
M32C IAR C/C++ COMPILER

Reference Guide

for Renesas
**M32C and M16C/8x Series of
CPU Cores**

COPYRIGHT NOTICE

© Copyright 1999-2004 IAR Systems. All rights reserved.

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

DISCLAIMER

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

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

TRADEMARKS

IAR Embedded Workbench, IAR visualSTATE, IAR MakeApp, and IAR PreQual are registered trademarks owned by IAR Systems. C-SPY is a trademark registered in the European Union by IAR Systems. IAR, IAR XLINK Linker, IAR XAR Library Builder, and IAR XLIB Librarian are trademarks owned by IAR Systems.

M32C and M16C/8x Series are registered trademarks of Renesas Technology Corporation. Microsoft is a registered trademark, and Windows is a trademark of Microsoft Corporation. Intel and Pentium are registered trademarks of Intel Corporation.

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

Second edition: June 2004

Part number: CM32C-2

WELCOME

Welcome to the M32C IAR C/C++ Compiler Reference Guide.

This guide provides reference information about the IAR Systems C/C++ Compiler for the M32C and M16C/8x Series of CPU cores.

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

For information about programming with the M32C Assembler, refer to the *M32C IAR Assembler Reference Guide*.

ABOUT THIS GUIDE

This guide consists of the following chapters:

Introduction to the M32C IAR C/C++ Compiler provides a brief summary of the M32C IAR C/C++ Compiler's features and describes how the compiler represents each of the C data types. There are also recommendations for efficient coding, and a summary of the available language extensions.

Configuration describes how to configure the C/C++ compiler for different requirements.

Compiler options explains how to set the C/C++ compiler options, it gives a summary of the options, and contains complete reference information for each C compiler option.

EC++ library functions gives an introduction to the C/C++ library functions, and summarizes the header files.

Extended keywords reference gives reference information about each of the extended keywords.

#pragma directives reference gives reference information about the #pragma keywords.

Predefined symbols reference gives reference information about the predefined symbols.

Intrinsic functions reference gives reference information about the intrinsic functions.

Assembler language interface describes the interface between C/C++ programs and assembler language routines.

Segment reference gives reference information about the C/C++ compiler's use of segments.

Migration hints provides information that is useful when porting code from the M16C IAR C Compiler to the M32C IAR C/C++ Compiler.

Implementation-defined behavior describes how IAR C handles the implementation-defined areas of the C language.

IAR C extensions describes the IAR extensions to the ISO/ANSI standard for the C programming language.

Diagnostics describes the diagnostic functions and lists M32C-specific warning and error messages.

ASSUMPTIONS



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

- ◆ The M32C and M16C/8x Series of CPU cores
- ◆ The C/C++ programming languages
- ◆ The operating system of your host machine
- ◆ The IAR Systems development tools and the project model, as described in the *M32C IAR Embedded Workbench™ IDE User Guide*.

Note: The illustrations in this guide show the IAR Embedded Workbench running in a Windows 95-style environment, and their appearance will be slightly different if you are using a different platform.

CONVENTIONS

This guide uses the following typographical conventions:

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

CONTENTS

INTRODUCTION TO THE M32C IAR C/C + + COMPILER	1
Key features	1
Data representation	2
Pointers	5
Programming hints	7
Embedded C + + overview	10
Language extensions	11
 CONFIGURATION	 17
Introduction	17
Processor	18
Memory model	18
Linker command file	21
Run-time library	24
Stack and heap size	24
Input and output using the IAR C Library	25
Input and output using the EC + + library	29
Register I/O	32
Initialization	32
Interrupt system	34
 COMPILER OPTIONS	 37
Setting compiler options	37
Environment variables	39
Options summary	40
 C LIBRARY FUNCTIONS.....	 63
Introduction	63
Library definitions summary	64
math functions	65
 EC + + LIBRARY FUNCTIONS	 67
Introduction	67
Library definitions summary	68

EXTENDED KEYWORDS REFERENCE.....	73
Summary of extended keywords	73
Storage	74
Functions	79
Embedded C + +	84
 #PRAGMA DIRECTIVES REFERENCE	 85
Type attribute	85
Object attribute	86
Dataseg	87
Constseg	88
Location	88
Vector	88
Diagnostics	88
Language	89
Optimize	90
Pack	90
 PREDEFINED SYMBOLS REFERENCE.....	 93
 INTRINSIC FUNCTIONS REFERENCE	 97
 ASSEMBLER LANGUAGE INTERFACE.....	 105
Creating a shell	105
C calling convention	108
Interrupt handling	110
Monitor functions	111
Calling assembler routines from C	112
Run-time model	112
Embedded C + +	112
 SEGMENT REFERENCE.....	 115
 MIGRATION HINTS.....	 121
Introduction	121
Extended keywords	122
#pragma directives	125
Predefined symbols	127
Intrinsic functions	128
C compiler options	128

Segments	134
IMPLEMENTATION-DEFINED BEHAVIOR	135
Translation	135
Environment	136
Identifiers	136
Characters	136
Integers	138
Floating point	138
Arrays and pointers	139
Registers	139
Structures, unions, enumerations, and bitfields	139
Qualifiers	140
Declarators	140
Statements	141
Preprocessing directives	141
C library functions	143
EC + + library functions	146
IAR C EXTENSIONS	151
Available extensions	151
Extensions accepted in normal	
EC + + mode	154
Language features not accepted in	
EC + +	155
DIAGNOSTICS.....	157
Severity levels	157
INDEX.....	159

INTRODUCTION TO THE M32C IAR C/C++ COMPILER

In this chapter you will find information about the M32C IAR C/C++ Compiler's key features and its data representation. There is also a section containing hints on how to write programs efficiently for the M32C IAR C/C++ Compiler, and information about the language extensions available.

KEY FEATURES

The M32C IAR C/C++ Compiler offers the standard features of the C/C++ languages, plus many extensions designed to take advantage of the M32C and M16C/8x Series-specific facilities. The compiler is supplied with the IAR Systems Assembler for the M32C and M16C/8x Series CPU cores, with which it shares linker and librarian manager tools.

It provides the following features:

LANGUAGE FACILITIES

- ◆ Conformance to the ISO/ANSI standard for a free-standing environment.
- ◆ Standard library of functions applicable to embedded systems, with source optionally available.
- ◆ IEEE-compatible floating-point arithmetic.
- ◆ Powerful extensions for M32C and M16C/8x Series-specific features.
- ◆ External references are type-checked at link time.
- ◆ Linkage of user code with assembler routines.
- ◆ Long identifiers—up to 255 significant characters.
- ◆ Up to 32000 external symbols.

PERFORMANCE

- ◆ Memory-based design which avoids temporary files or overlays.
- ◆ Extensive type checking at compile time.

- ◆ Extensive module interface type checking at link time.

CODE GENERATION

- ◆ Selectable optimization for code speed or size.
- ◆ Comprehensive output options, including relocatable binary, assembler, assembler + C, etc.
- ◆ Easy-to-understand error and warning messages.
- ◆ Compatibility with the C-SPY™ high-level debugger.

TARGET SUPPORT

- ◆ Near, far, and huge memory models.
- ◆ Flexible variable allocation.
- ◆ Interrupt functions can be written in C.
- ◆ Several processor-specific interrupt mechanisms supported.
- ◆ #pragma directives to maintain portability while using processor-specific extensions.

**DATA
REPRESENTATION**

This section describes how the M32C IAR C/C + + Compiler represents each of the C data types.

The M32C IAR C/C + + Compiler supports all ISO/ANSI C basic elements. Variables are stored with the least significant part located at low memory address.

INTEGER TYPES

The following table gives the size and range of each C integer data type:

<i>Data type</i>	<i>Alignment</i>	<i>Size</i>	<i>Range</i>
signed char	1	8 bit	-128 to 127
char	1	8 bit	0 to 255
unsigned char			
short	2	16 bit	-32768 to 32767
signed short			
unsigned short	2	16 bit	0 to 65535

<i>Data type</i>	<i>Alignment</i>	<i>Size</i>	<i>Range</i>
int signed int	2	16 bit	-32768 to 32767
unsigned int	2	16 bit	0 to 65535
long signed long	2	32 bit	-2^{31} to $2^{31}-1$
unsigned long	2	32 bit	0 to $2^{32}-1$

Enum type

The enum keyword creates each object with the shortest integer type (char, short, or long) required to contain its value.

Char type

The char type is, by default, unsigned in the compiler, but the **‘char’ is ‘signed char’** (`--char_is_signed`) option allows you to make it signed. Notice, however, that the library is compiled with char types as unsigned.

Bitfields

The char, short, and long bitfields are extensions to the ANSI C integer bitfields.

Bitfields in expressions will have the same data type as the base type (signed or unsigned char, short, int, or long).

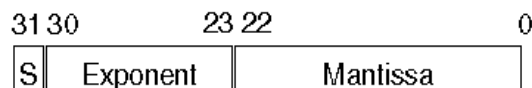
Bitfield variables are packed in elements of the specified type starting at the least significant position.

FLOATING-POINT TYPES

Floating-point values are represented by 4-byte numbers in standard IEEE format; float and double values have the same representation. Floating-point values below the smallest limit will be regarded as zero, and overflow gives undefined results. If the **64-bit floating point** (`-2`) option is used, double and long double types will use the 8-byte format. For additional information, see `-2`, page 61.

4-byte floating-point format

The memory layout of 4-byte floating-point numbers is:



POINTERS

This section describes the M32C IAR C/C++ Compiler's use of pointers.

Function pointers

The M32C IAR C/C++ Compiler has one type of function pointer, a 32-bit pointer that can access the whole code memory. Interrupt functions cannot be accessed through a function pointer.

Code pointers

Code pointers are always 24 bits, with a storage size of 4 bytes.

Data pointers

The data pointers are as follows:

<i>Keyword</i>	<i>Storage in bytes</i>	<i>Comment</i>
<code>__near</code>	2	Can only point into 0–64 Kbytes.
<code>__far</code>	4	Element pointed at must be inside a 64 Kbyte page.
<code>__huge</code>	4	No restrictions.

Casting

Casting an integer value to a pointer of a smaller size will be performed by truncation and casting to a larger pointer will be performed by zero extension.

Casting a pointer type to a smaller integer type will be performed by truncation. Casting to a larger integral type will be performed by first casting the pointer to the largest possible pointer that fits in the integer and then, if necessary, zero extended.

size_t

`size_t` is the unsigned integer type required to hold the maximum size of an object. The `size_t` integer type is unsigned in IAR C.

ptrdiff_t

`ptrdiff_t` is the type of integer required to hold the difference between two pointers to elements of the same array. The `ptrdiff_t` integer type is signed in IAR C.

STRUCTURES

Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address.

Anonymous structures and unions

An anonymous structure or union is a structure or union object that is declared without a name. Its members are promoted to the surrounding scope. An anonymous structure or union may not have a tag. In the example below, the members in the anonymous union can be accessed, in function `f`, without explicitly specifying the union name:

```
struct s
{
    char tag;
    union
    {
        long l;
        float f;
    };
} st;

void f()
{
    st.l = 5;
}
```

The member names must be unique in the surrounding scope. Having anonymous structures and unions at file scope, as a `global`, `external`, or `static` is also allowed. This is for instance used to declare special function registers (SFRs) as in the following example, where the union is anonymous:

```
__no_init volatile union
{
    unsigned char PORT_P0;
    struct
    {
        unsigned char mult    :1;
        unsigned char div     :1;
        unsigned char div_ovf :1;
        unsigned char         :5;
    } PORT_P0;
} @ 0x03E0;
```


The SFR has 3 bits declared: `mult`, `div`, and `div_ovf`. The SFR byte register (`PORT_P0`) is declared at address `0x03E0`.

PROGRAMMING HINTS

It is important to be aware of the limitations of the M32C and M16C/8x Series architecture in order to avoid the use of inefficient language constructs. The following list contains recommendations on how to write efficient code for the M32C IAR C/C++ Compiler:

- ◆ Avoid far and huge variables, since they require more space than the near variables.
- ◆ Avoid global variables if they are not needed. Local variables, and static variables in some cases, have a good chance of being allocated in register. Globals will always be allocated in memory.
- ◆ Unsigned data types (`unsigned char` and `unsigned short`) are usually handled more efficiently than their signed counterparts.
- ◆ For variables that do not need to be initialized, use the keyword `__no_init` to avoid the implicit zero initialization.
- ◆ Use ISO/ANSI function prototypes since they allow the compiler to generate more efficient code and give type checking of function parameters.

ACCESSING SPECIAL FUNCTION REGISTERS

A specific header file for M32C is included in the M32C IAR C/C++ Compiler delivery. The header file is named `iom32c.h` and defines the processor-specific SFRs.

Since the header file is also intended to be used with the M32C IAR Assembler, AM32C, the SFR declaration is made with macros. The macros that convert the declaration to assembler or compiler syntax are defined in the `iomacros.h` file.

Example

The following example shows how a control register (`CNTRL`) at address `0xE8` can be defined.

First we define the bits in the register:

```
typedef struct {  
    __REG8 tlenb :1;  
    __REG8 tlpndb :1;
```

```

    __REG8 wen      :1;
    __REG8 wpnd     :1;
    __REG8 t0en     :1;
    __REG8 t0pnd    :1;
    __REG8 lpen     :1;
    __REG8          :1;
} __icntrl_bits;

```

Then the register is defined:

```

__SFR_BITS(__CNTRL, 0xE8, __REG8, __READ_WRITE,
__cntrl_bits);

```

Note: __REG8 is converted to unsigned char and the __READ_WRITE attribute is currently not used.

The declaration is converted by the iomacros.h file:

```

__no_init volatile __nonbanked union
{
    unsigned char __CNTRL;
    __cntrl_bits __CNTRL_bit;
} @ 0xE8 ;

```

It is then possible to access either the whole register or any individual bit as follows:

```

// Byte access
__CNTRL = 0xAA;
// Bit access
__CNTRL_bit.t0en = 1;

```

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

```

#ifdef __IAR_SYSTEMS_ICC__
    (compiler-specific defines)
#endif

```

The header files are also suitable to use as templates, when creating new header files for other M32C and M16C/8x Series derivatives.

APPLICATION MEMORY USAGE

When using the command line version of the compiler and linker, the application memory usage will automatically be displayed on the screen after completed compilation or linking. The memory usage is displayed in decimal notation.

The compiler reports the memory usage for the compiled file per segment and then it summarizes for different memory types as follows:

```
2 bytes in segment IDATA0
2 bytes in segment CDATA0
2 bytes in segment NDATA0

9 bytes of CODE memory
70 bytes of NEARCODE memory
2 bytes of NEARCONST memory
4 bytes of NEARDATA memory
```

In this example, the 4 bytes of the NEARDATA memory is the sum of the IDATA0 and CDATA0 segments.

The linker reports the total memory usage for the linked application as follows:

```
144 bytes of CODE memory
68 bytes of DATA memory
```

Here the CODE memory size is the sum of all program and library code, variable initializers, and constant data.

The DATA memory size is the sum of the global variable memory size, the stack sizes defined in the linker definition file, and the heap size if used.

In the example above, the variable stack segment (CSTACK) has been defined to 64 bytes (0x40). Add the 4 bytes from the compiled file and the sum will be 68 bytes of DATA memory as reported by the linker.



In the IAR Embedded Workbench, these figures and other information can be found in the Message window after compiling or linking if the **Message Filtering Level** has been set to **All** in the **Make Control** page of the **Settings** dialog box. For more information, see the *IAR Embedded Workbench™ IDE User Guide*.

EMBEDDED C ++ OVERVIEW

Embedded C ++ is a subset of the C ++ programming language, which is aimed at embedded systems programming. It is defined by an industry consortium, the Embedded C ++ Technical Committee. The fact that performance and portability are particularly important in embedded systems development was considered when defining the language.

Like full C ++ , the following extensions of the C programming language are provided:

- ◆ 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 there is a sufficient difference in their argument lists.
- ◆ Type-safe memory management using operators `new` and `delete`.
- ◆ Inline functions, which are indicated as particularly suitable for inline expansion.

Excluded features in C ++ are those, which introduce overheads in execution time or code size that are beyond the control of the programmer. Also excluded are recent additions to the ISO/ANSI C ++ standard. This is motivated by potential portability problems, due to the fact that few development tools support the standard. Embedded C ++ thus offers a subset of C ++ , which is efficient and fully supported by existent development tools.

Embedded C ++ lacks the following C ++ features:

- ◆ Templates
- ◆ Multiple inheritance
- ◆ Exception handling
- ◆ Run-time type information
- ◆ New cast syntax (operators `dynamic_cast`, `static_cast`, `reinterpret_cast`, and `const_cast`)
- ◆ Name spaces.

The excluded language features also make the run-time 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 using templates.

Library features, which relate to exception handling and run-time type information (headers `<except>`, `<stdexcept>` and `<typeinfo>`) are excluded.

LANGUAGE EXTENSIONS

This section summarizes the extensions provided in the M32C IAR C/C++ Compiler to support specific features of the M32C and M16C/8x Series CPU cores.

The extensions are provided in the following ways:

- ◆ As extended keywords. The command line option `-e` makes the extended keywords available, and hence reserves them so that they cannot be used as variable names.

For a complete description of the extended keywords, see the chapter *Extended keywords reference*.

- ◆ As `#pragma` keywords. These provide `#pragma` directives which control how the compiler allocates memory, whether the compiler allows extended keywords, and whether the compiler outputs warning messages.

For a complete description of the `#pragma` directives, see the chapter *#pragma directives reference*.

- ◆ As intrinsic functions. These provide direct access to very low-level processor details.

For a complete description of the intrinsic functions, see the chapter *Intrinsic functions reference*.

EXTENDED KEYWORDS AND PRAGMAS

The extended keywords and `#pragma` directives provide the following facilities:

Storage

The compiler places variables in the default segments for each memory model. The program may achieve additional flexibility for special cases by overriding the default address range by using one of the storage modifiers:

```
__near  
__far  
__huge
```

Pointers

The compiler uses a default data pointer for each memory model. It is possible to override the default size of the data pointers by using one of the following modifiers:

```
__near  
__far  
__huge
```

Non-initialized memory

To avoid initialization of variables, the keyword `__no_init` can be used. Use this keyword to reduce the amount of initialization code generated, or for data that will be placed in non-volatile RAM.

Variables may be placed in non-initialized memory by the use of the following modifier:

```
__no_init
```

Absolute variable location

It is possible to specify the location of a variable (its absolute address) by using the `@` operator followed by a constant-expression. See *Absolute location*, page 77, for more information.

The `#pragma location` directive is, however, recommended for specifying an absolute variable location. See *Location*, page 88, for more information.

Functions

The default calling mechanism for functions can be overridden in special cases by the use of one of the following function modifiers:

◆ `__interrupt`

Specifies interrupt functions. The `#pragma vector` directive can be used to specify the interrupt vector.

◆ `__regbank_interrupt`

Specifies interrupt functions where the interrupt routine uses the secondary register bank.

◆ `__fast_interrupt`

Specifies interrupt functions where the interrupt routine uses the fast interrupt mechanism and where the return is made by a `FREIT` instruction.

◆ `__monitor`

Specifies a monitor function, i.e. a function that cannot be interrupted.

◆ `__tiny_func`

Calls function with JSRS via a vector in the special page. The normal calls are made with an 24-bit address JSR.

◆ `__c_task`

Specifies that the functions should not restore used registers.

INTRINSIC FUNCTIONS

Intrinsic functions allow very low-level control of the M32C and M16C/8x Series CPU cores. To use them in a C/C++ application, include the header file `im32c.h`. The intrinsic functions compile into in-line code, either as a single instruction or as a short sequence of instructions.

For details concerning the effects of the intrinsic functions, see the manufacturer's documentation of the M32C and M16C/8x Series of CPU cores.

<i>Intrinsic function</i>	<i>Description</i>
<code>asm</code>	Inserts an assembler instruction.
<code>void __break_instruction (void)</code>	Inserts a BRK instruction.
<code>void __disable_interrupt (void)</code>	Disables interrupts.

<i>Intrinsic function</i>	<i>Description</i>
<code>void __intrinsic_load_DCT (unsigned short <i>dmaChannel</i>, unsigned short <i>data</i>)</code>	Places 16-bit data in DCT register.
<code>void __intrinsic_load_DMA (unsigned short <i>dmaChannel</i>, unsigned long <i>data</i>)</code>	Places 24-bit data in DMA register.
<code>void __intrinsic_load_DMD (unsigned short <i>dmaChannel</i>, unsigned short <i>data</i>)</code>	Places 16-bit data in DMD register.
<code>void __intrinsic_load_DRA (unsigned short <i>dmaChannel</i>, unsigned long <i>data</i>)</code>	Places 24-bit data in DRA register.
<code>void __intrinsic_load_DRC (unsigned short <i>dmaChannel</i>, unsigned short <i>data</i>)</code>	Places 16-bit data in DRC register.
<code>void __intrinsic_load_DSA (unsigned short <i>dmaChannel</i>, unsigned long <i>data</i>)</code>	Places 24-bit data in DSA register.
<code>void __intrinsic_load_VCT (unsigned long <i>data</i>)</code>	Places 24-bit data in VCT register.
<code>unsigned short __intrinsic_store_DCT (unsigned short <i>dmaChannel</i>)</code>	Retrieves 16-bit data from DCT register.
<code>unsigned long __intrinsic_store_DMA (unsigned short <i>dmaChannel</i>)</code>	Retrieves 24-bit data from DMA register.
<code>unsigned short __intrinsic_store_DMD (unsigned short <i>dmaChannel</i>)</code>	Retrieves 16-bit data from DMD registers
<code>unsigned long __intrinsic_store_DRA (unsigned short <i>dmaChannel</i>)</code>	Retrieves 24-bit data from DRA register.
<code>unsigned long __intrinsic_store_DRC (unsigned short <i>dmaChannel</i>)</code>	Retrieves 16-bit data from DRC register.
<code>unsigned long __intrinsic_store_DSA (unsigned short <i>dmaChannel</i>)</code>	Retrieves 24-bit data from DSA register.
<code>unsigned long __intrinsic_store_VCT (void)</code>	Retrieves 24-bit data from VCT register.
<code>void __enable_interrupt (void)</code>	Enables interrupts.
<code>void __interrupt_on_overflow (void)</code>	Inserts an INTO instruction.

<i>Intrinsic function</i>	<i>Description</i>
<code>void __no_operation (void)</code>	Inserts a no operation, NOP, instruction.
<code>short __overflow_flag_value (void)</code>	Reads the overflow flag value from the flag register.
<code>unsigned char __read_ip1 (void)</code>	Reads the interrupt level.
<code>long __rmpa_instruction (short *s1, short *s2, unsigned short n)</code>	Emits and RMPA.W instruction.
<code>void __set_interrupt_table (unsigned long)</code>	Loads interrupt base register (INTB).
<code>long __short_rmpa_instruction (signed char *s1, signed char *s2, unsigned short n)</code>	Emits and RMPA.B instruction.
<code>void __software_interrupt (unsigned char <i>int_no</i>)</code>	Causes a software interrupt.
<code>void __und_instruction (void)</code>	Inserts an UND instruction.
<code>void __wait_for_interrupt (void)</code>	Inserts a WAIT instruction.
<code>void __write_ip1 (unsigned char <i>val</i>)</code>	Sets the interrupt level.

For additional information, see the chapter *Intrinsic functions reference*.

Inline assembler

The `asm` function assembles and inserts the supplied assembler statement inline. The statement can include instruction mnemonics, register mnemonics, constants, and/or a reference to a global variable. Example:

```
asm("MOV.W #3,R0");
```

Note: The `asm` function reduces the compiler's ability to optimize the code. We recommend the use of assembler-written modules instead of inline assembler.

CONFIGURATION

This chapter describes how to configure the IAR C/C++ Compiler for different requirements.

INTRODUCTION

Systems based on the M32C and M16C/8x Series CPU cores can vary considerably in their use of ROM and RAM, and in their stack requirements. They also differ in their need for libraries. Therefore, you may need to configure the M32C IAR C/C++ Compiler to suit your requirements.

The options specify ROM areas, which are used for functions, constants, and initial values and RAM areas, which are used for stack and variables.

The configurable elements of the compiler package are described as follows:

<i>Feature</i>	<i>Configurable elements</i>	<i>See</i>
Processor option	Compiler option, XLINK, command file (including run-time library)	page 18
Floating-point precision	Compiler option, XLINK, command file (including run-time library)	page 18
Memory model	Compiler option, XLINK, option (including run-time library)	page 18
Memory location	XLINK command file	page 20
Non-volatile RAM	XLINK command file	page 21
Stack size	XLINK command file	page 24
input/output functions	Run-time library module	page 29
Hardware/memory initialization	<code>__low_level_init</code> module	page 33

The following sections describe each of the above features.

Note: Many of the configuration procedures involve modifying the standard files. It is recommended that you stores copies of the originals before beginning.

PROCESSOR

The M32C IAR C/EC + + Compiler supports both the M32C and M16C/8x Series of CPU cores. The processor option reflects the addressing capability of the target processor. When you select a particular processor option for your project, several target-specific parameters are tuned to best suit that processor.



Use either the `--cpu` or `-v` option to specify which CPU core you are using; see the chapter *Compiler options* for syntax information.



See the chapter *General options* in the *IAR Embedded Workbench™ IDE User Guide* for information about setting project options in the IAR Embedded Workbench.

Mapping of processor option and CPU core

The following table shows the mapping of `--cpu` and `-v` options and which processors they support:

<i>Processor option</i>	<i>Alternative option</i>	<i>Supported processor</i>
<code>--cpu=0</code>	<code>-v0</code>	M32C
<code>--cpu=1</code>	<code>-v1</code>	M16C/80

Your program may use only one processor option at a time, and the same processor option must be used by all user and library modules in order to maintain module consistency.

MEMORY MODEL

The M32C IAR C/C + + Compiler supports three memory models. The choice of memory model depends on the RAM-memory requirements of your application. The choice of memory model affects execution speed and code/data size, versus maximum size of program code and/or data.

SELECTING MEMORY MODEL

The memory model selected affects the data memory management. Data memory is used for:

- ◆ Non-stacked variables, i.e. global data and variables declared as static. From here on this will be referred to as static data.
- ◆ Stacked data; for example, locally declared data.
- ◆ Dynamically allocated data, e.g. data allocated with `malloc` and `calloc`.

The following table summarizes the different memory models:

<i>Memory model</i>	<i>Data</i>	<i>Largest data element</i>	<i>Comment</i>
Near	0–0xFFFF	64 KBytes	
Far	0–0xFFFFFFFF	64 KBytes	16-bit offsets only.
Huge	0–0xFFFFFFFF	16 MBytes	

Default memory

Variables are placed in the default memory, depending on the chosen memory model. The following default memories are used:

<i>Memory model</i>	<i>Default segment</i>	<i>Memory keyword</i>
Near	UDATA0, IDATA0, NDATA0	<code>__near</code>
Far	UDATA1, IDATA1, NDATA1	<code>__far</code>
Huge	UDATA2, IDATA2, NDATA2	<code>__huge</code>

For additional information, see *Allocating the writable segments and constants*, page 22.

All default behaviors originating from the selected memory model option can be altered by the use of extended keywords and `#pragma` directives.

Default pointers

The default code pointer is 4 bytes. Each memory model has its own default data pointer, `__near`, `__far`, and `__huge`, respectively.

Note: The entire stack and data objects without memory attributes defined must be linked at addresses that can be reached by the default pointer type.

SPECIFYING THE MEMORY MODEL

Your program may use only one memory model at a time, and the same model must be used by all user modules and all library modules. Use one of the following target options to specify the memory model to the compiler:

<i>IAR Embedded Workbench option</i>	<i>Command line option</i>
Near (default)	--memory_model=near -mn
Far	--memory_model=far -mf
Huge	--memory_model=huge -mh

For example, to compile `myprog.c` for the huge memory model, use the **Huge** option in the IAR Embedded Workbench or the command:

```
iccm32c myprog --memory_model=huge
```

If you do not include the memory model option, the compiler uses the near memory model by default.

Before linking a project, you must specify the IAR XLINK Linker™ options, including the library module to be used. The library module must correspond to the memory model you have selected; see *Linker command file*, page 21, and *Run-time library*, page 24.

MEMORY LOCATION

You must specify the address ranges in ROM and RAM memory in the linker command file. See *Linker command file*, page 21, for information about which linker command file template to use and how to modify it.

For information about how to specify the memory address ranges, see the contents of the linker command file template and the *IAR Linker and Library Tools Reference Guide*.

NON-VOLATILE RAM

The compiler supports the declaration of variables that are to reside in non-volatile RAM through the `__no_init` type modifier and the object attribute `#pragma`. The compiler places such variables in separate segments, depending on which memory keyword is used. These segments should be assigned to, for example, the address range of the non-volatile RAM of the hardware environment. The run-time system does not initialize variables located in these segments.

To assign the `__no_init` segment to the address of the non-volatile RAM, you need to modify the linker command file. For details of assigning a segment to a given address, see the *IAR Linker and Library Tools Reference Guide*.

LINKER COMMAND FILE

The M32C IAR C/EC++ Compiler is provided with one XLINK extended linker command file (`.xcl`) for each known derivative.

To create an XLINK extended linker command file (`.xcl`) for a particular project you should first copy the most suitable linker command file and use it as a template.

Then modify the file—as described within the file—to specify the details of the target system’s memory map.

SPECIFYING THE LINKER COMMAND FILE



In the IAR Embedded Workbench, you specify the linker command file in the **Include** options in the **XLINK** category. For more information about setting linker options, see the *IAR Linker and Library Tools Reference Guide*, or the *IAR Embedded Workbench™ IDE User Guide*.



In the command line version of the M32C IAR C/C++ Compiler, you use the XLINK option `-f` to specify the linker command file and run-time library, for example:

```
xlink filename(s) -f lnkm32cn cln.r48
```

MODIFYING THE LINKER COMMAND FILE

The only change you will normally have to make to the supplied linker command file is to suit the details of your target's memory map. However, for special applications you may want to change the assignment of segments to memory areas. For details of individual segments, see the chapter *Segment reference*.

The following section explains the contents of a linker command file. The example is based on the `lnkm32cn.xcl` file, which is a linker command file for the near memory model. This file can also be used as a template if you want to create your own linker command file.

Note: In the linker command file, all values are hexadecimal.

Defining the CPU

In the first section of the linker command file, we use the XLINK option `-c` to specify the processor:

```
-cm32c
```

Allocating the writable segments and constants

Next we allocate the writable segments to the RAM area:

IDATA denotes initialized data, UDATA denotes zero uninitialized data, and NDATA denotes uninitialized data.

```
-Z(NEAR)IDATA0,UDATA0,NDATA0=401
```

The size of the user stack segment, CSTACK, is specified to 512 bytes:

```
-Z(NEAR)CSTACK+200
```

Then we specify the interrupt stack and give it a size of 64 bytes:

```
-Z(NEAR)ISTACK+40
```

Next we specify the constants that are reachable for near pointers:

```
-Z(NEARCONST)CONST0
```

Then we specify the far and huge data segments in RAM, starting at address `0x10000`:

```
-Z(FAR)IDATA1,UDATA1,NDATA1=10000
```

```
-Z(HUGE)IDATA2,UDATA2,NDATA2
```


Then we specify the far and huge constant segments in ROM, starting at address 0x80000:

```
-Z(FARCONST)CDATA0,CDATA1,CONST1=80000  
-Z(HUGECONST)CDATA2,CONST2
```

The last segment to specify is the CODE segment:

```
-Z(HUGECODE)CODE
```

Declaring the interrupt handling segments

Now we shall declare the interrupt handling segment INTVEC:

```
-Z(HUGECONST)INTVEC=FF0000
```

Next we set up the special page vector table:

```
-Z(HUGECONST)FLIST=FFFE00-FFFFDB
```

Then we set up the fixed interrupt table:

```
-Z(HUGECONST)INTVEC1=FFFFDC-FFFFF
```

Specifying the input and output formatters

Now we shall specify the formatters for the input and output functions.

First we select which `printf` and `sprintf` formatter to use. Here we use the default formatter called `_small_write`:

```
-e_small_write=_formatted_write
```

See *I/O functions*, page 29, for more information.

Next we select which `scanf` and `sscanf` formatter to use. Again we use the default formatter, which is called `_medium_read`.

```
-e_medium_read=_formatted_read
```

See *I/O functions*, page 29, for more information.

This completes the linker command file.

RUN-TIME LIBRARY

The following library modules are supplied with the product:

C LIBRARIES:

<i>Memory model</i>	<i>Floating-point option</i>	<i>Library file</i>
Near	32-bit doubles	cln.r48
Near	64-bit doubles	clnd.r48
Far	32-bit doubles	clf.r48
Far	64-bit doubles	clfd.r48
Huge	32-bit doubles	clh.r48
Huge	64-bit doubles	clhd.r48

EMBEDDED C+ + LIBRARIES:

<i>Memory model</i>	<i>Floating-point option</i>	<i>Library file</i>
Near	32-bit doubles	dln.r48
Near	64-bit doubles	dln.d.r48
Far	32-bit doubles	dlf.r48
Far	64-bit doubles	dlfd.r48
Huge	32-bit doubles	d1h.r48
Huge	64-bit doubles	d1hd.r48



Use the command line option - 2 to select 64-bit floating-point doubles.

Note: The run-time options for the selected library must be the same as the run-time options used when compiling the user-written modules.

STACK AND HEAP SIZE

The compiler uses the stack 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 small, the stack will normally overwrite the variable storage which is likely to result in program failure. If the given stack size is too large, RAM will be wasted.

ESTIMATING THE REQUIRED DATA STACK SIZE

The stack is used for the following:

- ◆ Storing local variables and parameters.
- ◆ Storing temporary results in expressions.
- ◆ Saving function return addresses.
- ◆ Storing temporary values in run-time library routines.
- ◆ Saving processor state during interrupts.

The total required stack size is the worst case total of the required sizes for each of the above plus the sum of all concurrently active interrupt functions.

CHANGING THE STACK SIZE

The default user stack size is set to 512 (0x200) bytes in the linker command file, with the expression `CSTACK+200` in the linker command:

```
-Z(NEAR)CSTACK+200,HEAP+200
```

There is also an interrupt stack segment, `ISTACK`, of 64 (0x40) bytes, defined by:

```
-Z(NEAR)ISTACK+40
```

To change either stack size, edit the linker command file and replace the current size by the stack size you want to use.

INPUT AND OUTPUT USING THE IAR C LIBRARY

The I/O functions are different depending on which set of libraries you are using. This information holds true for the IAR C Library.

PUTCHAR AND GETCHAR

The functions `putchar` and `getchar` are the fundamental functions through which C performs all 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 upon the files `putchar.c`, which serves as the low-level part of the `printf` function, and `getchar.c`.

The low-level I/O function `getchar` is supplied in the C file `getchar.c`, which can be customized using the method described for `putchar`.

The following section describes the procedure for replacing the original C library with one containing a customized `putchar`.

Customizing `putchar`

First make the required additions to the source `putchar.c`, and save it under the same name (or create your own routine using `putchar.c` as a model). The 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 DEV_IO @ address;

int putchar(int outchar)
{
    DEV_IO = outchar;
    return ( outchar );
}
```

Note: The exact address depends on the used derivative.

Creating a library module

Then compile the modified `putchar` using the appropriate processor option and the **Make library module** (`--library_module`) option.

For example, if your program uses the near memory model, compile `putchar.c` from the command line with the command:

```
iccm32c putchar --library_module
```

This will create an optimized replacement object module file named `putchar.r48`.

Testing the modified `putchar`

XLINK allows you to test the modified module before installing it in the library by using the **Load as PROGRAM** (`-A`) XLINK option. Place the following lines into your linker command file before the library reference:

```
-A putchar
```

This causes your version of `putchar.r48` to load instead of the one in the library. For information about the XLINK options, see the *M32C IAR Assembler Reference Guide*.

Modifying the C library

Finally add the new putchar module to the appropriate run-time library module, replacing the original.

Note: Be sure to save your original library file before you overwrite the putchar module.

For example, to add the new putchar module to the original library for the near memory model, use the command:

```
xlib
def-cpu m32c
rep-mod putchar cln
exit
```

The library module cln will now have the modified putchar instead of the original one.

Note: def-cpu and rep-mod are abbreviations of the XLIB commands DEFINE-CPU and REPLACE-MODULES. Notice also that in XLIB, module names are case sensitive. For additional information about the IAR XLIB Librarian, see the *M32C IAR Assembler Reference Guide*.

PRINTF AND SPRINTF

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

◆ `_medium_write`

As for `_formatted_write`, except that floating-point numbers are not supported. Any attempt to use a `%f`, `%g`, `%G`, `%e`, and `%E` specifier will produce the error:

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

`_medium_write` is considerably smaller than `_formatted_write`.

◆ `_small_write`

As for `_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 % of the size of `_formatted_write`.

The default version is `_small_write`.

SELECTING THE WRITE FORMATTER VERSION

The selection of a write formatter is made in the XLINK control file. The default formatter, `_small_write`, is selected with the line:

```
-e_small_write=_formatted_write
```

To select the full ANSI version, remove this line.

To select `_medium_write`, replace this line with:

```
-e_medium_write=_formatted_write
```

REDUCED PRINTF

For many applications `sprintf` is not required, and even `printf` with `_small_write` provides more facilities than are justified by the memory consumed. Alternatively, a custom output routine may be required to support particular formatting needs and/or non-standard output devices.

For such applications, a highly reduced version of the entire `printf` function (without `sprintf`) is supplied in source form in the file `intwri.c`. This file can be modified to your requirements and the compiled module inserted into the library in place of the original using the procedure described for `putchar`; for additional information, see *Modifying the C library*, page 27.

SCANF AND SSCANF

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

medium_read

As for `_formatted_read`, except that floating-point numbers are not supported. `_medium_read` is considerably smaller than `_formatted_read`.

The default version is `_medium_read`.

SELECTING READ FORMATTER VERSION

The selection of a read formatter is made in the XLINK control file. The default version, `_medium_read`, is selected with the line:

```
-e_medium_read=_formatted_read
```

To select the full ANSI version, remove this line.

INPUT AND OUTPUT USING THE EC++ LIBRARY

The standard library contains a large number of powerful functions for I/O operations. In order to simplify adaption to specific hardware, all I/O functions call a small set of primitive functions, each designed to accomplish one particular task; for example, `__open` acts as if it opens a file and `__write` outputs a number of characters.

The primitive I/O files are located in the `m32c\src\lib` directory.

<i>I/O function</i>	<i>File</i>	<i>Description</i>
<code>__open()</code>	<code>open.c</code>	Open a file.
<code>__close()</code>	<code>close.c</code>	Close a file.
<code>__read()</code>	<code>read.c</code>	Read a character buffer.
<code>__readchar()</code>	<code>readchar.c</code>	Read a character.
<code>__write()</code>	<code>write.c</code>	Write a character buffer.
<code>__writechar()</code>	<code>writchar.c</code>	Write a character.
<code>__lseek()</code>	<code>lseek.c</code>	Set the file position indicator.
<code>remove()</code>	<code>remove.c</code>	Remove a file.
<code>rename()</code>	<code>rename.c</code>	Rename a file.

I/O FUNCTIONS

The primitive I/O functions are the fundamental functions through which 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.

The creation of new I/O routines is based upon the files listed above.

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 file descriptors 0, 1, and 2, respectively.

The default implementation of the primitive functions maps the I/O streams associated with `stdin` and `stdout` to the debugger; all other operations are ignored.



Customizing a primitive I/O function on the command line

In most cases you can use the primitive I/O functions provided with the product. The following section describes how to modify a primitive function in case your application requires it. The example is based on `__writechar()` but applies also to the other primitive I/O functions.

Notice that `__writechar` serves as the low-level part of the `printf` function.

- 1 Copy the file `writechar.c`, which is provided in the `m32c\src\lib` directory, to your project directory.
- 2 Make the required additions to your copy of `writechar.c`, and save it under the same name. The code in the following example uses memory-mapped I/O to write to an LCD display:

```
#include <stdio.h>
#include <yfuncs.h>

_STD_BEGIN

int __writechar(int handle, unsigned char ch)
{
    unsigned char * LCD_IO;

    LCD_IO = (unsigned char *) 0x03E0;
    // Port P0 register
    *LCD_IO = ch;
    // ch on success, -1 on failure.
    return ch;
}

_STD_END
```


- 3 Compile the modified `writchar.c` using the appropriate processor and memory model options.

For example, if your program uses the far memory model and the M32C CPU core, compile `writchar.c` from the command line with the command:

```
iccm32c writchar -mf --cpu=0 --library_module
--module_name ?writchar -Ic:\program files\iar
systems\embedded workbench 3\m32c\inc\
```

Note: The name of each module in the standard library always begins with `?` in order to avoid name collision with user modules.

This will create an optimized replacement object module file named `writchar.r48`.

- 4 Add the following to your XLINK command line:

```
-A writchar
```

- 5 Link your code using the modified linker command file.



Customizing a primitive I/O function in the IAR Embedded Workbench

In most cases you can use the primitive I/O functions provided with the product. The following section describes how to modify a primitive function in case your application requires it. The example is based on `__writchar()` but applies also to the other primitive I/O functions.

Notice that `__writchar` serves as the low-level part of the `printf` function.

- 1 Copy the file `writchar.c`, which is provided in the `m32c\src\lib` directory, to your project directory.
- 2 Make the required additions to your copy of `writchar.c`, and save it under the same name. The code in the following example uses memory-mapped I/O to write to an LCD display:

```
#include <stdio.h>
#include <yfuncs.h>
```

```
_STD_BEGIN
```

```
int __writchar(int handle, unsigned char ch)
{
```

```

    unsigned char * LCD_IO;

    LCD_IO = (unsigned char *) 0x03E0;
    // Port P0 register
    *LCD_IO = ch;
    // ch on success, -1 on failure.
    return ch;
}

_STD_END

```

3 Add the modified `writchar` to your project.

4 Compile the modified `writchar.c` using the same processor configuration and memory model options as for the project.

This will create an optimized replacement object module file named `writchar.r48`.

5 Rebuild the project.

Maintaining library files



The IAR XLIB Librarian command `REPLACE-MODULES` allows you to permanently replace the original `CSTARTUP` with your customized version. See the *IAR Linker and Library Tools Reference Guide* for detailed information.

REGISTER I/O

A program may access the M32C and M16C/8x Series I/O system using the memory-mapped internal special function registers (SFRs).

All operators that apply to integral types may be applied to SFR registers. Predefined declarations for the M32C and M16C/8x Series are supplied with the product; see *Run-time library*, page 24.

INITIALIZATION

On processor reset, execution passes to a run-time system routine called `CSTARTUP`, which normally performs the following:

- ◆ Initializes the user and interrupt stack pointers.
- ◆ Initializes C file-level and static variables.
- ◆ Initializes the `INTB` register.

- ◆ Calls the start of the user program in `main()`.

CSTARTUP is also responsible for receiving and retaining control if the user program exits, whether through `exit` or `abort`.

VARIABLE AND I/O INITIALIZATION

In some applications you may want to initialize I/O registers, 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 before the data segments are initialized.

The value returned by `__low_level_init` determines whether data segments are initialized. The run-time library includes a dummy version of `__low_level_init` that simply returns 1, to cause CSTARTUP to initialize data segments.

The source code for `__low_level_init` is provided in the file `lowinit.c`, which is provided with the product. To perform your own I/O initializations, create a version of this routine containing the necessary code to make the initializations. If you also want to disable the initialization of data segments, make the routine return a zero. Compile the customized routine and link it with the rest of your code.

MODIFYING CSTARTUP

If you want to modify CSTARTUP you will need to reassemble CSTARTUP with options that match your selected compiler options.

The overall procedure for assembling a modified copy of CSTARTUP is as follows:

- ◆ Make any required modifications to the assembler source of CSTARTUP, supplied by default in the file `cstartup.s48` and save it under the same name.
- ◆ Assemble CSTARTUP using options that match your selected compiler options.

This will create an object module file named `cstartup.r48`.

You should then use the following commands in the linker command file to make XLINK use the CSTARTUP module that you have defined instead of the one in *library*:

```
-A cstartup      Load as program module
-C library      Load as library module
```

The modified `cstartup` will be used instead of the default one in the library. This is done with the options **Load as LIBRARY** (-C) and **Load as PROGRAM** (-A).

The XLINK options are described in the *IAR Linker and Library Tools Reference Guide*.



In the IAR Embedded Workbench, add the modified `cstartup` file to your project, and add -C before the library in the linker command file.

INTERRUPT SYSTEM

INTERRUPT KEYWORDS

The keywords `__interrupt`, `__fast_interrupt`, and `__regbank_interrupt` are available for interrupt functions.

For a complete description of the interrupt keywords, see *Functions*, page 79.

INTERRUPT VECTOR TABLE

The compiler will automatically create an interrupt vector table where it will enter the addresses of the defined interrupt functions. The vector table will be located in the INTVEC segment. It will be allocated to address 0xFF0000 by the supplied linker command files with the following command:

```
-Z(HUGECONST)INTVEC=FF0000
```

For the moment the only way to enter default values in the vector table is to edit the `cstartup` routine and fill in default values at all vector addresses not specified by the compiler (or assembler). There are comments in the `cstartup` file to show the user how to proceed.

INTERRUPT VECTOR TABLE OFFSET

The `#pragma vector` will be used in the C code to define the byte offset into the interrupt vector table where the `__interrupt` functions address will be stored, as in the following example:

```
#pragma vector=20
__interrupt void wTimer_ISR (void)
{
    C code ...
}
```

The `#pragma` causes the compiler to store the address of the `wTimer_ISR` function at byte offset 20 in the interrupt vector table located in the `INTVEC` segment. The `__interrupt` keyword will also cause the compiler to end the `Timer2B_ISR` function with a `REIT` instruction.

COMPILER OPTIONS

This chapter explains how to set the compiler options from the command line, and gives detailed reference information about each option.



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

SETTING COMPILER OPTIONS

To set compiler options from the command line, include them on the command line after the `icc32c` command, either before or after the source filename. For example, when compiling the source `prog.c`, use the following command to generate an object file with debug information:

```
icc32c prog --debug
```

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

```
icc32c prog -l list.lst
```

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

```
icc32c prog -DDEBUG=1
```

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

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

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

Specifying parameters

When a parameter is needed for an option with a short name, it can be specified either immediately following the option or as the next command line argument. For instance, an include file path of `\usr\include` can be specified either as:

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

or as

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

Note: `/` can be used instead of `\` as directory delimiter.

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

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

```
--diag_suppress=Pe0001
```

or

```
--diag_suppress Pe0001
```

The option `--preprocess` is, however, an exception as the filename must be preceded by space. In the following example comments are included in the preprocessor output:

```
--preprocess=c prog
```

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

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

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

```
iccm32c prog -l .
```

A file specified by `' - '` is standard input or output, whichever is appropriate.

Note: When an option takes a parameter, the parameter cannot start with a dash (`-`) followed by another character. Instead you can prefix the parameter with two dashes; the following example will create a list file called `-r`:

```
iccm32c prog -l ---r
```


Error return codes

The M32C IAR C/EC + + Compiler returns status information to the operating system which can be tested in a batch file.

The following command line error codes are supported:

<i>Code</i>	<i>Description</i>
0	Compilation successful, but there may have been warnings.
1	There were warnings, provided that the option <code>--warnings_affect_exit_code</code> was used.
2	There were non-fatal errors.
3	There were fatal errors (compiler aborted).

**ENVIRONMENT
VARIABLES**

Compiler options can also be specified in the QCCM32C environment variable. The compiler automatically appends the value of this variable to every command line, so it provides a convenient method of specifying options that are required for every compilation.

The following environment variables can be used with the M32C IAR C/EC + + Compiler:

<i>Environment variable</i>	<i>Description</i>
C_INCLUDE	Specifies directories to search for include files; for example: <code>C_INCLUDE=c:\iar\m32c\inc;c:\headers</code>
QCCM32C	Specifies command line options; for example: <code>QCCM32C=-lA asm.lst -z9</code>

See the *M32C IAR Assembler Reference Guide* for information about the environment variables that can be used by the M32C Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™.

OPTIONS SUMMARY	
The following table summarizes the compiler command line options:	
<i>Command line option</i>	<i>Description</i>
--char_is_signed	'char' is 'signed char'
--code_segment <i>name</i>	Code segment
--cpu= <i>cpu</i>	Processor variant
-D <i>symb</i> [= <i>value</i>]	Defines preprocessor symbols
--debug	Generates debug info
--diag_error= <i>tag, tag, ...</i>	Treats these as errors
--diag_remark= <i>tag, tag, ...</i>	Treats these as remarks
--diag_suppress= <i>tag, tag, ...</i>	Suppresses these diagnostics
--diag_warning= <i>tag, tag, ...</i>	Treats these as warnings
-e	Enables language extensions
--ec++	Enables Embedded C ++ syntax
-I <i>path</i>	Includes file path
-l[c C a A][N] <i>filename</i>	Creates list file
--library_module	Makes library module
-m[n f h]	Memory model
--memory_model=[near far huge]	Memory model
--module_name= <i>name</i>	Sets object module name
--no_code_motion	Disables code motion optimization
--no_cse	Disables common sub-expression elimination
--no_inline	Disables function inlining
--no_unroll	Disables loop unrolling
--no_warnings	Disables all warnings

<i>Command line option</i>	<i>Description</i>
<code>-o filename</code>	Sets object filename
<code>--only_stdout</code>	Uses standard output only
<code>--preprocess=[c][n][l] filename</code>	Preprocessor output to file
<code>-R name</code>	Code segment
<code>-r</code>	Generates debug information
<code>--remarks</code>	Enables remarks
<code>--require_prototypes</code>	Forces verification of function prototypes
<code>-s[3 6 9]</code>	Optimizes for speed
<code>--silent</code>	Sets silent operation
<code>--strict_ansi</code>	Enables strict ISO/ANSI
<code>-Usymb</code>	-
<code>-v[0 1]</code>	Processor variant
<code>--warnings_affect_exit_code</code>	Warnings affects exit code
<code>--warnings_are_errors</code>	Treats all warnings as errors
<code>-z[3 6 9]</code>	Optimizes for size
<code>-2</code>	64-bit floating point

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

--char_is_signed

‘char’ is ‘signed char’.

SYNTAX

`--char_is_signed`

DESCRIPTION

By default the compiler interprets the `char` type as `unsigned char`. Use this option to make the compiler interpret the `char` type as `signed char` instead, for example for compatibility with another compiler.

Note: The run-time library is compiled without the `--char_is_signed` option. If you use this option, you may get type mismatch warnings from the linker since the library uses `unsigned chars`.

Use this option to make the `char` type equivalent to `signed char`.



This option corresponds to the '**char**' is '**signed char**' option in the **ICCM32C** category in the IAR Embedded Workbench.

--code_segment, -R

Places executable code in a named segment.

DESCRIPTION

Normally, the compiler places executable code in the segment named `CODE`. If you want to be able to specify an explicit location for the code, use the `--code_segment` option to specify a code segment name, which you can then assign to a fixed address in the linker command file.

SYNTAX

Syntax: `--code_segment name`

--cpu

Processor variant.

SYNTAX

`--cpu=cpu`

DESCRIPTION

Use this option to select the processor for which the code is to be generated.

For example, use the following command to specify the M32C CPU core:

`--cpu=0`

See *Processor*, page 18, for a summary of the available processors.

Notice that to specify the processor, you can use either the `--cpu` option or the `-v` option. The `--cpu` option is, however, more precise since implicit assumptions are made about the processor when you use the `-v` option. For additional information, see page 18.



This option is related to the **Processor configuration** option in the **General** category in the IAR Embedded Workbench.

-D

Defines preprocessor symbols.

SYNTAX

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

DESCRIPTION

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

The option `-D` has the same effect as a `#define` statement at the top of the source file.

```
-D symb
```

is equivalent to:

```
#define symb
```

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

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

Then, you would select the version required on the command line as follows:

```
Production version: iccm32c prog  
Test version:      iccm32c prog -Dtestver
```

This option can be used one or more times.



This option is related to the **Preprocessor** options in the **ICCM32C** category in the IAR Embedded Workbench.

--debug, -r

Generates debug information.

SYNTAX

--debug
-r

DESCRIPTION

This option causes the compiler to include additional information required by C-SPY™ and other symbolic debuggers in the object modules.

Note: Including debug information will make the object files become larger than otherwise.



This option is related to the **Output** options in the **ICCM32C** category in the IAR Embedded Workbench.

--diag_error

Treats the specified diagnostic messages as errors.

SYNTAX

--diag_error=tag,tag,...

DESCRIPTION

An error indicates a violation of the C language rules, of such severity that object code will not be generated, and the exit code will not be 0. Use this option to classify diagnostic messages as errors.

The following example classifies warning Pe117 as an error:

--diag_error=Pe117



This option is related to the **Diagnostics** options in the **ICCM32C** category in the IAR Embedded Workbench.

--diag_remark

Treats the specified diagnostic messages as remarks.

SYNTAX

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

DESCRIPTION

A remark is the least severe type of diagnostic message and indicates a source code construct that may cause strange behavior in the generated code. Use this option to classify diagnostic messages as remarks.

The following example classifies the warning Pe177 as a remark:

--diag_remark=Pe177



This option is related to the **Diagnostics** options in the **ICCM32C** category in the IAR Embedded Workbench.

--diag_suppress

Suppresses the specified diagnostics messages.

SYNTAX

--diag_suppress=*tag, tag, ...*

DESCRIPTION

Suppresses the output of diagnostics for the specified tags.

Use this option to suppress diagnostic messages. The following example suppresses the warnings Pe117 and Pe177:

--diag_suppress=Pe117,Pe177



This option is related to the **Diagnostics** options in the **ICCM32C** category in the IAR Embedded Workbench.

--diag_warning

Treats the specified diagnostic messages as warnings.

SYNTAX

--diag_warning=*tag, tag, ...*

DESCRIPTION

A warning indicates an error or omission that is of concern, but which will not cause the compiler to stop before compilation is completed. Use this option to classify diagnostic messages as warnings.

The following example classifies the remark Pe826 as a warning:

```
--diag_warning=Pe826
```



This option is related to the **Diagnostics** options in the **ICCM32C** category in the IAR Embedded Workbench.

-e

Enables language extensions.

SYNTAX

-e

DESCRIPTION

Language extensions must be enabled for the M32C IAR Compiler to be able to accept M32C-specific keywords as extensions to the standard C language.

In the command line version of the M32C IAR Compiler, language extensions are disabled by default. Use the command line option -e to enable language extensions such as keywords and anonymous structs and unions.

Note: The -e option and the --strict_ansi option cannot be used at the same time.

For additional information, see *Language extensions*, page 11.



This option is related to the **Language** options in the **ICCM32C** category in the IAR Embedded Workbench.

--ec + +

Enables the Embedded C + + syntax.

SYNTAX

--ec++

DESCRIPTION

In the command line version of the M32C IAR Compiler, Embedded C++ syntax is disabled by default. If you are using Embedded C++ syntax in your source code, you must enable it by using this option.

The `--ec++` option is only available in the EC++ version of the product.



This option is related to the **Language** options in the **ICCM32C** category in the IAR Embedded Workbench.

-I

Specifies `#include` file paths.

SYNTAX

`-Ipath`

DESCRIPTION

Adds a path to the list of `#include` file paths, for example:

```
iccm32c prog -I\mylib1
```

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

This option may be used more than once on a single command line.

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

- ◆ If the name of the `#include` file is an absolute path, that file is opened.
- ◆ When the compiler encounters the name of an `#include` file in angle brackets such as:

```
#include <stdio.h>
```

it searches the following directories for the file to include:
 1. The directories specified with the `-I` option, in the order that they were specified.
 2. The directories specified using the `C_INCLUDE` environment variable, if any.
- ◆ When the compiler encounters the name of an `#include` file in double quotes such as:

```
#include "vars.h"
```

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

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

```
src.c in directory dir
#include "src.h"
...
src.h in directory dir\h
#include "io.h"
...
```

When `dir\exe` is the current directory, use the following command for compilation:

```
iccm32c ..\src.c -I..\dir\include
```

Then the following directories are searched for the `io.h` file, in the following order:

<code>dir\h</code>	Current file.
<code>dir</code>	File including current file.
<code>dir\include</code>	As specified with the <code>-I</code> option.



This option is related to the **Preprocessor** options in the **ICCM32C** category in the IAR Embedded Workbench.

Generates a listing to the specified filename.

SYNTAX

```
-l[c|C|a|A][N] filename
```

DESCRIPTION

Generates a listing to the named file with the default extension `lst`.

Normally, the compiler does not generate a listing. To generate a listing to a named file, you use the `-l` option. For example, to generate a listing to the file `list.lst`, use:

```
iccm32c prog -l list
```

The following modifiers are available:

<i>Option modifier</i>	<i>Description</i>
a	Assembler file
A (N is implied)	Assembler file with C source as comments
c	C list file
C (default)	C list file with assembler source as comments
N	No diagnostics in file



This option is related to the **List** options in the **ICCM32C** category in the IAR Embedded Workbench.

--library_module

Makes module a library module.

SYNTAX

```
--library_module
```

DESCRIPTION

A program module is always included during linking. Use this option to make a library module that will only be included if it is referenced in your program.

Use the `--library_module` option to make the object file be treated as a library module rather than as a program module.



This option is related to the **Output** options in the **ICCM32C** category in the IAR Embedded Workbench.

-m, --memory_model

Specifies the data memory model.

SYNTAX

```
-m[n|f|h]
```

```
--memory_model=[near|far|huge]
```

DESCRIPTION

Specifies the memory model for which the code is to be generated.

By default the compiler generates code for the near memory model. Use the `-m` or the `--memory_model` option if you want to generate code for a different memory model.

For example, to generate code for the far memory model, give the command:

```
iccm32c filename -mf
```

or:

```
iccm32c filename --memory_model=far
```



These options are related to the **Memory model** option in the **General** category in the IAR Embedded Workbench.

--module_name

Sets the object module name.

SYNTAX

```
--module_name=name
```

DESCRIPTION

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.

To set the object module name explicitly, use the option

`--module_name=name`, for example:

```
iccm32c prog --module_name=main
```

This option is particularly useful when several modules have the same filename, since 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.

The following example—in which %1 is an operating system variable containing the name of the source file—will give duplicate name errors from the linker:

```
preproc %1.c temp.c                ; preprocess source,
                                     ; generating temp.c
iccm32c temp.c                      ; module name is
                                     ; always 'temp'
```

To avoid this, use --module_name=*name* to retain the original name:

```
preproc %1.c temp.c                ; preprocess source,
                                     ; generating temp.c
iccm32c temp.c --module_name=%1    ; use original source
                                     ; name as module name
```

Note: In the above example, preproc is an external utility.



This option is related to the **Output** options in the **ICCM32C** category in the IAR Embedded Workbench.

--no_code_motion

Disables the code motion optimization.

SYNTAX

```
--no_code_motion
```

DESCRIPTION

Evaluation of loop-invariant expressions and common sub-expressions are moved to avoid redundant re-evaluation. This optimization, which is performed at optimization levels 4–9, normally reduces code size and execution time. The resulting code may however be difficult to debug.

Use --no_code_motion to disable code motion.

Note: This option has no effect at optimization levels 0–3.



This option is related to the **Optimization** options in the **ICCM32C** category in the IAR Embedded Workbench.

--no_cse

Disables the common sub-expression elimination.

SYNTAX

`--no_cse`

DESCRIPTION

Use `--no_cse` to disable common sub-expression elimination.

Redundant re-evaluation of common sub-expressions is by default eliminated at optimization levels 4–9. This optimization normally reduces both code size and execution time. The resulting code may however be difficult to debug.

Note: This option has no effect at optimization levels 0–3.



This option is related to the **Optimization** options in the **ICCM32C** category in the IAR Embedded Workbench.

--no_inline

Disables function inlining.

SYNTAX

`--no_inline`

DESCRIPTION

Use `--no_inline` 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 levels 7–9, normally reduces execution time, but increases code size. The resulting code may also be difficult to debug. In certain cases, the code size will decrease when this option is used.

The compiler heuristically decides which functions to inline. Different heuristics are used when optimizing for speed.

Note: This option has no effect at optimization levels 0–6.



This option is related to the **Optimization** options in the **ICCM32C** category in the IAR Embedded Workbench.

--no_unroll

Disables loop unrolling.

SYNTAX

--no_unroll

DESCRIPTION

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, which is performed at optimization levels 7–9, normally reduces execution time, but increases code size. The resulting code may also be difficult to debug.

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed and size. This option has no effect at optimization levels 0–6.

Note: Loop unrolling is permanently disabled in the M32C IAR C/EC + + Compiler. This option is available for compatibility with other IAR compilers.



This option is related to the **Optimization** options in the **ICCM32C** category in the IAR Embedded Workbench.

--no_warnings

Disables all warnings.

SYNTAX

--no_warnings

DESCRIPTION

Normally, the compiler issues standard warning messages. To disable all warning messages, use the --no_warnings option.



This option is related to the **Diagnostics** options in the **ICCM32C** category in the IAR Embedded Workbench.

-o

Sets object filename.

SYNTAX

`-o filename`

DESCRIPTION

If no object code filename is specified, the compiler stores the object code in a file whose name consists of the source filename, excluding the path, plus the filename extension `.r48`.

Use the `-o` option to specify a name for the output file. The filename may include a pathname. For example, to store it in the file `obj.r48` in the `mypath` directory, you would use:

```
iccm32c prog -o \mypath\obj
```

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



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

--only_stdout

Uses standard output only.

SYNTAX

`--only_stdout`

DESCRIPTION

Causes the compiler to use `stdout` also for messages that are normally directed to `stderr`.

--preprocess

Directs preprocessor output to file.

SYNTAX

`--preprocess=[c][n][l] filename`

DESCRIPTION

Use this option to generate preprocessor output to the named file, *filename.i*.

The filename consists of the filename itself, optionally preceded by a pathname and optionally followed by an extension. If no extension is given, the extension `i` is used. In the syntax description above, note that space is allowed in front of the filename.

The following table shows the mapping of the available preprocessor modifiers:

<i>Command line option</i>	<i>Description</i>
--preprocess=c	Preserve comments
--preprocess=n	Preprocess only
--preprocess=l	Generate <code>#line</code> directives



This option is related to the **Preprocessor** options in the **ICCM32C** category in the IAR Embedded Workbench.

--R, --code_segment

Places executable code in a named segment.

DESCRIPTION

Normally, the compiler places executable code in the segment named `CODE`. If you want to be able to specify an explicit location for the code, use the `--R` option to specify a code segment name, which you can then assign to a fixed address in the linker command file.

SYNTAX

Syntax: `--R name`

-r, --debug

Generates debug information.

SYNTAX

`--debug`
`-r`

DESCRIPTION

This option causes the compiler to include additional information required by C-SPY™ and other symbolic debuggers in the object modules.

Note: Including debug information will make the object files become larger than otherwise.



This option is related to the **Output** options in the **ICCM32C** category in the IAR Embedded Workbench.

--remarks

Enables remarks.

SYNTAX

--remarks

DESCRIPTION

The least severe diagnostic messages are called remarks (see *Severity levels*, page 157). A remark indicates a source code construct that may cause strange behavior in the generated code.

By default remarks are not generated. Use --remarks to make the compiler generate remarks.



This option is related to the **Diagnostics** options in the **ICCM32C** category in the IAR Embedded Workbench.

--require_prototypes

Forces verification of function prototypes

SYNTAX

--require_prototypes

DESCRIPTION

This option forces the compiler to verify that all functions have proper prototypes. Using this option means that code containing either of the following will generate an error:

- ◆ A function call of a function with no declaration or 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.

-s

Optimizes for speed.

SYNTAX`-s[3|6|9]`**DESCRIPTION**

Causes the compiler to optimize the code for maximum execution speed.

If no optimization option is specified `-z3` is used by default. If the `-s` or the `-z` option is used without specifying the optimization level, level 3 is used by default.

Note: The `-s` and `-z` options cannot be used at the same time.

The following table shows how the optimization levels are mapped:

<i>Option modifier</i>	<i>Description</i>
3	Fully debuggable
6	Heavy optimization can make the program flow hard to follow during debug
9	Full optimization



This option is related to the **Optimization** options in the **ICCM32C** category in the IAR Embedded Workbench.

--silent

Specifies silent operation.

SYNTAX`--silent`**DESCRIPTION**

By default the compiler issues introductory messages and a final statistics report. Use `--silent` to make the compiler operate without sending unessential messages to standard output (normally the screen). This does not affect the display of error and warning messages.

--strict_ansi

Specifies strict ISO/ANSI.

SYNTAX

`--strict_ansi`

DESCRIPTION

By default the compiler accepts a superset of ISO/ANSI C (see the chapter *IAR C extensions*). Use `--strict_ansi` to ensure that the program conforms to the ISO/ANSI C standard.

Note: The `-e` option and the `--strict_ansi` option cannot be used at the same time.



This option is related to the **Language** options in the **ICCM32C** category in the IAR Embedded Workbench.

-U

Removes the definition of the named symbol.

SYNTAX

Syntax: `-U symp`
 `-U symp`

DESCRIPTION

Normally, the compiler provides various predefined symbols. If you want to remove one of these, for example to avoid a conflict with your own symbol with the same name, use the undefine symbol (`-U`) option.

For example, to remove the symbol `__VER__`, use:

```
iccm32c prog -U __VER__
```

For a list of the predefined symbols, see the chapter *Predefined symbols reference*.

-v

Specifies the processor variant.

SYNTAX`-v[0|1]`**DESCRIPTION**

Use this option to select the processor for which the code is to be generated. The following processors are available:

<i>Command line option</i>	<i>Processor</i>
<code>-v0</code>	M32C
<code>-v1</code>	M16C/80

See also `--cpu`, page 42, and *Processor*, page 18.



This option is related to the **Processor configuration** option in the **General** category in the IAR Embedded Workbench.

--warnings_affect_exit_code

Makes warnings affect the exit code.

SYNTAX`--warnings_affect_exit_code`**DESCRIPTION**

By default the exit code is not affected by warnings, only errors produce a non-zero exit code. With this option, warnings will generate a non-zero exit code.



This option is related to the **Diagnostics** options in the **ICCM32C** category in the IAR Embedded Workbench.

--warnings_are_errors

Makes the compiler treat all warnings as errors.

SYNTAX`--warnings_are_errors`

DESCRIPTION

Use this option to make the compiler treat all warnings as errors. If the compiler encounters an error, no object code is generated.

If you want to keep some warnings, you can use this option in combination with the option `--diag_warning`. First make all warnings become treated as errors and then reset the ones that should still be treated as warnings, for example:

```
--diag_warning=Pe117
```

For additional information, see *--diag_warning*, page 45.



This option is related to the **Diagnostics** options in the **ICCM32C** category in the IAR Embedded Workbench.

-Z

Optimizes for size.

SYNTAX

```
-z[3|6|9]
```

DESCRIPTION

Causes the compiler to optimize the code for minimum size. If no optimization option is specified `-z3` is used by default. If the `-s` or the `-z` option is used without specifying the optimization level, level 3 is used by default.

Note: The `-s` and `-z` options cannot be used at the same time.

The following table shows how the optimization levels are mapped:

<i>Option modifier</i>	<i>Description</i>
3	Fully debuggable
6	Heavy optimization can make the program flow difficult to follow during debug
9	Full optimization



This option is related to the **Optimization** options in the **ICCM32C** category in the IAR Embedded Workbench.

-2

Forces the compiler to use the 64-bit IEEE floating-point format.

SYNTAX

-2

DESCRIPTION

By default the compiler uses 32-bit precision for the data types `double` and `long double`. This option selects the 64-bit IEEE floating-point format instead.

For example, to generate code for the huge memory model using the 64-bit floating-point format, give the command:

```
icc32c filename --memory_model=huge -2
```



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

C LIBRARY FUNCTIONS

This chapter gives an introduction to the C library functions. It also lists the header files used to access library definitions. If you are using the Embedded C++ library, you should read the *EC++ library functions* chapter instead.

For detailed information about the C library functions, see the file `c_library.pdf` which is provided with the product.

INTRODUCTION

The M32C IAR Compiler package provides most of the important C library definitions that apply to embedded systems. These are of the following types:

- ◆ Adherence to free-standing implementation of the ISO standard for the programming language - C.

For information about the IAR implementation of the standard, see the chapter *Implementation-defined behavior* in this guide.
- ◆ Standard C library definitions, for user programs.
- ◆ CSTARTUP, the single program module containing the start-up code. It is described in *Initialization*, page 32.
- ◆ Run-time support libraries, for example low-level floating-point routines.

LIBRARY OBJECT FILES

You must select the appropriate library object file for your chosen memory and code model. See *Run-time library*, page 24, for more information. The linker includes only those routines that are required (directly or indirectly) by the user's program.

Most of the library definitions can be used without modification, that is, directly from the library object files supplied. There are some I/O-oriented routines that you may need to customize for your target application.

HEADER FILES

The user program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into a number of 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 this can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY
DEFINITIONS
SUMMARY

This section lists the header files. Header files may additionally contain target-specific definitions; these are documented in the chapter *IAR C extensions*.

The standard C library header files are:

<i>Header file</i>	<i>Usage</i>
<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>iso646.h</code>	Alternative names for logical operators
<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
<code>stddef.h</code>	Defining several useful types and macros
<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

<i>Header file</i>	<i>Usage</i>
<code>time.h</code>	Converting between various time and date formats
<code>wchar.h</code>	Supporting wide characters
<code>wctype.h</code>	Classifying wide characters

MATH FUNCTIONS

The C library contains both 32-bit IEEE (`float`), and 64-bit IEEE (`double`) math functions. The 32-bit version is more effective in code size and execution speed. As C does not allow function name overloading, the 32-bit functions have the suffix `f`, for example, the standard library function for sinus is `sin` and the additional 32-bit IEEE is `sinf`.

The `sin` and `sinf` are actually macros mapping to `_Sin()` and `_FSin()` functions, and can be redefined by the user.

Examples

```
#include <math.h>
void f()
{
    double d1, d2; // 64-bit IEEE
    float f1, f2;  // 32-bit IEEE

    d2 = sin(d1);   // 64-bit library used
    d2 = sin(f1);   // NOTE! 64-bit library used, float f1
                    // is casted to double before the
                    // function call.
    f2 = sinf(f1);  // 32-bit library used.

    #undef sin
    #define sinf(x) sin(x)

    d2 = sin(d1)    // 32-bit library used
    f2 = sin(f1)    // 32 bit library used
```

EC++ LIBRARY FUNCTIONS

This chapter gives an introduction to the Embedded C++ library functions. It also lists the header files used for accessing library definitions.

If you are using the IAR C library, you should read the *C library functions* chapter instead.

INTRODUCTION

The M32C IAR Compiler package provides most of the important C library definitions that apply to PROM-based embedded systems. These are of the following types:

- ◆ Adherence to a free-standing implementation of the ISO standard for the programming language C. For additional information, see the chapter *Implementation-defined behavior*.
- ◆ Standard C library definitions, for user programs.
- ◆ Embedded C++ library definitions, for user programs.
- ◆ CSTARTUP, the single program module containing the start-up code. It is described in *Initialization*, page 32.
- ◆ Run-time support libraries; for example, low-level floating-point routines.

LIBRARY OBJECT FILES

You must select the appropriate library object file for your chosen memory model. See *Run-time library*, page 24, for more information. The linker will include only those routines that are required—directly or indirectly—by your application.

Most of the library definitions can be used without modification, that is, directly from the supplied library object files. There are some I/O-oriented routines (such as `__writechar` and `__readchar`) that you may need to customize for your application. For a description of how to modify the library definitions, see *Customizing a primitive I/O function on the command line*, page 30.

HEADER FILES

The user program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into a number of 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 this can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

VIEWING THE C LIBRARY DOCUMENTATION

The library documentation is located in the `m32c\doc` directory and can be accessed from the **Help** menu in the IAR Embedded Workbench.

- ◆ To view the *C library* documentation, you need access to Acrobat® Reader. Notice that the pdf file format must be associated with the Acrobat® Reader.
- ◆ To view the *Embedded C + + library* documentation, you need access to an Internet browser. Notice that the html file format must be associated with your Internet browser.

LIBRARY
DEFINITIONS
SUMMARY

This section lists the header files. Header files may additionally contain target-specific definitions; these are documented in the chapter *IAR C extensions*.

EMBEDDED C ++

The following table shows the Embedded C ++ library headers:

Header file	Usage
complex	Defining a class that supports complex arithmetic
exception	Defining several functions that control exception handling
fstream	Defining several I/O streams classes that manipulate external files
iomanip	Declaring several I/O streams manipulators that take an argument

<i>Header file</i>	<i>Usage</i>
<code>ios</code>	Defining the class that serves as the base for many I/O streams classes
<code>iosfwd</code>	Declaring several I/O streams classes before they are necessarily defined
<code>iostream</code>	Declaring the I/O streams 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 streams classes that manipulate string containers
<code>stdexcept</code>	Defining several classes useful for reporting exceptions
<code>streambuf</code>	Defining classes that buffer I/O streams operations
<code>string</code>	Defining a class that implements a string container
<code>strstream</code>	Defining several I/O streams classes that manipulate in-memory character sequences

USING STANDARD C LIBRARIES IN EC ++

The Embedded C ++ library works in conjunction with 15 of the headers from the standard C library, sometimes with small alterations. The headers come in two forms, new and traditional.

The following table shows the new headers:

<i>Header file</i>	<i>Usage</i>
<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>climits</code>	Testing integer type properties

<i>Header file</i>	<i>Usage</i>
<code>ctype</code>	Adapting to different cultural conventions
<code>cmath</code>	Computing common mathematical functions
<code>setjmp</code>	Executing non-local goto statements
<code>signal</code>	Controlling various exceptional conditions
<code>stdarg</code>	Accessing a varying number of arguments
<code>stddef</code>	Defining several useful types and macros
<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

STANDARD C

The following table shows the traditional standard C library headers:

<i>Header file</i>	<i>Usage</i>
<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>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
<code>stddef.h</code>	Defining several useful types and macros
<code>stdio.h</code>	Performing input and output

<i>Header file</i>	<i>Usage</i>
<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

COMPATIBILITY WITH STANDARD C ++

In this implementation, the Embedded C ++ library also includes several headers for compatibility with traditional C ++ libraries:

<i>Header file</i>	<i>Usage</i>
<code>fstream.h</code>	Defining several I/O streams template classes that manipulate external files
<code>iomanip.h</code>	Declaring several I/O streams manipulators that take an argument
<code>iostream.h</code>	Declaring the I/O streams objects that manipulate the standard streams
<code>new.h</code>	Declaring several functions that allocate and free storage

EXTENDED KEYWORDS REFERENCE

This chapter describes the non-standard keywords that support specific features of the M32C and M16C/8x Series CPU cores.

SUMMARY OF EXTENDED KEYWORDS

The following list summarizes the extended keywords that are available for the M32C IAR C/C++ Compiler:

- ◆ `__near` controls the storage of variables and the representation of pointers in near memory.
- ◆ `__far` controls the storage of variables and the representation of pointers in far memory.
- ◆ `__huge` controls the storage of variables and the representation of pointers in huge memory.
- ◆ `__no_init` controls storage of variables. A variable declared as `__no_init` will be placed in a special non-initialized segment for the above storage type, for example `NDATA`. Variables placed in these non-initialized segments will not be initialized at startup.
- ◆ `__sdata` provides SB-relative data access for 8-bit variables (which requires 1 byte) instead of accessing the absolute address location (which requires 2 bytes).
- ◆ `__sdata16` provides SB-relative data access for 16-bit variables (which requires 2 bytes) instead of accessing the absolute address location (which requires 3 bytes).
- ◆ `__bitvar` allows you to write code that is equivalent to the relocatable bit type.
- ◆ `__interrupt` supports interrupt functions.
- ◆ `__fast_interrupt` works as `__interrupt` but uses the fast interrupt mechanism and returns with a `FREIT` instruction.
- ◆ `__regbank_interrupt` works as `__interrupt` but uses the secondary register bank.

- ◆ `__tiny_func` calls functions with JSRS via a vector in the special page.
- ◆ `__monitor` supports atomic execution of a function.
- ◆ `__c_task` supports `main` and processes main functions that are never called by other functions. The local address area will not be restored upon exit.

STORAGE

By default the compiler places variables in a memory segment depending on the memory model used.

- ◆ In the near memory model, the default placement is `UDATA0` for zero-initialized variables, `IDATA0` for initialized variables, and `NDATA0` for uninitialized variables.
- ◆ In the far memory model, the default placement is `UDATA1` for zero-initialized variables, `IDATA1` for initialized variables, and `NDATA1` for uninitialized variables.
- ◆ In the huge memory model, the default placement is `UDATA2` for zero-initialized variables, `IDATA2` for initialized variables, and `NDATA2` for uninitialized variables.

Constants, i.e. `const` declared variables, and constant strings are by default placed in constant segments located in code memory.

The default location can be overridden by the following keywords:

- ◆ `__near` places the variable in near memory area, i.e. memory location `0x00–0xFFFF`. Access to these variables can sometimes be faster and generate less code than access to variables placed in any of the other memories.
- ◆ `__far` places the variable in far memory area, i.e. memory location `0x00–0xFFFFFFFF`.
- ◆ `__huge` places the variable in huge memory area, i.e. memory location `0x00–0xFFFFFFFF`.

There are several pointer types that can be used to access the memory areas specified above. The pointers differ in how they access memory and in size.

Code pointers are always 24 bits, with at storage size of 4 bytes.

The data pointers are as follows:

<i>Keyword</i>	<i>Storage in bytes</i>	<i>Comment</i>
<code>__near</code> (default)	2 bytes	Can only point into 0–64 Kbytes.
<code>__far</code>	4 bytes	Element pointed at must be inside a 64 Kbyte page.
<code>__huge</code>	4 bytes	No restrictions.

The keywords follow the same syntax as the type qualifiers `const` and `volatile`. The following declaration places the variable `i` in a `__far` memory segment:

```
__far int i;
```

Notice however the difference between placing the keyword before and after the type specifier. When the keyword is placed before the type, it affects all the identifiers of the declaration. Otherwise, the keyword only affects the identifier that follows immediately. In the following example, `a`, `b`, and `c` are thus placed in `__far` memory, whereas `d` is not affected by the keyword:

```
__far short a, b;
short __far c, d;
```

A keyword that is followed by an asterisk (*), affects the type of the pointer being declared. A pointer to `__near` memory is thus declared by:

```
char __near * p;
```

Notice that the location of the pointer variable `p` is not affected by the keyword. In the following example, however, the pointer variable `p2` is placed in `__far` memory. Like `p`, `p2` points to a character in `__near` memory.

```
__far char __near *p2;
```

Storage can also be specified using typedefs. The following two declarations are equivalent:

```
typedef char __far Byte;
typedef Byte *BytePtr;
Byte b;
BytePtr bp;
```

and

```
__far char b;  
char __far *bp;
```

It is possible to avoid the non-standard keywords in declarations by using `#pragma` directives. The `#pragma type_attribute` controls the storage of variables.

The previous example may be rewritten using the `#pragma type_attribute`:

```
#pragma type_attribute=__far  
typedef char Byte;  
typedef Byte *BytePtr;  
...
```

The `#pragma type_attribute` only affects the declaration of the identifier that follows immediately. The following two declarations are therefore equivalent:

```
#pragma type_attribute=__far  
short c, d;
```

and

```
short __far c, d;
```

This means that only `c` is affected by the keyword.

See the chapter *#pragma directives reference* for a complete description of the `#pragma` directives.

It is, for obvious reasons, impossible to place a variable in more than one memory segment. It is therefore not feasible to specify more than one of the keywords in a declaration. Multiple keywords result in a diagnostic message. The keyword that is specified "closest" to the identifier is used in this case. In the following declarations `x1` and `y1` are placed in `__far` memory, while `x2` and `y2` is placed in `__near` memory:

```
__far int x1, __near x2;  
__near int __far y1, y2;
```

Direct usage of keywords, as in the above example, overrides a keyword that is specified in a `#pragma`.

UNINITIALIZED VARIABLES

The `__no_init` keyword changes the definition of a variable. It places a variable in the NDATA non-volatile memory segment and suppresses initialization at startup.

- ◆ In the near memory model, the default placement is NDATA0.
- ◆ In the far memory model, the default placement is NDATA1.
- ◆ In the huge memory model, the default placement is NDATA2.

The keyword is placed in front of the type, for instance to place settings in non-volatile memory:

```
__no_init int settings[10];
```

`#pragma object_attribute` can also be used. The following declaration of settings is equivalent to the previous one:

```
#pragma object_attribute=__no_init  
int settings[10];
```

Note: `__no_init` cannot be used in typedefs.

Unlike the keywords that specify storage and access of a variable, it is not necessary to specify `__no_init` in declarations. The following example declares settings without a keyword (for example, in a header file:)

```
extern int settings[];
```

The definition of settings specifies that it is placed in non-volatile memory:

```
__no_init int settings[10];
```

If a keyword is specified in a declaration, it is used in the subsequent definition of the variable, for instance:

```
extern __no_init int settings[];  
...  
int settings[10];
```

ABSOLUTE LOCATION

It is possible to specify the location of a variable (its absolute address) using either of the following two constructs:

- ◆ The operator `@` followed by a constant-expression

◆ #pragma location

The following declaration locates CNTRL at address 0xE8:

```
__no_init volatile char CNTRL @ 0xE8;
```

The equivalent declaration using the #pragma location directive is:

```
#pragma location=0xE8
#pragma object_attribute=__no_init
volatile char CNTRL;
```

Absolute-located objects are by default __no_init and cannot be initialized.

SB-RELATIVE DATA ACCESS

Instead of accessing the absolute address location (which requires 2 or 3 bytes) it is possible to use the SB-relative addressing mode which only requires 1 or 2 bytes.

For this purpose, two type keywords are supplied:

<i>Keyword</i>	<i>Offset size</i>	<i>Data limit</i>	<i>Segment</i>
__sbddata	1 byte	Not more than 256 bytes for the application.	SBDATA
__sbddata16	2 bytes	Not more than 64 Kbytes for the application.	SBDATA16

These keywords can be used in the same manner as other type attributes, as in the following examples:

```
#pragma type_attribute=__sbddata
< variable declaration >
```

or:

```
__sbddata < variable declaration >
```

The data type is not a valid pointer type, but it is possible to take the address of a variable (using the & operator). The variables implicitly have the __no_init attribute and will therefore not be initialized. When generating code, the offset is calculated as the distance between the variable location and the beginning of the SBDATA segment. If the distance exceeds the limit, an error occurs when linking.

RELOCATABLE BIT VARIABLES

The M32C IAR C/C++ Compiler allows you to write code that is equivalent with the relocatable bit type available in the ICCM16C compiler. Absolute bits are not supported.

For this purpose, use the `__bitvar` keyword. The syntax is:

```
__bitvar struct {unsigned char NAME:1;}
```

The only declaration allowed for this data type is a structure with unsigned bitfields with size one (i.e. a bit). The variable will be stored in the BITVARS segment, where each bit variable only occupies one bit.

If you find this syntax inconvenient, you can create a macro to define bit variables:

```
#define __BIT(NAME) __bitvar struct {unsigned char  
NAME:1;}
```

Then use it like this:

```
__BIT(MY_BIT);
```

FUNCTIONS

The following keywords control the calling convention of a function:

- ◆ `__interrupt`, which declares a function as an interrupt function.
- ◆ `__fast_interrupt`, as `__interrupt` except that the interrupt routine uses the fast interrupt mechanism.
- ◆ `__regbank_interrupt`, as `__interrupt` except that the interrupt routine uses the second register bank.
- ◆ `__monitor`, which declares an atomic function that cannot be interrupted.
- ◆ `__tiny_func`, which calls a function with JSRS via a vector in the special page.

The keywords are specified before the return type:

```
__monitor void foo(void);
```

A keyword that is followed by an asterisk (*) affects the type of the pointer being declared. A pointer to a `__c_task` function is declared in the following example.

```
void (__c_task * fptr) (void);
```

It is possible to avoid the non-standard keywords in declarations by using `#pragma` directives. The `#pragma type_attribute` controls the calling convention of functions. See the chapter *#pragma directives reference* for a complete description of the `#pragma` directives.

The previous declaration of `foo` may be rewritten using `#pragma type_attribute`:

```
#pragma type_attribute=__monitor  
void foo(void);
```

INTERRUPT FUNCTIONS

The `__interrupt` keyword declares a function that is called upon a processor interrupt. The function must be void and have no arguments.

If a vector is specified, the address of the function is inserted in that vector. If no vector is specified, the user must provide an appropriate entry in the vector table—preferably placed in the `cstartup` module—for the interrupt function.

The following example declares an interrupt function with interrupt vector at address 08h offset in the INTVEC segment:

```
#pragma vector=0x08  
__interrupt void my_handler(void);
```

Fixed table

The fixed table contains nine interrupts and is stored in the INTVEC1 segment. All entries are allocated in `cstartup.s48` and use hard-coded names for the interrupt handler. If you do not define a fixed interrupt handler, a default handler in `cstartup.s48` is used. The supplied file `fixedint.c` contains definition examples for all fixed interrupts.

In other words, if you do not define a fixed interrupt, you will get an empty interrupt handler that contains an REIT instruction. If you want to define an interrupt handler for a fixed interrupt, you must name it exactly as the example in `fixedint.c`. The compiler recognizes these special handler names and will generate error messages if you try to use them as a normal non-interrupt function, or if you try to attach a vector.

The following list specifies the hard-coded names:

```
__undefined_instruction_handler  
__overflow_handler  
__break_instruction_handler  
__address_match_handler  
__single_step_handler  
__watchdog_timer_handler  
__DBC_handler  
__NMI_handler
```

The following example shows how the `__break_instruction_handler` could be used:

```
__interrupt void __break_instruction_handler (void)  
{  
    /* Code for interrupt routine */  
}
```

The relocatable table

All other interrupts are installed in the relocatable interrupt vector table. This table is stored in segment INTVEC. The `cstartup` code initializes the INTB register to point to the start of INTVEC before calling `__low_level_int()`.

Examples

The following interrupt will be installed as the first vector in INTVEC. It will use the alternate register bank for fast interrupt response.

```
#pragma vector=0  
__regbank_interrupt void int1 (void)  
{  
    /* Code for interrupt routine */  
}
```

The `#pragma` directive applies only to the definition following immediately after the `#pragma`. If there is not a `#pragma` definition, the interrupt will not be installed in the INTVEC segment. You must then provide the vector manually, maybe in a special assembler file.

MONITOR FUNCTIONS

The keyword `__monitor` 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 critical resources that are shared by multiple processes. A function declared as `monitor` is equivalent to a normal function in all other aspects.

In the following example a semaphore is implemented. The semaphore is tested, and if the resource is available, setting the flag claims it. The routine then returns indicating if the requested resource can be used and clears the interrupt mask. This function is short (as all monitor functions should be) so it does not interfere with the operation of other interrupt routines.

```
char printer_free;  /* printer-free semaphore */
__monitor int got_flag(char *flag)
{
    if (!*flag)
    {
        return (*flag = 1);
    }
    return (0);
}

void f(void)
{
    if (got_flag(&printer_free))
        /* act only if printer is free */
        .... action code ....
}
```

C TASK FUNCTIONS

The `__c_task` keyword only affects the definition of a function, and specifies a `main` or processes a `main` function. Normal functions save the contents of used non-scratch registers on stack upon entry, and restore them at exit.

Functions declared as `__c_task` do not save any registers, and therefore require less stack space. Such functions should not be called from any other functions.

The function `main` may be declared `__c_task` unless it is called by itself or by another function. In real-time applications with more than one task, the root function of each task may be declared `__c_task`.

The keyword is placed in front of the return type, for instance:

```
__c_task void my_handler(void);
```

The `#pragma object_attribute` can also be used. The following declaration of `my_handler` is equivalent with the previous one:

```
#pragma object_attribute=__c_task  
void my_handler(void);
```

Note: `__c_task` cannot be used in typedefs.

Unlike the keywords that specify the calling convention of a function, it is not necessary to specify `__c_task` in declarations. The following example declares `my_handler` without any keyword (for instance, in a header file):

```
extern void my_handler(void);
```

The definition of `my_handler` specifies the `__c_task` keyword:

```
__c_task void my_handler(void)  
{  
    ...  
}
```

If a keyword is specified in a declaration, it is used in the subsequent definition of the function, for instance:

```
extern __c_task void my_handler(void);  
...  
void my_handler(void)  
{  
    ...  
}
```

INTRINSIC

The `__intrinsic` keyword is used with the IAR Systems library functions, and allows the compiler to make function-specific optimizations. In the include files provided with the product, some of the library functions are declared with the `__intrinsic` keyword. If the `__intrinsic` declaration is removed, the function will be called like a normal function. Declaring other functions as `__intrinsic` has no effect.

EMBEDDED C + +

The usage of extended keywords, which is described above, applies to the common subset of Embedded C++ and C. In Embedded C++, it is thus possible to use the keywords in type declarations and declarations of variables and functions with file scope. There are, however, certain restrictions in the declaration of Embedded C++ class members.

In C, the location of a struct member is determined by the location of the entire struct. It is thus not possible to declare the storage location of a particular member. It is, however, possible to declare in which memory the entire struct is to reside.

```
<MAttr1> struct S ss;
```

This principle extends to member variables in Embedded C++. It is not possible to declare the storage location of a particular member, but it is possible to declare in which memory the class object is to reside. It is however required that the pointer to the object can be converted to the default pointer type, without loss of precision. This is necessary, since non-static member functions expect a pointer of that type.

```
class Y {
public:
    int len;
    <MAttr1> char buf[1000]; // Error!!!
};
```

```
<MAttr1> Y myBuf; // This is OK
```

Static member variables are treated as ordinary—file scope— variables with respect to extended keywords. The following declaration is legal:

```
class Z {
    static <MAttr1> int numZ; // OK since numZ is static
};
```

It is furthermore possible to specify the absolute location of static member variables using the operator @ or the directive #pragma location.

Controlling the calling convention of non-static member functions is not possible. The calling convention of static member functions may however be modified using extended keywords, for instance:

```
class Device {
    static __interrupt void handler();
};
```

#PRAGMA DIRECTIVES

REFERENCE

This chapter describes the `#pragma` directives of the M32C IAR C/C++ Compiler.

The `#pragma` directives are preprocessed, which means that macros are substituted in a `#pragma` directive.

All `#pragma` directives should be entered like:

```
#pragma pragmaname=pragmavalue
```

or

```
#pragma pragmaname = pragmavalue
```

Note: The `#pragma` directives `warnings`, `codeseg`, `bitfields`, `baseaddr`, `memory`, `function`, and `alignment` are recognized and will give a diagnostic message but will not work. It is important to be aware of this if you need to port existing code that contains any of those old-style `#pragma` directives.

TYPE ATTRIBUTE

The `#pragma` directive `type_attribute` affects the declaration of the identifier that follows immediately after the `#pragma`. In the following example, `myBuffer` is placed in a `__near` segment, whereas the variable `i` is not affected by the `#pragma`.

```
#pragma type_attribute=__near
char inBuffer[10];
int i;
```

The following declarations, which use extended keywords, are equivalent. See the chapter *Extended keywords reference* for more details.

```
__near char inBuffer[10];
int i;
```

The `#pragma` `type_attribute` modifies only the next variable or the next function.

The following keywords can be used with the `#pragma type_attribute` for a variable:

One of `__near`, `__far`, `__huge`, `__sdata`, and `__sdata16`.

The following keywords can be used with `#pragma type_attribute` for a pointer. They are only useful together with a typedef declaration:

```
__near  
__far  
__huge
```

For more information, see page 75.

The following keywords can be used with `#pragma type_attribute` for a function:

```
__monitor  
__interrupt  
__fast_interrupt  
__regbank_interrupt  
__tiny_func
```

OBJECT ATTRIBUTE

The `#pragma` directive `object_attribute` affects the declaration of the identifier that follows immediately after the `#pragma`. In the following example the variable `bar` is placed in the non-initialized segment:

```
#pragma object_attribute=__no_init  
char bar;
```

Except for the `#pragma` directive `type_attribute` that specifies the storage and access of a variable, it is not necessary to specify an object attribute in declarations. The following example declares `bar` without a `#pragma object_attribute`:

```
extern char bar;
```

The definition of `bar` specifies that it is placed in non-initialized memory:

```
#pragma object_attribute=__no_init  
char bar;
```

If an object attribute is specified in a declaration, it is used in the subsequent definition of the variable or function. The following example defines the function `foo` as `__c_task`:

```
extern __c_task void foo(void);
```


The definition of `foo` does not specify the function as `__c_task` but this object attribute is inherited from the declaration:

```
void foo(void)
{
    ...
}
```

Note: Object attributes cannot be used in typedefs.

The following keyword can be used with the `#pragma object_attribute` for a variable:

```
__no_init
```

The following keyword can be used with the `#pragma object_attribute` for a function:

```
__c_task
```

DATASEG

Use the following syntax to place variables in a named segment:

```
#pragma dataseg=MY_SEGMENT
__no_init char myBuffer[10];
#pragma dataseg=default
```

The segment name must not be a predefined segment, see the chapter *Segment reference* for more information. The variable `myBuffer` will not be initialized at startup.

The memory in which the segment resides is optionally specified using the following syntax:

```
#pragma dataseg=__far MY_OTHER_SEG
```

All variables in `MY_OTHER_SEG` will be accessed using `__far` addressing.

CONSTSEG

Use the following syntax to place constant variables in a named segment:

```
#pragma constseg=MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
```

The segment name must not be a predefined segment, see *Segment reference* for more information.

The memory in which the segment resides is optionally specified using the following syntax:

```
#pragma constseg=__far MY_OTHER_SEG
```

All variables in MY_OTHER_SEG will be accessed using __far addressing.

LOCATION

The #pragma location directive specifies the location (absolute address) of the variable, whose declaration follows the #pragma directive. For example:

```
#pragma location=0xD0
char PORTLD;
```

VECTOR

The #pragma vector directive specifies the interrupt vector of an interrupt function whose declaration follows the #pragma directive, for example:

```
#pragma vector=0x12
__interrupt void my_handler(void);
```

DIAGNOSTICS

The following #pragma directives are available for reclassifying, restoring, and suppressing diagnostics:

DIAG_REMARK

Syntax: #pragma diag_remark=tag,tag,...

Changes the severity level to remark for the specified diagnostics.

DIAG_WARNING

Syntax: `#pragma diag_warning=tag,tag,...`

Changes the severity level to warning for the specified diagnostics.

DIAG_ERROR

Syntax: `#pragma diag_error=tag,tag,...`

Changes the severity level to error for the specified diagnostics.

DIAG_DEFAULT

Syntax: `#pragma diag_default=tag,tag,...`

Changes the severity level back to default or as defined on the command line for the diagnostic messages with the specified tags.

DIAG_SUPPRESS

Syntax: `#pragma diag_suppress=tag,tag,...`

Suppresses the diagnostic messages with the specified tags.

See the chapter *Diagnostics* for more information about diagnostic messages.

LANGUAGE

The `#pragma language` directive is used to turn on the IAR language extensions or to use the language settings specified on the command line.

Syntax: `#pragma language=[extended|default]`

extended Turns on the IAR extensions and turns off the `strict_ansi` option.

default Uses the settings specified on the command line.

OPTIMIZE

The `#pragma optimize` directive is used to decrease the optimization level or to turn off some specific optimizations.

Syntax: `#pragma optimize=token [token] ...`

where *token* is one or more of the following:

<code>s</code>	Optimizes for speed
<code>z</code>	Optimizes for size
<code>0 - 9</code>	Specifies level of optimization
<code>no_sce</code>	Turns off common sub-expression elimination
<code>no_inline</code>	Turns off function inlining
<code>no_unroll</code>	Turns off loop unrolling
<code>no_code_motion</code>	Turns off code motion.

Note: It is not possible to optimize for speed and size at the same time. Only one of the `s` and `z` tokens can be used.

This `#pragma` directive affects only the function that follows immediately after the directive.

Note: If the `#pragma optimize` directive is used to specify an optimization level that is higher than what is specified on the command line, the token of this `#pragma` directive is ignored.

PACK

The `#pragma pack` directive is used for specifying the alignment of structures and union members.

Syntax: `#pragma pack([[push|pop}], [name], [n])`

n Packing alignment, one of:
1, 2, 4, 8 or 16

name Pushed or popped alignment label.

`pack(n)` sets the structure alignment to *n*. The `pack(n)` only effects declarations of structures following the `#pragma` and to the next `#pragma pack` or end of file. A `#pragma pack` within a function will only be active to the end of the function.

`pack()` resets the structure alignment to default.

`pack(push [, name] [, n])` pushes the current alignment with the label *name* and sets alignment to *n*. Notice that both *name* and *n* are optional.

`pack(pop [, name] [, n])` pops to the label *name* and sets alignment to *n*. Notice that both *name* and *n* are optional.

If *name* is omitted, only top alignment is removed. If *n* is omitted, alignment is set to the value popped from the stack.

PREDEFINED SYMBOLS

REFERENCE

This chapter gives reference information about the symbols predefined by the M32C IAR C/C++ Compiler.

__DATE__	Current date.
SYNTAX	
<code>__DATE__</code>	
DESCRIPTION	
The date of compilation is returned in the form Mmm dd yyyy.	
<hr/>	
__FILE__	Current source filename.
SYNTAX	
<code>__FILE__</code>	
DESCRIPTION	
The name of the file currently being compiled is returned.	
<hr/>	
__IAR_SYSTEMS_ICC__	IAR C Compiler identifier.
SYNTAX	
<code>__IAR_SYSTEMS_ICC__</code>	
DESCRIPTION	
The number 3 is returned. This symbol can be tested with <code>#ifdef</code> to detect that it was compiled by an IAR Systems C/C++ Compiler.	

__LINE__

Current source line number.

SYNTAX

__LINE__

DESCRIPTION

The current line number of the file currently being compiled is returned.

__STDC__

ISO/ANSI standard C identifier.

SYNTAX

__STDC__

DESCRIPTION

The number 1 is returned. This symbol can be tested with `#ifdef` to detect that the compiler used adheres to ANSI C.

__STDC_VERSION__

ISO/ANSI Standard C and version identifier.

SYNTAX

__STDC_VERSION__

DESCRIPTION

The number 199409L is returned.

__TID__

Target identifier.

SYNTAX

__TID__

DESCRIPTION

The target identifier contains the following parts:

- ◆ A number unique for each IAR Systems Compiler (i.e. unique for each target).

- ◆ The value of the corresponding `--cpu` option.
- ◆ The value corresponding to the `--memory_model` option, which is 0 for the near memory model, 1 for the far memory model, and 2 for the huge memory model.

For the M32C and M16C/8x Series CPU cores, the target identifier is 48.

The `__TID__` value is constructed as:

```
((t << 8) | (c << 4) | m)
```

You can extract the values as follows:

```
t = (__TID__ >> 8) & 0x7F; /* target identifier */
c = (__TID__ >> 4) & 0xF; /* cpu option */
m = __TID__ & 0x0F;      /* memory model */
```

To find the value of the target identifier for the current compiler, execute the following command:

```
printf("%ld", (__TID__ >> 8) & 0x7F)
```

For an example where `__TID__` is used, see the file `stdarg.h`.

__TIME__

Current time.

SYNTAX

```
__TIME__
```

DESCRIPTION

The time of compilation is returned in the form `hh:mm:ss`.

__VER__

Returns the compiler version number.

SYNTAX

```
__VER__
```

DESCRIPTION

The version number of the compiler is returned as an integer.

EXAMPLE

The example below prints a message for version 3.34.

```
#if __VER__ == 334  
#message "Compiler version 3.34"  
#endif
```

INTRINSIC FUNCTIONS REFERENCE

This chapter gives reference information about the intrinsic functions. To use the intrinsic functions, include the header file `inm32c.h`.

`__break_instruction`

Inserts a BRK instruction.

SYNTAX

```
void __break_instruction(void);
```

DESCRIPTION

This intrinsic function inserts a BRK instruction.

`__disable_interrupt`

Disables global interrupts.

SYNTAX

```
void __disable_interrupt(void);
```

DESCRIPTION

This intrinsic function disables interrupts by clearing the I flag of the FLG register.

`__enable_interrupt`

Enables global interrupts.

SYNTAX

```
void __enable_interrupt(void);
```

DESCRIPTION

This intrinsic function enables interrupts by setting the I flag of the FLG register.

<code>__interrupt_on_overflow</code>	Inserts an INTO instruction.
--------------------------------------	------------------------------

SYNTAX

```
void __interrupt_on_overflow(void);
```

DESCRIPTION

This intrinsic function inserts an INTO instruction.

<code>__intrinsic_load_DCT</code>	Places 16-bit data in DCT registers.
-----------------------------------	--------------------------------------

SYNTAX

Syntax: `void __intrinsic_load_DCT (unsigned short
 dmaChannel, unsigned short data);`

DESCRIPTION

This intrinsic function places 16-bit data in DCT registers.

<code>__intrinsic_load_DMA</code>	Places 24-bit data in DMA registers.
-----------------------------------	--------------------------------------

SYNTAX

Syntax: `void __intrinsic_load_DMA (unsigned short
 dmaChannel, unsigned long data);`

DESCRIPTION

This intrinsic function places 24-bit data in DMA registers.

<code>__intrinsic_load_DMD</code>	Places 16-bit data in DMD registers.
-----------------------------------	--------------------------------------

SYNTAX

Syntax: `void __intrinsic_load_DMD (unsigned short
 dmaChannel, unsigned short data);`

DESCRIPTION

This intrinsic function places 16-bit data in DMD registers.

__intrinsic_load_DRA

Places 24-bit data in DRA registers.

SYNTAX

Syntax: void __intrinsic_load_DRA (unsigned short
 dmaChannel, unsigned long *data*);

DESCRIPTION

This intrinsic function places 24-bit data in DRA registers.

__intrinsic_load_DRC

Places 16-bit data in DRC registers.

SYNTAX

Syntax: void __intrinsic_load_DRC (unsigned short
 dmaChannel, unsigned short *data*);

DESCRIPTION

This intrinsic function places 16-bit data in DRC registers.

__intrinsic_load_DSA

Places 24-bit data in DSA registers.

SYNTAX

Syntax: void __intrinsic_load_VCT (unsigned short
 dmaChannel, unsigned long *data*);

DESCRIPTION

This intrinsic function places 24-bit data in DSA registers.

__intrinsic_load_VCT

Places 24-bit data in VCT register.

SYNTAX

Syntax: void __intrinsic_load_VCT (unsigned long *data*);

DESCRIPTION

This intrinsic function places 24-bit data in VCT register.

__intrinsic_store_DCT

Retrieves 16-bit data from DCT registers.

SYNTAX

Syntax: unsigned short __intrinsic_store_DCT (unsigned short *dmaChannel*);

DESCRIPTION

This intrinsic function retrieves 16-bit data from DCT registers.

__intrinsic_store_DMA

Retrieves 24-bit data from DMA registers.

SYNTAX

Syntax: unsigned long __intrinsic_store_DMA (unsigned short *dmaChannel*);

DESCRIPTION

This intrinsic function retrieves 24-bit data from DMA registers.

__intrinsic_store_DMD

Retrieves 16-bit data from DMD registers.

SYNTAX

Syntax: unsigned short __intrinsic_store_DMD (unsigned short *dmaChannel*);

DESCRIPTION

This intrinsic function retrieves 16-bit data from DMD registers.

__intrinsic_store_DRA

Retrieves 24-bit data from DRA registers.

SYNTAX

Syntax: unsigned long __intrinsic_store_DRA (unsigned short *dmaChannel*);

DESCRIPTION

This intrinsic function retrieves 24-bit data from DRA registers.

intrinsic store DRC

Retrieves 16-bit data from DRC registers.

SYNTAX

Syntax: unsigned long __intrinsic_store_DRC (unsigned short *dmaChannel*);

DESCRIPTION

This intrinsic function retrieves 16-bit data from DRC registers.

intrinsic store DSA

Retrieves 24-bit data from DSA registers.

SYNTAX

Syntax: unsigned long __intrinsic_store_DSA (unsigned short *dmaChannel*);

DESCRIPTION

This intrinsic function retrieves 24-bit data from DSA registers.

__intrinsic_store_VCT

Retrieves 24-bit data from VCT register.

SYNTAX

Syntax: unsigned long __intrinsic_store_DSA (void);

DESCRIPTION

This intrinsic function retrieves 24-bit data from VCT register.

__no_operation

Inserts a NOP instruction.

SYNTAX

```
void __no_operation (void);
```

DESCRIPTION

This intrinsic function generates a NOP instruction.

`__overflow_flag_value`

Reads the overflow flag value from the flag register.

SYNTAX

```
short __overflow_flag_value (void);
```

DESCRIPTION

This function reads the overflow flag value from the flag register. This can be used after a RMPA instruction to see whether the result was too large.

`__read_ipl`

Reads the interrupt permission level.

SYNTAX

```
unsigned char __read_ipl (void);
```

DESCRIPTION

This intrinsic function reads the interrupt permission level.

`__rmpa_instruction`

Emits an RMPA.W instruction

SYNTAX

```
long __rmpa_instruction (short *s1, short *s2, unsigned  
short n)
```

DESCRIPTION

This function emits an RMPA.W instruction. n is the counter, and s1 and s2 are pointers to the multiplicand array and the multiplier array. The sum is returned.

`__set_interrupt_table`

Loads interrupt base register (INTB).

SYNTAX

```
void __set_interrupt_table (unsigned long);
```


DESCRIPTION

This function loads the interrupt base register (INTB). The argument must be a constant that can be calculated at compile time.

__short_rmpa_instruction Emits an RMPA.B instruction.

SYNTAX

```
long __short_rmpa_instruction (signed char *s1, signed
char *s2, unsigned short n)
```

DESCRIPTION

This function emits an RMPA.B instruction. *n* is the counter, and *s1* and *s2* are pointers to the multiplicand array and the multiplier array. The sum is returned.

__software_interrupt Inserts an INT instruction.

SYNTAX

```
void __software_interrupt (unsigned char int_no);
```

DESCRIPTION

This intrinsic function causes a software interrupt by generating an INT instruction. *int_no* must be a constant that can be calculated at compile time.

__und_instruction Inserts an UND instruction.

SYNTAX

```
void __und_instruction(void);
```

DESCRIPTION

This intrinsic function inserts a UND instruction.

`__wait_for_interrupt`

Inserts a WAIT instruction.

SYNTAX

```
void __wait_for_interrupt(void);
```

DESCRIPTION

This intrinsic function generates a WAIT instruction.

`__write_ipl`

Sets the interrupt level.

SYNTAX

```
void __write_ipl (unsigned char value);
```

DESCRIPTION

This intrinsic function is used to set the current interrupt level. *value* must be a constant that can be calculated at compile time.

ASSEMBLER LANGUAGE INTERFACE

The M32C IAR C/C++ Compiler allows assembler language modules to be combined with compiled C/C++ modules. This is particularly useful for small, time-critical routines that need to be written in assembler language and then called from a C/C++ main program. This chapter describes the interface between a C/C++ main program and the assembler language routines.

CREATING A SHELL

The recommended method of creating an assembler language routine with the correct interface is to start with an assembler language source created by the C/C++ compiler. To this shell you can easily add the functional body of the routine.

The shell source needs only to declare the variables required and perform simple accesses to them, for example:

```
char globChar;
int globInt;
long globLong;

char func(int arg1, char arg2, long arg3)
{
    char locChar = arg2;           /* set local */
    globInt = arg1;                 /* use globInt/arg1 */
    globChar = arg2;                /* use globChar/arg2 */
    globLong = arg3;                /* use globLong/arg3 */
    return locChar;                 /* set return value */
}

void main(void)
{
    long locLong = globLong;
    globChar = func(globInt, globChar, locLong);
}
```

Note: Use a low optimization level when compiling the code. Otherwise required references to local variables could be removed during optimization. The actual function declaration is not changed by the optimization level.



Compiling the program using the IAR Embedded Workbench™

Select **Options...** from the **Project** menu. In the **Options** dialog box, choose the **ICCM32C** category, and then the **List** tab. Select **Assembler file** and the suboption **C source**. Then click **OK** to close the **Options** dialog box. Compile the program by selecting **Compile** from the **Project** menu.



Compiling the program using the command line

Use the following command to compile the program:

```
iccm32c filename -lA .
```

The `-lA` option creates an assembler language output file including C/C++ source lines as assembler comments. The `.` (period) specifies that the assembler file should be named in the same way as the C/C++ module, i.e. `shell`, but with the `s48` extension.

The result is the assembler source `filename.s48` containing the declarations, function call, function return, and variable accesses.



The following list example shows an assembler output file with C source comments. The list file has been slightly modified to work as a good example.

```
NAME filename
```

```
RTMODEL "Double size", "32"
RTMODEL "Memory model", "near"
RTMODEL "Processor", "M32C"
```

```
RSEG CSTACK:NEARDATA:NOROOT(1)
```

```
EXTERN ?CLM32CN_2_10_L00
```

```
PUBLIC func
FUNCTION func,0203H
PUBLIC globChar
PUBLIC globInt
PUBLIC globLong
PUBLIC main
```

```

        FUNCTION main,0a03H

        RSEG UDATA0:NEARDATA:NOROOT(0)
;;      1 char globChar;
globChar:
        DS8 1

        RSEG UDATA0:NEARDATA:NOROOT(1)
;;      2 int globInt;
globInt:
        DS8 2

        RSEG UDATA0:NEARDATA:NOROOT(1)
;;      3 long globLong;
globLong:
        DS8 4
;;      4

        RSEG CODE:FARCODE:REORDER:NOROOT(0)

;;      5 char func(int arg1,char arg2,long arg3)
func:
        REQUIRE ?CLM32CN_2_10_L00
;;      6 {
        ENTER      #0
        PUSHM      R3,R2,R1
        MOV.W      R0,R2
        MOV.B      8[FB],R0H
        MOV.L      10[FB],R3R1
;;      7  char locChar =arg2; /*set local */
        MOV.B      R0H,R0L
;;      8  globInt =arg1;      /*use globInt/arg1 */
        MOV.W      R2,globInt:16
;;      9  globChar =arg2;      /*use globChar/arg2 */
        MOV.B      R0H,globChar:16
;;     10  globLong =arg3;      /*use globLong/arg3 */
        MOV.L      R3R1,globLong:16
;;     11  return locChar;      /*set return value */
        POPM       R3,R2,R1
        EXITD

```

```
:: 12 }
:: 13

        RSEG CODE:FARCODE:REORDER:NOROOT(0)
:: 14 void main(void)
main:
    REQUIRE ?CLM32CN_2_10_L00
:: 15 {
    PUSHM      R3,R1,R0
:: 16  long locLong =globLong;
    MOV.L      globLong:16,R3R1
:: 17  globChar =func(globInt,globChar,locLong);
    PUSH.L     R3R1
    PUSH.B     globChar:16
    MOV.W      globInt:16,R0
    JSR.A      func:24
    ADD.L      #6,SP
    MOV.B      R0L,globChar:16
:: 18 }
    POPM      R3,R1,R0
    RTS

        RSEG SBDATA:FARDATA:ROOT(1)

    END
```

The following section describes the interface in detail.

C CALLING CONVENTION

REGISTER USAGE AND PARAMETER PASSING

All registers that do not participate in the first parameter or return value must be preserved.

If R0L is used, the whole R0 register will be overwritten.

The first parameter is passed in the following register(s), unless it is a struct or a union:

Size	Register	Comment
1 byte	R0L	
2 bytes	R0	16-bit address

<i>Size</i>	<i>Register</i>	<i>Comment</i>
3 bytes	A0	24-bit address
4 bytes	R2 : R0	

If a parameter is a `struct` or a `union`, the data is passed on the stack along with a pointer to the data. If it is the first parameter, the pointer is placed in register.

RETURN VALUES

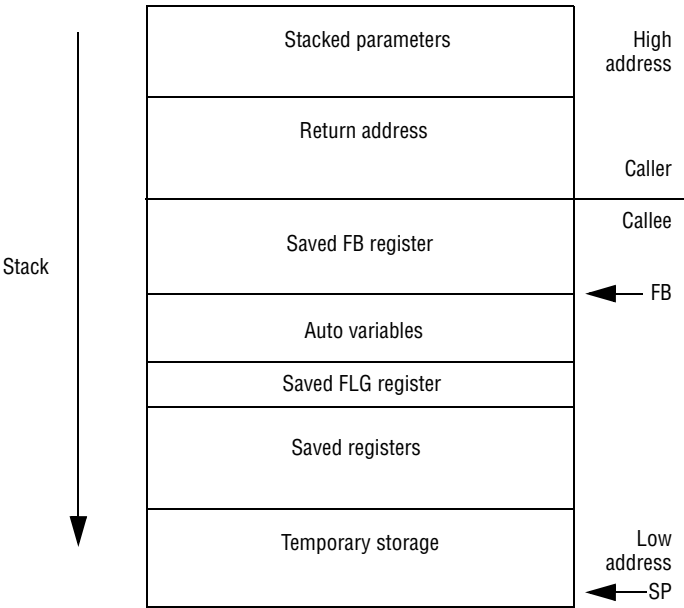
Return values are passed in the following registers:

<i>Size</i>	<i>Register</i>	<i>Comment</i>
1 byte	R0L	
2 bytes	R0	16-bit address
4 bytes	R2 : R0	24-bit address

If the return value is a `struct` or a `union`, the data is returned in a special memory area and the return value is a pointer to it.

STACK FRAMES

A function call creates a stack frame as follows:



The return address and saved FB registers are mandatory.

INTERRUPT HANDLING

INTERRUPT FUNCTIONS

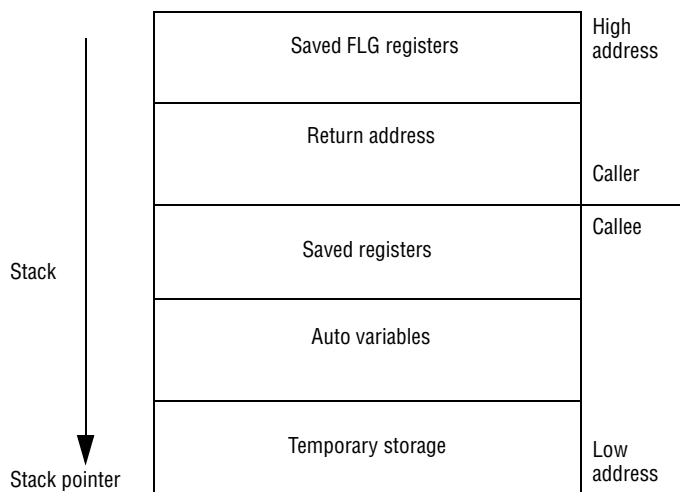
The calling convention for ordinary functions cannot be used for interrupt functions since the interrupt can occur at any time during program execution. Hence the requirements for an interrupt routine are different from those of a normal function, as follows:

- ◆ All registers that are changed by the interrupt service routine must be saved.
- ◆ The routine must consider all control registers and flags within them as undefined.
- ◆ Interrupt routines may call re-entrant functions, but the use of lengthy functions should be avoided to prevent conflicts with real-time interrupts.

- ◆ Parameters are not allowed for interrupt functions.

INTERRUPT STACK FRAMES

An interrupt function call creates a stack frame as follows:



For an example of a interrupt service routine written in C/C++, see `tutor3.c`, which is used in the compiler tutorials in the *IAR Embedded Workbench™ IDE User Guide*.

DEFINING INTERRUPT VECTORS

When you have an assembler-written interrupt function, you must install it in the interrupt vector table. See the `cstartup` file for a description.

The interrupt vectors are located in the `INTVEC` segment, and the fixed interrupts in the `INTVEC1` segment.

MONITOR FUNCTIONS

In the case of a monitor function, the compiler saves the FLG using a `PUSHC` instruction and clears the interrupt enable bit (disabling masked interrupts). On exiting from the function the compiler restores the FLG register using a `POPC` instruction.

CALLING ASSEMBLER ROUTINES FROM C

An assembler routine that is to be called from C must:

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

```
extern int foo(void)
```

or

```
extern int foo(int i, int j)
```

RUN-TIME MODEL

Use the assembler directive RTMODEL to enforce compatibility between modules. If a module defines a run-time model attribute, all modules that are linked and with this module and define the same run-time attributes, must have the same value for that key, or the special value * (asterisk).

In the current version of the M32C IAR C/C++ Compiler, run-time model attributes are not in use.

If you are using assembler routines in the C/C++ code, refer to the chapter *Assembler Directives Reference* in the *M32C IAR Assembler Reference Guide*.

EMBEDDED C++

The C calling convention, which is described on page 108, does not apply to Embedded C++ functions. Most importantly, a function name is not sufficient to identify an Embedded 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.

Using C linkage, the calling convention however conforms to the above description. An assembler routine may therefore be called from Embedded C++ when declared in the following manner:

```
extern "C" {  
    int my_routine(int x);  
}
```

Member functions cannot be given C linkage. It is however possible to construct the equivalent non-member functions. Member access control is not an issue, since there is no way of preventing an assembler routine from accessing private and protected members.

To achieve the equivalent to a non-static member function, the implicit pointer has to be made explicit:

```
class X;

extern "C" {
    void doit(X *ptr, int arg);
}
```

It is possible to "wrap" the call to the assembler routine in a member function. Using an inline member function removes the overhead of the extra call—provided that function inlining is enabled:

```
class X {
public:
    inline void doit(int arg) { ::doit(this, arg); }
};
```

SEGMENT REFERENCE

The M32C IAR C/C++ Compiler places code and data into named segments which are referred to by the IAR XLINK Linker™. Details of the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output of the compiler.

This section provides an alphabetical list of the segments.

The type read-only or read/write indicates if the segment should be placed in ROM or RAM memory areas.

BITVARS

Bit variables.

TYPE

Read/write.

DESCRIPTION

Holds bit variables and can also hold user-defined relocatable bit variables.

CDATA0, CDATA1, CDATA2

Initialization constants for near, far, and huge data, respectively.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible.

CSTARTUP copies initialization values from this segment to the corresponding IDATA0, IDATA1, or IDATA2 segments.

CODE	<p>Code.</p> <p>TYPE</p> <p>Read-only.</p> <p>DESCRIPTION</p> <p>Holds user program code and various library routines. Notice that any assembler language routines called from C/C++ must meet the calling convention in use. For more information, see <i>Calling assembler routines from C</i>, page 112.</p>
CONST0, CONST1, CONST2	<p>Near, far, and huge constants, respectively.</p> <p>TYPE</p> <p>Read-only.</p> <p>DESCRIPTION</p> <p>Used for storing constant objects. Can be used in assembler language routines for declaring constant data.</p>
CSTACK	<p>Data stack.</p> <p>TYPE</p> <p>Read/write.</p> <p>DESCRIPTION</p> <p>Holds the user data stack.</p> <p>This segment and length is normally defined in the XLINK file with the following command:</p> <p>-Z(DATA)CSTACK+size=start</p> <p>or</p> <p>-Z(DATA)CSTACK=start-end</p> <p>where <i>size</i> is the size of the segment, <i>start</i> is the first memory location, and <i>end</i> is the last memory location.</p>

FLIST

Table of `tiny_func` jumps.

TYPE

Read-only.

DESCRIPTION

Jump table for `tiny_func` functions.

HEAP

Used for the heap.

TYPE

Read/write.

MEMORY AREA

Data. The address range depends on the memory model:

<i>Memory model</i>	<i>Address range</i>
Near	0x0-0xFFFF
Far	0x0-0xFFFFFFFF
Huge	0x0-0xFFFFFFFF

DESCRIPTION

Holds the heap data used by `malloc`, `calloc`, and `free`.

This segment and its length is normally defined in the linker command file by the command:

```
-Z(DATA)HEAP+nn=start
```

where *nn* is the length and *start* is the location.

**IDATA0, IDATA1,
IDATA2**

Initialized static data for near, far, and huge data, respectively.

TYPE

Read/write.

DESCRIPTION

Holds static variables in internal data memory that are automatically initialized from the corresponding CDATA0, CDATA1, or CDATA2 in CSTARTUP. See also CDATA0, CDATA1, CDATA2, page 115.

INTVEC	Interrupt vectors.
	TYPE
	Read-only.
	DESCRIPTION
	Holds the interrupt vector table generated by the use of any of the extended keywords for interrupt functions. The keywords can also be used for user-written interrupt vector table entries).

INTVEC1	Fixed interrupt vectors.
	TYPE
	Read-only.
	DESCRIPTION
	Holds the interrupt vector table generated by the use of the __interrupt extended keyword with one of the following names:
	__undefined_instruction_handler
	__overflow_handler
	__break_instruction_handler
	__address_match_handler
	__single_step_handler
	__watchdog_timer_handler
	__DBC_handler
	__NMI_handler

ISTACK

Stack for interrupt functions.

TYPE

Read/write.

DESCRIPTION

Holds the stack used by interrupt functions.

**NDATA0, NDATA1,
NDATA2**

Non-volatile non-initialized variables.

TYPE

Read/write.

DESCRIPTION

Holds variables to be placed in non-volatile memory, which are automatically initialized from the corresponding segment NDATA0, NDATA1, or NDATA2. These will have been allocated by the compiler, declared `no_init` or created `no_init` by use of the `#pragma` directive, or created manually from assembler language source.

SBDATA, SBDATA16

Non-initialized SB-relative variables.

TYPE

Read/write.

DESCRIPTION

Holds variables that are accessed using the SB-relative addressing mode.

**UDATA0, UDATA1,
UDATA2**

Uninitialized static data.

TYPE

Read/write.

DESCRIPTION

Holds variables in memory that are not explicitly initialized from the corresponding segment UDATA0, UDATA1, or UDATA2. They are initialized to zero when performed by CSTARTUP.

MIGRATION HINTS

If you have C code that was originally written for version 1 of M16C IAR C/C++ Compiler, it can—with some modifications—be used also with M32C IAR C/C++ Compiler. (Code written for MC80 IAR C Compiler can be used as it is.)

This chapter contains information that is useful when migrating from version 1 of M16C IAR C/C++ Compiler to M32C/C++ IAR Compiler. It briefly describes both differences and similarities between the two products.

For information about migrating from an older version of the compiler to the new version, see the migration information supplied in the \doc directory on the installation media.

The header file `m16ccomp.h` contains compatibility definitions to facilitate the migration from M16C to M32C.

INTRODUCTION

The main difference between the two compilers is that the M32C IAR C/C++ Compiler uses a new generation of compiler front-end, which makes it possible to enhance your application code in a way that previously was not possible. One of the main advantages with the new front-end is its new global optimizer, which improves the efficiency of the generated code. The consistency of the compiler is also improved due to the new front-end.

Moreover, the new compiler front-end allows you to write source code that is easily portable since it adheres more strictly to the ISO/ANSI standard; for example, it is possible to use `#pragma` directives instead of extended keywords for defining special function registers (SFRs). Also the data type checking adheres more strictly to the ISO/ANSI standard in M32C than in M16C. Therefore, it is important to be aware that code written for M16C may generate warnings or errors in the M32C IAR C/C++ Compiler.

The set of language extensions has changed in M32C. Some extensions have been added, some extensions have been removed, or the syntax has changed. There is also a rare case where an extension has a different interpretation if typedefs are used. See *Extended keywords*, page 122, for detailed information.

EXTENDED KEYWORDS

In M32C, all extended keywords except `asm` start with two underscores, for example `__near`, compared to `near` in M16C.

`__NEAR`, `__FAR`, AND `__HUGE`

Both M16C and M32C allow keywords that specify memory location. Each of these keywords can be used either as a placement attribute for an object, or as a pointer type attribute denoting a pointer that can point to the specified memory.

When the keywords are used directly in the source code, they behave in the same way in both M16C and M32C. The usage of typedef and extended keywords is, however, more strict in M32C than in M16C.

The M16C compiler behaves unexpectedly in some cases:

```
typedef int far FINT;
FINT a,b;
FINT near c;          /* Illegal */
FINT *p;              /* p in far memory, points to
                      default memory attribute */
```

The first variable declaration works as expected, that is `a` and `b` are located in far memory. The declaration of `c` is however illegal, except when `far` is the default memory, in which case there is no need for an extended keyword in the typedef.

In the last declaration, the `far` keyword of the typedef affects the location of the pointer variable `p`, not the type of pointer. The type of the pointer is the default, which is given by the memory model.

The corresponding example for M32C is:

```
typedef int __far FINT;
FINT a,b;
FINT __near c;        /* c stored in near memory --
                      override attribute in typedef */
FINT *p;              /* p points to far memory, p stored
                      in default memory */
```

The declarations of `c` and `p` differ. The `__near` keyword in the declaration of `c` will always compile. It overrides the keyword of the typedef. In the last declaration the `__far` keyword of the typedef affects the type of the pointer. It is thus a `__far` pointer to `int`. The location of the variable `p` is however not affected.

__NO_INIT

The M16C keyword `no_init` specifies that an object is not initialized and that it resides in default memory. In M32C `__no_init` can be used together with a keyword specifying any memory location, for example:

```
__far __no_init char buffer [1000];
```

__INTERRUPT

In M16C the keyword `interrupt` specifies not only the type attribute setting but also the memory location. In M32C `__interrupt` is a type attribute only.

In M16C a vector can be attached to an `interrupt` function with the `#pragma` directive `function` or directly in the source code, for example:

```
interrupt [0x8] void f(void);
```

In M32C a vector can be attached to any function but only with the `#pragma` directive `vector`.

```
#pragma vector=2  
__interrupt void f(void);
```

__MONITOR

In M16C the keyword `monitor` specifies not only the type attribute setting but also the memory location. In M32C `__monitor` is a type attribute only.

USING

The M16C keyword `using`, which can be used to denote what register bank to use in an interrupt function, does not exist in M32C. Instead the keyword `__regbank_interrupt` is available.

SFR AND BIT

In M16C `sfr` and `sfrp` keywords denote an object of byte or word size residing in the Special Function Register (SFR) memory area for the chip, and having a `volatile` type. The SFR is always located at an absolute address. For example:

```
sfr PORT=100;
```

In M32C `sfr` and `sfrp` are not available. Instead you have the ability to:

- ◆ Place any object into the SFR memory, or any other memory, by using a memory attribute.
- ◆ Locate any object at an absolute address by using the `#pragma` directive `location` or by using the locator operator `@`; for example:

```
long PORT @ 100;
```

- ◆ Use the `volatile` attribute on any type, for example:

```
volatile __near char PORT@100;
```

A bit variable in M16C is a volatile boolean variable that can have an absolute bit-address, be co-located with an SFR or be a relocatable object, like ordinary variables. For example:

```
bit a = 87;           /* at bit-address 87 (M16C) */
bit p0 = PORT.5;      /* bit 5 of port (M16C) */
bit r;                /* relocatable bit (M16C) */
```

M32C uses bit fields of width 1 to implement bit-variables. The extended language feature anonymous structs allows the bits, which are struct members, to be used as if they were variables in the enclosing scope. The keyword `bit` is not available in M32C. For additional information about anonymous structs, see *Anonymous structures and unions*, page 6.

The following example shows an anonymous struct in M32C:

```
/* anonymous struct */
struct {
    char b0:1, b1:1, b2:1, :5, b7:1;
};
char foo() { return b7; }
void bar() { b0 = 1; }
```

To declare an absolute-located bit, the bit address must first be converted to a byte address. The bit `a`, in the above example, has bit address 87. Division by 8 yields byte address 10 and remainder 7, the latter of which is the bit-offset in that byte. Thus the corresponding M32C declaration is:

```
volatile __near struct { char :7, a:1; } @ 10;
```

Anonymous unions are used to locate an SFR and a bit field at the same address. For additional information, see *Anonymous structures and unions*, page 6.

The declaration of PORT (address 100) and p0 (bit 5 of PORT) are combined in the following way:

```
/* anonymous union */
volatile __near union {
char PORT;
struct { char :5, p0:1; };
} @ 100;
```

When it comes to relocatable bits, the same (maximal) packing as in M16C can be achieved by placing all bits in the same anonymous struct. For example:

```
struct
{
char r:1, s:1, t:1, u:1, v:1, x:1, y:1, z:1;
};
```

The M32C notation is not as brief as the one used in M16C. It is, on the other hand, more flexible. Bit fields can have any width (not only 1), can be located in any memory (not restricted to near) and are not necessarily volatile.

See the chapter *Extended keywords reference* for complete information about the extended keywords available in M32C.

#PRAGMA DIRECTIVES

M16C and M32C have different sets of #pragma directives for specifying attributes, which also behave differently:

- ◆ In M16C the #pragma directives change the default attribute to use for objects declared; they do not have an effect on pointer types. The #pragma directives are memory for setting the default memory placement for data objects, and function for setting the default memory placement for functions.
- ◆ In M32C the #pragma directives type_attribute and object_attribute change the next declared object or typedef. The #pragma directive memory still works—for backward compatibility—but we strongly recommend that you instead use the new pragma syntax which has cleaner semantics, giving you better control.

The rules for overriding a memory attribute differ between M16C and M32C. However, both give the highest priority to memory attribute keywords in the actual declaration, and the lowest priority to the specific segment placement `#pragma` directives.

The following M16C `#pragma` directives have been *removed* in M32C:

```
alignment
bitfields
codeseg
function
warnings
```

These are recognized and will give a diagnostic message but will not work in the M32C.

Note: Instead of the `#pragma` directive `codeseg`, the M32C command line option `-R` can be used, providing the same functionality.

The following table shows the mapping of `#pragma` directives:

<i>M16C #pragma directive</i>	<i>M32C #pragma directive</i>
<code>#pragma function=interrupt[xx]</code>	<code>#pragma type_attribute=__interrupt</code> <code>#pragma vector=xx</code>
<code>#pragma function=C_task</code>	<code>#pragma object_attribute=__c_task</code>
<code>#pragma function=interrupt</code>	<code>#pragma type_attribute=__interrupt</code>
<code>#pragma function=monitor</code>	<code>#pragma type_attribute=__monitor</code>
<code>#pragma memory=constseg</code>	<code>#pragma constseg</code>
<code>#pragma memory=dataseg</code>	<code>#pragma dataseg</code>
<code>#pragma memory=no_init</code>	<code>#pragma object_attribute=__no_init</code>
<code>#pragma memory=near</code>	<code>#pragma type_attribute=__near</code>
<code>#pragma memory=far</code>	<code>#pragma type_attribute=__far</code>
<code>#pragma memory=huge</code>	<code>#pragma type_attribute=__huge</code>

Note: All M32C `#pragma` directives in the table above affect only the declaration that follows immediately after the directive, except `#pragma memory=constseg` and `#pragma memory=dataseg` that are active until they are turned off.

The following `#pragma` directives are *identical* in M16C and M32C:


```
#pragma language=extended  
#pragma language=default
```

The following `#pragma` directives have been *added* in M32C:

```
#pragma diag_default  
#pragma diag_error  
#pragma diag_remark  
#pragma diag_suppress  
#pragma diag_warning  
#pragma location  
#pragma vector
```

Specific segment placement

In M16C the `#pragma` directive memory supports a syntax that enables subsequent data objects that match certain criterias to end up in a specified segment. In M32C the `#pragma` directives `dataseg` and `constseg` are available for this purpose.

In M16C each object found after the invocation of a segment placement directive will be placed in the segment, provided that it does not have a memory attribute placement and that it has the correct constant attributes.

In M32C only an object that has the correct memory placement will end up in the segment, provided that, for `constseg` it is a constant, and for `dataseg` that it has the `__no_init` attribute and no initializer.

See the chapter *#pragma directives reference* for complete information about the M32C `#pragma` directives.

PREDEFINED SYMBOLS

In M32C, all predefined symbols start and end with two underscores, for example `__IAR_SYSTEMS_ICC__`.

In M32C, the `__TID__` value is 48.

See the chapter *Predefined symbols reference* for complete information about the predefined symbols available in M32C.

INTRINSIC
FUNCTIONS

In M32C, all intrinsic functions start with two underscores, for example `__enable_interrupt`.

The M16C intrinsic functions `_args$` and `_argt$` are not available in M32C.

See the chapter *Intrinsic functions reference* for complete information about the intrinsic functions available for the IAR M32C C/C++ Compiler.

C COMPILER OPTIONS COMMAND LINE SYNTAX

- The M32C command line options follow two different syntax styles:
- ◆ A single letter prefixed with a single dash and sometimes followed by a modifier, for example `-r` or `-mf`. This style is the only style used in M16C.
 - ◆ One or more words prefixed with two dashes and sometimes followed by an equal sign and a modifier, for example `--strict_ansi` or `--memory_model=far`. This style is not available in M16C.

Some options appear in one style only, other options appear in both styles.

The following table shows the M16C command line options that have been *removed* in M32C:

<i>M16C option</i>	<i>Comment</i>
<code>-C</code>	Nested comments
<code>-F</code>	Form-feed after each function
<code>-f filename</code>	Extend the command line
<code>-G</code>	Open standard input as source. Replaced by <code>-</code> (dash) as source file in M32C.
<code>-g</code>	Global strict type check. In M32C, global strict type checking is always enabled.
<code>-g0</code>	No type information in object code

<i>M16C option</i>	<i>Comment</i>
-K	// comments. In M32C, // comments are allowed unless the option <code>--strict_ansi</code> is used.
-O <i>prefix</i>	Set object filename prefix. In M32C, use <code>-o filename</code> instead.
-P	Generate PROMable code. This functionality is always enabled in M32C.
-p <i>nn</i>	Lines/page
-T	Active lines only
-t	Tab spacing
-U <i>symp</i>	Undefine symbol
-u{1 2}	Object alignment
-X	List C declarations
-x[DFT2]	Cross-reference
-y	Writable strings

Note: Instead of the command line option `-f`, the following methods may be used, depending on your operating system, for extending the command line:

- ◆ Use a command file to add the options; for example, a bat file in DOS and Windows.
- ◆ Use the environment variables for flags, for example QCCM32C.
- ◆ Define your own variables to be used on the command line; for example, in Windows 95 or NT:

```
set F=-memory_model=far -e
ICCM32C %F% -z9 foo.c
```

The following table shows the command line options that are *identical* in M16C and M32C:

<i>Option</i>	<i>Comment</i>
-D <i>symp=value</i>	Define symbols
-e	Language extensions

<i>Option</i>	<i>Comment</i>
-I	Include paths. (Syntax is more free in M32C.)
-o <i>filename</i>	Set object filename
-R <i>name</i>	Set code segment name
-s[3 6 9]	Optimize for speed
-Wn	Stack optimize size
-z[3 6 9]	Optimize for size
-2	64-bit floating point

The following M16C command line options have been *renamed* and/or *modified*:

<i>M16C option</i>	<i>M32C option</i>	<i>Comment</i>
-A	-la .	Assembly output. See
-a <i>filename</i>	-la <i>filename</i>	<i>Filenames</i> , page 131.
-b	--library_module	Make object a library module
-c	--char_is_signed	Char is signed char
-gA	--strict_ansi	Flag old-style functions
-H <i>name</i>	--module_name= <i>name</i>	Set object module name
-L[<i>prefix</i>], -l[c C a A][N] <i>filename</i>	-l[c C a A][N] <i>filename</i>	List file. The modifiers specify the type of list file to create.
-m[d f h]	--memory_model=[n f h] -m[n f h]	Memory model
-N <i>prefix</i> , -n	--preprocess=[c][n][l] <i>filename</i>	Preprocessor output
-q	-lA, -lC	Insert mnemonics. List file syntax has changed.
-r[012][i][n]	-r --debug	Generate debug information. The modifiers have been removed.
-S	--silent	Set silent operation

<i>M16C option</i>	<i>M32C option</i>	<i>Comment</i>
-w	--no_warnings	Disable warnings

The M32C command line options `--memory_model=near` or `-mn` both correspond to the M16C command line option `-md`, which can be used also in M32C.

Note: Some new command line options have been added in M32C. For a complete list of the command line options available in the M32C IAR C/C++ Compiler, see *Options summary*, page 40.

FILENAMES

In M16C, file references can be made in either of the following ways:

- ◆ With a specific filename, and maybe with a default extension added, using a command line option such as `-a filename` (Assembly output to named file).
- ◆ With a prefix string added to the default name, using a command line option such as `-A[prefix]` (Assembly output to prefixed filename).

In M32C, a file reference is always regarded as a *file path* that can either be a directory, which the compiler will check and then add a default filename to, or it will be treated explicitly as a filename.

The following table shows some examples where it is assumed that the source file is named `test.c`, `aaaa` is *not* a directory and `bbbb` is a directory:

<i>M16C command</i>	<i>M32C command</i>	<i>Result</i>
-l aaaa	-l aaaa	aaaa.lst
-Laaaa	-l aaaatest	aaaatest.lst
-L	-l .	test.lst
-Lbbbb/	-l bbbb -l bbbb/	bbbb/test.lst

LIST FILES

In M16C, no more than one C list file and one assembler list file can be produced; in M32C there is no upper limit on the number of list files that can be generated. The M32C command line option `-l[c|C|a|A][N]` *filename* is used to specify the behavior of each list file.

OBJECT FILE FORMAT

UBROF6 is the generated object format for M16C. When using the M16C command line option `-r` two types of source references can be generated in the object file: either the source statements was referred to (`-r`), or the actual source was embedded in the object format (`-re`).

UBROF9 is the generated object file format for M32C. When the M32C command line option `-r` or `--debug` is used, source file references are always generated in UBROF, i.e. embedding of the source is not supported.

NESTED COMMENTS

In M16C, nested comments were allowed if the option `-C` was used. In M32C, nested comments are never allowed. For example, if a comment is used to remove a statement as in the following example, it will not have the desired effect.

```
/*  
/* x is a counter */  
int x = 0;  
*/
```

The variable `x` will still be defined, there will be a warning where the inner comment begins, and there will be an error where the outer comment ends.

```
/* x is a counter */  
^  
"c:\bar.c",2 Warning[Pe009]: nested comment is not  
allowed  
  
*/  
^  
"c:\bar.c",4 Error[Pe040]: expected an identifier
```

The solution is to use `#if 0` to "hide" portions of the source code when compiling:

```
#if 0
/* x is a counter */
int x = 0;
#endif
```

Note: `#if` statements may be nested.

PREPROCESSOR FILE

In M16C, a preprocessor file can be generated as a side effect of compiling a source file.

In M32C a preprocessor file is either generated as a side effect, or as the whole purpose when parsing of the source code is not required. You may also choose to include or exclude comments and/or `#line` directives.

CROSS-REFERENCE INFORMATION

In M16C cross-reference information can be generated. This possibility is not available in M32C.

SIZEOF IN PREPROCESSOR DIRECTIVES

In M16C, `sizeof` could be used in `#if` directives, for example:

```
#if sizeof(int)==2
int i = 0;
#endif
```

In M32C, `sizeof` is not allowed in `#if` directives. The following error message will be produced:

```
    #if sizeof(int)==2
        ^
"c:\bar.c",1  Error[Pe059]: function call is not allowed
in a constant expression.
```

Macros can be used instead, for example `SIZEOF_INT`. Macros can be defined using the `-D` option, or using a `#define` in the source code:

```
#define SIZEOF_INT 2
#if SIZEOF_INT==2
int i = 0;
#endif
```

To find the size of a predefined data type, see *Data representation*, page 2. Complex data types may be computed using one of several methods:

- 1 Write a small program, and run it in the simulator, with terminal I/O.

```
#include <stdio.h>
struct s { char c; int a; };

void main(void)
{
    printf("sizeof(struct s)=%d \n", sizeof(struct s));
}
```

- 2 Write a small program, compile it with the option `-la` . to get an assembler listing in the current directory and look for the definition of the constant `x`.

```
struct s { char c; int a; };
const int x = sizeof(struct s);
```

SEGMENTS

The `CONST` segment is replaced with three segments named `CONST0`, `CONST1`, and `CONST2`. The segments correspond to the near, far, and huge memory models respectively. The default segment depends on the memory model, in the same way as for other data segments. This gives you more control over the usage and allocation of the constant segments.

IMPLEMENTATION-DEFINED BEHAVIOR

This chapter describes how IAR C handles 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: IAR C adheres 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. IAR has not implemented the following parts of the standard library: `locale`, `files` (but streams `stdin` and `stdout`), `time`, and `signal`.

This chapter 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)

IAR C produces diagnostics 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; *message* is an explanatory message, possibly several lines.

ENVIRONMENT**ARGUMENTS TO MAIN (5.1.2.2.1)**

In IAR C, the function called at program startup is called `main`. There is no prototype declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior, see the CSTARTUP description, page 32.

INTERACTIVE DEVICES (5.1.2.3)

IAR C treats the streams `stdin` and `stdout` 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)

IAR C treats identifiers with external linkage 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. In IAR C, the source character set is the standard ASCII character set.

The execution character set is the set of legal characters that can appear in the execution environment. In IAR C, the execution character set is the standard ASCII character set.

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, i.e. using the same representation value 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 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 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.

CONVERTING MULTIBYTE CHARACTERS (6.1.3.4)

The current and only locale supported in IAR C is the 'C' locale.

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 two's-complement form. The most-significant bit holds the sign; 1 for negative, 0 for positive and zero.

See *Data representation*, page 2, for information about the ranges for the different integer types: `char`, `short`, `int`, and `long`.

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, i.e. a large enough value will be converted into a negative value.

SIGNED BITWISE OPERATIONS (6.3)

Bitwise operations on signed integers work the same as bitwise operations on unsigned integers, i.e. 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 3, for information about the ranges and sizes for the different floating-point types: `float`, `double`, and `long 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 5, for information about *size_t* in IAR C.

CONVERSION FROM/TO POINTERS (6.3.4)

See *Casting*, page 5, for information about casting of data pointers and function pointers.

PTRDIFF_T (6.3.6, 7.1.1)

See *ptrdiff_t*, page 5, for information about the *ptrdiff_t* in IAR C.

REGISTERS**HONORING THE REGISTER KEYWORD (6.5.1)**

IAR C does not honor user requests for register variables. Instead it makes its own choices when optimizing.

**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 *Data representation*, page 2, for information about the alignment requirement for data objects in IAR C.

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 bitfield integer type chosen.

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)

IAR C does not limit the number of declarators. The number is limited only by the available memory.

STATEMENTS**MAXIMUM NUMBER OF CASE STATEMENTS (6.6.4.2)**

IAR C does not limit the number of case statements (case values) in a switch statement. 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 has 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 has not been found, the search continues as if the filename were enclosed in angle brackets.

CHARACTER SEQUENCES (6.8.2)

Preprocessor directives use the source character set, with the exception of 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)

The following #pragma directives are recognized in IAR C:

alignment
ARGSUSED
baseaddr
bitfields
can_instantiate
codeseg
constseg
daseg
define_type_info
diag_default
diag_error
diag_remark
diag_suppress
diag_warning
do_not_instantiate
function
hdrstop
instantiate
language
location
memory
message
none
no_pch
NOTREACHED
object_attribute
once
optimize
pack
__printf_args
__scanf_args
type_attribute
VARARGS
vector
warnings

For a description of the #pragma directives, see the chapter *#pragma directives reference*.

DEFAULT __DATE__ AND __TIME__ (6.8.8)

The definitions for __TIME__ and __DATE__ are always available.

**C 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)

IAR C does not support the signal part of the library.

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 out to the stdout stream immediately before a newline character are preserved. There is no way to read in the line through the stream `stdin` that was written out through the stream `stdout` in IAR C.

NULL CHARACTERS APPENDED TO DATA WRITTEN TO BINARY STREAMS (7.9.2)

There are no binary streams implemented in IAR C.

FILES (7.9.3)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

REMOVE() (7.9.4.1)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

RENAME() (7.9.4.2)

There are no streams other than `stdin` and `stdout` in IAR C. 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 that 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 streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

MESSAGE GENERATED BY PERROR() (7.9.10.4)

`perror()` is not supported in IAR C.

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, since this is an unsupported feature in IAR C.

BEHAVIOR OF `EXIT()` (7.10.4.3)

The `exit()` function does not return in IAR C.

ENVIRONMENT (7.10.4.4)

An environment is not supported in IAR C.

SYSTEM() (7.10.4.5)

A system is not supported in IAR C.

MESSAGE RETURNED BY `STRERROR()` (7.11.6.2)

The messages returned by `strerror()` depending on the argument is:

<i>Argument</i>	<i>Message</i>
EZERO	no error
EDOM	domain error
ERANGE	range error
<0 >99	unknown error
all others	error No.xx

THE TIME ZONE (7.12.1)

Time is not supported in IAR C.

CLOCK() (7.12.2.1)

Time is not supported in IAR C.

**EC++ LIBRARY
FUNCTIONS****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)

IAR C does not support the signal part of the library.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

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 out to the stdout stream immediately before a newline character are preserved. There is no way to read in the line through the stream `stdin` that was written out through the stream `stdout` in IAR C.

NULL CHARACTERS APPENDED TO DATA WRITTEN TO BINARY STREAMS (7.9.2)

There are no binary streams implemented in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

FILES (7.9.3)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

REMOVE() (7.9.4.1)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

RENAME() (7.9.4.2)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

%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 that 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)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

MESSAGE GENERATED BY PERROR() (7.9.10.4)

The generated message is:

usersuppliedprefix:errmsg

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, since this is an unsupported feature in IAR C.

BEHAVIOR OF EXIT() (7.10.4.3)

The `exit()` function does not return in IAR C.

ENVIRONMENT (7.10.4.4)

An environment is not supported in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

SYSTEM() (7.10.4.5)

The `system()` function is not supported in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

MESSAGE RETURNED BY STRERROR() (7.11.6.2)

The messages returned by `strerror()` depending on the argument is:

<i>Argument</i>	<i>Message</i>
EZERO	no error

<i>Argument</i>	<i>Message</i>
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>

THE TIME ZONE (7.12.1)

Time is not supported in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

CLOCK() (7.12.2.1)

Time is not supported in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

IAR C EXTENSIONS

This chapter describes IAR extensions to the ISO standard for the C programming language.

See the compiler option `-e`, page 46 for information about enabling and disabling language extensions from the command line.

AVAILABLE EXTENSIONS

The following language extensions are available:

- ◆ Functions and data may be declared with memory, type, and object attributes. The attributes follow the syntax for qualifiers but not the semantics.
- ◆ The operator `@` may be used for specifying either the location of an absolute addressed variable or the segment placement of a function or variable. The directive `#pragma location` has the same effect.
- ◆ A translation unit (input file) is allowed to contain no declarations.
- ◆ Comment text can appear at the end of preprocessing directives.
- ◆ `__ALIGNOF__` is similar to `sizeof`, but returns the alignment requirement value for a type, or 1 if there is no alignment requirement. It may be followed by a type or expression in parenthesis, `__ALIGNOF__(type)`, or `__ALIGNOF__(expression)`. The expression in the second form is not evaluated.
- ◆ Bitfields may have base types that are enums or integral types besides `int` and `unsigned int`. This matches *G.5.8* in the appendix to the ISO standard, *ISO Portability issues*.
- ◆ The last member of a struct may have an incomplete array type. It may not be the only member of the struct (otherwise, the struct would have zero size).
- ◆ A file-scope array may have an incomplete struct, union, or enum type as its element type. The type must be completed before the array is subscripted (if it is), and by the end of the compilation if the array is not external.
- ◆ Static functions may be declared in function and block scopes. Their declarations are moved to the file scope.

- ◆ enum tags may be incomplete: one may define the tag name and resolve it (by specifying the brace-enclosed list) later.
- ◆ The values of enumeration constants may be given by expressions that evaluate to unsigned quantities that fit in the `unsigned int` range but not in the `int` range. A warning is issued for suspicious cases.
- ◆ An extra comma is allowed at the end of an enum list. A remark is issued.
- ◆ The final semicolon preceeding the closing `}` of a `struct` or `union` specifier may be omitted. A warning is issued.
- ◆ A label definition may be immediately followed by a closing `}` (normally a statement must follow a label definition). A warning is issued.
- ◆ An empty declaration (a semicolon with nothing before it) is allowed. A remark is issued.
- ◆ An initializer expression that is a single value and is used for initializing an entire static array, `struct`, or `union` need not be enclosed in braces. ISO C requires the braces.
- ◆ In an initializer, a pointer constant value may be cast to an integral type if the integral type is large enough to contain it.
- ◆ The address of a variable with a `register` storage class may be taken. A warning is issued.
- ◆ In an integral constant expression, an integer constant may be cast to a pointer type and then back to an integral type.
- ◆ In duplicate size and sign specifiers (for example `short short` or `unsigned unsigned`) the redundancy is ignored. A warning is issued.
- ◆ `long float` is accepted as synonym of `double`.
- ◆ Benign redeclarations of `typedef` names are allowed. That is, a `typedef` name may be redeclared in the same scope as the same type. A warning is issued.
- ◆ Dollar signs are accepted in identifiers.

- ◆ Numbers are scanned according to the syntax for numbers rather than the pp-number syntax. Thus, `0x123e+1` is scanned as three tokens instead of one invalid token (if `--strict_ansi` is specified, the pp-number syntax is used).
- ◆ 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 (for example, `short *` and `int *`). A warning is issued. Assignment of a string constant to a pointer to any kind of character is allowed without a warning.
- ◆ Assignment of pointer types 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 **`). Comparisons and pointer difference of such pairs of pointer types are also allowed. A warning is issued.
- ◆ 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 C, some operators allow such things, while others do not allow them.
- ◆ `asm` statements are accepted, like `asm("LD A, #5");`. This is disabled in strict ISO/ANSI C mode.
- ◆ Anonymous structs and unions (similar to the C++ anonymous unions) are allowed. An anonymous structure type defines an unnamed object (and not a type) whose member names are promoted to the surrounding scope. The member names must be unique in the surrounding scope. External anonymous structure types are allowed.
- ◆ External entities declared in other scopes are visible. A warning is issued. Example:

```
void f1(void) { extern void f(); }  
void f2() { f(); }
```
- ◆ End-of-line comments (`//`, as in C++) are allowed.
- ◆ A `non-lvalue` array expression is converted to a pointer to the first element of the array when it is subscripted or similarly used.

EXTENSIONS ACCEPTED IN NORMAL EC++ MODE

The following extensions are accepted in all modes (except when strict ANSI violations are diagnosed as errors):

- ◆ A friend declaration for a class may omit the `class` keyword:

```
class B;
class A {
    friend B;           // Should be 'friend class B'
};
```

- ◆ Constants of scalar type may be defined within classes (this is an old form; the modern form uses an initialized static data member):

```
class A {
    const int size = 10;
    int a[size];
};
```

- ◆ In the declaration of a class member, a qualified name may be used:

```
struct A {
    int A::f();         // Should be int f ();
};
```

- ◆ The preprocessing symbol `cplusplus` is defined in addition to the standard `_cplusplus`.

- ◆ An extension is supported to allow an anonymous union to be introduced into a containing class by a typedef name—it does not have to be declared directly, as with a true anonymous union. For example:

```
typedef union {
    int i, j;
} U;           // U identifies a reusable anonymous

// union.
class A {
    U;           // Okay -- references to A::i and
                // A::j are allowed.
}
```

In addition, the extension also permits ‘anonymous classes’ and ‘anonymous structures’, as long as they have no EC + + feature (for example, no static data members or member functions, and no non-public members) and have no nested types other than other anonymous classes, structures, or unions.

For example:

```
struct A {
    struct {
        int i, j;
    };
    // Okay -- references to A::i and
    // A::j are allowed.
};
```

- ◆ An assignment operator declared in a derived class with a parameter type matching one of its base classes is treated as a ‘default’ assignment operator—that is, such a declaration blocks the implicit generation of a copy assignment operator. For example:

```
struct A { };
struct B : public A {
    B& operator=(A&);
};
```

- ◆ By default, as well as in C front compatibility mode, there will be no implicit declaration of `B::operator=(const B&)`, whereas in strict ANSI mode `B::operator=(A&)` is *not* a copy assignment operator and `B::operator=(const B&)` is implicitly declared.

LANGUAGE FEATURES NOT ACCEPTED IN EC + +

The following ISO/ANSI C + + features are not accepted in EC + +:

- ◆ `reinterpret_cast` does not allow casting a pointer to a member of one class to a pointer to a member of another class if the classes are unrelated.
- ◆ In a reference of the form `f()->g()`, with `g` being a static member function, `f()` is not evaluated.
- ◆ Class name injection is not implemented.
- ◆ Friend functions of the argument class types cannot be found by name lookup on the function name in calls since this feature is not implemented.

- ◆ String literals do not have the `const` type.
- ◆ Universal character set escape sequences (for example, `\uabcd`) are not implemented.

DIAGNOSTICS

A normal diagnostic from the compiler is produced in the form:

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

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

SEVERITY LEVELS

The diagnostics are divided into different levels of severity:

Remark

A diagnostic that is produced when the compiler finds a source code construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued but can be enabled, see *--remarks*, page 56.

Warning

A diagnostic 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 using the command-line option *--no_warnings*, see *--no_warnings*, page 53.

Error

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

Fatal error

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

SETTING THE SEVERITY LEVEL

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

See the chapter *Compiler options*, page 37, for a description of the options that are available for setting severity levels.

See *Diagnostics*, page 88, for a description of the `#pragma` directives that are available for setting severity levels.

INTERNAL ERROR

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

Internal error: *message*

where *message* is an explanatory message. Internal errors should not occur and should be reported to your software distributor or IAR Technical Support. Your report should include all possible information about the problem and preferably, in electronic form, a minimal source file that generates the internal error.

EXAMPLE

The following examples show the style of the diagnostic messages:

[Og0001] Assembler error: string

Indicates erroneous syntax in an `asm` operation. The string specifies the actual error.

[Be0003] A located initialized variable must be constant: *full symbol name and place*

Specifying absolute address variables that must be initialized at start up is not allowed.

A

absolute location	12, 77
arrays	139
asm (inline assembler)	15
asm (intrinsic function)	13, 15
assembler	
inline	15
interface	105
assembler routines, calling from C	112
assert.h (library header file)	70
assumptions	iv

B

bitfields	3
in implementation-defined behavior	139
BITVARS (segment)	115

C

C library functions	146
C task functions	82
calling convention	108
Embedded C + +	112
cassert (library header file)	69
casting	5
cctype (library header file)	69
CDATA0 (segment)	115
CDATA1 (segment)	115
CDATA2 (segment)	115
cerrno (library header file)	69
cfloat (library header file)	69
char (data type)	2–3
signed and unsigned	41
characters, in implementation-defined behavior	136
climits (library header file)	69

clocale (library header file)	70
cmath (library header file)	70
code	
generation	2
pointers	5
code motion, disabling	51
CODE (segment)	116
coding, efficient	7
common sub-expression elimination, disabling	52
compiler	
code generation	2
features	1
introduction	1
language facilities	1
performance	1
target support	2
version number	95
compiler environment variables	39
compiler error return codes	39
compiler listing, generating	48
compiler object file	
including debug information	44, 55
compiler options	
setting	37
specifying parameters	38
summary	40
-A	106
-D	43
-e	46
-I	47
-l	48
-m	49, 95
-o	54
-R	55
-r	44, 55
-s	57
-U	58
-v	58–59, 95

INDEX

-z	60	CSTARTUP	32
--char_is_signed	41	modifying	33
--code_segment	42	cstartup.s48	111
--cpu	42, 95	cstdarg (library header file)	70
--debug	44, 55	cstddef (library header file)	70
--diag_error	44	cstdio (library header file)	70
--diag_remark	45	cstdlib (library header file)	70
--diag_suppress	45	cstring (library header file)	70
--diag_warning	45	ctime (library header file)	70
--ec + +	46	ctype.h (library header file)	70
--library_module	49	C-SPY	iii
--memory_model	49	C_INCLUDE (environment variable)	39, 47
--module_name	50		
--no_code_motion	51		
--no_cse	52	D	
--no_inline	52	data pointers	5
--no_unroll	53	data representation	2
--no_warnings	53	bitfields	3
--only_stdout	54	char	2–3
--preprocess	54	floating point format	3
--remarks	56	floating point types	3
--require_prototypes	56	int	3
--silent	57	integers	2
--strict_ansi	58	long	3
--warnings_affect_exit_code	39, 59	short	2
--warnings_are_errors	59	signed char	2
-2	61	signed int	3
complex (library header file)	68	signed long	3
configuration	17	signed short	2
constseg (#pragma directive)	88	unsigned char	2
CONST0 (segment)	116	unsigned int	3
CONST1 (segment)	116	unsigned long	3
CONST2 (segment)	116	unsigned short	2
conventions	v	dataseg (#pragma directive)	87
CPU, defining in .xcl file	22	debug information, including in object file	44, 55
csetjmp (library header file)	70	debugger	iii
csignal (library header file)	70	declarators, in implementation-defined behavior	140
CSTACK (segment)	116	default	

memory	19	environment variables	39
pointers	19	C_INCLUDE	39, 47
diagnostic messages		QCCM32C	39
classifying as errors	44	environment, in implementation-defined behavior	136
classifying as remarks	45	errno.h (library header file)	70
classifying as warnings	45	error	157
disabling warnings	53	fatal	157
enabling remarks	56	error return codes	39
suppressing	45	errors, classifying	44
diagnostics	157	exception (library header file)	68
error	157	extended keywords	12, 73
fatal error	157	absolute location	12, 77
internal error	158	C task functions	82
Og0001	158	enabling	46
remark	157	enum	3
severity levels	157	functions	79
warnings	157	in Embedded C + +	84
diagnostics (#pragma directives)	88	interrupt functions	80
diag_default (#pragma directive)	89	monitor functions	82
diag_error (#pragma directive)	89	near	5, 75, 78
diag_remark #pragma directive)	88	storage	74
diag_suppress (#pragma directive)	89	syntax	75
diag_warning #pragma directive)	89	__bitvar	73
<hr/>		__c_task	13, 74, 82
E		__far	5, 73, 75, 78
efficient coding	7	__fast_interrupt	13, 34, 73, 79
Embedded C + +		__huge	5, 73, 75
calling convention	112	__interrupt	12, 34, 73, 79–80, 118
differences from C + +	10	__monitor	13, 74, 79, 82
enabling	46	__near	73
extended keywords, using	84	__no_init	12, 73, 77
language extensions	10, 154	__regbank_interrupt	13, 34, 73, 79
overview	10	__sbddata	73
Embedded Workbench		__sbddata16	73
setting project options	18	__tiny_func	13, 74, 79
enum (keyword)	3	extended keywords, summary	73
enumerations, in implementation-defined behavior	139	extended linker command line file	21
		extensions. <i>See</i> language extensions	

F		
far (memory model)	19	
fatal error	157	
features, compiler	1	
file paths, specifying for #include files	47	
FLIST (segment)	117	
floating point (data type)		
specifying 64 bits	61	
floating-point		
format	3	
types	3	
floating-point format		
implementation-defined behavior	138	
float.h (library header file)	70	
formatters, specifying in .xcl file	23	
fstream (library header file)	68	
fstream.h (library header file)	71	
function inlining, disabling	52	
function pointers	5	
function prototypes, requiring	56	
functions		
I/O	29	
G		
getchar (library function)	25	
getchar, customizing	26	
H		
header files	64, 68	
SFR	7	
heap		
size	24	
HEAP (segment)	117	
hints		
		migration 121
		programming 7
		html (file format) 68
		huge (memory model) 19
I		
		IAR Assembler Reference Guide iii
		IAR C-SPY Debugger iii
		IAR Embedded Workbench
		reference information iii
		User Guide iii
		IDATA0 (segment) 117
		IDATA1 (segment) 117
		IDATA2 (segment) 117
		identifiers, in implementation-defined behavior 136
		implementation-defined behavior 135
		inheritance, in Embedded C + + 10
		initialization 32
		inline assembler 15
		input 25
		input functions, in standard library 29
		installation iii
		int (data type) 3
		integer types 2
		integers
		in implementation-defined behavior 138
		internal error 158
		Internet
		browser 68
		interrupt
		functions 80, 110
		handling 110
		handling segments, declaring 23
		vector table 34
		vectors, defining 111
		intrinsic functions

asm	13, 15	iostream (library header file)	69
summary	13	iostream.h (library header file)	71
__break_instruction	13, 97	ISO/ANSI	
__disable_interrupt	13, 97	C + + standard	10
__enable_interrupt	14, 97	specifying strict usage	58
__interrupt_on_overflow	14, 98	ISO/ANSI prototypes	7
__intrinsic_load_DCT	14, 98	iso646.h (library header file)	70
__intrinsic_load_DMA	14, 98	ISTACK (segment)	119
__intrinsic_load_DMD	14, 98	istream (library header file)	69
__intrinsic_load_DRA	14, 99	I/O functions	29
__intrinsic_load_DRC	14, 99	customizing	30–31
__intrinsic_load_DSA	14, 99		
__intrinsic_load_VCT	14, 99		
__intrinsic_store_DCT	14, 100	K	
__intrinsic_store_DMA	14, 100	keywords, extended	73
__intrinsic_store_DMD	14, 100		
__intrinsic_store_DRA	14, 100		
__intrinsic_store_DRC	14, 101	L	
__intrinsic_store_DSA	14, 101	language	
__intrinsic_store_VCT	14, 101	extensions, summary	11
__no_operation	15, 101	facilities	1
__overflow_flag_value	15, 102	language extensions	
__read_ipl	15, 102	Embedded C + +	10
__rmpa_instruction	15, 102	enabling	46
__set_interrupt_table	15, 102	reference information	151
__short_rmpa_instruction	15, 103	language (#pragma directive)	89
__software_interrupt	15, 103	library documentation	68
__und_instruction	15, 103	library functions	67, 143
__wait_for_interrupt	15, 104	getchar	25
__write_ipl	15, 104	header files	68
INTVEC (segment)	111, 118	in implementation-defined behavior	146
INTVEC1 (segment)	111, 118	object files	67
iomacros.h	7	printf	27
iomanip (library header file)	68	putchar	25
iomanip.h (library header file)	71	remove	29
iom32c.h	7	rename	29
ios (library header file)	69	scanf	28
iosfwd (library header file)	69		

sprintf	27	model, choosing	19
sscanf	28	model, specifying	20
summary	64, 68	non-initialized	12
__close	29	usage, application	9
__lseek	29	memory models	
__open	29	far	19
__read	29	huge	19
__readchar	29	near	19
__write	29	specifying	49
__writechar	29	migration	121
library module, creating	49	module name, specifying	50
library, run-time	24	monitor functions	82, 111
limits.h (library header file)	70		
linker command file	21		
allocating writable segments	22	N	
contents	22	name, specifying for object file	54
declaring interrupt handling		NDATA0 (segment)	119
segment	23	NDATA1 (segment)	119
defining the CPU	22	NDATA2 (segment)	119
input and output formatters	23	near (memory model)	19
modifying	22	new (library header file)	69
specifying	21	new.h (library header file)	71
template	22	non-initialized memory	12
listing, generating	48	non-volatile RAM	21
locale.h (library header file)	70	NO_INIT (segment)	21
location (#pragma directive)	88		
long (data type)	3	O	
loop unrolling, disabling	53	object attribute (#pragma directive)	86
		object filename, specifying	54
M		object module name, specifying	50
math.h (library header file)	70	Og0001 (diagnostics)	158
member functions	113	optimization	
calling convention	84	code motion, disabling	51
in Embedded C + +	113	common sub-expression elimination, disabling	52
member variables	84	function inlining, disabling	52
memory		loop unrolling, disabling	53
default	19		

165 —

INDEX

return values	109	UDATA2	119
RTMODEL	112	setjmp.h (library header file)	70
run-time		severity level	157
library	24	specifying	158
model	112	SFR	7
		short (data type)	2
		signal.h (library header file)	70
		signed char (data type)	2
		specifying	42
		signed int (data type)	3
		signed long (data type)	3
		signed short (data type)	2
		silent operation, specifying	57
		size optimization	
		specifying	60
		size_t	5
		special function register	7
		speed optimization	
		specifying	57
		sprintf (library function)	27
		sscanf (library function)	28
		sstream (library header file)	69
		stack	
		size	24
		stack frames	110
		standard error	54
		standard output, specifying	54
		statements, in implementation-defined behavior	141
		stdarg.h (header file)	95
		stdarg.h (library header file)	70
		stddef.h (library header file)	70
		stderr	30, 54
		stdexcept (library header file)	69
		stdin	30
		stdio.h (library header file)	70
		stdlib.h (library header file)	71
		stdout	30, 54
		storage	12
SBDATA (segment)	119		
SBDATA16 (segment)	119		
scanf (library function)	28		
search procedure, #include files	47		
segments	115		
BITVARS	115		
CDATA0	115		
CDATA1	115		
CDATA2	115		
CODE	116		
CONST0	116		
CONST1	116		
CONST2	116		
CSTACK	116		
FLIST	117		
HEAP	117		
IDATA0	117		
IDATA1	117		
IDATA2	117		
INTVEC	118		
INTVEC1	118		
ISTACK	119		
NDATA0	119		
NDATA1	119		
NDATA2	119		
NO_INIT	21		
SBDATA	119		
SBDATA16	119		
UDATA0	119		
UDATA1	119		

storage (extended keywords)	74	vector (#pragma directive)	88
streambuf (library header file)	69		
string (library header file)	69		
string.h (library header file)	71	W	
strstream (library header file)	69	warnings	157
structures		classifying	45
in implementation-defined behavior	139	disabling	53
symbols		exit code	59
preprocessor, defining	43	treating as errors	59
syntax		wchar.h (library header file)	71
extended keywords	75	wctype.h (library header file)	71
		writable segments, allocating in .xcl file	22
		write formatter, selecting	28
		writechar.c	30
T			
target identifier	94	X	
target support	2		
time.h (library header file)	71	XLINK command file	21
translation, in implementation-defined behavior	135	XLINK options, -A	26
tutorials	iii		
type attribute (#pragma directive)	85		
		Symbols	
U		#include file paths, specifying	47
UDATA0 (segment)	119	#include files, search procedure	47
UDATA1 (segment)	119	#pragma directives	12, 85
UDATA2 (segment)	119	constseg	88
unions		dataseg	87
in implementation-defined behavior	139	diagnostics	88
unsigned char (data type)	2	diag_default	89
changing to signed char	42	diag_error	89
unsigned int (data type)	3	diag_remark	88
unsigned long (data type)	3	diag_suppress	89
unsigned short (data type)	2	diag_warning	89
		language	89
V		location	88
variables, uninitialized	77	object attribute	86
		optimize	90

pack	90	--require_prototypes (compiler option)	56
type_attribute	85	--silent (compiler option)	57
vector	35, 88	--strict_ansi (compiler option)	58
-A (compiler option)	106	--warnings_affect_exit_code (compiler option)	39, 59
-A (XLINK option)	26	--warnings_are_errors (compiler option)	59
-D (compiler option)	43	-2 (compiler option)	61
-e (compiler option)	46	.xcl file	21
-I (compiler option)	47	__close (library function)	29
-l (compiler option)	48	__lseek (library function)	29
-m (compiler option)	49, 95	__open (library function)	29
-md (M16/C compiler option)	131	__read (library function)	29
-o (compiler option)	54	__readchar (library function)	29
-R (compiler option)	55	__write (library function)	29
-r (compiler option)	44, 55	__writechar (library function)	29
-s (compiler option)	57	__formatted_write (library function)	27
-U (compiler option)	58	__medium_write (library function)	27
-v (compiler option)	58–59, 95	__small_write (library function)	27
-z (compiler option)	60	__bitvar (extended keyword)	73
--char_is_signed (compiler option)	41	__break_instruction (intrinsic function)	13, 97
--code_segment (compiler option)	42	__c_task (extended keyword)	13, 74, 82
--cpu (compiler option)	42, 95	__DATE__ (predefined symbol)	93
--debug (compiler option)	44, 55	__disable_interrupt (intrinsic function)	13, 97
--diag_error (compiler option)	44	__enable_interrupt (intrinsic function)	14, 97
--diag_remark (compiler option)	45	__far (extended keyword)	5, 73, 75, 78
--diag_suppress (compiler option)	45	__fast_interrupt (extended keyword)	13, 34, 73, 79
--diag_warning (compiler option)	45	__FILE__ (predefined symbol)	93
--ec + + (compiler option)	46	__huge (extended keyword)	5, 73, 75
--library_module (compiler option)	49	__IAR_SYSTEMS_ICC__ (predefined symbol)	93
--memory_model (compiler option)	49	__interrupt (extended keyword)	12, 34, 73, 79–80, 118
--module_name (compiler option)	50	__interrupt_on_overflow (intrinsic function)	14, 98
--no_code_motion (compiler option)	51	__intrinsic (IAR keyword)	83
--no_cse (compiler option)	52	__intrinsic_load_DCT (intrinsic function)	14, 98
--no_inline (compiler option)	52	__intrinsic_load_DMA (intrinsic function)	14, 98
--no_unroll (compiler option)	53	__intrinsic_load_DMD (intrinsic function)	14, 98
--no_warnings (compiler option)	53	__intrinsic_load_DRA (intrinsic function)	14, 99
--only_stdout (compiler option)	54	__intrinsic_load_DRC (intrinsic function)	14, 99
--preprocess (compiler option)	54		
--remarks (compiler option)	56		

__intrinsic_load_DSA (intrinsic function)	14, 99
__intrinsic_load_VCT (intrinsic function)	14, 99
__intrinsic_store_DCT (intrinsic function)	14, 100
__intrinsic_store_DMA (intrinsic function)	14, 100
__intrinsic_store_DMD (intrinsic function)	14, 100
__intrinsic_store_DRA (intrinsic function)	14, 100
__intrinsic_store_DRC (intrinsic function)	14, 101
__intrinsic_store_DSA (intrinsic function)	14, 101
__intrinsic_store_VCT (intrinsic function)	14, 101
__LINE__ (predefined symbol)	94
__low_level_init	33
__monitor (extended keyword)	13, 74, 79, 82
__near (extended keyword)	5, 73, 75, 78
__no_init (extended keyword)	12, 73, 77
__no_operation (intrinsic function)	15, 101
__overflow_flag_value (intrinsic function)	15, 102
__read_ipl (intrinsic function)	15, 102
__regbank_interrupt (extended keyword)	13, 34, 73, 79
__rmpa_instruction (intrinsic function)	15, 102
__sbdata (extended keyword)	73
__sbdata16 (extended keyword)	73
__set_interrupt_table (intrinsic function)	15, 102
__short_rmpa_instruction (intrinsic function)	15, 103
__software_interrupt (intrinsic function)	15, 103
__STDC__ (predefined symbol)	94
__STDC__VERSION__ (predefined symbol)	94
__TID__ (predefined symbol)	94
__TIME__ (predefined symbol)	95
__tiny_func (extended keyword)	13, 74, 79
__und_instruction (intrinsic function)	15, 103
__VER__ (predefined symbol)	95
__wait_for_interrupt (intrinsic function)	15, 104
__write_ipl (intrinsic function)	15, 104

Numerics

64-bit floating point, specifying	61
-----------------------------------	----

