AVR® IAR COMPILER

Reference Guide

for Atmel® Corporation's

AVR® Microcontroller

COPYRIGHT NOTICE

DISCLAIMER

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

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

TRADEMARKS

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

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

First edition: March 2000

Part number: CAVR-1

WELCOME

Welcome to the AVR IAR Compiler Reference Guide.

This guide provides reference information about the IAR Systems C and Embedded C++ Compiler for Atmel's AVR microcontroller.

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

For information about programming with the AVR IAR Assembler, refer to the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

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

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

ABOUT THIS GUIDE

This guide consists of the following parts:

- ***Part 1: Using the AVR IAR compiler***

  What's new in this product summarizes the new product features.

  Efficient coding techniques provides programming hints and information about how to fine-tune your application and make it benefit from the features of the AVR IAR Compiler's features.

  Configuration describes how to configure the compiler to suit the requirements of your application.

  Assembly language interface describes the interface between C or Embedded C++ programs and assembly language routines.

- ***Part 2: Compiler reference***

  Data representation describes the available data types, pointers, and structure types.

  Segments 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 AVR-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 intrinsic functions.

  Library functions gives an introduction to the C or Embedded C++ library functions, and summarizes the header files.

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

- ***Part 3: Migration and portability***

  Migrating to the AVR IAR Compiler contains information that can be useful when migrating from an existing IAR product to the new AVR IAR 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.

ASSUMPTIONS AND CONVENTIONS

ASSUMPTIONS

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

- The architecture and instruction set of Atmel's AVR microcontroller.
- The C or Embedded C++ programming language.
- Windows 95/98 or Windows NT, depending on your host system.

CONVENTIONS

This guide uses the following typographical conventions:

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

CONTENTS

PART 1: USING THE AVR IAR COMPILER

This part of the *AVR IAR Compiler Reference Guide* includes the following chapters:

- *What's new in this product*
- *Efficient coding techniques*
- *Configuration*
- *Assembly language interface*.

WHAT'S NEW IN THIS PRODUCT

The AVR IAR Compiler supports C and Embedded C++ for the Atmel AVR microcontrollers with RAM, including the Classic and Mega families.

It is based on the latest IAR compiler technology and replaces the A90 IAR C Compiler, ICCA90.

This chapter summarizes the new key features in the AVR IAR Compiler. For details about the differences between this product and the A90 IAR C Compiler, ICCA90, see the chapter *Migrating to the AVR IAR Compiler*.

EMBEDDED C++

Embedded C++, or EC++, can be seen as a subset to C++ designed to suit the development of embedded applications. The C++ features not found in EC++ are those that were considered to be costly for some reason, for example due to the risk of increasing code size or increased execution speed overhead.

The resulting language, EC++, is well suited for modern modeling and development techniques. Compared to C—the most widely used language in the embedded field—EC++ adds a number of new language features.

Obviously, most important is the support for object-oriented programming (OOP), a technique used for modularizing an application which makes it easier to structure and maintain larger applications. OOP also increases the reusability of code, and in the same time reduces the time to test new applications inherited from existing ones, both factors working to shorten the time to market for new products.

The AVR IAR Compiler conforms to the free-standing implementation of the ISO/ANSI C standard and Embedded C++ specifications. All required data types are supported. Notice, however, that by default the type `double` is implemented as `float`.

CODE AND DATA STORAGE

The AVR microcontroller has three separate address spaces: one space each for code, data, and the inbuilt EEPROM. The AVR IAR Compiler provides various memory models, pointer types, and language extensions for efficient use of memory.

INBUILT EEPROM

The AVR IAR Compiler introduces a new keyword, `__eeprom`, that allows the programmer to place initialized and non-initialized variables in the inbuilt EEPROM of the AVR microcontroller. These variables can be used as any other variable and provides a convenient way to access the inbuilt EEPROM.

It is also possible to override the supplied support functions to make the `__eeprom` keyword access an EEPROM or flash memory that is placed externally but not on the normal data bus, for example on an I2C bus.

OPTIMIZATION TECHNIQUES

To generate efficient and compact code for the AVR architecture, the AVR IAR Compiler has an inbuilt data-flow analyzer and a global optimizer. The compiler also uses many optimization techniques, for example dead-code elimination, jump optimizations, and loop optimizations.

The basic approach is to optimize for either size or speed, and to enable or disable the optimizations during the different phases of your application development project.

For additional information, see *Optimizations*, page 36.

EFFICIENT CODING TECHNIQUES

In this chapter you will find hints on how to write programs that make efficient use of the AVR IAR Compiler.

The chapter *Configuration* contains information about how to use and modify the IAR toolkit to satisfy your application requirements.

If you are porting code or migrating from a previous IAR Systems product such as the AT90S IAR C Compiler, see the chapter *Migrating to the AVR IAR Compiler* in *Part 3: Migration and portability* of this guide, for additional information.

PROGRAMMING HINTS

This section contains recommendations on how to write efficient code for the AVR microcontroller using the AVR IAR Compiler.

OPTIMIZING

- To achieve minimum code size, you would normally use the size optimization. Notice, however, that in certain cases a high level of speed optimization can generate smaller code than the size optimization would. See *Optimizations*, page 36, for additional information.
- Sensible use of the memory attributes (see the chapter *Extended keywords*) can enhance both speed and code size in critical applications.
- Small local functions may be inlined by the compiler if declared `static`, which allows further optimizations. This feature can be turned off with the `--no_inline` option. See page 98 for information about this option.
- Global scalar variables that are not used outside their module should be declared as `static`.
- Avoid using inline assembler. Instead, try writing the code in C or Embedded C++, use intrinsic functions, or write a separate assembler module.

SAVING STACK SPACE AND RAM MEMORY

- Avoid long call chains and recursive functions in order to save stack space.
- Declare variables with a *long* life span as global in order to save stack space.
- Declare variables with a *short* life span as local variables. When the life spans for these variables end, they will be popped from the stack and the previously occupied memory can than 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; in order to save RAM memory, you should instead pass them as pointers. Small parameters can be passed in registers; see *C calling convention*, page 37, for additional information.

USING EFFICIENT DATA TYPES

- Use ISO/ANSI prototypes since they allow the compiler to generate efficient code and provide type checking of function parameters.
- Use small and unsigned data types (`unsigned char` and `unsigned short`) unless your application really requires a greater precision.
- Try to avoid 64-bit data types, such as `double` and `long long`.
- Pointers to the `near` memory are smaller than the default pointer in the large memory model. Pointers to the `tiny` memory are smaller than the default pointer in the small memory model.
- Bitfields with sizes other than 1 bit should be avoided since they will result in inefficient code compared to bit operations.

MIGRATING FROM ICCA90

If you have code written for the A90 IAR Compiler, ICCA90, use the file `comp_a90.h` for the migration. This file, which is provided with the product, contains macros that facilitate the migration. You should also modify the code by replacing, for example, keywords and intrinsics. More information about the differences between the A90 IAR Compiler and the AVR IAR Compiler is provided in the chapter *Migrating to the AVR IAR Compiler*.

EMBEDDED C++ OVERVIEW

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

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

- Classes, which are user-defined types that incorporate both data structure and behavior. The essential feature of inheritance allows data structure and behavior to be shared among classes.
- Polymorphism, which means that an operation can behave differently on different classes, is provided by virtual functions.
- Overloading of operators and function names, which allows several operators or functions with the same name, provided that there is a sufficient difference in their argument lists.
- Type-safe memory management using operators `new` and `delete`.
- Inline functions, which are indicated as particularly suitable for inline expansion.

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

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

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

The excluded language features also make the run-time library significantly more efficient. The Embedded C++ library furthermore differs from the full C++ library in that:

- The Standard Template Library (STL) is excluded.
- Streams, strings, and complex numbers are supported, without using templates.
- Library features, which relate to exception handling and run-time type information (headers `<except>`, `<stdexcept>` and `<typeinfo>`) are excluded.

LANGUAGE EXTENSIONS OVERVIEW

This section briefly describes the extensions provided in the AVR IAR Compiler to support specific features of the AVR microcontroller.

Note: For a summary of the IAR extensions to the ISO standard for the C programming language, see the chapter *IAR C extensions* in this guide.

EXTENDED KEYWORDS

By default the address range in which the compiler places data and functions is determined by which memory model you select for your application. In order to achieve maximum efficiency in your application, you may want to override these and other defaults by using the extended keywords which provide the following facilities:

- Keywords such as `__tiny`, `__near`, `__far`, and `__huge` allow you to override the default storage of data objects.
- The keywords `__farfunc` and `__nearfunc` allow you to define the memory range where a function will be located.
- The `__no_init` keyword prevents initialization of variables. Use it to reduce the amount of initialization code that is generated or for data that will be placed in non-volatile RAM.
- The keyword `__C_task` allows you to save stack size and reduce code size. It is typically used for `main`.
- Function modifiers allow you to override the default mechanism by which the compiler calls a function, for example `__interrupt` and `__monitor`.

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

The command line option `-e` make the extended keywords available, and reserves them so that they cannot be used as variable names; see page 89 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 outputs warning messages.

The `#pragma` directives are always enabled in the AVR IAR 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 time and date of compilation.

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` intrinsic function assembles and inserts the supplied assembler statement in-line. The statement can include instruction mnemonics, register mnemonics, constants, and/or a reference to a global variable. For example:

```
asm("LDI R16,0\n LDI R17,0\n RET");
```

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

CONFIGURATION

The IAR toolkit for the AVR microcontroller contains a number of components that you can modify according to the requirements of your application. This chapter provides information about the configuration:

- The options used for specifying the AVR derivative and memory model for your project.
- The linker command file which is used for specifying how the application will fit on your selected chip. This file contains information about the segments and their address ranges, the stack size, and the heap size.
- Run-time libraries
- Initialization procedure
- I/O operations
- Module consistency
- Optimization.

Note: Some of the configuration procedures involve customizing the standard files. We strongly recommend that you never alter the original files. Instead, make copies of the originals and store the copies in a project directory, together with your project source code files. There you can edit the files to suit your application requirements.

For information about how to configure the hardware or peripheral devices, refer to the hardware documentation.

PROJECT OPTIONS

This section describes the project options that are used for configuring the IAR development tools according to your AVR derivative and your application requirements.

PROCESSOR

The AVR IAR Compiler supports many of the AVR microcontroller derivatives. The processor option reflects the addressing capability of the target processor. When you select a particular processor option for your project, several target-specific parameters are tuned to best suit that derivative.

Use either the `--cpu` or `-v` option to specify the AVR derivative; see the chapter *Compiler options* for syntax information.

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

Mapping of processor options and AVR derivatives

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

Processor option	*Alternative option*	*Supported* AVR *derivative*
`--cpu=2313`	`-v0`	AT90S2313
`--cpu=2323`	`-v0`	AT90S2323
`--cpu=2333`	`-v0`	AT90S2333
`--cpu=2343`	`-v0`	AT90S2343
`--cpu=4414`	`-v1`	AT90S4414
`--cpu=4433`	`-v0`	AT90S4433
`--cpu=4434`	`-v1`	AT90S4434
`--cpu=8515`	`-v1`	AT90S8515
`--cpu=8534`	`-v1`	AT90S8534
`--cpu=8535`	`-v1`	AT90S8535
`--cpu=m103`	`-v3`	AT90mega103
`--cpu=m161`	`-v3`	AT90mega161
`--cpu=m603`	`-v3`	AT90mega603
`--cpu=tiny22`	`-v0`	AT90tiny22

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

Implicit assumptions when using -v

Notice that the `--cpu` option is more precise since it has more information about the intended target than the generic `-v` option. The `--cpu` option, for example, knows how much flash memory is available in the given target and allows the compiler to optimize accesses to code memory in a way that the `-v` option does not.

It is also important to remember that all implicit assumptions in a given `-v` options are also true for the `--cpu` option.

`-v0` and `-v2`: When compiling code for the `-v0` or `-v2` processor option, the compiler assumes that the index registers `X`, `Y`, and `Z` are eight bit wide when accessing the inbuilt SRAM. It also assumes that it is not possible to attach any external memory to the microcontroller and that it therefore should not generate any `_C` segment. Instead the compiler adds an implicit `-y` command line option. It will also try to place all aggregate initializers in flash memory, i.e. an implicit `--initializers_in_flash` is also added to the command line.

`-v0` and `-v1`: When compiling code for the `-v0` or `-v1` processor options, the compiler assumes that `RJMP` and `RCALL` can reach the entire code space. It also assumes that the interrupt vectors are two bytes each.

`-v1`, `-v3`, `-v4`, `-v5`, and `-v6`: The compiler assumes that it is possible to have external memory and will therefore generate `_C` segments.

`-v2`, `-v4`, `-v5`, and `-v6`: There are currently no derivatives that match these processor options which have been added to support future derivatives.

The following table summarizes the code characteristics:

Processor option	*Default function memory attribute*	*Max module and/or program size*	*Comment*
`-v0`, `-v1`	`__nearfunc`	≤ 8 Kbytes	The code memory space is physically limited to 8 Kbytes. Interrupt vectors are 2 bytes long.

Processor option	*Default function memory attribute*	*Max module and/or program size*	*Comment*
`-v2`, `-v3`, `-v4`	`__nearfunc`	≤ 128 Kbytes	Two bytes are used for all function pointers. Interrupt vectors are 4 bytes long.
`-v5`, `-v6`	`__farfunc`	≤ 8 Mbytes	Three bytes can be used for function pointers. Interrupt vectors are 4 bytes long.

MEMORY MODEL

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

- Non-stacked variables, i.e. global data and variables declared as static.
- Stacked data, for example locally declared data.
- Dynamically allocated data, for example data allocated with `malloc` and `calloc`.

You choice of processor option determines which memory models are available. The following table summarizes the characteristics of the different memory models:

Memory model	*Default memory attribute*	*Default data pointer*	*Max. stack size*	*Processor option*
`tiny`	`__tiny`	`__tiny`	≤ 256 bytes	`-v0`, `-v1`, `-v2`, `-v3`, `-v5`
`small`	`__near`	`__near`	≤ 64 Kbytes	`-v1`, `-v3`, `-v4`, `-v5`, `-v6`
`large`	`__far`	`__far`	≤ 64 Kbytes	`-v4`, `-v6`
`generic`	`__tiny`	`__generic`	≤ 256 bytes	`-v0`, `-v2`
	`__near`		≤ 64 Kbytes	`-v1`, `-v3`, `-v5`
	`__far`		≤ 64 Kbytes	`-v4`, `-v6`

For information about the mapping of processor option and AVR derivative, see the table on page 12.

Your program may use only one memory model at a time, and the same model must be used by all user modules and all library modules.

If you do not specify a memory model option, the compiler will use the `tiny` memory model for all processor options except `-v4` and `-v6` where the `small` memory model will be used.

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

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

Use the `--memory_model` option to specify the memory model for your project; see page 95 for syntax information.

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

MEMORY LOCATION

Code and different types of data (e.g. volatility and address range) are located in the memory areas as follows:

- The internal FLASH areas are used for code, constants, and initial values.
- The external ROM areas are used for constants.
- The RAM areas are used for the stack and for variables.

A number of named segments are available to locate code and the different types of data in the most efficient part of the memory. The compiler automatically selects a segment depending on your choice of memory model and processor option. If you use `#pragma` directives and extended keywords to override the default memory attributes, this also affects the location.

The available segments are described in the chapter *Segments* in *Part 2: Compiler reference* in this guide.

Use the segment control directives in the AVR IAR Assembler to control the segments; these directives are described in the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

The AVR IAR Compiler uses the following memory types:

- `CODE` is used for code, constants and initial values.
- `DATA` is used for stacks and variables.
- `CONST` is used for constants.
- `EEPROM` is used for variables declared with the `__eeprom` keyword. These variables are located to the XLINK segment type `XDATA`.

It is important to understand the fact that the `CONST` segments are assumed to be placed in an external PROM. Targets that do not have the possibility to add external memory should not place variables and constants in the `CONST` area. Use the `-y` command line option to redirect all objects to the initialized DATA area instead. For additional information, see *-y*, page 107.

It is also important to notice that `DATA` and `CONST` share address spaces, i.e. they are both data space areas, whereas the `CODE` area resides in code space and the `EEPROM` area resides in the inbuilt EEPROM.

Non-initialized memory
The compiler supports the declaration of variables that are not to be initialized at startup through the `__no_init` type modifier and the `#pragma object_attribute` directive. The compiler places such variables in separate segments, depending on which memory keyword is used. These segments should be assigned to, for example, the address range of the non-volatile RAM of the hardware environment.

Notice that `__no_init` can also be used for other types of variables that need not be initialized, for example input buffers. The run-time system does not initialize variables located in these segments.

To assign the `__no_init` segment to the address of the non-volatile RAM, you need to modify the linker command file.

For information about the `__no_init` keyword, see *__no_init*, page 115. For information about the `#pragma object_attribute`, see *Object attribute*, page 128.

LINKER COMMAND FILE

The linker command file is an extended command line file, required for the generation of object code. The IAR XLINK Linker uses the file to select the program modules to combine and to specify the location of the generated code and data, thereby assuring that your application fits on the selected chip.

Since the chip-specific details are specified in the linker command file and not in the source code, the linker command file also ensures code portability. Basically, you can use the same source code with different derivatives just by recompiling the code with different processor and memory model options, and by specifying the appropriate linker command file.

In particular, the linker command file specifies:

- The placement of segments
- The stack size
- The heap size.

This section first discusses the stack size and the heap size, and then describes how to modify the contents of a linker command file.

For further information about the IAR XLINK Linker and the XLINK options, see the XLINK section in the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

STACK SIZE

The compiler uses the internal data stack, `CSTACK`, for a variety of user program operations, and the required stack size depends heavily on the details of these operations. If the given stack size is too 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.

Notice that there is also an internal return stack, `RSTACK`. See *Cstack*, page 64, and *Rstack*, page 75, for additional information about the segments used for the data stack and return stack, respectively.

Estimating the required data stack size
The stack is used for storing:

- Local variables and parameters not passed in registers.
- Temporary results in expressions.

- Function return addresses (RSTACK).
- Temporary values in run-time library routines (CSTACK/RSTACK).
- Processor state during interrupts.

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

Notice that there is a C-SPY macro, StackChk.mac, that helps you calculate the required stack size. To use this macro, follow the instructions in the macro file.

To change the data stack size, edit the linker command file and replace the default size by the value of your choice. The procedure is described in the section *Customizing the linker command file* in this chapter.

HEAP SIZE

If the library functions malloc or calloc are used in the program, they allocate memory from a heap of memory in the HEAP segment.

To change the heap size, edit the linker command file and replace the default size with the value of your choice. See *Defining the malloc HEAP segment*, page 22.

CUSTOMIZING THE LINKER COMMAND FILE

The only change you will normally have to make to the supplied linker command file is to suit the details of the target hardware memory map. For details of individual segments, see the chapter *Segments*.

The avr\config directory contains ready-made linker command files. These files contain the information required by the linker and are ready to be used. If, for example, your application uses external RAM, you will only need to provide details about the external RAM memory area.

The name of the linker command file indicates the supported derivative; for example, lnk2313.xcl supports the AT90S2313 derivative.

If a ready-made linker command file is not availabile for your derivative, select the linker command file of a similar derivative in the avr\config directory, and create a copy that suits your derivative. To create a linker command file for a particular project, you could also use one of the templates located in the avr\src\template directory.

The template filename indicates the supported processor option and memory model; for example, `lnk0t.xcl` supports the `-v0` processor option with the tiny memory model, and `lnk3s.xcl` supports the `-v3` processor option with the small memory model.

Notice that each of the supplied linker command files includes comments explaining the entire contents.

The following section explains the contents of a linker command file. The example is based on the `lnk1s.xcl` file, which is the linker command file template for the `-v1` processor option and the small memory model. The file will be modified to suit the AT90S8515 derivative, where the flash memory area is located within the `0x0-0x1FFF` address range, and the RAM memory area is located within the `0x60-0x25F` address range.

Defining the stack and heap segments
In the first section of the linker command file, we use the XLINK option `-D` to specify the size of the segments used for the data stack (`CSTACK`), the return stack (`RSTACK`), and the heap (`HEAP`), respectively:

Modify the template to suit the requirements of your application, for example:

```
-D_CSTACK_SIZE=40   /* 64 bytes for auto variables and
                    register save. */
-D_RSTACK_SIZE=10   /* 16 bytes for return addresses,
                    equivalent to 8 levels of calls, */
                    /* including interrupts. */
-D_HEAP_SIZE=10     /* 16 bytes of heap. */
```

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

```
-cavr
```

We will then define segments in flash memory.

Defining the reset and interrupt vectors
We will use the XLINK option `-Z` to define each segment in the program address space, the internal flash memory.

The AT90S8515 derivative has 13 interrupt vectors and one reset vector. We therefore specify 14 interrupt vectors, each of two bytes.

The reset vector and interrupt vectors must be placed at address 0 and forwards:

```
-Z(CODE)INTVEC=0-1B /* 14 Interrupt vectors * 2 bytes
                    each */
```

Defining objects declared __tinyflash
We will now define the segment for constant objects that have been declared with the `__tinyflash` keyword:

```
-Z(CODE)TINY_F=0-FF
```

Defining compiler-generated segments
Next we will define the segments that are generated by the compiler. These are used for storing information that is vital to the operation of the program.

- The `SWITCH` segment which contains data statements used in the switch library routines. These tables are encoded in such a way as to use as little space as possible.
- The `INITTAB` segment contains the segment initialization description blocks that are used by the `__segment_init` function which is called by `CSTARTUP`. This table consist of a number of `SegmentInitBlock_Type` objects. This type is declared in the `segment_init.h` file which is located in the `avr\src\lib` directory.
- The `DIFUNCT` segment is only used when a source file has been compiled in EC++ mode and the file contains global objects (class instances). The segment will then contain a number of function pointers that point to constructor code that should be performed for each object.

```
-Z(CODE)SWITCH,INITTAB,DIFUNCT=0-1FFF
```

Defining objects declared __flash
Constant objects that have been declared with the `__flash` keyword will be placed in this segment:

```
-Z(CODE)NEAR_F=0-1FFF
```

Defining functions declared __nearfunc
Code that comes from functions declared `__nearfunc` is placed in the `CODE` segment. This segment must be placed in the first 128 Kbytes of the flash memory. The reason for this is that function pointers to `__nearfunc` functions are 2 bytes large. All function pointers are word pointers.

```
-Z(CODE)CODE=0-1FFF
```

Defining initialization segments
We will then define the `TINY_ID` and `NEAR_ID` segments.

If any non-zero initialized variables are present in the application the corresponding *`segment`*`_ID` segment will contain the initial values of those variables. The `CSTARTUP` module will initialize the *`segment`*`_I` segments at system start-up by calling the `__segment_init` function.

```
-Z(CODE)TINY_ID,NEAR_ID=0-1FFF
```

This completes the definition of segments in code memory and we will now define data memory segments.

Defining objects declared __tiny
The `TINY_I`, `TINY_Z`, and `TINY_N` segments contain objects that have been declared with the `__tiny` keyword. After `CSTARTUP` has run, *`segment`*`_I` has been initialized with the values in *`segment`*`_ID` and *`segment`*`_Z` has been cleared so that it contains all zeros.

```
-Z(DATA)TINY_I,TINY_Z,TINY_N=60-FF
```

Defining objects declared __near
We will now define the segments that contain objects that have been declared with the `__near` keyword. After `CSTARTUP` has run, *`segment`*`_I` has been initialized with the values in *`segment`*`_ID` and *`segment`*`_Z` has been cleared so that it contains all zeros.

```
-Z(DATA)NEAR_I,NEAR_Z=60-25F
```

Defining the data stack
The data stack, `CSTACK`, is used for auto variables, function parameters, and temporary storage. It is therefore important that the data stack is large enough. However, a too large stack will waste valuable RAM space.

To determine approximately how much data stack an application requires, simply add the stack information given at the end of each compiler list file in your project.

The given value does not include stack used by interrupts and assembler functions written by the user. It is therefore necessary to add a small safety margin to the value given in the list files.

```
-Z(DATA)CSTACK+_CSTACK_SIZE=60-25F
```

Notice that the segment size was defined earlier in the linker command file, see *Defining the stack and heap segments*, page 19.

If external SRAM is available it is possible to place the stack there. However, the external memory is slower than the internal so moving the stacks to external memory will decrease the system performance.

Defining the malloc HEAP segment

If the application uses the library functions `malloc` or `calloc`, the memory is allocated from the `HEAP` segment. It is therefore important that the segment is placed so that a default pointer can point to it. Since this linker command file assumes that the small memory model is used, the heap must be placed in the range `0x0–0xFFFF`.

The library technology is such that if no calls are made to `malloc`, `calloc`, or `realloc`, this segment will not be included. It is therefore safe to include it in all your linker command files.

```
-Z(DATA)HEAP+_HEAP_SIZE=60-25F
```

Notice that the segment size was defined earlier in the linker command file, see *Defining the stack and heap segments*, page 19.

Defining the return address stack

The return address stack is used for storing the return address when a `CALL`, `RCALL`, `ICALL`, or `EICALL` instruction is executed. Each call will use three bytes of return address stack. An interrupt will also place a return address on this stack.

To determine the size of the return address stack, use the same technique as when determining the size of the data stack; see page 21. Notice however that if the cross-call optimization has been used (`-z9` without `--no_cross_call`), the value can be off by as much as a factor of six depending on how many times the cross-call optimizer has been run (`--cross_call_passes`). Each cross-call pass adds one level of calls, for example, two cross-call passes would result in a tripled stack usage.

```
-Z(DATA)RSTACK+_RSTACK_SIZE=60-25F
```

The segment size was defined earlier in the linker command file, see *Defining the stack and heap segments*, page 19.

If external SRAM is available it is possible to place the stack there. However, the external memory is slower than the internal memory so moving the stacks to external memory will decrease the system performance.

Defining external memory
The linker command file must contain information about any external memory devices used. Notice that you must also turn on the external data and address buses in `__low_level_init` if external memory is used.

If your application uses external EPROM, include the following definition:

```
-Z(CONST)NEAR_C=_EXT_EPROM_BASE-(_EXT_EPROM_BASE+_EXT_EPR
OM_SIZE)
```

If your application uses external EEPROM, include the following definition:

```
-Z(DATA)NEAR_N=_EXT_EEPROM_BASE-(_EXT_EEPROM_BASE+_EXT_EE
PROM_SIZE)
```

Note: The `_EXT_EEPROM_BASE` and `_EXT_EEPROM_SIZE` variables are defined in the linker command file template, where they have the value 0. If you want to use these variables, you must provide values that suit the hardware.

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

First we select which `printf` and `sprintf` formatter to use. Here we use the smaller `printf` in order to reduce the library size:

```
-e_Printf_1=_Printf
```

Next we select which `scanf` and `sscanf` formatter to use. In order to reduce the library size, we disable the support for floating point, format modifiers, and field widths in `scanf`:

```
-e_Scanf_1=_Scanf
```

For further information about the input and output formatters, see *Input and output*, page 29.

Suppressing warnings
Then we suppress one XLINK warning that is not relevant for this processor:

```
-w29
```

This completes the contents of the linker command file.

Defining a linker command file
To specify a linker command file, use the XLINK option `-f`. For example, to use the AT90S2313 linker command file (`lnk2313.xcl`), enter the command:

```
xlink filename(s) -f lnk2313
```

In the IAR Embedded Workbench, the appropriate linker command file is selected automatically based on which processor configuration and memory model you have select for your project. You can use the XLINK options in the IAR Embedded Workbench to override the default choice. See the *AVR IAR Embedded Workbench™ User Guide* for additional information.

RUN-TIME LIBRARY

The run-time library includes all run-time functions and the program initialization module `CSTARTUP`; see *Initialization*, page 26, for additional information. The linker will include only those routines that are required—directly or indirectly—by your application.

In order to support many AVR microcontroller derivatives, a large number of run-time libraries are supplied. The library files are by default located in the `avr\lib` directory.

A very simple version of the `__low_level_init` function is also supplied in the run-time library. This function does nothing except to return 1 (one).

Use the `-C` XLINK option on the command line to specify which run-time library to use.

Use the **Include** options in the **XLINK** category of the project options to specify which run-time library to use. See the chapter *XLINK options* in the *AVR IAR Embedded Workbench™ User Guide* for additional information.

The following table shows the mapping of run-time libraries, processor options, and memory models:

Library file	*Processor option*	*Memory model*
dl10t.r90	-v0	tiny
dl10g.r90	-v0	generic
dl11t.r90	-v1	tiny
dl11s.r90	-v1	small
dl11g.r90	-v1	generic
dl12t.r90	-v2	tiny
dl12g.r90	-v2	generic
dl13t.r90	-v3	tiny
dl13s.r90	-v3	small
dl13g.r90	-v3	generic
dl14s.r90	-v4	small
dl14l.r90	-v4	large
dl14g.r90	-v4	generic
dl15t.r90	-v5	tiny
dl15s.r90	-v5	small
dl15g.r90	-v5	generic
dl16s.r90	-v6	small
dl16l.r90	-v6	large
dl16g.r90	-v6	generic

For information about the mapping of the `-v` processor option and AVR derivative, see *Mapping of processor options and AVR derivatives*, page 12.

The IAR XLIB Librarian can be used for maintaining and modifying libraries. For additional information, see the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

INITIALIZATION

This section describes the initialization of variables and I/O.

CSTARTUP

The assembly module CSTARTUP is a highly target-specific, vital part of the run-time model. On processor reset, execution passes to the run-time system routine CSTARTUP, which normally performs the following:

- Enables the external data and address buses if requested in `__low_level_init`.
- Initializes the stack pointers to the end of CSTACK and RSTACK, respectively.
- Calls `__low_level_init`; the return value states whether the next step should be performed.

  Note: If you use Embedded C++ and have either global objects, i.e. the constructor runs before `main`, or objects with initialized—zero or non-zero—static data members, you must run the segment initialization.
- Initializes C file-level and static variables except `__no_init` variables.
- Calls the start of the user program in `main()`.

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

Customizing CSTARTUP using the command line

Do not modify CSTARTUP unless required by your application. If you need to modify it, the overall procedure for creating a modified copy of CSTARTUP and adding it to your project is as follows:

1. Copy the assembly source file `cstartup.s90`, which is supplied in the `avr\src\lib\` directory, to your project directory.
2. Make any required modifications and save the file under the same name.
3. Assemble your copy of CSTARTUP using target options that match your selected compiler options. Also define a symbol for the appropriate memory model.

The assembler option `-v` corresponds to the compiler option `-v`. For information about how the compiler options `-v` and `--cpu` map, see *Mapping of processor options and AVR derivatives*, page 12.

For example, if you have compiled for the AT90S8515 processor variant (`--cpu=8515` or `-v1`) and the small memory model, assemble `cstartup.s90` with the command:

```
aavr cstartup -v1 -DMEMORY_MODEL=s
```

The assembler option `-D` defines the memory model symbol.

This will create an object module file named `cstartup.r90`.

4 Use the `-C` XLINK option on the command line to specify the library.

5 Link your code using the modified linker command file.

Customizing CSTARTUP using the IAR Embedded Workbench

1 Copy the assembly source file `cstartup.s90`, which is supplied in the `avr\src\lib\` directory, to your project directory.

2 Make any required modifications and save the file under the same name.

Define the assembler symbol `MEMORY_MODEL=s` using the preprocessor options in the assembler category in the IAR Embedded Workbench.

3 Assemble your copy of `cstartup.s90` using the same processor configuration and memory model as you have specified for your project.

This will create an object module file named `cstartup.r90`.

4 Add the customized `CSTARTUP` module to your project.

5 Select the option **Ignore CSTARTUP in library** on the **Include** page in the **XLINK** category of project options. See the chapter *XLINK options* in the *AVR IAR Embedded Workbench™ User Guide* for additional information.

6 Rebuild your project.

Maintaining library files
The IAR XLIB Librarian command `REPLACE-MODULES` allows you to permanently replace the original `CSTARTUP` with your customized version. See *Part 3: The IAR XLIB Librarian* in *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide* for detailed information.

VARIABLE AND I/O INITIALIZATION

In some applications you may want to initialize I/O registers, or omit the default initialization of data segments performed by `CSTARTUP`.

You can do this by providing a customized version of the routine `__low_level_init`, which is called from `CSTARTUP` before the data segments are initialized.

The value returned by `__low_level_init` determines whether data segments are initialized.

If the application has external memory this functions *must* be modified, so that it enables the external data bus. A skeleton for this function is supplied in the file `low_level_init.c` which is by default located in the `avr\src\lib` directory. It is vital that the external data bus has been enabled if `RSTACK` or `CSTACK` is placed in the external memory.

To enable the external data bus in `__low_level_init`, uncomment the line:

```
__require(__RSTACK_in_external_ram);
```

in the skeleton source file `low_level_init.c`.

This line will make sure that code which enables the external data bus with one wait state per access is added at the start of the `CSTARTUP` module.

Customizing __low_level_init from the command line
In most cases you can use the `__low_level_init` module provided with the product. If your application requires that you modify it, the overall procedure for creating a modified copy of `__low_level_init` is as follows:

1 Copy `low_level_init.c`, by default located in the `avr\src\lib` directory, to your project directory.

2 Make any required modifications, including the code necessary for the initializations. If you also want to disable the initialization of data segments, make the routine return 0. Save the file using the same filename.

3 Compile the customized version of `low_level_init.c` using the same processor option and memory model as for your project.

4 Link the file to the rest of your code.

Customizing __low_level_init in the IAR Embedded Workbench
In most cases you can use the `__low_level_init` module provided with the product. If your application requires that you modify it, the overall procedure for creating a modified copy of `__low_level_init` is as follows:

1 Copy `low_level_init.c`, by default located in the `avr\src\lib` directory, to your project directory.

2 Make any required modifications, including the code necessary for the initializations. If you also want to disable the initialization of data segments, make the routine return 0. Save the file using the same filename.

3 Add the customized version of `low_level_init.c` to your project.

4 Compile the customized routine using the same processor configuration and memory model as for the project.

5 Rebuild the project.

INPUT AND OUTPUT

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

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

I/O function	*File*	*Description*
`__open()`	`open.c`	Open a file.
`__close()`	`close.c`	Close a file.

I/O function	*File*	*Description*
`__read()`	`read.c`	Read a character buffer.
`__readchar()`	`readchar.c`	Read a character.
`__write()`	`write.c`	Write a character buffer.
`__writechar()`	`writechar.c`	Write a character.
`__lseek()`	`lseek.c`	Set the file position indicator.
`remove()`	`remove.c`	Remove a file.
`rename()`	`rename.c`	Rename a file.

I/O FUNCTIONS

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

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

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

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

Customizing a primitive I/O function on the command line

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

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

1 Copy the file `writechar.c`, which is provided in the `avr\src\lib` directory, to your project directory.

2 Make the required additions to your copy of `writechar.c`, and save it under the same name. The code in the following example uses memory-mapped I/O to write to an LCD display:

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

_STD_BEGIN

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

  LCD_IO = (unsigned char *) 0x8000;
  *LCD_IO = ch;
  // ch on success, -1 on failure.
  return ch;
}

_STD_END
```

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

For example, if your program uses the small memory model and the AT90S8515 microcontroller, compile `writechar.c` from the command line with the command:

```
iccavr writechar -ms --cpu=8515 --library_module
--module_name ?writechar -Ic:\program files\iar
systems\ew23\avr\inc\
```

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

This will create an optimized replacement object module file named `writechar.r90`.

4 Add the following to your XLINK command line:

```
-A writechar
```

5 Link your code using the modified linker command file.

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

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

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

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

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

_STD_BEGIN

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

  LCD_IO = (unsigned char *) 0x8000;
  *LCD_IO = ch;
  // ch on success, -1 on failure.
  return ch;
}

_STD_END
```

3. Add the modified `writechar` to your project.
4. Compile the modified `writechar.c` using the same processor configuration and memory model options as for the project.

   This will create an optimized replacement object module file named `writechar.r90`.
5. Rebuild the project.

Maintaining library files

The IAR XLIB Librarian command `REPLACE-MODULES` allows you to permanently replace the original `CSTARTUP` with your customized version. See *Part 3: The IAR XLIB Librarian* in *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide* for detailed information.

ACCESSING THE I/O SYSTEM

Your application may access the AVR I/O system by using the memory-mapped internal special function registers (SFRs).

To efficiently access the AVR I/O system, the `__io` keyword should be included in the code.

All operators that apply to integral types may be applied to SFR registers. Predefined declarations for the AVR Family are supplied; see *Input and output*, page 29.

Predefined special function registers (SFRs) and interrupt vectors for the most common AVR derivatives are given in the `io`*`chip`*`.h` files. These files are provided with the product. The filename indicates which derivative the file supports; for example, `iom103.h` supports the AT90mega103 derivative. Notice that these target-specific header files can be included also in assembly source code.

I/O files for other AVR derivatives can easily be created by using the I/O file of a similar derivative as a template.

MODULE CONSISTENCY

The overall purpose of maintaining module consistency is to avoid unexpected behavior in the generated code and to make sure that the modules can be linked properly.

PROJECT OPTIONS

In order to support many different AVR derivatives, the AVR IAR Compiler includes a large number of processor variants and memory models.

It is very important to use the same project options for all modules in an application since, for example, the maximum code size and maximum stack size differ between the processor variants and memory models.

DATA TYPES

By default the `double` type is represented by 32-bit numbers in standard IEEE format. The compiler option `--64bit_doubles` allows you to use 64-bit numbers instead; see *Floating-point types*, page 52, and *--64bit_doubles*, page 109, for detailed information.

If your application requires the `double` type, make sure to use the same representation in all modules.

REGISTER USAGE

The AVR IAR Compiler allows you to lock registers that can be used as global register variables. It is possible to lock up to 12 registers.

In order to maintain module consistency, make sure to lock the same number of registers in all modules.

For additional information, see *Register usage*, page 37, and *--lock_regs*, page 94.

RUN-TIME MODEL ATTRIBUTES

Use the assembler directive `RTMODEL` to enforce compatibility between modules. If a module defines a run-time model attribute, all modules that are linked with this module must have the same value for that attribute, or the special wild-card value `*`.

The following table shows the run-time model attributes that are available for the AVR IAR Compiler. These can be included in assembler code or in mixed C or Embedded C++ and assembler code, and will at link time be used by XLINK to ensure consistency between modules.

Run-time model attribute	*Value*	*Description*
`__rt_version`	2.20	Version number for run-time model attributes.
`__no_rampd`	Enabled or disabled	Defined for targets with > 64 Kbytes of data memory. Disabled if the target processor has a `RAMPD` register, otherwise enabled.
`__cpu`	0–6	Corresponds to the `-v` option.

Run-time model attribute	*Value*	*Description*
`__memory_model`	`1-4`	Corresponds to the memory model option: *Memory model* — *Value* Tiny — 1 Small — 2 Large — 3 Generic — 4
`__cpu_name`	`AT90`*`XXXX`*	Corresponds to the processor specified with the `--cpu` compiler option, for example AT90S2313 when the `--cpu=2313` option is used.
`__enhanced_core`	`enabled`	Available only if the compiler option `--enhanced_core` or `--cpu` for a target processor with enhanced core has been specified.
`__double_size`	`32` or `64`	States the size of `double`. The default is 32. Use the compiler option `--64bit_doubles` to override the default.

The easiest way to find the proper settings of the `RTMODEL` directive is to compile a C or Embedded C++ module and examine the result.

If you are using assembler routines in the C or Embedded C++ code, refer to the chapter *Assembler directives reference* in the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide.*

OPTIMIZATIONS

The AVR IAR 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* contains reference information about the following command line options, which are used for specifying optimization type and level: `--no_code_motion`, `--no_cross_call`, `--no_cse`, `--no_inline`, `-s[0-9]`, and `-z[0-9]`.

  Notice that the `--no_cross_call` option decreases the `RSTACK` usage on small derivatives and improves the readibility of the list file when you use high levels of size optimizations.

  Refer to the *AVR IAR Embedded Workbench™ User Guide* for information about the compiler options available in the IAR Embedded Workbench.

- Refer to *Optimize*, page 131, 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, for example a time-critical function.

The purpose of optimization is either to reduce the code size or to improve the execution speed. In the AVR IAR Compiler, however, most speed optimization alternatives also reduce the code 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. We therefore recommend that you use a low optimization level during the development and test phases of your project, and a high optimization level for the release version.

ASSEMBLY LANGUAGE INTERFACE

The AVR IAR Compiler allows assembly language modules to be combined with compiled C or Embedded C++ modules. This is particularly useful for small, time-critical routines that need to be written in assembly language and then called from a C or Embedded C++ main program.

This chapter describes the interface between a C or Embedded C++ main program and the assembly language routines.

C CALLING CONVENTION

REGISTER USAGE

The registers `R4-R15` and `R24-R27` are preserved by the called function. This means that they are saved on the stack if used within the function. All other registers, `R0-R3`, `R16-R23`, and `R30-R31`, are scratch registers.

Registers `R15` and downward, in total 12 registers, can be locked from the command line and used for global register variables; see the compiler option *--lock_regs*, page 94, and *Placing data in registers*, page 113, for details.

The return address stack and the data stack are separate. The return data stack, `RSTACK`, uses internal I/O ports. These ports and the return address stack pointer, `SP`, are described in the `io`*`chip`*`.h` include files provided with the product.

The registers `R16–R23` are used for passing as many as possible of the parameters of the function. The remaining parameters are passed on the stack. The compiler may change the order of the parameters in order to achieve efficient register usage.

Notice that if you call C routines from assembly language, the values in the scratch registers will be destroyed.

PARAMETER PASSING

Local or auto variables are by default placed on the stack. The compiler stores local variables in registers when this is more efficient, regardless of the `register` keyword. The compiler will then use either scratch or non-scratch registers. The variables are stored according to the frequency of their use, in decreasing order—less frequently used—from the stack pointer.

Function return values are passed in registers `R16–R19`.

`struct` and `union` return values larger than 4 bytes are passed using a pointer. The pointer is returned in the return registers, see the table below. If the returned `struct` or `union` is 4 bytes or less, it will be passed in registers. An implicit first parameter is passed to the function, pointing to the memory to be used when storing the return value. If the return value is larger than 4 bytes, it it always passed on the stack.

The `__farfunc` and `__nearfunc` keywords only affect the size of the function pointers. It is always possible to call `__farfunc` functions from `__nearfunc` functions and vice versa.

The called function exits by deallocating auto variables, restoring registers, deallocating stack parameters, and finally performing a `RET` instruction which pops the return address from the return stack.

Functions with ellipsis parameters (...) do not deallocate stack parameters; this is done by the calling function instead.

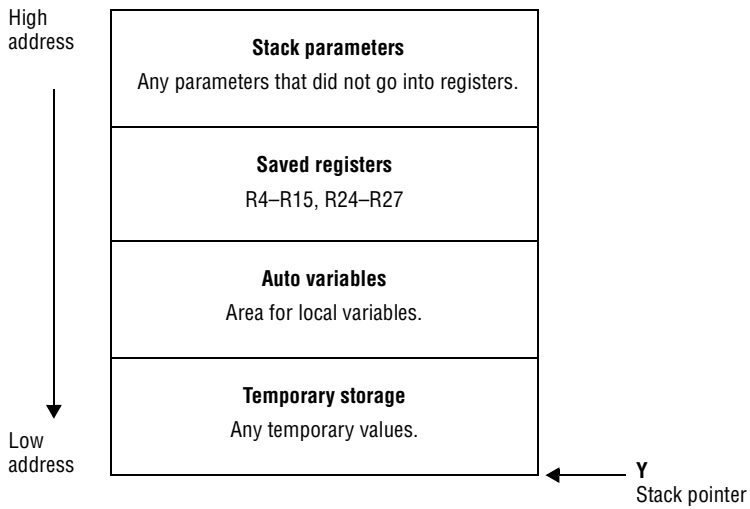
RETURN VALUES

The following table shows in which registers the return values are passed:

Size	*Registers*
1 byte	`R16`
2 bytes	`R16-R17`
3 bytes	`R16-R18`
4 bytes	`R16-R19`

STACK FRAMES

A function call creates a stack frame as follows:



Notice that only the registers that are used will be saved.

MONITOR FUNCTIONS

A monitor function causes interrupts to be disabled during execution of the function. At function entry the status register `SREG` is saved and global interrupts are disabled. At function exit the global interrupt-enabled bit (`I`) is restored in the `SREG` register, and thereby the interrupt status existing before the function call is also restored.

For additional information, see *Monitor functions*, page 118.

CALLING ASSEMBLY ROUTINES FROM C

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

- Conform to the calling convention described on page 37.
- 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)
```

To fulfil these requirements, you should create a code skeleton as described on page 41.

ICCA90 CALLING CONVENTION

This section describes the `ICCA90` (version 1.*x*) calling convention, which can be selected by the use of the compiler option `--version1_calls`, which is described on page 106, or the extended keyword `__version_1`, which is described on page 121.

Register usage
The 30 registers (not counting the `Y` register pair, which is used as the data stack pointer) are divided into two regions: 14 scratch registers (`R0-R3`, `R16-R23`, and `R30-R31`) and 16 local registers. The local registers are preserved across function calls whereas the scratch registers are not.

Register variables and temporary values are placed in local registers, which are preserved during function calls. Scratch registers are used when passing parameters and return values and can also be used for in-between calls for other purposes.

Stack frames and parameter passing
During a function call the calling function places up to two parameters in the scratch registers (as detailed below) and then pushes any other parameters onto the data stack. Control is then passed to the called function with the return address being pushed onto the return stack.

The called function stores any local registers required by the function on the data stack. It allocates space for the function´s auto variables and temporary values and then proceeds to run the function itself.

The called function exits by deallocating auto variables, restoring registers, deallocating stack parameters, and finally performing a `RET` instruction which pops the return address from the return stack.

Functions with ellipsis parameters (...) do not deallocate stack parameters; this is done by the calling function instead.

The leftmost two parameters are passed in registers if they are scalar and up to 32 bits in size. Unscalar values, structs, unions, and actual parameters whose corresponding formal parameter is an elipsis (...), are always passed on the stack, as are all parameters after the second.

The following table shows some of the possible combinations:

*Parameters**	*Parameter 1*	*Parameter 2*
`f(b1,b2,…)`	`R16`	`R20`
`f(b1,w2,…)`	`R16`	`R20, R21`
`f(w1,l1,…)`	`R16, R17`	`R20, R21, R22, R23`
`f(l1,b2,…)`	`R16, R17, R18, R19`	`R20`
`f(l1,l2,…)`	`R16, R17, R18, R19`	`R20, R21, R22, R23`

* Where `b` denotes an 8-bit data type, `w` denotes a 16-bit data type, and `l` denotes a 32-bit data type. If the first and/or second parameter is a 3-byte pointer, it will be passed in `R16-R19` or `R20-R22` respectively.

See the table on page 38 for information about how the return values are passed.

Interrupt functions
Interrupt functions differ from ordinary C functions in that:

- Flags and scratch registers are saved.
- Calls to interrupt functions are made via interrupt vectors, direct calls are not allowed.
- No arguments can be passed to an interrupt function.

CREATING SKELETON CODE

The recommended way to create an assembly language routine with the correct interface is to start with an assembly language source created by the compiler. Notice that you must create a skeleton for each given routine. You must also test the register allocation since the calling convention may change the order of arguments.

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

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

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

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

Note: 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.

The skeleton code should be compiled as follows:

```
iccavr shell --cpu=2313 -lA . -z3
```

The `-lA` option creates an assembly language output file including C or Embedded C++ source lines as assembler comments. The `.` (period) specifies that the assembler file should be named in the same way as the C or Embedded C++ module, i.e. `shell`, but with the filename extension `s90`.

In the IAR Embedded Workbench, use the same options as for your project to compile the skeletod code but make sure to specify a low level of optimization.

The result is the assembler source `shell.s90` containing the declarations, function call, function return, and variable accesses.

Viewing 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 (autovariables).
- The following list shows an example of an assembler output file with C or Embedded C++ source comments.The list file has been slightly modified to work as a good example.

```
NAME shell

        RTMODEL "__cpu", "0"
        RTMODEL "__cpu_name", "AT90S2313"
        RTMODEL "__memory_model", "1"
        RTMODEL "__rt_version", "2.20"

        RSEG CSTACK:DATA:NOROOT(0)
        RSEG RSTACK:DATA:NOROOT(0)


        PUBLIC func
        FUNCTION func,0203H
        PUBLIC globChar
        PUBLIC globInt
        PUBLIC globLong
        PUBLIC main
        FUNCTION main,021a03H
        LOCFRAME RSTACK, 2, STACK
;     1 char globChar;
;     2 int globInt;

        RSEG TINY_Z:DATA:NOROOT(0)
;     3 long globLong;
globLong:
 DS 4
globInt:
 DS 2
globChar:
 DS 1
;     4

        RSEG CODE:CODE:NOROOT(1)
```

```
;     5 char func(int arg1, char arg2, long arg3)
;     6 {
;     7   char locChar = arg2; /* set local */
func:
 MOV   R19,R18
;     8
;     9   globInt = arg1; /* use globInt/arg1 */
 LDI   R30,LOW(globLong)
 LDI   R31,globLong >> 8
 STD   Z+4,R16
 STD   Z+5,R17
;    10   globChar = arg2; /* use globChar/arg2 */
 LDI   R30,LOW(globLong)
 LDI   R31,globLong >> 8
 STD   Z+6,R18
;    11   globLong = arg3; /* use globLong/arg3 */
 LDI   R30,LOW(globLong)
 LDI   R31,globLong >> 8
 ST    Z,R20
 STD   Z+1,R21
 STD   Z+2,R22
 STD   Z+3,R23
;    12
;    13   return locChar; /* set return value */
 MOV   R16,R19
 RET
;    14 }
;    15

        RSEG CODE:CODE:NOROOT(1)
;    16 void main(void)
;    17 {
;    18   long locLong = globLong;
main:
        FUNCALL main, func
        LOCFRAME RSTACK, 2, STACK
 LDI   R30,LOW(globLong)
 LDI   R31,globLong >> 8
 LD    R20,Z
 LDD   R21,Z+1
 LDD   R22,Z+2
```

```
 LDD   R23,Z+3
;    19   globChar = func(globInt, globChar, locLong);
 LDI   R30,LOW(globLong)
 LDI   R31,globLong >> 8
 LDD   R18,Z+6
 LDI   R30,LOW(globLong)
 LDI   R31,globLong >> 8
 LDD   R16,Z+4
 LDD   R17,Z+5
 RCALL func
 LDI   R30,LOW(globLong)
 LDI   R31,globLong >> 8
 STD   Z+6,R16
;    20 }
 RET

        END
;
;     68 bytes in segment CODE
;      7 bytes in segment TINY_Z
;
;     68 bytes of CODE memory
;      7 bytes of DATA memory
;
;Errors: none
;Warnings: none
```

For information about the run-time model attributes used in this example, see *Module consistency*, page 33.

COMPILER FUNCTION DIRECTIVES

The compiler function directives are generated by the compiler to pass information about functions and function calls to the IAR XLINK Linker. To view these directives, you must create an assembler list file by using the compiler option **Assembler file** (`-lA`).

SYNTAX

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

PARAMETERS

label	Label to be decared 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; one of 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.

INTERRUPT HANDLING

INTERRUPT FUNCTIONS

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

- All registers that are changed by the interrupt service routine must be saved. The compiler will automatically generate code to save all used registers.
- The routine must also save the values of the `SREG` status register and possible `RAMP` registers.
- Interrupt routines may call reentrant functions, but the use of lengthy functions should be avoided to prevent conflicts with real-time interrupts.

Notice than interrupts should not be enabled within an interrupt routine.

Templates for an interrupt service routine module written in C++ (`tutor3.cpp`) is provided with the product. For an example where it is used, see the third compiler tutorial in the *AVR IAR Embedded Workbench™ User Guide.*

DEFINING INTERRUPT VECTORS

When you have an assembler-written interrupt function, you must install it in the interrupt vector table. See the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide* for a description.

The interrupt vectors are located in the `INTVEC` segment.

EMBEDDED C++

The C calling convention, which is described on page 37, does not apply to Embedded C++ functions. Most importantly, a function name is not sufficient to identify an Embedded C++ function. The scope and the type of the function are also required to guarantee type-safe linkage and to resolve overloading.

Another difference is that non-static member functions get an extra, hidden argument, the `this` pointer.

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

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

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

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

```
class X;

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

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

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

PART 2: COMPILER REFERENCE

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

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

DATA REPRESENTATION

This chapter describes the data types and pointers supported in the AVR IAR Compiler.

See the chapter *Efficient coding techniques* for information about which data types and pointers provide the most efficient code.

DATA TYPES

This section describes how the AVR IAR Compiler represents each of the C data types.

The AVR IAR Compiler supports all ISO/ANSI C basic data types. Signed variables are stored in two's complement form.

Notice that the AVR microcontroller has a byte-oriented architecture. This means that while the compiled code has alignment 2, all data types have alignment 1.

INTEGER TYPES

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

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

Data type	*Size*	*Range*
`signed long long`	64 bits	-2^{63} to $2^{63}-1$
`unsigned long long`	64 bits	0 to $2^{64}-1$

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

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

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

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

By default the AVR IAR 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

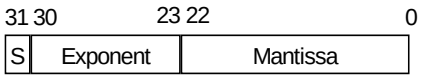
Floating-point values are represented by 32-bit numbers in standard IEEE format.

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

Type	*Range (+/-)*	*Decimals*	*Exponent*	*Mantissa*
`float`	± 1.18E-38 to ± 3.39E + 38	7	8	23
`double`	± 2.23E-308 to ± 1.79E + 308	15	11	52

Notice that the `double` size is controlled using the compiler option `--64bit_doubles`; see page 109 for additional information.

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

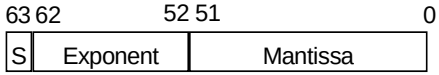


The value of the number is:

$(-1)^{S} * 2^{(Exponent-127)} * 1.Mantissa$

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

8-byte floating-point format
The memory layout of 8-byte floating-point numbers is:



The value of the number is:

$(-1)^{S} * 2^{(Exponent-1023)} * 1.Mantissa$

The precision of the float operators (+, -, *, and /) is approximately 15 decimal digits.

Special cases
For both 4-byte and 8-byte floating-point formats:

- 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.
- Not a number (NaN) is represented by setting the exponent to the highest positive value and the mantissa to a non-zero value. The value of the sign bit is ignored.

POINTERS

This section describes the available function pointers and data pointers.

FUNCTION POINTERS

The following function pointers are available:

Pointer	*Address range*	*Pointer size*	*Description*
`__nearfunc`	`0-0x1FFFE`	2 bytes	Can be called from any part of the code memory, but must reside in the first 128K of that space.
`__farfunc`	`0-0x7FFFFE`	3 bytes	No restrictions on code placement.

DATA POINTERS

The following data pointers are available:

Keyword	*Storage*	*Memory*	*Range*
`__tiny`	8 bits	Data	`0x0–0xFF`
`__near`	16 bits	Data	`0x0–0xFFFF`
`__far`	24 bits	Data	`0x0–0xFFFFFF` (16-bit arithmetics)
`__huge`	24 bits	Data	`0x0–0xFFFFFF`
`__tinyflash`	8 bits	Code	`0x0–0xFF`
`__flash`	16 bits	Code	`0x0–0xFFFF`
`__farflash`	24 bits	Code	`0x0–0xFFFFFF` (16-bit arithmetics)
`__hugeflash`	24 bits	Code	`0x0–0xFFFFFF`
`__eeprom`	8 or 16 bits	EEPROM	`0x0–0xFF` or `0x0–0xFFFF`

Keyword	*Storage*	*Memory*	*Range*
`__generic`	1 + 15 bits 1 + 23 bits	Data/Code	The most significant bit (MSB) determines whether `__generic` points to `CODE` (1) or `DATA` (0). The small generic pointer is generated for processor options `-v0` and `-v1`.

CASTING

Casting a *value* of an integer type to a pointer of a smaller size is performed by truncation. Casting to a larger pointer is performed by zero extension.

Casting a *pointer type* to a smaller integer type is performed by truncation. Casting to a larger integer type is performed by first casting the pointer to the largest possible pointer that fits in the integer.

Casting data pointers to function pointers and vice versa is illegal.

Casting function pointers to integer types would give an undefined result.

size_t

`size_t` is the unsigned integer type required to hold the maximum size of an object. The following table shows the `typedef` of `size_t` depending on the processor option:

Processor option	*Typedef*
`-v0`, `-v1`	`unsigned int`
`-v2`, `-v3`, `-v4`, `-v5`, and `-v6`	`unsigned long`

ptrdiff_t

`ptrdiff_t` is the type of integer required to hold the difference between two pointers to elements of the same array. The following table shows the `typedef` of `ptrdiff_t` depending on the processor option:

Processor option	*Typedef*
`-v0`, `-v1`	`signed int`
`-v2`, `-v3`, `-v4`, `-v5`, and `-v6`	`signed long`

STRUCTURE TYPES

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

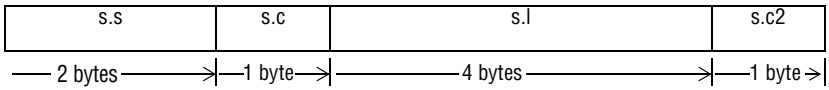
GENERAL LAYOUT

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

For example:

```
struct {
   short s;     // stored in byte 0 and 1
   char c;      // stored in byte 2
   long l;      // stored in byte 3, 4, 5, and 6
   char c2;     // stored in byte 7
} s;
```

The following diagram shows the layout in memory:



ANONYMOUS STRUCTURES AND UNIONS

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

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

void f()
{
```

```
    st.l = 5;
}
```

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

```
__no_init volatile __io union
  {
    unsigned char PIND;
    struct
      {
        unsigned char Bit0:1;
        unsigned char Bit1:1;
        unsigned char Bit2:1;
        unsigned char Bit3:1;
        unsigned char Bit4:1;
        unsigned char Bit5:1;
        unsigned char Bit6:1;
        unsigned char Bit7:1;
      } PIND_bits;
  } @ 0x10
```

This declares an SFR byte register (`PIND`) at address `0x10`.

Notice that anonymous structures and unions are only available when language extensions are enabled in the AVR IAR Compiler.

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

Use the `-e` compiler option to enable language extensions. See *-e*, page 89 for additional information.

SEGMENTS

The AVR IAR 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 assembly language modules, and are also useful when interpreting the assembly language output from the compiler.

This chapter mentions many of the extended keywords. For detailed information about the keywords, see the chapter *Extended keywords*.

For information about how to define segments in the linker command file, see *Customizing the linker command file*, page 18.

INTRODUCTION

The following section describes how the segment names are constructed, and how the declaration of an object affects how it is placed in a segment.

NAMING CONVENTION

Variable segments
All variable segment names consist of a prefix named after the storage keyword plus a suffix that tells what kind of segment it is.

For example, the keyword:

```
__near
```

would yield the prefix `NEAR`.

For information about the storage keywords, see *Data storage*, page 112.

Segment suffix
The suffix states the kind of segment. The following suffixes are available:

Suffix	*Description*
`_AN`	Located data, for example `EEPROM_AN`. For additional information, see *Absolute location*, page 116, and *Segment placement*, page 117.
`_C`	Constants. See *-y*, page 107.
`_F`	Flash memory. See *Storing data in code memory*, page 112.

Suffix	*Description*
_I	Initialized memory. See *-y*, page 107, for additional information.
_ID	Initial data. Copied to _I at startup.
_N	Uninitialized memory. See *__no_init*, page 115.
_Z	Zero initialized memory.

DECLARATION OF OBJECTS

Uninitialized variables

The __no_init keyword gives the user the responsibility for initializing objects. 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. __no_init const objects are normally located, but may be useful if you want to declare the object in a special data segment and to assign the address at link time.

For additional information about the keyword, see *__no_init*, page 115.

Static objects

Declaring an object as volatile does not affect its placement. It is possible to combine volatile with all of the extended keywords.

Auto objects

const or non-const auto objects are always stored on the stack in segment CSTACK. They are not allowed to have memory attributes or the __no_init attribute.

Notice that the address range for some segments may exceed the available address range on your selected derivative. The address range should then be truncated from the end to where the address range of the derivative ends. For example, the AT90S8515 has a 16 bit external address range allowing a data space of 64 Kbytes or 0-0xFFFF. The address range of the ABSOLUTE segment below should the be read as 0x0-0xFFFF.

SUMMARY OF SEGMENTS

The following table lists the segments that are available in the AVR IAR Compiler. Notice that *located* denotes absolute location using the `@` operator or the `#pragma location` directive.

Segment	*Description*
`ABSOLUTE`	Used for located variables.
`CODE`	Holds program code declared `__nearfunc`.
`CSTACK`	Holds the internal data stack.
`DIFUNCT`	Holds pointers to constructor blocks that should be executed by `CSTARTUP` before `main` is called.
`EEPROM_AN`	Holds initialized located EEPROM variables.
`EEPROM_I`	Holds EEPROM variables that are initialized when downloaded to the chip.
`EEPROM_N`	Holds initialized EEPROM variables.
`FAR_C`	Used for storing `__far` constant data, including literal strings.
`FAR_F`	Holds static and global `__farflash` variables.
`FAR_I`	Holds static and global `__far` variables that have been declared with non-zero initial values.
`FAR_ID`	Holds initial values for the variables located in the `FAR_I` segment.
`FAR_N`	Holds static and global `__far` variables to be placed in non-volatile memory.
`FAR_Z`	Holds static and global `__far` variables that have been declared without initial value or with zero initial values.
`FARCODE`	Holds program code declared `__farfunc`.
`HEAP`	Holds the heap data used by `malloc`, `calloc`, and `free`.
`HUGE_C`	Used for storing `__huge` constant data, including literal strings.
`HUGE_F`	Holds static and global `__hugeflash` variables.
`HUGE_I`	Holds static and global `__huge` variables that have been declared with non-zero initial values.

Segment	*Description*
HUGE_ID	Holds initial values for the variables located in the HUGE_I segment.
HUGE_N	Holds static and global __huge variables to be placed in non-volatile memory.
HUGE_Z	Holds static and global __huge variables that have been declared without initial value or with zero initial values.
INITTAB	Contains compiler-generated table entries that describe the segment initialization that will be performed at system start up.
INTVEC	Contains the reset and interrupt vectors.
NEAR_C	Used for storing __tiny and __near constant data, including literal strings.
NEAR_F	Holds static and global __flash variables.
NEAR_I	Holds static and global __near variables that have been declared with non-zero initial values.
NEAR_ID	Holds initial values for the variables located in the NEAR_I segment.
NEAR_N	Holds static and global __near to be placed in non-volatile memory.
NEAR_Z	Holds static and global __near variables that have been declared without initial value or with zero initial values.
RSTACK	Holds the internal return stack.
SWITCH	Holds switch tables for all functions.
TINY_F	Holds static and global __tinyflash variables.
TINY_I	Holds static and global __tiny variables that have been declared with non-zero initial values.
TINY_ID	Holds initial values for the variables located in the TINY_I segment.
TINY_N	Holds static and global __tiny variables to be placed in non-volatile memory.

Segment	*Description*
`TINY_Z`	Holds static and global `__tiny` variables that have been declared without initial value or with zero initial values.

The following sections describe each segment.

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

ABSOLUTE

Holds located variables.

TYPE

Read/write.

MEMORY AREA

Data. The address range is `0x0-0xFFFFFF`.

DESCRIPTION

Used for located variables, i.e. variables that have been assigned an absolute location by use of the `@` operator or `#pragma location`.

CODE

Holds user program code.

TYPE

Read-only.

MEMORY AREA

Code. The address range is `0x0-0x01FFFE`.

DESCRIPTION

Holds user program code that has been declared `__nearfunc` and various library routines.

Notice that any assembly language routines called from C or Embedded C++ must meet the calling convention in use. For more information, see *C calling convention*, page 37, and *Embedded C++*, page 47.

CSTACK

Holds the data stack.

TYPE

Read/write.

MEMORY AREA

Data. The address range depends on the memory model:

Memory model	*Address range*	*Comment*
Tiny	`0x0-0xFF`	
Small	`0x0-0xFFFF`	
Large	`0x0-0xFFFFFF`	Maximum 64K stack.
Generic	`0x0-0xFFFFFF`	Maximum 64K stack.

DESCRIPTION

Holds the internal data stack. This segment and its length is normally defined in the linker command file with the following command:

`-Z(DATA)CSTACK+`*`nn`*`=`*`start`*

or

`-Z(DATA)CSTACK=`*`start-end`*

where *`nn`* is the length, *`start`* is the first memory location, and *`end`* is the last memory location.

DIFUNCT

Holds pointers to constructor blocks in EC++ code.

TYPE

Read-only.

MEMORY AREA

Code. The address range is `0x0-0xFFFF`.

DESCRIPTION

When using EC++ and global objects, it is necessary to call the constructor methods of these global objects before `main` is called.

EEPROM_AN

Used for programming the inbuilt EEPROM.

TYPE

Read/write.

MEMORY AREA

EEPROM.

DESCRIPTION

This segment is not copied to EEPROM during system start up. Instead it is used for programming the EEPROM during the download of the code.

Use the command line option --eeprom_size to set the address range for this segment; see *--eeprom_size*, page 90, for additional information.

EEPROM_I

Used for programming the inbuilt EEPROM.

TYPE

Read/write.

MEMORY AREA

EEPROM.

DESCRIPTION

This segment is not copied to EEPROM during system start up. Instead it is used for programming the EEPROM during the download of the code.

Use the command line option --eeprom_size to set the address range for this segment; see *--eeprom_size*, page 90, for additional information.

EEPROM_N

Used for programming the inbuilt EEPROM.

TYPE

Read/write.

MEMORY AREA

EEPROM.

DESCRIPTION

This segment is used for programming the EEPROM during the download of the code.

Use the command line option --eeprom_size to set the address range for this segment; see *--eeprom_size*, page 90, for additional information.

FARCODE

Holds __farfunc program code.

TYPE

Read-only.

MEMORY AREA

Code. The address range is 0x0-0x7FFFFE.

DESCRIPTION

Holds user program code that has been declared __farfunc.

FAR_C

Holds __far constant data, including string literals.

TYPE

Read-only.

MEMORY AREA

Data. The address range is 0x0-0xFFFFFF.

DESCRIPTION

Holds __far constant data, including string literals.

Note: This segment is located in external ROM. Systems without external ROM may not use this segment.

When the -y compiler option is used, __far constant data is located in the FAR_I segment.

FAR_F

Holds static and global `__farflash` variables.

TYPE

Read-only.

MEMORY AREA

Code. The address range is `0x0-0x7FFFFF`.

DESCRIPTION

Holds static and global `__farflash` variables and aggregate initializers.

FAR_I

Holds `__far` variables.

TYPE

Read/write.

MEMORY AREA

Data. The address range is `0x0-0xFFFFFF`.

DESCRIPTION

Holds static and global `__far` variables that have been declared with non-zero initial values.

When the `-y` compiler option is used, `__far` constant data is located in this segment.

FAR_ID

Holds variable initializers.

TYPE

Read-only.

MEMORY AREA

Code. The address range is `0x0-0x7FFFFF`.

DESCRIPTION

Holds initial values for the variables located in the FAR_I segment. These values are copied from FAR_ID to FAR_I during system initialization.

FAR_N

Holds static and global variables.

TYPE

Read/write.

MEMORY AREA

Data. The address range is 0x0-0xFFFFFF.

DESCRIPTION

Holds static and global __far variables that will not be initialized at system startup, for example variables that are to be placed in non-volatile memory. These variables have been declared __no_init, created __no_init by use of the #pragma memory directive, or allocated by the compiler.

FAR_Z

Holds static and global variables.

TYPE

Read/write.

MEMORY AREA

Data. The address range is 0x0-0xFFFFFF.

DESCRIPTION

Holds static and global __far variables that have been declared without initial value or with zero initial values.

HEAP

Used for the heap.

TYPE

Read/write.

MEMORY AREA

Data. The address range depends on the memory model:

Memory model	*Address range*
Tiny	`0x0-0xFF`
Small	`0x0-0xFFFF`
Large	`0x0-0xFFFFFF`
Generic	`0x0-0xFFFFFF`

DESCRIPTION

Holds the heap data used by `malloc`, `calloc`, and `free`.

This segment and its length is normally defined in the linker command file by the command:

`-Z(DATA)HEAP+`*`nn`*`=`*`start`*

where *`nn`* is the length and *`start`* is the location.

HUGE_C

Holds `__huge` constant data, including string literals.

TYPE

Read-only.

MEMORY AREA

Data. The address range is `0x0-0xFFFFFF`.

DESCRIPTION

Holds `__huge` constant data, including string literals.

Note: This segment is located in external ROM. Systems without external ROM may not use this segment.

HUGE_F

Holds static and global `__hugeflash` variables.

TYPE

Read-only.

MEMORY AREA

Code. The address range is `0x0-0xFFFFFF`.

DESCRIPTION

Holds static and global `__hugeflash` variables and aggregate initializers.

HUGE_I

Holds `__huge` variables.

TYPE

Read/write.

MEMORY AREA

Data. The address range is `0x0-0xFFFFFF`.

DESCRIPTION

Holds static and global `__huge` variables that have been declared with non-zero initial values.

When the `-y` compiler option is used, `__huge` constant data is located in this segment.

HUGE_ID

Holds variable initializers.

TYPE

Read-only.

MEMORY AREA

Code. The address range is `0x0-0x7FFFFF`.

DESCRIPTION

Holds initial values for the variables located in the HUGE_I segment. These values are copied from HUGE_ID to HUGE_I during system initialization.

HUGE_N

Holds static and global variables.

TYPE

Read/write.

MEMORY AREA

Data. The address range is 0x0-0xFFFFFF.

DESCRIPTION

Holds static and global __huge variables that will not be initialized at system startup, for example variables that are to be placed in non-volatile memory. These variables have been declared __no_init, created __no_init by use of the #pragma memory directive, or allocated by the compiler.

HUGE_Z

Holds static and global variables.

TYPE

Read/write.

MEMORY AREA

Data. The address range is 0x0-0xFFFFFF.

DESCRIPTION

Holds static and global __huge variables that have been declared without initial value or with zero initial values.

INITTAB

Segment initialization descriptions.

TYPE

Read-only.

MEMORY AREA

Code. The address range is `0x0-0xFFFF`.

DESCRIPTION

Contains compiler-generated table entries that describe the segment initialization which will be performed at system startup.

INTVEC

Interrupt vector table.

TYPE

Read-only.

MEMORY AREA

Code. The address range is approximately `0-64`.

DESCRIPTION

Holds the interrupt vector table generated by the use of the `__interrupt` extended keyword.

Note: This segment *must* be placed at address `0` and forwards.

NEAR_C

Holds `__tiny` and `__near` constant data, including string literals.

TYPE

Read-only.

MEMORY AREA

Data. The address range is `0x0-0xFFFF`.

DESCRIPTION

Holds `__tiny` and `__near` constant data, including string literals.

Note: This segment is located in external ROM. Systems without external ROM may not use this segment.

NEAR_F

Holds static and global `__flash` variables.

TYPE

Read-only.

MEMORY AREA

Code. The address range is `0x0-0xFFFF`.

DESCRIPTION

Holds static and global `__flash` variables and aggregate initializers.

NEAR_I

Holds `__near` variables.

TYPE

Read/write.

MEMORY AREA

Data. The address range is `0x0-0xFFFF`.

DESCRIPTION

Holds static and global `__near` variables that have been declared with non-zero initial values.

When the `-y` compiler option is used, `NEAR_C` data (`__near` or `__tiny`) is located in this segment.

NEAR_ID

Holds variable initializers.

TYPE

Read-only.

MEMORY AREA

Code. The address range is 0x0-0x7FFFFF.

DESCRIPTION

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

NEAR_N

Holds static and global variables.

TYPE

Read/write.

MEMORY AREA

Data. The address range is 0x0-0xFFFF.

DESCRIPTION

Holds static and global __near variables that will not be initialized at system startup, for example variables that are to be placed in non-volatile memory. These variables have been declared __no_init, created __no_init by use of the #pragma memory directive, or allocated by the compiler.

NEAR_Z

Holds static and global variables.

TYPE

Read/write.

MEMORY AREA

Data. The address range is 0x0-0xFFFF.

DESCRIPTION

Holds static and global __near variables that have been declared without initial value or with zero initial values.

RSTACK

Internal return stack.

TYPE

Read/write.

MEMORY AREA

Data. The address range is `0x0-0xFFFF`.

DESCRIPTION

Holds the internal return stack.

SWITCH

Code memory.

TYPE

Read-only.

MEMORY AREA

Code. The address range is `0x0-0xFFFF`.

DESCRIPTION

The `SWITCH` segment is for compiler internal use only and should always be defined. The segment allocates, if necessary, jump tables for C switch statements.

TINY_F

Holds static and global `__tinyflash` variables.

TYPE

Read-only.

MEMORY AREA

Code. The address range is `0x0-0xFF`.

DESCRIPTION

Holds static and global `__tinyflash` variables and aggregate initializers.

TINY_I

Holds `__tiny` variables.

TYPE

Read/write.

MEMORY AREA

Data. The address range is `0x0-0xFF`.

DESCRIPTION

Holds static and global `__tiny` variables that have been declared with non-zero initial values.

When the `-y` compiler option is used, `FAR_C` data is located in this segment.

TINY_ID

Holds variable initializers.

TYPE

Read-only.

MEMORY AREA

Code. The address range is `0x0-0x7FFFFF`.

DESCRIPTION

Holds initial values for the variables located in the `TINY_I` segment. These values are copied from `TINY_ID` to `TINY_I` during system initialization.

TINY_N

Holds static and global variables.

TYPE

Read/write.

MEMORY AREA

Data. The address range is `0x0-0xFF`.

DESCRIPTION

Holds static and global `__tiny` variables that will not be initialized at system startup, for example variables that are to be placed in non-volatile memory. These variables have been declared `__no_init`, created `__no_init` by use of the `#pragma memory` directive, or allocated by the compiler.

TINY_Z

Holds static and global variables.

TYPE

Read/write.

MEMORY AREA

Data. The address range is `0x0-0xFF`.

DESCRIPTION

Holds static and global `__tiny` variables that have been declared without initial value or with zero initial values.

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

SETTING COMPILER OPTIONS

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

```
iccavr 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`:

```
iccavr 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:

```
iccavr prog -DDEBUG=1
```

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

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

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

Specifying parameters

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

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

or as

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

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

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

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

```
--diag_suppress=Pe0001
```

or

```
--diag_suppress Pe0001
```

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

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

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

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

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

```
iccavr 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`:

```
iccavr prog -l ---r
```

Error return codes
The AVR IAR 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.
3	There were fatal errors (compiler aborted).

ENVIRONMENT VARIABLES

Compiler options can also be specified in the `QCCAVR` 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 AVR IAR Compiler:

Environment variable	*Description*
`C_INCLUDE`	Specifies directories to search for include files; for example: `C_INCLUDE=c:\iar\avr\inc;c:\headers`
`QCCAVR`	Specifies command line options; for example: `QCCAVR=-lA asm.lst -z9`

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

OPTIONS SUMMARY

The following table summarizes the compiler command line options:

Command line option	*Description*
`--char_is_signed`	'char' is 'signed char'
`--cpu=`*cpu*	Processor variant
`--cross_call_passes=`*N*	Cross-call optimization
`-D`*symb*[`=`*value*]	Defines preprocessor symbols
`--debug`	Generates debug info
`--dependencies={i|m}`	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
`--disable_direct_mode`	Disables direct addressing mode
`-e`	Enables language extensions
`--ec++`	Enables Embedded C + + syntax
`--eeprom_size=`*N*	Specifies EEPROM size
`--enhanced_core`	Enables enhanced instruction set
`-I`*path*	Includes file path
`--initializers_in_flash`	Places aggregate initializers in flash memory
`-l[c|C|a|A][N]` *filename*	Creates list file
`--library_module`	Makes library module
`--lock_regs` *N*	Lock sregister
`-m`*name*	Memory model

Command line option	*Description*
`--memory_model=`*`name`*	Memory model
`--module_name=`*`name`*	Sets object module name
`--no_code_motion`	Disables code motion optimization
`--no_cross_call`	Disables cross-call optimization
`--no_cse`	Disables common sub-expression elimination
`--no_inline`	Disables function inlining
`--no_rampd`	Uses `RAMPZ` instead of `RAMPD`
`--no_ubrof_messages`	Minimizes object file size
`--no_unroll`	Disables loop unrolling
`--no_warnings`	Disables all warnings
`-o` *`filename`*	Sets object filename
`--only_stdout`	Uses standard output only
`--preprocess=[c][n][l]` *`filename`*	Preprocessor output to file
`-r`	Generates debug information
`--remarks`	Enables remarks
`--root_variables`	Specifies variables as `__root`
`-s[0-9]`	Optimizes for speed
`--segment` *`memory_attr=segment_name`*	Changes segment name base
`--silent`	Sets silent operation
`--strict_ansi`	Enables strict ISO/ANSI
`-v[0|1|2|3|4|5|6]`	Processor variant
`--version1_calls`	Uses `ICCA90` calling convention

Command line option	*Description*
`--warnings_affect_exit_code`	Warnings affects exit code
`--warnings_are_errors`	Treats all warnings as errors
`-y`	Places constants and literals
`-z[0-9]`	Optimizes for size
`--zero_register`	Specifies register R15 as zero register
`--64bit_doubles`	Use 64-bit doubles.

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

--char_is_signed

'char' is 'signed char'.

SYNTAX

```
--char_is_signed
```

DESCRIPTION

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

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

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

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

--cpu

Processor variant.

SYNTAX

```
--cpu=cpu
```

DESCRIPTION

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

For example, use the following command to specify the AT90S4414 derivative:

```
--cpu=4414
```

See *Processor*, page 11, for a summary of the available processor variants.

Notice that to specify the processor, you can use either the `--cpu` option or the `-v` option. The `--cpu` option is, however, more precise since implicit assumptions are made about the processor when you use the `-v` option. For additional information, see page 13.

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

--cross_call_passes

Cross-call optimization.

SYNTAX

```
--cross_call_passes=N
```

DESCRIPTION

Use this option to decrease the `RSTACK` usage by running the cross-call optimizer *N* times, where *N* can be 1–5. The default is to run it twice.

For additional information, see *--no_cross_call*, page 97.

Note: Use this option if you have a target processor with a hardware stack or a small internal return stack segment, `RSTACK`.

This option is related to the **Optimizations** options in the **ICCAVR** category in the IAR Embedded Workbench.

-D

Defines preprocessor symbols.

SYNTAX

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

DESCRIPTION

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

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

`-D`*symb*

is equivalent to:

`#define` *symb*

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

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

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

Production version: `iccavr prog`

Test version: `iccavr prog -Dtestver`

This option can be used one or more times.

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

--debug, -r

Generates debug information.

SYNTAX

```
 --debug
-r
```

DESCRIPTION

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

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

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

--dependencies

Lists file dependencies.

SYNTAX

```
--dependencies={i|m}
```

DESCRIPTION

Causes the compiler to list file dependencies. The following table shows the effect of the modifiers:

Option and modifier	*Description*
`--dependencies=i`	Include filename only (default)
`--dependencies=m`	Use Make-file style

--diag_error

Treats the specified diagnostic messages as errors.

SYNTAX

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

DESCRIPTION

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

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

```
--diag_error=Pe117
```

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

--diag_remark

Treats the specified diagnostic messages as remarks.

SYNTAX

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

DESCRIPTION

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

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

```
--diag_remark=Pe177
```

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

--diag_suppress

Suppresses the specified diagnostics messages.

SYNTAX

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

DESCRIPTION

Suppresses the output of diagnostics for the specified tags.

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

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

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

--diag_warning

Treats the specified diagnostic messages as warnings.

SYNTAX

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

DESCRIPTION

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

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

```
--diag_warning=Pe826
```

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

--disable_direct_mode

Disables direct addressing mode.

SYNTAX

```
--disable_direct_mode
```

DESCRIPTION

This option prevents the compiler from generating the direct addressing mode instructions `LDS` and `STS`.

Using this option may in some cases reduce the size of the object code.

-e

Enables language extensions.

SYNTAX

```
-e
```

DESCRIPTION

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

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

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

For additional informatioin, see *Language extensions overview, page 8.*

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

--ec++

Enables the Embedded C++ syntax.

SYNTAX

```
--ec++
```

DESCRIPTION

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

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

--eeprom_size

Specifies the `EEPROM` size.

SYNTAX

```
--eeprom_size=N
```

DESCRIPTION

Use this option to enable the `__eeprom` extended keyword by specifying the size of the inbuilt `EEPROM`. The value *N* can be 0–65536.

Note: To use the `__eeprom` extended keyword, the language extensions must be enabled. For additional information, see *-e*, page 89, and *Language*, page 131.

This option is related to the **Code** options in the **ICCAVR** category in the IAR Embedded Workbench.

--enhanced_core

Enables the enhanced instruction set.

SYNTAX

```
--enhanced_core
```

DESCRIPTION

Use this option to allow the compiler to generate instructions from the enhanced instruction set that is available in some AVR derivatives, for example AT90mega161.

This option corresponds to the **Enhanced core** option in the **General** category in the IAR Embedded Workbench.

-I

Specifies `#include` file paths.

SYNTAX

`-I`*path*

DESCRIPTION

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

```
iccavr prog -I\mylib1
```

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

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

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

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

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

  it searches the following directories for the file to include:

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

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

- When the compiler encounters the name of an #include file in double quotes such as:

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

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

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

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

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

  ```
  iccavr ..\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.

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

--initializers_in_flash

Places aggregate initializers in flash memory.

SYNTAX

```
--initializers_in_flash
```

DESCRIPTION

Use this option to place aggregate initializers in flash memory. These initializers are otherwise placed either in the external `const` segment or in the initialized data segments if the compiler option `-y` was also specified.

See *-y*, page 107, and the chapter *Segments* for additional information.

This option is related to the **Code** options in the **ICCAVR** category in the IAR Embedded Workbench.

-l

Generates a listing to the specified filename.

SYNTAX

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

DESCRIPTION

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

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

```
iccavr prog -l list
```

The following modifiers are available:

Option modifier	*Description*
`a`	Assembler file
`A` (`N` is implied)	Assembler file with C or Embedded C++ source as comments
`c`	C or Embedded C++ list file
`C` (default)	C or Embedded C++ list file with assembler source as comments

Option modifier	*Description*
N	No diagnostics in file

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

--library_module

Makes module a library module.

SYNTAX

```
--library_module
```

DESCRIPTION

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

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

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

--lock_regs

Locks the specified registers.

SYNTAX

```
--lock_regs N
```

DESCRIPTION

Use this option to lock registers that are to be used for global register variables. The value *N* can be `0–12` where `0` means that no registers are locked. When you use this option, the registers `R15` and downwards will be locked.

In order to maintain module consistency, make sure to lock the same number of registers in all modules.

This option is related to the **Code** options in the **ICCAVR** category in the IAR Embedded Workbench.

-m, --memory_model

Specifies the data memory model.

SYNTAX

```
-m[tiny|t|small|s|large|l|generic|g]
--memory_model=[tiny|t|small|s|large|l|generic|g]
```

DESCRIPTION

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

By default the compiler generates code for the tiny memory model for all processor options except `-v4` and `-v6` where the small memory model is the default.

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

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

```
iccavr filename -ml
```

or:

```
iccavr filename --memory_model=large
```

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

--module_name

Sets the object module name.

SYNTAX

```
--module_name=name
```

DESCRIPTION

Normally, the internal name of the object module is the name of the source file, without a directory name or extension. Use this option to specify an object module name.

To set the object module name explicitly, use the option `--module_name=name`, for example:

```
iccavr prog --module_name=main
```

This option is particularly useful when several modules have the same filename, since the resulting duplicate module name would normally cause a linker error; for example, when the source file is a temporary file generated by a preprocessor.

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

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

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

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

--no_code_motion

Disables the code motion optimization.

SYNTAX

```
--no_code_motion
```

DESCRIPTION

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

Use `--no_code_motion` to disable code motion.

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

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

--no_cross_call

Disables the cross-call optimization.

SYNTAX

```
--no_cross_call
```

DESCRIPTION

Use this option to disable the cross-call optimization. This is highly recommended if your target processor has a hardware stack or a small internal return stack segment, RSTACK, since this option reduces the usage of RSTACK.

This optimization is performed at size optimization, level 7–9. Notice that, although it can drastically reduce the code size, this option increases the execution time.

For additional information, see *--cross_call_passes*, page 85.

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

--no_cse

Disables the common sub-expression elimination.

SYNTAX

```
--no_cse
```

DESCRIPTION

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

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

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

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

--no_inline

Disables function inlining.

SYNTAX

```
--no_inline
```

DESCRIPTION

Use `--no_inline` to disable function inlining.

Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call.

This optimization, which is performed at optimization levels 7–9, normally reduces execution time, but increases code size. The resulting code may also be difficult to debug. In certain cases, the code size will decrease when this option is used.

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

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

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

--no_rampd

Uses the `RAMPZ` register in direct address mode.

SYNTAX

```
--no_rampd
```

DESCRIPTION

Specifying this option makes the compiler use the `RAMPZ` register instead of `RAMPD`. This option corresponds to the instructions `LDS` and `STS`.

Notice that this option is only useful on processor variants with more than 64 Kbyte data (`-v4` and `-v6`).

--no_ubrof_messages

Minimizes object file size.

SYNTAX

```
--no_ubrof_messages
```

DESCRIPTION

Use this option to minimize the size of your application object file by excluding messages from the UBROF files. A file size decrease of up to 60 % can be expected. Notice that the XLINK diagnostic messages will, however, be less useful when you use this option.

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

--no_unroll

Disables loop unrolling.

SYNTAX

```
--no_unroll
```

DESCRIPTION

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

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

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

Note: Loop unrolling is permanently disabled in the AVR IAR Compiler. This option is available for compatibility with other IAR compilers.

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

--no_warnings

Disables all warnings.

SYNTAX

```
--no_warnings
```

DESCRIPTION

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

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

-o

Sets object filename.

SYNTAX

`-o` *`filename`*

DESCRIPTION

If no object code filename is specified, the compiler stores the object code in a file whose name consists of the source filename, excluding the path, plus the filename extension `r90`.

Use the `-o` option to specify a name for the output file. The filename may include a pathname. For example, to store it in the file `obj.r90` in the `mypath` directory, you would use:

```
iccavr prog -o \mypath\obj
```

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

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

--only_stdout

Uses standard output only.

SYNTAX

```
--only_stdout
```

DESCRIPTION

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

--preprocess

Directs preprocessor output to file.

SYNTAX

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

DESCRIPTION

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

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

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

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

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

-r, --debug

Generates debug information.

SYNTAX

```
--debug
-r
```

DESCRIPTION

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

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

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

--remarks

Enables remarks.

SYNTAX

```
--remarks
```

DESCRIPTION

The least severe diagnostic messages are called remarks (see *Severity levels*, page 151). A remark indicates a source code construct that may cause strange behavior in the generated code.

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

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

--root_variables

Specifies variables as `__root`.

SYNTAX

```
--root_variables
```

DESCRIPTION

Use this option to apply the `__root` extended keyword to all global and static variables. This will make sure that the variables are not removed by the IAR XLINK Linker.

Notice that the `--root_variables` option is always available, even if you do not specify the compiler option `-e`, language extensions.

This option is related to the **Code** options in the **ICCAVR** category in the IAR Embedded Workbench.

-s

Optimizes for speed.

SYNTAX

```
-s[0-9]
```

DESCRIPTION

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

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

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

The following table shows how the optimization levels are mapped:

Option modifier	*Description*
0	No optimization
1-3	Fully debuggable
4-6	Heavy optimization can make the program flow hard to follow during debug
7-9	Full optimization

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

--segment

Specifies segment name.

SYNTAX

```
--segment memory_attribute=segment_name
```

DESCRIPTION

Use this option to place all variables or functions with the memory attribute *memory_attribute* in segments with names that begin with *segment_name*.

For example, the following command places the `__near int a;` variable in the `FOO_Z` segment:

```
--segment __near=FOO
```

For a description of the memory attributes, see *Data storage*, page 112.

For a description of the segment name suffixes, see *Segment suffix*, page 59.

--silent

Specifies silent operation.

SYNTAX

```
--silent
```

DESCRIPTION

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

--strict_ansi

Specifies strict ISO/ANSI.

SYNTAX

```
--strict_ansi
```

DESCRIPTION

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

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

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

-v

Specifies the processor variant.

SYNTAX

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

DESCRIPTION

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

Command line option	*Processor variant*
`-v0`	AT90S2313 AT90S2323 AT90S2333 AT90S2343 AT90S4433
`-v1`	AT90S4414 AT90S4434 AT90S8515 AT90S8534 AT90S8535
`-v2`	Reserved for future derivatives
`-v3`	AT90mega103 AT90mega161 AT90mega603
`-v4`	Reserved for future derivatives
`-v5`	Reserved for future derivatives

Command line option	*Processor variant*
`-v6`	Reserved for future derivatives

See also *--cpu*, page 84, and *Processor*, page 11.

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

--version1_calls

Specifies the `ICCA90` calling convention.

SYNTAX

```
--version1_calls
```

DESCRIPTION

This option is provided for backward compatibility. It makes all functions and function calls use the calling convention of the A90 IAR Compiler, `ICCA90`, which is described in *ICCA90 calling convention*, page 40.

To change the calling convention of a single function, use the `__version_1` function type attribute. See *Version 1 calling convention*, page 121, for detailed information.

This option is related to the **Code** options in the **ICCAVR** category in the IAR Embedded Workbench.

--warnings_affect_exit_code

Makes warnings affect the exit code.

SYNTAX

```
--warnings_affect_exit_code
```

DESCRIPTION

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

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

--warnings_are_ errors

Makes the compiler treat all warnings as errors.

SYNTAX

```
--warnings_are_errors
```

DESCRIPTION

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

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

```
--diag_warning=Pe117
```

For additional information, see *--diag_warning*, page 88.

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

-y

Places constants and literals in initialized data segments.

SYNTAX

```
-y
```

DESCRIPTION

Use this option to override the default placement of constants and literals.

Without this option, constants and literals are placed in an external `const` segment, *`segment`*`_C`. *With* this option, constants and literals will instead be placed in the initialized *`segment`*`_I` data segments that are copied from the *`segment`*`_ID` segments by `CSTARTUP`.

Notice that `-y` is implicit in the tiny memory model.

This option can be combined with the option `--initializers_in_flash`; see page 93 for additional information.

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

-z

Optimizes for size.

SYNTAX

```
-z[0-9]
```

DESCRIPTION

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

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

The following table shows how the optimization levels are mapped:

Option modifier	*Description*
0	No optimization
1–3	Fully debuggable
4–6	Heavy optimization can make the program flow difficult to follow during debug
7–9	Full optimization

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

--zero_register

Specifies register `R15` as zero register.

SYNTAX

```
--zero_register
```

DESCRIPTION

Enabling this option will make the compiler use register `R15` as zero register, i.e. register `R15` is assumed to always contain zero.

This option can in some cases reduce the size of the generated code, especially in the large memory model.

--64bit_doubles

Forces the compiler to use the 64-bit `double` type.

SYNTAX

```
--64bit_doubles
```

DESCRIPTION

Use this option to force the compiler to use 64-bit doubles instead of 32-bit doubles which is the default. For additional information, see *Floating-point types*, page 52.

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

EXTENDED KEYWORDS

This chapter describes the non-standard keywords that support specific features of the AVR microcontroller.

Notice that the keywords and the `@` operator are only available when language extensions are enabled in the AVR IAR Compiler.

Use the `-e` compiler option to enable language extensions. See *-e*, page 89 for additional information.

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

SUMMARY OF EXTENDED KEYWORDS

The following list summarizes the extended keywords that are available to the AVR IAR Compiler:

- `__tiny`, `__near`, `__far`, and `__huge` which control the storage of variables.
- `__tinyflash`, `__flash`, `__farflash`, and `__hugeflash` place objects in code space.
- `__regvar` permanently places a variable in the specified register.
- `__io` and `__eeprom` control how objects can be accessed.
- `__generic` represents a generic pointer that can point to objects in both code and data memory.
- `__no_init` supports non-volatile memory.
- `__root` ensures that a function or variable is included in the object code even if unused.
- `__interrupt` supports interrupt functions.
- `__monitor` supports atomic execution of a function.
- `__C_task` prevents the function from pushing used registers on the stack.
- `__version_1` makes the function use the A90 IAR C Compiler, `ICCA90`, calling convention.
- `__nearfunc` and `__farfunc` which control the storage of functions.

DATA STORAGE

The data storage keywords control the storage of variables and constants, and determine how objects are accessed.

STORING DATA IN DATA MEMORY

By default the compiler places variables in a memory segment depending on which memory model is used. See *Memory model*, page 14, for detailed information. The default location can be overriden by use of the following keywords:

Keyword	*Max. object size*	*Pointer size*	*Address range*
`__tiny`	127 bytes	1 byte	`0x0-0xFF`
`__near`	32 Kbytes	2 bytes	`0x0-0xFFFF`
`__far`	32 Kbytes	3 bytes	`0x0-0xFFFFFF`
`__huge`	16 Mbytes	3 bytes	`0x0-0xFFFFFF`

Note: When the `__far` keyword is used, objects cannot cross a 64K boundary. Arithmetics will only be performed on the two lower bytes, except comparison which is always performed on the entire 24-bit address.

STORING DATA IN CODE MEMORY

The following table shows the keywords that can be used for placing data objects in code memory. Notice that the object must be declared as constants.

Keyword	*Max. object size*	*Pointer size*	*Address range*
`__tinyflash`	127 bytes	1 byte	`0x0-0xFF`
`__flash`	32 Kbytes	2 bytes	`0x0-0xFFFF`
`__farflash`	32 Kbytes	3 bytes	`0x0-0x7FFFFF`
`__hugeflash`	8 Mbytes	3 bytes	`0x0-0x7FFFFF`

Note: The `__farflash` and `__hugeflash` keywords are only available for derivatives with at least 64 Kbytes of flash memory. When the `__farflash` keyword is used, objects cannot cross a 64 Kbyte boundary. Arithmetics will only be performed on the two lower bytes, except comparison which is always performed on the entire 24-bit address.

PLACING DATA IN REGISTERS

The `__regvar` keyword is used for declaring that a *global* or *static* variable should be placed permanently in the specified register or registers. The registers `R4-R15` can be used for this purpose, provided that they have been locked with the `--lock_regs` compiler option; see page 94 for additional information. Also refer to *Register usage*, page 37.

Note: It is *not* possible to point to an object that has been declared `__regvar`. An object declared `__regvar` cannot have an initial value.

I/O AND EEPROM

The following keywords support the special I/O instructions and ports of the AVR microcontroller:

Keyword	*Max. object size*	*Comment*
`__io`	4 bytes	Implies that objects are `__no_init` and `volatile`.
`__eeprom`	127 bytes or 32 Kbytes	The size of the inbuilt EEPROM must be specified by use of the `--eeprom_size=`*N* or the `--cpu` option; see page 90 or page 84, respectively.

GENERIC POINTER

The `__generic` keyword declares a generic pointer that can point to objects in both code and data memory. The size of the generic pointer depends on which processor option is used:

Processor option	*Generic pointer size*
`-v0`, `-v1`	2 bytes
`-v2`, `-v3`, `-v4`, `v5`, `v6`	3 bytes

It is not possible to place objects in a generic memory area, only to point to it.

When using generic pointers, make sure that objects that have been declared `__far` and `__huge` are located in the range `0x0–0x7FFFF`. Objects may still be placed in the entire data address space, but a generic pointer cannot point to objects in the upper half of the data address space.

Note: The `__generic` keyword cannot be used with the `#pragma type_attribute` directive for a pointer. For more information, see *Pointers*, page 127.

SYNTAX

The keywords follow the same syntax as the type qualifiers `const` and `volatile`. The following declarations all place the variable `i` and `j` in a `NEAR` memory segment:

```
__near int i, j;
int __near i, j;
```

Notice that the keyword affects all the identifiers.

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

```
char __huge * 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 `NEAR` memory. Like `p`, `p2` points to a character in `__huge` memory.

```
__near char __huge *p2;
```

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

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

and

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

#PRAGMA DIRECTIVES

It is possible to avoid the non-standard keywords in declarations by using `#pragma` directives. The `#pragma type_attribute` controls the storage of variables but does not affect their type.

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

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

It is important to notice that the #pragma type_attribute directive affects *only* the declaration of the identifier that follows immediately after the directive. The following two declarations are therefore equivalent:

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

and

```
short __near c;
short d;
```

That is, only c is affected by the keyword.

It is, for obvious reasons, impossible to place a variable in more than one memory segment. It is therefore not feasible to specify more than one of the keywords in a declaration. Multiple keywords result in a diagnostic message.

Note: Direct use of keywords overrides a keyword that is specified in a #pragma directive.

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

Pointers

The #pragma type_attribute can also be used for declaring pointers. The following example will place the variable in NEAR memory. The variable will be a pointer to FAR memory:

```
#pragma type_attribute=__near
int __far *c;
```

__NO_INIT

The __no_init keyword is used for placing a variable in a non-volatile memory segment and for suppressing initialization at startup.

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

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

#pragma object_attribute can also be used. The following declaration is equivalent with 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 on a function or variable ensures that, if the module containing the function or variable is included in linked output, the function or variable is also included, whether or not it is referenced by the rest of the program.

Normally, only the part of the run-time library calling main and any interrupt vectors are root. This attribute can be used for making other functions and/or variables behave the same way.

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

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

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

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

Note: The __root keyword cannot be used in typedefs.

ABSOLUTE LOCATION

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

- The @ operator followed by a constant-expression.

  The following declaration locates PIND at address 10h:

  ```
  __no_init __io char PIND @ 0x10;
  ```

The following declaration locates `i` at address 20 with the value 10:

```
const int i@20=10;
```

- The `#pragma location` directive followed by a constant-expression.

  The following declarations are equivalent to the previous declarations:

  ```
  #pragma location=0x10
  __no_init __io char PIND;

  #pragma location=20
  const int i=10;
  ```

SEGMENT PLACEMENT

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

- The `@` operator followed by a string such as "`constseg`".

  The following example declares the constant `K` to reside in the `constseg` segment:

  ```
  const int K @ "constseg"=10;
  ```

  The following declaration places `i` in the `BRAVO` segment:

  ```
  __no_init int i @ "BRAVO";
  ```

- The `#pragma location` directive followed by a string.

  The following declaration is equivalent to the previous one:

  ```
  #pragma location="BRAVO"
  __no_init int i;
  ```

FUNCTION EXECUTION

The following keywords control the execution of a function:

- `__interrupt`, which specifies interrupt functions. The `#pragma vector` directive can be used for specifying the `interrupt` vector. An interrupt function must be of type `void` and must not have any parameters.
- `__monitor`, which specifies a monitor function.
- `__C_task`, which declares a function that does not save registers. It is normally used for `main`.

- `__root` ensures that a function or variable is included in the object code even if unused.

The keywords are specified before the return type:

```
__interrupt void foo(void);
```

It is possible to avoid the non-standard keywords in declarations by using `#pragma type_attribute` directive.

The `#pragma type_attribute` controls the calling conventions of functions. See the chapter *#pragma directives* for a complete description of the `#pragma` directives.

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

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

INTERRUPT FUNCTIONS

The following example declares an interrupt function with interrupt vector with offset `0x2` in the `INTVEC` segment:

```
#pragma vector=0x2
__interrupt void my_interrupt_handler(void);
```

An interrupt function must be of type void and it cannot take any parameters.

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

It is possible to define an interrupt function without a vector but you must then add the interrupt vector to the interrupt vector table, `INTVEC`.

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

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

MONITOR FUNCTIONS

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.

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 141, *__enable_interrupt*, page 141, *__restore_interrupt*, page 143, and *__save_interrupt*, page 143.

A function declared with monitor is equivalent to any other function in all other respects.

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 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 sempahore.

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

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

C-TASK FUNCTIONS

The `__C_task` keyword affects the definition of a function. It is typically used for `main`.

By default functions save the contents of used non-scratch registers on stack upon entry, and restore them at exit. Functions declared as `__C_task` do not save any registers, and therefore require less stack space. Such functions should only be called from assembly language routines.

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

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

```
__C_task void my_handler(void);
```

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

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

Note: `__C_task` cannot be used in type definitions (`typedefs`).

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

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

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

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

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

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

__ROOT

The `__root` attribute can be used on either a function or a variable. It ensures that the function or variable is included in the object code even if unused.

For a description of this attribute, see *__root*, page 116.

FUNCTION CALLING CONVENTION

The following keywords control the calling convention of a function:

- `__version_1`
- `__intrinsic`, which declares a predefined in-line or library function.

VERSION 1 CALLING CONVENTION

The `__version_1` keyword is available for backward compatibility. It makes a function use the calling convention of the A90 IAR C Compiler instead of the default calling convention, both which are described in *C calling convention*, page 37, and *Embedded C++*, page 47.

INTRINSIC

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

FUNCTION STORAGE

The following keywords control in which memory range a function is placed:

Keyword	*Address range*	*Pointer size*	*Description*
`__nearfunc`	`0x0-0x1FFFE`	2 bytes	Can be called from the entire memory area, but must reside in the first 128 Kbytes of the code memory.
`__farfunc`	`0x0-0x7FFFFE`	3 bytes	No restrictions on code placement.

Notice that pointers with function memory attributes have restrictions in implicit and explicit casts when casting between pointers and also when casting between pointers and integer types.

It is possible to call a `__nearfunc` function from a `__farfunc` function and vice versa. Only the size of the function pointer is affected.

A function can be placed in a specific segment also by the use of either:

- The `@` operator followed by "*segment*"
- The `#pragma location` directive followed by "*segment*"

For example:

```
void f() @ "segment";
void g() @ "segment"{}
#pragma location="segment"
void h();
```

EMBEDDED C++

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

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

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

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

```
class Y {
public:
  int len;
  <Mattr1> char buf[1000];  // Error!!!
};

<Mattr1> Y myBuf;  // This is OK
```

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

```
class Z {
  static <Mattr1> int numZ;  // OK since numZ is static
};
```

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

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

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

#PRAGMA DIRECTIVES

This chapter describes the #pragma directives of the AVR IAR Compiler.

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

All #pragma directives should be entered like:

```
#pragma pragmaname=pragmavalue
```

or

```
#pragma pragmaname = pragmavalue
```

Note: The #pragma directives `warnings`, `codeseg`, `baseaddr`, `function`, and `alignment`, which were used in the A90 IAR Compiler, are recognized and will give a diagnostic message but will not work. It is important to be aware of this if you need to port existing code that contains any of those old-style #pragma directives.

TYPE ATTRIBUTE

The directive #pragma `type_attribute` affects the declaration of the identifier, the next variable or the next function, that follows immediately after the #pragma. It only affects the variable, not its type. This means that the `__generic` keyword cannot be used with #pragma `type_attribute`.

In the following example, `myBuffer` is placed in a `NEAR` segment, 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.

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

In the small memory model, the default pointer is `__near`. In the following example, the pointer is located in `tiny` memory, pointing at `__near`:

```
#pragma type_attribute=__tiny
int * pointer;
```

VARIABLES

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

- One of `__tiny`, `__near`, `__far`, and `__huge`, which control how variables are located in memory.
- `__io`, which allows objects to be accessed by use of the special I/O instructions in the AVR microcontroller.
- `__eeprom`, which places objects in the internal EEPROM and make them accessible through special I/O ports in the AVR microcontroller.
- The `__regvar` keyword can be used with the `#pragma type_attribute` for a *global* or *static* variable. It specifies that the variable should be permanently located in the specified register or registers. For additional information, see *Register usage*, page 37, and *--lock_regs*, page 94.

CONSTANTS

The following keywords can be used with the `#pragma type_attribute` for a `constant`-declared object:

- One of `__tinyflash`, `__flash`, `__farflash`, and `__hugeflash`.

FUNCTIONS

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

- `__interrupt`, which specifies interrupt functions. The `#pragma vector` directive can be used for specifying the `interrupt` vector.
- `__monitor`, which specifies a monitor function.
- `__C_task`, which prevents the function from pushing used registers on the stack. A function that has been declared `__C_task` can only be called from assembly language.

- __version_1, which specifies that the calling convention of the A90 IAR C Compiler, ICCA90, should be used instead of the default calling convention, both which are described in *C calling convention*, page 37.

POINTERS

Access to a __generic pointer is implemented with a function call to an assembler-written library routine. Since this type of access is slow and generates a lot of code, __generic pointer should be avoided when possible. The most significant bit in the pointer indicates if data in code or data memory is being referenced.

Note: The __generic keyword cannot be used with the #pragma type_attribute directive for a *pointer* since it determines the location, not the type of a variable or function. It provides access to data regardless of whether it is located in code or data memory.

For additional information, see *Generic pointer*, page 113.

MEMORY

The #pragma memory directive directs variables to a specified memory segment. The following keywords can be used with #pragma memory:

```
__tiny
__near
__far
__huge
__tinyflash
__flash
__farflash
__hugeflash
```

For example:

```
#pragma memory=__huge
```

The #pragma memory directive is active until it is explicitly turned off with #pragma memory=default.

#pragma memory overrides both #pragma constseg and #pragma dataseg.

Note: #pragma memory is available for backward compatibility reasons. We recommend you to use #pragma type_attribute, the @ operator, or #pragma location instead.

OBJECT ATTRIBUTE

The directive #pragma object_attribute affects the declaration of the identifier that follows immediately after the directive.

The following keywords can be used with the #pragma object_attribute for a *variable*:

- __no_init, which places a variable in a non-volatile memory segment and for suppressing initialization at startup.
- __root, which ensures that the variable is included in the object code even if unused.

The following keywords can be used with the #pragma object_attribute for a *function*:

- __C_task, which declares a function that does not save registers.
- __root, which ensures that the function is included in the object code even if unused.

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;
```

DATASEG

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

```
#pragma dataseg=MY_SEGMENT
__no_init char myBuffer[1000];
#pragma dataseg=default
```

The segment name must not be a predefined segment, see the chapter *Segments* 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=__huge MyOtherSeg
```

All variables in `MyOtherSeg` will be accessed using `__huge` addressing.

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 *Segments* for more information.

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

```
#pragma constseg=__huge MyOtherSeg
```

All variables in `MyOtherSeg` will be accessed using `__huge` addressing.

LOCATION

The `#pragma location` directive specifices the location—the absolute address—of the variable whose declaration follows the `#pragma` directive. For example:

```
#pragma location=0x10
char PIND; // PIND is located at address 10h
```

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*, page 116, *Segment placement*, page 117, and *Function storage*, page 122.

VECTOR

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

```
#pragma vector=0x02
__interrupt void my_handler(void);
```

DIAGNOSTICS

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

DIAG_REMARK

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

Changes the severity level to remark for the specified diagnostics. For example:

```
#pragma diag_remark=Pe177
```

DIAG_WARNING

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

Changes the severity level to warning for the specified diagnostics. For example:

```
#pragma diag_warning=Pe826
```

DIAG_ERROR

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

Changes the severity level to error for the specified diagnostics. For example:

```
#pragma diag_error=Pe117
```

DIAG_DEFAULT

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

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

```
#pragma diag_default=Pe117
```

DIAG_SUPPRESS

Syntax: `#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.

LANGUAGE

The `#pragma language` directive is used for turning on the IAR language extensions or for using the language settings specified on the command line.

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

extended	Turns on the IAR extensions and turns off the `--strict_ansi` command line compiler option.
default	Uses the settings specified on the command line.

OPTIMIZE

The `#pragma optimize` directive is used for decreasing the optimization level or for turning off some specific optimizations.

Syntax: `#pragma optimize=token token token`

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

`s`	Optimize for speed
`z`	Optimize for size
`0–9`	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.

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, this `#pragma` directive is ignored.

PACK

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

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

n	Packing alignment, one of: 1, 2, 4, 8 or 16
name	Pushed or popped alignment label.

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

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

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

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

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

Note: In the AVR IAR Compiler, alignment is always 1 and this `#pragma` directive has no effect. It is available in order to maintain compatibility with other IAR Systems compilers.

BITFIELDS

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

Syntax: #pragma bitfields=[reversed|default]

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

PREDEFINED SYMBOLS

This chapter gives reference information about the predefined preprocessor symbols that are supported in the AVR IAR Compiler.

__CPU__

Processor identifier.

SYNTAX

```
__CPU__
```

DESCRIPTION

Expands to a number which corresponds to the processor option `--cpu` or `-v`*n* in use.

__DATE__

Current date.

SYNTAX

```
__DATE__
```

DESCRIPTION

Expands to the date of compilation is returned in the form `Mmm dd yyyy`.

__cplusplus

C++ identifier.

SYNTAX

```
__cplusplus
```

DESCRIPTION

Expands to the number 1 when the compiler runs in Embedded C++ mode. When the compiler runs in ANSI C mode, the symbol is undefined.

This symbol can be used with `#ifdef` to detect that the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and Embedded C++ code.

__embedded_cplusplus

Embedded C++ identifier.

SYNTAX

```
__embedded_cplusplus
```

DESCRIPTION

Expands to the number 1 when the compiler runs in Embedded C++ mode. When the compiler runs in ANSI C mode, the symbol is undefined. This symbol can be used with `#ifdef` to detect that the compiler accepts only the Embedded C++ subset of the C++ language.

__FILE__

Current source filename.

SYNTAX

```
__FILE__
```

DESCRIPTION

Expands to the name of the file currently being compiled.

__IAR_SYSTEMS_ICC__

IAR Compiler identifier.

SYNTAX

```
__IAR_SYSTEMS_ICC__
```

DESCRIPTION

Expands to a number that identifies the IAR Compiler platform. The current identifier is 4. Notice 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 an IAR Compiler.

__ICCAVR__

AVR IAR Compiler identifier.

SYNTAX

```
__ICCAVR__
```

DESCRIPTION

Expands to the number 1 when the code is compiled with the AVR IAR Compiler.

__LINE__

Current source line number.

SYNTAX

```
__LINE__
```

DESCRIPTION

Expands to the current line number of the file currently being compiled.

__MEMORY_MODEL__

AVR IAR Compiler memory model identifier.

SYNTAX

```
__MEMORY_MODEL__
```

DESCRIPTION

Expands to a value reflecting the selected memory model according to the following table:

Value	*Memory model*
1	Tiny
2	Small
3	Large
4	Generic

__STDC__

ISO/ANSI standard C identifier.

SYNTAX

```
__STDC__
```

DESCRIPTION

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

__STDC_VERSION__

ISO/ANSI Standard C and version identifier.

SYNTAX

```
__STDC_VERSION__
```

DESCRIPTION

Expands to the number 1994092.

Note: This predefined symbol does not apply to the EC++ version of the product.

__TID__

Target identifier for the AVR IAR Compiler.

SYNTAX

```
__TID__
```

DESCRIPTION

Expands to the target identifier containing the following parts:

- A number unique for each IAR Compiler (i.e. unique for each target).
- The value of the `--cpu` or `-v` option. For details, see *Processor*, page 11.
- The value corresponding to the `--memory_model` option.

For the AVR microcontroller, the target identifier is `0x5A`.

The `__TID__`value is constructed as:

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

You can extract the values as follows:

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

```
m = __TID__ & 0x0F;         /* memory model */
```

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

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

__TIME__

Current time.

SYNTAX

```
__TIME__
```

DESCRIPTION

Expands to the time of compilation in the form `hh:mm:ss`.

__VER__

Compiler version number.

SYNTAX

```
__VER__
```

DESCRIPTION

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
#message "Compiler version 3.34"
#endif
```

INTRINSIC FUNCTIONS

This chapter gives reference information about the intrinsic functions.

To use intrinsic functions in an application, include the header file `inavr.h`.

Notice that the intrinsic function names start with double underscores, for example:

```
__enable_interrupt
```

__delay_cycles

SYNTAX

```
__delay_cycles(unsigned long int);
```

DESCRIPTION

Use this intrinsic to make the compiler generate code that takes the given amount of cycles to perform, i.e. it inserts a time delay that lasts the specified number of cycles.

Note: The specified value must be a constant integer expression and not an expression that is evaluated at run time.

__disable_interrupt

SYNTAX

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

DESCRIPTION

Inserts a disable interrupt instruction, `DI`.

__enable_interrupt

SYNTAX

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

DESCRIPTION

Inserts an enable interrupt instruction, `EI`.

__extended_load_program_memory

SYNTAX

```
unsigned char __extended_load_program_memory(unsigned
char __farflash *)
```

DESCRIPTION

Returns one byte from code memory.

Use this intrinsic function to access constant data in code memory.

__insert_opcode

SYNTAX

```
void __insert_opcode(unsigned short);
```

DESCRIPTION

Inserts a `DW unsigned` directive.

__load_program_memory

SYNTAX

```
unsigned char __load_program_memory(unsigned char __flash
*);
```

DESCRIPTION

Returns one byte from code memory. The constants must be placed within the first 64 Kbytes of memory.

__no_operation

SYNTAX

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

DESCRIPTION

Inserts a `NOP` instruction.

__require

SYNTAX

```
__require(void *);
```

DESCRIPTION

Emits a `REQUIRE` statement on the given symbol.

__restore_interrupt

SYNTAX

```
void __restore_interrupt(unsigned char oldState)
```

DESCRIPTION

This intrinsic will restore the interrupt flag to the state it had when `__save_interrupt` was called.

Note: The value of *`oldState`* must be the result of a call to the `__save_interrupt` intrinsic function.

__save_interrupt

SYNTAX

```
unsigned char __save_interrupt(void)
```

DESCRIPTION

This intrinsic will save the state of the interrupt flag in the byte returned. This value can then be used for restoring the state of the interrupt flag with the `__restore_interrupt` intrinsic.

EXAMPLE

```
unsigned char oldState;

oldState = __save_interrupt();
__disable_interrupt();

/* Critical section goes here */

__restore_interrupt(oldState);
```

__segment_begin

SYNTAX

```
__segment_begin(const char *);
```

DESCRIPTION

Returns the address of the first byte of the named segment. The argument be a constant string literal.

__segment_end

SYNTAX

```
__segment_end(const char *);
```

DESCRIPTION

Returns the address of the first byte *after* the named segment. The argument must be a constant string literal.

__sleep

SYNTAX

```
void __sleep(void);
```

DESCRIPTION

Inserts a sleep instruction, `SLEEP`. To use this intrinsic function, make sure that the instruction has been enabled in the `MCUCR` register.

__watchdog_reset

SYNTAX

```
void __watchdog_reset(void);
```

DESCRIPTION

Inserts a watchdog reset instruction.

LIBRARY FUNCTIONS

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

INTRODUCTION

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

- Adherence to a free-standing implementation of the ISO standard for the programming language C. For additional information, see the chapter *Implementation-defined behavior*.
- Standard C library definitions, for user programs.
- Embedded C++ library definitions, for user programs.
- `CSTARTUP`, the single program module containing the start-up code. It is described in *Initialization*, page 26.
- Run-time support libraries; for example, low-level floating-point routines.

LIBRARY OBJECT FILES

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

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

HEADER FILES

The user program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into a number of different header files each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do this can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

VIEWING THE C OR EMBEDDED C++ LIBRARY DOCUMENTATION

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

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

LIBRARY DEFINITIONS SUMMARY

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

EMBEDDED C++

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

Header file	*Usage*
`complex`	Defining a class that supports complex arithmetic
`exception`	Defining several functions that control exception handling
`fstream`	Defining several I/O streams classes that manipulate external files
`iomanip`	Declaring several I/O streams manipulators that take an argument
`ios`	Defining the class that serves as the base for many I/O streams classes
`iosfwd`	Declaring several I/O streams classes before they are necessarily defined

Header file	*Usage*
`iostream`	Declaring the I/O streams objects that manipulate the standard streams
`istream`	Defining the class that performs extractions
`new`	Declaring several functions that allocate and free storage
`ostream`	Defining the class that performs insertions
`sstream`	Defining several I/O streams classes that manipulate string containers
`stdexcept`	Defining several classes useful for reporting exceptions
`streambuf`	Defining classes that buffer I/O streams operations
`string`	Defining a class that implements a string container
`strstream`	Defining several I/O streams classes that manipulate in-memory character sequences

USING STANDARD C LIBRARIES IN EC++

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

The following table shows the new headers:

Header file	*Usage*
`cassert`	Enforcing assertions when functions execute
`cctype`	Classifying characters
`cerrno`	Testing error codes reported by library functions
`cfloat`	Testing floating-point type properties
`climits`	Testing integer type properties
`clocale`	Adapting to different cultural conventions
`cmath`	Computing common mathematical functions
`csetjmp`	Executing non-local goto statements

Header file	*Usage*
`csignal`	Controlling various exceptional conditions
`cstdarg`	Accessing a varying number of arguments
`cstddef`	Defining several useful types and macros
`cstdio`	Performing input and output
`cstdlib`	Performing a variety of operations
`cstring`	Manipulating several kinds of strings
`ctime`	Converting between various time and date formats

STANDARD C

The following table shows the traditional standard C library headers:

Header file	*Usage*
`assert.h`	Enforcing assertions when functions execute
`ctype.h`	Classifying characters
`errno.h`	Testing error codes reported by library functions
`float.h`	Testing floating-point type properties
`iso646.h`	Using Amendment 1—`iso646.h` standard header
`limits.h`	Testing integer type properties
`locale.h`	Adapting to different cultural conventions
`math.h`	Computing common mathematical functions
`setjmp.h`	Executing non-local goto statements
`signal.h`	Controlling various exceptional conditions
`stdarg.h`	Accessing a varying number of arguments
`stddef.h`	Defining several useful types and macros
`stdio.h`	Performing input and output
`stdlib.h`	Performing a variety of operations
`string.h`	Manipulating several kinds of strings
`time.h`	Converting between various time and date formats

Header file	*Usage*
`wchar.h`	Support for wide characters
`wctype.h`	Classifying wide characters

COMPATIBILITY WITH STANDARD C++

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

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

DIAGNOSTICS

A normal diagnostic from the compiler is produced in the form:

filename,*linenumber level*[*tag*]: *message*

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

SEVERITY LEVELS

The diagnostics are divided into different levels of severity:

Remark
A diagnostic that is produced when the compiler finds a source code construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued but can be enabled, see *Enables remarks.*, page 102.

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 100.

Error
A diagnostic that is produced when the compiler has found a construct which clearly violates the C or Embedded 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 82, for a description of the compiler options that are available for setting severity levels.

See *Diagnostics*, page 130, 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 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.
- The version number of the compiler. To display it at sign-on, run the compiler, `iccavr`, without parameters.

MESSAGES

This section lists the AVR-specific warning and error messages.

AVR-SPECIFIC WARNING MESSAGES

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

[Ta006] Interrupt function has no assigned vector

AVR-SPECIFIC ERROR MESSAGES

In addition to the general error messages, the AVR IAR Compiler may generate the error messages in the following list:

[Ta001] `--no_rampd` **cannot be used with processor option** `-v`*option*

[Ta002] **The** *model* **memory model is not allowed with processor option** `-v`*option*

[Ta003] **Processor option** `-v`*option* **is not allowed in Base-line version**

[Ta004] **Cannot combine** `__C_task` **with** *function-attribute*

[Ta005] **Interrupt functions cannot return a result**

[Ta007] **An** `__io` **declared variable must be located**

[Ta008] **Location out of range for an** `__io` **declared variable**

[Ta009] **Argument to** `__insert_opcode` **is a non-constant expression**

[Ta010] **Argument to** `__insert_opcode` **is not in the range 0–65536**

[Ta011] **Registers must be locked with** `--lock_regs` **before** `__regvar` **can be used**

[Ta012] **A** `__regvar` **declared variable must be located**

[Ta013] **Illegal location for a** `__regvar` **declared variable**

[Ta014] **A** `__regvar` **declared variable must have global scope**

[Ta015] **Cannot write to flash memory**

[Ta016] **Illegal alignment of** `__regvar` **declared variable**

[Ta017] **Segment name must be a string literal**

[Ta018] **Expected a literal symbol and not an expression**

[Ta019] Unknown attribute *string*

[Ta020] Malformed `--segment` **option:** *option***. Correct format is** *memory-attribute***=***segment-name*

[Ta021] The function *function* **generated more code (***n* **bytes) than is available in the target (***n* **bytes)**

[Ta022] Interrupt functions cannot take any parameters

[Ta023]
This message does not exist.

[Ta024] __io declared object partially out of range 0 .. 63

AVR-SPECIFIC FATAL ERROR MESSAGES

In addition to the general error messages, the AVR IAR Compiler may generate the following fatal error message:

[Ta000] General target error *string*

PART 3: MIGRATION AND PORTABILITY

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

- *Migrating to the AVR IAR Compiler*
- *Implementation-defined behavior*
- *IAR C extensions.*

MIGRATING TO THE AVR IAR COMPILER

C source code that was originally written for the A90 IAR Compiler can be used also with the AVR IAR Compiler, although some modifications may be required.

This chapter contains information that is useful when migrating from the A90 IAR Compiler (ICCA90) to the AVR IAR Compiler (ICCAVR). It briefly describes both differences and similarities between the products.

INTRODUCTION

The main difference between ICCA90 and ICCAVR is that the latter is based on new compiler technology, which makes it possible to enhance your application code in a way that previously was not possible.

The most obvious difference is that with the new compiler technology, support for Embedded C++ has become available. Other main advantages include a new global optimizer, which improves the efficiency of the generated code. The consistency of the compiler is also improved due to the new technology.

Moreover, the new compiler technology allows you to write source code that is easily portable since it adheres more strictly to the ISO/ANSI standard; for example, it is possible to use `#pragma` directives instead of extended keywords for defining special function registers (SFRs).

Also the checking of data types adheres more strictly to the ISO/ANSI standard in ICCAVR than in products using a previous compiler technology. You have the opportunity to identify and correct problems in the code, which improves the quality of the object code. Therefore, it is important to be aware of the fact that code written for the ICCA90 may generate warnings or errors in ICCAVR.

THE MIGRATION PROCESS

Use the file `comp_a90.h` during the migration process. This file, which is provided with the product, contains translation macros that facilitate the migration.

In short, the process of migrating from ICCA90 to ICCAVR involves the following steps:

1. Modify the pointers and doubles in the source code. For a description of the ICCAVR data types, see *Data types*, page 51.
2. Replace ICCA90 extended keywords in the source code with ICCAVR keywords.
3. Replace ICCA90 `#pragma` directives with ICCAVR directives. Notice that the behavior differs between the two products; see page 161 for detailed information.
4. Replace ICCA90 intrinsics with ICCAVR intrinsics.
5. Compile the code using appropriate ICCAVR compiler options.
6. Replace ICCA90 segments with ICCAVR segments in the linker command file. See *Linker command file*, page 17, for detailed information.
7. Link the code and run the project in the IAR C-SPY Debugger.

The following sections describe the differences between ICCA90 and ICCAVR in detail.

EXTENDED KEYWORDS

The set of language extensions has changed in ICCAVR. Some extensions have been added, some extensions have been removed, and for some of them the syntax has changed. There is also a rare case where an extension has a different interpretation if `typedefs` are used. This is described in the following section.

In ICCAVR, all extended keywords except `asm` start with two underscores, for example `__near`.

STORAGE MODIFIERS

Both ICCA90 and ICCAVR allow keywords that specify memory location. Each of these keywords can be used either as a placement attribute for an object, or as a pointer type attribute denoting a pointer that can point to the specified memory.

When the keywords are used directly in the source code, they behave in a similar way in ICCA90 and ICCAVR. The usage of type definitions and extended keywords is, however, more strict in ICCAVR than in ICCA90.

Products based on the previous compiler front end behave unexpectedly in some cases:

```
typedef int near NINT;
NINT a,b;
NINT huge c;          /* Illegal */
NINT *p;              /* p stored in near memory, points to
                      default memory attribute */
```

The first variable declaration works as expected, that is `a` and `b` are located in near memory. The declaration of `c` is however illegal, except when near is the default memory, in which case there is no need for an extended keyword in the `typedef`.

In the last declaration, the `near` keyword of the `typedef` affects the location of the pointer variable `p`, not the pointer type. The pointer type is the default, which is given by the memory model.

The corresponding example for ICCAVR is:

```
typedef int __near NINT;
NINT a,b;
NINT __huge c;        /* c stored in huge memory --
                      override attribute in typedef */
NINT *p;              /* p stored in default memory, points
                      to near memory */
```

The declarations of `c` and `p` differ. The `__huge` keyword in the declaration of `c` will always compile. It overrides the keyword of the `typedef`. In the last declaration the `__near` keyword of the `typedef` affects the type of the pointer. It is thus a `__near` pointer to `int`. The location of the variable `p` is however not affected.

__NO_INIT

In ICCA90 the keyword `no_init` is used for specifying that an object is not initialized. In ICCAVR `__no_init` can be used together with a keyword specifying any memory location, for example:

```
__near __no_init char buffer [1000];
```

__INTERRUPT

In ICCA90, a vector can be attached to an `interrupt` function with the `#pragma` directive `function` or directly in the source code, for example:

```
interrupt [8] void f(void);
```

In ICCAVR a vector can be attached to an `__interrupt` function with the `#pragma` directive `vector`, for example:

```
#pragma vector=8
__interrupt void f(void);
```

__MONITOR

In some products using the previous generation of compiler technology, the keyword `monitor` specifies not only the type attribute setting but also the memory location. In ICCAVR `__monitor` is a type attribute only.

SFR

In ICCA90 the keywords `sfrb` and `sfrw` denote an object of byte or word size residing in the Special Function Register (SFR) memory area for the chip, and having a `volatile` type. The SFR is always located at an absolute address. For example:

```
sfr PORT=0x10;
```

In ICCAVR the keywords `sfrb` and `sfrw` are not available. Instead you have the ability to:

- Place any object into any memory, by using a memory attribute; for example:

  ```
  __io int b;
  ```

- Locate any object at an absolute address by by using the `#pragma` directive `location` or by using the locator operator `@`; for example:

  ```
  long PORT @ 100;
  ```

- Use the `volatile` attribute on any type, for example:

  ```
  volatile __io char PORT@100;
  ```

See the chapter *Extended keywords* for complete information about the extended keywords available in ICCAVR.

#PRAGMA DIRECTIVES

ICCA90 and ICCAVR have different sets of #pragma directives for specifying attributes, and they also behave differently:

- In ICCA90 the #pragma directives change the default attribute to use for declared objects; they do not have an effect on pointer types. These directives are #pragma memory that specifies the default location of data objects, and #pragma function that specifies the default location of functions.
- In ICCAVR the #pragma directives type_attribute and object_attribute change the next declared object or typedef.

The rules for overriding a memory attribute differ between ICCA90 and ICCAVR. However, both give the highest priority to memory attribute keywords in the actual declaration, and the lowest priority to the specific segment placement #pragma directives.

The following ICCA90 #pragma directives have been removed in ICCAVR:

```
codeseg
function
warnings
```

These are recognized and will give a diagnostic message but will not work in ICCAVR.

Note: Instead of the #pragma codeseg directive, in ICCAVR the #pragma location directive or the @ operator can be used for specifying an absolute location.

The following table shows the mapping of #pragma directives:

ICCA90 #pragma directive	*ICCAVR #pragma directive*
#pragma function=interrupt	#pragma type_attribute=__interrupt #pragma vector=*long_word* offset
#pragma function=C_task	#pragma object_attribute=__c_task
#pragma function=interrupt	#pragma type_attribute=__interrupt
#pragma function=monitor	#pragma type_attribute=__monitor
#pragma memory=constseg	#pragma constseg, #pragma location
#pragma memory=dataseg	#pragma dataseg, #pragma location

ICCA90 #pragma directive	*ICCAVR #pragma directive*
`#pragma memory=far`	`#pragma type_attribute=__far,` `#pragma location`
`#pragma memory=flash`	`#pragma type_attribute=__flash,` `#pragma location`
`#pragma memory=huge`	`#pragma type_attribute=__huge`
`#pragma memory=near`	`#pragma type_attribute=__near`
`#pragma memory=no_init`	`#pragma object_attribute=__no_init`
`#pragma memory=tiny`	`#pragma type_attribute=__tiny`

It is important to notice that the ICCAVR directives `#pragma type_attribute`, `#pragma object_attribute`, and `#pragma vector` affect only the *first* of the declarations that follow after the directive. In the following example `x` is affected, but `z` and `y` are not affected by the directive:

```
#pragma object_attribute==__no_init
int x,z;
int y;
```

The ICCAVR directives `#pragma constseg` and `#pragma dataseg` are active until they are explicitly turned off with the directive `#pragma constseg=default` and `#pragma dataseg=default`, respectively. For example:

```
#pragma constseg=myseg
__no_init f;
#pragma constseg=default
```

The ICCAVR directive `#pragma memory=__xxxx` is active until it is explicitly turned off with the `#pragma memory=default` directive, for example:

```
#pragma memory=__near
int x,y,z;
#pragma memory=default
int myfunc()
```

The following `#pragma` directives are identical in ICCA90 and ICCAVR:

```
#pragma language=extended
#pragma language=default
```

The following #pragma directives have been *added* in ICCAVR:

```
#pragma constseg
#pragma dataseg
#pragma diag_default
#pragma diag_error
#pragma diag_remark
#pragma diag_suppress
#pragma diag_warning
#pragma location
#pragma object_attribute,
#pragma optimize
#pragma pack
#pragma type_attribute,
#pragma vector
```

Specific segment placement
In ICCA90 the #pragma memory directive supports a syntax enabling subsequent data objects that match certain criterias to end up in a specified segment. Each object found after the invocation of a segment placement directive will be placed in the segment, provided that it does not have a memory attribute placement and that it has the correct constant attribute. For constseg it must be a constant, while for dataseg they must not be declared const.

In ICCAVR, the directive #pragma location and the @ operator are available for this purpose.

See the chapter *#pragma directives* for detailed information about the #pragma directives available in ICCAVR.

PREDEFINED SYMBOLS

In both ICCA90 and ICCAVR, all predefined symbols start with two underscores, for example __IAR_SYSTEMS_ICC__. Notice however that in ICCAVR all predefined symbols also end with two underscores.

See the chapter *Predefined symbols* for complete information about the predefined symbols available in ICCAVR.

INTRINSIC FUNCTIONS

In ICCAVR, the intrinsic functions start with two underscores, for example `__enable_interrupt`.

The ICCA90 intrinsic functions `_args$` and `_argt$` are not available in ICCAVR. Other ICCA90 intrinsic functions have equivalents with other names in ICCAVR.

See the chapter *Intrinsic functions* for complete information about the intrinsic functions available in ICCAVR.

COMPILER OPTIONS

COMMAND LINE SYNTAX

The command line options in ICCAVR follow two different syntax styles:

- Long option names containing one or more words prefixed with two dashes and sometimes followed by an equal sign and a modifier, for example `--strict_ansi` and `--cpu=2343`. This is the preferred style in ICCAVR.
- Short option names consisting of a single letter prefixed with a single dash and sometimes followed by a modifier, for example `-r` or `-mt`. This style is available in ICCAVR mainly for backward compatibility.

Some options appear in one style only, other options appear in both styles.

Removed ICCA90 options
The following table shows the ICCA90 command line options that have been removed in ICCAVR:

ICCA90 option	*Description*
`-C`	Nested comments
`-F`	Form-feed after each function
`-f`*`filename`*	Extend the command line
`-G`	Open standard input as source. Replaced by `-` (dash) as source file in ICCAVR.
`-g`	Global strict type check. In ICCAVR, global strict type checking is always enabled.
`-g0`	No type information in object code

ICCA90 option	*Description*
`-K`	`'//'` comments. In ICCAVR, `'//'` comments are allowed unless the option `--strict_ansi` is used.
`-O`*`prefix`*	Set object filename prefix. In ICCAVR, use `-o` *`filename`* instead.
`-P`	Generate PROMable code. This functionality is always enabled in ICCAVR.
`-p`*`nn`*	Lines/page
`-T`	Active lines only
`-t`	Tab spacing
`-U`*`symb`*	Undefine symbol
`-X`	Explain C declarations
`-x[DFT2]`	Cross-reference
`-y`	Writable strings

Note: Instead of the command line option `-f`, the following methods may be used, depending on your operating system, for extending the command line:

- Use a command file to add the options, for example, a `bat` file.
- Use the environment variables for flags, for example `QCCAVR`.
- Define your own variables to be used on the command line; for example, in Windows 95/98 or NT:

```
set F=--frame_pointer -e
ICCAVR %F% -z9 foo.c
```

Identical options

The following table shows the command line options that are *identical* in ICCA90 and ICCAVR:

Option	*Comment*
`-D`*`symb`*`=`*`value`*	Define symbols
`-e`	Language extensions

Option	*Comment*
`-I`	Include paths. (Syntax is more free in ICCAVR.)
`-o` *`filename`*	Set object filename
`-s[0-9]`	Optimize for speed
`-z[0-9]`	Optimize for size

Renamed or modified ICCA90 options
The following ICCA90 command line options have been *renamed* and/or *modified*:

ICCA90 option	*ICCAVR option*	*Description*
`-A` `-a` *`filename`*	`-la .` `-la` *`filename`*	Assembly output. See *Filenames*, page 167.
`-b`	`--library_module`	Make object a library module
`-c`	`--char_is_signed`	‘`char`’ is ‘`signed char`’
`-gA`	`--strict_ansi`	Flag old-style functions
`-H`*`name`*	`--module_name=`*`name`*	Set object module name
`-L[`*`prefix`*`]`, `-l[c|C|a|A][N]` *`filename`*	`-l[c|C|a|A|][N]` *`filename`*	List file. The modifiers specify the type of list file to create.
`-N`*`prefix`*`, -n`	`--preprocess=[c][n][l]` *`filename`*	Preprocessor output
`-q`	`-lA, -lC`	Insert mnemonics. List file syntax has changed.
`-R` *`name`*	`--segment`	Set code segment name.
`-r[012][i][n]`	`-r` `--debug`	Generate debug information. The modifiers have been removed.
`-S`	`--silent`	Set silent operation
`-v[0|1|2|3]`	`--cpu=`*`xxxx`* `-v[0|1|2|3|4|5|6]`	Processor configuration.
`-w`	`--no_warnings`	Disable warnings

Note: A number of new command line options have been added in ICCAVR. For a complete list of the available command line options, see *Options summary*, page 82.

FILENAMES

In ICCA90, file references can be made in either of the following ways:

- With a specific filename, and in some cases with a default extension added, using a command line option such as `-a filename` (Assembly output to named file).
- With a prefix string added to the default name, using a command line option such as `-A[prefix]` (Assembly output to prefixed filename).

In ICCAVR, a file reference is always regarded as a *file path* that can be a directory, which the compiler will check and then add a default filename to, or a filename.

The following table shows some examples where it is assumed that the source file is named `test.c`, `myfile` is *not* a directory and `mydir` is a directory:

ICCA90 command	*ICCAVR command*	*Result*
`-l myfile`	`-l myfile`	`myfile.lst`
`-Lmyfile`	`-l myfiletest`	`myfiletest.lst`
`-L`	`-l .`	`test.lst`
`-Lmydir/`	`-l mydir` `-l mydir/`	`mydir/test.lst`

LIST FILES

In ICCA90, only one C list file and one assembler list file can be produced; in ICCAVR there is no upper limit on the number of list files that can be generated. The ICCAVR command line option `-l[c|C|a|A][N] filename` is used for specifying the behavior of each list file.

OBJECT FILE FORMAT

In some products using the previous generation of compiler technology, two types of source references can be generated in the object file. When the command line option `-r` is used, the source statements are being referred to, and when the command line option `-re` is used, the actual source code is embedded in the object format.

In ICCAVR, when the command line option `-r` or `--debug` is used, source file references are always generated, i.e. embedding of the source code is not supported.

NESTED COMMENTS

In ICCA90, nested comments were allowed if the option `-C` was used. In ICCAVR, nested comments are never allowed. For example, if a comment were used for removing a statement as in the following example, it would not have the desired effect.

```
/*
/* x is a counter */
int x = 0;
*/
```

The variable *x* will still be defined, there will be a warning where the inner comment begins, and there will be an error where the outer comment ends.

```
  /* x is a counter */
  ^
"c:\bar.c",2  Warning[Pe009]: nested comment is not
allowed

  */
   ^
"c:\bar.c",4  Error[Pe040]: expected an identifier
```

The solution is to use `#if 0` to "hide" portions of the source code when compiling:

```
#if 0
/* x is a counter */
int x = 0;
#endif
```

Note: `#if` statements may be nested.

PREPROCESSOR FILE

In ICCA90, a preprocessor file can be generated as a side effect of compiling a source file.

In ICCAVR a preprocessor file is either generated as a side effect, or as the whole purpose when parsing of the source code is not required. You may also choose to include or exclude comments and/or `#line` directives.

CROSS-REFERENCE INFORMATION

In ICCA90, cross-reference information can be generated. This possibility is not available in ICCAVR.

SIZEOF IN PREPROCESSOR DIRECTIVES

In `ICCA90`, `sizeof` could be used in `#if` directives, for example:

```
#if sizeof(int)==2
int i = 0;
#endif
```

In ICCAVR, `sizeof` is not allowed in `#if` directives. The following error message will be produced:

```
  #if sizeof(int)==2
      ^
"c:\bar.c",1  Error[Pe059]: function call is not allowed
in a constant expression.
```

Macros can be used instead, for example `SIZEOF_INT`. Macros can be defined using the `-D` option, or a `#define` in the source code:

```
#define SIZEOF_INT 2
#if SIZEOF_INT==2
int i = 0;
#endif
```

To find the size of a predefined data type, see *Data types*, page 51.

Complex data types may be computed using one of several methods:

1. Write a small program, and run it in the simulator, with terminal I/O.

   ```
   #include <stdio.h>
   struct s { char c; int a; };
   ```

```
void main(void)
{
  printf("sizeof(struct s)=%d \n", sizeof(struct s));
}
```

2 Write a small program, compile it with the option `-la .` to get an assembler listing in the current directory and look for the definition of the constant `x`.

```
struct s { char c; int a; };
const int x = sizeof(struct s);
```

Note: The file `limits.h` contains macro definitions that can be used instead of `#if sizeof`.

IMPLEMENTATION-DEFINED BEHAVIOR

This chapter describes how IAR C handles the implementation-defined areas of the C language.

ISO 9899:1990, the International Organization for Standardization standard - *Programming Languages - C* (revision and redesign of ANSI X3.159-1989, American National Standard), changed by the ISO Amendment 1:1994, *Technical Corrigendum 1,* and *Technical Corrigendum 2*, contains an appendix called *Portability Issues*. The ISO appendix lists areas of the C language that ISO leaves open to each particular implementation.

Note: IAR C adheres to a freestanding implementation of the ISO standard for the C programming language. This means that parts of a standard library can be excluded in the implementation. IAR has not implemented the following parts of the standard library: `locale`, `files` (but streams `stdin` and `stdout`), `time`, and `signal`.

This chapter follows the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

TRANSLATION

DIAGNOSTICS (5.1.1.3)

IAR C produces diagnostics in the form:

`filename``,`*`linenumber level`*`[`*`tag`*`]: `*`message`*

where *`filename`* is the name of the source file in which the error was encountered; *`linenumber`* is the line number at which the compiler detected the error; *`level`* is the level of seriousness of the message (remark, warning, error, or fatal error); *`tag`* is a unique tag that identifies the message; *`message`* is an explanatory message, possibly several lines.

ENVIRONMENT

ARGUMENTS TO MAIN (5.1.2.2.2.1)

In IAR C, the function called at program startup is called `main`. There is no prototype declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior, see the `CSTARTUP` description, page 26.

INTERACTIVE DEVICES (5.1.2.3)

IAR C treats the streams `stdin` and `stdout` as interactive devices.

IDENTIFIERS

SIGNIFICANT CHARACTERS WITHOUT EXTERNAL LINKAGE (6.1.2)

The number of significant initial characters in an identifier without external linkage is 200.

SIGNIFICANT CHARACTERS WITH EXTERNAL LINKAGE (6.1.2)

The number of significant initial characters in an identifier with external linkage is 200.

CASE DISTINCTIONS ARE SIGNIFICANT (6.1.2)

IAR C treats identifiers with external linkage as case-sensitive.

CHARACTERS

SOURCE AND EXECUTION CHARACTER SETS (5.2.1)

The source character set is the set of legal characters that can appear in source files. In IAR C, the source character set is the standard ASCII character set.

The execution character set is the set of legal characters that can appear in the execution environment. In IAR C, the execution character set is the standard ASCII character set.

BITS PER CHARACTER IN EXECUTION CHARACTER SET (5.2.4.2.1)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

MAPPING OF CHARACTERS (6.1.3.4)

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way, i.e. using the same representation value for each member in the character sets, except for the escape sequences listed in the ISO standard.

UNREPRESENTED CHARACTER CONSTANTS (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant, generates a diagnostic and will be truncated to fit the execution character set.

CHARACTER CONSTANT WITH MORE THAN ONE CHARACTER (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character, generates a diagnostic.

CONVERTING MULTIBYTE CHARACTERS (6.1.3.4)

The current and only locale supported in IAR C is the ‘C’ locale.

RANGE OF ’PLAIN’ CHAR (6.2.1.1)

A ‘plain’ `char` has the same range as an `unsigned char`.

INTEGERS

RANGE OF INTEGER VALUES (6.1.2.5)

The representation of integer values are in two's-complement form. The most-significant bit holds the sign; 1 for negative, 0 for positive and zero.

See *Data types*, page 51, 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 52, for information about the ranges and sizes for the different floating-point types: `float`, `double`, and `long double`.

CONVERTING INTEGER VALUES TO FLOATING-POINT VALUES (6.2.1.3)

When an integral number is cast to a floating-point value that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

DEMOTING FLOATING-POINT VALUES (6.2.1.4)

When a floating-point value is converted to a floating-point value of narrower type that cannot exactly represent the value, the value is rounded(up or down) to the nearest suitable value.

ARRAYS AND POINTERS

SIZE_T (6.3.3.4, 7.1.1)

See *size_t*, page 55, for information about `size_t` in IAR C.

CONVERSION FROM/TO POINTERS (6.3.4)

See *Casting*, page 55, for information about casting of data pointers and function pointers.

PTRDIFF_T (6.3.6, 7.1.1)

See *ptrdiff_t*, page 55, 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 51, 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
instantiate
language
location
memory
message
none
no_pch
NOTREACHED
object_attribute
once
optimize
pack
__printf_args
__scanf_args
type_attribute
VARARGS
vector
warnings
```

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

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 `0`.

DIAGNOSTIC PRINTED BY THE ASSERT FUNCTION (7.2)

The `assert()` function prints:

`filename``:`*`linenr expression`* `-- assertion failed`

when the parameter evaluates to zero.

DOMAIN ERRORS (7.5.1)

`NaN` (Not a Number) will be returned by the mathematic functions on domain errors.

UNDERFLOW OF FLOATING-POINT VALUES SETS ERRNO TO ERANGE (7.5.1)

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

FMOD() FUNCTIONALITY (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns `NaN`; `errno` is set to `EDOM`.

SIGNAL() (7.7.1.1)

IAR C does not support the signal part of the library.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

TERMINATING NEWLINE CHARACTER (7.9.2)

`Stdout` stream functions recognize either `newline` or `end of file (EOF)` as the terminating character for a line.

BLANK LINES (7.9.2)

Space characters written out to the `stdout` stream immediately before a newline character are preserved. There is no way to read in the line through the stream `stdin` that was written out through the stream `stdout` in IAR C.

NULL CHARACTERS APPENDED TO DATA WRITTEN TO BINARY STREAMS (7.9.2)

There are no binary streams implemented in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

FILES (7.9.3)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

REMOVE() (7.9.4.1)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

RENAME() (7.9.4.2)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

%P IN PRINTF() (7.9.6.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `void *`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

%P IN SCANF() (7.9.6.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts that into a value with the type `void *`.

READING RANGES IN SCANF() (7.9.6.2)

A `-` (dash) character is always treated as a range symbol.

FILE POSITION ERRORS (7.9.9.1, 7.9.9.4)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

MESSAGE GENERATED BY PERROR() (7.9.10.4)

The generated message is:

usersuppliedprefix`:`*errormessage*

ALLOCATING ZERO BYTES OF MEMORY (7.10.3)

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`.

BEHAVIOR OF ABORT() (7.10.4.1)

The `abort()` function does not flush stream buffers, and it does not handle files, since this is an unsupported feature in IAR C.

BEHAVIOR OF EXIT() (7.10.4.3)

The `exit()` function does not return in IAR C.

ENVIRONMENT (7.10.4.4)

An environment is not supported in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

SYSTEM() (7.10.4.5)

The `system()` function is not supported in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

MESSAGE RETURNED BY STRERROR() (7.11.6.2)

The messages returned by `strerror()` depending on the argument is:

Argument	*Message*
`EZERO`	`no error`
`EDOM`	`domain error`
`ERANGE`	`range error`
`EFPOS`	`file positioning error`
`EILSEQ`	`multi-byte encoding error`
`<0 || >99`	`unknown error`
all others	`error` *`nnn`*

THE TIME ZONE (7.12.1)

Time is not supported in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

CLOCK() (7.12.2.1)

Time is not supported in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

IAR C EXTENSIONS

This chapter describes IAR extensions to the ISO standard for the C programming language.

See the compiler option `-e`, page 89 for information about enabling and disabling language extensions from the command line.

AVAILABLE EXTENSIONS

The following language extensions are available:

- Functions and data may be declared with memory, type, and object attributes. The attributes follow the syntax for qualifiers but not the semantics.
- The operator `@` may be used for specifying either the location of an absolute addressed variable or the segment placement of a function or variable. The directive `#pragma location` has the same effect.
- A translation unit (input file) is allowed to contain no declarations.
- Comment text can appear at the end of preprocessing directives.
- `__ALIGNOF__` is similar to `sizeof`, but returns the alignment requirement value for a type, or 1 if there is no alignment requirement. It may be followed by a type or expression in parenthesis, `__ALIGNOF__(`*`type`*`)`, or `__ALIGNOF__(`*`expression`*`)`. The expression in the second form is not evaluated.
- Bitfields may have base types that are `enums` or integral types besides `int` and `unsigned int`. This matches *G.5.8* in the appendix to the ISO standard, *ISO Portability issues*.
- The last member of a `struct` may have an incomplete `array` type. It may not be the only member of the `struct` (otherwise, the `struct` would have zero size).
- A file-scope `array` may have an incomplete `struct`, `union`, or `enum` type as its element type. The type must be completed before the `array` is subscripted (if it is), and by the end of the compilation if the `array` is not external.
- Static functions may be declared in function and block scopes. Their declarations are moved to the file scope.

- `enum` tags may be incomplete: one may define the tag name and resolve it (by specifying the brace-enclosed list) later.
- The values of enumeration constants may be given by expressions that evaluate to unsigned quantities that fit in the `unsigned int` range but not in the `int` range. A warning is issued for suspicious cases.
- An extra comma is allowed at the end of an `enum` list. A remark is issued.
- The final semicolon preceeding the closing `}` of a `struct` or `union` specifier may be omitted. A warning is issued.
- A label definition may be immediately followed by a closing `}` (normally a statement must follow a label definition). A warning is issued.
- An empty declaration (a semicolon with nothing before it) is allowed. A remark is issued.
- An initializer expression that is a single value and is used for initializing an entire static `array`, `struct`, or `union` need not be enclosed in braces. ISO C requires the braces.
- In an initializer, a pointer constant value may be cast to an integral type if the integral type is large enough to contain it.
- The address of a variable with a register storage class may be taken. A warning is issued.
- In an integral constant expression, an integer constant may be cast to a pointer type and then back to an integral type.
- In duplicate size and sign specifiers (for example `short short` or `unsigned unsigned`) the redundancy is ignored. A warning is issued.
- `long float` is accepted as synonym of `double`.
- Benign redeclarations of `typedef` names are allowed. That is, a `typedef` name may be redeclared in the same scope as the same type. A warning is issued.
- Dollar signs are accepted in identifiers.

- Numbers are scanned according to the syntax for numbers rather than the pp-number syntax. Thus, `0x123e+1` is scanned as three tokens instead of one invalid token (if `--strict_ansi` is specified, the pp-number syntax is used).
- Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical, for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size (for example, `short *` and `int *`). A warning is issued. Assignment of a string constant to a pointer to any kind of character is allowed without a warning.
- Assignment of pointer types is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example `int **` to `int const **`). Comparisons and pointer difference of such pairs of pointer types are also allowed. A warning is issued.
- In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a `null` pointer constant is always implicitly converted to a `null` pointer of the right type if necessary. In ISO C, some operators allow such things, while others do not allow them.
- `asm` statements are accepted, like `asm("LD A,#5");`. This is disabled in strict ISO/ANSI C mode.
- Anonymous `structs` and `unions` (similar to the C++ anonymous unions) are allowed. An anonymous structure type defines an unnamed object (and not a type) whose member names are promoted to the surrounding scope. The member names must be unique in the surrounding scope. External anonymous structure types are allowed.
- External entities declared in other scopes are visible. A warning is issued. Example:

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

- End-of-line comments (`//`, as in C++) are allowed.
- A non-`lvalue` array expression is converted to a pointer to the first element of the array when it is subscripted or similarly used.

EXTENSIONS ACCEPTED IN NORMAL EC++ MODE

The following extensions are accepted in all modes (except when strict ANSI violations are diagnosed as errors):

- A `friend` declaration for a class may omit the `class` keyword:

```
class B;
class A {
  friend B;       // Should be 'friend class B'
};
```

- Constants of scalar type may be defined within classes (this is an old form; the modern form uses an initialized static data member):

```
class A {
  const int size = 10;
  int a[size];
};
```

- In the declaration of a class member, a qualified name may be used:

```
struct A {
  int A::f();     // Should be int f ();
};
```

- The preprocessing symbol `c_plusplus` is defined in addition to the standard `_cplusplus`.

- An extension is supported to allow an anonymous union to be introduced into a containing class by a `typedef` name—it does not have to be declared directly, as with a true anonymous union. For example:

```
typedef union {
  int i, j;
} U;              // U identifies a reusable anonymous
                  // union.
class A {
  U;              // Okay -- references to A::i and
                  // A::j are allowed.
}
```

  In addition, the extension also permits 'anonymous classes' and 'anonymous structures', as long as they have no EC++ feature (for example, no static data members or member functions, and no non-public members) and have no nested types other than other anonymous classes, structures, or unions.

For example:

```
struct A  {
   struct {
     int i, j;
   };              // Okay -- references to A::i and
                   // A::j are allowed.
};
```

- An assignment operator declared in a derived class with a parameter type matching one of its base classes is treated as a 'default' assignment operator—that is, such a declaration blocks the implicit generation of a copy assignment operator. For example:

```
struct A { };
struct B : public A {
  B& operator=(A&);
};
```

- By default, as well as in C front compatibility mode, there will be no implicit declaration of `B::operator=(const B&)`, whereas in strict ANSI mode `B::operator=(A&)` is *not* a copy assignment operator and `B::operator=(const B&)` *is* implicitly declared.

LANGUAGE FEATURES NOT ACCEPTED IN EC++

The following ISO/ANSI C++ features are not accepted in EC++:

- `reinterpret_cast` does not allow casting a pointer to a member of one class to a pointer to a member of another class if the classes are unrelated.
- In a reference of the form `f()->g()`, with `g` being a static member function, `f()` is not evaluted.
- Class name injection is not implemented.
- Friend functions of the argument class types cannot be found by name lookup on the function name in calls since this feature is not implemented.
- String literals do not have the `const` type.
- Universal character set escape sequences (for example, `\uabcd`) are not implemented.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Symbols

Numerics