SAM8 IAR C Compiler

Reference Guide

for Samsung's

SAM8 Microcontroller Family

COPYRIGHT NOTICE

© Copyright 1997–2003 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, IAR Embedded Workbench, IAR XLINK Linker, IAR XAR Library Builder, IAR XLIB Librarian, IAR MakeApp, and IAR PreQual are trademarks owned by IAR Systems. C-SPY is a trademark registered in Sweden by IAR Systems. IAR visualSTATE is a registered trademark owned by IAR Systems.

SAM8 and Samsung are registered trademarks of Samsung Electronics Co., Ltd.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

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

EDITION NOTICE

Second edition: April 2003

Part number: CSAM8-2

Contents

Tables xi

Preface xiii

Who should read this guide xiii
How to use this guide xiii
What this guide contains xiv
Other documentation xv
Further reading xv
Document conventions xvi
Typographic conventions xvi

Part 1. Using the compiler 1

Introduction 3

Building applications 3
Compiling 3
Linking 3
Data storage 4
Code models 4
Optimization techniques 4
IAR language extension overview 5
Special function types 5
Extended keywords 5
#pragma directives 5
Predefined symbols 5
Intrinsic functions 6
Inline assembler 6
Runtime libraries 6

Customization 7

Code model 7
Data model 7

Runtime library ... 8

Data storage ... 11

Stack, static, and heap memory ... 11

The stack and auto variables ... 11

Static memory ... 13

Dynamic memory on the heap ... 14

Memory access methods and memory types ... 14

Memory access methods ... 14

Memory types ... 15

Structures and memory types ... 16

Non-initialized memory ... 16

Located variables ... 17

Absolute location placement ... 17

Segment placement ... 17

Accessing special function registers ... 18

Anonymous structs and unions ... 18

Functions ... 21

Code models ... 21

Special function types ... 21

Interrupt functions ... 21

Fast functions ... 22

Monitor functions ... 22

Segment placement ... 23

Assembler language interface ... 25

Introduction ... 25

Runtime model attributes ... 25

Specifying runtime attributes ... 26

Predefined runtime attributes ... 27

Calling convention ... 27

Function declarations ... 27

Function parameters ... 28

Returning a value from a function ... 29

Permanent versus scratch registers 29
Return location 29
Examples 30
Calling assembler routines from C 31
Creating skeleton code 31
Call frame information 33
Function directives 35
Syntax 35
Parameters 35
Description 36
Segments and memory 37
What is a segment? 37
Linker segment type 38
Placeholder segments 38
Placing segments in memory 38
The contents of the linker configuration file 39
Customizing a linker configuration file 39
Data segments 40
Static memory segments 40
The stack 43
The heap 44
Located data 44
Code segments 45
Startup code 45
Normal code 45
Exception vectors 45
Runtime environment 47
The cstartup.s18 file 47
System startup 47
System termination 48
__low_level_init 48

Customizing cstartup.s18 48
Modules and segment parts 49
Call frame information 50
Modifying the cstartup.s18 file 50
Input and output 51
Library object files 52
Header files 52
Library definitions summary 52
C-SPY debugger interface 53
The debugger terminal I/O window 53
Programming hints 55
General programming hints 55
Function prototypes 55
Bitfields 56
Arrays 56
Floating-point types 56
Saving stack space and RAM memory 56
Optimization techniques 56
Specifying the optimization type and level 57
Optimization hints 57
Part 2. Compiler reference 59
Data representation 61
Alignment 61
Data types 61
Integer types 61
Floating-point types 62
Pointers 63
Size 63
Casting 63

Structure types ... 64
Alignment ... 64
General layout ... 64
Segment reference ... 65
Summary of segments ... 65
Descriptions of segments ... 66
Compiler options ... 75
Setting command line options ... 75
Specifying parameters ... 76
Specifying environment variables ... 77
Error return codes ... 77
Options summary ... 77
Descriptions of options ... 79
Extended keywords ... 99
Summary of extended keywords ... 99
Using extended keywords ... 100
Data storage ... 100
Functions ... 101
Descriptions of extended keywords ... 102
#pragma directives ... 109
Summary of #pragma directives ... 109
Descriptions of #pragma directives ... 110
Predefined symbols ... 119
Summary of predefined symbols ... 119
Descriptions of predefined symbols ... 120
Intrinsic functions ... 123
Intrinsic functions summary ... 123
Descriptions of intrinsic functions ... 124
Library functions ... 127
Introduction ... 127

IAR CLIB library 127
Library object files 127
Header files 128
Library definitions summary 128
Restrictions on ANSI C libraries 129
Diagnostics 131
Message format 131
Severity levels 131
Setting the severity level 132
Internal error 132
Part 3. Migration and portability 133
Migrating to the SAM8 IAR C Compiler V2.x 135
Differences 135
Implementation-defined behavior 139
Descriptions of implementation-defined behavior 139
Translation 139
Environment 140
Identifiers 140
Characters 140
Integers 141
Floating point 142
Arrays and pointers 143
Registers 143
Structures, unions, enumerations, and bitfields 143
Qualifiers 144
Declarators 144
Statements 144
Preprocessing directives 144
C library functions 146

IAR C extensions 151

Why should language extensions be used? 151

Descriptions of language extensions 151

Index 163

Tables

1: Typographic conventions used in this guide xvi
2: Code models 7
3: Data models 8
4: Runtime libraries 9
5: Memory types and keywords 13
6: Memory types 15
7: Example of runtime model attributes 26
8: Runtime model attributes 27
9: Register use in different code models 30
10: Call frame information 33
11: XLINK segment types 38
12: Linker configuration file example 39
13: Memory types 41
14: Segment groups 41
15: Segments in segment groups 42
16: Preprocessor symbols for code and data models 51
17: IAR C Library header files 52
18: Miscellaneous IAR C Library header files 53
19: Integer types 61
20: Floating-point types 62
21: Segment summary 65
22: Environment variables 77
23: Error return codes 77
24: Compiler options summary 77
25: Available code models 80
26: Available data models 81
27: Generating a list of dependencies (--dependencies) 82
28: Generating a compiler list file (-l) 88
29: Directing preprocessor output to file (--preprocess) 93
30: Specifying speed optimization (-s) 95
31: Specifying size optimization (-z) 96

32: Extended keywords summary 99
33: #pragma directives summary 109
34: Predefined symbols summary 119
35: Intrinsic functions summary 123
36: IAR CLIB Library header files 128
37: Miscellaneous IAR CLIB Library header files 128
38: #pragma directives in V1.x with new syntax 135
39: Assembler processor option mappings 136
40: Compiler processor option mappings 136
41: Compiler memory model option mappings 137
42: Other compiler option mappings 137
43: Message returned by strerror() 148

Preface

Welcome to the SAM8 IAR C Compiler Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use the SAM8 IAR C Compiler to best suit your application requirements. This guide also gives you suggestions on coding techniques so that you can develop applications with maximum efficiency.

Who should read this guide

You should read this guide if you plan to develop an application using the C language for the SAM8 microcontroller and need to get detailed reference information on how to use the SAM8 IAR C Compiler. In addition, you should have a working knowledge of the following:

- The architecture and instruction set of the SAM8 microcontroller. Refer to the documentation from Samsung for information about the SAM8 microcontroller
- The C programming language
- Application development for embedded systems
- The operating system of your host machine.

How to use this guide

When you start using the SAM8 IAR C Compiler, you should read *Part 1. Using the compiler* in this reference guide.

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

If you are new to using the IAR toolkit, we recommend that you first read the initial chapters of the *SAM8 IAR Embedded Workbench™ IDE User Guide*. They give product overviews, as well as tutorials that can help you get started. The *SAM8 IAR Embedded Workbench™ IDE User Guide* also contains a glossary.

What this guide contains

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

Part 1. Using the compiler

- *Introduction* gives an overview of the compiler techniques that allow an application to take full advantage of the SAM8 microcontroller: code and data storage features, optimization techniques, and language extensions.
- *Customization* describes the available customization options: code model, data model, and runtime libraries.
- *Data storage* describes how data can be stored in memory, with an emphasis on the different memory types.
- *Functions* describes the different ways code can be generated and introduces the concept of code models. Special function types such as interrupt functions are also covered.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention and the runtime model attributes.
- *Segments and memory* describes the concept of segments, introduces the linker configuration file, and describes how code and data are placed in memory.
- *Runtime environment* describes system initialization, introduces the `cstartup` file, and describes some low-level I/O routines in the runtime library.
- *Programming hints* gives hints about programming for the SAM8 IAR C Compiler.

Part 2. Compiler reference

- *Data representation* describes the available data types, pointers, and structure types.
- *Segment reference* gives reference information about the compiler's use of segments.
- *Compiler options* explains how to set the compiler options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Extended keywords* gives reference information about each of the SAM8-specific keywords that are extensions to the standard C language.
- *#pragma directives* gives reference information about the `#pragma` directives.
- *Predefined symbols* gives reference information about the predefined preprocessor symbols.
- *Intrinsic functions* gives reference information about the functions that can use SAM8-specific low-level features.
- *Library functions* gives an introduction to the C library functions, and summarizes the header files.
- *Diagnostics* describes how the compiler's diagnostic system works.

Part 3. Migration and portability

- *Migrating to the SAM8 IAR C Compiler V2.x* gives hints for porting code written for a version V1.x of the SAM8 IAR 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.

Other documentation

The complete set of IAR Systems development tools for the SAM8 microcontroller is described in a series of guides. For information about:

- Using the IAR Embedded Workbench™ IDE with the IAR C-SPY™ Debugger, refer to the *SAM8 IAR Embedded Workbench™ IDE User Guide*
- Programming for the SAM8 IAR Assembler, refer to the *SAM8 IAR Assembler Reference Guide*
- Using the IAR XLINK Linker™, the IAR XAR Library Builder™, and the IAR XLIB Librarian™, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR C Library, refer to the *IAR C Library Functions Reference Guide*, available from the SAM8 IAR Embedded Workbench IDE **Help** menu.

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

FURTHER READING

The following books may be of interest to you when using the IAR Systems development tools:

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

We recommend that you visit the websites of Samsung and IAR Systems:

- The Samsung website, **www.samsung.com**, contains information and news about the SAM8 microcontrollers.
- The IAR website, **www.iar.com**, holds application notes and other product information.

Document conventions

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

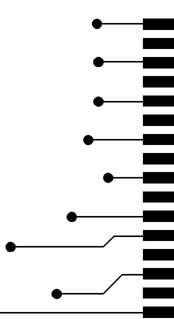
Style	Used for
`computer`	Text that you enter or that appears on the screen.
`parameter`	A label representing the actual value you should enter as part of a command.
`[option]`	An optional part of a command.
`{a | b | c}`	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog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

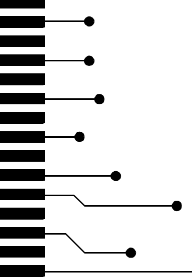
Table 1: Typographic conventions used in this guide

Part 1. Using the compiler

This part of the SAM8 IAR C Compiler Reference Guide includes the following chapters:

- Introduction
- Customization
- Data storage
- Functions
- Assembler language interface
- Segments and memory
- Runtime environment
- Programming hints.





Introduction

The SAM8 IAR C Compiler supports the C language for Samsung's SAM8 microcontroller family.

This chapter first introduces the concepts of compiling and linking when describing how an application is built.

Then the compiler is introduced, including an overview of the techniques that enable applications to take full advantage of the SAM8 microcontroller. In the following chapters these techniques will be studied in more detail.

Building applications

A typical application is built from a number of source files and libraries. The source files could be written in C, or assembler language and can be compiled into object files by the SAM8 IAR C Compiler or the SAM8 IAR Assembler.

A library is a collection of object files. A typical example of a library is the compiler library containing the runtime environment and the C standard library. Libraries can also be built using the IAR XAR Library Builder, the IAR XLIB Librarian, or be provided by external suppliers.

The IAR XLINK Linker is used for building the final application. XLINK normally uses a linker configuration file describing the available resources of the target system.

COMPILING

In the command line interface, the following line compiles the source file `myfile.c` into the object file `myfile.r18` using the default settings:

```
iccsam8 myfile.c
```

LINKING

The IAR XLINK Linker is used to build the final application. Normally XLINK requires the following:

- A number of object files and possibly some libraries
- The standard library containing the runtime environment and the standard language functions
- A linker configuration file that describes the memory layout of the target system.

In the IAR Embedded Workbench, XLINK is started automatically when you choose the **Build** option.

In the command line interface, the following line can be used to start XLINK:

```
xlink myfile.r18 myfile2.r18 -f lnksam8.xcl clsam8ss.r18
```

In this example, `myfile.r18` and `myfile2.r18` are object files, `lnksam8.xcl` is the linker configuration file, and `clsam8ss.r18` is the runtime library.

Data storage

One of the characteristics of the SAM8 microcontroller is that there is a trade-off regarding the way memory is accessed, ranging from cheap access to small memory areas up to more expensive access methods that can access any location.

One of the decisions a developer of embedded systems must make is to decide where the different memory access methods should be used.

The SAM8 IAR C Compiler allows you to set a default memory access method by using data models. The compiler also allows the access method to be specified explicitly for each individual variable.

The *Data storage* chapter covers memory access methods in greater detail.

Code models

The SAM8 IAR C Compiler supports the small and large *code models*.

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

Optimization techniques

The SAM8 IAR C Compiler is a state-of-the-art compiler with a C level optimizer that performs, among other things, dead-code elimination, constant propagation, inlining, common sub-expression elimination, and precision reduction. It also performs loop optimizations such as unrolling and induction variable elimination.

The user can control the level of optimization and decide if the basic approach is to optimize for speed or for size. It is also possible to disallow individual optimizations.

For more information about optimization, see the chapter *Programming hints*.

IAR language extension overview

This section briefly describes the extensions provided by the SAM8 IAR C Compiler to support specific features of the SAM8 microcontroller.

SPECIAL FUNCTION TYPES

The special hardware features of the SAM8 microcontroller are supported by the compiler's special function types: interrupt, fast, and monitor. These allow you to write a complete application without having to write any part of it in assembler language.

For detailed information, see *Special function types*, page 21.

EXTENDED KEYWORDS

The SAM8 IAR C Compiler provides a set of keywords that can be used to control the behavior of the program. There are, for example, keywords for controlling the memory type for individual variables as well as for declaring special function types.

By default language extensions are always enabled in the IAR Embedded Workbench.

The command line option `-e` makes the extended keywords available, and reserves them so that they cannot be used as variable names. See page 85 for additional information.

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

#PRAGMA DIRECTIVES

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

The `#pragma` directives are always enabled in the SAM8 IAR C Compiler. They are consistent with the ISO/ANSI C and are very useful when you want to make sure that the source code is portable.

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

PREDEFINED SYMBOLS

With the predefined preprocessor symbols, you can inspect your compile-time environment, for example the code and data models.

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

INTRINSIC FUNCTIONS

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into in-line code, either as a single instruction or as a short sequence of instructions.

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

INLINE ASSEMBLER

The `asm` keyword assembles and inserts the supplied assembler statement in-line, for example:

```
asm("LD R0,R1");
```

Note: The `asm` keyword reduces the compiler's ability to optimize the code. We recommend the use of modules written in assembler language instead of inline assembler, since the function call to an assembler routine causes less performance reduction.

Runtime libraries

The SAM8 IAR C Compiler supports the IAR CLIB Library, which is a small, efficient library well-suited for 8- and 16-bit processors. This library is not fully compliant with ISO/ANSI C, and does not fully support IEEE 754 floating-point numbers.

Customization

This chapter covers the configuration of the SAM8 IAR C Compiler including an overview of the available code and data models. The last section describes the standard runtime libraries that are included and how they correspond to the compiler options.

You should read this chapter before you read the remaining chapters in *Part 1. Using the compiler* and the chapters in *Part 2. Compiler reference*.

Code model

The code model specifies the way in which code is generated and called. All object files in an application must use the same code model.

The following code models are available:

Code model	Max. stack size	Description
Small (default)	256 bytes	Internal stack
Large	64 Kbyte	External stack Not for SAM8xRI or SAM8xRCRI

Table 2: Code models

See the chapter *General options* in the *SAM8 IAR Embedded Workbench™ IDE User Guide* for information about setting options in the IAR Embedded Workbench.

Use the `--code_model` option to specify the code model for your project; see *--code_model*, page 79.

Data model

The data model specifies the data memory which is used for storing:

- Non-stacked variables, that is, global data and variables declared as `static`
- Dynamically allocated data, for example data, allocated with `malloc`.

Note that if the data model and the code model differ, the default pointer will not be able to point to stack objects.

The following table summarizes the characteristics of the different data models:

Data model	Default data memory attribute	Default data pointer	Description
Small (default)	`__tiny`	`__tinyp`	Internal RAM
Large	`__near`	`__near`	External RAM

Table 3: Data models

Your program can only use one data model at a time, and the same model must be used by all user modules and all library modules. If you do not specify a data model option, the compiler will use the small data model.

The default memory attribute can—for each individual variable—be overridden by the use of extended keywords or `#pragma` directives.

See the *SAM8 IAR Embedded Workbench™ IDE User Guide* for information about setting options in the IAR Embedded Workbench.

Use the `--data_model` option to specify the data model for your project; see *--data_model*, page 81.

Runtime library

The runtime library includes the runtime environment and the C standard library. The linker will include only those routines that are required—directly or indirectly—by your application.

When building an application all parts must use the same customization settings. This also applies to the runtime library. For the SAM8 IAR C Compiler this means that there is a runtime library for each combination of data and code models.

The runtime library names are constructed in the following way:

```
<type><cpu_variant><code_model><data_model><eeprom_support>.r18
```

where

- *<type>* is `cl` for the IAR CLIB library
- *<cpu_variant>* is `sam8/sam8x/sam8xri`
- *<code_model>* is one of `s` or `l` for small or large code
- *<data_model>* is one of `s` or `l` for small or large data
- *<eeprom_support>* is `e` if EEPROM support is enabled.

The following table shows the mapping of runtime libraries, cores, code models, data models, and EEPROM support:

Library	Core	Code model	Data model	EEPROM support
clsam8ss.r18	sam8	Small	Small	No
clsam8xss.r18	sam8x / sam8xrc	Small	Small	No
clsam8xriss.r18	sam8xri / sam8xrcri	Small	Small	No
clsam8sse.r18	sam8	Small	Small	Yes
clsam8xsse.r18	sam8x / sam8xrc	Small	Small	Yes
clsam8xrisse.r18	sam8xri / sam8xrcri	Small	Small	Yes
clsam8ll.r18	sam8	Large	Large	No
clsam8xll.r18	sam8x / sam8xrc	Large	Large	No
clsam8xrill.r18	sam8xri / sam8xrcri	Large	Large	No
clsam8lle.r18	sam8	Large	Large	Yes
clsam8xlle.r18	sam8x / sam8xrc	Large	Large	Yes
clsam8xrille.r18	sam8xri / sam8xrcri	Large	Large	Yes
clsam8sl.r18	sam8	Small	Large	No
clsam8xsl.r18	sam8x / sam8xrc	Small	Large	No
clsam8xrisl.r18	sam8xri / sam8xrcri	Small	Large	No
clsam8sle.r18	sam8	Small	Large	Yes
clsam8xsle.r18	sam8x / sam8xrc	Small	Large	Yes
clsam8xrisle.r18	sam8xri / sam8xrcri	Small	Large	Yes
clsam8ls.r18	sam8	Large	Small	No
clsam8xls.r18	sam8x / sam8xrc	Large	Small	No
clsam8lse.r18	sam8	Large	Small	Yes
clsam8xlse.r18	sam8x / sam8xrc	Large	Small	Yes

Table 4: Runtime libraries

Data storage

This chapter starts by describing the fundamental ways data can be stored in memory: on the stack, in static (global) memory, or in heap memory. Then the different memory access methods and corresponding memory types are described.

Memory types are discussed in relation to pointers, structures, and non-initialized memory. Then placement in memory of global and static variables is described. Finally, the structure types struct and union are discussed.

Stack, static, and heap memory

Data can be stored in memory in three different ways:

- On the stack. This is memory space that can be used by a function as long as it is executing. When the function returns to its caller, the memory space is no longer valid.
- In static memory. This kind of memory is allocated once and for all; it remains valid all through the execution of the application. Variables that are either global or declared static are placed in this kind of memory.
- On the heap. Once memory has been allocated on the heap it remains valid until it is explicitly released back to the system by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that there are potential risks connected with using the heap in systems with a limited amount of memory or systems that are expected to run for a long time.

THE STACK AND AUTO VARIABLES

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

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

The stack can contain:

- Local variables and parameters not stored in registers
- Temporary results of expressions
- The return value of functions (unless it is passed in registers)
- Processor state during interrupts
- Processor registers that should be restored before the function returns (callee-save registers).

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

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

Advantages

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

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

Potential problems

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

```
int * MyFunction()
{
  int x;
  ... do something ...
  return &x;
}
```

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

STATIC MEMORY

All global and static variables will be placed in static memory. The word "static" in this context means that the amount of memory allocated for this type of variables does not change while the application is running.

The SAM8 microcontroller can access memory in different ways. The access methods range from generic but expensive methods that can access the full memory space, to cheap methods that can access limited memory areas.

The following memory types and corresponding keywords exist:

Memory type	Max. object size	Name
IRAM Page 0	128 bytes	`__tiny`
IRAM Page 0	64 bytes	`__tiny2`
IRAM Page 1-15, cannot be initialized	128 bytes	`__tinyp`
IRAM Page 1-15, cannot be initialized	64 bytes	`__tiny2p`
IRAM Page *n*=1-15, cannot be initialized	128 bytes	`__tinyp`*n*
IRAM Page *n*=1-15, cannot be initialized	64 bytes	`__tiny2p`*n*
XRAM		`__near`
SFR area n=0,1		`__bank`*n*
ROM		`__code`

Table 5: Memory types and keywords

A variable can be placed in a non-default memory area by declaring it using extended keywords or `#pragma` directives, as in these examples:

```
__tiny int x;

#pragma type_attribute=__near
int y;
```

See *Memory access methods and memory types*, page 14, for a description of the limitations and advantages of each of these methods.

DYNAMIC MEMORY ON THE HEAP

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

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

Potential problems

Systems that are using heap-allocated objects must be designed very carefully, since it is easy to end up in a situation where it is not possible to allocate objects on the heap, either because there is not enough free memory on the heap or because it has become fragmented.

The heap can become exhausted because the system simply uses too much memory. It can also become full if memory that no longer is in use has not been released back to the system.

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

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

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

Memory access methods and memory types

This section describes the concept of access methods and the corresponding memory types used by the SAM8 IAR C Compiler to access data. For each memory type the capabilities and limitations are discussed.

MEMORY ACCESS METHODS

The SAM8 microcontroller has three separate memory spaces. Data memory, which can be accessed efficiently, code memory, and external RAM, which requires more code space and execution time to access. Code memory is only used for const-declared variables, string literals, and initializer data. In the small data model, variables are

placed in data memory, and the resulting code is faster and more compact. Const-declared variables can be placed in most memories by combining the `const` keyword with a memory specifier, e.g. `const __near int a=34;` Not all memory specifiers are allowed.

The data memory for SAM8 is divided into different zones, depending on access methods.

The code memory is a 64 kbytes large memory used for code, constants, and initializer data. Data placed in code memory is accessed using `LDC` instructions. This type of access is much slower than memory access in data memory, and should generally be avoided to gain speed for the application.

Example

The example below defines three variables—`alpha`, `beta`, and `gamma`—to be placed in near, tiny, and in the default memory type, respectively. Note that the `#pragma` directive only controls the memory placement of the next defined variable.

```
int __near alpha;
#pragma type_attribute=__tiny
int beta;
int gamma;
```

MEMORY TYPES

Name	Address range	Object size	Pointer size	Description
`__tiny`	0x00-0xBF	128 bytes	1 byte	IRAM Page 0
`__tiny2`	0xC0-0xFF	64 bytes	1 byte	IRAM Page 0
`__tinyp`	0x00-0xBF	128 bytes	2 bytes	IRAM Page 1-15 Cannot be initialized
`__tiny2p`	0xC0-0xFF	64 bytes	2 bytes	IRAM Page 1-15 Cannot be initialized
`__tinyp`*n*	0x00-0xBF	128 bytes	1 byte	IRAM Page *n*=1-15
`__tiny2p`*n*	0xC0-0xFF	64 bytes	1 byte	IRAM Page *n*=1-15
`__near`	0-0xFFFF	32 Kbytes	2 bytes	XRAM
`__bank`*n*	0xC0-0xFF		1 byte	SFR area n=0,1
`__code`	0-0xFFFF	32 Kbytes	2 bytes	ROM
`__generic`	0x000000-0x000FFF 0x010000-0x01FFFF 0x020000-0x02FFFF		3 bytes	IRAM XRAM ROM Pointer only

Table 6: Memory types

The chapter *Assembler language interface* covers this in more detail.

Structures and memory types

When a variable is defined, it will be placed in a memory of a certain type. Normally the default memory type is used but another memory type can be specified. For structures, the entire object is placed in the same memory type. It is not possible to place individual structure members in different memory types.

Example

In the example below, the variable `gamma` is a structure placed in near memory.

```
struct MyStruct
{
  int alpha;
  int beta;
};
__near struct MyStruct gamma;
```

The following declaration is incorrect:

```
struct MySecondStruct
{
  int blue;
  __near int green;   /* Error! */
};
```

Non-initialized memory

Normally the runtime environment will initialize all global and static variables when the application is started.

The compiler supports the declaration of variables that will not be initialized using the `__no_init` type modifier. They can be specified either as a keyword or using the `#pragma object_attribute` directive. The compiler places such variables in separate segments, according to the specified memory keyword. See the chapter *Segments and memory* for more information.

For `__no_init`, the `const` keyword implies that an object is read only, rather than that the object is stored in read-only memory. It is not possible to give a `__no_init` object an initial value.

Variables declared using the `__no_init` keyword could, for example, be large input buffers or mapped to special RAM that keeps its content even with the application turned off.

For information about the `__no_init` keyword, see page 105. Note that to use this keyword, language extensions must be enabled; see *-e*, page 85. For information about the `#pragma object_attribute`, see page 113.

Located variables

Global and static variables can be explicitly placed at absolute addresses or in named segments using the `@` operator or `#pragma location`. The variables must be declared either `__no_init` or `const`. If declared `const`, it is legal for them to have initializers.

ABSOLUTE LOCATION PLACEMENT

To place a variable at an absolute address, the argument to the operator `@` and the `#pragma location` directive should be a literal number, representing the actual address. The absolute location must fulfill the alignment requirement for this type of variable.

Example

Assuming you are using the `large` data model:

```
__no_init char alpha @ 0x2000; /* OK */

#pragma location=0x2002
const int beta;                /* OK */

const int gamma @ 0x2004 = 3;  /* OK */

int delta @ 0x2006;            /* Error, neither */
                               /* "__no_init" nor "const". */

const int epsilon @ 0x2007;    /* Error, misaligned. */
```

SEGMENT PLACEMENT

It is possible to place variables into named segments using either the `@` operator or the `#pragma location` directive. The segment is specified as a string literal.

For information about segments, see the chapter *Segments and memory*.

Example

```
__no_init int alpha @ "MYSEGMENT"; /* OK */

#pragma location="MYSEGMENT"
const int beta;                    /* OK */
```

```
const int gamma @ "MYSEGMENT" = 4; /* OK */

int delta @ "MYSEGMENT";        /* Error, neither */
                                /* "__no_init" nor "const" */
```

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of SAM8 derivatives are included in the SAM8 IAR C Compiler delivery. The header files are named `io`*`chip`*`.h` and define the processor-specific special function registers (SFRs).

Note: Each header file contains one section used by the compiler, and one section used by the assembler.

SFRs with bitfields are declared in the header file. Example:

```
__no_init __bank0 volatile union
{
  unsigned char SYM;
  struct
  {
    unsigned char IE   : 1;
    unsigned char FIE  : 1;
    unsigned char FILS : 3;
    unsigned char      : 2;
    unsigned char TSE  : 1;
  } SYM_bit;
} @ 0xDE;
```

By including the appropriate `io`*`chip`*`.h` file into the user code it is possible to access either the whole register or any individual bit (or bitfields) from C code as follows:

```
// whole register access
SYM = 0x01;

// Bitfield accesses
SYM_bit.IE = 1;
```

You can also use the header files as templates when you create new header files for other SAM8 derivatives.

Anonymous structs and unions

An anonymous `struct` or `union` is a `struct` or `union` object that is declared without a name. Its members are promoted to the surrounding scope. An anonymous `struct` or `union` must not have a tag.

Note that anonymous struct and union objects are only available when language extensions are enabled in the SAM8 IAR C Compiler.

In the IAR Embedded Workbench, language extensions are enabled by default.

Use the -e compiler option to enable language extensions. See *-e*, page 85, for additional information.

Example

In the following example, 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 an anonymous struct or union at file scope, as a global, external, or static variable is also allowed. This could for instance be used for declaring I/O registers, as in the following example:

```
__no_init __bank0 volatile
union
   {
     unsigned char IOPORT;
     struct
       {
            unsigned char way: 1;
            unsigned char out: 1;
       };
   } @ 0xE2;
```

This declares an I/O register byte IOPORT at address 0xE2. The I/O register has 2 bits declared, way and out.

The following example illustrates how variables declared this way can be used:

```
void test()
{
   IOPORT=0;
   way=1;
   out=1;
}
```

Functions

This chapter contains information about functions. First the different ways normal code can be generated and the concept of code models are introduced. Then the special function types interrupt, monitor, and fast are described. The last section describes how to place functions into named segments.

Code models

The code model controls how code is generated for an application. All object files of a system must be compiled using the same code model.

In the chapter *Assembler language interface*, the generated code is studied in more detail when we describe how to call a C function from assembler language and vice versa.

The SAM8 microcontroller can be used in two modes: small mode and large register mode. The SAM8 IAR C Compiler supports these modes by means of code models:

- The small code model, which is the default, uses the internal stack
- The large code model uses the external stack.

Special function types

This section describes the special function types interrupt, monitor, and fast. The SAM8 IAR C Compiler allows an application to fully take advantage of these powerful SAM8 features without forcing the developers to implement anything in assembler language.

INTERRUPT FUNCTIONS

In embedded systems, the use of interrupts is a method of detecting external events immediately; for example, detecting that a button has been pressed.

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

The SAM8 microcontroller supports many interrupt sources. For each interrupt source, an interrupt routine can be written. Each interrupt routine is associated with a vector number which is specified in the SAM8 microcontroller documentation from the chip manufacturer. The `io*chip*.h` header file, which corresponds to the selected derivative, contains predefined names for the existing exception vectors.

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

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

Note: An interrupt function must have a return type of `void` and it cannot specify any parameters.

When an interrupt function is defined with a vector, the processor interrupt vector table is populated. It is also possible to define an interrupt function without a vector. This is useful if an application is capable of populating or changing the interrupt vector table at runtime. See the chip manufacturer's SAM8 microcontroller documentation for more information about the interrupt vector table.

The chapter *Assembler language interface* in this guide contains more information about the runtime environment used by interrupt routines.

FAST FUNCTIONS

A fast function is a quicker type of interrupt function, used for fast function processing. See the hardware manual.

MONITOR FUNCTIONS

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

For additional information, see *__monitor*, page 103.

Segment placement

It is possible to place functions into named segments using either the `@` operator or the `#pragma location` directive. When placing functions into segments the segment is specified as a string literal.

Example

```
void f() @ "MYSEGMENT";
void g() @ "MYSEGMENT"
{
}

#pragma location="MYSEGMENT"
void h();
```

Assembler language interface

This chapter describes how to write library functions in assembler language that work together with an application written in C.

Introduction

When an application is written partly in assembler language and partly in C, the developers are faced with a number of questions.

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

The first three items will be covered in the section *Calling convention*, page 27.

The section on memory access methods below will cover how data in memory is accessed.

The answer to the question asked in the last item above is that the call stack can be displayed when you run assembler code in the debugger. However, the debugger requires information about the *call frame*, which must be supplied as annotations in the assembler source file.

The section *Runtime model attributes*, page 25, covers how it is possible to prevent incompatible modules from being linked together.

Finally, the section *Function directives*, page 35, covers some directives generated by the compiler that are not normally required when writing assembler code.

Runtime model attributes

This section introduces the concept of runtime attributes, a mechanism designed to prevent incompatible modules from being linked together into an application.

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

There is one exception: if the value of an attribute is *, then that attribute matches any value.

Example

Study the object files below that could (but do not have to) define the two runtime attributes color and taste:

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

Table 7: Example of runtime model attributes

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

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

SPECIFYING RUNTIME ATTRIBUTES

Runtime attributes can be specified for a module written in assembler language by using the RTMODEL directive. For detailed syntax information, see the *SAM8 IAR Assembler Reference Guide*.

Example

```
RTMODEL color, red
```

Note: IAR Systems' own, built-in runtime attributes all start with two underscores. If you want to eliminate the risk that any attribute names you specify yourself will be identical to future IAR runtime attribute names, you should not specify them with two initial underscores in the name.

PREDEFINED RUNTIME ATTRIBUTES

The table below shows the runtime model attributes that are available for the SAM8 IAR C Compiler. These can be included in assembler code or in mixed C and assembler code, and will at link time be used by the IAR XLINK Linker to ensure consistency between modules.

Runtime model attribute	Value	Description
`__rt_version`	*n*	This runtime key is always present in all modules generated by the SAM8 IAR C Compiler. If a major change in the runtime characteristics occurs, the value of this key changes
`__code_model`	`small` or `large`	Corresponds to the code model used in the project.
`__data_model`	`small` or `large`	Corresponds to the data model used in the project.

Table 8: Runtime model attributes

The easiest way to find the proper settings of the `RTMODEL` directive is to compile a C module and examine the list file.

If you are using assembler routines in the C code, refer to the chapter *Assembler directives* in the *SAM8 IAR Assembler Reference Guide*.

Calling convention

A calling convention is the way one function in a program calls another function. The compiler handles this automatically, but if a function is written in assembler language you must know where and how its parameters can be found, how to return to the program location from where it was called, and how to return the resulting value.

It is also important to know which registers an assembler-level routine must preserve. If the program preserves too many registers, the program might be ineffective. If it preserves too few registers the result would be an incorrect program.

FUNCTION DECLARATIONS

In C, a function must be declared in order for the compiler to know how to call it. A declaration could look as follows:

```
int a_function(int first, char * second);
```

This means that the function takes two parameters: an integer and a pointer to a character. The function returns a value, an integer.

In the general case, this is the only knowledge that the compiler has about a function. Hence it must be able to deduce the calling convention from this information, as described below.

FUNCTION PARAMETERS

When deciding how to pass parameters to a function, each parameter is considered in turn from left to right. The method selected is based on the type of the parameter. Passing parameters in registers is faster than placing them on the stack.

Register parameters versus stack parameters

Parameters can be passed to a function using two basic methods: in registers or on the stack. Clearly it is much more efficient to use registers than to take a detour via memory. The calling convention is designed to utilize registers as much as possible. There is only a limited number of registers that can be used for passing parameters; when no more registers are available, the remaining parameters are passed on the stack. In addition, the parameters are passed on the stack in the following cases:

- Structure types: `struct` and `union` greater than 4 bytes
- Unnamed parameters to variable length functions, in other words functions declared as `foo(`*`param1`*`, ...)`, for instance `printf`.

Hidden parameters

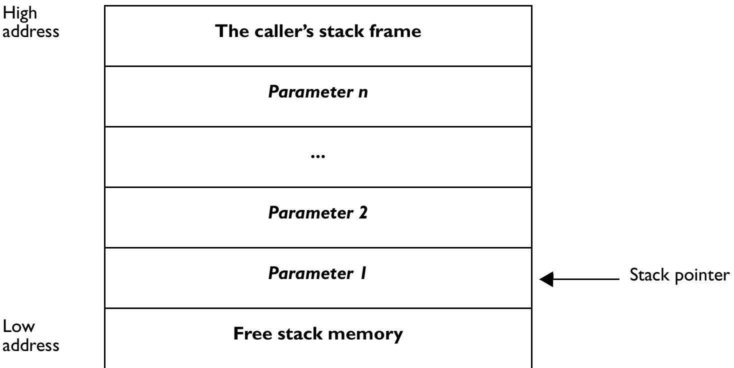
In addition to the parameters visible in a function declaration and definition, there can be hidden parameters. If the function returns a structure larger than 4 bytes, the memory location where to store the structure is passed in the register `R15` or in the register `RR14` as a hidden parameter depending on the size of the default pointer. The pointer is entered as a hidden parameter before all others.

Register parameters

The assignment of registers to parameters is a straightforward process. Each parameter is assigned to the first available register or registers. Should there be no more available registers, the parameter is passed on the stack.

Stack parameters

Stack parameters are stored in the main memory starting at the location pointed to by the stack pointer. Below the stack pointer (towards low memory) there is free space that the called function can use. The first stack parameter is stored at the location pointed to by the stack pointer. The next one is stored at the next location on the stack, etc. It is the responsibility of the caller to clean the stack after the called function has returned.



RETURNING A VALUE FROM A FUNCTION

The return value of a function, if any, can be scalar (such as integers and pointers), floating point, or a structure.

Return value pointer

If a structure greater than 4 bytes is returned, the caller passes a pointer to a location where the called function should write the result.

The called function must return the pointer in the register `R15` or in `RR14`, respectively.

PERMANENT VERSUS SCRATCH REGISTERS

Any of the registers `R10-R15` as well as the return address registers can be used as a *scratch register* by the function. This means that the original value does not have to be preserved.

The registers `R1`, `R4-R9` through to, but not including, the return address registers, are *permanent registers*. The values of permanent registers are assumed to survive a function call. This means that if a function uses a permanent register, its original value must be restored.

RETURN LOCATION

A number of registers are used by the system during function calls. If the return value is a `struct` larger than 4 bytes, an extra 'secret' parameter is passed as the first parameter containing the address of the result.

In the small code model, `R0` is used as the stack pointer if one is required, and in the large code model, `RR2` is used.

	Size	Small code model	Large code model
Return register	8	R15	R15
	16	RR14	RR14
	24	RR14:R12	RR14:R12
	32	RR12:RR14	RR12:RR14
Scratch register		R1, R4–R9, plus any register parameters	
Stack base pointer		R0	RR2

Table 9: Register use in different code models

EXAMPLES

The following section shows a series of examples of declarations and the corresponding calling convention. The complexity of the examples increases towards the end.

Example 1

Assume that we have the following function declaration:

```
int add1(int);
```

This function takes one parameter in register `RR14` and the return value is passed back to its caller in register `RR14`.

The following assembler routine is compatible with the declaration; it will return a value that is one number higher than the value of its parameter:

```
INCW      RR14
RET
```

Example 2

This example shows how structures are passed on the stack. Assume that we have the following declarations:

```
struct a_struct { int a; long b; };
int a_function(struct a_struct x, int y);
```

The calling function must reserve six bytes on the top of the stack and copy the contents of the `struct` to that location. The integer parameter `y` is passed in register `RR12`.

Example 3

The function below will return a `struct`. It is the responsibility of the calling function to allocate a memory location for the return value and pass a pointer to it as a hidden first parameter.

```
struct a_struct { int a; };
struct a_struct  a_function(int x);
```

The pointer to the location where the return value should be stored is passed in `R15` in the small data model and in `R14` in the other models. The parameter `x` is passed in `RR12`.

Assume that the function instead would have been declared to return a pointer to the structure:

```
struct a_struct *  a_function(int x);
```

In this case the return value is a scalar so there is no hidden parameter. The parameter `x` is passed in `RR14` and the return value is returned in `R15` or `RR14` depending on the code model.

Calling assembler routines from C

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

- Conform to the calling convention described on page 27
- 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)
  ```

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

CREATING SKELETON CODE

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

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

```
extern int globInt;
extern double globDouble;

int func(int arg1, double arg2)
{
  int locInt = arg1;
  globInt = arg1;
  globDouble = arg2;
  return locInt;
}

int main()
{
  int locInt = globInt;
  globInt = func(locInt, globDouble);
  return 0;
}
```

Note: In this example we use a low optimization level when compiling the code to show local and global variable access. If a higher level of optimization is used, the required references to local variables could be removed during the optimization. The actual function declaration is not changed by the optimization level.

Compiling the code

In the IAR Embedded Workbench, specify list options on file level. Select the file in the Project window. Then choose **Project>Options**. In the **ICCSAM8** category, select **Override inherited settings**. On the **List** page, deselect **Output list file** and instead select the **Output assembler file** option and its suboption **Include source**. Also, be sure to specify a low level of optimization.

Use the following options to compile the skeleton code:

```
iccsam8 shell -lA . -s3
```

The `-lA` option creates an assembler language output file including C source lines as assembler comments. The `.` (period) specifies that the assembler file should be named in the same way as the C module, i.e. `shell`, but with the filename extension `s18`. Remember also to specify the data model you are using.

The result is the assembler source `shell.s18` containing the declarations, function call, function return, and variable accesses.

The output file

The output file contains the following important information:

- The calling conventions
- The return values
- The global variables
- The function parameters
- How to create space on the stack (auto variables)
- Call frame information.

The call frame information needed by the Call Stack window in the IAR C-SPY™ Debugger is described by the `CFI` assembler directive. This directive is described in the *SAM8 IAR Assembler Reference Guide*.

Call frame information

When debugging an application using C-SPY it is possible to view the *call stack*. The compiler makes this possible by supplying debug information describing the layout of the call frame, in particular information about where the return address is stored.

If the call stack should be available when debugging a routine written in assembler language, equivalent debug information must be supplied by the author of the routine using the assembler directive `CFI`. This directive is described in detail in the *SAM8 IAR Assembler Reference Guide*.

When describing the call frame information, the following three components must be present:

- A *names block* describing the available resources to be tracked
- A *common block* corresponding to the calling convention
- Directives describing the changes that are performed on the call frame. This typically includes information about when the stack pointer is changed and when permanent registers are stored or restored on the stack.

The header file `cfi.m18` contains the macros `XCFI_NAMES` and `XCFI_COMMON` which declare a typical names block and a typical common block. These two macros declare a number of resources, both concrete and virtual. The following are of particular interest:

Resource	Description
`CFA_SP`	The call frames of the regular stack and of the interrupt stack, respectively
`R0–R15`	Normal registers

Table 10: Call frame information

Resource	Description
?RET	The return address register
SP, SPL	The stack pointer

Table 10: Call frame information (Continued)

Example

The following is an example of an assembler routine that stores a permanent register as well as the return register to the stack:

```
        PROGRAM cfiexample

        PUBLIC cfiexample

        RSEG CODE:CODE:NOROOT(0)

        CFI Names myNames
        CFI StackFrame CFA SPL IDATA
        CFI Resource R8:8, R9:8, R14:8, R15:8
        CFI VirtualResource ?RET:16
        CFI Resource SPL:8
        CFI EndNames myNames

        CFI Common myCommon Using myNames
        CFI CodeAlign 1
        CFI DataAlign 1
        CFI ReturnAddress ?RET CODE
        CFI CFA SPL+2
        CFI R8 Undefined
        CFI R9 Undefined
        CFI R14 Undefined
        CFI R15 Undefined

        CFI ?RET Frame(CFA, -2)
        CFI EndCommon myCommon

        CFI Block myBlock Using myCommon
        CFI Function `cfiexample`

cfiexample:
        PUSH      R9
        CFI R9 Frame(CFA, -3)
        CFI CFA SPL+3

        PUSH      R8
        CFI R8 Frame(CFA, -4)
        CFI CFA SPL+4
```

```
ADD       R15,#12
ADC       R14,#34
POP       R8
CFI CFA SPL+3

POP       R9
CFI CFA SPL+2

RET
CFI EndBlock myBlock

END
```

Function directives

The function directives are generated by the SAM8 IAR C Compiler to pass information about functions and function calls to the IAR XLINK Linker. These directives can be seen if you create an assembler list file by using the compiler option **Assembler file** (`-lA`).

Note: These directives are primarily intended to support static overlay, a feature which is useful in smaller microcontrollers. The SAM8 IAR C Compiler does not use static overlay, as it has no use for it.

SYNTAX

`FUNCTION` <*label*>,<*value*>
`ARGFRAME` <*segment*>, <*size*>, <*type*>
`LOCFRAME` <*segment*>, <*size*>, <*type*>
`FUNCALL` <*caller*>, <*callee*>

PARAMETERS

label	Label to be declared as function.
value	Function information.
segment	Segment in which argument frame or local frame is to be stored.
size	Size of argument frame or local frame.
type	Type of argument or local frame; either `STACK` or `STATIC`.
caller	Caller to a function.
callee	Called function.

DESCRIPTION

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

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

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

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

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

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

Segments and memory

This chapter introduces the concept of segments and describe the different segment groups and segment types. It also describes how they correspond to the memory and function types and how they interact with the runtime environment. The chapter also contains an overview of the linker configuration file, which is used for controlling the placement of segments in memory.

Note that the information in this chapter is conceptual; it is strictly generic and not related to any particular compiler or microcontroller. For product-specific details, see the linker configuration file included in your product package.

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

What is a segment?

A segment is a piece of data or code that should be mapped to a physical location in memory. The segment could either be placed in RAM or in ROM. Segments that are placed in RAM do not have any content, they only occupy space.

The compiler has a number of predefined segments for different purposes. Each segment has a name describing the contents of the segment. In addition, you can define your own segments.

The IAR XLINK Linker™ is responsible for placing the segments in the physical memory range in accordance with the rules specified in the linker configuration file. It is important to remember that, from the linker's point of view, all segments are equal, they are simply named parts of memory.

For detailed information about individual segments, see the *Segment reference* chapter in *Part 2. Compiler reference*.

LINKER SEGMENT TYPE

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

XLINK supports a number of other segment types than the ones described below. However, most of them exist to support other types of microcontrollers.

By default the compiler uses only the following XLINK segment types:

XLINK segment type	Description
`CODE`	Program memory (ROM)
`IDATA`	Internal Page Memory (RAM)
`DATA`	Internal Bank Memory (RAM)
`XDATA`	External memory (RAM)

Table 11: XLINK segment types

PLACEHOLDER SEGMENTS

The runtime environment of the compiler uses *placeholder segments*, empty segments that are used for marking a location in memory. Any type of segment can be used for placeholder segments.

Placing segments in memory

The placement of segments in memory is performed by the IAR XLINK Linker. It uses a linker configuration file that contains command line options which specify the locations where the segments can be placed, thereby assuring that your application fits on the target chip.

Since the chip-specific details are specified in the linker configuration file and not in the source code, the linker configuration file also ensures code portability. Basically, you can use the same source code with different derivatives just by rebuilding the code using an appropriate linker configuration file.

The `config` directory contains at least one ready-made linker configuration file. The file contains the information required by the linker and is ready to be used. If, for example, your application uses additional external RAM, you need to add details about the external RAM memory area. Remember not to change the original file. We recommend that you make a copy in the working directory, and modify the copy instead.

Notice that the supplied linker configuration file includes comments explaining the entire contents.

THE CONTENTS OF THE LINKER CONFIGURATION FILE

In particular, the linker configuration file specifies:

- The placement of segments
- The stack size

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

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

CUSTOMIZING A LINKER CONFIGURATION FILE

The examples below show the general principles for how to set up a linker configuration file. The target system is assumed to have the following fictitious memory layout:

Range	Type
0x0–0xFFFF	ROM
0x0000–0x1FFF	XRAM
0x0–0xFF	IRAM

Table 12: Linker configuration file example

The ROM can be used to store `CONST` and `CODE` memory. IRAM can be used to store IDATA memory, and XRAM can be used to store XDATA memory.

The only change you will normally have to make to the supplied linker configuration file is to suit the details of the target hardware memory map.

Example 1

The following will place the segments `MYSEGMENTA` and `MYSEGMENTB` in `CONST` memory (that is ROM) in the memory range of `0x2000–0xCFFF`.

```
-Z(CONST)MYSEGMENTA,MYSEGMENTB=2000-CFFF
```

Two segments of different types can be placed in the same memory area by not specifying a range for the second segment. In the following example the `MYSEGMENTA` segment is first located in memory. Then the rest of the memory range could be used by `MYCODE`.

```
-Z(CONST)MYSEGMENTA=2000-CFFF
-Z(CODE)MYCODE
```

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

```
-Z(CONST)MYSMALLSEGMENT=2000-20FF
-Z(CONST)MYLARGESEGMENT=2000-CFFF
```

Even though it is not strictly required, make sure to always specify the end of each memory range. If you do this, the IAR XLINK Linker will alert you if your segments do not fit. If you do not specify the end of memory ranges, you will not be alerted by the linker. See the *IAR Linker and Library Tools Reference Guide* for more details.

Example 2

The following example will place the data segment `MYDATA` in IDATA memory (that is, in IRAM) in a fictitious memory range:

```
-P(IDATA)MYDATA=0-FF,100-1FF
```

If your application has an additional RAM area in the memory range `0x200-0x2FF`, you just add that to the original definition:

```
-P(IDATA)MYDATA=0-FF,200-2FF,100-1FF
```

Note the XLINK `-P` option, which will make efficient use of the memory area.

Data segments

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

STATIC MEMORY SEGMENTS

Static memory is memory that contains variables that are global or are declared static, as described in *Memory access methods and memory types*, page 14.

This section describes how the segment types correspond to segment groups, and the segments that are part of the segment groups.

Segment naming

The memory types in the fictitious example started in *Customizing a linker configuration file*, page 39, can use the following ranges:

Memory type	Range
BANK*N*	`0xC0+n*0x100` to `0xFF+n*0x100`
NEAR	`0x0000-0xFFFF`
TINY	`0x00-0xBF`
TINY2	`0xC0-0xFF`
TINYP	`0x00-0xBF`
TINY2P	`0xC0-0xFF`
TINYP*N*	`0x00+n*0x100` to `0xBF+n*0x100`
TINY2P*N*	`0xC0+n*0x100` to `0xFF+n*0x100`

Table 13: Memory types

The static memory types in this fictitious example correspond to the following basic segment groups. The first part of the name of a segment in each segment group corresponds to the segment keyword:

Segment group	First part of name
Tiny	`TINY`
Tiny2	`TINY2`

Table 14: Segment groups

The variables declared in each of the groups can be divided into the following categories:

- Variables that are initialized to non-zero values
- Variables that should be initialized to zero
- Variables that are declared as const and therefore can be stored in ROM
- Variables defined with the `__no_init` keyword, denoting that they should not be initialized at all.

When an application is started, the `cstartup` module initializes memory in two steps:

1 It clears the memory of the variables that should be initialized to zero

2 It initializes the non-zero variables by copying a block of ROM to the location of the variables in RAM.

For each of the segment groups, some of the following segments exist:

Usage	Type	Suffix
Zero-initialized data	DATA	Z
Non-zero initialized data	DATA	I
Initializers for the above	CONST	ID
Constants	CONST	C
Non-initialized data	DATA	N
Absolute addressed data	DATA	A

Table 15: Segments in segment groups

The names of the actual segments are *NAME_SUFFIX*. For example, the segment TINY2_Z contains the tiny2 variables that should be initialized to zero when the system starts.

Initialized data

The data in the ROM segment with suffix ID is copied to the corresponding I segment when the system starts.

This works only when both segments are placed in continuous memory.

Tiny

The TINY segments must be placed in the theoretical memory range 0x00-0xBF. In this example these segments are placed in the available RAM area 0x00–0x1F.

The segment TINY_ID can be placed anywhere in code memory.

Tiny2

The TINY2 segments data must be placed in the theoretical memory range 0xC0-0xFF, which is anywhere in this example.

The segment TINY2_ID can be placed anywhere in code memory.

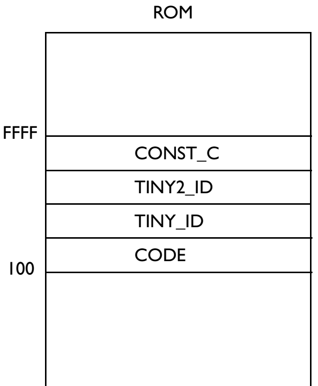
The linker configuration file

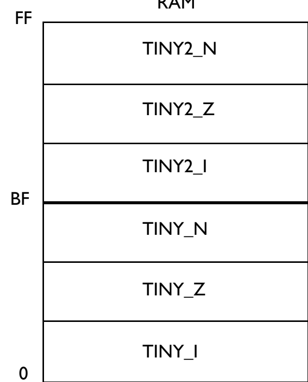
In this fictitious example the directives for placing the segments in the linker configuration file would be:

```
// The ROM segments
-Z(CODE)CODE,TINY_ID,TINY2_ID,NEAR_ID,CONST_C=100-FFFF

// The IRAM segments
-Z(IDATA)TINY_I,TINY_N,TINY_Z,TINYP_N,CSTACK+_CSTACK_SIZE=00-BF
```

```
-Z(IDATA)TINY2_I,TINY2_N,TINY2_Z,TINY2P_N,CSTACK2+_CSTACK2_SIZE=
   C0-FF
```

This gives the following placement of index segments:





THE STACK

The stack is used by functions to store variables and other information that is used locally by functions, as described in the chapter *Data storage*. There are two stacks depending on the code model, small or large. The small code model stack comes in two versions, one with and one without the reduced instruction set. The stack is a continuous block of memory pointed to by the processor stack pointer register. The `cstartup` module initializes the stack pointer to the end of the stack segment called `CSTACK` (small code model, reduced instruction set), `CSTACK2` (small code model, excluding reduced instruction set), or `CSTACKN` (large code model).

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

```
-D_CSTACKN_SIZE=size
-D_CSTACK2_SIZE=size
-D_CSTACK_SIZE=size
```

Note that the size is written hexadecimally without the `0x` notation.

At the end of the linker file the actual segment is defined in the memory area available for the stack:

```
-Z(IDATA)CSTACK+_CSTACK_SIZE#start-end
-Z(IDATA)CSTACK2+_CSTACK2_SIZE#start-end
-Z(XDATA)CSTACKN+_CSTACKN_SIZE#start-end
```

Stack size

The compiler uses the internal data 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.

THE HEAP

The heap contains data allocated by use of the C function `malloc` (or one of its relatives).

The default heap size is 64 bytes for the small code model, and 512 bytes for the large code model.

IAR CLIB Library

To change the heap size in the IAR CLIB Library you must include the file `heap.c`, found in the `src` directory, into the application. When the file is compiled the size of the heap is controlled by the preprocessor symbol `MALLOC_BUFSIZE`.

IAR Embedded Workbench

Add the file `heap.c` to the project.

Select **Project>Options**. In the **ICCSAM8** category, define the preprocessor symbol `MALLOC_BUFSIZE` on the **Preprocessor** page.

Command line

Compile the file `heap.c` using the command line option `-DMALLOC_BUFSIZE=`*`xxx`*, where *`xxx`* is the desired heap size.

Add the object file `heap.`r18 to the list of object files that is used for building the application.

LOCATED DATA

A variable that has been explicitly placed at an address, for example by using the compiler `@` syntax, will be placed in either the `CONST_A` or the `TINY_A` segment. The former is used for constant initialized data and the latter for items declared as `__no_init`. The individual segment part of the segment knows its location in the memory space and it does not have to be specified in the linker configuration file.

Code segments

This section contains descriptions of the segments used for storing code and the interrupt vector table.

STARTUP CODE

The segment RESET contains code used during system setup. The startup code should be placed at the location where the chip starts executing code after a reset.

In this example, the following line in the linker configuration file will place the RESET segment at address 0x100:

```
-Z(CODE)RESET=100
```

NORMAL CODE

Code for normal functions is placed in the CODE segment. Again, this is a simple operation in the linker configuration file:

```
-Z(CODE)CODE=2000-BFFF
```

EXCEPTION VECTORS

The exception vectors are typically placed in the segment INTVEC.

Runtime environment

This chapter describes the cstartup file which handles system initialization and termination. It presents how an application can control what happens before the start function main is called, by using either a custom __low_level_init or a modified cstartup file.

The standard library uses a small set of low-level input and output routines as a base for a wide range of I/O routines. This chapter describes how the low-level routines can be replaced by an application, so that it can use the standard function to, for example, communicate with the outside world or providing a memory-based file system.

This chapter also covers the methods used for communicating with the IAR C-SPY™ Debugger.

The cstartup.s18 file

This section will cover what actions the runtime environment performs during startup and termination of applications. In the next couple of sections customization is discussed.

SYSTEM STARTUP

When an application is initialized, a number of steps are performed:

- The `RESET` vector for the SAM8 CPU1 chip is initialized if needed
- The stack pointer is initialized
- The `EMT` register stack area bit is initialized depending on the code model used
- The custom-provided function `__low_level_init` is called, allowing the application a chance to perform early initializations
- Static variables are initialized. This includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the rest of the initialized variables
- The `main` function is called, which starts the application.

SYSTEM TERMINATION

An application can perform a normal termination in two different ways:

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

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

The default `exit` function is provided by the `cstartup` file.

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

__low_level_init

Some applications may need 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 or not data segments are initialized. If the function returns `0`, the data segment will not be initialized.

A skeleton for this function is supplied in the `low_level_init.c` file, which is installed with the product.

Note: The file `intrinsics.h` must be included by `low_level_init.c` to assure correct behavior of the `__low_level_init` routine.

Customizing cstartup.s18

The `cstartup.s18` file itself is well commented and is not described in detail in this guide. However, this section presents some general techniques used in the file including background information that might be useful if you need to modify the `cstartup.s18` file. It then describes how the customized `cstartup.s18` file could be used.

Note: Do not modify the `cstartup.s18` file unless required by your application. Your first option should always be to use a customized version of `__low_level_init` for initialization code.

For information about assembler source files, see the *SAM8 IAR Assembler Reference Guide*.

MODULES AND SEGMENT PARTS

In order to understand how the cstartup code is designed, it is imperative to have a clear understanding of modules and segment parts, and how the IAR XLINK Linker™ treats them.

An assembler module starts with a MODULE directive and ends with an ENDMOD directive. Inside the module a number of segment parts reside. Each segment part begins with an RSEG directive.

When XLINK builds an application, it starts with a small number of modules that have been declared as root. It then continues to include all modules that are referred from the already included modules. XLINK then discards unused segment parts.

Segment parts, REQUIRE, and the falling-through trick

The cstartup.s18 file has been designed to use the mechanism described above so that as little as possible of unused code will be included in the linked application.

For example, every piece of code used for initializing one type of memory is stored in a segment part of its own. If a variable is stored in a certain memory type, the corresponding initialization code will be referenced by the code generated by the compiler and hence included in your application. Should no variables of a certain type exist, the code is simply discarded.

A piece of code or data is not included if it is not used or referred to with the REQUIRE assembler directive.

The segment parts of cstartup defined in the cstartup.s18 file are guaranteed to be placed immediately after each other. XLINK will not change the order of the segment parts or modules since the segments are placed using the -Z option.

The above lets the cstartup.s18 file specify code in subsequent segment parts and modules that are designed so that some of the parts may not be included by XLINK. The following example shows this technique:

```
    MODULE  doSomething

    RSEG    MYSEG:CODE:NOROOT(1)    // First segment part.
    PUBLIC  ?do_something
    EXTERN  ?end_of_test
    REQUIRE ?end_of_test

?do_something:  // This will be included if someone refers to
    ...         // ?do_something. If this is included then
                // the REQUIRE directive above ensures that
                // the JP instruction below is included.
```

```
    RSEG    MYSEG:CODE:NOROOT(1)    // Second segment part.
    PUBLIC  ?do_something_else

?do_something_else:
    ...     // This will only be included in the linked
            // application if someone outside this function
            // refers to or requires ?do_something_else

    RSEG    MYSEG:CODE:NOROOT(1)    // Third segment part.
    PUBLIC  ?end_of_test

?end_of_test:
    JP    (?somewhere)            // This is included if
                                  // ?do_something above is
                                  // included.
    ENDMOD
```

CALL FRAME INFORMATION

When debugging an application, C-SPY is capable of displaying the call stack, that is, the functions that have called the current function. In order to ensure that the call stack is correctly displayed when executing code written in assembler language, information about the call frame must be provided. This is done by use of the assembler directive `CFI`, which is described in the *SAM8 IAR Assembler Reference Guide*.

MODIFYING THE CSTARTUP.S18 FILE

As noted earlier, you should not modify the `cstartup.s18` file if using a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the `cstartup.s18` file, we recommend that you follow this overall procedure for creating a modified copy of the file and adding it to your project.

In the IAR Embedded Workbench

1. Copy the assembler source file `cstartup.s18`, which is supplied in the product directory, to your project directory. Make any required modifications to the copy and save the file under the same name.
2. Select the appropriate code and data model options on the **Target** page in the **General** category of project options. See the *SAM8 IAR Embedded Workbench™ IDE User Guide* for additional information.
3. Add the file `cstartup.s18` to your project.
4. Select the option **Ignore CSTARTUP in library** on the **Include** page in the **XLINK** category of project options. See the *SAM8 IAR Embedded Workbench™ IDE User Guide* for additional information.

5 Rebuild your project.

From the command line

1 Copy the assembler source file `cstartup.s18`, which is supplied in the product directory, to your project directory. Make any required modifications to the copy.

2 Set the preprocessor symbol as specified in the tables below, to specify the data model.

Use one of the following preprocessor symbols to specify the appropriate code or data model:

Model	Preprocessor symbol
Small code model	`__CODE_MODEL_SMALL__`
Large code model	`__CODE_MODEL_LARGE__`
Small data model	`__DATA_MODEL_SMALL__`
Large data model	`__DATA_MODEL_LARGE__`

Table 16: Preprocessor symbols for code and data models

3 Use the assembler option `-D` to specify the data model symbol, for example:

```
asam8 cstartup -D__DATA_MODEL_SMALL__
```

This will create an object module file named `cstartup.r18`.

4 Specify the XLINK option `-C` in front of the name of the library to ignore the standard `cstartup` file that is part of the runtime library. See *Linking*, page 3. Then link your application.

Input and output

The SAM8 IAR C Compiler package provides most of the important C library definitions that apply to embedded systems. These are of three types:

- Standard C library definitions available for user programs. These are documented in this chapter.
- `CSTARTUP`, the single program module containing the start-up code. This is described in the *Run-time environment* chapter.
- Intrinsic functions, allowing low-level use of SAM8 features. See the chapter *Intrinsic functions* for more information.

LIBRARY OBJECT FILES

You must create an appropriate library object file for the chosen memory model and pointer type. See the *Run-time environment* chapter for more information. The IAR XLINK 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 that are supplied with the product. There are some I/O-oriented routines (such as `putchar` and `getchar`) 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. To avoid wasting time at compilation, the definitions are divided into a number of different header files. Each of these covers 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.

Header file	Description
`assert.h`	Assertions.
`ctype.h`	Character handling.
`iccbutl.h`	Low-level routines.
`math.h`	Mathematics.
`setjmp.h`	Non-local jumps.
`stdarg.h`	Variable arguments.
`stdio.h`	Input/output.
`stdlib.h`	General utilities.
`string.h`	String handling.

Table 17: IAR C Library header files

The following table shows header files that do not contain any functions, but specify various definitions and data types:

Header file	Description
`errno.h`	Error return values.
`float.h`	Limits and sizes of floating-point types.
`limits.h`	Limits and sizes of integral types.
`stddef.h`	Common definitions including `size_t`, `NULL`, `ptrdiff_t`, and `offsetof`.

Table 18: Miscellaneous IAR C Library header files

C-SPY debugger interface

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

THE DEBUGGER TERMINAL I/O WINDOW

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

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

When the label `?C_GETCHAR` is reached, C-SPY returns the next character in the input field. Should no input be given, C-SPY waits until the user has typed some input and pressed the Return key.

To make the Terminal I/O window available, the application must be linked with the XLINK option **Debug info with terminal I/O** selected. See the *SAM8 IAR Embedded Workbench™ IDE User Guide*.

Termination

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

Programming hints

This chapter provides hints on how to write efficient code.

General programming hints

This section contains general programming hints that will make your applications robust by using the available resources in an efficient way.

FUNCTION PROTOTYPES

In Kernighan & Ritchie C (K&R C), it was not possible to declare a function prototype. Instead an empty parameter list was used in the function declaration. Also, the definition looked different. Even though the old system still is valid we do not recommend using it since it makes it harder for the compiler to find problems in the application code. In addition, the code could be less efficient since type promotion (implicit casting) often is needed.

Examples

The following examples of a declaration and a definition show the differences between the old Kernighan & Ritchie form and the modern ISO/ANSI version.

Kernighan & Ritchie system

```
int test();                   /* old declaration */
int test(a,b)                 /* old definition */
char a;
int b;
{
  .....
}
```

ISO/ANSI system

```
int test(char, int);          /* declaration */
int test(char a, int b)       /* definition */
{
  .....
}
```

BITFIELDS

Using bitfields larger than 1 bit generates code that is both larger and slower than if non-bitfields integers were used.

ARRAYS

When using arrays it is more efficient if the type of the index expression matches the index type of the memory of the array.

Floating-point types

Using floating-point types on a microprocessor without a math co-processor is very inefficient both in terms of code size and execution speed.

Consider replacing code using floating-point operations with code using integers since these are more efficient.

Saving stack space and RAM memory

The following is a list of programming techniques that will, when followed, save memory and stack space:

- If stack space is limited, avoid long call chains and recursive functions.
- Declare variables with a short life span as auto variables. When the life spans for these variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution. Be careful with auto variables, though, as the stack size can exceed its limits.
- Avoid passing large non-scalar parameters to functions; in order to save stack space, you should instead pass them as pointers.

Optimization techniques

The purpose of optimization is to reduce the code size and to improve the execution speed. When only one of these goals can be satisfied, the compiler prioritizes according to the settings specified by the user. Note that one optimization sometimes enables other optimizations to be performed, and an application may become smaller even when optimizing for speed rather than size.

A high level of optimization will result in increased compile time and may also make debugging more difficult since it will be less clear how the generated code relates to the source code. However, we have made an effort to make the compiler output as debuggable as possible even at higher optimization levels. At any time, if you experience difficulties when debugging your code, try lowering the optimization level.

SPECIFYING THE OPTIMIZATION TYPE AND LEVEL

The SAM8 IAR C Compiler allows you to generate code that is optimized either for size or for speed, at a selectable optimization level. Both compiler options and `#pragma` directives are available for specifying the preferred type and level of optimization:

- The chapter *Compiler options* in *Part 2. Compiler reference* contains reference information about the command line options used for specifying optimization type and level. Refer to the *SAM8 IAR Embedded Workbench™ IDE User Guide* for information about the compiler options available in the IAR Embedded Workbench.
- Refer to *#pragma optimize*, page 114, for information about the `#pragma` directives that can be used for specifying optimization type and level. Normally you would use the same optimization level for an entire project or file, but the `#pragma optimize` directive allows you to fine-tune the optimization for a specific code section such as a time-critical function.

OPTIMIZATION HINTS

The following is a list of programming techniques that will, when followed, enable the compiler to better optimize the application.

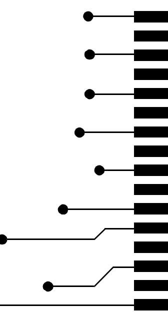
- The use of local variables is preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions may modify non-local variables.
- Avoid taking the address of local variables using the `&` operator. There are two main reasons why this is inefficient. First, the variable must be placed in memory and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables. Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions. This means that instead of calling a function, the compiler inserts the content of the function at the location where the function was called. The result is a faster but often larger application. Also, inlining may enable further optimizations. The compiler often inlines small functions declared static. The use of the `#pragma inline` directive gives the application developer fine-grained control. This feature can be disabled using the `--no_inline` command line option; see *--no_inline*, page 91.

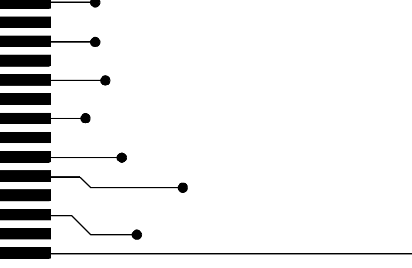
- Avoid using inline assembler. Instead, try writing the code in C, use intrinsic functions, or write a separate module in assembler language.

Part 2. Compiler reference

This part of the SAM8 IAR C Compiler Reference Guide contains the following chapters:

- Data representation
- Segment reference
- Compiler options
- Extended keywords
- #pragma directives
- Predefined symbols
- Intrinsic functions
- Library functions
- Diagnostics.





Data representation

This chapter describes the data types, pointers, and structure types supported by the SAM8 IAR C Compiler.

See the chapter *Programming hints* for information about which data types and pointers provide the most efficient code.

Alignment

The alignment of a data object controls how it will be stored in memory. The reason for using alignment is that the SAM8 microcontroller can access aligned objects more efficiently than non-aligned objects.

Objects with alignment 2 must be stored at addresses dividable by 2.

Data types

The compiler supports all ISO/ANSI C basic data types.

INTEGER TYPES

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

Data type	Size	Range	Alignment
signed char	8 bits	-128 to 127	1
unsigned char	8 bits	0 to 255	1
short, int	16 bits	-32768 to 32767	2
unsigned short, unsigned int	16 bits	0 to 65535	2
long	32 bits	-2^{31} to $2^{31}-1$	2
unsigned long	32 bits	0 to $2^{32}-1$	2

Table 19: Integer types

Signed variables are stored in the two's complement form.

The enum type

ISO/ANSI C specifies that constants defined using the `enum` construction should be representable using the type `int`. The compiler will use the shortest signed or unsigned type required to contain the values.

When IAR Systems language extensions are enabled, the `constant` and `enum` types can also be of the type `long` or `unsigned long`.

Char type

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

Bitfields

In ISO/ANSI C, `int` and `unsigned int` can be used as the base type for integer bitfields. In the SAM8 IAR C Compiler, any integer type can be used as the base type when language extensions are enabled.

Bitfields in expressions will have the same data type as the integer base type.

By default the compiler places bitfield members from the least significant to the most significant bit in the container type. By using the directive `#pragma bitfields=reversed` the bitfield members are placed from the most significant to the least significant bit.

FLOATING-POINT TYPES

Floating-point values are represented by 32-bit numbers in standard IEEE format.

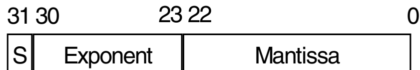
The ranges and sizes for the different floating-point types are:

Type	Size	Range (+/-)	Exponent	Mantissa
float	32 bits	±1.18E-38 to ±3.39E+38	8 bits	23 bits
double	32 bits	±1.18E-38 to ±3.39E+38	8 bits	23 bits

Table 20: Floating-point types

32-bit floating-point format

The data type `float` is represented by the 32-bit floating-point format. The representation of a 32-bit floating-point number as an integer is:



The value of the number is:

```
(-1)S * 2(Exponent-127) * 1.Mantissa
```

The precision of the float operators (`+`, `-`, `*`, and `/`) is approximately 7 decimal digits.

Special cases

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.

Pointers

The SAM8 IAR C Compiler has two basic types of pointers: code pointers and data pointers.

SIZE

The size of code pointers is always 16 bits and they can address the entire memory.

Data pointers have one of three sizes: 8, 16, or 24 bits.

CASTING

Casts between pointers have the following characteristics:

- Casting an integer value to a pointer of a smaller size is performed by truncation
- Casting an integer to a larger pointer id performed by zero extension
- Casting a pointer type to a smaller integer type is performed by truncation
- Casting a pointer to a larger integer type is 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. In the SAM8 IAR C Compiler, the size of `size_t` is 16 bits.

ptrdiff_t

`ptrdiff_t` is the type of the signed integer required to hold the difference between two pointers to elements of the same array. In the SAM8 IAR C Compiler, the size of `ptrdiff_t` is 16 bits.

intptr_t

`intptr_t` is a signed integer type large enough to contain a `void *`. In the SAM8 IAR C Compiler the size of `intptr_t` is 32 bits.

uintptr_t

`uintptr_t` is equivalent to `intptr_t` with the exception that it is unsigned.

Structure types

The members of a `struct` are stored sequentially in the order in which they are declared: the first member has the lowest memory address.

ALIGNMENT

The `struct` and `union` types inherit the alignment requirements of their members. In addition, the size of a `struct` is adjusted to allow arrays of aligned structure objects.

GENERAL LAYOUT

Members of a `struct` (fields) are always allocated in the order given in the declaration. The members are placed in memory according to the given alignment (offsets).

Example

```
struct {
  short s;  /* stored in byte 0 and 1 */
  char c;   /* stored in byte 2 */
  long l;   /* stored in byte 4, 5, 6, and 7 */
  char c2;  /* stored in byte 8 */
} s;
```

The following diagram shows the layout in memory:

s.s 2 bytes	s.c 1 byte	pad 1 byte	s.l 4 bytes	s.c2 1 byte	pad 1 byte

The alignment of the structure is 2 bytes and its size is 10 bytes.

Segment reference

The SAM8 IAR C Compiler places code and data into named segments which are referred to by the IAR XLINK Linker™. Details about the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output from the compiler.

For information about how to define segments in the linker configuration file, see *Customizing a linker configuration file*, page 39.

Summary of segments

The table below lists the segments that are available in the SAM8 IAR C Compiler. Notice that *located* denotes absolute location using the `@` operator or the `#pragma location` directive. The linker segment type `CODE`, `CONST`, or `DATA` indicates whether the segment should be placed in ROM or RAM memory areas; see Table 11, *XLINK segment types*, page 38.

Segment	Description	Type
`BANKn_A`	Holds the SFRs, system registers and working registers.	`DATA`
`CALLT_C`	Holds `__tiny_func` function addresses.	`CODE`
`CODE`	Holds user program code.	`CODE`
`CONST`	Holds constants.	`CODE`
`CSTACK`	Holds the internal stack in the small code model, including the reduced instruction set.	`IDATA`
`CSTACK2`	Holds the internal stack in the small code model, except for the reduced instruction set.	`IDATA`
`CSTACKN`	Holds the internal stack in the large code model.	`XDATA`
`INTVEC`	Holds the interrupt vectors generated by use of the `interrupt` extended keyword.	`CODE`
`NEAR_C`	Used for storing `__near` constant data.	`XDATA`
`NEAR_I`	Holds `__near` static variables declared with non-zero initial values.	`XDATA`
`NEAR_ID`	Holds initial values for variables located in the `NEAR_I` segment.	`CODE`
`NEAR_N`	Holds `__near` variables to be placed in non-volatile memory.	`XDATA`

Table 21: Segment summary

Segment	Description	Type
NEAR_Z	Holds __near variables with static storage declared without initial values or with zero values.	XDATA
RESET	Holds the startup code.	CODE
TINY_I	Holds __tiny static variables declared with non-zero initial values.	IDATA
TINY_ID	Holds initial values for variables located in the TINY_I segment.	CODE
TINY_N	Holds __tiny variables to be placed in non-volatile memory.	IDATA
TINY_Z	Holds __tiny variables with static storage declared without initial values or with zero values.	IDATA
TINY2_I	Holds __tiny2 static variables declared with non-zero initial values.	IDATA
TINY2_ID	Holds initial values for variables located in the TINY2_I segment.	CODE
TINY2_N	Holds __tiny2 variables to be placed in non-volatile memory.	IDATA
TINY2_Z	Holds __tiny2 variables with static storage declared without initial values or with zero values.	IDATA
TINYP_N	Holds __tinyp variables to be placed in non-volatile memory.	IDATA
TINYP*i*_N	Holds __tinyp*i* variables to be placed in non-volatile memory.	IDATA
TINY2P_N	Holds __tiny2p variables to be placed in non-volatile memory.	IDATA
TINY2P*i*_N	Holds __tiny2p*i* variables to be placed in non-volatile memory.	IDATA

Table 21: Segment summary (Continued)

Descriptions of segments

The following section gives reference information about each segment. Many of the extended keywords supported by the compiler are mentioned here. For detailed information about the keywords, see the chapter *Extended keywords*.

BANK0_A

Holds the SFRs, system registers, and working registers.

Linker segment type

DATA

Memory range

0xC0–0xFF

`BANK1_A` Holds the SFRs, system registers, and working registers.

Linker segment type

`DATA`

Memory range

`0x1C0–0x1FF`

`CALLT_C` Holds `__tiny_func` function addresses.

Linker segment type

`CODE`

Memory range

`0x00–0xFF`

`CODE` Holds user program code.

Linker segment type

`CODE`

Memory range

`0x0000–0xFFFF`

`CONST` Holds constants.

Linker segment type

`CODE`

Memory range

`0x0000–0xFFFF`

CSTACK Holds the internal data stack in the small code model, including the reduced instruction set.

Linker segment type

IDATA

Memory range

0–0xBF

CSTACK2 Holds the internal data stack in the small code model, except the reduced instruction set.

Linker segment type

IDATA

Memory range

0x00–0xFF

CSTACKN Holds the internal data stack in the large code model.

Linker segment type

XDATA

Memory range

0x0000–0xFFFF

INTVEC Holds the interrupt vectors generated by the use of the `interrupt` extended keyword.

Linker segment type

CODE

Memory range

0x00–0xFF

NEAR_C Used for storing __near constant data.

Linker segment type

XDATA

Memory range

0x0000–0xFFFF

NEAR_I Holds __near static variables that have been declared with non-zero initial values. The initial values are copied from the NEAR_ID segment by cstartup during initialization.

Linker segment type

XDATA

Memory range

0x0000–0xFFFF

NEAR_ID Holds initial values for variables located in the NEAR_I segment. These values are copied from NEAR_I by cstartup during initialization.

Linker segment type

CODE

Memory range

0x0000–0xFFFF

NEAR_N Holds __near variables to be placed in non-volatile memory. These have been allocated by the compiler, declared __no_init, or created __no_init by use of the #pragma memory directive.

Linker segment type

XDATA

Memory range

0x0000–0xFFFF

NEAR_Z Holds __near variables with static storage that were declared without initial values or with zero values. Standard C specifies that such variables be set to zero before they are encountered by the program, so they are set to zero by cstartup during initialization.

Linker segment type

XDATA

Memory range

0x0000–0xFFFF

RESET Holds the startup code.

Linker segment type

CODE

Memory range

0x0000–0xFFFF

TINY_I Holds __tiny static variables that have been declared with non-zero initial values. The initial values are copied from the TINY_ID segment by cstartup during initialization.

Linker segment type

IDATA

Memory range

0x00–0xBF

TINY_ID Holds initial values for variables located in the TINY_I segment. These values are copied from TINY_I by cstartup during initialization.

Linker segment type

CODE

Memory range

0x0000–0xFFFF

TINY_N

Holds __tiny variables to be placed in non-volatile memory. These have been allocated by the compiler, declared __no_init, or created __no_init by use of the #pragma memory directive.

Linker segment type

IDATA

Memory range

0x00–0xBF

TINY_Z

Holds __tiny variables with static storage that were declared without initial values or with zero values. Standard C specifies that such variables be set to zero before they are encountered by the program, so they are set to zero by cstartup during initialization.

Linker segment type

IDATA

Memory range

0x00–0xBF

TINY2_I

Holds __tiny2 static variables that have been declared with non-zero initial values. The initial values are copied from the TINY2_ID segment by cstartup during initialization.

Linker segment type

IDATA

Memory range

0xC0–0xFF

TINY2_ID

Holds initial values for variables located in the TINY2_I segment. These values are copied from TINY2_I by cstartup during initialization.

Linker segment type

CODE

Memory range

0x0000–0xFFFF

TINY2_N Holds __tiny2 variables to be placed in non-volatile memory. These have been allocated by the compiler, declared __no_init, or created __no_init by use of the #pragma memory directive.

Linker segment type

IDATA

Memory range

0xC0–0xFF

TINY2_Z Holds __tiny2 variables with static storage that were declared without initial values or with zero values. Standard C specifies that such variables be set to zero before they are encountered by the program, so they are set to zero by cstartup during initialization.

Linker segment type

IDATA

Memory range

0xC0–0xFF

TINYP_N Holds __tinyp variables to be placed in non-volatile memory. These have been allocated by the compiler, declared __no_init, or created __no_init by use of the #pragma memory directive.

Linker segment type

IDATA

Memory range

0x00–0xBF, 0x100–0x1BF, ..., 0xF00–0xFBF

TINYP*i*_N Holds __tinyp*i* variables to be placed in non-volatile memory. These have been allocated by the compiler, declared __no_init, or created __no_init by use of the #pragma memory directive.

Linker segment type

IDATA

Memory range

0x*i*00–0x*i*BF, where *i* = 1 to F

TINY2P_N Holds __tiny2p variables to be placed in non-volatile memory. These have been allocated by the compiler, declared __no_init, or created __no_init by use of the #pragma memory directive.

Linker segment type

IDATA

Memory range

0xC0–0xFF, 0x1C0–0x1FF, . . ., 0xFC0–0xFFF

TINY2P*i*_N Holds __tiny2p*i* variables to be placed in non-volatile memory. These have been allocated by the compiler, declared __no_init, or created __no_init by use of the #pragma memory directive.

Linker segment type

IDATA

Memory range

0x*i*C0–0x*i*FF, where *i* = 1 to F

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 *SAM8 IAR Embedded Workbench™ IDE User Guide* for information about the compiler options available in the IAR Embedded Workbench and how to set them.

Setting command line options

To set compiler options from the command line, include them on the command line after the `iccsam8` 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:

```
iccsam8 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`:

```
iccsam8 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:

```
iccsam8 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 `--warnings_are_errors`.

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:

```
iccsam8 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`:

```
iccsam8 prog -l ---r
```

SPECIFYING ENVIRONMENT VARIABLES

Compiler options can also be specified in the `QCCSAM8` 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 SAM8 IAR C Compiler:

Environment variable	Description
C_INCLUDE	Specifies directories to search for include files; for example: C_INCLUDE=c:\program files\iar systems\embedded workbench 3.n\sam8\inc;c:\headers
QCCSAM8	Specifies command line options; for example: QCCSAM8=-lA asm.lst -z9

Table 22: Environment variables

ERROR RETURN CODES

The SAM8 IAR C Compiler returns status information to the operating system which can be tested in a batch file.

The following command line error codes are supported:

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

Table 23: Error return codes

Options summary

The following table summarizes the compiler command line options:

Command line option	Description
`--char_is_signed`	'char' is 'signed char'
`--code_model {small|large}`	Specifies the code model
`--core = {sam8|sam8x|sam8xri|sam8xrc|sam8xrcri}`	Specifies the processor core
`-Dsymbol[=value]`	Defines preprocessor symbols
`--data_model {small|large}`	Specifies the data model

Table 24: Compiler options summary

Command line option	Description
--debug	Generates debug information
--dependencies=[i][m] {*filename*|*directory*}	Lists file dependencies
--diag_error=*tag*,*tag*,...	Treats these as errors
--diag_remark=*tag*,*tag*,...	Treats these as remarks
--diag_suppress=*tag*,*tag*,...	Suppresses these diagnostics
--diag_warning=*tag*,*tag*,...	Treats these as warnings
--diagnostics_tables	Lists all diagnostic messages
-e	Enables language extensions
--enable_eeprom_support	Enables EEPROM support
--enable_multibytes	Enables support for multibyte characters
-f *filename*	Extends the command line
--generate_tinyfunc_runtime_library_calls	Generates __tiny_func runtime library calls
--header_context	Lists all referred source files
-I*path*	Includes file path
-l[c|C|a|A][N][H] {*filename*|*directory*}	Creates list file
--library_module	Makes library module
--migration_preprocessor_extensions	Extends the preprocessor
--module_name=*name*	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
--no_wrap_diagnostics	Disables wrapping of diagnostic messages
-o {*filename*|*directory*}	Sets object filename
--omit_types	Excludes type information
--only_stdout	Uses standard output only

Table 24: Compiler options summary (Continued)

Command line option	Description
`--place_constants_in_rom`	Places constants and string literals in code memory
`--preprocess[=[c][n][l]] {`*filename*|*directory*`}`	Generates preprocessor output
`--public_equ` *symbol*`[=`*value*`]`	Defines a global, named assembler label
`-r`	Generates debug information
`--remarks`	Enables remarks
`-s[2|3|6|9]`	Optimizes for speed
`--silent`	Sets silent operation
`--strict_ansi`	Enables strict ISO/ANSI
`--warnings_affect_exit_code`	Warnings affects exit code
`--warnings_are_errors`	Warnings are treated as errors
`-z[2|3|6|9]`	Optimizes for size

Table 24: Compiler options summary (Continued)

Descriptions of options

The following section gives detailed reference information about each compiler option.

--char_is_signed

`--char_is_signed`

By default the compiler interprets the `char` type as unsigned. The `--char_is_signed` option causes the compiler to interpret the `char` type as signed instead. This can be useful when you, for example, want to maintain compatibility with another compiler.

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Language**.

--code_model

`--code_model {small|large}`

The SAM8 microcontroller can be used in two modes: normal mode and short register mode. The SAM8 IAR C Compiler supports these models by means of code models.

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

Code model	Description
`small` (default)	Supports the tiny stack
`large`	Supports the near stack

Table 25: Available code models

If you do not include any of the code model options, the compiler uses the small code model as default.

Note that all modules of your application must use the same code model.

Example

For example, use the following command to specify the short code model:

```
--code_model small
```

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>General>Target**.

`--core` `--core = {sam8|sam8x|sam8xri|sam8xrc|sam8xrcri}`

This option selects the processor core. (The default is `sam8`.)

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

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

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

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

`-D`*`symbol`*

is equivalent to:

`#define` *`symbol`* `1`

In order to get the equivalence of:

```
#define foo
```

specify the `=` sign but nothing after, for example:

```
-Dfoo=
```

This option can be used one or more times on the command line.

Example

You may want to 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: iccsam8 prog
Test version: iccsam8 prog -DTESTVER

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Preprocessor**.

--data_model

--data_model {small|large}

Use this option to select the data model for which the code is to be generated:

Data model	Default memory attribute
small (default)	__tiny
large	__near

Table 26: Available data models

If you do not include any of the data model options, the compiler uses the small data model as default.

Note that all modules of your application must use the same data model.

Example

For example, use the following command to specify the large data model:

```
--data_model large
```

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>General>Target**.

--debug, -r

--debug

-r

Use the --debug or the -r option to make the compiler include information required by the IAR C-SPY™ Debugger and other symbolic debuggers in the object modules.

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Output**.

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

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

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

Table 27: Generating a list of dependencies (--dependencies)

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

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

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

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

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

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

```
foo.r18: c:\iar\product\include\stdio.h
foo.r18: d:\myproject\include\foo.h
```

Example 1

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

```
iccsam8 prog --dependencies=i listing
```

Example 2

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

```
iccsam8 prog --dependencies \mypath\listing
```

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

Example 3

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

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

```
%.r18 : %.c
      $(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

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

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

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

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

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

Use this option to classify diagnostic messages as errors. 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.

Example

The following example classifies warning `Pe117` as an error:

```
--diag_error=Pe117
```

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Diagnostics**.

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

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

Example

The following example classifies the warning `Pe177` as a remark:

```
--diag_remark=Pe177
```

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Diagnostics**.

--diag_suppress

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

Use this option to suppress diagnostic messages.

Example

The following example suppresses the warnings `Pe117` and `Pe177`:

```
--diag_suppress=Pe117,Pe177
```

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Diagnostics**.

--diag_warning

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

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

Example

The following example classifies the remark `Pe826` as a warning:

```
--diag_warning=Pe826
```

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Diagnostics**.

--diagnostics_tables

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

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

This option cannot be given together with other options.

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

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

Example 1

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

```
--diagnostics_tables diag
```

Example 2

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

```
--diagnostics_tables .
```

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

`-e` `-e`

In the command line version of the SAM8 IAR C Compiler, language extensions are disabled by default. If you use language extensions such as SAM8-specific keywords and anonymous structs and unions in your source code, you must enable them by using this option.

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

For additional information, see *IAR language extension overview, page 5.*

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Language**.

`--enable_eeprom_support` `--enable_eeprom_support`

This option enables EEPROM support.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>General>Target**.

`--enable_multibytes` `--enable_multibytes`

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

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

To set the equivalent option in the IAR Embedded Workbench, choose **Project>Options>ICCSAM8>Language**.

-f

`-f filename`

Reads command line options from the named file, with the default extension `xcl`.

By default the compiler accepts command parameters only from the command line itself and the `QCCSAM8` environment variable. To make long command lines more manageable, and to avoid any operating system command line length limit, you use the `-f` option to specify a command file, from which the compiler reads command line items as if they had been entered at the position of the option.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines since the newline character acts just as a space or tab character.

Both C and C++ style comments are allowed in the file. Double quotes behave as in the Microsoft Windows command line environment.

Example

For example, you could replace the command line:

```
iccsam8 prog -r "-DUsername=John Smith" -DUserid=463760
```

with

```
iccsam8 prog -r -f userinfo
```

and the file `userinfo.xcl` containing:

```
"-DUsername=John Smith"
-DUserid=463760
```

--generate_tinyfunc_runtime_library_calls

`--generate_tinyfunc_runtime_library_calls`

This option generates `__tiny_func` runtime library calls.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Language**.

--header_context

--header_context

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

-I

-I*path*

Use this option to specify paths for `#include` files. 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, for example:

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

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

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

  ```
  src.c in directory dir
    #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:

  ```
  iccsam8 ..\src.c -I..\dir\include
  ```

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

`dir\h`	Current file.
`dir`	File including current file.
`dir\include`	As specified with the `-I` option.

Use angle brackets for standard header files like `stdio.h`, and double quotes for files that are part of your application.

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Preprocessor**.

`-l` `-l[c|C|a|A][N][H]` {*filename*|*directory*}

By default the compiler does not generate a listing. Use this option to generate a listing to a file.

The following modifiers are available:

Option modifier	Description
`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
`H`	Include source lines from header files in output. Without this option only source lines from the primary source file are included.

Table 28: Generating a compiler list file (-l)

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

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

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

Example 1

To generate a listing to the file `list.lst`, use:

```
iccsam8 prog -l list
```

Example 2

If you compile the file `mysource.c` and want to generate a listing to a file `mysource.lst` in the working directory, you could use:

```
iccsam8 prog -l .
```

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>List**.

--library_module

```
--library_module
```

Use this option to make the compiler treat the object file as a library module rather than as a program module. A program module is always included during linking. A library module will only be included if it is referenced in your program.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Output**.

--migration_preprocessor_extensions

```
--migration_preprocessor_extensions
```

Migration preprocessor extensions extend the preprocessor in order to ease migration of code from earlier IAR compilers. The preprocessor extensions include:

- The availability of floating point in preprocessor expressions.
- The availability of basic type names and `sizeof` in preprocessor expressions.
- The availability of all symbol names (including typedefs and variables) in preprocessor expressions.

If you need to migrate code from an earlier IAR C or C/EC++ compiler, you may want to enable these preprocessor extensions.

Note: If you use this option, not only will the compiler accept code that is not standard conformant, but it will also reject some code that *does* conform to standard.

Important! Do not depend on these extensions in newly written code. Support for them may be removed in future compiler versions.

--module_name

```
--module_name=name
```

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:

```
iccsam8 prog --module_name=main
```

This option is 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.

Example

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
iccsam8 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
iccsam8 temp.c --module_name=%1    ; use original source
                                   ; name as module name
```

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Output**.

--no_code_motion

`--no_code_motion`

Use this option to disable optimizations that move code. These optimizations, which are performed at optimization levels 6 and 9, normally reduce code size and execution time. The resulting code may however be difficult to debug.

Note: This option has no effect at optimization level 3.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Code**.

--no_cse

`--no_cse`

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

On optimization levels 6 and 9, the compiler avoids calculating the same expression more than once. 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 level 3.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Code**.

--no_inline

`--no_inline`

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 level 9, normally reduces execution time and increases code size. The resulting code may also be difficult to debug.

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 3 and 6.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Code**.

--no_unroll

`--no_unroll`

Use this option to disable loop unrolling.

The code body of a small loop, whose number of iterations can be determined at compile time, is duplicated to reduce the loop overhead.

For small loops, the overhead required to perform the looping can be large compared to the work performed in the loop body.

The loop unrolling optimization duplicates the body several times, reducing the loop overhead. The unrolled body also opens up for other optimization opportunities, for example the instruction scheduler.

This optimization, which is performed at optimization level 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.

Note: This option has no effect at optimization levels 3 and 6.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Code**.

--no_warnings

`--no_warnings`

By default the compiler issues standard warning messages. Use this option to disable all warning messages.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Diagnostics**.

--no_wrap_diagnostics

`--no_wrap_diagnostics`

By default, long lines in compiler diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.

-o

`-o {`*`filename`*`|`*`directory`*`}`

Use the `-o` option to specify an output file.

If a *`filename`* is specified, the compiler stores the object code in that file.

If a *`directory`* is specified, the compiler stores the object code in that directory, in a file with the same name as the name of the compiled source file, but with the extension `r18`. To specify the working directory, replace *`directory`* with a period (`.`).

Example 1

To store the compiler output in a file called `obj.r18` in the `mypath` directory, you would use:

```
iccsam8 prog -o \mypath\obj
```

Example 2

If you compile the file `mysource.c` and want to store the compiler output in a file `mysource.r18` in the working directory, you could use:

```
iccsam8 prog -o .
```

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>General>Output Directories**.

--omit_types

`--omit_types`

By default, the compiler includes type information about variables and functions in the object output.

Use this option if you instead want the compiler to ignore such type information in the output. The object file will then only contain type information that is a part of a symbol's name. This means that the linker cannot check symbol references for type correctness, which is useful when you build a library that should not contain type information.

--only_stdout

`--only_stdout`

Use this option to make the compiler use the standard output stream (`stdout`) also for messages that are normally directed to the error output stream (`stderr`).

--place_constants_in_rom

`--place_constants_in_rom`

Use this option to place constants and string literals in code memory, in the segment `CONST_C`. These are otherwise placed in data memory, in the segments `TINYP_I` or `NEAR_I`.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Language**.

--preprocess

`--preprocess[=[c][n][l]]` {*filename*|*directory*}

Use this option to direct preprocessor output to a named file.

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

Command line option	Description
`--preprocess=c`	Preserve comments
`--preprocess=n`	Preprocess only
`--preprocess=l`	Generate `#line` directives

Table 29: Directing preprocessor output to file (--preprocess)

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

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

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

Example 1

To store the compiler output with preserved comments to the file `output.i`, use:

```
iccsam8 prog --preprocess=c output
```

Example 2

If you compile the file `mysource.c` and want to store the compiler output with `#line` directives to a file `mysource.i` in the working directory, you could use:

```
iccsam8 prog --preprocess=l .
```

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Preprocessor**.

`--public_equ`

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

This option is equivalent to defining a label in assembler language by using the `EQU` directive and exporting it using the `PUBLIC` directive.

`-r, --debug`

`-r`

`--debug`

Use this option to make the compiler include information required by the IAR C-SPY Debugger and other symbolic debuggers in the object modules.

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Output**.

`--remarks`

`--remarks`

The least severe diagnostic messages are called remarks (see *Severity levels*, page 131). A remark indicates a source code construct that may cause strange behavior in the generated code.

By default the compiler does not generate remarks. Use this option to make the compiler generate remarks.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Diagnostics**.

`-s` `-s[2|3|6|9]`

Use this option to make the compiler optimize the code for maximum execution speed.

If no optimization option is specified, the compiler will use the size optimization `-z2` by default. If the `-s` option is used without specifying the optimization level, speed optimization at level 2 is used by default.

The following table shows how the optimization levels are mapped:

Option modifier	Optimization level
2	None*
3	Low
6	Medium
9	High

Table 30: Specifying speed optimization (-s)

***The most important difference between `-s2` and `-s3` is that at level 2, all non-static variables will live during their entire scope.**

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and conversely a high level of optimization makes it relatively hard.

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Code**.

`--silent` `--silent`

By default the compiler issues introductory messages and a final statistics report. Use `--silent` to make the compiler operate without sending these messages to the standard output stream (normally the screen).

This option does not affect the display of error and warning messages.

`--strict_ansi` `--strict_ansi`

By default the compiler accepts a relaxed 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.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Language**.

--warnings_affect_exit_code

`--warnings_affect_exit_code`

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Diagnostics**.

--warnings_are_errors

`--warnings_are_errors`

Use this option to make the compiler treat all warnings as errors. If the compiler encounters an error, no object code is generated. Warnings that have been changed into remarks are not treated as errors.

Note: Any diagnostic messages that have been reclassified as warnings by the compiler option `--diag_warning` or the `#pragma diag_warning` directive will also be treated as errors when `--warnings_are_errors` is used.

For additional information, see *--diag_warning*, page 84 and *#pragma diag_warning*, page 112.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Diagnostics**.

-z

`-z[2|3|6|9]`

Use this option to make the compiler optimize the code for minimum size. If no optimization option is specified, `-z2` is used by default.

The following table shows how the optimization levels are mapped:

Option modifier	Optimization level
2	None*
3	Low
6	Medium
9	High

Table 31: Specifying size optimization (-z)

***The most important difference between `-z2` and `-z3` is that at level 2, all non-static variables will live during their entire scope.**

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and conversely a high level of optimization makes it relatively hard.

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCSAM8>Code**.

Extended keywords

This chapter describes the extended keywords that support specific features of the SAM8 microcontroller, the general syntax rules for the keywords, and a detailed description of each keyword.

For information about the address ranges of the different memory areas, see the chapter *Segment reference*.

Summary of extended keywords

The following table summarizes the extended keywords that are available to the SAM8 IAR C Compiler:

Extended keyword	Description	Type
`__bank`*n*	Defines the SFR area, n=0,1	Data storage
`__code`	Places a variable in the code area (ROM)	Data storage
`__fast`	Enables fast interrupt support. Not for SAM8xRI or SAM8xRCRI.	Special function object attribute
`__generic`	A pointer that can point to all data memory types (IRAM, XRAM, ROM)	Data storage
`__interrupt`	Supports interrupt functions	Special function type
`__intrinsic`	Reserved for compiler internal use only	--
`__monitor`	Supports atomic execution of a function	Special function type
`__near`	Storage and pointer modifier (XRAM)	Data storage
`__no_init`	Supports non-volatile memory	Data storage
`__root`	Ensures that a function or variable is included in the object code even if unused	--
`__tiny`	Storage and pointer modifier (IRAM Page 0)	Data storage
`__tiny_func`	Callable from any segment. Gives a 2 byte vectored CALL instruction. Not for SAM8xRI or SAM8xRCRI.	--
`__tiny2`	Storage and pointer modifier (IRAM Page 0)	Data storage
`__tinyp`	Storage and pointer modifier (IRAM Page 0-15) Cannot be initialized.	Data storage

Table 32: Extended keywords summary

Extended keyword	Description	Type
`__tiny2p`	Storage and pointer modifier (IRAM Page 0-15) Cannot be initialized.	Data storage
`__tinyp`*n*	Storage and pointer modifier (IRAM Page *n* = 1-15) Cannot be initialized.	Data storage
`__tiny2p`*n*	Storage and pointer modifier (IRAM Page *n* = 1-15) Cannot be initialized.	Data storage

Table 32: Extended keywords summary (Continued)

Using extended keywords

This section covers how extended keywords can be used when declaring and defining data and functions. The syntax rules for extended keywords are also described.

In addition to the rules presented here—to place the keyword directly in the code—the directives `#pragma type_attribute` and `#pragma object_attribute` can be used for specifying the keywords. Refer to the chapter *#pragma directives* for details about how to use the extended keywords together with `#pragma` directives.

The keywords and the `@` operator are only available when language extensions are enabled in the SAM8 IAR C Compiler.

In the IAR Embedded Workbench, language extensions are enabled by default.

Use the `-e` compiler option to enable language extensions. See *-e*, page 85 for additional information.

DATA STORAGE

The extended keywords that can be used for data can be divided into four groups that control the following:

- The memory type of objects and pointers: `__tiny`, `__tiny2`, `__tinyp`, `__tinyp`*n*, `__tiny2p`, `__tiny2p`*n*, `__near`, and `__code`
- Other characteristics of objects: `__root` and `__no_init`
- Pointer type only: `__generic`
- Mem type only: `__bank0`, `__bank1`

See the chapter *Data storage* in *Part 1. Using the compiler* for more information about memory types.

Syntax

The keywords follow the same syntax as the type qualifiers `const` and `volatile`. The following declarations place the variable `i` and `j` in near memory. The variables `k` and `l` behave in the same way:

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

Notice that the keyword affects both identifiers.

Pointers

A keyword that is followed by an asterisk (`*`), affects the type of the pointer being declared. A pointer to external RAM 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 tiny memory. Like `p`, `p2` points to a character in near memory.

```
char __near * __tiny p2;
```

Type definitions

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

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

and

```
__near char b;
char __near *bp;
```

FUNCTIONS

The extended keywords that can be used when functions are declared can be divided into three groups:

- Keywords that control the type of the functions. Keywords of this group must be specified both when the function is declared and when it is defined: `__interrupt` and `__monitor`.
- A keyword that controls the behavior of the functions. This keyword is only necessary when the function is defined: `__fast`.
- Keywords that only control the defined function: `__root` and `__noadjust`.

Syntax

The extended keywords are specified before the return type, for example:

```
__interrupt void alpha(void);
```

The keywords that are *type* attributes must be specified both when they are defined and in the declaration. *Object* attributes only have to be specified when they are defined since they do not affect the way an object or function is used.

Descriptions of extended keywords

The following sections give detailed information about each extended keyword.

`__bankn`

The `__bankn` extended keyword defines the SFR area, where *n* can be either `0` or `1`.

`__code`

The `__code` extended keyword places a variable in the code area (ROM from `0` to `0xFFFF`).

Maximum object size

32 Kbytes.

`__fast`

The `__fast` extended keyword enables fast interrupt support.

This keyword cannot be used with the SAM8xRI or SAM8xRCRI cores.

`__generic`

The `__generic` extended keyword is a data pointer type that can point to all data memory areas.

`__interrupt`

The `__interrupt` keyword specifies interrupt functions. The `#pragma vector` directive can be used for specifying the interrupt vector. An interrupt function must have a `void` return type and cannot have any parameters.

The following example declares an interrupt function with interrupt vector with offset `0x14` in the `INTVEC` segment:

```
#pragma vector=0x14
__interrupt void my_interrupt_handler(void);
```

An interrupt function cannot be called directly from a C program. It can only be executed as a response to an interrupt request.

It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table. For additional information, see *INTVEC*, page 68.

The range of the interrupt vectors depends on the device used.

The `iochip.h` header file, which corresponds to the selected derivative, contains predefined names for the existing interrupt vectors.

For additional information, see *Interrupt functions*, page 21.

`__intrinsic` The `__intrinsic` keyword is reserved for compiler internal use only.

`__monitor` The `__monitor` keyword causes interrupts to be disabled during execution of the function. This allows atomic operations to be performed, such as operations on semaphores that control access to resources by multiple processes. A function declared with the `__monitor` keyword is equivalent to any other function in all other respects.

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

For additional information, see the intrinsic functions *__disable_interrupt*, page 124, and *__enable_interrupt*, page 124.

Example

In the following example a semaphore is implemented using one static variable and two monitor functions. A semaphore can be locked by one process and is used for preventing processes to simultaneously use resources that can only be used by one process at a time, for example a printer.

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

/* get_lock -- Try to lock the lock.
 * Return 1 on success and 0 on failure. */

__monitor int get_lock(void)
{
  if (the_lock == 0)
  {
    /* Success, we managed to lock the lock. */
    the_lock = 1;
    return 1;
  }
  else
```

```
  {
    /* Failure, someone else has locked the lock. */
    return 0;
  }
}


/* release_lock -- Unlock the lock. */

__monitor void release_lock(void)
{
  the_lock = 0;
}
```

The following is an example of a program fragment that uses the semaphore:

```
void my_program(void)
{
  if (get_lock())
  {
    /* ... Do something ... */

    /* When done, release the lock. */
    release_lock();
  }
}
```

`__near` The `__near` extended keyword is a storage and pointer modifier.

In the small memory model, the compiler normally places data objects in the `tiny` segment (internal RAM `0x00` to `0xBF`), accessing them by 8-bit addressing, and also allocates space for a `__tiny` address in pointers to such data objects.

The `__near` modifier allows you to place a data object in external RAM, or to specify that a pointer is to point to a data object in external RAM using 16-bit addressing.

When using the large memory model, `__near` is the default.

Maximum object size

32 Kbytes.

`__no_init` The `__no_init` keyword is used for suppressing initialization of a variable at system startup.

The `__no_init` keyword is placed in front of the type. In this example, `settings` is placed in the non-initialized segment:

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

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

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

Note: The `__no_init` keyword cannot be used in `typedefs`.

`__root` The `__root` attribute can be used on either a function or a variable to ensure that, when the module containing the function or variable is linked, the function or variable is also included, whether or not it is referenced by the rest of the program.

By default only the part of the runtime library calling `main` and any interrupt vectors are root. All other functions and variables are included in the linked output only if they are referenced by the rest of the program.

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

```
__root int settings[10];
```

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

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

Note: The `__root` keyword cannot be used in `typedefs`.

`__tiny` The `__tiny` extended keyword is a storage and pointer modifier.

When using the large memory model, the compiler normally places data objects in the `near` segment (external RAM `0` to `0xFFFF`), accessing them by 16-bit addressing, and allocating space for a `__near` address in pointers to such data types.

The `__tiny` modifier allows you to place a data object in internal RAM (`0x00` to `0xBF`, page 0), so it is accessed by the more efficient 8-bit addressing, or to specify that a pointer is to point to a data object in internal RAM. This lets you place frequently-accessed variables so they will be accessed more efficiently, and so that pointers to them will occupy 8 rather than 16 bits.

When using the small memory model, `__tiny` is the default.

Maximum object size

128 bytes.

`__tiny_func` The `__tiny_func` extended keyword is callable from any segment and gives a 2 byte vectored `CALL` instruction.

This keyword cannot be used with the SAM8xRI or SAM8xRCRI cores.

`__tiny2` The `__tiny2` extended keyword is a storage and pointer modifier.

The `__tiny2` modifier allows you to place a data object in the `tiny2` segment (internal RAM `0xC0` to `0xFF`, page 0). Data objects of this type can only be accessed indirectly.

Maximum object size

64 bytes.

`__tinyp` The `__tinyp` extended keyword is a storage and pointer modifier.

The `__tinyp` modifier allows you to place a data object in the `tinyp` segment (internal RAM `0x00` to `0xBF`, page 0 to 15).

Maximum object size

128 bytes.

`__tiny2p` The `__tiny2p` extended keyword is a storage and pointer modifier.

The `__tiny2p` modifier allows you to place a data object in the `tiny2p` segment (internal RAM `0xC0` to `0xFF`, page 0 to 15).

Maximum object size

64 bytes.

`__tinypn` The `__tinypn` extended keyword is a storage and pointer modifier.

The `__tinypn` modifier allows you to place a data object in the `tinypn` segment (internal RAM `0x00` to `0xBF`, page *n*, *n* = 1 to 15).

Maximum object size

128 bytes.

`__tiny2pn` The `__tiny2pn` extended keyword is a storage and pointer modifier.

The `__tiny2pn` modifier allows you to place a data object in the `tiny2pn` segment (internal RAM `0xC0` to `0xFF`, page *n*, *n* = 1 to 15).

Maximum object size

64 bytes.

#pragma directives

This chapter describes the #pragma directives of the SAM8 IAR C Compiler.

The #pragma directives control the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages. The #pragma directives are preprocessed, which means that macros are substituted in a #pragma directive.

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

Summary of #pragma directives

The following table shows the `#pragma` directives of the compiler:

#pragma directive	Description
`#pragma bitfields`	Controls the order of bitfield members
`#pragma constseg`	Places constant variables in a named segment
`#pragma dataseg`	Places variables in a named segment
`#pragma diag_default`	Changes the severity level of diagnostic messages
`#pragma diag_error`	Changes the severity level of diagnostic messages
`#pragma diag_remark`	Changes the severity level of diagnostic messages
`#pragma diag_suppress`	Suppresses diagnostic messages
`#pragma diag_warning`	Changes the severity level of diagnostic messages
`#pragma inline`	Inlines a function
`#pragma language`	Controls the IAR language extensions
`#pragma location`	Specifies the absolute address of a variable
`#pragma message`	Prints a message
`#pragma object_attribute`	Changes the definition of a variable or a function
`#pragma optimize`	Specifies type and level of optimization
`#pragma required`	Introduces a requirement
`#pragma rtmodel`	Inserts a runtime model attribute

Table 33: #pragma directives summary

#pragma directive	Description
`#pragma segment`	Specifies a segment name
`#pragma type_attribute`	Changes the declaration and definitions of a variable or function
`#pragma vector`	Specifies the vector of an interrupt or trap function

Table 33: #pragma directives summary (Continued)

Note: For portability reasons, some old-style `#pragma` directives are recognized but will give a diagnostic message. It is important to be aware of this if you need to port existing code that contains any of those `#pragma` directives. For additional information, see the chapter *Migrating to the SAM8 IAR C Compiler V2.x*.

Descriptions of #pragma directives

This section gives detailed information about each `#pragma` directive.

All `#pragma` directives using `=` for value assignment should be entered like:

`#pragma` *`pragmaname`*`=`*`pragmavalue`*

or

`#pragma` *`pragmaname`* `=` *`pragmavalue`*

#pragma bitfields

`#pragma bitfields={reversed|default}`

The `#pragma bitfields` directive controls the order of bitfield members.

By default the SAM8 IAR C Compiler places bitfield members from the least significant bit to the most significant bit in the container type. Use the `#pragma bitfields=reversed` directive to place the bitfield members from the most significant to the least significant bit. This setting remains active until you turn it off again with the `#pragma bitfields=default` directive.

#pragma constseg

The `#pragma constseg` directive places constant variables in a named segment. Use the following syntax:

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

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

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

```
#pragma constseg=__code MyOtherSeg
```

All constants defined following this directive will be placed in the segment `MyOtherSeg` and accessed using near addressing.

#pragma dataseg

The `#pragma dataseg` directive places variables in a named segment. Use the following syntax:

```
#pragma dataseg=MY_SEGMENT
__no_init char myBuffer[1000];
#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 and must thus not have any initializer.

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

```
#pragma dataseg=__near MyOtherSeg
```

All variables in `MyOtherSeg` will be accessed using near addressing.

#pragma diag_default

`#pragma diag_default=`*`tag`*`,`*`tag`*`,...`

Changes the severity level back to default or as defined on the command line for the diagnostic messages with the specified tags. See the chapter *Diagnostics* for more information about diagnostic messages.

Example

```
#pragma diag_default=Pe117
```

#pragma diag_error

`#pragma diag_error=`*`tag`*`,`*`tag`*`,...`

Changes the severity level to `error` for the specified diagnostics. See the chapter *Diagnostics* for more information about diagnostic messages.

Example

```
#pragma diag_error=Pe117
```

#pragma diag_remark

```
#pragma diag_remark=tag,tag,...
```

Changes the severity level to `remark` for the specified diagnostics. For example:

```
#pragma diag_remark=Pe177
```

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

#pragma diag_suppress

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

Suppresses the diagnostic messages with the specified tags. For example:

```
#pragma diag_suppress=Pe117,Pe177
```

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

#pragma diag_warning

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

Changes the severity level to `warning` for the specified diagnostics. For example:

```
#pragma diag_warning=Pe826
```

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

#pragma inline

```
#pragma inline[=forced]
```

The `#pragma inline` directive advises the compiler that the function whose declaration follows immediately after the directive should be inlined—that is, expanded into the body of the calling function. Whether the inlining actually takes place is subject to the compiler's heuristics.

This is similar to the C++ keyword `inline`, but has the advantage of being available in C code.

Specifying `#pragma inline=forced` disables the compiler's heuristics and forces the inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question—like `printf`—an error message is emitted.

#pragma language

```
#pragma language={extended|default}
```

The `#pragma language` directive is used for turning on the IAR language extensions or for using the language settings specified on the command line:

`extended`	Turns on the IAR language extensions and turns off the `--strict_ansi` command line option.
`default`	Uses the settings specified on the command line.

`#pragma location`

`#pragma location=address`

The `#pragma location` directive specifies the location—the absolute address—of the variable whose declaration follows the `#pragma` directive. For example:

```
#pragma location=0xFF2000
char PORT1; /* PORT1D is located at address 0xFF2000 */
```

The directive can also take a string specifying the segment placement for either a variable or a function, for example:

```
#pragma location="foo"
```

For additional information and examples, see *Absolute location placement*, page 17 and *Segment placement*, page 17.

`#pragma message`

`#pragma message(message)`

Makes the compiler print a message on `stdout` when the file is compiled. For example:

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

`#pragma object_attribute`

`#pragma object_attribute=keyword`

The `#pragma object_attribute` directive affects the declaration of the identifier that follows immediately after the directive.

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

`__no_init`	Suppresses initialization of a variable at startup.

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

`__root`	Ensures that a function or data object is included in the linked application even if not referenced.

Example

In the following example, the variable `bar` is placed in the non-initialized segment:

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

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

```
__no_init char bar;
```

#pragma optimize

#pragma optimize=*token token token*

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

s	Optimizes for speed
z	Optimizes for size
2|3|6|9	Specifies level of optimization
no_cse	Turns off common sub-expression elimination
no_inline	Turns off function inlining
no_unroll	Turns off loop unrolling
no_code_motion	Turns off code motion.

The #pragma optimize directive is used for decreasing the optimization level or for turning off some specific optimizations. This #pragma directive only affects the function that follows immediately after the directive.

Notice that 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.

Note: If you use the #pragma optimize directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the #pragma directive is ignored.

Example

```
#pragma optimize=s 9
int small_and_used_often()
{
    ...
}

#pragma optimize=z 9
int big_and_seldom_used()
{
    ...
}
```

#pragma required

#pragma required

The #pragma required directive will introduce a requirement from a symbol to another symbol. That is, if the first symbol is in the produced output when linking, the other symbol should also be in that output. This is useful if, for example, a function that handles certain data should only be used if there is any data to handle.

Syntax

`#pragma required=`*symbol*

where *symbol* is any statically linked function or variable. The #pragma must be placed in front of a symbol definition.

Example

```
void f(void)
{
// handle segment S here
}
#pragma required=f // l requires f
long l @ "S"; // l resides in S
```

#pragma rtmodel

`#pragma rtmodel("`*key*`","`*value*`")`

The #pragma rtmodel directive inserts the runtime model attribute *key* with the value *value*. It must be followed by a variable, since the pragma directive is associated with a variable. Keys beginning with __ are reserved by the compiler.

```
#pragma rtmodel("myattr","blue")
char is_blue=1;
```

The runtime model attribute is then passed to the linker. If the same key is found in another file, it must have the same value, otherwise it will not link. See *RTMODEL* in the *SAM8 IAR Assembler Reference Guide* for a more detailed explanation.

#pragma segment

`#pragma segment=`*<name>* [*<memory>*]

The #pragma segment directive declares a segment name that can be used in the segment operators __segment_begin and __segment _end. The optional *<memory>* must, if present, be a memory attribute and will be used in the return type of the operators.

```
__segment_begin(<name>)
__segment_end(<name>)
```

Example

`__segment_begin("MYSEG")`

The `__segment_begin` operator denotes the start address of the segment with the name *<name>*, which must be a string literal, and must have been declared in a segment pragma at an earlier point in the compilation unit. The `__segment_end` operator denotes the address immediately after the last byte in the segment. The type of these operators is pointer to void. If a memory attribute was entered in the segment pragma declaring the segment, the type is pointer to *<memory>* void, otherwise the type is a default pointer to void. The operator given in the example has the type "`void __huge *`".

#pragma type_attribute

`#pragma type_attribute=`*`keyword`*

The `#pragma type_attribute` directive affects the declaration of the identifier, the next variable, or the next function, that follows immediately after the `#pragma` directive. It only affects the variable, not its type.

All memory attributes can be used with the `#pragma type_attribute` directive for a variable.

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

`__interrupt`	Specifies interrupt functions. Use the `#pragma vector` directive to specify the interrupt vector; see page 117.
`__monitor`	Specifies a monitor function

For interrupt and trap, use the `#pragma vector` directive to specify the exception vector.

Example

In the following example, `myBuffer` is placed in `near` memory, whereas the variable `i` is not affected by the `#pragma` directive.

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

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

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

In the small data model, the default pointer is `__tiny`. In the following example, the pointer is located in near memory, pointing at `__tiny`:

```
#pragma type_attribute=__near
int * pointer;
```

`#pragma vector`

```
#pragma vector=vector
```

The `#pragma vector` directive specifies the vector of an interrupt or trap function whose declaration follows the `#pragma` directive.

Example

```
#pragma vector=0x14
__interrupt void my_handler(void);
```

Predefined symbols

This chapter gives reference information about the predefined preprocessor symbols that are supported in the SAM8 IAR C Compiler. These symbols allow you to inspect the compile-time environment, for example the time and date of compilation.

Summary of predefined symbols

The following table summarizes the predefined symbols:

Predefined symbol	Identifies
`__CODE_MODEL__`	The code model in use
`__CORE__`	The chip core in use
`__DATA_MODEL__`	The data model in use
`__DATE__`	The date of compilation
`__FILE__`	The name of the file being compiled
`__IAR_SYSTEMS_ICC__`	The IAR compiler platform
`__ICCSAM8__`	The SAM8 IAR C Compiler
`__LINE__`	The current source line number
`__LITTLE_ENDIAN__`	The byte order used
`__STDC__`	ISO/ANSI Standard C
`__STDC_VERSION__`	The version of ISO/ANSI Standard C in use
`__TID__`	The target processor of the IAR compiler in use
`__TIME__`	The time of compilation
`__VER__`	The version number of the IAR compiler in use

Table 34: Predefined symbols summary

Note: The predefined symbol `__TID__` is available for backwards compatibility. We recommend that you use the symbols `__CODE_MODEL__`, `__CORE__`, `__DATA_MODEL__`, and `__ICCSAM8__` instead.

Descriptions of predefined symbols

The following section gives reference information about each predefined symbol.

`__CODE_MODEL__`

Use this symbol to identify the used code model.

The value of this symbol is `__CODE_MODEL_SMALL__` or `__CODE_MODEL_LARGE__` for the small and large code models, respectively.

Example

```
#if __CODE_MODEL__==__CODE_MODEL_SMALL__
int my_array[10];
#else
int my_array[20];
#endif
```

`__CORE__`

Identifies the chip core used.

This can be either of `__SAM8__`, `__SAM8X__`, `__SAM8XRC__`, `__SAM8XRI__`, or `__SAM8XRCRI__`.

`__DATA_MODEL__`

Use this symbol to identify the used data model.

The value of this symbol is `__DATA_MODEL_SMALL__` or `__DATA_MODEL_LARGE__` for the small and large data models, respectively.

Example

```
#if __DATA_MODEL__==__DATA_MODEL_SMALL__
... code used in small data model only
#endif
```

`__DATE__`

Use this symbol to identify when the file was compiled. This symbol expands to the date of compilation which is returned in the form `"Mmm dd yyyy"`, for example `"Nov 30 2003"`.

`__FILE__`

Use this symbol to identify which file is currently being compiled. This symbol expands to the name of that file.

`__IAR_SYSTEMS_ICC__` This predefined symbol expands to a number that identifies the IAR compiler platform. The current identifier is 5. Note that the number could be higher in a future version of the product.

This symbol can be tested with `#ifdef` to detect that the code was compiled by a compiler from IAR Systems.

`__ICCSAM8__` This predefined symbol expands to the number 1 when the code is compiled with the SAM8 IAR C Compiler.

`__LINE__` This predefined symbol expands to the current line number of the file currently being compiled.

`__LITTLE_ENDIAN__` This predefined symbol expands to 0 (false), as SAM8 is big endian.

`__STDC__` This predefined symbol expands to the number 1. This symbol can be tested with `#ifdef` to detect that the compiler in use adheres to ISO/ANSI C.

`__STDC_VERSION__` ISO/ANSI C and version identifier.

This predefined symbol expands to the number 199409L.

`__TID__` Target identifier for the SAM8 IAR C Compiler.

Included, but obsolete. In this version, the symbols `__CODE_MODEL__`, `__CORE__`, `__DATA_MODEL__`, and `__ICCSAM8__` are used instead.

`__TIME__` Current time.

Expands to the time of compilation in the form `hh:mm:ss`.

`__VER__` Compiler version number.

Expands to an integer representing the version number of the compiler.

Example

The example below prints a message for version 3.34:

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

Intrinsic functions

This chapter gives reference information about the intrinsic functions.

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into in-line code, either as a single instruction or as a short sequence of instructions.

Intrinsic functions summary

The following table summarizes the intrinsic functions:

Intrinsic function	Description
`__disable_interrupt`	Disables interrupts
`__enable_interrupt`	Inserts an `EI` instruction
`__get_interrupt_state`	Returns the interrupt state
`__idle`	Inserts an idle instruction
`__no_operation`	Generates a `NOP` instruction
`__segment_begin`	Returns the start address of a segment
`__segment_end`	Returns the end address of a segment
`__set_interrupt_state`	Restores the interrupt state
`__stop`	Inserts a `STOP` instruction
`__wait_for_interrupt`	Inserts a `WFI` instruction

Table 35: Intrinsic functions summary

To use intrinsic functions in an application, include the header file `intrinsics.h`.

Note that the intrinsic function names start with double underscores, for example:

```
__enable_interrupt
```

Descriptions of intrinsic functions

The following section gives reference information about each intrinsic function.

__disable_interrupt

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

Disables interrupts by inserting the `DI` instruction.

__enable_interrupt

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

Enables interrupts by inserting the `EI` instruction.

__get_interrupt_state

```
istate_t __get_interrupt_state(void);
```

The return value is a target-specific value that specifies the current state of global interrupts. For the simplest targets, it will only be the value of a global interrupt bit. The only intended use for the return value is to be a parameter to `__set_interrupt_state(istate_t)`, which will restore the interrupt state.

__idle

```
void __idle(void);
```

Inserts an `idle` instruction.

__no_operation

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

Generates a `NOP` instruction.

__segment_begin

```
__segment_begin(segment);
```

Returns the address of the first byte of the named *`segment`*. The named *`segment`* must be a string literal that has been declared earlier with the `#pragma segment` directive. See *#pragma segment*, page 115.

If the segment was declared with a memory attribute *`memattr`*, the type of the `__segment_begin` function is pointer to *`memattr`* `void`. Otherwise, the type is a default pointer to `void`.

Example

```
#pragma segment="MYSEG" __huge
...
__segment_begin("MYSEG")
```

Here the type of the `__segment_begin` intrinsic function is `void __huge *`.

`__segment_end` `__segment_end(segment);`

Returns the address of the first byte *after* the named *`segment`*. The named *`segment`* must be a string literal that has been declared earlier with the `#pragma segment` directive. See *#pragma segment*, page 115.

If the segment was declared with a memory attribute *`memattr`*, the type of the `__segment_end` function is pointer to *`memattr`* `void`. Otherwise, the type is a default pointer to `void`.

Example

```
#pragma segment="MYSEG" __huge
...
__segment_end("MYSEG")
```

Here the type of the `__segment_end` intrinsic function is `void __huge *`.

`__set_interrupt_state` `void __set_interrupt_state(istate_t);`

Restores the interrupt status that was saved by `__get_interrupt_state(void)`.

`__stop` `void __stop(void);`

Inserts a `stop` instruction.

`__wait_for_interrupt` `void __wait_for_interrupt(void);`

Inserts a `WFI` instruction.

Library functions

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

For detailed information about the library functions, see the online documentation supplied with the product.

The SAM8 IAR C Compiler provides one library: the IAR CLIB Library. The list of header files provided by this library is presented in this chapter.

Introduction

One library is provided by the SAM8 IAR C Compiler:

- IAR CLIB Library is the traditional C library used by IAR Systems. Basically it implements the free-standing part of C. This library is not fully ISO/ANSI-compliant.

IAR CLIB library

The SAM8 IAR C Compiler package provides most of the important C library definitions that apply to embedded systems. These are of the following types:

- Standard C library definitions available for user programs. These are documented in this chapter.
- `CSTARTUP`, the single program module containing the start-up code. It is described in the *Runtime environment* chapter in this guide.
- Runtime support libraries; for example, low-level floating-point routines.
- Intrinsic functions, allowing low-level use of SAM8 features. See the chapter *Intrinsic functions* for more information.

LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. There are some I/O-oriented routines (such as `putchar` and `getchar`) that you need to customize before using them in 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 that may contain additional target-specific definitions.

Header file	Description
`assert.h`	Assertions.
`ctype.h`	Character handling.
`iccbutl.h`	Low-level routines.
`math.h`	Mathematics.
`setjmp.h`	Non-local jumps.
`stdarg.h`	Variable arguments.
`stdio.h`	Input/output.
`stdlib.h`	General utilities.
`string.h`	String handling.

Table 36: IAR CLIB Library header files

The following table shows header files that do not contain any functions, but specify various definitions and data types:

Header file	Description
`errno.h`	Error return values.
`float.h`	Limits and sizes of floating-point types.
`limits.h`	Limits and sizes of integral types.
`stddef.h`	Common definitions including `size_t`, `NULL`, `ptrdiff_t`, and `offsetof`.

Table 37: Miscellaneous IAR CLIB Library header files

RESTRICTIONS ON ANSI C LIBRARIES

The default data pointer cannot point to a variable on the stack when:

- the code model is small and the data model is large
- the code model is large and the data model is small.

When a program uses the default data pointer to point to a stack element, the compiler will generate an error.

Because of this, the following C library functions:

```
log
printf
qsort
scanf
sinus
sprintf
sqrt
sscanf
tan
```

are not available in the following libraries:

```
clsam8sl.r18
clsam8sle.r18
clsam8xsl.r18
clsam8xsle.r18
clsam8xrisl.r18
clsam8xrisle.r18
clsam8ls.r18
clsam8xls.r18
clsam8lse.r18
clsam8xlse.r18
```

Diagnostics

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

Message format

All diagnostic messages are issued as complete, self-explanatory messages. A typical diagnostic message from the 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; *message* is a self-explanatory message, possibly several lines long.

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

Severity levels

The diagnostics are divided into different levels of severity:

Remark

A diagnostic message 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 94.

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 use of the command-line option `--no_warnings`, see page 92.

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. An error will produce a non-zero exit code.

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 *Options summary*, page 77, for a description of the compiler options that are available for setting severity levels.

See the chapter *#pragma directives*, for a description of the `#pragma` directives that are available for setting severity levels.

INTERNAL ERROR

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

```
Internal error: message
```

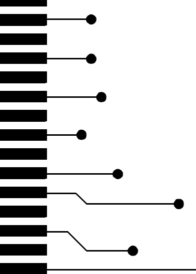
where *message* is an explanatory message. If internal errors occur, they should be reported to your software distributor or IAR Technical Support. Please include information enough to reproduce the problem. This would typically include:

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

Part 3. Migration and portability

This part of the SAM8 IAR C Compiler Reference Guide contains the following chapters:

- Migrating to the SAM8 IAR C Compiler V2.x
- Implementation-defined behavior
- IAR C extensions.



Migrating to the SAM8 IAR C Compiler V2.x

C source code that was originally written for the SAM8 IAR C Compiler V1.x can also be used with the SAM8 IAR C Compiler V2.x, although some modifications may be required.

This chapter very briefly describes issues to keep in mind when migrating from V1.x to V2.x.

Differences

- V2.x is by default more strictly ISO/ANSI-compliant than V1.x.
- The following V1.x preprocessor functions are no longer supported in V2.x:
  - using the `sizeof()` operator
  - using floats in `#if`-statements
  - using all symbol names
- The following keywords in V1.x are supported in V2.x, but are prefixed with double underscores:

  `code`, `fast`, `generic`, `interrupt`, `monitor`, `near`, `no_init`, `tiny`, `tiny2`, `tinyp`, `tiny2p`, and `tiny_func`

  For example, the keyword `near` in V1.x is now `__near` in V2.x.
- The following `#pragma` directives in V1.x have a new syntax in V2.x:

#pragma	The new syntax uses:
`memory`	`type_attribute`, `object_attribute`, `constseg`, or `dataseg`
`function`	`type_attribute` or `object_attribute`
`warnings`	`diag_xxx`, where *xxx* is one of `suppress`, `remark`, `warning`, `error`, or `default`
`codeseg`	`location`

Table 38: `#pragma` *directives in V1.x with new syntax*

- The following keywords/constructions in V1.x are no longer supported in V2.x:

`sfr`, `sfrp`	Declaration of SFRs is done by using absolute declarations with the `@` or `#pragma location` syntax.
`interrupt [nn]`	The vector number `[nn]` has to be declared using the `#pragma vector` directive.

- The following files from V1.x need to be modified for use in V2.x:
  - Linker files (the segment names are changed)
  - Make files for the command line tools (the command line options are changed)
- The memory model is split into code and data model
- The V1.x assembler processor variant options, `-v0` to `-v5`, have been changed to:

V1.x assembler	V2.x assembler
`-v0`	`-v0`
`-v1`	`-v0`
`-v2`	`-v3`
`-v3`	`-v1`
`-v4`	`-v1`
`-v5`	`-v2`
n/a	`-v4`

Table 39: Assembler processor option mappings

- The V1.x compiler processor variant options, `-v0` to `-v5`, have been changed to:

V1.x compiler	V2.x compiler
`-v0`	`--core SAM8`
`-v1`	`--core SAM8 --enable_eeprom_support`
`-v2`	`--core SAM8xRI`
`-v3`	`--core SAM8x`
`-v4`	`--core SAM8x --enable_eeprom_support`
`-v5`	`--core SAM8xRC`
n/a	`--core SAM8xRCRI`

Table 40: Compiler processor option mappings

- The V1.x compiler memory model options, -ms and -ml, have been changed to:

V1.x compiler	V2.x compiler
-ms	--code_model_small --data_model_small (internal stack, internal data)
-ml	--code_model_large --data_model_large (external stack, external data)
n/a	--code_model_small --data_model_large (internal stack, external data)
n/a	--code_model_large --data_model_small (external stack, internal data)

Table 41: Compiler memory model option mappings

- A few other V1.x compiler options have been changed:

V1.x compiler	V2.x compiler
-h	--generate_tinyfunc_runtime_library_calls
-u	n/a
-E{wod}	n/a
-j	n/a

Table 42: Other compiler option mappings

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.

Descriptions of implementation-defined behavior

This section follows the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

TRANSLATION

Diagnostics (5.1.1.3)

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 (5.1.2.2.2.1)

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 *Customizing cstartup.s18*, page 48.

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 message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character, generates a diagnostic message.

Converting multibyte characters (6.1.3.4)

The 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 the two's complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.

See *Data types*, page 61, 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 62, for information about the ranges and sizes for the different floating-point types: `float` and `double`.

Converting integer values to floating-point values (6.2.1.3)

When an integral number is cast to a floating-point value that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

Demoting floating-point values (6.2.1.4)

When a floating-point value is converted to a floating-point value of narrower type that cannot exactly represent the value, the value is rounded(up or down) to the nearest suitable value.

ARRAYS AND POINTERS

size_t (6.3.3.4, 7.1.1)

See *size_t*, page 63, for information about `size_t` in IAR C.

Conversion from/to pointers (6.3.4)

See *Casting*, page 63, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 63, 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 it own choices when optimizing.

STRUCTURES, UNIONS, ENUMERATIONS, AND BITFIELDS

Improper access to a union (6.3.2.3)

If a union get 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 types*, page 61, 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
dataseg
define_type_info
diag_default
diag_error
diag_remark
diag_suppress
diag_warning
do_not_instantiate
function
hdrstop
inline
instantiate
language
location
memory
message
none
no_pch
NOTREACHED
```

```
object_attribute
once
optimize
__printf_args
__scanf_args
type_attribute
VARARGS
vector
warnings
```

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

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)

The `system()` function is not supported in IAR C.

Message returned by strerror() (7.11.6.2)

The messages returned by `strerror()` depending on the argument are:

Argument	Message
`EZERO`	`no error`
`EDOM`	`domain error`
`ERANGE`	`range error`
`<0 || >99`	`unknown error`
all others	`error No.xx`

Table 43: Message returned by strerror()

The time zone (7.12.1)

The time zone function is not supported in IAR C.

clock() (7.12.2.1)

The `clock()` function is not supported in IAR C.

IAR C extensions

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

In the IAR Embedded Workbench™ IDE, language extensions are enabled by default.

See the compiler options -e on page 85 and --strict_ansi on page 95 for information about enabling and disable language extensions from the command line.

Why should language extensions be used?

By using language extensions, you gain full control over the resources and features of the target microcontroller, and can thereby fine-tune your application.

If you want to use the source code with different compilers, note that language extensions may cause minor modifications before the code can be compiled. A compiler typically supports microcontroller-specific language extensions as well as vendor-specific ones.

Descriptions of language extensions

The language extensions can be categorized into different groups according to their functionality.

Memory, type, and object attributes

Entities such as variables and functions may be declared with memory, type, and object attributes. The syntax follows the syntax for qualifiers—such as `const`—but the semantics is different.

- A memory attribute controls the placement of the entity. There can be only one memory attribute.
- A type attribute controls other aspects of the object. There can be many different type attributes and they must be included when the object is declared.
- An object attribute only has to be specified at the definition but not at the declaration of an object.

See the *Extended keywords* chapter for a complete list of attributes.

Absolute placement

The operator `@` or the directive `#pragma location` can be used for specifying either the location of an absolute addressed variable or the segment placement of a variable or function. For example:

```
no_init int x @ 0x1000;

void test(void) @ "MYOWNSEGMENT"
{
  ...
}
```

_Pragma

The preprocessor operator `_Pragma` can be used in defines and has the equivalent effect of the pragma directive. The syntax is:

```
_Pragma("string")
```

where *`string`* follows the syntax for the corresponding pragma directive. For example:

```
#if NO_OPTIMIZE
  #define NOOPT _Pragma("optimize=2")
#else
  #define NOOPT
#endif
```

See the chapter *Pragma directives*.

Variadic macros

Variadic macros are the preprocessor macro equivalent to `printf` style functions.

Syntax

```
#define P(...)       __VA_ARGS__
#define P(x,y,...)   x + y + __VA_ARGS__
```

Here, `__VA_ARGS__` will contain all variadic arguments concatenated together, including the separating commas.

Example

```
#if DEBUG
  #define DEBUG_TRACE(...) printf(S,__VA_ARGS__)
#else
  #define DEBUG_TRACE(...)
#endif
...
```

```
DEBUG_TRACE("The value is:%d\n",value);
```

will result in:

```
printf("The value is:%d\n",value);
```

Inline assembler

Inline assembler can be used for inserting assembler instructions into the generated function. This is seldom needed since almost all can be expressed in C with the help of intrinsic functions.

The syntax for inline assembler is:

```
asm("LD R4,R7");
```

In strict ISO/ANSI mode the use of inline assembler is disabled.

C++ style comments

C++ style comments are accepted. A C++ style comment starts with the character sequence `//` and continues to the end of the line. For example:

```
// The length of the bar, in centimeters.
int length;
```

__ALIGNOF__

Every C data object has an alignment that controls how the object can be stored in memory. Should an object have an alignment of, say four, it must be stored on an address that is dividable by four.

The reason for the concept of alignment is that some processors have hardware limitations for how the memory can be accessed.

Assume that a processor can read 4 bytes of memory using one instruction but only when the memory read is placed on an address dividable by 4. Then 4-byte objects, such as `long` integers, will have alignment 4.

Another processor might only be able to read 2 bytes at a time; in that environment the alignment for a 4-byte `long` integer might be 2.

A structure type will inherit the alignment from its components.

All objects must have a size that is a multiple of the alignment. If is not true, only the first element of an array would be placed in accordance with the alignment requirements.

In the example below, the alignment of the structure is 4, under the assumption that `long` has alignment 4. Its size is 8, even though only 5 bytes are effectively used.

```
struct str {
  long a;
  char b;
};
```

In standard C, the size of an object can be accessed using the `sizeof` operator.

The `__ALIGNOF__` operator can be used to access the alignment of an object. It can take two forms:

- `__ALIGNOF__` (type)
- `__ALIGNOF__` (expression)

In the second form the expression is not evaluated.

Anonymous structs and unions

C++ includes a feature named anonymous unions. The IAR Systems compilers allow a similar feature for both structs and unions.

An anonymous structure type (i.e. one without a name) defines an unnamed object (and not a type) whose members are promoted to the surrounding scope. External anonymous structure types are allowed.

For example, the structure `str` below contains an anonymous union. The members of the union are accessed using the names `b` and `c`, for example `obj.b`.

Without anonymous structure types the union would have to be named—for example `u`—and the member elements accessed using the syntax `obj.u.b`.

```
struct str
{
  int a;
  union
  {
    int b;
    int c;
  }
};

struct str obj;
```

Bitfields and non-standard types

In ISO/ANSI C, a bitfield must be of type `int` or `unsigned int`. Using IAR extensions any integer type and enums may be used.

For example, in the following structure an `unsigned char` is used for holding three bits. The advantage is that the struct will be smaller.

```
struct str
{
  unsigned char bitOne   : 1;
  unsigned char bitTwo   : 1;
  unsigned char bitThree : 1;
};
```

This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*.

Incomplete arrays at end of structs

The last element of a `struct` may be an incomplete array. This is useful since one chunk of memory can be allocated for the `struct` itself and for the array, regardless of the size of the array.

Note: The array may not be the only member of the `struct`. If that were the case, then the size of the `struct` would be zero, which is not allowed in ISO/ANSI C.

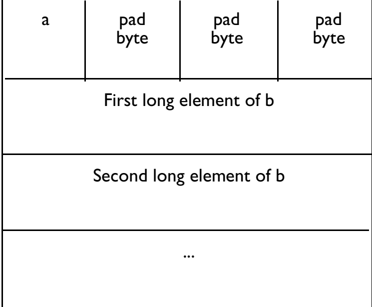
Example

```
struct str
{
  char a;
  unsigned long b[];
};

struct str * GetAStr(int size)
{
  return malloc(sizeof(struct str)
              + sizeof(unsigned long)*size);
}

void UseStr(struct str * s)
{
  s->b[10] = 0;
}
```

The `struct` will inherit the alignment requirements from all elements, including the alignment of the incomplete array. The array itself will not be included in the size of the struct. However, the alignment requirements will ensure that the struct will end exactly at the beginning of the array; this is known as padding.

In the example above the alignment of `struct str` will be 4 and the size is also 4. (Assuming a processor where the alignment of `unsigned long` is 4.)

The memory layout of `struct str` is:



Arrays of incomplete types

An array may have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), and by the end of the compilation unit if it is not.

Empty translation units

A translation unit (source file) is allowed to be empty, i.e. it does not contain any declarations.

In strict ISO/ANSI mode a warning is issued if the compilation unit is empty.

Example

The following source file is only used in a debug build. (In a debug build the `NDEBUG` preprocessor flag is undefined.) Since the entire content of the file is conditionally compiled using the preprocessor, the translation unit will be empty when the application is compiled in release mode. Without this extension, this would be considered an error.

```
#ifndef NDEBUG

void PrintStatusToTerminal()
{
  /* Do something */
}

#endif
```

Comments at the end of preprocessor directives

This extension, which makes it legal to place text after preprocessor directives, is enabled unless strict ISO/ANSI mode is used. This language extension exists to support compilation of old legacy code; we do *not* recommend you to write new code in this fashion.

Example

```
#ifdef FOO

  ... something ...

#endif FOO /* This is allowed but not recommended. */
```

Forward declaration of enums

The IAR Systems language extensions allow that you first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.

Extra comma at end of enum list

It is allowed to place an extra comma at the end of an `enum` list. In strict ISO/ANSI mode a warning is issued.

Note: ISO/ANSI C allows extra commas in similar situations, for example after the last element of the initializers to an array. The reason is that it is easy to get the commas wrong if parts of the list are moved using a normal cut-and-paste operation.

Example

```
enum
{
   kOne,
   kTwo,   /* This is now allowed. */
};
```

Missing semicolon at end of struct or union specifier

A warning is issued if the semicolon at the end of a `struct` or union specifier is missing.

NULL and void

In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ISO/ANSI C some operators allow such things, while others do not allow them.

A label preceding a "}"

In ISO/ANSI C, a label must be followed by at least one statement. Hence it is illegal to place the label at the end of a block. In the SAM8 IAR C Compiler, a warning is issued.

To create a standard-compliant C program (so that you will not have to see the warning) you can place an empty statement after the label. An empty statement is a single `;` (semi-colon).

Example

```
void test()
{
  if (...) goto end;

   /* Do something */

  end:  /* Illegal at the end of block. */
}
```

Note: This also applies to the labels of switch statements.

The following piece of code will generate the warning.

```
switch (x)
{
case 1:
  ...;
  break;

default:
}
```

A good way to convert this into a standard-compliant C program is to place a `break;` statement after the `default:` label.

Empty declarations

An empty declaration (a semicolon by itself) is allowed but a remark is issued (provided that remarks are enabled).

This is useful when preprocessor macros are used that could expand to nothing. Consider the following example. In a debug build the macros `DEBUG_ENTER` and `DEBUG_LEAVE` could be defined to something useful. In a release build, however, they could expand into nothing, leaving the `;` character in the code.

```
void test()
{
  DEBUG_ENTER();
```

```
  do_something();

  DEBUG_LEAVE();
}
```

Single value initialization

ISO/ANSI C requires that all initializer expressions of static arrays, `structs`, and `unions` should be enclosed in braces.

Single-value initializers are allowed to appear without braces, but a warning is issued.

Example

In the SAM8 IAR C Compiler, the following expression is allowed:

```
struct str
{
  int a;
} x = 10;
```

Casting pointers to integers

In an initializer, a pointer constant value may be cast to an integral type if the integral type is large enough to contain it.

In the example below we assume that pointers to `__near` and `__huge` are 16 and 24 bits, respectively. The first initialization is correct since it is possible to cast the 16-bit address to a 16-bit `unsigned short` variable. However, it is illegal to use the 32-bit address of `b` as initializer for a 16-bit value.

```
__near int a;
__huge int b;

unsigned short ap = (unsigned short)&a;  /* Correct */
unsigned short bp = (unsigned short)&b;  /* Error */
```

Casting integers to pointers and back in constant expressions

In constant integer expressions, it is allowed to cast an integer to a pointer and back.

Hexadecimal floating point constants

Floating point constants can be given in hexadecimal style. The syntax is `0x`*`MANT`*`p{+|-}`*`EXP`* where *`MANT`* is the mantissa in hexadecimal digits, including an optional `.` (decimal point), and *`EXP`* is the exponent with decimal digits, representing an exponent of 2.

Examples

`0x1p0` is `1`

`0xA.8p2` is `10.5*2^2`

Taking the address of a register variable

In ISO/ANSI C it is illegal to take the address of a variable specified as a register variable.

The SAM8 IAR C Compiler allows this but a warning is issued.

Duplicated size and sign specifiers

Should the size or sign specifiers be duplicated (for example, `short short` or `unsigned unsigned`), an error is issued.

"long float" means "double"

`long float` is accepted as synonym for `double`.

Repeated typedefs

Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.

Mixing pointer types

Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical, for example, `unsigned char` * and `char` *. This includes pointers to integral types of the same size. A warning is issued.

Assignment of a string constant to a pointer to any kind of character is allowed, and no warning will be produced.

Non-top level const

Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). It is also allowed to compare and take the difference of such pointers.

Declarations in other scopes

External and static declarations in other scopes are visible. In the following example the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

```
int test(int x)
{
  if (x)
  {
    extern int y;
    y = 1;
  }

  return y;
}
```

Non-lvalue arrays

A non-`lvalue` array expression is converted to a pointer to the first element of the array when it is used.

A

absolute location 17
  #pragma location 113
addressing. *See* memory types
alignment 61
anonymous structures 18
applications
  building 3
  initializing 47
  terminating. 48
architecture, SAM8. xiii
ARGFRAME (compiler function directive) 35
arrays
  hints 56
  implementation-defined behavior. 143
asm (inline assembler) 6
assembler directives
  CFI. 33, 50
  ENDMOD 49
  EQU. 94
  MODULE 49
  PUBLIC. 94
  REQUIRE 49
  RSEG. 49
  RTMODEL 26
assembler instructions
  DI. 124
  EI 124
  NOP. 124
  WAIT. 125
assembler labels
  ?C_EXIT 53
  ?C_GETCHAR 53
  ?C_PUTCHAR 53
assembler language interface 25
  creating skeleton code 31
assembler list file 35
assembler routines, calling from C 31
assembler, inline 6, 58
assert.h (library header file) 52, 128
assumptions (programming experience) xiii
atomic operations, performing 103
auto variables 11–12
  saving stack space 56

B

_ _bankn (extended keyword). 102
BANK0_A (segment) 66
BANK1_A (segment) 67
bitfields
  data representation 62
  hints 56
  implementation-defined behavior. 143
bitfields (#pragma directive). 62, 110

C

_ _CODE_MODEL_ _ (predefined symbol). 120
_ _CORE_ _ (predefined symbol). 120
C calling convention 27
C library
  general definitions 51
  introduction 51
call chains 56
call frame information 33, 50
call stack. 33
  displaying 50
callee-save registers, stored on stack. 12
calling convention
  C 27
calloc (standard library function) 14
CALLT_C (segment) 67
casting, of pointers and integers 63
CFI (assembler directive) 33, 50
char (data type), signed and unsigned 62, 79
characters, implementation-defined behavior 140

--char_is_signed (compiler option) 79
CLIB. 127
documentation 127
header files. 128
heap size 44
library object files 127
code
excluding when linking 49
placement of 65
portability 38
startup 45
_ _code (extended keyword) 102
code execution 4
code models 4, 7
characteristics 7
default 7
large 7, 21
overview 21
small 7, 21
specifying on command line 79
code motion, disabling 90
CODE (segment) 45, 67
--code_model (compiler option) 79
_ _code_model (runtime model attribute) 27
common sub-expression elimination, disabling 90
compiler environment variables 77
compiler error return codes 77
compiler listing, generating. 88
compiler object file
including debug information 81, 94
specifying filename 92
compiler options
setting 75
specifying parameters 76
summary 77
typographic convention xvi
-D. 80
-e 85
-f 86
-I 87
-l 32, 88
-o 92
-r 81, 94
-s 95
-z 96
--char_is_signed. 79
--code-model 79
--core 80
--data_model 81
--debug. 81, 94
--dependencies 82
--diagnostics_tables 84
--diag_error 83
--diag_remark 83
--diag_suppress 84
--diag_warning. 84
--enable_eeprom_support. 85
--enable_multibytes 85
--generate_tinyfunc_runtime_library_calls 86
--header_context 87
--library_module 89
--migration_preprocessor_extensions. 89
--module_name 89
--no_code_motion 90
--no_cse 90
--no_inline 91
--no_unroll. 91
--no_warnings 92
--no_wrap_diagnostics. 92
--omit_types. 93
--only_stdout 93
--place_constants_in_rom 93
--preprocess 93
--public_equ. 94
--remarks 94
--silent 95
--strict_ansi 95
--warnings_affect_exit_code 77, 96

--warnings_are_errors 96
compiler version number 121
compiling, from the command line 3
computer style, typographic convention xvi
configuration 7
consistency, module 25
CONST (segment) 67
constseg (#pragma directive) 110
conventions, typographic xvi
copyright notice ii
--core (compiler option) 80
core, specifying on command line 80
CSTACK (segment) 68
example 43
See also stack
CSTACKN (segment) 68
example 43
CSTACK2 (segment) 68
example 43
cstartup, customizing 48, 50–51
cstartup.s18 47
ctype.h (library header file) 52, 128
customization 7
cstartup 48, 50–51
_ _low_level_init 48
C-SPY, low-level interface 53
?C_EXIT (assembler label) 53
?C_GETCHAR (assembler label) 53
C_INCLUDE (environment variable) 77, 87
?C_PUTCHAR (assembler label) 53

D

--data_model (compiler option) 81
_ _DATA_MODEL_ _ (predefined symbol) 120
data
alignment of 61
excluding when linking 49
placement of 65
specifying 7
data memory, specifying 7
data models
characteristics 7
memory attribute, default 8
pointer size, default 8
data representation 61
data storage 4, 11
extended keywords 100
data types 61
floating point 62
integers 61
dataseg (#pragma directive) 111
_ _data_model (runtime model attribute) 27
_ _DATE_ _ (predefined symbol) 120
--debug (compiler option) 81, 94
debug information, including in object file 81, 94
declaration, of functions 27
declarators, implementation-defined behavior 144
--dependencies (compiler option) 82
DI (assembler instruction) 124
diagnostic messages 131
classifying as errors 83
classifying as remarks 83
classifying as warnings 84
disabling warnings 92
disabling wrapping of 92
enabling remarks 94
suppressing 84
diagnostics, listing all used 84
--diagnostics_tables (compiler option) 84
diag_default (#pragma directive) 111
--diag_error (compiler option) 83
diag_error (#pragma directive) 111
--diag_remark (compiler option) 83
diag_remark (#pragma directive) 112
--diag_suppress (compiler option) 84
diag_suppress (#pragma directive) 112
--diag_warning (compiler option) 84

diag_warning #pragma directive) 112
directives
function 35
#pragma 5, 109
_ _disable_interrupt (intrinsic function) 124
disclaimer ii
document conventions. xvi
documentation, library 127
double (data type) 62
dynamic memory 14

E

eeprom support
specifying on command line 85
EI (assembler instruction) 124
--enable_eeprom_support (compiler option) 85
_ _enable_interrupt (intrinsic function). 124
--enable_multibytes (compiler option) 85
ENDMOD (assembler directive). 49
enumerations, implementation-defined behavior. 143
enum, data representation 61
environment
implementation-defined behavior. 140
runtime. 47
environment variables 77
C_INCLUDE. 77, 87
QCCSAM8 77
EQU (assembler directive) 94
errno.h (library header file) 53, 128
error messages 131
classifying 83
error return codes 77
exception vectors 45
experience, programming xiii
extended keywords 99
data storage 100
enabling 85
functions 101
overriding default behaviors. 8
overview 5
summary 99
syntax. 101
using 100
_ _bankn 102
_ _code 102
_ _fast 102
_ _generic 102
_ _interrupt 22, 102
using in #pragma directives. 116–117
_ _intrinsic 103
_ _monitor 103
using in #pragma directives. 116
_ _near 104
_ _no_init. 16, 105
using in #pragma directives. 113
_ _root 105
using in #pragma directives. 113
_ _tiny 105
_ _tinyp 106
_ _tinypn 106
_ _tiny_func. 106
_ _tiny2 106
_ _tiny2p 106
_ _tiny2pn 107

F

-f (compiler option). 86
_ _fast (extended keyword) 102
fast functions. 22
fatal error messages 132
_ _FILE_ _ (predefined symbol). 120
file dependencies, tracking 82
file paths, specifying for #include files 87
filename, of object file 92
float (data type). 62

floating-point constants
hexadecimal notation . . . 159
floating-point format. . . . 62
hints . . . 56
implementation-defined behavior. . . . 142
special cases. . . . 63
32-bits . . . 62
float.h (library header file) . . . 53, 128
formats
floating-point values . . . 62
standard IEEE (floating point) . . . 62
fragmentation, of heap memory . . . 14
free (standard library function) . . . 14
FUNCALL (compiler function directive) . . . 35
function directives. . . . 35
function inlining, disabling . . . 91
function prototypes . . . 55
function type information, omitting in object output. . . . 93
FUNCTION (compiler function directive) . . . 35
functions
declaring . . . 27
executing . . . 11
extended keywords. . . . 101
fast . . . 22
interrupt . . . 21–22
intrinsic . . . 6, 58
I/O . . . 51
monitor . . . 22
omitting type info . . . 93
overview . . . 21
parameters . . . 28
placing in segments . . . 23
recursive. . . . 56
storing data on stack . . . 12–13
return values from . . . 29
special function types. . . . 21

G

--generate_tinyfunc_runtime_library_calls
(compiler option) . . . 86
_ _generic (extended keyword) . . . 102
_ _get_interrupt_state (intrinsic function) . . . 124
glossary . . . xiii
guidelines, reading . . . xiii

H

header files . . . 52
assert.h. . . . 52, 128
CLIB . . . 128
ctype.h . . . 52, 128
errno.h . . . 53, 128
float.h. . . . 53, 128
iccbutl.h . . . 52, 128
limits.h. . . . 53, 128
math.h . . . 52, 128
setjmp.h . . . 52, 128
special function registers . . . 18
stdarg.h . . . 52, 128
stddef.h . . . 53, 128
stdio.h . . . 52, 128
stdlib.h . . . 52, 128
string.h. . . . 52, 128
using as templates . . . 18
--header_context (compiler option). . . . 87
heap . . . 14
size. . . . 44
storing data . . . 11
hidden parameters. . . . 28
hints
migration . . . 135
optimization. . . . 57
programming . . . 55

I

-I (compiler option). 87
IAR CLIB Library. *See* CLIB
IAR Technical Support 132
_ _IAR_SYSTEMS_ICC_ _ (predefined symbol) 121
iccbutl.h (library header file). 52, 128
_ _ICCSAM8_ _ (predefined symbol) 121
identifiers, implementation-defined behavior 140
_ _idle (intrinsic function) 124
IEEE format, floating-point values 62
implementation-defined behavior 139
initialization, modifying cstartup 50–51
inline assembler 6, 58
  See also assembler language interface
inline (#pragma directive). 112
input functions, in standard library 51
instruction set, SAM8 xiii
int (data type) 61
integers 61
  casting 63
  implementation-defined behavior. 141
  intptr_t 63
  ptrdiff_t 63
  size_t 63
  uintptr_t 64
internal error 132
_ _interrupt (extended keyword) 22, 102
  using in #pragma directives 116–117
interrupt functions. 21
  placement in memory. 45
interrupt vector table. 22
  INTVEC segment 68
interrupt vectors, specifying with #pragma directive. 117
interrupts
  disabling 103
  disabling during function execution 22
  processor state 12
intptr_t (integer type) 63
_ _intrinsic (extended keyword) 103
intrinsic functions 58
  overview 6
  summary 123
  _ _disable_interrupt 124
  _ _enable_interrupt 124
  _ _get_interrupt_state. 124
  _ _idle 124
  _ _no_operation 124
  _ _segment_begin 124
  _ _segment_end 125
  _ _set_interrupt_state. 125
  _ _stop 125
  _ _wait_for_interrupt 125
intrinsics.h (header file) 123
INTVEC (segment). 45, 68
ISO/ANSI C
  language extensions 151
  specifying strict usage 95
I/O functions 51

K

keywords, extended. 5, 99

L

-l (compiler option). 32, 88
language extensions
  anonymous structs and unions 19
  descriptions 151
  enabling 85
  overview 5
  using anonymous structures and unions 19
language (#pragma directive) 112
large (code model) 7, 21
large (data model) 8
libraries 3
  runtime. 8

library documentation 127
library functions 127
summary 52, 128
library modules, creating 89
library object files, CLIB 127
--library_module (compiler option) 89
limits.h (library header file) 53, 128
_ _LINE_ _ (predefined symbol) 121
linker configuration files
contents 39
customizing 39–45
introduction 38
template 38
linking, from the command line 4
listing, generating 88
literature, recommended xv
_ _LITTLE_ENDIAN_ _ (predefined symbol) 121
location (#pragma directive) 17, 23, 113
LOCFRAME (compiler function directive) 35
long (data type) 61
loop unrolling, disabling 91
low-level processor operations 6, 123
_ _low_level_init, customizing 48

M

macros
variadic 152
malloc (standard library function) 14
math.h (library header file) 52, 128
memory
access methods 14
accessing 4
dynamic 14
heap 14
non-initialized 16
RAM, saving 56
stack 11
stack, saving 56
static 11, 13
used by executing functions 11
used by global or static variables 13
memory types 13–14
structures 16
summary 15
message (#pragma directive) 113
migration
from SAM8 IAR C Compiler V1.x to V2.x 89, 135
--migration_preprocessor_extensions (compiler option) . . . 89
module consistency 25
module name, specifying 89
MODULE (assembler directive) 49
modules, assembler 49
--module_name (compiler option) 89
_ _monitor (extended keyword) 103
using in #pragma directives 116
monitor functions 22, 103
multibyte character support 85

N

name, specifying for object file 92
_ _near (extended keyword) 104
NEAR_C (segment) 69
NEAR_I (segment) 69
NEAR_ID (segment) 69
NEAR_N (segment) 69
NEAR_Z (segment) 70
non-initialized variables 16
non-scalar parameters 56
NOP (assembler instruction) 124
--no_code_motion (compiler option) 90
--no_cse (compiler option) 90
_ _no_init (extended keyword) 16, 105
using in #pragma directives 113
--no_inline (compiler option) 91
_ _no_operation (intrinsic function) 124
--no_unroll (compiler option) 91

--no_warnings (compiler option) 92
--no_wrap_diagnostics (compiler option) 92
NULL 53, 128

O

-o (compiler option) 92
object filename, specifying 92
object module name, specifying 89
object_attribute (#pragma directive) 16, 113
offsetof 53, 128
--omit_types (compiler option) 93
--only_stdout (compiler option) 93
operators, @ 17, 23
optimization
  code motion, disabling 90
  common sub-expression elimination, disabling 90
  function inlining, disabling 91
  hints 57
  loop unrolling, disabling 91
  size, specifying 96
  speed, specifying 95
  techniques 4, 56
  types and levels 57
optimize (#pragma directive) 114
options summary, compiler 77
output functions, in standard library 51
output, preprocessor 93

P

parameters
  function 28
  hidden 28
  non-scalar 56
  register 28
  specifying 76
  stack 28
  typographic convention xvi
permanent registers 29
placeholder segments 38
placement of code and data 65
--place_constants_in_rom (compiler option) 93
pointers
  casting 63
  implementation-defined behavior 143
  size of 63
  using instead of large non-scalar parameters 56
porting, of code 38
  containing #pragma directives 110
_Pragma (preprocessor operator) 152
predefined symbols
  backward compatibility 119
  overview 5
  summary 119
  _ _CODE_MODEL_ _ 120
  _ _CORE_ _ 120
  _ _DATA_MODEL_ _ 120
  _ _DATE_ _ 120
  _ _FILE_ _ 120
  _ _IAR_SYSTEMS_ICC_ _ 121
  _ _ICCSAM8_ _ 121
  _ _LINE_ _ 121
  _ _LITTLE_ENDIAN_ _ 121
  _ _STDC_ _ 121
  _ _STDC_VERSION_ _ 121
  _ _TID_ _ 121
  _ _TIME_ _ 121
  _ _VER_ _ 121
--preprocess (compiler option) 93
preprocessing directives
  implementation-defined behavior 144
preprocessor output 93
preprocessor symbols 51
  defining 80
preprocessor, extending 89
prerequisites (programming experience) xiii
processor operations, low-level 6, 123
programming experience, required xiii

programming hints 55
ptrdiff_t (integer type). 53, 63, 128
PUBLIC (assembler directive) 94
--public_equ (compiler option) 94

Q

QCCSAM8 (environment variable). 77
qualifiers, implementation-defined behavior. 144

R

-r (compiler option). 81, 94
RAM memory, saving 56
reading guidelines. xiii
reading, recommended xv
realloc (standard library function). 14
recursive functions 56
    storing data on stack 12–13
reference information, typographic convention. xvi
register parameters 28
registered trademarks ii
registers
    assigning to parameters 28
    callee-save, stored on stack 12
    implementation-defined behavior. 143
    permanent 29
    scratch 29
remark (diagnostic message). 131
    classifying 83
    enabling 94
--remarks (compiler option) 94
REQUIRE (assembler directive). 49
required (#pragma directive). 115
RESET (segment) 45, 70
return values, from functions 29
_ _root (extended keyword) 105
    using in #pragma directives 113
routines, time-critical 6, 123
RSEG (assembler directive) 49
RTMODEL (assembler directive) 26
rtmodel (#pragma directive) 115
_ _rt_version (runtime model attribute). 27
runtime environment. 47
runtime libraries 8
    introduction 127
    naming convention. 8
    summary 9
runtime model attributes 25
    _ _code_model. 27
    _ _data_model 27
    _ _rt_version 27

S

-s (compiler option) 95
SAM8
    architecture xiii
    code execution 4
    instruction set. xiii
    memory access. 4
SAM8 IAR C Compiler
    migrating from V1.x to V2.x 89, 135
scratch registers 29
segment parts, unused 49
segment types, in XLINK 38
segment (#pragma directive). 115
segments 65
    introduction 37
    placeholder 38
    summary 65
    BANK0_A 66
    BANK1_A 67
    CALLT_C 67
    CODE 45, 67
    CONST 67
    CSTACK 68
        example. 43

CSTACKN . . . 68
example. . . . 43
CSTACK2 . . . 68
example. . . . 43
INTVEC . . . 45, 68
NEAR_C . . . 69
NEAR_I. . . . 69
NEAR_ID . . . 69
NEAR_N . . . 69
NEAR_Z . . . 70
RESET. . . . 45, 70
TINYPi_N . . . 73
TINYP_N . . . 72
TINY_I . . . 70
TINY_ID . . . 70
TINY_N. . . . 71
TINY_Z. . . . 71
TINY2Pi_N . . . 73
TINY2P_N . . . 73
TINY2_I . . . 71
TINY2_ID . . . 71
TINY2_N. . . . 72
TINY2_Z. . . . 72
_ _segment_begin (intrinsic function). . . . 124
_ _segment_end (intrinsic function) . . . 125
semaphores, operations on . . . 103
setjmp.h (library header file). . . . 52, 128
_ _set_interrupt_state (intrinsic function) . . . 125
severity level, of diagnostic messages. . . . 131
specifying . . . 132
SFR (special function registers) . . . 18
short register mode . . . 79
short (data type) . . . 61
signed char (data type) . . . 61–62
specifying . . . 79
--silent (compiler option) . . . 95
silent operation, specifying . . . 95
size optimization, specifying. . . . 96
size_t (integer type) . . . 53, 63, 128
skeleton code, creating for assembler language interface . . 31
small (code model) . . . 7, 21
small (data model) . . . 8
source files
list all referred . . . 87
special function registers (SFR) . . . 18
special function types . . . 21
overview . . . 5
speed optimization, specifying . . . 95
stack . . . 11
advantages and problems using . . . 12
contents of . . . 12
function usage . . . 11
internal data . . . 68
saving space. . . . 56
size. . . . 44
stack parameters . . . 28
stack pointer . . . 12
standard error . . . 93
standard output, specifying . . . 93
startup code. . . . 45
See also RESET
startup, system . . . 47
statements, implementation-defined behavior. . . . 144
static memory . . . 11, 13
stdarg.h (library header file) . . . 52, 128
_ _STDC_ _ (predefined symbol). . . . 121
_ _STDC_VERSION_ _ (predefined symbol) . . . 121
stddef.h (library header file) . . . 53, 128
stderr. . . . 93
stdio.h (library header file) . . . 52, 128
stdlib.h (library header file). . . . 52, 128
stdout . . . 93
_ _stop (intrinsic function) . . . 125
--strict_ansi (compiler option). . . . 95
string.h (library header file) . . . 52, 128
structure types
alignment. . . . 64
layout. . . . 64

structures
anonymous . . . 18
implementation-defined behavior . . . 143
placing in memory type . . . 16
Support, Technical . . . 132
symbols
predefined
overview of . . . 5
preprocessor, defining . . . 80
syntax, extended keywords . . . 101
system startup . . . 47
system termination . . . 48

T

Technical Support, IAR . . . 132
Terminal I/O window, in C-SPY . . . 53
termination, system . . . 48
terminology . . . xiii
32-bit (floating-point format) . . . 62
_ _TID_ _ (predefined symbol) . . . 121
_ _TIME_ _ (predefined symbol) . . . 121
time-critical routines . . . 6, 123
_ _tiny (extended keyword) . . . 105
_ _tinyp (extended keyword) . . . 106
TINYPi_N (segment) . . . 73
_ _tinypn (extended keyword) . . . 106
TINYP_N (segment) . . . 72
_ _tiny_func (extended keyword) . . . 106
TINY_I (segment) . . . 70
TINY_ID (segment) . . . 70
TINY_N (segment) . . . 71
TINY_Z (segment) . . . 71
_ _tiny2 (extended keyword) . . . 106
_ _tiny2p (extended keyword) . . . 106
TINY2Pi_N (segment) . . . 73
_ _tiny2pn (extended keyword) . . . 107
TINY2P_N (segment) . . . 73
TINY2_I (segment) . . . 71
TINY2_ID (segment) . . . 71
TINY2_N (segment) . . . 72
TINY2_Z (segment) . . . 72
tips, programming . . . 55
trademarks . . . ii
translation, implementation-defined behavior . . . 139
trap vectors, specifying with #pragma directive . . . 117
type information, omitting . . . 93
type_attribute (#pragma directive) . . . 116
typographic conventions . . . xvi

U

uintptr_t (integer type) . . . 64
unions
anonymous . . . 18
implementation-defined behavior . . . 143
unsigned char (data type) . . . 61–62
changing to signed char . . . 79
unsigned int (data type) . . . 61
unsigned long (data type) . . . 61
unsigned short (data type) . . . 61

V

variable type information, omitting in object output . . . 93
variables
auto . . . 11–12, 56
defined inside a function . . . 11
global
placement in memory . . . 13
local. *See* auto variables
non-initialized . . . 16
omitting type info . . . 93
placing at absolute addresses . . . 17
placing in named segments . . . 17
placing in segments . . . 17
static, placement in memory . . . 13
vector (#pragma directive) . . . 22, 117

_ _VER_ _ (predefined symbol) 121
version, of compiler 121

W

WAIT (assembler instruction) 125
_ _wait_for_interrupt (intrinsic function) 125
warnings 131
classifying 84
disabling 92
exit code. 96
--warnings_affect_exit_code (compiler option) 77
--warnings_are_errors (compiler option) 96

X

XLINK segment types 38

Z

-z (compiler option) 96

Symbols

#include files, specifying 87
#pragma directives 5
bitfields 62, 110
constseg 110
dataseg 111
diag_default 111
diag_error 111
diag_remark 112
diag_suppress. 112
diag_warning 112
inline 112
language. 112
location 17, 23, 113
message 113
object_attribute 16, 113
optimize 114
overriding default behaviors. 8
required 115
rtmodel. 115
segment 115
summary 109
syntax. 110
type_attribute 116
vector 22, 117
-D (compiler option) 80
-e (compiler option) 85
-f (compiler option). 86
-I (compiler option). 87
-l (compiler option). 32, 88
-o (compiler option) 92
-r (compiler option). 81, 94
-s (compiler option) 95
-z (compiler option) 96
--char_is_signed (compiler option) 79
--code_model (compiler option) 79
--core (compiler option) 80
--data_model (compiler option) 81
--debug (compiler option) 81, 94
--dependencies (compiler option) 82
--diagnostics_tables (compiler option) 84
--diag_error (compiler option) 83
--diag_remark (compiler option). 83
--diag_suppress (compiler option) 84
--diag_warning (compiler option) 84
--enable_eeprom_support (compiler option) 85
--enable_multibytes (compiler option) 85
--generate_tinyfunc_runtime_library_calls
(compiler option) 86
--header_context (compiler option). 87
--library_module (compiler option) 89
--migration_preprocessor_extensions (compiler option) . . . 89
--module_name (compiler option) 89
--no_code_motion (compiler option) 90
--no_cse (compiler option) 90
--no_inline (compiler option) 91

--no_unroll (compiler option) 91
--no_warnings (compiler option) 92
--no_wrap_diagnostics (compiler option) 92
--omit_types (compiler option) 93
--only_stdout (compiler option) 93
--place_constants_in_rom (compiler option). 93
--preprocess (compiler option) 93
--remarks (compiler option) 94
--silent (compiler option) 95
--strict_ansi (compiler option). 95
--warnings_affect_exit_code (compiler option) 77, 96
--warnings_are_errors (compiler option) 96
?C_EXIT (assembler label). 53
?C_GETCHAR (assembler label) 53
?C_PUTCHAR (assembler label) 53
@ (operator) 17, 23
__bankn (extended keyword) 102
__code (extended keyword) 102
__code_model (runtime model attribute) 27
__CODE_MODEL__ (predefined symbol). 120
__CORE__ (predefined symbol). 120
__data_model (runtime model attribute) 27
__DATA_MODEL__ (predefined symbol) 120
__DATE__ (predefined symbol) 120
__disable_interrupt (intrinsic function) 124
__enable_interrupt (intrinsic function). 124
__fast (extended keyword) 102
__FILE__ (predefined symbol). 120
__generic (extended keyword) 102
__get_interrupt_state (intrinsic function) 124
__IAR_SYSTEMS_ICC__ (predefined symbol) 121
__ICCSAM8__ (predefined symbol) 121
__idle (intrinsic function) 124
__interrupt (extended keyword) 22, 102
  using in #pragma directives 116–117
__intrinsic (extended keyword) 103
__LINE__ (predefined symbol) 121
__LITTLE_ENDIAN__ (predefined symbol) 121
__low_level_init, customizing 48
__monitor (extended keyword) 103
  using in #pragma directives 116
__near (extended keyword) 104
__no_init (extended keyword) 16, 105
  using in #pragma directives 113
__no_operation (intrinsic function) 124
__root (extended keyword) 105
  using in #pragma directives 113
__rt_version (runtime model attribute). 27
__segment_begin (intrinsic function). 124
__segment_end (intrinsic function) 125
__set_interrupt_state (intrinsic function) 125
__STDC__ (predefined symbol) 121
__STDC_VERSION__ (predefined symbol) 121
__stop (intrinsic function) 125
__TID__ (predefined symbol). 121
__TIME__ (predefined symbol) 121
__tiny (extended keyword). 105
__tinyp (extended keyword). 106
__tinypn (extended keyword). 106
__tiny_func (extended keyword) 106
__tiny2 (extended keyword). 106
__tiny2p (extended keyword). 106
__tiny2pn (extended keyword). 107
__VER__ (predefined symbol) 121
__wait_for_interrupt (intrinsic function) 125
_Pragma (preprocessor operator) 152

Numerics

32-bit (floating-point format) 62