Allow IAR extensions** (`-e`) option is used.

### **stdint.h**

This include file provides integer characteristics.

**stdio.h**

In `stdio.h`, the following C99 functions are defined:

`vscanf`, `vfscanf`, `vsscanf`, `vsnprintf`, `snprintf`

The functions `printf`, `scanf`, and all their variants have added functionality from the C99 standard. For reference information about these functions, see the library reference available from the **Help** menu.

The following functions providing I/O functionality for libraries built without `FILE` support are defined:

`__write_array` Corresponds to `fwrite` on `stdout`.

`__ungetchar` Corresponds to `ungetc` on `stdout`.

`__gets` Corresponds to `fgets` on `stdin`.

**stdlib.h**

In `stdlib.h`, the following C99 functions are defined:

`_Exit`, `llabs`, `lldiv`, `strtoll`, `strtoull`, `atoll`, `strtof`, `strtold`.

The function `strtod` has added functionality from the C99 standard. For reference information about this functions, see the library reference available from the **Help** menu.

The `__qsortbbl` function is defined; it provides sorting using a bubble sort algorithm. This is useful for applications that have a limited stack.

**wchar.h**

In `wchar.h`, the following C99 functions are defined:

`vfwscanf`, `vswscanf`, `vwscanf`, `wcstof`, `wcstolb`.

**wctype.h**

In `wctype.h`, the C99 function `iswblank` is defined.

---

## IAR CLIB Library

The IAR CLIB Library provides most of the important C library definitions that apply to embedded systems. These are of the following types:

- Standard C library definitions available for user programs. These are documented in this chapter.

- The system startup code. It is described in the chapter *The CLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of 78K features. See the chapter *Intrinsic functions* for more information.

## LIBRARY DEFINITIONS SUMMARY

This following table lists the header files specific to the CLIB library:

| Header file            | Description                                                                                                               |
|------------------------|---------------------------------------------------------------------------------------------------------------------------|
| <code>assert.h</code>  | Assertions                                                                                                                |
| <code>ctype.h*</code>  | Character handling                                                                                                        |
| <code>errno.h</code>   | Error return values                                                                                                       |
| <code>fenv.h</code>    | Floating-point exception flags                                                                                            |
| <code>float.h</code>   | Limits and sizes of floating-point types                                                                                  |
| <code>iccbutl.h</code> | Low-level routines                                                                                                        |
| <code>limits.h</code>  | Limits and sizes of integral types                                                                                        |
| <code>math.h</code>    | Mathematics                                                                                                               |
| <code>setjmp.h</code>  | Non-local jumps                                                                                                           |
| <code>stdarg.h</code>  | Variable arguments                                                                                                        |
| <code>stdbool.h</code> | Adds support for the <code>bool</code> data type in C                                                                     |
| <code>stddef.h</code>  | Common definitions including <code>size_t</code> , <code>NULL</code> , <code>ptrdiff_t</code> , and <code>offsetof</code> |
| <code>stdio.h</code>   | Input/output                                                                                                              |
| <code>stdlib.h</code>  | General utilities                                                                                                         |
| <code>string.h</code>  | String handling                                                                                                           |

Table 49: IAR CLIB Library header files

\* The functions `isxxx`, `toupper`, and `tolower` declared in the header file `ctype.h` evaluate their argument more than once. This is not according to the ISO/ANSI standard.



# Segment reference

The compilers place code and data into named segments which are referred to by the IAR XLINK Linker. Details about the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output from the compilers.

For more information about segments, see the chapter *Placing code and data*.

---

## Summary of segments

The table below lists the segments that are available in the compilers:

| Segment   | Description                                                                                                                                |
|-----------|--------------------------------------------------------------------------------------------------------------------------------------------|
| BCODE     | Holds all banked program code for the 78K0 and 78K0S cores.                                                                                |
| CHECKSUM  | Holds the checksum generated by the linker.                                                                                                |
| CLTVEC    | Holds the call table vectors generated by use of the <code>__callt</code> extended keyword.                                                |
| CODE      | Holds the program code.                                                                                                                    |
| CONST     | Holds <code>const</code> declared variables. 78K0/78K0S only.                                                                              |
| CSTACK    | Holds the stack used by C or C++ programs.                                                                                                 |
| DIFUNCT   | Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before <code>main</code> is called. |
| FAR_A     | Holds <code>__far</code> located data. 78K0R only.                                                                                         |
| FAR_CONST | Holds <code>const</code> declared <code>__far</code> variables. 78K0R only.                                                                |
| FAR_HEAP  | Holds the heap used for dynamically allocated far data. Requires the DLIB environment. 78K0R only.                                         |
| FAR_I     | Holds <code>__far</code> static and global initialized variables. 78K0R only.                                                              |
| FAR_ID    | Holds initial values for <code>__far</code> static and global variables in <code>FAR_I</code> . 78K0R only.                                |
| FAR_N     | Holds <code>__no_init __far</code> static and global variables. 78K0R only.                                                                |
| FAR_Z     | Holds zero-initialized <code>__far</code> static and global variables. 78K0R only.                                                         |
| FCODE     | Holds program code declared with the extended keyword <code>__callf</code> . 78K0 only.                                                    |

---

Table 50: Segment summary

| Segment    | Description                                                                                                        |
|------------|--------------------------------------------------------------------------------------------------------------------|
| HEAP       | Holds the heap used for dynamically allocated data. 78K0/78K0S and 78K0R when using the CLIB environment.          |
| INTVEC     | Contains the reset and interrupt vectors.                                                                          |
| NEAR_A     | Holds <code>__near</code> located data.                                                                            |
| NEAR_CONST | Holds <code>const</code> declared <code>__near</code> variables. 78K0R only.                                       |
| NEAR_HEAP  | Holds the heap used for dynamically allocated <code>__near</code> data. Requires the DLIB environment. 78K0R only. |
| NEAR_I     | Holds <code>__near</code> static and global initialized variables.                                                 |
| NEAR_ID    | Holds initial values for <code>__near</code> static and global variables in <code>NEAR_I</code> .                  |
| NEAR_N     | Holds <code>__no_init __near</code> static and global variables.                                                   |
| NEAR_Z     | Holds zero-initialized <code>__near</code> static and global variables.                                            |
| RCODE      | Holds the C/C++ startup and run-time library code.                                                                 |
| SADDR_A    | Holds <code>__saddr</code> located data.                                                                           |
| SADDR_I    | Holds <code>__saddr</code> static and global initialized variables.                                                |
| SADDR_ID   | Holds initial values for <code>__saddr</code> static and global variables in <code>SADDR_I</code> .                |
| SADDR_N    | Holds <code>__no_init __saddr</code> static and global variables.                                                  |
| SADDR_Z    | Holds zero-initialized <code>__saddr</code> static and global variables.                                           |
| SWITCH     | Holds switch tables.                                                                                               |
| WRKSEG     | Holds the short address working area; an extra register variable area.                                             |
| XCODE      | Holds program code declared with the attribute <code>__far_func</code> . 78K0R only.                               |

Table 50: Segment summary (Continued)

## Descriptions of segments

This section gives reference information about each segment.

The segments are placed in memory by the segment placement linker directives `-z` and `-p`, for sequential and packed placement, respectively. Some segments cannot use packed placement, as their contents must be continuous.

In each description, the segment memory type—`CODE`, `CONST`, or `DATA`—indicates whether the segment should be placed in ROM or RAM memory; see Table 11, *XLINK segment memory types*, page 40.

For information about the `-z` and the `-p` directives, see the *IAR Linker and Library Tools Reference Guide*.

For information about how to define segments in the linker command file, see *Customizing the linker command file*, page 40.

For detailed information about the extended keywords mentioned here, see the chapter *Extended keywords*.

## BCODE

|                     |                                                                                                                                                                                                                                                                                                                                                                                      |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds all banked program code for the 78K0 and 78K0S cores. This means:</p> <ul style="list-style-type: none"> <li>● All code not declared <code>__non_banked</code> in the banked code model.</li> <li>● All code declared <code>__banked</code> in the standard code model.</li> </ul> <p>Place this segment in memory using the <code>-P</code> segment control directive.</p> |
| Segment memory type | CODE                                                                                                                                                                                                                                                                                                                                                                                 |
| Memory placement    | For information about segment placement, see <i>Banked code (78K0/78K0S only)</i> , page 50.                                                                                                                                                                                                                                                                                         |
| Access type         | Read-only                                                                                                                                                                                                                                                                                                                                                                            |

## CHECKSUM

|                     |                                                                                                                                                                                                           |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds the checksum bytes generated by the linker. This segment also holds the <code>__checksum</code> symbol. Note that the size of this segment is affected by the linker option <code>-J</code>.</p> |
| Segment memory type | CONST                                                                                                                                                                                                     |
| Memory placement    | This segment can be placed anywhere in ROM memory.                                                                                                                                                        |
| Access type         | Read-only                                                                                                                                                                                                 |

## CLTVEC

|                     |                                                                                                                                                                                                    |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds call table vectors generated by use of the <code>__callt</code> keyword.</p> <p>Place this segment in memory using the <code>-Z</code> or <code>-P</code> segment control directives.</p> |
| Segment memory type | CODE                                                                                                                                                                                               |
| Memory placement    | 0x00040–0x000FF (78K0 and 78K0S)                                                                                                                                                                   |

0x00080–0x000BF (78K0R)

Access type Read-only

## CODE

**Description** Default segment for program code.  
 This segment holds user program code declared with the `__non_banked` (78K0/78K0S), `__near_func` (78K0R), `__interrupt`, or `__callt` attribute. Place this segment in memory using the `-Z` or `-P` segment control directive.

**Segment memory type** CODE

**Memory placement** 0x00000–0x0FFFF

**Access type** Read-only

## CONST

**Description** Holds `const` declared variables for the 78K0 and 78K0S cores, except for those variables declared `const __saddr`.  
 Place this segment in memory using the `-P` segment control directives.

**Segment memory type** CONST

**Memory placement** 0x00000–0x0FFFF

**Access type** Read-only

## CSTACK

**Description** Holds the internal data stack.

**Segment memory type** DATA

**Memory placement** 0x00000–0x0FFFF (78K0 and 78K0S)  
 0xF0000–0xFFFFF (78K0R)

**Access type** Read/write

See also *The stack*, page 46.

## DIFUNCT

|                     |                                                      |
|---------------------|------------------------------------------------------|
| Description         | Holds the dynamic initialization vector used by C++. |
| Segment memory type | CONST                                                |
| Memory placement    | This segment can be placed anywhere in memory.       |
| Access type         | Read-only                                            |

## FAR\_A

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Holds <code>__far</code> located data that is declared <code>__no_init</code> or <code>const</code>. Also holds located constants that are not explicitly declared <code>__near</code> when the Near data model is being used.</p> <p><i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker command file.</p> <p>This segment only applies to the 78K0R core.</p> |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## FAR\_CONST

|                     |                                                                                                                                                                                                                                       |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>This segment holds <code>__far</code> constant data for the 78K0R core that is not placed at an absolute location.</p> <p>Place this segment in memory using the <code>-Z</code> or <code>-P</code> segment control directive.</p> |
| Segment memory type | FARCONST                                                                                                                                                                                                                              |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                                                                                        |
| Access type         | Read-only                                                                                                                                                                                                                             |

## FAR\_HEAP

|                     |                                                                                                                                                                                                                                                                             |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | When the DLIB runtime environment is being used, this segment holds the heap used for dynamically allocated data in far memory, in other words data allocated by <code>far_malloc</code> and <code>far_free</code> , and in C++, <code>new</code> and <code>delete</code> . |
| Segment memory type | DATA                                                                                                                                                                                                                                                                        |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                                                                                                                              |
| Access type         | Read/write                                                                                                                                                                                                                                                                  |
| See also            | <i>The heap</i> , page 47 and <i>New and Delete operators (78K0R only)</i> , page 114.                                                                                                                                                                                      |

## FAR\_I

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__far</code> static and global initialized variables initialized by copying from the segment <code>FAR_ID</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the <code>-z</code> directive must be used.</p> <p><b>Note:</b> This segment only applies to the 78K0R core.</p> |
| Segment memory type | FARDATA                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| Memory placement    | This segment can be placed anywhere in memory, but it must start at the same offset from a 64-Kbyte offset as <code>FAR_ID</code> .                                                                                                                                                                                                                                                                                                                                                             |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |

## FAR\_ID

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds initial values for <code>__far</code> static and global variables in the <code>FAR_I</code> segment. These values are copied from <code>FAR_ID</code> to <code>FAR_I</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the <code>-z</code> directive must be used.</p> <p><b>Note:</b> This segment only applies to the 78K0R core.</p> |
| Segment memory type | FARCONST                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |

|                  |                                                                                                                                    |
|------------------|------------------------------------------------------------------------------------------------------------------------------------|
| Memory placement | This segment can be placed anywhere in memory, but it must start at the same offset from a 64-Kbyte offset as <code>FAR_I</code> . |
| Access type      | Read-only                                                                                                                          |

## FAR\_N

|                     |                                                                                                                              |
|---------------------|------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __far</code> variables.<br><b>Note:</b> This segment only applies to the 78K0R core. |
| Segment memory type | DATA                                                                                                                         |
| Memory placement    | This segment can be placed anywhere in memory.                                                                               |
| Access type         | Read/write                                                                                                                   |

## FAR\_Z

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds zero-initialized <code>__far</code> static and global variables. The contents of this segment is declared by the system startup code.<br><br>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the <code>-z</code> directive must be used.<br><br><b>Note:</b> This segment only applies to the 78K0R core. |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |

## FCODE

|                     |                                                                                                                                                                                                |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds program code for the 78K0 core, declared with the attribute <code>__callf</code> . Place this segment in memory using the <code>-z</code> or <code>-P</code> segment control directives. |
| Segment memory type | CODE                                                                                                                                                                                           |
| Memory placement    | 0x00800–0x00FFF                                                                                                                                                                                |

Access type                   Read-only

## HEAP

Description                   Holds the heap used for dynamically allocated data for the 78K0 and 78K0S cores, and—if the CLIB runtime environment is used—the 78K0R core. In other words data allocated by `malloc` and `free`, and in C++, `new` and `delete`.

Segment memory type         DATA

Memory placement            0x00000–0x0FFFF

Access type                   Read/write

See also                      *The heap*, page 47.

## INTVEC

Description                   Holds the interrupt vector table generated by the use of the `__interrupt` extended keyword.

Segment memory type         CODE

Memory placement            0x00000–0x0003F (78K0 and 78K0S)  
0x00000–0x0007F (78K0R)

Access type                   Read-only

## NEAR\_A

Description                   Holds `__near` located data that is declared `__no_init`. Located constants are placed here only if they are explicitly declared `__near`.

*Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

## NEAR\_CONST

|                     |                                                                                                                                                                                                                                 |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | This segment holds <code>__near</code> constant data for the 78K0R core that is not placed at an absolute location.<br><br>Place this segment in memory using the <code>-Z</code> or <code>-P</code> segment control directive. |
| Segment memory type | CONST                                                                                                                                                                                                                           |
| Memory placement    | 0x00000–0x0FFFF                                                                                                                                                                                                                 |
| Access type         | Read-only                                                                                                                                                                                                                       |

## NEAR\_HEAP

|                     |                                                                                                                                                                                                                                                                                    |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | When the the DLIB runtime environment is being used, this segment holds the heap used for dynamically allocated data in near memory, in other words data allocated by <code>near_malloc</code> and <code>near_free</code> , and in C++, <code>new</code> and <code>delete</code> . |
| Segment memory type | DATA                                                                                                                                                                                                                                                                               |
| Memory placement    | 0xF0000–0xFFFFF                                                                                                                                                                                                                                                                    |
| Access type         | Read/write                                                                                                                                                                                                                                                                         |
| See also            | <i>The heap</i> , page 47 and <i>New and Delete operators (78K0R only)</i> , page 114.                                                                                                                                                                                             |

## NEAR\_I

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds <code>__near</code> static and global initialized variables initialized by copying from the segment <code>NEAR_ID</code> at application startup.<br><br>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the <code>-Z</code> directive must be used. |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Memory placement    | 0x00000–0x0FFFF (78K0 and 78K0S)<br>0xF0000–0xFFFFF (78K0R)                                                                                                                                                                                                                                                                                                                                                               |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                                                |

## NEAR\_ID

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds initial values for <code>__near</code> static and global variables in the <code>NEAR_I</code> segment. These values are copied from <code>NEAR_ID</code> to <code>NEAR_I</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | CONST                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| Access type         | Read-only                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

## NEAR\_N

|                     |                                                                        |
|---------------------|------------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __near</code> variables.       |
| Segment memory type | DATA                                                                   |
| Memory placement    | <p>0x00000-0x0FFFF (78K0 and 78K0S)</p> <p>0xF0000-0xFFFFF (78K0R)</p> |
| Access type         | Read/write                                                             |

## NEAR\_Z

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds zero-initialized <code>__near</code> static and global variables. The contents of this segment is declared by the system startup code.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Memory placement    | <p>0x00000-0x0FFFF (78K0 and 78K0S)</p> <p>0xF0000-0xFFFFF (78K0R)</p>                                                                                                                                                                                                                                                                                                                                                 |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                                             |

## RCODE

|                     |                                         |
|---------------------|-----------------------------------------|
| Description         | Holds startup and runtime library code. |
| Segment memory type | CODE                                    |
| Memory placement    | 0x00000-0x0FFFF                         |
| Access type         | Read-only                               |

## SADDR\_A

|             |                                                                                                                                                                                                                                                                                                                                                                |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Holds <code>__saddr</code> located data declared <code>__no_init</code> or <code>const</code>.</p> <p><i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker command file.</p> |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## SADDR\_I

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__saddr</code> static and global initialized variables initialized by copying from the segment <code>SADDR_ID</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the <code>-z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                               |
| Memory placement    | <p>0x0FE20-0x0FF1F (78K0 and 78K0S)</p> <p>0x0FE20-0x0FF1F (78K0R)</p>                                                                                                                                                                                                                                                                                                                                                             |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                                                         |

## SADDR\_ID

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Holds initial values for <code>__saddr</code> static and global variables in the <code>SADDR_I</code> segment. These values are copied from <code>SADDR_ID</code> to <code>SADDR_I</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the <code>-z</code> directive must be used.</p> |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|                     |                                                |
|---------------------|------------------------------------------------|
| Segment memory type | CONST                                          |
| Memory placement    | This segment can be placed anywhere in memory. |
| Access type         | Read-only                                      |

## SADDR\_N

|                     |                                                                                                                                                                       |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __saddr</code> variables. Variables defined using the <code>__no_init</code> object attribute will be placed in this segment. |
| Segment memory type | DATA                                                                                                                                                                  |
| Memory placement    | 0x0FE20-0xFF1F (78K0 and 78K0S)<br>0xFFE20-0xFFFF1F (78K0R)                                                                                                           |
| Access type         | Read/write                                                                                                                                                            |

## SADDR\_Z

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                  |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds zero-initialized <code>__saddr</code> static and global variables. The contents of this segment is declared by the system startup code.<br><br>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the <code>-Z</code> directive must be used. |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                             |
| Memory placement    | 0x0FE20-0xFF1F (78K0 and 78K0S)<br>0xFFE20-0xFFFF1F (78K0R)                                                                                                                                                                                                                                                                                                                                                      |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                                       |

## SWITCH

|                     |                                                |
|---------------------|------------------------------------------------|
| Description         | Holds switch tables.                           |
| Segment memory type | CONST                                          |
| Memory placement    | This segment can be placed anywhere in memory. |

|             |           |
|-------------|-----------|
| Access type | Read-only |
|-------------|-----------|

## WRKSEG

|             |                                                                                                                                |
|-------------|--------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds the short address working area when the option <code>--workseg_area</code> option is used. Extra register variable area. |
|-------------|--------------------------------------------------------------------------------------------------------------------------------|

|                     |      |
|---------------------|------|
| Segment memory type | DATA |
|---------------------|------|

|                  |                                                            |
|------------------|------------------------------------------------------------|
| Memory placement | 0x0FE20-0x0FF1F (78K0 and 78K0S)<br>0xFFE20-0xFF1F (78K0R) |
|------------------|------------------------------------------------------------|

|             |            |
|-------------|------------|
| Access type | Read/write |
|-------------|------------|

## XCODE

|             |                                                                                                                                                                                                                                                                                    |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__far_func</code> declared program code for the 78K0R core.<br><br>This segment holds user application code declared with the attribute <code>__far_func</code> . Place this segment in memory using the <code>-Z</code> or <code>-P</code> segment control directive. |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|                     |         |
|---------------------|---------|
| Segment memory type | FARCODE |
|---------------------|---------|

|                  |                                                |
|------------------|------------------------------------------------|
| Memory placement | This segment can be placed anywhere in memory. |
|------------------|------------------------------------------------|

|             |           |
|-------------|-----------|
| Access type | Read-only |
|-------------|-----------|



# Implementation-defined behavior

This chapter describes how the compilers handle the implementation-defined areas of the C language.

ISO 9899:1990, the International Organization for Standardization standard - *Programming Languages - C* (revision and redesign of ANSI X3.159-1989, American National Standard), changed by the ISO Amendment 1:1994, *Technical Corrigendum 1*, and *Technical Corrigendum 2*, contains an appendix called *Portability Issues*. The ISO appendix lists areas of the C language that ISO leaves open to each particular implementation.

Note: The compilers adhere to a freestanding implementation of the ISO standard for the C programming language. This means that parts of a standard library can be excluded in the implementation.

---

## Descriptions of implementation-defined behavior

This section follows the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

### Translation

#### Diagnostics (5.1.1.3)

Diagnostics are 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 compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

## Environment

### Arguments to main (5.1.2.2.1)

The function called at program startup is called `main`. No prototype was declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the IAR DLIB runtime environment, see *Customizing system initialization*, page 66. To change this behavior for the IAR CLIB runtime environment, see *Customizing system initialization*, page 88.

### Interactive devices (5.1.2.3)

The streams `stdin` and `stdout` are treated as interactive devices.

## Identifiers

### Significant characters without external linkage (6.1.2)

The number of significant initial characters in an identifier without external linkage is 200.

### Significant characters with external linkage (6.1.2)

The number of significant initial characters in an identifier with external linkage is 200.

### Case distinctions are significant (6.1.2)

Identifiers with external linkage are treated as case-sensitive.

## Characters

### Source and execution character sets (5.2.1)

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library

needs a multibyte character scanner to support a multibyte execution character set. The IAR CLIB Library does not support multibyte characters.

See *Locale*, page 72.

#### **Bits per character in execution character set (5.2.4.2.1)**

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

#### **Mapping of characters (6.1.3.4)**

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

#### **Unrepresented character constants (6.1.3.4)**

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

#### **Character constant with more than one character (6.1.3.4)**

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

#### **Converting multibyte characters (6.1.3.4)**

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compilers—is the ‘C’ locale. If you use the command line option `--enable_multibytes`, the IAR DLIB Library will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library. The IAR CLIB Library does not support multibyte characters.

See *Locale*, page 72.

**Range of 'plain' char (6.2.1.1)**

A 'plain' char has the same range as an unsigned char.

**Integers****Range of integer values (6.1.2.5)**

The representation of integer values are in the two's complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.

See *Basic data types*, page 182, for information about the ranges for the different integer types.

**Demotion of integers (6.2.1.2)**

Converting an integer to a shorter signed integer is made by truncation. If the value cannot be represented when converting an unsigned integer to a signed integer of equal length, the bit-pattern remains the same. In other words, a large enough value will be converted into a negative value.

**Signed bitwise operations (6.3)**

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

**Sign of the remainder on integer division (6.3.5)**

The sign of the remainder on integer division is the same as the sign of the dividend.

**Negative valued signed right shifts (6.3.7)**

The result of a right-shift of a negative-valued signed integral type preserves the sign-bit. For example, shifting `0xFF00` down one step yields `0xFF80`.

**Floating point****Representation of floating-point values (6.1.2.5)**

The representation and sets of the various floating-point numbers adheres to IEEE 854–1987. A typical floating-point number is built up of a sign-bit (*s*), a biased exponent (*e*), and a mantissa (*m*).

See *Floating-point types*, page 184, for information about the ranges and sizes for the different floating-point types: `float` and `double`.

**Converting integer values to floating-point values (6.2.1.3)**

When an integral number is cast to a floating-point value that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

**Demoting floating-point values (6.2.1.4)**

When a floating-point value is converted to a floating-point value of narrower type that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

**Arrays and pointers****size\_t (6.3.3.4, 7.1.1)**

See *size\_t*, page 186, for information about *size\_t*.

**Conversion from/to pointers (6.3.4)**

See *Casting*, page 186, for information about casting of data pointers and function pointers.

**ptrdiff\_t (6.3.6, 7.1.1)**

See *ptrdiff\_t*, page 186, for information about the *ptrdiff\_t*.

**Registers****Honoring the register keyword (6.5.1)**

User requests for register variables are not honored.

**Structures, unions, enumerations, and bitfields****Improper access to a union (6.3.2.3)**

If a union gets its value stored through a member and is then accessed using a member of a different type, the result is solely dependent on the internal storage of the first member.

**Padding and alignment of structure members (6.5.2.1)**

See the section *Basic data types*, page 182, for information about the alignment requirement for data objects.

### **Sign of 'plain' bitfields (6.5.2.1)**

A 'plain' `int` bitfield is treated as a signed `int` bitfield. All integer types are allowed as bitfields.

### **Allocation order of bitfields within a unit (6.5.2.1)**

Bitfields are allocated within an integer from least-significant to most-significant bit.

### **Can bitfields straddle a storage-unit boundary (6.5.2.1)**

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

### **Integer type chosen to represent enumeration types (6.5.2.2)**

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

## **Qualifiers**

### **Access to volatile objects (6.5.3)**

Any reference to an object with volatile qualified type is an access.

## **Declarators**

### **Maximum numbers of declarators (6.5.4)**

The number of declarators is not limited. The number is limited only by the available memory.

## **Statements**

### **Maximum number of case statements (6.6.4.2)**

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

## Preprocessing directives

### Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a `signed` character.

### Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

### Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file is not found, the search continues as if the filename was enclosed in angle brackets.

### Character sequences (6.8.2)

Preprocessor directives use the source character set, except for escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile", "rt");
```

### Recognized pragma directives (6.8.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect:

```
alignment
baseaddr
building_runtime
can_instantiate
codeseg
cspy_support
define_type_info
```

do\_not\_instantiate  
 early\_dynamic\_initialization  
 function  
 hdrstop  
 important\_typedef  
 instantiate  
 keep\_definition  
 memory  
 module\_name  
 no\_pch  
 once  
 \_\_printf\_args  
 public\_equ  
 \_\_scanf\_args  
 section  
 STDC  
 system\_include  
 warnings

### **Default `__DATE__` and `__TIME__` (6.8.8)**

The definitions for `__TIME__` and `__DATE__` are always available.

## **IAR DLIB Library functions**

The information in this section is valid only if the runtime library configuration you have chosen supports file descriptors. See the chapter *The DLIB runtime environment* for more information about runtime library configurations.

### **NULL macro (7.1.6)**

The `NULL` macro is defined to 0.

### **Diagnostic printed by the `assert` function (7.2)**

The `assert()` function prints:

```
filename:linenr expression -- assertion failed
```

when the parameter evaluates to zero.

**Domain errors (7.5.1)**

NaN (Not a Number) will be returned by the mathematic functions on domain errors.

**Underflow of floating-point values sets `errno` to `ERANGE` (7.5.1)**

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

**`fmod()` functionality (7.5.6.4)**

If the second argument to `fmod()` is zero, the function returns NaN; `errno` is set to `EDOM`.

**`signal()` (7.7.1.1)**

The signal part of the library is not supported.

**Note:** Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 75.

**Terminating newline character (7.9.2)**

`stdout` stream functions recognize either `newline` or end of file (EOF) as the terminating character for a line.

**Blank lines (7.9.2)**

Space characters written to the `stdout` stream immediately before a newline character are preserved. There is no way to read the line through the `stdin` stream that was written through the `stdout` stream.

**Null characters appended to data written to binary streams (7.9.2)**

No null characters are appended to data written to binary streams.

**Files (7.9.3)**

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 71.

**`remove()` (7.9.4.1)**

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 71.

**rename() (7.9.4.2)**

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 71.

**%p in printf() (7.9.6.1)**

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `void *`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

**%p in scanf() (7.9.6.2)**

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.

**Reading ranges in scanf() (7.9.6.2)**

A - (dash) character is always treated as a range symbol.

**File position errors (7.9.9.1, 7.9.9.4)**

On file position errors, the functions `fgetpos` and `ftell` store `EFPOS` in `errno`.

**Message generated by perror() (7.9.10.4)**

The generated message is:

*usersuppliedprefix: errormessage*

**Allocating zero bytes of memory (7.10.3)**

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`.

**Behavior of abort() (7.10.4.1)**

The `abort()` function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

**Behavior of exit() (7.10.4.3)**

The argument passed to the `exit` function will be the return value returned by the `main` function to `cstartup`.

### Environment (7.10.4.4)

The set of available environment names and the method for altering the environment list is described in *Environment interaction*, page 74.

### system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 74.

### Message returned by strerror() (7.11.6.2)

The messages returned by `strerror()` depending on the argument is:

| Argument   | Message                   |
|------------|---------------------------|
| EZERO      | no error                  |
| EDOM       | domain error              |
| ERANGE     | range error               |
| EFPOS      | file positioning error    |
| EILSEQ     | multi-byte encoding error |
| <0    >99  | unknown error             |
| all others | error <i>nnn</i>          |

Table 51: Message returned by `strerror()`—IAR DLIB library

### The time zone (7.12.1)

The local time zone and daylight savings time implementation is described in *Time*, page 75.

### clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 75.

## IAR CLIB Library functions

### NULL macro (7.1.6)

The `NULL` macro is defined to `(void *) 0`.

### Diagnostic printed by the assert function (7.2)

The `assert()` function prints:

Assertion failed: *expression*, file *Filename*, line *linenumber*

when the parameter evaluates to zero.

### **Domain errors (7.5.1)**

`HUGE_VAL`, the largest representable value in a double floating-point type, will be returned by the mathematic functions on domain errors.

### **Underflow of floating-point values sets `errno` to `ERANGE` (7.5.1)**

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

### **`fmod()` functionality (7.5.6.4)**

If the second argument to `fmod()` is zero, the function returns zero (it does not change the integer expression `errno`).

### **`signal()` (7.7.1.1)**

The signal part of the library is not supported.

### **Terminating newline character (7.9.2)**

`stdout` stream functions recognize either `newline` or end of file (EOF) as the terminating character for a line.

### **Blank lines (7.9.2)**

Space characters written to the `stdout` stream immediately before a newline character are preserved. There is no way to read the line through the `stdin` stream that was written through the `stdout` stream.

### **Null characters appended to data written to binary streams (7.9.2)**

There are no binary streams implemented.

### **Files (7.9.3)**

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

### **`remove()` (7.9.4.1)**

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

**rename() (7.9.4.2)**

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

**%p in printf() (7.9.6.1)**

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `'char *'`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

**%p in scanf() (7.9.6.2)**

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `'void *'`.

**Reading ranges in scanf() (7.9.6.2)**

A `-` (dash) character is always treated explicitly as a `-` character.

**File position errors (7.9.9.1, 7.9.9.4)**

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

**Message generated by perror() (7.9.10.4)**

`perror()` is not supported.

**Allocating zero bytes of memory (7.10.3)**

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`.

**Behavior of abort() (7.10.4.1)**

The `abort()` function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

**Behavior of exit() (7.10.4.3)**

The `exit()` function does not return.

**Environment (7.10.4.4)**

Environments are not supported.

### **system() (7.10.4.5)**

The `system()` function is not supported.

### **Message returned by strerror() (7.11.6.2)**

The messages returned by `strerror()` depending on the argument are:

| <b>Argument</b> | <b>Message</b>      |
|-----------------|---------------------|
| EZERO           | no error            |
| EDOM            | domain error        |
| ERANGE          | range error         |
| <0    >99       | unknown error       |
| all others      | error No. <i>xx</i> |

*Table 52: Message returned by strerror()—IAR CLIB library*

### **The time zone (7.12.1)**

The time zone function is not supported.

### **clock() (7.12.2.1)**

The `clock()` function is not supported.

## A

- abort
  - implementation-defined behavior (CLIB) . . . . . 279
  - implementation-defined behavior (DLIB) . . . . . 276
  - system termination (DLIB) . . . . . 66
- absolute location
  - data, placing at (@) . . . . . 127
  - language support for . . . . . 194
  - #pragma location . . . . . 225
- addressing. *See* memory types, data models, and code models
- aggressive\_inlining (compiler option) . . . . . 154
- algorithm (STL header file) . . . . . 247
- alignment . . . . . 181
  - forcing stricter (#pragma data\_alignment) . . . . . 220
  - in structures (#pragma pack) . . . . . 227
  - in structures, causing problems . . . . . 124
  - of an object (\_\_ALIGNOF\_\_) . . . . . 195
  - of data types. . . . . 182
  - restrictions for inline assembler . . . . . 93
- alignment (pragma directive) . . . . . 273
- \_\_ALIGNOF\_\_ (operator) . . . . . 195
- anonymous structures . . . . . 125
- anonymous symbols, creating . . . . . 197
- application
  - building, overview of . . . . . 4
  - startup and termination (CLIB) . . . . . 87
  - startup and termination (DLIB) . . . . . 64
- architecture
  - 78K0R . . . . . 11
  - 78K0/78K0S . . . . . 11
- ARGFRAME (assembler directive) . . . . . 103
- arrays
  - designated initializers in . . . . . 198
  - hints about index type . . . . . 123
  - implementation-defined behavior. . . . . 271
  - incomplete at end of structs . . . . . 197
  - non-lvalue . . . . . 201
  - of incomplete types . . . . . 199
  - single-value initialization . . . . . 201
- asm, \_\_asm (language extension) . . . . . 196
- assembler code
  - calling from C . . . . . 94
  - calling from C++ . . . . . 96
  - inserting inline . . . . . 93
- assembler directives
  - for call frame information . . . . . 104
  - for static overlay . . . . . 103
  - using in inline assembler code . . . . . 93
- assembler instructions
  - CALLF . . . . . 209
  - CALLT . . . . . 209
  - inserting inline . . . . . 93
- assembler labels, making public (--public\_eq) . . . . . 177
- assembler language interface . . . . . 91
  - calling convention. *See* assembler code
- assembler list file, generating . . . . . 167
- assembler output file . . . . . 96
- assembler, inline . . . . . 196
- asserts . . . . . 76
  - implementation-defined behavior of, (CLIB). . . . . 277
  - implementation-defined behavior of, (DLIB). . . . . 274
  - including in application . . . . . 240
- assert.h (CLIB header file) . . . . . 251
- assert.h (DLIB header file) . . . . . 245
- atoll, C99 extension . . . . . 250
- atomic operations . . . . . 24
  - \_\_monitor . . . . . 211
- attributes
  - object . . . . . 206
  - type . . . . . 204
- auto variables . . . . . 18
  - at function entrance . . . . . 99
  - programming hints for efficient code . . . . . 135
  - using in inline assembler code . . . . . 94

|                                                                                              |          |
|----------------------------------------------------------------------------------------------|----------|
| <b>B</b>                                                                                     |          |
| backtrace information <i>See</i> call frame information                                      |          |
| bank number                                                                                  | 29       |
| bank switching routine                                                                       | 34       |
| modifying                                                                                    | 36       |
| bank (pragma directive)                                                                      | 218      |
| __banked (extended keyword)                                                                  | 208      |
| banked applications, debugging                                                               | 37       |
| banked code                                                                                  |          |
| downloading to memory                                                                        | 36       |
| memory map, example                                                                          | 30       |
| banked systems, coding hints                                                                 | 31       |
| Banked (code model)                                                                          | 21       |
| setting                                                                                      | 155      |
| using                                                                                        | 28       |
| Barr, Michael                                                                                | xxiv     |
| baseaddr (pragma directive)                                                                  | 273      |
| __BASE_FILE__ (predefined symbol)                                                            | 238      |
| basic type names, using in preprocessor expressions<br>(--migration_preprocessor_extensions) | 169      |
| basic_template_matching (pragma directive)                                                   | 219      |
| using                                                                                        | 116      |
| batch files, error return codes                                                              | 146      |
| BCODE (segment)                                                                              | 255      |
| binary streams (CLIB)                                                                        | 278      |
| binary streams (DLIB)                                                                        | 275      |
| bit negation                                                                                 | 137      |
| bitfields                                                                                    |          |
| data representation of                                                                       | 183      |
| hints                                                                                        | 123      |
| implementation-defined behavior of                                                           | 271      |
| non-standard types in                                                                        | 195      |
| specifying order of members (#pragma bitfields)                                              | 219      |
| bitfields (pragma directive)                                                                 | 219      |
| bold style, in this guide                                                                    | xxv      |
| bool (data type)                                                                             | 182      |
| adding support for in CLIB                                                                   | 251      |
| adding support for in DLIB                                                                   | 246, 248 |
| making available in C code                                                                   | 249      |
| __break (intrinsic function)                                                                 | 234      |
| BRK (assembler instruction)                                                                  | 234      |
| bubble sort function, defined in stdlib.h                                                    | 250      |
| building_runtime (pragma directive)                                                          | 273      |
| __BUILD_NUMBER__ (predefined symbol)                                                         | 238      |
| byte order, identifying (__LITTLE_ENDIAN__)                                                  | 239      |
| <b>C</b>                                                                                     |          |
| C and C++ linkage                                                                            | 98       |
| C/C++ calling convention. <i>See</i> calling convention                                      |          |
| C header files                                                                               | 245      |
| call frame information                                                                       | 104      |
| in assembler list file                                                                       | 95       |
| in assembler list file (-IA)                                                                 | 167      |
| call stack                                                                                   | 104      |
| callee-save registers, stored on stack                                                       | 19       |
| __calf (extended keyword)                                                                    | 209      |
| CALLF (assembler instruction)                                                                | 209      |
| calling convention                                                                           |          |
| C++, requiring C linkage                                                                     | 96       |
| in compiler                                                                                  | 97       |
| calloc (library function)                                                                    | 20       |
| <i>See also</i> heap                                                                         |          |
| implementation-defined behavior of (CLIB)                                                    | 279      |
| implementation-defined behavior of (DLIB)                                                    | 276      |
| __callt (extended keyword)                                                                   | 209      |
| callt functions, placement in memory                                                         | 51       |
| CALLT (assembler instruction)                                                                | 209      |
| can_instantiate (pragma directive)                                                           | 273      |
| cassert (DLIB header file)                                                                   | 248      |
| cast operators                                                                               |          |
| in Extended EC++                                                                             | 110      |
| missing from Embedded C++                                                                    | 110      |
| casting, pointers and integers                                                               | 186      |
| cctype (DLIB header file)                                                                    | 248      |
| cerrno (DLIB header file)                                                                    | 248      |

- cexit (system termination code)
  - in DLIB ..... 64
  - placement in segment ..... 49
- CFI (assembler directive) ..... 104
- cfloat (DLIB header file) ..... 248
- char (data type) ..... 182
  - changing default representation (`--char_is_signed`) ... 155
  - signed and unsigned ..... 183
- characters, implementation-defined behavior of ..... 268
- character-based I/O
  - in CLIB ..... 85
  - in DLIB ..... 68
  - overriding in runtime library ..... 61
- `--char_is_signed` (compiler option) ..... 155
- CHECKSUM (segment) ..... 255
- cinttypes (DLIB header file) ..... 248
- class memory (extended EC++) ..... 112
- class template partial specialization
  - matching (extended EC++) ..... 116
- classes ..... 111
- CLIB ..... 8, 250
  - reference information
  - runtime environment ..... 83
  - summary of definitions ..... 251
- `--clib` (compiler option) ..... 155
- climits (DLIB header file) ..... 248
- locale (DLIB header file) ..... 248
- clock (CLIB library function),
  - implementation-defined behavior of ..... 280
- clock (DLIB library function),
  - implementation-defined behavior of ..... 277
- clock.c ..... 75
- `__close` (DLIB library function) ..... 71
- CLTVEC (segment) ..... 255
- cmath (DLIB header file) ..... 248
- code
  - banked, downloading to memory ..... 36
  - interruption of execution ..... 23
  - verifying linked result ..... 51
- code execution ..... 7
- code models ..... 21
  - Banked ..... 21
    - setting ..... 155
  - configuration ..... 7
  - Far ..... 22
    - setting ..... 155
  - identifying (`__CODE_MODEL__`) ..... 238
  - Near ..... 21
    - setting ..... 155
  - specifying on command line (`--code_model`) ..... 155
  - Standard ..... 21
    - setting ..... 155
- code motion (compiler transformation) ..... 134
  - disabling (`--no_code_motion`) ..... 170
- code segment names
  - specifying ..... 156
- code segments, used for placement ..... 49
- CODE (segment) ..... 256
- codeseg (pragma directive) ..... 273
- `__CODE_MODEL__` (predefined symbol) ..... 238
- `__code_model` (runtime model attribute) ..... 80
- `--code_model` (compiler option) ..... 155
- command line options
  - part of compiler invocation syntax ..... 143
  - passing ..... 143
  - See also* compiler options
  - typographic convention ..... xxv
- command prompt icon, in this guide ..... xxv
- comments
  - after preprocessor directives ..... 201
  - C++ style, using in C code ..... 196
- common block (call frame information) ..... 104
- common subexpr elimination (compiler transformation) . 133
  - disabling (`--no_cse`) ..... 170
- compilation date
  - exact time of (`__TIME__`) ..... 240
  - identifying (`__DATE__`) ..... 238
- compiler
  - environment variables ..... 144

|                                                                |     |                                               |          |
|----------------------------------------------------------------|-----|-----------------------------------------------|----------|
| invocation syntax . . . . .                                    | 143 | non-top level . . . . .                       | 200      |
| output from . . . . .                                          | 145 | CONST (segment) . . . . .                     | 256      |
| compiler listing, generating (-l). . . . .                     | 167 | constants, placing in named segment . . . . . | 220      |
| compiler object file . . . . .                                 | 4   | constseg (pragma directive) . . . . .         | 220      |
| including debug information in (--debug, -r) . . . . .         | 158 | const_cast (cast operator) . . . . .          | 110      |
| output from compiler . . . . .                                 | 145 | contents, of this guide . . . . .             | xxii     |
| compiler optimization levels . . . . .                         | 131 | conventions, used in this guide . . . . .     | xxiv     |
| compiler options . . . . .                                     | 149 | copyright notice . . . . .                    | ii       |
| passing to compiler . . . . .                                  | 143 | __CORE__ (predefined symbol). . . . .         | 238      |
| reading from file (-f) . . . . .                               | 165 | core                                          |          |
| specifying parameters . . . . .                                | 151 | identifying . . . . .                         | 238      |
| summary . . . . .                                              | 152 | specifying on command line . . . . .          | 156      |
| syntax. . . . .                                                | 149 | --core (compiler option) . . . . .            | 156      |
| for creating skeleton code . . . . .                           | 95  | __cplusplus (predefined symbol) . . . . .     | 238      |
| --clib . . . . .                                               | 155 | csetjmp (DLIB header file) . . . . .          | 248      |
| --disable_data_alignment. . . . .                              | 161 | csignal (DLIB header file) . . . . .          | 248      |
| --generate_callt_runtime_library_calls. . . . .                | 166 | cspy_support (pragma directive) . . . . .     | 273      |
| --generate_far_runtime_library_calls. . . . .                  | 166 | CSTACK (segment) . . . . .                    | 256      |
| --near_constant_location . . . . .                             | 170 | example . . . . .                             | 46       |
| --warnings_affect_exit_code . . . . .                          | 146 | <i>See also</i> stack                         |          |
| --workseg_area . . . . .                                       | 179 | CSTART (segment). . . . .                     | 49       |
| compiler platform, identifying . . . . .                       | 239 | cstartup (system startup code). . . . .       | 49, 87   |
| compiler subversion number . . . . .                           | 240 | customizing . . . . .                         | 67       |
| compiler transformations . . . . .                             | 130 | overriding in runtime library . . . . .       | 61       |
| compiler version number . . . . .                              | 240 | cstartup.s26. . . . .                         | 64       |
| compiling                                                      |     | cstdarg (DLIB header file) . . . . .          | 248      |
| from the command line . . . . .                                | 5   | cstdbool (DLIB header file) . . . . .         | 248      |
| syntax. . . . .                                                | 143 | cstddef (DLIB header file) . . . . .          | 248      |
| complex numbers, supported in Embedded C++. . . . .            | 110 | cstdio (DLIB header file) . . . . .           | 248      |
| complex (library header file). . . . .                         | 246 | cstdlib (DLIB header file). . . . .           | 248      |
| compound literals . . . . .                                    | 197 | cstring (DLIB header file). . . . .           | 248      |
| computer style, typographic convention . . . . .               | xxv | ctime (DLIB header file). . . . .             | 248      |
| configuration                                                  |     | ctype.h (library header file). . . . .        | 245, 251 |
| basic project settings . . . . .                               | 6   | added C functionality. . . . .                | 249      |
| __low_level_init . . . . .                                     | 67  | cwctype.h (library header file) . . . . .     | 248      |
| configuration symbols, in library configuration files. . . . . | 63  | ?C_EXIT (assembler label). . . . .            | 89       |
| consistency, module . . . . .                                  | 78  | ?C_GETCHAR (assembler label). . . . .         | 89       |
| const                                                          |     | C_INCLUDE (environment variable) . . . . .    | 144      |
| declaring objects . . . . .                                    | 190 | ?C_PUTCHAR (assembler label) . . . . .        | 89       |

- C-SPY
    - interface to system termination . . . . . 66
    - low-level interface . . . . . 77, 88
    - STL container support . . . . . 118
  - C++
    - See also* Embedded C++ and Extended Embedded C++
    - absolute location . . . . . 129
    - calling convention . . . . . 96
    - dynamic initialization in . . . . . 51
    - features excluded from EC++ . . . . . 109
    - header files . . . . . 246–247
    - language extensions . . . . . 120
    - special function types . . . . . 27
    - static member variables . . . . . 129
    - support for . . . . . 3
  - C++ names, in assembler code . . . . . 97
  - C++ objects, placing in memory type . . . . . 17
  - C++ terminology. . . . . xxiv
  - C++-style comments . . . . . 196
  - C99 standard, added functionality from . . . . . 248
- ## D
- D (compiler option) . . . . . 157
  - data\_model (compiler option) . . . . . 157
  - data
    - alignment of . . . . . 181
    - different ways of storing . . . . . 11
    - located, declaring extern . . . . . 128
    - placing . . . . . 126, 221, 253
      - at absolute location . . . . . 127
    - representation of . . . . . 181
    - storage . . . . . 11
    - verifying linked result . . . . . 51
  - data block (call frame information) . . . . . 104
  - data memory attributes, using . . . . . 14
  - data models . . . . . 12
    - configuration . . . . . 7
    - Far . . . . . 13
      - identifying (`__DATA_MODEL__`) . . . . . 238
      - Near . . . . . 13
  - data pointers . . . . . 185
  - data segments . . . . . 43
  - data types . . . . . 182
    - avoiding signed . . . . . 123
    - floating point . . . . . 184
    - in C++ . . . . . 191
    - integers . . . . . 182
  - dataset (pragma directive) . . . . . 221
  - data\_alignment (pragma directive) . . . . . 220
  - `__DATA_MODEL__` (predefined symbol) . . . . . 238
  - `__data_model` (runtime model attribute) . . . . . 80
  - `__DATE__` (predefined symbol) . . . . . 238
  - date (library function), configuring support for . . . . . 75
  - DC32 (assembler directive) . . . . . 93
  - debug (compiler option) . . . . . 158
  - debug information, including in object file . . . . . 158, 177
  - declarations
    - empty . . . . . 201
    - in for loops . . . . . 196
    - Kernighan & Ritchie . . . . . 136
    - of functions . . . . . 98
  - declarations and statements, mixing . . . . . 196
  - declarators, implementation-defined behavior . . . . . 272
  - define\_type\_info (pragma directive) . . . . . 273
  - delete operator (extended EC++) . . . . . 114
  - delete (keyword) . . . . . 20
  - denormalized numbers. *See* subnormal numbers
  - dependencies (compiler option) . . . . . 158
  - deque (STL header file) . . . . . 247
  - destructors and interrupts, using . . . . . 119
  - DI (assembler instruction) . . . . . 234
  - diagnostic messages . . . . . 146
    - classifying as compilation errors . . . . . 159
    - classifying as compilation remarks . . . . . 160
    - classifying as compiler warnings . . . . . 160
    - disabling compiler warnings . . . . . 173
    - disabling wrapping of in compiler . . . . . 173

|                                                                    |          |
|--------------------------------------------------------------------|----------|
| enabling compiler remarks . . . . .                                | 177      |
| listing all used by compiler . . . . .                             | 161      |
| suppressing in compiler . . . . .                                  | 160      |
| --diagnostics_tables (compiler option) . . . . .                   | 161      |
| diag_default (pragma directive) . . . . .                          | 221      |
| --diag_error (compiler option) . . . . .                           | 159      |
| diag_error (pragma directive) . . . . .                            | 222      |
| --diag_remark (compiler option) . . . . .                          | 160      |
| diag_remark (pragma directive) . . . . .                           | 222      |
| --diag_suppress (compiler option) . . . . .                        | 160      |
| diag_suppress (pragma directive) . . . . .                         | 222      |
| --diag_warning (compiler option) . . . . .                         | 160      |
| diag_warning (pragma directive) . . . . .                          | 223      |
| DIFUNCT (segment) . . . . .                                        | 51, 257  |
| directives                                                         |          |
| function for static overlay . . . . .                              | 103      |
| pragma . . . . .                                                   | 10, 217  |
| directory, specifying as parameter. . . . .                        | 150      |
| --disable_data_alignment (compiler option) . . . . .               | 161      |
| __disable_interrupt (intrinsic function) . . . . .                 | 234      |
| --discard_unused_publics (compiler option) . . . . .               | 163      |
| disclaimer . . . . .                                               | ii       |
| DLIB. . . . .                                                      | 8, 245   |
| building customized library . . . . .                              | 55       |
| configurations . . . . .                                           | 55       |
| configuring. . . . .                                               | 54, 163  |
| debug support. . . . .                                             | 56       |
| reference information. <i>See</i> the online help system . . . . . | 243      |
| runtime environment . . . . .                                      | 53       |
| --dlib_config (compiler option) . . . . .                          | 163      |
| DLib_defaults.h (library configuration file) . . . . .             | 63       |
| dl78klibname.h . . . . .                                           | 63       |
| document conventions. . . . .                                      | xxiv     |
| documentation, library . . . . .                                   | 243      |
| domain errors, implementation-defined behavior . . . . .           | 275, 278 |
| double (data type) . . . . .                                       | 184      |
| double_t, C99 extension . . . . .                                  | 249      |
| do_not_instantiate (pragma directive) . . . . .                    | 274      |

|                                  |        |
|----------------------------------|--------|
| dynamic initialization . . . . . | 64, 87 |
| in C++ . . . . .                 | 51     |
| dynamic memory . . . . .         | 20     |

## E

|                                                        |          |
|--------------------------------------------------------|----------|
| -e (compiler option) . . . . .                         | 164      |
| early_initialization (pragma directive) . . . . .      | 274      |
| --ec++ (compiler option) . . . . .                     | 164      |
| EC++ header files . . . . .                            | 246      |
| edition, of this guide . . . . .                       | ii       |
| --eec++ (compiler option) . . . . .                    | 164      |
| EI (assembler instruction) . . . . .                   | 234      |
| Embedded C++. . . . .                                  | 109      |
| differences from C++. . . . .                          | 109      |
| enabling . . . . .                                     | 164      |
| function linkage . . . . .                             | 98       |
| language extensions . . . . .                          | 109      |
| overview . . . . .                                     | 109      |
| Embedded C++ Technical Committee . . . . .             | xxiv     |
| embedded systems, IAR special support for . . . . .    | 9        |
| __embedded_cplusplus (predefined symbol) . . . . .     | 239      |
| __enable_interrupt (intrinsic function) . . . . .      | 234      |
| --enable_multibytes (compiler option) . . . . .        | 165      |
| entry label, program . . . . .                         | 64       |
| enumerations, implementation-defined behavior. . . . . | 271      |
| enums                                                  |          |
| data representation . . . . .                          | 183      |
| forward declarations of . . . . .                      | 200      |
| environment                                            |          |
| implementation-defined behavior. . . . .               | 268      |
| runtime (CLIB) . . . . .                               | 83       |
| runtime (DLIB) . . . . .                               | 53       |
| environment variables                                  |          |
| C_INCLUDE. . . . .                                     | 144      |
| QCC78K . . . . .                                       | 144      |
| EQU (assembler directive) . . . . .                    | 177      |
| errno.h (library header file) . . . . .                | 245, 251 |

error messages . . . . . 147  
     classifying for compiler . . . . . 159  
 error return codes . . . . . 146  
 --error\_limit (compiler option) . . . . . 165  
 exception flags, for floating-point values . . . . . 184  
 exception handling, missing from Embedded C++ . . . . . 109  
 exception vectors . . . . . 51  
 exception (library header file) . . . . . 246  
 \_Exit (library function) . . . . . 66  
 exit (library function) . . . . . 66  
     implementation-defined behavior. . . . . 276, 279  
 \_exit (library function) . . . . . 66  
 \_\_exit (library function) . . . . . 66  
 export keyword, missing from Extended EC++ . . . . . 116  
 extended command line file  
     for compiler . . . . . 165  
     passing options. . . . . 143  
 Extended Embedded C++ . . . . . 110  
     enabling . . . . . 164  
     standard template library (STL) . . . . . 247  
 extended keywords . . . . . 203  
     enabling (-e) . . . . . 164  
     overview . . . . . 10  
     summary . . . . . 207  
     syntax. . . . . 15  
     object attributes. . . . . 207  
     type attributes on data objects . . . . . 204  
     type attributes on data pointers . . . . . 205  
     type attributes on function pointers . . . . . 206  
     type attributes on functions . . . . . 206  
 extern "C" linkage . . . . . 114

## F

-f (compiler option) . . . . . 165  
 \_\_far (extended keyword) . . . . . 210  
 Far (code model) . . . . . 22  
     setting . . . . . 155  
 Far (data model) . . . . . 13

far (memory type) . . . . . 14  
 FAR\_A (segment) . . . . . 257  
 ?FAR\_CALL\_L07 (bank switching routine) . . . . . 34  
 FAR\_CONST (segment) . . . . . 257  
 ?FAR\_FAST\_CALL\_L07 (bank switching routine) . . . . . 34  
 FAR\_FAST\_RETURN\_L07 (bank switching routine) . . . . . 34  
 \_\_far\_func (extended keyword) . . . . . 210  
 FAR\_HEAP (segment) . . . . . 258  
 FAR\_I (segment) . . . . . 258  
 FAR\_ID (segment) . . . . . 258  
 FAR\_N (segment) . . . . . 259  
 FAR\_RETURN\_L07 (bank switching routine) . . . . . 35  
 \_\_far\_size\_t . . . . . 114  
 FAR\_Z (segment) . . . . . 259  
 fatal error messages . . . . . 148  
 FCODE (segment) . . . . . 259  
 fenv.h (library header file) . . . . . 251  
 fgetpos (library function), implementation-defined  
     behavior . . . . . 276  
 field width, library support for . . . . . 86  
 \_\_FILE\_\_ (predefined symbol) . . . . . 239  
 file dependencies, tracking . . . . . 158  
 file paths, specifying for #include files . . . . . 167  
 file systems . . . . . 278  
 filename, specifying as parameter . . . . . 150  
 float (data type) . . . . . 184  
 floating-point constants  
     hexadecimal notation . . . . . 198  
 floating-point expressions,  
     using in preprocessor extensions. . . . . 169  
 floating-point format. . . . . 184  
     hints . . . . . 124  
     implementation-defined behavior. . . . . 270  
     special cases. . . . . 184  
     32-bits . . . . . 184  
 floating-point numbers, library support for . . . . . 86  
 float\_t, C99 extension . . . . . 249  
 float.h (library header file) . . . . . 245, 251  
 fmod (library function),  
     implementation-defined behavior . . . . . 275, 278

|                                                                                  |          |
|----------------------------------------------------------------------------------|----------|
| for loops, declarations in . . . . .                                             | 196      |
| formats                                                                          |          |
| floating-point values . . . . .                                                  | 184      |
| standard IEEE (floating point) . . . . .                                         | 184      |
| <code>_formatted_write</code> (library function) . . . . .                       | 85       |
| <code>fpclassify</code> , C99 extension . . . . .                                | 249      |
| <code>FP_INFINITE</code> , C99 extension . . . . .                               | 249      |
| <code>FP_NAN</code> , C99 extension . . . . .                                    | 249      |
| <code>FP_NORMAL</code> , C99 extension . . . . .                                 | 249      |
| <code>FP_SUBNORMAL</code> , C99 extension . . . . .                              | 249      |
| <code>FP_ZERO</code> , C99 extension . . . . .                                   | 249      |
| fragmentation, of heap memory . . . . .                                          | 20       |
| <code>free</code> (library function). <i>See also</i> heap . . . . .             | 20       |
| <code>fstream</code> (library header file) . . . . .                             | 246      |
| <code>fstream.h</code> (library header file) . . . . .                           | 247      |
| <code>ftell</code> (library function), implementation-defined behavior . . . . . | 276      |
| Full DLIB (library configuration) . . . . .                                      | 55       |
| <code>__func__</code> (predefined symbol) . . . . .                              | 202, 239 |
| <code>FUNCALL</code> (assembler directive) . . . . .                             | 103      |
| <code>__FUNCTION__</code> (predefined symbol) . . . . .                          | 202, 239 |
| function calls                                                                   |          |
| calling convention . . . . .                                                     | 97       |
| stack image after . . . . .                                                      | 101      |
| function calls, banked vs. non-banked . . . . .                                  | 32       |
| function declarations, Kernighan & Ritchie . . . . .                             | 136      |
| function directives for static overlay . . . . .                                 | 103      |
| function inlining (compiler transformation) . . . . .                            | 133      |
| disabling ( <code>--no_inline</code> ) . . . . .                                 | 171      |
| function pointers . . . . .                                                      | 185      |
| function prototypes . . . . .                                                    | 136      |
| enforcing . . . . .                                                              | 178      |
| function template parameter deduction (extended EC++) . . . . .                  | 116      |
| function type information, omitting in object output . . . . .                   | 175      |
| <code>FUNCTION</code> (assembler directive) . . . . .                            | 103      |
| function (pragma directive) . . . . .                                            | 274      |
| functional (STL header file) . . . . .                                           | 247      |
| functions . . . . .                                                              | 21       |
| C++ and special function types . . . . .                                         | 27       |
| declared without attribute, placement . . . . .                                  | 49       |

|                                   |                    |
|-----------------------------------|--------------------|
| declaring . . . . .               | 98, 136            |
| inlining . . . . .                | 133, 135, 196, 224 |
| interrupt . . . . .               | 23–24              |
| intrinsic . . . . .               | 91, 135            |
| monitor . . . . .                 | 24                 |
| omitting type info . . . . .      | 175                |
| parameters . . . . .              | 99                 |
| placing in memory . . . . .       | 126, 129, 156      |
| recursive                         |                    |
| avoiding . . . . .                | 136                |
| storing data on stack . . . . .   | 19                 |
| reentrancy (DLIB) . . . . .       | 244                |
| related extensions . . . . .      | 21                 |
| return values from . . . . .      | 101                |
| special function types . . . . .  | 23                 |
| verifying linked result . . . . . | 51                 |

## G

|                                                                                    |     |
|------------------------------------------------------------------------------------|-----|
| <code>--generate_callt_runtime_library_calls</code><br>(compiler option) . . . . . | 166 |
| <code>--generate_far_runtime_library_calls</code><br>(compiler option) . . . . .   | 166 |
| <code>getchar</code> (library function) . . . . .                                  | 85  |
| <code>getenv</code> (library function), configuring support for . . . . .          | 74  |
| <code>getzone</code> (library function), configuring support for . . . . .         | 75  |
| <code>getzone.c</code> . . . . .                                                   | 75  |
| <code>__get_interrupt_state</code> (intrinsic function) . . . . .                  | 234 |
| global variables                                                                   |     |
| initialization . . . . .                                                           | 45  |
| guidelines, reading . . . . .                                                      | xxi |

## H

|                                                    |      |
|----------------------------------------------------|------|
| <code>__halt</code> (intrinsic function) . . . . . | 235  |
| Harbison, Samuel P. . . . .                        | xxiv |
| hardware multiplier/divider unit . . . . .         | 76   |
| hardware support . . . . .                         | 76   |
| in compiler . . . . .                              | 54   |

- hash\_map (STL header file) . . . . . 247
  - hash\_set (STL header file) . . . . . 247
  - hdrstop (pragma directive) . . . . . 274
  - header files
    - C . . . . . 245
    - C++ . . . . . 246–247
    - EC++ . . . . . 246
    - library . . . . . 243
    - special function registers . . . . . 138
    - STL . . . . . 247
    - assert.h . . . . . 251
    - ctype.h . . . . . 251
    - DLib\_defaults.h . . . . . 63
    - dl78klibname.h . . . . . 63
    - errno.h . . . . . 251
    - fenv.h . . . . . 251
    - float.h . . . . . 251
    - iccbutl.h . . . . . 251
    - intrinsics.h . . . . . 233
    - limits.h . . . . . 251
    - math.h . . . . . 251
    - setjmp.h . . . . . 251
    - stdarg.h . . . . . 251
    - stdbool.h . . . . . 182, 246, 251
    - stddef.h . . . . . 183, 251
    - stdio.h . . . . . 251
    - stdlib.h . . . . . 251
    - string.h . . . . . 251
  - header\_context (compiler option) . . . . . 166
  - heap
    - DLib support for multiple . . . . . 76
    - dynamic memory . . . . . 20
    - segments for . . . . . 47
    - storing data . . . . . 12
  - heap segments
    - CLIB . . . . . 48
    - DLib . . . . . 47
    - FAR\_HEAP (segment) . . . . . 258
    - HEAP (segment) . . . . . 260
    - NEAR\_HEAP (segment) . . . . . 261
    - placing . . . . . 48
  - heap size
    - and standard I/O . . . . . 48
    - changing default . . . . . 48
  - HEAP (segment) . . . . . 48, 260
  - hints
    - banked systems . . . . . 31
    - optimization . . . . . 135
  - HUGE\_VALF, C99 extension . . . . . 249
  - HUGE\_VALL, C99 extension . . . . . 249
- I**
- I (compiler option) . . . . . 167
  - IAR Command Line Build Utility . . . . . 62
  - IAR Systems Technical Support . . . . . 148
  - iarbuild.exe (utility) . . . . . 62
  - \_\_IAR\_SYSTEMS\_ICC\_\_ (predefined symbol) . . . . . 239
  - iccbutl.h (library header file) . . . . . 251
  - \_\_ICC78K\_\_ (predefined symbol) . . . . . 239
  - icons, in this guide . . . . . xxv
  - identifiers, implementation-defined behavior . . . . . 268
  - IDE, building applications from . . . . . 4
  - IEEE format, floating-point values . . . . . 184
  - implementation-defined behavior . . . . . 267
  - important\_typedef (pragma directive) . . . . . 274
  - include files
    - including before source files . . . . . 176
    - specifying . . . . . 144
  - include\_alias (pragma directive) . . . . . 223
  - infinity . . . . . 184
  - INFINITY, C99 extension . . . . . 249
  - inheritance, in Embedded C++ . . . . . 109
  - initialization
    - dynamic . . . . . 64, 87
    - single-value . . . . . 201
  - initialized data segments . . . . . 45
  - initializers, static . . . . . 200

|                                              |         |
|----------------------------------------------|---------|
| inline assembler                             | 93, 196 |
| avoiding                                     | 135     |
| <i>See also</i> assembler language interface |         |
| inline functions                             | 196     |
| in compiler                                  | 133     |
| inline (pragma directive)                    | 224     |
| instantiate (pragma directive)               | 274     |
| integer characteristics, adding              | 249     |
| integers                                     | 182     |
| casting                                      | 186     |
| implementation-defined behavior              | 270     |
| intptr_t                                     | 187     |
| ptrdiff_t                                    | 186     |
| size_t                                       | 186     |
| uintptr_t                                    | 187     |
| integral promotion                           | 137     |
| internal error                               | 148     |
| __interrupt (extended keyword)               | 23, 211 |
| using in pragma directives                   | 232     |
| interrupt functions                          | 23      |
| placement in memory                          | 51      |
| interrupt state, restoring                   | 235     |
| interrupt vector                             |         |
| specifying with pragma directive             | 232     |
| interrupt vector table                       | 23–24   |
| in linker command file                       | 51      |
| INTVEC segment                               | 260     |
| interrupts                                   |         |
| disabling                                    | 211     |
| during function execution                    | 24      |
| processor state                              | 19      |
| using with EC++ destructors                  | 119     |
| intptr_t (integer type)                      | 187     |
| __intrinsic (extended keyword)               | 211     |
| intrinsic functions                          | 135     |
| overview                                     | 91      |
| summary                                      | 233     |
| intrinsics.h (header file)                   | 233     |
| inttypes.h (library header file)             | 245     |
| inttypes.h, added C functionality            | 249     |
| INTVEC (segment)                             | 51, 260 |
| intwri.c (library source code)               | 86      |
| invocation syntax                            | 143     |
| iomanip (library header file)                | 246     |
| iomanip.h (library header file)              | 247     |
| ios (library header file)                    | 246     |
| iosfwd (library header file)                 | 246     |
| iostream (library header file)               | 246     |
| iostream.h (library header file)             | 247     |
| isblank, C99 extension                       | 249     |
| isfinite, C99 extension                      | 249     |
| isgreater, C99 extension                     | 249     |
| isinf, C99 extension                         | 249     |
| islessequal, C99 extension                   | 249     |
| islessgreater, C99 extension                 | 249     |
| isless, C99 extension                        | 249     |
| isnan, C99 extension                         | 249     |
| isnormal, C99 extension                      | 249     |
| ISO/ANSI C                                   |         |
| compiler extensions                          | 193     |
| C++ features excluded from EC++              | 109     |
| library compliance with                      | 8, 243  |
| specifying strict usage                      | 178     |
| iso646.h (library header file)               | 245     |
| istream (library header file)                | 246     |
| isunordered, C99 extension                   | 249     |
| iswblank, C99 extension                      | 250     |
| italic style, in this guide                  | xxv     |
| iterator (STL header file)                   | 247     |
| I/O debugging, support for                   | 77      |
| I/O module, overriding in runtime library    | 61      |
| I/O, character-based                         | 85      |
| <br><b>K</b>                                 |         |
| keep_definition (pragma directive)           | 274     |
| Kernighan & Ritchie function declarations    | 136     |
| disallowing                                  | 178     |

Kernighan, Brian W. . . . . xxiv  
 keywords. . . . . 203  
   extended, overview of . . . . . 10

## L

-l (compiler option). . . . . 167  
   for creating skeleton code . . . . . 95  
 labels. . . . . 201  
   assembler, making public. . . . . 177  
   \_\_program\_start. . . . . 64  
 Labrosse, Jean J. . . . . xxiv  
 Lajoie, Josée . . . . . xxiv  
 language extensions  
   descriptions . . . . . 193  
   Embedded C++ . . . . . 109  
   enabling . . . . . 224  
   enabling (-e). . . . . 164  
 language overview . . . . . 3  
 language (pragma directive) . . . . . 224  
 libraries  
   building DLIB . . . . . 55  
   CLIB . . . . . 83  
   definition of . . . . . 4  
   runtime. . . . . 56  
   standard template library . . . . . 247  
 library configuration files  
   DLIB . . . . . 55  
   DLib\_defaults.h . . . . . 63  
   dl78klibname.h . . . . . 63  
   modifying . . . . . 63  
   specifying . . . . . 163  
 library documentation. . . . . 243  
 library features, missing from Embedded C++ . . . . . 110  
 library functions . . . . . 243  
   reference information. . . . . xxiii  
   summary, CLIB . . . . . 251  
   summary, DLIB . . . . . 245  
 library header files . . . . . 243

library modules  
   creating . . . . . 168  
   overriding. . . . . 61  
 library object files . . . . . 244  
 library options, setting . . . . . 9  
 library project template. . . . . 8, 63  
 --library\_module (compiler option) . . . . . 168  
 lightbulb icon, in this guide. . . . . xxv  
 limits.h (library header file) . . . . . 245, 251  
 \_\_LINE\_\_ (predefined symbol) . . . . . 239  
 linkage, C and C++. . . . . 98  
 linker command file . . . . . 40  
   customizing . . . . . 40, 51  
   using the -P command . . . . . 29, 42–43  
   using the -Z command . . . . . 41  
 linker map file. . . . . 52  
 linker output files . . . . . 5  
 linker segment. *See* segment  
 linking  
   from the command line . . . . . 5  
   required input. . . . . 5  
 Lippman, Stanley B. . . . . xxiv  
 list (STL header file). . . . . 247  
 listing, generating . . . . . 167  
 literals, compound. . . . . 197  
 literature, recommended . . . . . xxiv  
 \_\_LITTLE\_ENDIAN\_\_ (predefined symbol). . . . . 239  
 llabs, C99 extension . . . . . 250  
 lldiv, C99 extension . . . . . 250  
 local variables, *See* auto variables  
 locale support  
   DLIB . . . . . 72  
   adding . . . . . 73  
   changing at runtime. . . . . 73  
   removing . . . . . 73  
 locale.h (library header file) . . . . . 245  
 located data segments . . . . . 49  
 located data, declaring extern . . . . . 128  
 location (pragma directive) . . . . . 128, 225

|                                                      |          |
|------------------------------------------------------|----------|
| LOCFRAME (assembler directive) . . . . .             | 103      |
| long double (data type) . . . . .                    | 184      |
| long float (data type), synonym for double . . . . . | 200      |
| loop overhead, reducing . . . . .                    | 133, 173 |
| loop unrolling (compiler transformation) . . . . .   | 133      |
| disabling . . . . .                                  | 173      |
| loop-invariant expressions . . . . .                 | 134      |
| __low_level_init . . . . .                           | 64       |
| customizing . . . . .                                | 67       |
| low_level_init.c . . . . .                           | 64, 87   |
| low-level processor operations . . . . .             | 193, 233 |
| accessing . . . . .                                  | 91       |
| __lseek (library function) . . . . .                 | 71       |

## M

|                                             |          |
|---------------------------------------------|----------|
| macros, variadic . . . . .                  | 242      |
| main (function), definition . . . . .       | 268      |
| malloc (library function)                   |          |
| <i>See also</i> heap . . . . .              | 20       |
| implementation-defined behavior . . . . .   | 276, 279 |
| Mann, Bernhard . . . . .                    | xxiv     |
| map (STL header file) . . . . .             | 247      |
| map, linker . . . . .                       | 52       |
| MATH_ERREXCEPT, C99 extension . . . . .     | 249      |
| math_errhandling, C99 extension . . . . .   | 249      |
| MATH_ERRNO, C99 extension . . . . .         | 249      |
| math.h (library header file) . . . . .      | 245, 251 |
| math.h, added C functionality . . . . .     | 249      |
| __medium_write (library function) . . . . . | 86       |
| member functions, pointers to . . . . .     | 119      |
| memory                                      |          |
| accessing . . . . .                         | 13       |
| allocating in C++ . . . . .                 | 20       |
| dynamic . . . . .                           | 20       |
| heap . . . . .                              | 20       |
| non-initialized . . . . .                   | 139      |
| RAM, saving . . . . .                       | 136      |
| releasing in C++ . . . . .                  | 20       |

|                                                                 |         |
|-----------------------------------------------------------------|---------|
| stack . . . . .                                                 | 18      |
| saving . . . . .                                                | 136     |
| used by global or static variables . . . . .                    | 12      |
| memory consumption, reducing . . . . .                          | 85      |
| memory layout                                                   |         |
| 78K0R . . . . .                                                 | 11      |
| 78K0/78K0S . . . . .                                            | 11      |
| memory management, type-safe . . . . .                          | 109     |
| memory map                                                      |         |
| customizing the linker command file for . . . . .               | 40      |
| for banked code . . . . .                                       | 30      |
| memory placement                                                |         |
| using pragma directive . . . . .                                | 15      |
| using type definitions . . . . .                                | 16, 205 |
| memory segment. <i>See</i> segment                              |         |
| memory types . . . . .                                          | 13      |
| C++ . . . . .                                                   | 17      |
| far . . . . .                                                   | 14      |
| near . . . . .                                                  | 14      |
| placing variables in . . . . .                                  | 17      |
| pointers . . . . .                                              | 16      |
| SFR . . . . .                                                   | 14      |
| short address area . . . . .                                    | 14      |
| specifying . . . . .                                            | 14      |
| structures . . . . .                                            | 16      |
| summary . . . . .                                               | 15      |
| memory (pragma directive) . . . . .                             | 274     |
| memory (STL header file) . . . . .                              | 247     |
| __memory_of, operator . . . . .                                 | 113     |
| message (pragma directive) . . . . .                            | 225     |
| messages                                                        |         |
| disabling . . . . .                                             | 178     |
| forcing . . . . .                                               | 225     |
| --mfc (compiler option) . . . . .                               | 168     |
| --migration_preprocessor_extensions (compiler option) . . . . . | 169     |
| --misrac_verbose (compiler option) . . . . .                    | 153     |
| --misrac1998 (compiler option) . . . . .                        | 153     |
| --misrac2004 (compiler option) . . . . .                        | 153     |

module consistency . . . . . 78  
     rtmodel . . . . . 229  
 module map, in linker map file . . . . . 52  
 module name, specifying . . . . . 169  
 module summary, in linker map file . . . . . 52  
 --module\_name (compiler option) . . . . . 169  
 module\_name (pragma directive) . . . . . 274  
 \_\_monitor (extended keyword) . . . . . 138, 211  
 monitor functions . . . . . 24, 211  
 multibyte character support . . . . . 165  
 multiple inheritance, missing from Embedded C++ . . . . 109  
 multi-file compilation . . . . . 131  
 mutable attribute, in Extended EC++ . . . . . 110, 119

## N

names block (call frame information) . . . . . 104  
 namespace support  
     in Extended EC++ . . . . . 110, 119  
     missing from Embedded C++ . . . . . 110  
 naming conventions . . . . . xxv  
 NAN, C99 extension . . . . . 249  
 NDEBUG (preprocessor symbol) . . . . . 240  
 \_\_near (extended keyword) . . . . . 212  
 Near (code model) . . . . . 21  
     setting . . . . . 155  
 Near (data model) . . . . . 13  
 near (memory type) . . . . . 14  
 NEAR\_A (segment) . . . . . 260  
 NEAR\_CONST (segment) . . . . . 261  
 --near\_constant\_location (compiler option) . . . . . 170  
 \_\_near\_func (extended keyword) . . . . . 212  
 NEAR\_HEAP (segment) . . . . . 261  
 NEAR\_I (segment) . . . . . 261  
 NEAR\_ID (segment) . . . . . 262  
 NEAR\_N (segment) . . . . . 262  
 \_\_near\_size\_t . . . . . 114  
 NEAR\_Z (segment) . . . . . 262  
 new operator (extended EC++) . . . . . 114

new (keyword) . . . . . 20  
 new (library header file) . . . . . 246  
 new.h (library header file) . . . . . 247  
 \_\_non\_banked (extended keyword) . . . . . 213  
 non-initialized variables, hints for . . . . . 139  
 non-scalar parameters, avoiding . . . . . 136  
 NOP (assembler instruction) . . . . . 235  
 \_\_noreturn (extended keyword) . . . . . 214  
 Normal DLIB (library configuration) . . . . . 55  
 Not a number (NaN) . . . . . 184  
 \_\_no\_bit\_access (extended keyword) . . . . . 213  
 --no\_code\_motion (compiler option) . . . . . 170  
 --no\_cse (compiler option) . . . . . 170  
 \_\_no\_init (extended keyword) . . . . . 139, 213  
 --no\_inline (compiler option) . . . . . 171  
 \_\_no\_operation (intrinsic function) . . . . . 235  
 --no\_path\_in\_file\_macros (compiler option) . . . . . 171  
 no\_pch (pragma directive) . . . . . 274  
 \_\_no\_save (extended keyword) . . . . . 214  
 --no\_typedefs\_in\_diagnostics (compiler option) . . . . . 172  
 --no\_unroll (compiler option) . . . . . 173  
 --no\_warnings (compiler option) . . . . . 173  
 --no\_wrap\_diagnostics (compiler option) . . . . . 173  
 NULL . . . . . 251  
 NULL (macro), implementation-defined behavior . . 274, 277  
 numeric (STL header file) . . . . . 247

## O

-O (compiler option) . . . . . 174  
 -o (compiler option) . . . . . 174  
 object attributes . . . . . 206  
 object filename, specifying in compiler . . . . . 174  
 object module name, specifying . . . . . 169  
 object\_attribute (pragma directive) . . . . . 139, 226  
 offsetof . . . . . 251  
 --omit\_types (compiler option) . . . . . 175  
 once (pragma directive) . . . . . 274  
 --only\_stdout (compiler option) . . . . . 175

|                                                        |          |
|--------------------------------------------------------|----------|
| __open (library function) . . . . .                    | 71       |
| operators                                              |          |
| <i>See also</i> @ (operator)                           |          |
| __memory_of. . . . .                                   | 113      |
| optimization                                           |          |
| code motion, disabling . . . . .                       | 170      |
| common sub-expression elimination, disabling . . . . . | 170      |
| configuration . . . . .                                | 7        |
| disabling . . . . .                                    | 132      |
| function inlining, disabling (--no_inline). . . . .    | 171      |
| hints . . . . .                                        | 135      |
| loop unrolling, disabling . . . . .                    | 173      |
| specifying (-O). . . . .                               | 174      |
| summary . . . . .                                      | 131      |
| techniques . . . . .                                   | 132      |
| type-based alias analysis, disabling (--tbaa). . . . . | 172      |
| using inline assembler code . . . . .                  | 93       |
| using pragma directive . . . . .                       | 226      |
| optimization levels . . . . .                          | 131      |
| optimize (pragma directive) . . . . .                  | 226      |
| option parameters . . . . .                            | 149      |
| options, compiler. <i>See</i> compiler options         |          |
| Oram, Andy . . . . .                                   | xxiv     |
| ostream (library header file) . . . . .                | 246      |
| output                                                 |          |
| from preprocessor . . . . .                            | 176      |
| specifying for linker. . . . .                         | 5        |
| supporting non-standard. . . . .                       | 86       |
| --output (compiler option). . . . .                    | 175      |
| --output (linker option) . . . . .                     | 175      |
| overhead, reducing . . . . .                           | 133, 173 |

## P

|                                   |          |
|-----------------------------------|----------|
| pack (pragma directive) . . . . . | 188, 227 |
| packed structure types. . . . .   | 188      |
| parameters                        |          |
| function . . . . .                | 99       |
| hidden . . . . .                  | 100      |

|                                                                           |          |
|---------------------------------------------------------------------------|----------|
| non-scalar, avoiding . . . . .                                            | 136      |
| register . . . . .                                                        | 99–100   |
| rules for specifying a file or directory . . . . .                        | 150      |
| specifying . . . . .                                                      | 151      |
| stack. . . . .                                                            | 99–100   |
| parameters, typographic convention . . . . .                              | xxv      |
| part number, of this guide . . . . .                                      | ii       |
| permanent registers. . . . .                                              | 99       |
| pererror (library function),<br>implementation-defined behavior . . . . . | 276, 279 |
| placement                                                                 |          |
| code and data . . . . .                                                   | 253      |
| in named segments. . . . .                                                | 129      |
| pointer types . . . . .                                                   | 185      |
| mixing . . . . .                                                          | 200      |
| pointers                                                                  |          |
| casting . . . . .                                                         | 186      |
| data . . . . .                                                            | 185      |
| function . . . . .                                                        | 185      |
| implementation-defined behavior. . . . .                                  | 271      |
| polymorphism, in Embedded C++ . . . . .                                   | 109      |
| porting, code containing pragma directives. . . . .                       | 218      |
| _Pragma (predefined symbol). . . . .                                      | 241      |
| pragma directives . . . . .                                               | 10       |
| summary . . . . .                                                         | 217      |
| basic_template_matching, using . . . . .                                  | 116      |
| bitfields . . . . .                                                       | 183      |
| for absolute located data . . . . .                                       | 128      |
| list of all recognized. . . . .                                           | 273      |
| pack . . . . .                                                            | 188, 227 |
| type_attribute, using. . . . .                                            | 15       |
| precision arguments, library support for . . . . .                        | 86       |
| predefined symbols                                                        |          |
| overview . . . . .                                                        | 10       |
| summary . . . . .                                                         | 238      |
| --predef_macro (compiler option). . . . .                                 | 176      |
| --preinclude (compiler option) . . . . .                                  | 176      |
| --preprocess (compiler option) . . . . .                                  | 176      |
| preprocessor                                                              |          |
| output. . . . .                                                           | 176      |

- overview . . . . . 237
- preprocessor directives, implementation-defined behavior 273
- preprocessor extensions
  - compatibility . . . . . 169
  - `__VA_ARGS__` . . . . . 242
  - `#warning` message . . . . . 241
- preprocessor symbols . . . . . 238
  - defining . . . . . 157
- preserved registers . . . . . 99
  - `__PRETTY_FUNCTION__` (predefined symbol). . . . . 240
- primitives, for special functions . . . . . 23
- print formatter, selecting . . . . . 60
- `printf` (library function) . . . . . 59, 85
  - choosing formatter . . . . . 59
  - configuration symbols . . . . . 70
  - customizing . . . . . 86
  - implementation-defined behavior. . . . . 276, 279
  - selecting. . . . . 86
- `__printf_args` (pragma directive). . . . . 228
- processor cores, supported . . . . . 6
- processor operations
  - accessing . . . . . 91
  - low-level . . . . . 193, 233
- program entry label. . . . . 64
- programming hints . . . . . 135
  - banked systems . . . . . 31
- `__program_start` (label). . . . . 64
- projects, basic settings for. . . . . 6
- prototypes, enforcing . . . . . 178
- `ptrdiff_t` (integer type). . . . . 186, 251
- `PUBLIC` (assembler directive) . . . . . 177
- publication date, of this guide . . . . . ii
- `--public_equ` (compiler option) . . . . . 177
- `public_equ` (pragma directive) . . . . . 274
- `putchar` (library function) . . . . . 85
- `putenv` (library function), absent from DLIB . . . . . 74

## Q

- QCC78K (environment variable) . . . . . 144
- qualifiers
  - const and volatile . . . . . 189
  - implementation-defined behavior. . . . . 272
- queue (STL header file) . . . . . 247

## R

- `-r` (compiler option). . . . . 177
- `raise` (library function), configuring support for . . . . . 75
- `raise.c` . . . . . 75
- RAM
  - non-zero initialized variables . . . . . 45
  - saving memory. . . . . 136
- range errors, in linker . . . . . 52
- RCODE (segment) . . . . . 263
- `__read` (library function). . . . . 71
  - customizing . . . . . 68
- read formatter, selecting . . . . . 61, 87
- reading guidelines. . . . . xxv
- reading, recommended . . . . . xxiv
- `realloc` (library function)
  - implementation-defined behavior. . . . . 276, 279
  - See also* heap . . . . . 20
- recursive functions
  - avoiding . . . . . 136
  - storing data on stack . . . . . 19
- reenfrancy (DLIB). . . . . 244
- reference information, typographic convention. . . . . xxv
- register parameters . . . . . 99–100
- registered trademarks . . . . . ii
- registers
  - assigning to parameters . . . . . 100
  - callee-save, stored on stack . . . . . 19
  - for function returns . . . . . 101
  - implementation-defined behavior. . . . . 271
  - in assembler-level routines. . . . . 97

|                                        |              |
|----------------------------------------|--------------|
| preserved                              | 99           |
| scratch                                | 99           |
| reinterpret_cast (cast operator)       | 110          |
| remark (diagnostic message)            | 147          |
| classifying for compiler               | 160          |
| enabling in compiler                   | 177          |
| --remarks (compiler option)            | 177          |
| remove (library function)              | 71           |
| implementation-defined behavior        | 275, 278     |
| rename (library function)              | 71           |
| implementation-defined behavior        | 276, 279     |
| __ReportAssert (library function)      | 76           |
| required (pragma directive)            | 228          |
| --require_prototypes (compiler option) | 178          |
| return values, from functions          | 101          |
| Ritchie, Dennis M.                     | xxiv         |
| __root (extended keyword)              | 214          |
| routines, time-critical                | 91, 193, 233 |
| rtmodel (assembler directive)          | 80           |
| rtmodel (pragma directive)             | 229          |
| rtti support, missing from STL         | 110          |
| __rt_version (runtime model attribute) | 80           |
| runtime environment                    |              |
| CLIB                                   | 83           |
| DLIB                                   | 53           |
| setting options                        | 9            |
| runtime libraries                      |              |
| choosing                               | 9            |
| introduction                           | 243          |
| CLIB                                   | 83           |
| naming convention                      | 84           |
| DLIB                                   | 56           |
| choosing                               | 58           |
| customizing without rebuilding         | 58           |
| naming convention                      | 57           |
| overriding modules in                  | 61           |
| runtime model attributes               | 78           |
| __code_model                           | 80           |
| __data_model                           | 80           |

|                                                     |     |
|-----------------------------------------------------|-----|
| runtime model definitions                           | 229 |
| runtime type information, missing from Embedded C++ | 110 |

## S

|                                 |          |
|---------------------------------|----------|
| __saddr (extended keyword)      | 215      |
| SADDR_A (segment)               | 263      |
| SADDR_I (segment)               | 263      |
| SADDR_ID (segment)              | 263      |
| SADDR_N (segment)               | 264      |
| SADDR_Z (segment)               | 264      |
| scanf (library function)        | 87       |
| choosing formatter              | 60       |
| configuration symbols           | 70       |
| implementation-defined behavior | 276, 279 |
| __scanf_args (pragma directive) | 229      |
| scratch registers               | 99       |
| section (pragma directive)      | 274      |
| segment group name              | 44       |
| segment map, in linker map file | 52       |
| segment memory types, in XLINK  | 40       |
| segment names, declaring        | 230      |
| segment (pragma directive)      | 230      |
| segments                        | 253      |
| code                            | 49       |
| data                            | 43       |
| definition of                   | 39       |
| initialized data                | 45       |
| introduction                    | 39       |
| located data                    | 49       |
| naming                          | 44       |
| packing in memory               | 42       |
| placing in sequence             | 41       |
| static memory                   | 43       |
| summary                         | 253      |
| too long for address range      | 52       |
| too long, in linker             | 52       |
| HEAP                            | 48       |
| INTVEC                          | 51       |

- \_\_segment\_begin (extended operator) . . . . . 195
- \_\_segment\_end (extended operator) . . . . . 195
- semaphores
  - C example . . . . . 24
  - C++ example . . . . . 26
  - operations on . . . . . 211
- set (STL header file) . . . . . 247
- setjmp.h (library header file) . . . . . 245, 251
- setlocale (library function) . . . . . 73
- settings, basic for project configuration . . . . . 6
- \_\_set\_interrupt\_state (intrinsic function) . . . . . 235
- severity level, of diagnostic messages . . . . . 147
  - specifying . . . . . 148
- SFR
  - accessing special function registers . . . . . 138
  - declaring extern special function registers . . . . . 128
- \_\_sfr (extended keyword) . . . . . 215
- SFR (memory type) . . . . . 14
- shared object . . . . . 146
- short address area (memory type) . . . . . 14
- short (data type) . . . . . 182
- signal (library function)
  - configuring support for . . . . . 75
  - implementation-defined behavior . . . . . 275
- signal.c . . . . . 75
- signal.h (library header file) . . . . . 245
- signbit, C99 extension . . . . . 249
- signed char (data type) . . . . . 182–183
  - specifying . . . . . 155
- signed int (data type) . . . . . 182
- signed long (data type) . . . . . 182
- signed short (data type) . . . . . 182
- signed values, avoiding . . . . . 123
- silent (compiler option) . . . . . 178
- silent operation, specifying in compiler . . . . . 178
- sizeof, using in preprocessor extensions . . . . . 169
- size\_t (integer type) . . . . . 186, 251
- skeleton code, creating for assembler language interface . . 94
- skeleton.s26 (assembler source output) . . . . . 95
- slist (STL header file) . . . . . 247
- \_small\_write (library function) . . . . . 86
- snprintf, C99 extension . . . . . 250
- source files, list all referred . . . . . 166
- special function registers (SFR) . . . . . 138
- special function types . . . . . 23
  - overview . . . . . 10
- sprintf (library function) . . . . . 59, 85
  - choosing formatter . . . . . 59
  - customizing . . . . . 86
- sscanf (library function) . . . . . 87
  - choosing formatter . . . . . 60
- sstream (library header file) . . . . . 246
- stack . . . . . 18, 46
  - advantages and problems using . . . . . 19
  - changing default size of . . . . . 46
  - contents of . . . . . 18
  - internal data . . . . . 256
  - layout . . . . . 100
  - placing segment in memory . . . . . 47
  - saving space . . . . . 136
  - size . . . . . 47
- stack parameters . . . . . 99–100
- stack pointer . . . . . 19
- stack (STL header file) . . . . . 247
- standard error, redirecting in compiler . . . . . 175
- standard input . . . . . 68
- standard output . . . . . 68
  - specifying in compiler . . . . . 175
- standard template library (STL)
  - in Extended EC++ . . . . . 110, 117, 247
  - missing from Embedded C++ . . . . . 110
- Standard (code model) . . . . . 21
  - setting . . . . . 155
- startup code
  - placement of . . . . . 49
  - See also* CSTART
- startup, system
  - CLIB . . . . . 87

|                                             |               |
|---------------------------------------------|---------------|
| DLIB                                        | 64            |
| statements, implementation-defined behavior | 272           |
| static data, in linker command file         | 46            |
| static memory segments                      | 43            |
| static overlay                              | 103           |
| static variables                            | 11            |
| initialization                              | 45            |
| taking the address of                       | 135           |
| static_cast (cast operator)                 | 110           |
| std namespace, missing from EC++            |               |
| and Extended EC++                           | 119           |
| stdarg.h (library header file)              | 245, 251      |
| stdbool.h (library header file)             | 182, 246, 251 |
| added C functionality                       | 249           |
| __STDC__ (predefined symbol)                | 240           |
| STDC (pragma directive)                     | 274           |
| __STDC_VERSION__ (predefined symbol)        | 240           |
| stddef.h (library header file)              | 183, 246, 251 |
| stderr                                      | 71, 175       |
| stdexcept (library header file)             | 246           |
| stdin                                       | 71            |
| implementation-defined behavior             | 275, 278      |
| stdint.h (library header file)              | 246, 248      |
| stdint.h, added C functionality             | 249           |
| stdio.h (library header file)               | 246, 251      |
| stdio.h, additional C functionality         | 250           |
| stdlib.h (library header file)              | 246, 251      |
| stdlib.h, additional C functionality        | 250           |
| stdout                                      | 71, 175       |
| implementation-defined behavior             | 275, 278      |
| Steele, Guy L.                              | xxiv          |
| STL                                         | 117           |
| __stop (intrinsic function)                 | 235           |
| streambuf (library header file)             | 246           |
| streams, supported in Embedded C++          | 110           |
| strerror (library function)                 |               |
| implementation-defined behavior             | 277, 280      |
| --strict_ansi (compiler option)             | 178           |
| string (library header file)                | 246           |
| strings, supported in Embedded C++          | 110           |

|                                                    |             |
|----------------------------------------------------|-------------|
| string.h (library header file)                     | 246, 251    |
| Stroustrup, Bjarne                                 | xxiv        |
| strstream (library header file)                    | 246         |
| strtod (library function), configuring support for | 75          |
| strtod, in stdlib.h                                | 250         |
| strtof, C99 extension                              | 250         |
| strtold, C99 extension                             | 250         |
| strtoll, C99 extension                             | 250         |
| strtoull, C99 extension                            | 250         |
| structure types                                    |             |
| alignment                                          | 187–188     |
| layout of                                          | 187         |
| packed                                             | 188         |
| structures                                         |             |
| aligning                                           | 227         |
| anonymous                                          | 125, 195    |
| implementation-defined behavior                    | 271         |
| incomplete arrays as last element                  | 197         |
| packing and unpacking                              | 124         |
| placing in memory type                             | 16          |
| subnormal numbers                                  | 184         |
| __SUBVERSION__ (predefined symbol)                 | 240         |
| support, technical                                 | 148         |
| SWITCH (segment)                                   | 264         |
| symbol names, using in preprocessor extensions     | 169         |
| symbols                                            |             |
| anonymous, creating                                | 197         |
| including in output                                | 228         |
| listing in linker map file                         | 52          |
| overview of predefined                             | 10          |
| preprocessor, defining                             | 157         |
| syntax                                             |             |
| command line options                               | 149         |
| extended keywords                                  | 15, 204–207 |
| invoking compiler                                  | 143         |
| system startup                                     |             |
| CLIB                                               | 87          |
| customizing                                        | 66          |
| DLIB                                               | 64          |

system termination  
 CLIB ..... 88  
 C-SPY interface to ..... 66  
 DLIB ..... 65

system (library function)  
 configuring support for ..... 74  
 implementation-defined behavior ..... 277, 280

system\_include (pragma directive) ..... 274

## T

technical support, IAR Systems ..... 148

template support  
 in Extended EC++ ..... 110, 116  
 missing from Embedded C++ ..... 109

Terminal I/O window ..... 89  
 making available ..... 78

terminal output, speeding up ..... 78

termination, of system  
 CLIB ..... 88  
 DLIB ..... 65

terminology ..... xxiv

32-bits (floating-point format) ..... 184

this (pointer) ..... 96  
 class memory ..... 112  
 data type of ..... 18  
 referring to a class object ..... 112

\_\_TIME\_\_ (predefined symbol) ..... 240

time zone (library function)  
 implementation-defined behavior ..... 277, 280

time (library function), configuring support for ..... 75

time-critical routines ..... 91, 193, 233

time.c ..... 75

time.h (library header file) ..... 246

tips, programming ..... 135

tools icon, in this guide ..... xxv

trademarks ..... ii

transformations, compiler ..... 130

translation, implementation-defined behavior ..... 267

trap vectors, specifying with pragma directive ..... 232

type attributes ..... 204  
 specifying ..... 230

type definitions, used for specifying memory storage ..... 16, 205

type information, omitting ..... 175

type qualifiers  
 const and volatile ..... 189  
 implementation-defined behavior ..... 272

typedefs  
 excluding from diagnostics ..... 172  
 repeated ..... 200  
 using in preprocessor extensions ..... 169

type\_attribute (pragma directive) ..... 15, 230

type-based alias analysis (compiler transformation) ..... 134  
 disabling ..... 172

type-safe memory management ..... 109

typographic conventions ..... xxv

## U

UBROF  
 format of linkable object files ..... 145  
 specifying, example of ..... 5

uintptr\_t (integer type) ..... 187

underflow range errors,  
 implementation-defined behavior ..... 275, 278

unions  
 anonymous ..... 125, 195  
 implementation-defined behavior ..... 271

unroll (pragma directive) ..... 231

unsigned char (data type) ..... 182–183  
 changing to signed char ..... 155

unsigned int (data type) ..... 182

unsigned long (data type) ..... 182

unsigned short (data type) ..... 182

utility (STL header file) ..... 247

## V

|                                                              |         |
|--------------------------------------------------------------|---------|
| variable type information, omitting in object output . . . . | 175     |
| variables                                                    |         |
| auto . . . . .                                               | 18      |
| defined inside a function . . . . .                          | 18      |
| global, placement in memory . . . . .                        | 12      |
| hints for choosing . . . . .                                 | 135     |
| local. <i>See</i> auto variables                             |         |
| non-initialized . . . . .                                    | 139     |
| omitting type info . . . . .                                 | 175     |
| placing at absolute addresses . . . . .                      | 129     |
| placing in named segments . . . . .                          | 129     |
| static                                                       |         |
| placement in memory . . . . .                                | 12      |
| taking the address of . . . . .                              | 135     |
| static and global, initializing . . . . .                    | 45      |
| vector (pragma directive) . . . . .                          | 23, 231 |
| vector (STL header file) . . . . .                           | 247     |
| __VER__ (predefined symbol) . . . . .                        | 240     |
| version                                                      |         |
| compiler . . . . .                                           | 240     |
| IAR Embedded Workbench . . . . .                             | ii      |
| version, of compiler . . . . .                               | 240     |
| vfscanf, C99 extension . . . . .                             | 250     |
| vfwscanf, C99 extension . . . . .                            | 250     |
| void, pointers to . . . . .                                  | 200     |
| volatile (keyword) . . . . .                                 | 137     |
| volatile, declaring objects . . . . .                        | 189     |
| vscanf, C99 extension . . . . .                              | 250     |
| vsnprintf, C99 extension . . . . .                           | 250     |
| vsscanf, C99 extension . . . . .                             | 250     |
| vswscanf, C99 extension . . . . .                            | 250     |
| vwscanf, C99 extension . . . . .                             | 250     |
| __V2_call (extended keyword) . . . . .                       | 216     |
| __V3_call (extended keyword) . . . . .                       | 216     |

## W

|                                                        |          |
|--------------------------------------------------------|----------|
| #warning message (preprocessor extension) . . . . .    | 241      |
| warnings . . . . .                                     | 147      |
| classifying in compiler . . . . .                      | 160      |
| disabling in compiler . . . . .                        | 173      |
| exit code . . . . .                                    | 179      |
| warnings icon, in this guide . . . . .                 | xxv      |
| warnings (pragma directive) . . . . .                  | 274      |
| --warnings_affect_exit_code (compiler option) . . . .  | 146, 179 |
| --warnings_are_errors (compiler option) . . . . .      | 179      |
| wchar_t (data type), adding support for in C . . . . . | 183      |
| wchar.h (library header file) . . . . .                | 246, 248 |
| wchar.h, added C functionality . . . . .               | 250      |
| wcstof, C99 extension . . . . .                        | 250      |
| wcstolb, C99 extension . . . . .                       | 250      |
| wctype.h (library header file) . . . . .               | 246      |
| wctype.h, added C functionality . . . . .              | 250      |
| web sites, recommended . . . . .                       | xxiv     |
| --workseg_area (compiler option) . . . . .             | 179      |
| __write (library function) . . . . .                   | 71       |
| customizing . . . . .                                  | 68       |
| write formatter, selecting . . . . .                   | 86–87    |
| WRKSEG (segment) . . . . .                             | 265      |

## X

|                                      |     |
|--------------------------------------|-----|
| XCODE (segment) . . . . .            | 265 |
| XLINK errors                         |     |
| range error . . . . .                | 52  |
| segment too long . . . . .           | 52  |
| XLINK segment memory types . . . . . | 40  |
| xreportassert.c . . . . .            | 76  |

## Symbols

|                                    |     |
|------------------------------------|-----|
| _Exit (library function) . . . . . | 66  |
| _exit (library function) . . . . . | 66  |
| _Exit, C99 extension . . . . .     | 250 |

|                                                                   |          |                                                                   |          |
|-------------------------------------------------------------------|----------|-------------------------------------------------------------------|----------|
| <code>__formatted_write</code> (library function) . . . . .       | 85       | <code>__LITTLE_ENDIAN__</code> (predefined symbol) . . . . .      | 239      |
| <code>__medium_write</code> (library function) . . . . .          | 86       | <code>__low_level_init</code> . . . . .                           | 64       |
| <code>__Pragma</code> (predefined symbol) . . . . .               | 241      | <code>__low_level_init, customizing</code> . . . . .              | 67       |
| <code>__small_write</code> (library function) . . . . .           | 86       | <code>__lseek</code> (library function) . . . . .                 | 71       |
| <code>__ALIGNOF__</code> (operator) . . . . .                     | 195      | <code>__memory_of, operator</code> . . . . .                      | 113      |
| <code>__asm</code> (language extension) . . . . .                 | 196      | <code>__monitor</code> (extended keyword) . . . . .               | 138, 211 |
| <code>__banked</code> (extended keyword) . . . . .                | 208      | <code>__near</code> (extended keyword) . . . . .                  | 212      |
| <code>__BASE_FILE__</code> (predefined symbol) . . . . .          | 238      | <code>__near_func</code> (extended keyword) . . . . .             | 212      |
| <code>__break</code> (intrinsic function) . . . . .               | 234      | <code>__near_size_t</code> . . . . .                              | 114      |
| <code>__BUILD_NUMBER__</code> (predefined symbol) . . . . .       | 238      | <code>__non_banked</code> (extended keyword) . . . . .            | 213      |
| <code>__callf</code> (extended keyword) . . . . .                 | 209      | <code>__noreturn</code> (extended keyword) . . . . .              | 214      |
| <code>__callt</code> (extended keyword) . . . . .                 | 209      | <code>__no_bit_access</code> (extended keyword) . . . . .         | 213      |
| <code>__close</code> (library function) . . . . .                 | 71       | <code>__no_init</code> (extended keyword) . . . . .               | 139, 213 |
| <code>__code_model</code> (runtime model attribute) . . . . .     | 80       | <code>__no_operation</code> (intrinsic function) . . . . .        | 235      |
| <code>__CODE_MODEL__</code> (predefined symbol) . . . . .         | 238      | <code>__no_save</code> (extended keyword) . . . . .               | 214      |
| <code>__CORE__</code> (predefined symbol) . . . . .               | 238      | <code>__open</code> (library function) . . . . .                  | 71       |
| <code>__cplusplus</code> (predefined symbol) . . . . .            | 238      | <code>__PRETTY_FUNCTION__</code> (predefined symbol) . . . . .    | 240      |
| <code>__data_model</code> (runtime model attribute) . . . . .     | 80       | <code>__printf_args</code> (pragma directive) . . . . .           | 228, 274 |
| <code>__DATA_MODEL__</code> (predefined symbol) . . . . .         | 238      | <code>__program_start</code> (label) . . . . .                    | 64       |
| <code>__DATE__</code> (predefined symbol) . . . . .               | 238      | <code>__qsortbbl, C99 extension</code> . . . . .                  | 250      |
| <code>__disable_interrupt</code> (intrinsic function) . . . . .   | 234      | <code>__read</code> (library function) . . . . .                  | 71       |
| <code>__embedded_cplusplus</code> (predefined symbol) . . . . .   | 239      | customizing . . . . .                                             | 68       |
| <code>__enable_interrupt</code> (intrinsic function) . . . . .    | 234      | <code>__ReportAssert</code> (library function) . . . . .          | 76       |
| <code>__exit</code> (library function) . . . . .                  | 66       | <code>__root</code> (extended keyword) . . . . .                  | 214      |
| <code>__far</code> (extended keyword) . . . . .                   | 210      | <code>__rt_version</code> (runtime model attribute) . . . . .     | 80       |
| <code>__far_func</code> (extended keyword) . . . . .              | 210      | <code>__saddr</code> (extended keyword) . . . . .                 | 215      |
| <code>__far_size_t</code> . . . . .                               | 114      | <code>__scanf_args</code> (pragma directive) . . . . .            | 229, 274 |
| <code>__FILE__</code> (predefined symbol) . . . . .               | 239      | <code>__segment_begin</code> (extended operator) . . . . .        | 195      |
| <code>__FUNCTION__</code> (predefined symbol) . . . . .           | 202, 239 | <code>__segment_end</code> (extended operators) . . . . .         | 195      |
| <code>__func__</code> (predefined symbol) . . . . .               | 202, 239 | <code>__set_interrupt_state</code> (intrinsic function) . . . . . | 235      |
| <code>__gets, in stdio.h</code> . . . . .                         | 250      | <code>__sfr</code> (extended keyword) . . . . .                   | 215      |
| <code>__get_interrupt_state</code> (intrinsic function) . . . . . | 234      | <code>__STDC_VERSION__</code> (predefined symbol) . . . . .       | 240      |
| <code>__halt</code> (intrinsic function) . . . . .                | 235      | <code>__STDC__</code> (predefined symbol) . . . . .               | 240      |
| <code>__IAR_SYSTEMS_ICC__</code> (predefined symbol) . . . . .    | 239      | <code>__stop</code> (intrinsic function) . . . . .                | 235      |
| <code>__ICC78K__</code> (predefined symbol) . . . . .             | 239      | <code>__SUBVERSION__</code> (predefined symbol) . . . . .         | 240      |
| <code>__interrupt</code> (extended keyword) . . . . .             | 23, 211  | <code>__TIME__</code> (predefined symbol) . . . . .               | 240      |
| using in pragma directives . . . . .                              | 232      | <code>__ungetchar, in stdio.h</code> . . . . .                    | 250      |
| <code>__intrinsic</code> (extended keyword) . . . . .             | 211      | <code>__VA_ARGS__</code> (preprocessor extension) . . . . .       | 242      |
| <code>__LINE__</code> (predefined symbol) . . . . .               | 239      | <code>__VER__</code> (predefined symbol) . . . . .                | 240      |

|                                                                       |     |                                                                 |          |
|-----------------------------------------------------------------------|-----|-----------------------------------------------------------------|----------|
| __V2_call (extended keyword) . . . . .                                | 216 | --header_context (compiler option) . . . . .                    | 166      |
| __V3_call (extended keyword) . . . . .                                | 216 | --library_module (compiler option) . . . . .                    | 168      |
| __write (library function) . . . . .                                  | 71  | --mfc (compiler option) . . . . .                               | 168      |
| customizing . . . . .                                                 | 68  | --migration_preprocessor_extensions (compiler option) . . . . . | 169      |
| __write_array, in stdio.h . . . . .                                   | 250 | --misrac_verbose (compiler option) . . . . .                    | 153      |
| __write_buffered (DLIB library function) . . . . .                    | 78  | --misrac1998 (compiler option) . . . . .                        | 153      |
| -D (compiler option) . . . . .                                        | 157 | --misrac2004 (compiler option) . . . . .                        | 153      |
| -e (compiler option) . . . . .                                        | 164 | --module_name (compiler option) . . . . .                       | 169      |
| -f (compiler option) . . . . .                                        | 165 | --near_constant_location (compiler option) . . . . .            | 170      |
| -I (compiler option) . . . . .                                        | 167 | --no_code_motion (compiler option) . . . . .                    | 170      |
| -l (compiler option) . . . . .                                        | 167 | --no_cse (compiler option) . . . . .                            | 170      |
| for creating skeleton code . . . . .                                  | 95  | --no_inline (compiler option) . . . . .                         | 171      |
| -O (compiler option) . . . . .                                        | 174 | --no_path_in_file_macros (compiler option) . . . . .            | 171      |
| -o (compiler option) . . . . .                                        | 174 | --no_tbaa (compiler option) . . . . .                           | 172      |
| -r (compiler option) . . . . .                                        | 177 | --no_typedefs_in_diagnostics (compiler option) . . . . .        | 172      |
| --aggressive_inlining (compiler option) . . . . .                     | 154 | --no_unroll (compiler option) . . . . .                         | 173      |
| --char_is_signed (compiler option) . . . . .                          | 155 | --no_warnings (compiler option) . . . . .                       | 173      |
| --clib (compiler option) . . . . .                                    | 155 | --no_wrap_diagnostics (compiler option) . . . . .               | 173      |
| --code_model (compiler option) . . . . .                              | 155 | --omit_types (compiler option) . . . . .                        | 175      |
| --code_segment (compiler option) . . . . .                            | 156 | --only_stdout (compiler option) . . . . .                       | 175      |
| --core (compiler option) . . . . .                                    | 156 | --output (compiler option) . . . . .                            | 175      |
| --data_model (compiler option) . . . . .                              | 157 | --output (linker option) . . . . .                              | 175      |
| --debug (compiler option) . . . . .                                   | 158 | --predef_macro (compiler option) . . . . .                      | 176      |
| --dependencies (compiler option) . . . . .                            | 158 | --preinclude (compiler option) . . . . .                        | 176      |
| --diagnostics_tables (compiler option) . . . . .                      | 161 | --preprocess (compiler option) . . . . .                        | 176      |
| --diag_error (compiler option) . . . . .                              | 159 | --remarks (compiler option) . . . . .                           | 177      |
| --diag_remark (compiler option) . . . . .                             | 160 | --require_prototypes (compiler option) . . . . .                | 178      |
| --diag_suppress (compiler option) . . . . .                           | 160 | --silent (compiler option) . . . . .                            | 178      |
| --diag_warning (compiler option) . . . . .                            | 160 | --strict_ansi (compiler option) . . . . .                       | 178      |
| --disable_data_alignment (compiler option) . . . . .                  | 161 | --warnings_affect_exit_code (compiler option) . . . . .         | 146, 179 |
| --discard_unused_publics (compiler option) . . . . .                  | 163 | --warnings_are_errors (compiler option) . . . . .               | 179      |
| --dlib_config (compiler option) . . . . .                             | 163 | --workseg_area (compiler option) . . . . .                      | 179      |
| --ec++ (compiler option) . . . . .                                    | 164 | ?C_EXIT (assembler label) . . . . .                             | 89       |
| --eec++ (compiler option) . . . . .                                   | 164 | ?C_GETCHAR (assembler label) . . . . .                          | 89       |
| --enable_multibytes (compiler option) . . . . .                       | 165 | ?C_PUTCHAR (assembler label) . . . . .                          | 89       |
| --error_limit (compiler option) . . . . .                             | 165 | ?FAR_CALL_L07 (bank switching routine) . . . . .                | 34       |
| --generate_callt_runtime_library_calls<br>(compiler option) . . . . . | 166 | ?FAR_FAST_CALL_L07 (bank switching routine) . . . . .           | 34       |
| --generate_far_runtime_library_calls<br>(compiler option) . . . . .   | 166 | @ (operator)<br>placing at absolute address . . . . .           | 128      |

placing in segments ..... 129  
 #include files, specifying ..... 144, 167  
 #warning message (preprocessor extension) ..... 241

## Numerics

16-bit pointers, accessing memory ..... 13  
 24-bit pointers, accessing memory ..... 13  
 32-bits (floating-point format) ..... 184  
 78K, supported devices ..... 4  
 78K0R Compiler, introduction to ..... xxi  
 78K0R, memory layout ..... 11  
 78K0/78K0S Compiler, introduction to ..... xxi  
 78K0/78K0S, memory layout ..... 11