8051 IAR C/C++ Compiler

Reference Guide

for the

MCS-51 Microcontroller Family

EDITION NOTICE

Fourth edition: February 2008

Part number: C8051-4

This guide applies to version 7.4x of 8051 IAR Embedded Workbench®.

Internal reference: ISUD.

Brief contents

Contents

Tables

Preface

Welcome to the 8051 IAR C/C++ Compiler Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use the 8051 IAR C/C++ Compiler to best suit your application requirements. This guide also gives you suggestions on coding techniques so that you can develop applications with maximum efficiency.

Who should read this guide

You should read this guide if you plan to develop an application using the C or C++ language for the MCS-51 microcontroller and need to get detailed reference information on how to use the 8051 IAR C/C++ Compiler. In addition, you should have a working knowledge of the following:

- The architecture and instruction set of the MCS-51 microcontroller. Refer to the documentation from chip manufacturer for information about the MCS-51 microcontroller
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host computer.

How to use this guide

When you start using the 8051 IAR C/C++ Compiler, you should read *Part 1. Using the compiler* in this guide.

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

If you are new to using the IAR Systems build tools, we recommend that you first study the *IAR Embedded Workbench® IDE User Guide*. This guide contains a product overview, tutorials that can help you get started, conceptual and user information about the IAR Embedded Workbench IDE and the IAR C-SPY® Debugger, and corresponding reference information.

What this guide contains

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

Part 1. Using the compiler

- *Getting started* gives the information you need to get started using the 8051 IAR C/C++ Compiler for efficiently developing your application.
- *Understanding memory architecture* gives an overview of the MCS-51 microcontroller memory configuration in terms of the different memory spaces available. The chapter also gives an overview of the concepts related to memory available in the 8051 IAR C/C++ Compiler and the linker.
- *Data storage* describes how data can be stored in memory, with emphasis on the different data models and data memory type attributes.
- *Functions* gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.
- *Banked functions* introduces the banking technique; when to use it, what it does, and how it works.
- *Placing code and data* describes the concept of segments, introduces the linker command file, and describes how code and data are placed in memory.
- *The DLIB runtime environment* describes the DLIB runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization and introduces the file `cstartup`, as well as how to use modules for locale, and file I/O.
- *The CLIB runtime environment* gives an overview of the CLIB runtime libraries and how they can be customized. The chapter also describes system initialization and introduces the file `cstartup`.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

Part 2. Reference information

- *External interface details* provides reference information about how the compiler interacts with its environment—the invocation syntax, methods for passing options to the compiler, environment variables, the include file search procedure, and the different types of compiler output. The chapter also describes how the compiler's diagnostic system works.

- *Compiler options* explains how to set options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Data representation* describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.
- *Compiler extensions* gives a brief overview of the compiler extensions to the ISO/ANSI C standard. More specifically the chapter describes the available C language extensions.
- *Extended keywords* gives reference information about each of the 8051-specific keywords that are extensions to the standard C/C++ language.
- *Pragma directives* gives reference information about the pragma directives.
- *Intrinsic functions* gives reference information about the functions that can be used for accessing 8051-specific low-level features.
- *The preprocessor* gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.
- *Library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.
- *Segment reference* gives reference information about the compiler's use of segments.
- *Implementation-defined behavior* describes how the 8051 IAR C/C++ Compiler. handles the implementation-defined areas of the C language standard.

Other documentation

The complete set of IAR Systems development tools for the MCS-51 microcontroller is described in a series of guides. For information about:

- Using the IAR Embedded Workbench® IDE with the IAR C-SPY Debugger®, refer to the *IAR Embedded Workbench® IDE User Guide*
- Programming for the 8051 IAR Assembler, refer to the *8051 IAR Assembler Reference Guide*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR DLIB Library functions, refer to the online help system
- Using the IAR CLIB Library functions, refer to the *IAR C Library Functions Reference Guide*, available from the online help system.

- Porting application code and projects created with a previous 8051 IAR Embedded Workbench IDE, refer to *8051 IAR Embedded Workbench® Migration Guide*
- Developing safety-critical applications using the MISRA C guidelines, refer to the *IAR Embedded Workbench® MISRA C Reference Guide*.

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

FURTHER READING

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

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

We recommend that you visit the following web sites:

- The website of your chip manufacturer
- The IAR Systems web site, **www.iar.com**, holds application notes and other product information.
- Finally, the Embedded C++ Technical Committee web site, **www.caravan.net/ec2plus**, contains information about the Embedded C++ standard.

Document conventions

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

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

Style	Used for
`computer`	Text that you enter or that appears on the screen.
`parameter`	A label representing the actual value you should enter as part of a command.
`[option]`	An optional part of a command.
`{option}`	A mandatory part of a command.
`a|b|c`	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
reference	A cross-reference within this guide or to another guide.
…	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
🛠	Identifies instructions specific to the IAR Embedded Workbench interface.
🖥	Identifies instructions specific to the command line interface.
💡	Identifies helpful tips and programming hints.

Table 1: Typographic conventions used in this guide

Part 1. Using the compiler

This part of the 8051 IAR C/C++ Compiler Reference Guide includes the following chapters:

Getting started

This chapter gives the information you need to get started using the 8051 IAR C/C++ Compiler for efficiently developing your application.

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

Next, the compiler is introduced. You will get an overview of the basic settings needed for a project setup, including an overview of the techniques that enable applications to take full advantage of the MCS-51 microcontroller. In the following chapters, these techniques will be studied in more detail.

To get a deeper understanding about the concepts and available features presented here, you need to read the other chapters in this guide.

Device support

To get a smooth start with your product development, the IAR product installation comes with wide range of device-specific support.

SUPPORTED MCS-51 DEVICES

The 8051 IAR C/C++ Compiler supports all devices based on the standard MCS-51 microcontroller. The following extensions are also supported:

- Multiple data pointers (DPTRs). Support for up to eight data pointers is integrated in the code generator
- Extended code memory, up to 16 Mbytes. (Used for example in Maxim (Dallas Semiconductors) DS80C390/DS80C400 devices.)
- Extended data memory, up to 16 Mbytes. (Used for example in the Analog Devices ADuC812 device and in Maxim DS80C390/DS80C400 devices.)
- Maxim DS80C390/DS80C400 devices and similar devices, including support for the extended instruction set, multiple data pointers, and extended stack (a call stack located in xdata memory)
- Mentor Graphics M8051W/M8051EW core and devices based on this, including support for the banked `LCALL` instruction, banked `MOVC A,@A+DPTR`, and for placing interrupt service routines in banked memory.

To read more about how to set up your project depending on device you are using, see *Basic project settings*, page 6.

PRECONFIGURED SUPPORT FILES

The IAR product installation contains a vast amount of preconfigured files for supporting different devices.

Header files for I/O

Standard peripheral units are defined in device-specific I/O header files with the filename extension `h`. The product package supplies I/O files for many devices. You can find these files in the `8051\inc` directory. Make sure to include the appropriate include file in your application source files. If you need additional I/O header files, they can be created using one of the provided ones as a template.

Linker command files

The `8051\config` directory contains ready-made linker command files for some supported devices. The files have the filename extension `xcl` and contain the information required by the linker. To read more about the linker command file, see *Placing code and data*, page 67.

Device description files

The IAR C-SPY Debugger handles several of the device-specific requirements, such as definitions of peripheral registers and groups of these, by using device description files. These files are located in the `8051\config\devices` directory and they have the filename extension `ddf`. To read more about these files, see the *IAR Embedded Workbench® IDE User Guide*.

EXAMPLES FOR GETTING STARTED

The `8051\src\examples` directory contains many examples of working applications to give you a smooth start with your development. The complexity of the examples ranges from simple LED blink to more advanced types of examples. There are examples provided for most of the supported devices.

IAR language overview

There are two high-level programming languages you can use with the 8051 IAR C/C++ Compiler:

- C, the most widely used high-level programming language used in the embedded systems industry. Using the 8051 IAR C/C++ Compiler, you can build freestanding applications that follow the standard ISO 9899:1990. This standard is commonly known as ANSI C.

- C++, a modern object-oriented programming language with a full-featured library well suited for modular programming. IAR Systems supports two levels of the C++ language:
  - Embedded C++ (EC++), a subset of the C++ programming standard, which is intended for embedded systems programming. It is defined by an industry consortium, the Embedded C++ Technical committee. See the chapter *Using C++*.
  - IAR Extended Embedded C++, with additional features such as full template support, multiple inheritance, namespace support, the new cast operators, as well as the Standard Template Library (STL).

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

It is also possible to implement parts of the application, or the whole application, in assembler language. See the *8051 IAR Assembler Reference Guide*.

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

Building applications—an overview

A typical application is built from a number of source files and libraries. The source files can be written in C, C++, or assembler language, and can be compiled into object files by the 8051 IAR C/C++ Compiler or the 8051 IAR Assembler.

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

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

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

COMPILING

In the command line interface, the following line compiles the source file `myfile.c` into the object file `myfile.r51` using the default settings:

```
icc8051 myfile.c
```

In addition, you need to specify some critical options, see *Basic project settings*, page 6.

LINKING

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

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

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

```
xlink myfile.r51 myfile2.r51 -s __program_start -f lnk51.xcl
cl-pli-nsid-1e16x01.r51 -o aout.a51 -r
```

In this example, `myfile.r51` and `myfile2.r51` are object files, `lnk51.xcl` is the linker command file, and `cl-pli-nsid-1e16x01.r51` is the runtime library. The option `-s` specifies the label where the application starts. The option `-o` specifies the name of the output file, and the option `-r` is used for specifying the output format UBROF, which can be used for debugging in C-SPY®.

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

Basic project settings

This section gives an overview of the basic settings for the project setup that are needed to make the compiler generate the best code for the MCS-51 device you are using. You can specify the options either from the command line interface or in the IAR Embedded Workbench IDE.

Depending on your hardware memory configuration, there are basic settings for:

- Core

  The 8051 IAR C/C++ Compiler supports the classic Intel 8051 microcontroller core, the Maxim (Dallas Semiconductor) DS80C390 core, the Mentor Graphics M8051W/M8051EW core, as well as similar devices of these. Use the --core option to choose the core—Plain, Extended1, or Extended2—that matches the device you are using. This option reflects the addressing capability of the target microcontroller. For information about the cores, see *Basic project settings for hardware memory configuration*, page 15.

- Data model and code model

  One of the characteristics of the MCS-51 microcontroller is that there is a trade-off regarding the way memory is accessed, between the range from cheap access to small memory areas, up to more expensive access methods that can access external data. For more information, see the chapter *Understanding memory architecture*.

  Use the `--data_model` compiler option to specify (among other things) the default memory placement of static and global variables, which also implies a default memory access method.

  Use the `--code_model` compiler option to specify the default placement of functions, which also implies the default function call method. For more information, see *Code models for function storage*, page 47.

- Calling convention

  Use the `--calling_convention` compiler option to control whether the compiler by default should use a stack model or an overlay model for local data and in which memory the stack or overlay frame should be placed. In other words, whether local variables and parameters should be placed on the stack or via an overlay memory area, and where in memory the stack/memory area should be placed. For more information, see *Choosing a calling convention*, page 35.

- DPTR setup

  Some devices provide up to 8 data pointers and if used, execution of some library routines can be improved.

  Use the `--dptr` compiler option to specify the number of data pointers to use, the size of the data pointers, and where the data pointers should be located. The memory addresses for data pointers are specified in the linker command file or in the IAR Embedded Workbench IDE.

Not all combinations of core variant, code model, data model, calling convention, and DPTR setup are possible. You must choose the settings depending on your hardware memory configuration based on which device you are using. For detailed information, see *Basic project settings for hardware memory configuration*, page 15.

See the chapter *Compiler options* for syntax information and for information about corresponding options in the IAR Embedded Workbench IDE.

OPTIMIZATION FOR SPEED AND SIZE

The 8051 IAR C/C++ Compiler is a state-of-the-art compiler with an optimizer that performs, among other things, dead-code elimination, constant propagation, inlining, common sub-expression elimination, and precision reduction. It also performs loop optimizations, such as unrolling and induction variable elimination.

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

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

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

RUNTIME ENVIRONMENT

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

There are two different sets of runtime libraries provided:

- The IAR DLIB Library, which supports ISO/ANSI C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, et cetera. (This library is the default for the C++ language).
- The IAR CLIB Library is a light-weight library, which is not fully compliant with ISO/ANSI C. Neither does it fully support floating-point numbers in IEEE 754 format nor does it support Embedded C++. (This library is used by default for the C language).

The runtime library you choose can be one of the prebuilt libraries, or a library that you have customized and built yourself. The IAR Embedded Workbench IDE provides a library project template for both libraries, that you can use for building your own library version. This gives you full control of the runtime environment. If your project only contains assembler source code, there is no need to choose a runtime library.

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

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

Choosing a runtime library in the IAR Embedded Workbench IDE

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

Note that for the DLIB library there are different configurations—Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, et cetera. See *Library configurations*, page 87, for more information.

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

Choosing runtime environment from the command line

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

Command line	Description
`-I 8051\inc`	Specifies the include paths
`-I 8051\inc\{clib|dlib}`	Specifies the library-specific include path. Use `clib` or `dlib` depending on which library you are using.
`libraryfile``.r51`	Specifies the library object file
`--dlib_config C:\...\`*`configfile`*`.h`	Specifies the library configuration file (for the IAR DLIB Library only)

Table 2: Command line options for specifying library and dependency files

For information about prebuilt library object files for the IAR DLIB Library, see *Using a prebuilt library*, page 88. For information about prebuilt object files for the IAR CLIB Library, see *Runtime environment*, page 115.

In these chapters you can also find information about how the object files correspond to the dependent project options, and the corresponding configuration files. Make sure to use the object file that matches your other project options.

Setting library and runtime environment options

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

- The formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 91 (DLIB) and *Input and output*, page 118 (CLIB).
- The size of the stack and the heap, see *The stacks*, page 76, and *The heap*, page 79, respectively.

Building your own library

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

- DLIB and CLIB runtime environment
- Core variant
- Code models
- Data model
- Calling convention
- Stack location
- Constant location
- Number of data pointers
- Data pointer visibility
- Data pointer size
- Data pointer selection method
- Library configuration: Normal.

Some of these combinations are delivered as prebuilt runtime libraries. When you need a compiler configuration for which there is no prebuilt library, you can build your own library. This is simple, as the IAR Embedded Workbench IDE provides a library project template which can be used for customizing the runtime environment configuration according to your needs.

For detailed information, see *Building and using a customized library*, page 95.

Special support for embedded systems

This section briefly describes the extensions provided by the 8051 IAR C/C++ Compiler to support specific features of the MCS-51 microcontroller.

EXTENDED KEYWORDS

The 8051 IAR C/C++ Compiler provides a set of keywords that can be used for configuring how the code is generated. For example, there are keywords for controlling the memory type for individual variables as well as for declaring special function types.

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

The command line option `-e` makes the extended keywords available, and reserves them so that they cannot be used as variable names. See, *-e*, page 200 for additional information.

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

PRAGMA DIRECTIVES

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

The pragma directives are always enabled in the 8051 IAR C/C++ Compiler. They are consistent with ISO/ANSI C, and are very useful when you want to make sure that the source code is portable.

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

PREDEFINED SYMBOLS

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

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

SPECIAL FUNCTION TYPES

The special hardware features of the MCS-51 microcontroller are supported by the compiler's special function types: interrupt, monitor, and task. You can write a complete application without having to write any of these functions in assembler language.

For detailed information, see *Primitives for interrupts, concurrency, and OS-related programming*, page 48.

ACCESSING LOW-LEVEL FEATURES

For hardware-related parts of your application, accessing low-level features is essential. The 8051 IAR C/C++ Compiler supports several ways of doing this: intrinsic functions, mixing C and assembler modules, and inline assembler. For information about the different methods, see *Mixing C and assembler*, page 125.

Understanding memory architecture

This chapter gives an overview of the MCS-51 microcontroller memory configuration in terms of the different memory spaces available. To use memory efficiently you also need to be familiar with the concepts related to memory available in the 8051 IAR C/C++ Compiler and the linker.

Finally, for each type of supported device you will get a brief overview about basic project settings to get started setting up the tools for the device you are using.

The MCS-51 microcontroller memory configuration

The MCS-51 microcontroller has three separate *memory spaces*: code memory, internal data memory, and external data memory.

The different memory spaces are accessed in different ways. Internal data memory can always be efficiently accessed, whereas external data memory and code memory cannot always be accessed in the same efficient way.

CODE MEMORY SPACE

In a classic 8051, the code memory space is a 64-Kbyte address area of ROM memory that is used for storing program code, including all functions and library routines, but also constants. Depending on the device you are using and your hardware design, code memory can be internal (on-chip), external, or both.

For classic 8051/8052 devices, code memory can be expanded with up to 256 banks of additional ROM memory. The compiler uses 2-byte pointers to access the different banks. Silabs C8051F12x and Texas Instruments CC2430 are examples of this device type.

Furthermore, some devices have *extended code memory* which means they can have up to 16 Mbytes of linear code memory (used for example in the Maxim DS80C390/DS80C400 devices). The compiler uses 3-byte pointers to access this area.

Devices based on the Mentor Graphics M8051W/M8051EW core divide their code memory in 16 memory banks where each bank is 64 Kbytes. The compiler uses 3-byte pointers to access the different banks.

INTERNAL DATA MEMORY SPACE

Depending on the device, internal data memory consists of up to 256 bytes of on-chip read and write memory that is used for storing data, typically frequently used variables. In this area, memory accesses uses either the direct addressing mode or the indirect addressing mode of the `MOV` instruction. However, in the upper area (`0x80-0xFF`), direct addressing accesses the dedicated SFR area, whereas indirect addressing accesses the IDATA area.

The SFR area, is used for memory-mapped registers, such as `DPTR`, `A`, the serial port destination register `SBUF`, and the user port `P0`. Standard peripheral units are defined in device-specific I/O header files with the filename extension `h`. For more details about these files, see *Header files for I/O*, page 4.

The area between `0x20` and `0x2F` is bit-addressable, as well as `80`, `88`, `90`, `98...F0`, and `F8`. Note that the compiler reserves one byte of the bit-addressable memory for internal use, see *Virtual registers*, page 44.

EXTERNAL DATA MEMORY SPACE

External data can consist of up to 64 Kbytes of read and write memory, which can be addressed only indirectly via the `MOVX` instruction. For this reason, external memory is slower than internal memory.

Many modern devices provide on-chip XDATA (external data) in the external data memory space. For example, the Texas Instruments CC2430 device has 8 Kbytes on-chip XDATA.

Some devices extend the external data memory space to 16 Mbytes. In this case, the compiler uses 3-byte pointers to access this area.

Run-time model concepts for memory configuration

COMPILER CONCEPTS

The 8051 IAR C/C++ Compiler associates each part of memory area with a *memory type*. By mapping different memories—or part of memories—to memory types, the compiler can generate code that can access data or functions efficiently.

For each memory type, the compiler provides a keyword—a *memory attribute*—that you can use directly in your source code. These attributes let you explicitly specify a memory type for individual objects, which means the object will reside in that memory.

You can specify a default memory type to be used by the compiler by selecting a *data model*. It is possible to override this for individual variables and pointers by using the memory attributes. Conversely, you can select a *code model* for default placement of code.

For detailed information about available data models, memory types and corresponding memory attributes, see *Data models*, page 22 and *Memory types*, page 24, respectively.

LINKER CONCEPTS

The compiler generates a number of *segments*. A segment is a logical entity containing a piece of data or code that should be mapped to a physical location in memory.

The 8051 IAR C/C++ Compiler has a number of predefined segments for different purposes. Each segment has a name that describes the contents of the segment, and a *segment memory type* that denotes the type of content. In addition to the predefined segments, you can define your own segments.

At compile time, the compiler assigns each segment its contents. For example, the segment `DATA_Z` holds zero-initialized static and global variables.

The IAR XLINK Linker is responsible for placing the segments in the physical memory range, in accordance with the rules specified in the *linker command file*.

To read more about segments, see *Placing code and data*, page 67.

Basic project settings for hardware memory configuration

Depending on the device you are using and its memory configuration, there are basic project settings required for:

- Core
- Code model
- Data model
- Calling convention
- DPTR setup.

Not all combinations of these are possible. The following paragraphs provide you with information about which settings that can be used for a specific type of device.

CLASSIC 8051/8052 DEVICES

For classic 8051/8052 devices, there are three main categories of hardware memory configurations:

- No external RAM (single chip, with or without external ROM)

- External RAM present (from 0 to 64 Kbytes)
- Banked mode (more than 64 Kbytes of ROM).

A combination of the following compiler settings can be used:

--core	**--code_model**	**--data_model**	**--calling_convention**	**--dptr**
`plain`	`near`	`tiny|small`	`do|io`	`16|24`
`plain`	`near|banked`	`tiny|small`	`ir`	`16|24`
`plain`	`near|banked`	`far`	`pr|xr`[††]`|er`[†]	`24`
`plain`	`near|banked`	`large`	`pr|xr`[††]`|er`[†]	`16`
`plain`	`near|banked`	`generic`	`do|io|ir|pr|xr`[††]`|er`[†]	`16`

Table 3: Possible combinations of compiler options for core Plain

† Requires the extended stack compiler option to be used (`--extended_stack`).
†† Requires the extended stack compiler option not to be used (`--extended_stack`).

If you use the Banked code model, you can explicitly place individual functions in the root bank by using the `__near_func` memory attribute.

You can use the `__code` memory attribute to place constants and strings in the code memory space.

The following memory attributes are available for placing individual data objects in different data memory than default: `__sfr`, `__bit`, `__bdata`, `__data`, `__idata`, `__pdata`, `__xdata`, `__xdata_rom`, `__ixdata`, `__generic`.

MAXIM (DALLAS SEMICONDUCTOR) 390 AND SIMILAR DEVICES

This type of devices can have memory extended up to 16 Mbytes of external continuous data memory and code memory.

A combination of the following compiler settings can be used:

--core	**--code_model**	**--data_model**	**--calling_convention**	**--dptr**
`extended1`	`far`	`far|large`	`pr|er`[†]`|xr`[††]	`24`
`extended1`	`far`	`tiny|small`	`ir`	`24`

Table 4: Possible combinations of compiler options for core Extended 1

† Requires an extended stack, which means the compiler option `--extended_stack` to be used.
†† Requires the compiler option `--extended_stack` not to be used.

You can use the `__far_code` and `__huge_code` memory attributes to place constants and strings in the code memory space.

The following memory attributes are available for placing individual data objects in a non-default data memory: `__sfr`, `__bit`, `__bdata`, `__data`, `__idata`, `__pdata`.

For the tiny and large data models also: `__xdata`, `__xdata_rom`, `__ixdata`, `__generic`.

For the far data model also: `__far`, `__far_rom`, `__huge`, `__huge_rom`.

Note: Although the 8051 IAR C/C++ Compiler supports the code and data memory extension model of the Maxim (Dallas Semiconductor) DS80C390 device, it does *not* support the 40-bit accumulator of the device.

DEVICES BASED ON MENTOR GRAPHICS M8051W/M8051EW CORE

This core and devices based on it, provide an extended addressing mechanism which means you can extend your code memory with up to 16 banks where each bank is 64 Kbytes.

The following combination of compiler settings can be used:

--core	**--code_model**	**--data_model**	**--calling_convention**	**--dptr**
`extended2`	`banked_ext2`	`large`	`xdata_reentrant`	`16`

Table 5: Possible combinations of compiler options for core Extended 2

The following memory attributes are available for placing individual data objects in different data memory than default: `__sfr`, `__bit`, `__bdata`, `__data`, `__idata`, `__pdata`.

For the tiny and large data models also: `__xdata`, `__xdata_rom`, `__ixdata`, `__generic`.

Using the DPTR register

Some devices have up to 8 data pointers that can be used for speeding up memory accesses. Devices that support extended memory must use 24-bit data pointers, whereas classic devices that do not support extended memory use 16-bit data pointers.

The 8051 IAR C/C++ Compiler supports up to 8 data pointers using the `DPTR` register. Using the `DPTR` register can in some cases generate more compact and efficient code. In many applications, the data pointer is a heavily used resource that often has to be saved on the stack or in a register, and later restored. If the application can use more than one data pointer, the overhead can be considerably reduced.

If you use the `DPTR` register you must specify:

- The number of data pointers to use
- The size of the data pointers
- Where the data pointers are located
- How to select a data pointer.

To set options for the `DPTR` register in the IAR Embedded Workbench IDE, choose **Project>Options>General Options>Data Pointer**.

On the command line, use the compiler option `--dptr` to specify the necessary options, see *--dptr*, page 199.

LOCATION IN MEMORY

Different 8051 devices represent the `DPTR` register in different ways. One of two methods is used for locating the data pointer register in memory.

Use the one that is supported by the device you are using.

- *Shadowed* visibility

  Means that the same SFR (`DPS` and `DPH`) addresses are used for all data pointers; the data pointer select register (`DPS`) specifies which data pointer is visible at that address.

- *Separate* visibility

  Means that different locations `DPL0`, `DPL1`, etc and `DPH0`, `DPH1` etc are used for the different data pointers. If you use this method, the `DPS` register specifies which data pointer is currently active.

If the data pointers have different locations in memory, these memory locations must be individually specified. For most devices these addresses are set up automatically by IAR Embedded Workbench. If this information is missing for your device, you can easily specify these addresses.

Specifying the location in memory

The memory addresses used for the data pointers are specified in the linker command file.

The following lines exemplifies a setup for a device that uses two 24-bit data pointers located at different addresses:

```
-D?DPS=86
-D?DPX=93
-D?DPL1=84
-D?DPH1=85
-D?DPX1=95
```

The symbol `?DPS` specifies where the `DPS` register is located. `?DPX` specifies the location of extended byte of the first data pointer. (The low and high address of the first data pointer is always located at the addresses `0x82` and `0x83`, respectively.) `?DPL1`, `?DPH1`, and `?DPX1` specify the location of the second data pointer.

SELECTING THE ACTIVE DATA POINTER

There are two different ways of switching active data pointers: incremental or XOR.

The *incrementation* method (INC) is the more efficient method, but it is not supported by all 8051 devices. Using this method, the bits in the `DPS` register can be incremented to select the active data pointer. This is only possible if the contents of the `DPS` register that is not relevant for the data pointer selection can be destroyed during the switch operation, or if bits that must not be destroyed are guarded by a *dead bit* that prevents the switch operation to overflow into them. The selection bits in the `DPS` register must also be located in sequence and start with the least significant bit.

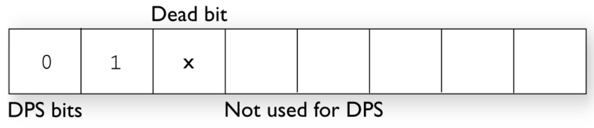


Figure 1: Selecting active data pointer

The number of DPTRs that can be used together with the INC method depends on the location of the dead bit. If, for example, four data pointers are available and bit `0` and `1` in the `DPS` register are used for data pointer selection and bit `2` is a dead bit, the INC method can only be used when all four data pointers are used. If only two of the four data pointers are used, the XOR selection method must be used instead.

If on the other hand bit `0` and `2` are used for data pointer selection and bit `1` is a dead bit, the INC method can be used when two data pointers are used, but if all four data pointers are used, the XOR method must be used instead.

The *XOR* method is not always as efficient but it can always be used. Only the bits used for data pointer selection are affected by the XOR selection operation. The bits are specified in a bit mask that must be specified if this method is used. The selection bits are marked as a *set bit* in the bit mask. For example, if four data pointers are used and the selection bits are bit `0` and bit `2`, the selection mask should be `0x05` (`00000101` in binary format).

The XOR data pointer select method can thus always be used regardless of any dead bits, of which bits are used for the selection, or of other bits used in the `DPS` register.

Note: INC is the default switching method for the Extended1 core. For other cores XOR is the default method. The default mask depends on the number of data pointers specified. Furthermore, it is assumed that the least significant bits are used, for example, if 6 data pointers are used, the default mask will be `0x07`.

Data storage

This chapter gives a brief introduction to the memory layout of the MCS-51 microcontroller and the fundamental ways data can be stored in memory: for auto variables, for global variables and local variables declared static, and for dynamically allocated data.

For efficient memory usage, 8051 IAR C/C++ Compiler provides a set of data models and data memory attributes, allowing you to fine-tune the access methods, resulting in smaller code size. The concepts of data models and memory types are described in relation to pointers, structures, Embedded C++ class objects, and non-initialized memory. Finally, detailed information about data storage on the stack and the heap is provided.

Different ways to store data

In a typical application, you can handle data in three different ways:

- Auto variables.

  All variables that are local to a function, except those declared static, are stored either in registers or in the local frame of each function. These variables can be used as long as the function executes. When the function returns to its caller, the memory space is no longer valid.

  The local frame can either be allocated at runtime from the stack—stack frame—or be statically allocated at link time—static overlay frame. Functions that use static overlays for their frame are usually not reentrant and do not support recursive calls. For more information, see *Auto variables—stack and static overlay*, page 35.

- Global variables and local variables declared `static`.

  In this case, the memory is allocated once and for all. The word static in this context means that the amount of memory allocated for this kind of variables does not change while the application is running. For more information, see *Data models*, page 22 and *Memory types*, page 24.

- Dynamically allocated data.

  An application can allocate data on the *heap*, where the data it remains valid until it is explicitly released back to the system by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that there are potential risks connected with using dynamically allocated data in systems with a limited amount of memory, or systems that are expected to run for a long time. For more information, see *Dynamic memory on the heap*, page 42.

Note: Only MCS-51 devices with external memory can have a heap.

Data models

The 8051 IAR C/C++ Compiler supports a number of data models for applications with different data requirements.

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

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

It is possible to override the default memory placement for individual variables and pointers. This makes it possible to create an application that can contain a large amount of data, and at the same time make sure that variables that are used often are placed in memory that can be efficiently accessed. For information about how to specify a memory type for individual objects, see *Using data memory attributes*, page 30.

The chosen data model may restrict the calling conventions and the location of constants and strings. When compiling non-typed ANSI C or C++ code, including the standard C libraries, the default pointer in the chosen data model must be able to reach all default data objects. Thus, the default pointer must be able to reach variables located on the stack as well as constant and strings objects. Therefore, not all combinations of data model, calling convention, and constant location are permitted, see *Calling conventions and matching data models*, page 37.

SPECIFYING A DATA MODEL

Five data models have been implemented: Tiny, Small, Large, Generic, and Far. The data models range from Tiny, which is suitable for applications—with less than 128 bytes of data—to Far, which supports up to 16 Mbytes of data. These models are controlled by the `--data_model` option. Each model has a default memory type and a default pointer size.

The following table summarizes the different data models:

Data model	Default data memory attribute	Default data pointer	Default in Core
Tiny	`__data`	`__idata`	—
Small	`__idata`	`__idata`	Plain
Large	`__xdata`	`__xdata`	—
Generic	`__xdata`	`__generic`	—
Far	`__far`	`__far`	Extended1

Table 6: Data model characteristics

Your project can only use one data model at a time, and the same model must be used by all user modules and all library modules. However, you can override the default memory type for individual data objects by explicitly specifying a memory attribute, using either keywords or the `#pragma type_attribute` directive.

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

Use the `--data_model` option to specify the data model for your project; see *--data_model*, page 194.

The Tiny data model

The Tiny data model uses Tiny memory by default, which is located in the first 128 bytes of the internal data memory space. This memory can be accessed using direct addressing. The advantage is that only 8 bits are needed for pointer storage. The default pointer type passed as a parameter will use one register or one byte on the stack.

The Small data model

The Small data model uses the first 256 Kbytes of memory by default. This memory can be accessed using 8-bit pointers. The advantage is that only 8 bits are needed for pointer storage. The default pointer type passed as a parameter will use one register or one byte on the stack.

The Large data model

The Large data model uses the first 64 Kbytes of memory by default. This memory can be accessed using 16-bit pointers. The default pointer type passed as a parameter will use two register or two bytes on the stack.

The Generic data model

The Generic data model uses 64 Kbytes of the code memory space, 64 Kbytes of the external data memory space, and 256 Kbytes of the internal data memory space. This is the only memory type that can be accessed using 24-bit pointers. The default pointer type passed as a parameter will use three registers or 3 bytes on the stack.

The Far data model

The Far data model uses the 16 Mbytes of the external data memory space. This is the only memory that can be accessed using 24-bit pointers. The default pointer type passed as a parameter will use three register or 3 bytes on the stack.

Requires that you use 24-bit data pointers, which you set explicitly using the `--dptr` compiler option.

Memory types

This section describes the concept of *memory types* used for accessing data by the 8051 IAR C/C++ Compiler. It also discusses pointers in the presence of multiple memory types. For each memory type, the capabilities and limitations are discussed.

The 8051 IAR C/C++ Compiler uses different memory types to access data that is placed in different areas of the memory. There are different methods for reaching memory areas, and they have different costs when it comes to code space, execution speed, and register usage. The access methods range from generic but expensive methods that can access the full memory space, to cheap methods that can access limited memory areas. Each memory type corresponds to one memory access method. By mapping different memories—or part of memories—to memory types, the compiler can generate code that can access data efficiently.

For example, the memory accessible using the xdata memory access method is called memory of xdata type, or simply xdata memory. For more information about memory access methods, see *Memory access methods*, page 142.

Figure 2, *The memory and associated memory types* illustrates the different memory types and how they are associated with the different parts of memory:

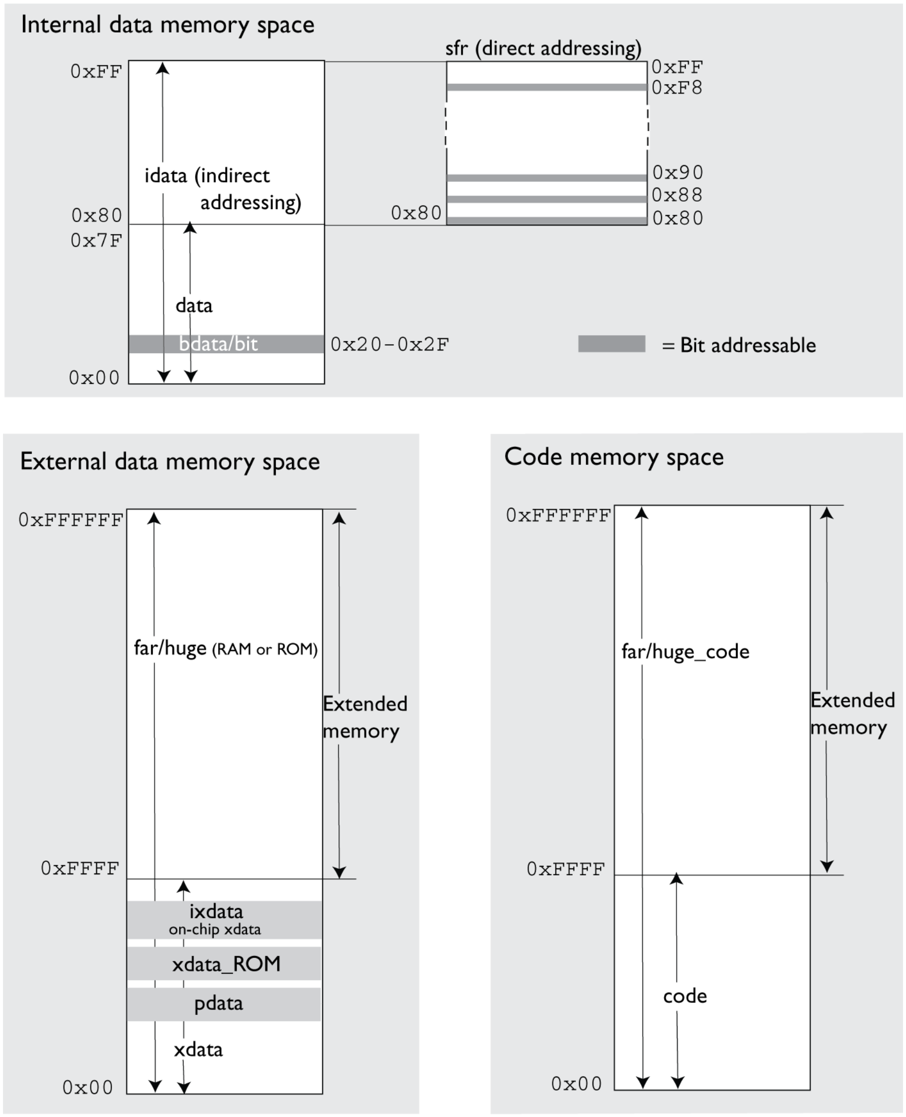


Figure 2: The memory and associated memory types

The following table summarizes the available memory types and their corresponding keywords:

Memory type	Memory space *	Memory type attribute	Address range	Max. object size	Pointer type attribute
Data	IDATA	`__data`	`0x0–0x7F`	128 bytes	`__idata`
SFR	IDATA	`__sfr`	`0x80–0xFF`	128 bytes	n/a
Idata	IDATA	`__idata`	`0x0–0xFF`	256 bytes	`__idata`
Bit	IDATA	`__bit`	`0x0–0xFF`	1 bit	n/a
Bdata	IDATA	`__bdata`	`0x0–0x2F`	16 bytes	n/a
Pdata	XDATA	`__pdata`	`0x0–0xFF`	256 bytes	`__pdata`
Ixdata	XDATA	`__ixdata`	`0x0–0xFFFF`	64 Kbytes	`__xdata`
Xdata	XDATA	`__xdata`	`0x0–0xFFFF`	64 Kbytes	`__xdata`
Far	XDATA	`__far`	`0x0–0xFFFFFF`	64 Kbytes	`__far`
Huge	XDATA	`__huge`	`0x0–0xFFFFFF`	16 Mbytes	`__huge`
Xdata ROM	XDATA	`__xdata_rom`	`0x0–0xFFFF`	64 Kbytes	`__xdata`
Far ROM	XDATA	`__far_rom`	`0x0–0xFFFFFF`	64 Kbytes	`__far`
Huge ROM	XDATA	`__huge_rom`	`0x0–0xFFFFFF`	16 Mbytes	`__huge`
Code	CODE	`__code`	`0x0–0xFFFF`	64 Kbytes	`__code`
Far code	CODE	`__far_code`	`0x0–0xFFFFFF`	64 Kbytes	`__far_code`
Huge code	CODE	`__huge_code`	`0x0–0xFFFFFF`	16 Mbytes	`__huge_code`

Table 7: Memory types and pointer type attributes

*** In this table, the term IDATA refers at the internal data memory space, XDATA refers at the external data memory space, CODE refers at the code memory space. See *The MCS-51 micro-controller memory configuration*, page 13 for more information about the memory spaces.**

All memory types are not always available; for more information, see *Basic project settings for hardware memory configuration*, page 15. For detailed information about the pointers that can be used, see *Pointer types*, page 221.

MEMORY TYPES FOR INTERNAL DATA MEMORY SPACE

In the internal data memory space, the following memory types are available:

- data
- idata
- bit/bdata
- sfr.

data

The data memory covers the first 128 bytes (`0x0-0x7F`) of the internal data memory space. To place a variable in this memory, use the `__data` memory attribute. This means the variable will be directly addressed using `MOV A,10`, which is the most compact access to variables possible. The size of such an object is limited to 128 bytes-8 bytes (the register area). This memory is default in the Tiny data model.

idata

The idata memory covers all 256 bytes of internal data memory space (`0x0-0xFF`). An idata object can be placed anywhere in this range, and the size of such an object is limited to 256 bytes-8 (the register area). To place a variable in this memory, use the `__idata` memory attribute. Such an object will be accessed with indirect addressing using the following construction `MOV R0,H10` and `MOV A,@R0`, which is slower compared to objects accessed directly in the data memory. Idata memory is default in the Small data model.

bit/bdata

The bit/bdata memory covers the 32-byte bit-addressable memory area (`0x20-0x2F` and all SFR addresses that start on `0x0` and `0x08`) in the internal data memory space. The `__bit` memory attribute can access individual bits in this area, whereas the `__bdata` attribute can access 8 bits with the same instruction.

sfr

The sfr memory covers the 128 upper bytes (`0x80-0xFF`) of the internal data memory space and is accessed using direct addressing. Standard peripheral units are defined in device-specific I/O header files with the filename extension `h`. For more details about these files, see *Header files for I/O*, page 4.

Use the `__sfr` memory attribute to define your own SFR definitions.

MEMORY TYPES FOR EXTERNAL DATA MEMORY SPACE

In the external data memory space, the following memory types are available:

- xdata
- pdata
- ixdata
- far
- huge
- Memory types for ROM memory in the external data memory space.

xdata

The xdata memory type refers to the 64-Kbyte memory area (`0x0–0xFFFF`) in the external data memory space. Use the `__xdata` memory attribute to place an object in this memory area, which will be accessed using `MOVX A, @DPTR`. This memory type is default in the Large and Generic data models.

pdata

The pdata memory type refers to a 256-byte area placed anywhere in the memory range `0x0–0xFFFF` of the external data memory space. Use the `__pdata` memory attribute to place an object in this memory area. The object which will be accessed using `MOVX A, @Ri`, which is more efficient compared to using the xdata memory type.

ixdata

Some devices provide on-chip xdata (external data) memory that is accessed faster than normal external data memory.

If available, this on-chip data memory is placed anywhere in the memory range `0x0–0xFFFF` of the external data memory space. Use the `__ixdata` memory attribute to access objects in this on-chip memory.

If used, this memory is enabled in the system startup code (`cstartup.s51`). You should verify that it is set up according to your requirements.

far

The far memory type refers to the whole 16-Mbyte memory area (`0x0–0xFFFFFF`) in the external data memory space. Use the `__far` memory attribute to place an object in this memory area, which will be accessed using `MOVX`. This memory type is only available when the Far data model is used, and in that case it is used by default.

Note that the size of such an object is limited to 64 Kbytes-1. For details about placement restrictions, see Table 42, *Data pointers*, page 222.

huge

The huge memory type refers to the whole 16-Mbyte memory area (`0x0–0xFFFFFF`) in the external data memory space. Use the `__huge` memory attribute to place an object in this memory area, which will be accessed using `MOVX`.

The drawback of the huge memory type is that the code generated to access the memory is larger and slower than that of any of the other memory types. In addition, the code uses more processor registers, which may force local variables to be stored on the stack rather than being allocated in registers.

Memory types for ROM memory in the external data memory space

Some devices provide ROM memory in the external data memory space that is accessed in the same way as other external data memory. Depending on where in the 16-Mbyte address space this ROM memory is available, different memory types are associated with the ROM memory:

Memory type	Attribute	Address range	Comments
Xdata ROM	`__xdata_rom`	`0x0-0xFFFF`	The size of such an object is limited to 64 Kbytes-1, and it cannot cross a 64-Kbyte physical segment boundary.
Far ROM	`__far_rom`	`0x0-0xFFFFFF`	The size of such an object is limited to 64 Kbytes-1. For details about placement restrictions, see Table 42, *Data pointers*, page 222.
Huge ROM	`__far_rom`	`0x0-0xFFFFFF`	The code generated to access the memory is larger and slower than that of any of the other memory types. In addition, the code uses more processor registers, which may force local variables to be stored on the stack rather than being allocated in registers.

Table 8: Memory types for ROM memory in external data memory space

Use this memory for constants and strings, as this memory can only be read from not written to.

MEMORY TYPES FOR CODE MEMORY SPACE

In the code data memory space, the following memory types are available:

- code
- far code
- huge code.

For information about placing functions in the code memory space, see *Code models for function storage*, page 47.

code

The code memory type refers to the 64-Kbyte memory area (`0x0-0xFFFF`) in the code memory space. Use the `__code` memory attribute to place constants and strings in this memory area, which will be accessed using `MOVC`.

far code

The far code memory type refers to the whole 16-Mbyte memory area (`0x0-0xFFFFFF`) in the code memory space. Use the `__far_code` memory attribute to place constants and strings in this memory area, which will be accessed using `MOVC`.

Note that the size of such an object is limited to 64 Kbytes-1. For details about placement restrictions, see Table 42, *Data pointers*, page 222.

This memory type is only available when the Far data model is used.

huge code

The huge code memory type refers to the whole 16-Mbyte memory area (`0x0-0xFFFFFF`) in the external data memory space. Use the `__huge_code` memory attribute to place an object in this memory area, which will be accessed using `MOVC`.

The drawback of the huge memory type is that the code generated to access the memory is larger and slower than that of any of the other memory types. In addition, the code uses more processor registers, which may force local variables to be stored on the stack rather than being allocated in registers.

This memory type is only available when the Far data model is used.

USING DATA MEMORY ATTRIBUTES

The 8051 IAR C/C++ Compiler provides a set of *extended keywords*, which can be used as *data memory attributes*. These keywords let you override the default memory type for individual data objects, which means that you can place data objects in other memory areas than the default memory. This also means that you can fine-tune the access method for each individual data object, which results in smaller code size.

The keywords are only available if language extensions are enabled in the 8051 IAR C/C++ Compiler.

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

Use the `-e` compiler option to enable language extensions. See *-e*, page 200 for additional information.

For reference information about each keyword, see *Descriptions of extended keywords*, page 242.

Syntax

The keywords follow the same syntax as the type qualifiers `const` and `volatile`. The memory attributes are *type attributes* and therefore they must be specified both when variables are defined and in the declaration, see *General syntax rules for extended keywords*, page 237.

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

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

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

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

Type definitions

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

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

and

```
__idata char b;
char __idata *bp;
```

POINTERS AND MEMORY TYPES

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

In the 8051 IAR C/C++ Compiler, a pointer also points to some type of memory. The memory type is specified using a keyword before the asterisk. For example, a pointer that points to an integer stored in idata memory is declared by:

```
int __idata * p;
```

Note 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 xdata memory (assuming the Large data model). Like `p`, `p2` points to a character in idata memory.

```
char __idata * __xdata p2;
```

Differences between pointer types

A pointer must contain information needed to specify a memory location of a certain memory type. In the 8051 IAR C/C++ Compiler, a smaller pointer can be implicitly converted to a pointer of a larger type if they both point to the same type of memory. For example, a `__pdata` pointer can be implicitly converted to an `__xdata` pointer. A pointer cannot be implicitly converted to a pointer that points to an incompatible memory area (that is, a code pointer cannot be implicitly converted to a data pointer and vice versa) and a larger pointer cannot be implicitly converted to a smaller pointer if it means that precision is being lost.

Whenever possible, pointers should be declared without memory attributes. For example, the functions in the standard library are all declared without explicit memory types.

STRUCTURES AND MEMORY TYPES

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

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

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

The following declaration is incorrect:

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

MORE EXAMPLES

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

`int a;`	A variable defined in default memory determined by the data model in use.

`int __xdata b;`	A variable in xdata memory.
`__idata int c;`	A variable in idata memory.
`int * d;`	A pointer stored in default memory. The pointer points to an integer in default memory.
`int __xdata * e;`	A pointer stored in default memory. The pointer points to an integer in xdata memory.
`int __xdata * __idata f;`	A pointer stored in idata memory pointing to an integer stored in xdata memory.
`int __idata * myFunction( int __xdata *);`	A declaration of a function that takes a parameter which is a pointer to an integer stored in xdata memory. The function returns a pointer to an integer stored in idata memory.

C++ and memory types

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

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

All restrictions that apply to the default pointer type also apply to the `this` pointer. This means that it must be possible to convert a pointer to the object to the default pointer type. Also note that for non-static member functions—unless class memory is used, see *Classes*, page 151—the `this` pointer will be of the default data pointer type.

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

Example

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

```
// The class declaration (placed in a header file):
class MyClass
{
public:
  int alpha;
  int beta;

  __pdata static int gamma;
};

// Definitions needed (should be placed in a source file):
__pdata int MyClass::gamma;

// A variable definition:
__xdata MyClass delta;
```

Constants and strings

The placement of constants and strings can be handled in one of three ways:

- Constants and strings are copied from ROM (non-volatile memory) to RAM at system initialization

  Strings and constants will be handled in the same way as initialized variables. This is the default behavior and all prebuilt libraries delivered with the product use this method. If the application only uses a small amount of constants and strings and the microcontroller does not have non-volatile memory in the external data memory space, this method should be selected. Note that this method requires space for constants and strings in both non-volatile and volatile memory.
- Constants are placed in ROM (non-volatile memory) located in the external data memory space

  Constants and strings are accessed using the same access method as ordinary external data memory. This is the most efficient method but only possible if the microcontroller has non-volatile memory in the external data memory space. To use this method, you should explicitly declare the constant or string with the memory attribute `__xdata_rom`, `__far_rom`, or `__huge_rom`. This is also the default behavior if the option `--place_constants=data_rom` has been used. Note that this option can be used if one of the data models Far, Generic, or Large is used.

- Constants and strings are located in code memory and are not copied to data memory.

  This method occupies memory in the external data memory space. However, constants and strings located in code memory can only be accessed through the pointers `__code`, `__far_code`, `__huge_code`, or `__generic`. This method should only be considered if a large amount of strings and constants are used and neither of the other two methods are appropriate. There are some complications when using the runtime library together with this method, see *Placing constants and strings in code memory*.

Placing constants and strings in code memory

If you want to locate constants and strings in the code memory space, you have to compile your project with the option `--place_constants=code`. It is important to note that runtime library functions that take constant parameters as input, such as `sscanf` and `sprintf`, will still expect to find these parameters in data memory rather than in code memory.

Auto variables—stack and static overlay

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

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

CHOOSING A CALLING CONVENTION

The 8051 IAR C/C++ Compiler provides six different *calling conventions* that control how memory is used for parameters and locally declared variables. You can specify a default calling convention or you can explicitly declare the calling convention for each individual function.

The following table lists the available calling conventions:

Calling convention	Function attribute	Stack pointer	Description
Data overlay	`__data_overlay`	`--`	An overlay frame in data memory is used for local data and parameters.
Idata overlay	`__idata_overlay`	`--`	An overlay frame in idata memory is used for local data and parameters.
Idata reentrant	`__idata_reentrant`	`SP`	The idata stack is used for local data and parameters.
Pdata reentrant	`__pdata_reentrant`	`PSP`	An emulated stack located in pdata memory is used for local data and parameters.
Xdata reentrant	`__xdata_reentrant`	`XSP`	An emulated stack located in xdata memory is used for local data and parameters.
Extended stack reentrant	`__ext_stack_reentrant`	`ESP:SP`	The extended stack is used for local data and parameters.

Table 9: Calling conventions

Only the reentrant models adhere strictly to ANSI C. The overlay calling conventions do not allow recursive functions and they are not reentrant, but they adhere to ANSI C in all other respects.

Choosing a calling convention also affects how a function calls another function. The compiler handles this automatically, but if you write a function in the assembler language you must know where and how its parameter can be found and how to return correctly. For detailed information, see *Calling convention*, page 131.

Specifying a default calling convention

You can choose a default calling convention by using the command line option `--calling_convention=`*`convention`*, see *--calling_convention*, page 192.

To specify the calling convention in the IAR Embedded Workbench IDE, choose **Project>Options>General Options>Target>Calling convention**.

Specifying a calling convention for individual functions

In your source code, you can declare individual functions to use, for example, the Idata reentrant calling convention by using the `__idata_reentrant` function attribute, for example:

```
extern __idata_reentrant void doit(int arg);
```

Calling conventions and matching data models

The choice of calling convention is closely related to the default pointer used by the application (specified by the data model). An application written without specific memory attributes or pragma directives will only work correctly if the default pointer can access the stack or static overlay area used by the selected calling convention.

The following table shows which data models and calling conventions you can combine without having to explicitly type-declare pointers and parameters:

Data model	Default pointer	Calling convention	Memory for stack or overlay frame
Tiny	Idata	Data overlay	Data
		Idata overlay	Idata
		Idata reentrant	Idata
Small	Idata	Data overlay	Data
		Idata overlay	Idata
		Idata reentrant	Idata
Large	Xdata	Pdata reentrant	Pdata
		Xdata reentrant	Xdata
		Extended stack reentrant	Xdata
Generic	Generic	Data overlay	Data
		Idata overlay	Idata
		Idata reentrant	Idata
		Pdata reentrant	Pdata
		Xdata reentrant	Xdata
		Extended stack reentrant	Xdata
Far	Far	Pdata reentrant	Pdata
		Xdata reentrant	Xdata
		Extended stack reentrant	Far

Table 10: Data models and calling convention

The compiler will not permit inconsistent default models, but if any functions have been explicitly declared to use a non-default calling convention, you might have to explicitly specify the pointer type when you declare pointers.

Example

```
__idata_reentrant void f(void)
{
  int x;
  int * xp = &x;
}
```

If the above example is compiled using the Small data model, the compilation will succeed. In the example, the pointer `xp` will be of the default pointer type, which is idata. An idata pointer can be instantiated by the address of the local variable `x`, located on the idata stack.

If on the other hand the application is compiled using the Large data model, the compilation will fail. The variable `x` will still be located on the idata stack, but the pointer `xp`, of the default type, will be an xdata pointer which is incompatible with an idata address. The compiler will generate the following error message:

```
Error[Pe144]: a value of type "int __idata *" cannot be used to
initialize an entity of type "int *"
```

Here, the `int *` pointer is of the default type, that is, it is in practice `int __xdata *`. However, if the pointer `xp` is explicitly declared as an `__idata` pointer, the application can be successfully compiled using the Large data model. The source code would then look like this:

```
__idata_reentrant void f(void)
{
  int x;
  int __idata * xp = &x;
}
```

Mixing calling conventions

Not all calling conventions can coexist in the same application, and passing local pointers and returning a `struct` can only be performed in a limited way if you use more than one calling convention.

Only one internal stack—the stack used by, for example, `PUSH` and `CALL` instructions—can be used at the same time. Most 8051 devices only support an internal stack located in IDATA. A notable exception is the extended devices which allow the internal stack to be located in XDATA if you use the extended stack option. This means that the idata reentrant and the extended stack reentrant calling conventions cannot be combined in the same application. Furthermore, the xdata reentrant calling convention cannot be combined with the extended stack reentrant calling convention, because there can be only one stack located in XDATA at the same time.

Mixing calling conventions can also place restrictions on parameters and return values. These restrictions only apply to locally declared pointers passed as parameters and when returning a `struct`-declared variable. The problem occurs if the default pointer (specified by the data model) cannot refer to the stack implied by the calling convention in use. If the default pointer can refer to an object located on the stack, the calls are not restricted.

Example

```
__xdata_reentrant void foo(int *ptr)
{
  *ptr = 20;
}

__idata_reentrant void bar(void)
{
  int value;

  foo(&value);
}
```

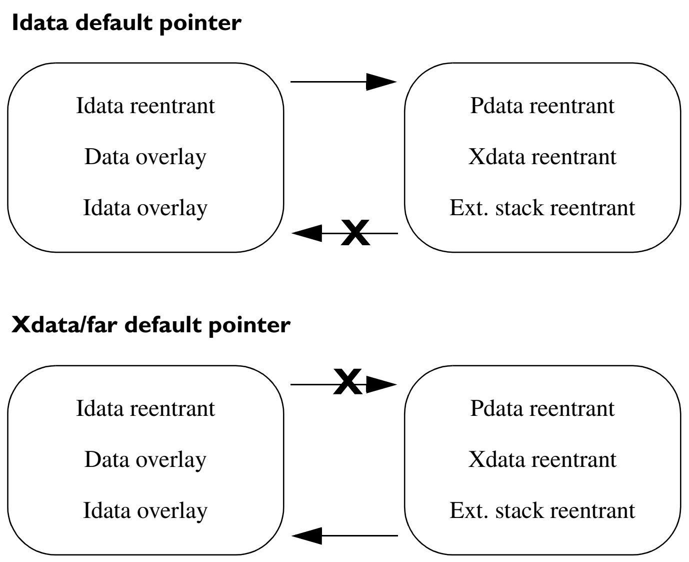
The application will be successfully compiled if the Small data model is used and the variable `value` will be located on the idata stack. The actual argument `&value`, referring to the variable `value`, will become an `__idata *` pointer when it refers to an object located on the idata stack. The formal argument to the function `foo` will be of the default pointer type, that is, `__idata`, which is the same type as the actual argument used when the function is called.

The same reasoning applies to return values of a structure type. The calling function will reserve space on the calling function's stack and pass a hidden parameter to the called function. This hidden parameter is a pointer referring to the location on the caller's stack where the return value should be located.

Applications are thus restricted in how functions using different calling convention can call each other. Functions using a stack which is reachable with the default pointer type can call all other functions regardless of the calling convention used by the called function.

Functions calling other functions

The following picture shows how functions can call each other:



The arrow means that a call is possible. The **X** means that no call is possible unless the source code has been explicitly type-declared using function attributes or pragma directives.

Figure 3: Function calls between different calling conventions

Thus, if an application is compiled using the Small data model, an `__idata_reentrant` function can call an `__xdata_reentrant` function. But an `__xdata_reentrant` function cannot call an `__idata_reentrant` function. However, all applications can be explicitly type-declared so that they work as intended.

Using the extended stack excludes the possibility to use the idata and xdata stacks; extended stack reentrant functions can only coexist with pdata reentrant functions. The extended stack is located in xdata memory and can be viewed in the same way as the xdata stack in the context of calling convention compatibility.

THE STACK

The stack can contain:

- Local variables and parameters not stored in registers
- Temporary results of expressions

- The return value of a function (unless it is passed in registers)
- Processor state during interrupts
- Processor registers that should be restored before the function returns (callee-save registers)
- Registers saved by the calling function (caller-save registers).

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

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

Advantages

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

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

Potential problems

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

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

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

STATIC OVERLAY

Static overlay is a system where local data and function parameters are stored at static locations in memory. Each function is associated with an overlay frame that has a fixed size and contains space for local variables, function parameters and temporary data.

Static overlay can produce very efficient code on the MCS-51 microcontroller because it has good support for direct addressing. However, the amount of directly accessible memory is limited, so static overlay systems are only suitable for small applications.

There is a problem with the static overlay system, though; it is difficult to support recursive and reentrant applications. In reentrant and recursive systems, several instances of the same function can be alive at the same time, so it is not enough to have one overlay frame for each function; instead the compiler must be able to handle multiple overlay frames for the same function. Therefore, the static overlay system is restricted and does not support recursion or reentrancy.

For information about the function directives used by the static overlay system, see the *8051 IAR Assembler Reference Guide*.

Dynamic memory on the heap

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

Note: Only MCS-51 devices with external memory can have a heap. When the CLIB library is used, the heap can be located in xdata and far memory. When the DLIB library is used, the heap can also be located in huge memory. The use of heaps is depending on the selected data model. For more information about this, see *The heap*, page 79.

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

In C++, there is a special keyword, `new`, designed to allocate memory and run constructors. Memory allocated with `new` must be released using the keyword `delete`.

Potential problems

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

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

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

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

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

Alternative memory allocation functions

Because the internal memory is very limited on the MCS-51 microcontroller, the MCS-51 IAR C/C++ Compiler only supports a heap located in the external data memory space. To fully support this heap when you use the 8051 IAR CLIB runtime library, additional functions are included in the library:

```
void __xdata *__xdata_malloc(unsigned int)
void __xdata *__xdata_realloc(void __xdata *, unsigned int)
void __xdata *__xdata_calloc(unsigned int, unsigned int)
void __xdata_free(void __xdata *)
void __far *__far_malloc(unsigned int)
void __far *__far_realloc(void __far *, unsigned int)
void __far *__far_calloc(unsigned int, unsigned int)
void __far_free(void __far *)
```

It is recommended that these alternative functions are used instead of the standard C library functions `malloc`, `calloc`, `realloc`, and `free`. The `__xdata` versions of the functions are available when 16-bit data pointers (DPTRs) are used and the `__far` versions when 24-bit data pointers are used.

The standard functions can be used together with the Large, Generic and Far data models; they will call the corresponding `__xdata` or `__far` alternative function, depending on the size of the DPTRs you are using. However, if the Tiny or Small data model is used, the standard `malloc`, `calloc`, `realloc`, and `free` functions cannot be used at all. In these cases, you must explicitly use the corresponding alternative functions to use a heap in external memory.

The difference between the alternative `__xdata` and `__far` memory allocation functions and the standard functions is the pointer type of the return value and the pointer type of the arguments. The functions `malloc`, `calloc`, and `realloc` return a pointer to the allocated memory area and the `free` and `realloc` functions take a pointer argument to a previously allocated area. These pointers must be a pointer of the same type as the memory that the heap is located in, independent of the default memory and pointer attributes.

Note: The corresponding functionality is also available in the DLIB runtime environment.

Virtual registers

The 8051 IAR C/C++ Compiler uses a set of virtual registers—located in data memory—to be used like any other registers. A minimum of 8 virtual registers are required by the compiler, but as many as 32 can be used. A larger set of virtual registers makes it possible for the compiler to allocate more variables into registers. However, a larger set of virtual registers also requires a larger data memory area. In the Large data model you should probably use a larger number of virtual registers, for example 32, to help the compiler generate better code.

In the IAR Embedded Workbench IDE, choose **Project>Options>General Options>Target>Number of virtual registers**.

On the command line, use the option `--nr_virtual_regs` to specify the number of virtual registers. See *--nr_virtual_regs*, page 210.

You must also allocate the data memory space required for the virtual registers in the linker command file by setting the constant `_NR_OF_VIRTUAL_REGISTERS` to the same number of registers as set by the `--nr_virtual_regs` compiler option.

Note: The allocated memory space must be *at least* equal in size to the number of virtual registers used. For example, the linker command file can contain the following lines if 32 virtual registers are used:

```
-D_NR_OF_VIRTUAL_REGISTERS=0x20
-Z(DATA)VREG+_NR_OF_VIRTUAL_REGISTERS=08-7F
```

THE VIRTUAL BIT REGISTER

The 8051 IAR C/C++ Compiler reserves one byte of bit-addressable memory to be used internally, for storing locally declared `bool` variables. The virtual bit register can be located anywhere in the bit-addressable memory area (`0x20`–`0x2F`). It is recommended that you use the first or last byte in this area.

Specify the location of the virtual bit register in the linker command file by defining `?VB` to the appropriate value, for instance:

```
-D?VB=2F
```

Functions

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

For details about extensions related to banked functions, see the chapter *Banked functions*.

Function-related extensions

In addition to the ISO/ANSI C standard, the 8051 IAR C/C++ Compiler provides several extensions for writing functions in C. Using these, you can:

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

The compiler supports this by means of compiler options, extended keywords, pragma directives, and intrinsic functions.

For more information about optimizations, see *Writing efficient code*, page 173. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

Code models for function storage

By means of *code models*, the 8051 IAR C/C++ Compiler supports placing functions in a default part of memory, or in other words, use a default size of the function address. Technically, the code models control the following:

- The default memory range for storing the function, which implies a memory attribute
- The maximum module size
- The maximum application size.

The compiler supports four code models. If you do not specify a code model, the compiler will by default use a code model depending on your choice of core. Your project can only use one code model at a time, and the same model must be used by all user modules and all library modules.

The following code models are available:

Code model	Default for core	Pointer size	Description
Near	Plain	2 bytes	Supports up to 64 Kbytes ROMable code, can access the entire 16-bit address space
Banked	--	2 bytes	Supports banked function calls, see *Code models for banked systems*, page 56. Functions can be explicitly placed in the root bank by using the `__near_func` memory attribute, see *__near_func*, page 253.
Banked Extended2	Extended2	3 bytes	Supports banked function calls, see *Code models for banked systems*, page 56.
Far	Extended1	3 bytes	Supports true 24-bit calls; only to be used for devices with extended code memory and true 24-bit instructions

Table 11: Code models

To choose a code model in the IAR Embedded Workbench IDE, choose **Project>Options>General Options>Code model**.

Use the `--code_model` option to specify the code model for your project; see *--code_model*, page 193.

Pointers with different sizes have restrictions in implicit and explicit casts between pointers and between pointers and integer types. For details about the restrictions, see *Casting*, page 223.

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

Primitives for interrupts, concurrency, and OS-related programming

The 8051 IAR C/C++ Compiler provides the following primitives related to writing interrupt functions, concurrent functions, and OS-related functions:

- The extended keywords `__interrupt`, `__task`, and `__monitor`
- The pragma directives `#pragma vector`

- The intrinsic functions `__enable_interrupt`, `__disable_interrupt`, `__get_interrupt_state`, and `__set_interrupt_state`.

INTERRUPT FUNCTIONS

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

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

The MCS-51 microcontroller supports many interrupt sources. For each interrupt source, an interrupt routine can be written. Each interrupt routine is associated with a vector number which is specified in the MCS-51 microcontroller documentation from the chip manufacturer. The interrupt vector is the offset into the interrupt vector table. If you want to handle several different interrupts using the same interrupt function, you can specify several interrupt vectors. For the MCS-51 microcontroller, the interrupt vector table always starts at the address `0x03`.

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

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

```
#pragma vector=0xIE0_int  /* Symbol defined in I/O header file */
__interrupt void my_interrupt_routine(void)
{
  /* Do something */
}
```

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

If a vector is specified in the definition of an interrupt function, the processor interrupt vector table is populated. It is also possible to define an interrupt function without a vector. This is useful if an application is capable of populating or changing the interrupt vector table at runtime. See the chip manufacturer's MCS-51 microcontroller documentation for more information about the interrupt vector table.

Register banks

The basic MCS-51 microcontroller supports four register banks. If you have specified a register bank to be used by an interrupt, the registers `R0`–`R7` are not saved on the stack when the interrupt function is entered. Instead the application switches to the bank you have specified and then switches back when the interrupt function exits. This is a useful way to speed up important interrupts.

One register bank, usually bank `0`, is used internally by the runtime system of the compiler. You can change the default register bank used by the runtime system by redefining the value of the symbol `REGISTER_BANK` in the linker command file. Other register banks can be associated with an interrupt routine by using the `#pragma register_bank` directive.

Note: The bank used by the runtime system of the compiler must not be used for interrupts, and different interrupts that can interrupt each other must not use the same register bank.

Example

```
#pragma register_bank=1
#pragma vector=0xIE0_int
__interrupt void my_interrupt_routine(void)
{
  /* Do something */
}
```

In the linker command file it can look like this:

```
-D?REGISTER_BANK=1          /* Default register bank (0,1,2,3) */
-D_REGISTER_BANK_START=08    /* Start address for default
                                 register bank (00,08,10,18) */
-Z(DATA)REGISTERS+8=_REGISTER_BANK_START
```

If you use a register bank together with interrupt routines, the space occupied by the register bank cannot be used for other data.

MONITOR FUNCTIONS

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

To define a monitor function, you can use the `__monitor` keyword. For reference information, see *__monitor*, page 252.

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

Example of implementing a semaphore in C

In the following example, a 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 from simultaneously using resources that can only be used by one process at a time, for example a UART. The `__monitor` keyword assures that the lock operation is atomic; in other words it cannot be interrupted.

```
/* When the_lock is non-zero, someone owns the lock. */
static volatile 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 semaphore:

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

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

Example of implementing a semaphore in C++

In C++, it is common to implement small methods with the intention that they should be inlined. However, the 8051 IAR C/C++ Compiler does not support inlining of functions and methods that are declared using the `__monitor` keyword.

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

```
#include <intrinsics.h>

volatile long tick_count = 0;

/* Class for controlling critical blocks */
class Mutex
{
public:
  Mutex ()
  {
    _state = __get_interrupt_state();
    __disable_interrupt();
  }

  ~Mutex ()
  {
    __set_interrupt_state(_state);
  }

private:
  __istate_t _state;
};


void f()
{
  static long next_stop = 100;
  extern void do_stuff();
  long tick;

  /* A critical block */
  {
    Mutex m;
    /* Read volatile variable 'tick_count' in a safe way
       and put the value in a local variable */

    tick = tick_count;
  }

  if (tick >= next_stop)
  {
    next_stop += 100;
    do_stuff();
  }
}
```

C++ AND SPECIAL FUNCTION TYPES

C++ member functions can be declared using special function types. However, there is one restriction: interrupt member functions must be static. When calling a non-static member function, it must be applied to an object. When an interrupt occurs and the interrupt function is called, there is no such object available.

Banked functions

This chapter introduces the banking technique. It is important to know when to use it, what it does, and how it works. More specifically, this chapter contains the following:

- Introduction to the banking system, which introduces the code models available for banked systems, exemplifies standard bank memory layouts, and describes how to set up the compiler and linker
- Writing source code for banked memory, which describes how to write and partition source code and interrupt service routines for banked applications
- Bank switching, which describes this mechanism in detail
- Downloading to memory, which discusses linker output formats suitable for handling banked addresses and methods for downloading to multiple PROMs
- Debugging banked applications, which describes how to set up the C-SPY debugger and gives some hints for debugging.

Note that when you read this chapter, you should be familiar with the other concepts described in *Part 1. Using the compiler*.

Introduction to the banking system

If you are using an MCS-51 microcontroller with a natural address range of 64 Kbytes of memory, it has a 16-bit addressing capability. *Banking* is a technique for extending the amount of memory that can be accessed by the processor beyond the limit set by the size of the natural addressing scheme of the processor. In other words, more code can be accessed.

Banked memory is used in projects which require such a large amount of executable code that the natural 64 Kbytes address range of a basic MCS-51 microcontroller is not sufficient to contain it all.

CODE MODELS FOR BANKED SYSTEMS

The 8051 IAR C/C++ Compiler provides two code models for banking your code allowing for up to 16 Mbytes of ROM memory:

- The Banked code model

  allows the code memory area of the MCS-51 microcontroller to be extended with up to 256 banks of additional ROM memory. Each bank can be up to 64 Kbytes, minus the size of a root bank.
- The Banked extended2 code model

  allows the code memory area of the MCS-51 microcontroller to be extended with up to 16 banks of additional ROM memory. Each bank can be up to 64 Kbytes, You do not need to reserve space for a traditional root bank.

Your hardware must provide these additional physical memory banks, as well as the logic required to decode the high address bits which represent the *bank number*.

Because a basic MCS-51 microcontroller cannot handle more than 64 Kbytes of memory at any single time, the extended memory range introduced by banking implies that special care must be taken. Only one bank at a time is visible for the CPU, and the remaining banks must be brought into the 64 Kbytes address range before they can be used.

THE MEMORY LAYOUT FOR THE BANKED CODE MODEL

You can place the banked area anywhere in code memory, but there must always be a *root area* for holding the runtime environment code and relay functions for bank switches.

The following illustration shows an example of a MCS-51 banked code memory layout:

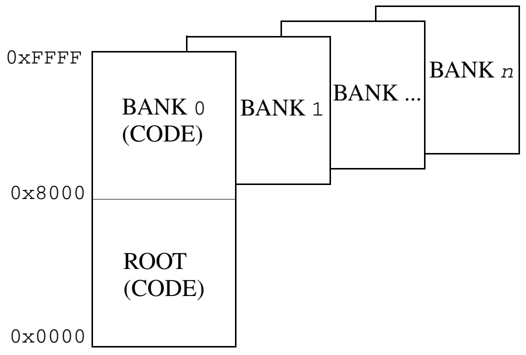


Figure 4: Banked code memory layout (example)

It is practical to place the root area at address 0 and upwards. For information about how to explicitly place functions in the root area, see *Banked versus non-banked function calls*, page 60.

THE MEMORY LAYOUT FOR THE BANKED EXTENDED2 CODE MODEL

Devices based on the Mentor Graphics M8051EW core—which implies that you are using the `--core=extended2` compiler option—use a different banked mechanism than the classic MCS-51 microcontroller.

The following illustration shows an example of a MCS-51 banked code memory layout:

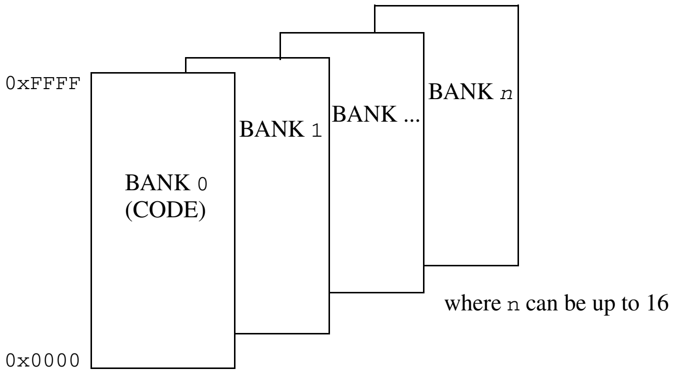


Figure 5: Banked code memory layout for the extended2 core (example)

Bank 0 holds the runtime environment code.

SETTING UP THE COMPILER FOR BANKED MODE

To specify the banked code model in the IAR Embedded Workbench IDE, choose **Project>Options>General Options>Target>Code model**. Choose either **Banked** or, if you are using the **Exteded2** core option, **Banked extended2**.

To compile your modules for banked mode, use the compiler option with the appropriate parameter:

```
--code_model={banked|banked_ext2}
```

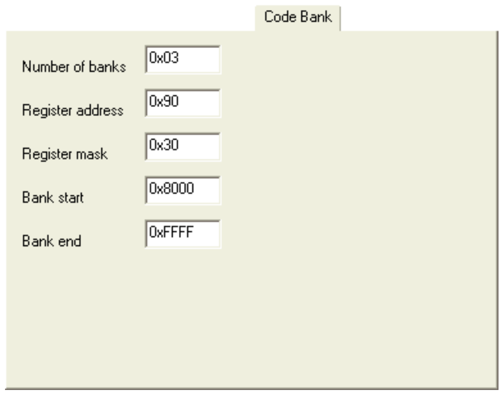
Note: The Banked code model is available when using the core Plain (classic 8051) and the Banked extended 2 code model is available when using the core Extended2 (for the Mentor Graphics M8051EW core).

SETTING UP THE LINKER FOR BANKED MODE

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

The simplest way to set up the code banks for the linker is to use the IAR Embedded Workbench IDE. However, if your hardware memory configuration is very unusual, you can instead use the linker command file to set up the bank system for the linker.

To set code bank options in the IAR Embedded Workbench IDE; choose **Project>Options>General Options>Code Bank**.



Number of banks	0x03
Register address	0x90
Register mask	0x30
Bank start	0x8000
Bank end	0xFFFF

Figure 6: Code Bank setup

Use the text boxes to set up for banking according to your memory configuration.

1 Specify the number of banks in the **Number of banks** text box.

2 The bank-switching routine is based on an SFR port being used for the bank switching. By default, `P1` is used. To set up a different SFR port, add the required port in the **Register address** text box.

3 If the entire SFR is not used for selecting the active bank, specify the appropriate bitmask in the **Register mask** text box to indicate the bits used. For example, specify `0x03` if only bits 0 and `1` are used.

4 In the **Bank start** text box, type the start address of the banked area. Correspondingly, type the end address in the **Bank End** text box.

In the predefined linker command file, a set of symbols is predefined:

```
-D?CBANK=90       /* Most significant byte of a banked address */
-D_FIRST_BANK_ADDR=0x10000
-D_NR_OF_BANKS=0x10
-D_CODEBANK_START=3500             /* Start of banked segments */
-D_CODEBANK_END=FFFF                /* End for banked segments */
-D?CBANK_MASK=FF      /* Bits used for toggling bank, set to 1 */
```

If the values are not appropriate, you can simply change them to match your hardware memory configuration. However, if the symbols are not appropriate for your configuration, you have to adapt them according to your needs.

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

Writing source code for banked memory

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

C/C++ LANGUAGE CONSIDERATIONS

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

BANK SIZE AND CODE SIZE

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

However, some optimizations require that all segment parts produced from banked functions in the same module (source file) must be linked together as a unit. In this case, the combined size of *all* banked functions in the same module must be smaller than the bank size.

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

If you need to specify the location of any code individually, you can rename the code segment for each function to a distinct name that will allow you to refer to it individually. To assign a function to a specific segment, use the `@` operator or the `#pragma location` directive. See *Data and function placement in segments*, page 168.

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

BANKED VERSUS NON-BANKED FUNCTION CALLS

In the Banked code model, differentiating between the non-banked versus banked function calls is important because non-banked function calls are faster and take up less code space than banked function calls. Therefore, it is useful to be familiar with which types of function declarations that result in non-banked function calls.

Note: In the Banked extended2 code model, all code is banked which means there is no need to differentiate between non-banked versus banked function calls

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

Assuming that you are using the Banked code model, there are two function call sequences:

- Non-banked function calls: The return address and new execution address are 16-bit values. Functions declared `__near_func` will have non-banked function calls.
- Banked function calls: The return address and new execution address are 24-bit (3 bytes) values (default in the Banked code model)

In the Banked code model, all untyped functions will be located in banked memory. However, it is possible to explicitly declare functions to be non-banked by using the `__near_func` memory attribute. Such functions will not generate banked calls and will be located in the `NEAR_CODE` segment instead of in the `BANKED_CODE` segment. The `NEAR_CODE` segment must be located in the root bank when no banked call is produced.

It can often be a good idea to place frequently called functions in the root bank, to reduce the overhead of the banked call and return.

Example

In this example you can see how the banked function `f1` calls the non-banked function `f2`:

```
__near_func void f2(void) /* Non-banked function */
{
  / * code here … */
}

void f1(void) /* Banked function in the Banked code model */
{
  f2();
}
```

The actual call to `f2` from within `f1` is performed in the same way as an ordinary function call (`LCALL`).

Note: There is a `__banked_func` memory attribute available, but you do not need to use it explicitly. The attribute is available for compiler internal use only.

CODE THAT CANNOT BE BANKED

In the Banked code model, code banking is achieved by dividing the address space used for program code into two parts: *non-banked* and *banked* code. In Figure 5, *Banked code memory layout for the extended2 core (example)*, page 57, the part that contains the non-banked code is referred to as the *root bank*. There must always be a certain amount of code that is non-banked. For example, interrupt service routines must always be available, as interrupts can occur at any time.

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

- The cstartup routine, located in the `CSTART` segment
- Interrupt vectors, located in the `INTVEC` segment
- Interrupt service routines, located in the `NEAR_CODE` segment
- Segments containing constants. These are all segments ending with `_AC`, `_C`, and `_N`, see the chapter *Segment reference*
- Segments containing initial values for initialized variables can be located in the root bank or in bank 0 but in no other bank. These are all segments ending with `_ID`, see the chapter *Segment reference*
- The assembler-written routines included in the runtime library. They are located in the `RCODE` segment
- Relay functions, located in the `BANK_RELAYS` segment
- Bank-switching routines, that is, those routines that will perform the call and return to and from banked routines. They are located in the `CSTART` segment.

Banking is not supported for functions using one of the overlay calling conventions (`__data_overlay` or `__idata_overlay`) or for far functions (`__far_func`) because only one of the function type attributes `__far_func` and `__banked_func` can be used as keywords in the system at the same time.

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

Note: Even though interrupt functions cannot be banked, the interrupt routine itself can call a banked function.

Calling banked functions from assembler language

In an assembler language program, calling a C/C++ function located in another bank requires using the same calling convention as the compiler. For information about this calling convention, see *Calling convention*, page 131. To generate an example of a banked function call, use the technique described in the section *Calling assembler routines from C*, page 128.

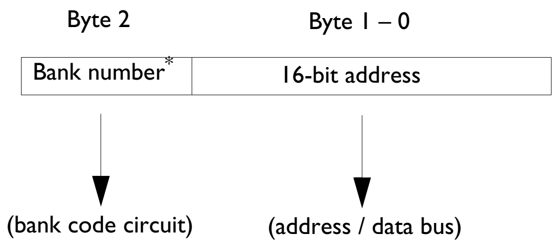
If you are writing banked functions using assembler language, you must also consider the calling convention.

Bank switching

For banked systems written in C or C++, you normally do not need to consider the bank switch mechanism, as the compiler handles this for you. However, if you write banked functions in assembler language it might be necessary to pay attention to the bank switch mechanism in your assembler functions. Also, if you want to use a completely different solution for bank switching, you must implement your own bank-switching routine.

ACCESSING BANKED CODE

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



* For the Banked extended2 code model, the upper four bits hold the number of the
whereas the lower four bits hold the number of the next bank.

Figure 7: The 3-byte banked address

For further details of pointer representation, see *Pointer types*, page 221.

The default bank-switching code is available in the supplied assembler language source file `iar_banked_code_support.s51`, which you can find in the directory `8051\src\lib`.

The bank-switching mechanism differs between the two code models.

BANK SWITCHING IN THE BANKED CODE MODEL

The default bank-switching routine is based on an SFR port (`P1`) being used for the bank switching. When a function—the *caller*—calls another function—the *callee*—a bank switch is performed. The compiler does not know in which bank a function will be placed by the linker.

More precisely, the following actions are performed:

- The caller function performs an `LCALL` to a relay function located in the root bank. The return address—the current `PC` (the program counter)—is pushed on the idata stack.
- The relay function performs an `LCALL` to a dispatch function. The relay function contains the address of the callee.
- The call to the dispatch function implies that the current bank number (caller's bank number) is saved on the idata stack. The next bank number is saved in `P1` as the current bank. The address of the callee is moved to `PC`, which means the execution moves to the callee function. In other words, a bank switch has been performed.

Figure 8, *Bank switching in the Banked code model* illustrates the actions:

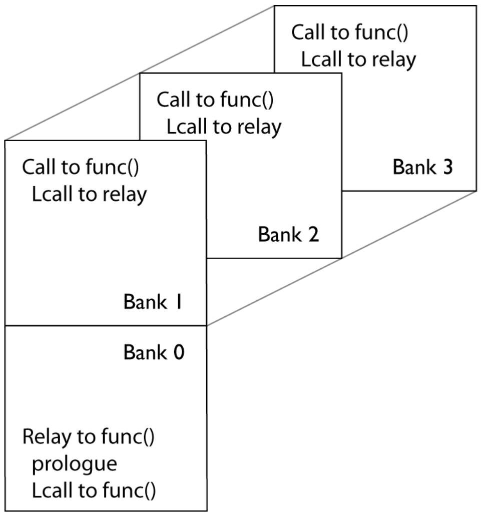


Figure 8: Bank switching in the Banked code model

When the callee function has executed, it performs an `?LCALL` to `?BRET`. In `?BRET`, the bank number of the caller function is popped from the idata stack and then a `RET` instruction is executed. The execution is now back in the caller function.

BANK SWITCHING IN THE BANKED EXTENDED2 CODE MODEL

The default bank-switching uses the `MEX` register, the memory extension stack, and the `MEXSP` register as the stack pointer.

When a function—the *caller*—calls another function—the *callee*—a bank switch is performed. Before the bank switch, the following actions are performed in the caller function:

- The high byte of the 3-byte pointer—described in Figure 7, *The 3-byte banked address*—is placed in the `MEX.NB` register (the next bank register).
- An `LCALL` is performed, which implies that the following steps are performed in hardware:
  - The register `MEX.CB` (the current bank register) is placed on the memory extension stack.
  - The return address, that is the current `PC`, is placed on the idata stack.
  - The next bank `MEX.NB` is copied to the current bank `MEX.CB`.

- The two low bytes of the 3-byte pointer is copied to `PC`, which means the execution moves to the callee function. In other words, a bank switch has been performed.

When the callee function has executed, the `RET` instruction performs the bank-switching procedure in reversed order, which means the execution is back in the caller function.

Note: The memory extension stack is limited to 128 bytes, which means the function call depth cannot exceed that limit.

MODIFYING THE DEFAULT BANK-SWITCHING ROUTINE

The default bank-switching code is available in the supplied assembler language source file `iar_banked_code_support.s51`, which you can find in the directory `8051\src\lib`.

The bank-switching routine is based on an SFR port being used for the bank switching. By default `P1` is used. The SFR port is defined in the linker command file by the line:

`-D?CBANK=`*`PORTADDRESS`*

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

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

Downloading to memory

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

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

When you download the code into physical memory, special considerations might be needed.

For instance, assume a banked system with two 32-Kbytes banks of ROM starting at `0x8000`. If the banked code exceeds 32 Kbytes in size, when you link the project the result will be a single output file where the banked code starts at `0x8000` and crosses the upper bank limit. A modern EPROM programmer does not require downloading one file to one EPROM at a time; it handles the download automatically by splitting the file and downloading it. However, older types of programmers do not always support relocation, or are unable to ignore the high byte of the 3-byte address. This means that you have to edit the file manually to set the high bytes of each address to `0` so that the programmer can locate them properly.

Debugging banked applications

For the Banked code model, the C-SPY debugger supports banked mode debugging. To set banked mode debugging options in the IAR Embedded Workbench IDE, choose **Project>Options**, select the **General Options** category, and click the **Code Bank** tab. Type the appropriate values for the following options:

- **Register address** specifies the SFR address used as bank register
- **Bank start** specifies the bank start address
- **Bank end** specifies the bank end address
- **Bank mask** specifies the bits used for selecting the active bank
- **Number of banks** specifies the number of banks available on the hardware.

Banked mode debugging with other debuggers

If your emulator does not support banked mode, one common technique is to divide your source code in smaller parts that do not exceed the size of the bank. You can then compile, link, and debug each part using the Near or Far code model. Repeat this procedure for various groupings of functions. Then, when you actually test the final banked system on target hardware, many C/C++ programming-related issues will already have been resolved.

Placing code and data

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

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

Segments and memory

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

WHAT IS A SEGMENT?

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

Note: Here, ROM memory means all types of read-only memory including flash memory.

The 8051 IAR C/C++ Compiler has a number of predefined segments for different purposes. Each segment has a name that describes the contents of the segment, and a *segment memory type* that denotes the type of content. In addition to the predefined segments, you can define your own segments.

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

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

Segment memory type

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

By default, the 8051 IAR C/C++ Compiler uses only the following XLINK segment memory types:

Segment memory type	Description
`BIT`	For bit-addressable data placed in internal RAM
`DATA`	For data placed in internal RAM; address `0-75` is the same as for IDATA, address `80-FF` is the SFR area
`IDATA`	For data placed in internal RAM
`XDATA`	For data placed in external RAM
`CODE`	For executable code
`CONST`	For data placed in ROM

Table 12: XLINK segment memory types

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

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

Placing segments in memory

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

In particular, the linker command file specifies:

- The placement of segments in memory
- The maximum stack size
- The maximum heap size (only for the IAR DLIB runtime environment).

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

CUSTOMIZING THE LINKER COMMAND FILE

The `config` directory contains the following ready-made linker command files:

- `lnk51.xcl` is the basic, default linker command file, used on the command line
- `lnk51o.xcl` supports overlay calling conventions, used on the command line
- `lnk51e.xcl` supports the extended1 core, used on the command line
- `lnk51e2.xcl` supports the extended2 core, used on the command line
- `lnk51ew.xcl` is the linker command file used in the IAR Embedded Workbench IDE
- `lnk_base.xcl` defines a basic set of symbols necessary to link your application. All other linker command files include this file. Normally, you do not need to customizeS the `lnk_base.xcl` file.

The files contain the information required by the linker, and are ready to be used. The only change you will normally have to make to the supplied linker command file is to customize it so it fits the target system memory map. If, for example, your application uses additional external RAM, you need to add details about the external RAM memory area.

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

Range	Type
`0x0–0xFF`	Internal RAM
`0x0–0x7FFF`	External RAM
`0x0–0xFFFF`	ROM

Table 13: Memory layout of a target system (example)

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

Remember not to change the original file. We recommend that you make a copy of the file and place it in your working directory; modify the copy instead.

The contents of the linker command file

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

- The CPU used:

  ```
  -cx51
  ```

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

In the linker command file, all numbers are specified in hexadecimal format.

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

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

Using the -Z command for sequential placement

Use the -Z command when you need to keep a segment in one consecutive chunk, when you need to preserve the order of segment parts in a segment, or, more unlikely, when you need to put segments in a specific order.

The following illustrates how to use the `-Z` command to place the segment `MYSEGMENTA` followed by the segment `MYSEGMENTB` in `CONST` memory (that is, ROM) in the memory range `0x0-0x1FFF`.

```
-Z(CONST)MYSEGMENTA,MYSEGMENTB=0-1FFF
```

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

```
-Z(CONST)MYSEGMENTA=0-1FFF
-Z(CODE)MYCODE
```

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

```
-Z(CONST)MYSMALLSEGMENT=0-7FF
-Z(CONST)MYLARGESEGMENT=0-1FFF
```

Normally, when you place segments sequentially with the `-Z` option, each segment is placed entirety into one of the address ranges you have specified. If you use the modifier `SPLIT-`, each part of the segment is placed separately in sequence, allowing address gaps between different segment parts, for example:

```
-Z(SPLIT-XDATA)FAR_Z=10000-1FFFF,20000-2FFFF
```

In most cases, using packed segment placement (`-P`) is better. The only case where using `-Z(SPLIT-`*`type`*`)` is better is when the start and end addresses of the segment are important, for example, when the entire segment must be initialized to zero at program startup, and individual segment parts cannot be placed at arbitrary addresses.

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

Using the -P command for packed placement

The `-P` command differs from `-Z` in that it does not necessarily place the segments (or segment parts) sequentially. With `-P` it is possible to put segment parts into holes left by earlier placements.

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

```
-P(DATA)MYDATA=0-FF,1000-3FFF
```

If your application has an additional RAM area in the memory range `0xF000-0xF7FF`, you just add that to the original definition:

```
-P(DATA)MYDATA=0-FF,F000–F7FF,10000-3FFF
```

Note: Copy initialization segments—*`BASENAME`*`_I` and *`BASENAME`*`_ID`—must be placed using `-Z`.

Using the -P command for banked placement

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

```
// First some defines for the banks
-D_CODEBANK_START=8000
```

```
-D_CODEBANK_END=9FFF
-D?CBANK=90

-P(CODE)BANKED_CODE=[_CODEBANK_START–_CODEBANK_END]*4+10000
```

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

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

For more information about these symbols and how to configure for a banked system, see *Setting up the linker for banked mode*, page 58.

Data segments

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

To get a clear understanding about how the data segments work, you must be familiar with the different memory types and the different data models available in the 8051 IAR C/C++ Compiler. If you need to refresh these details, see the chapter *Data storage*.

STATIC MEMORY SEGMENTS

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

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

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

- The segment naming
- How the memory types correspond to segment groups and the segments that are part of the segment groups
- Restrictions for segments holding initialized data

- The placement and size limitation of the segments of each group of static memory segments.

Segment naming

The names of the segments consist of two parts—the segment group name and a *suffix*—for instance, `IDATA_Z`.

The names of the segment groups are derived from the memory type and the corresponding keyword, for example `IDATA` and `__idata`. The suffix indicates the category of declared data. The following table summarizes the memory types and the corresponding segment groups:

Memory type	Segment group	Segment memory type	Address range
Data	`DATA`	`DATA`	`0–7F`
SFR	`SFR`	`DATA`	`80-FF`
Idata	`IDATA`	`IDATA`	`0–FF`
Bdata	`BDATA`	`DATA`	`20–2F`
Bit	`BIT`	`BIT`	`0–FF`
Pdata	`PDATA`	`XDATA`	`0–FF`
Ixdata	`IXDATA`	`XDATA`	`0–FFFF`
Xdata	`XDATA`	`XDATA`	`0–FFFF`
Far	`FAR`	`XDATA`	`0–FFFFFF`
Huge	`HUGE`	`XDATA`	`0–FFFFFF`
Code	`CODE`	`CODE`	`0-FFFF`
Far code	`FAR_CODE`	`CODE`	`0-FFFFFF`
Huge code	`HUGE_CODE`	`CODE`	`0-FFFFFF`
Xdata ROM	`XDATA_ROM`	`CONST`	`0–FFFF`
Far ROM	`FAR_ROM`	`CONST`	`0–FFFFFF`
Huge ROM	`HUGE_ROM`	`CONST`	`0–FFFFFF`

Table 14: Memory types with corresponding memory groups

Some of the declared data is placed in non-volatile memory, for example ROM, and some of the data is placed in RAM. For this reason, it is also important to know the XLINK segment memory type of each segment. For more details about segment memory types, see *Segment memory type*, page 68.

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

Categories of declared data	Segment group	Suffix
Zero-initialized non-located data	`BDATA/DATA/IDATA/XDATA/IXDATA/FAR/HUGE`	`Z`
Non-zero initialized non-located data	`BDATA/DATA/IDATA/XDATA/IXDATA/FAR/HUGE`	`I`
Initializers for the above	`BDATA/DATA/IDATA/XDATA/IXDATA/FAR/HUGE`	`ID`
Initialized located constants	`CODE/XDATA_ROM/FAR_CODE/FAR_ROM/HUGE_CODE/HUGE_ROM`	`AC`
Initialized non-located constants	`CODE/XDATA_ROM/FAR_CODE/FAR_ROM/HUGE_CODE/HUGE_ROM`	`C`
Non-initialized located data	`SFR/CODE/FAR_CODE/HUGE_CODE/BDATA/DATA/IDATA/XDATA/IXDATA/FAR/HUGE`	`AN`
Non-initialized non-located data	`BIT/CODE/FAR_CODE/HUGE_CODE/BDATA/DATA/IDATA/XDATA/IXDATA/FAR/HUGE`	`N`

Table 15: Segment name suffixes

For a summary of all supported segments, see *Summary of segments*, page 295.

Examples

Assume the following examples:

`__xdata int j;`
`__xdata int i = 0;`
	The xdata variables that are to be initialized to zero when the system starts will be placed in the segment `XDATA_Z`.

`__no_init __xdata int j;`	The xdata non-initialized variables will be placed in the segment `XDATA_N`.
`__xdata int j = 4;`	The xdata non-zero initialized variables will be placed in the segment `XDATA_I` in RAM, and initializer data in segment `XDATA_ID` in ROM.
`__xdata_rom const int i = 5;`	The xdata_rom initialized constant will be placed in the segment `XDATA_ROM_C`.

`__code const int i = 6;`	The initialized constant located in code memory space will be placed in the `CODE_C` segment.

Initialized data

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

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

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

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

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

XDATA_I	`0x1000-0x10F2` and `0x1100-0x11E7`
XDATA_ID	`0x4000-0x41DA`

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

XDATA_I	`0x1000-0x10F2` and `0x1100-0x11E7`
XDATA_ID	`0x4000-0x40F2` and `0x4100-0x41E7`

   The `ID` segment can, for all segment groups, be placed anywhere in the code memory space, because it is not accessed using the corresponding access method. It is only used for initializing the corresponding `I` segment. Note that the gap between the ranges will also be copied.
3. Finally, global C++ objects are constructed, if any.

Data segments for static memory in the default linker command file

The `IDATA` segments must be placed in the theoretical memory range `0x1-0xFF`. In the example below, these segments are placed in the available RAM area `0x30`–`0x7F`. The segment `IDATA_ID` can be placed anywhere in ROM.

The segments in the `XDATA` segment type must be placed in the theoretical memory range `0x1-0xFFFFFF`. In the example below, they are placed in the available RAM area, `0x1–0x7FFF`. The segment `XDATA_ID` can be placed anywhere in ROM.

The default linker command file contains the following directives to place the static data segments:

```
//The segments to be placed in ROM are defined:
-Z(CODE)IDATA_ID,XDATA_ID=0-FFFF

//The RAM data segments are placed in memory:
-Z(IDATA)IDATA_I,IDATA_Z,IDATA_N=30-FF
-Z(XDATA)XDATA_I,XDATA_Z,XDATA_N=1-FFFF
```

Note: Segments that contain data that can be pointed to cannot be placed on address 0, because a pointer cannot point at address 0 as that would be a NULL pointer.

THE STACKS

You can configure the compiler to use three different stacks; one of two hardware stacks (supported internally by the processor and used by, for example, the `PUSH`, `POP`, `CALL`, and `RET` instructions) and two emulated stacks.

Only one hardware stack at a time is supported by the microcontroller. For standard MCS-51 devices this is the idata stack, located in idata memory. On extended MCS-51 devices there is an option to instead use an extended hardware stack located in xdata memory. The emulated xdata and pdata stacks have no support in the hardware; they are instead supported by the compiler software.

The stacks are used by functions to store variables and other information that is used locally by functions, as described in the chapter *Data storage*. They are continuous blocks of memory pointed to by a stack pointer.

The data segment used for holding the stack is one of `ISTACK`, `PSTACK`, `XSTACK`, or `EXT_STACK`. The system startup code initializes the stack pointer to the beginning or the end of the stack segment, depending on the stack type.

Allocating a memory area for the stack is done differently when you use the command line interface compared to when you use the IAR Embedded Workbench IDE.

Stack size allocation in the IAR Embedded Workbench IDE

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

Add the required stack size in the stack size text boxes.

Stack size allocation from the command line

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

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

```
-D_STACK_SIZE=size
```

where `STACK` can be one of `IDATA_STACK`, `XDATA_STACK`, `PDATA_STACK`, or `EXTENDED_STACK`

Specify an appropriate size for your application. Note that the size is written hexadecimally.

Placement of stack segment

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

```
-Z(IDATA)ISTACK+_IDATA_STACK_SIZE=start-end
-Z(XDATA)XSTACK+_XDATA_STACK_SIZE=start-end
```

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

Idata stack

The idata stack is pointed to by the hardware register `SP`. The stack grows towards higher addresses and `cstartup` initializes SP to the beginning of the idata stack segment.

Note: The idata stack and the extended stack cannot exist at the same time.

Example

```
-Z(IDATA)ISTACK+_IDATA_STACK_SIZE=start-end
```

Extended stack

On some devices, you can use the option `--extended_stack` to select the extended stack in xdata memory instead of the idata stack. The extended stack is pointed to by the register pair `?ESP:SP`. The `?ESP` register is defined in the linker command file. The stack grows towards higher addresses and `cstartup` initializes the register pair `?ESP:SP` to the beginning of the extended stack segment.

Note: The extended stack cannot exist at the same time as an idata stack or xdata stack. It is possible, however, to use both an extended stack and a pdata stack.

Example

```
-Z(XDATA)EXT_STACK+_EXTENDED_STACK_SIZE=start-end
```

Pdata stack

The pdata stack is pointed to by an 8-bit emulated stack pointer, `PSP`, and the stack grows towards lower addresses. The pdata stack must be located in the pdata range of xdata memory. The `cstartup` module initializes `PSP` to the end of the stack segment.

Note: The pdata stack can exist in parallel with all other types of stacks.

Example

```
-Z(XDATA)PSTACK+_PDATA_STACK_SIZE=start-end
```

The pdata stack pointer is a segment in itself and must be located in data memory.

Example

```
-Z(DATA)PSP=08-7F
```

Xdata stack

The xdata stack is pointed to by a 16-bit emulated stack pointer, `XSP`, and the stack grows towards lower addresses. The xdata stack must be located in xdata memory. The `cstartup` module initializes `XSP` to the end of the stack segment.

Note: The xdata stack and the extended stack cannot both exist at the same time.

Example

```
-Z(XDATA)XSTACK+_XDATA_STACK_SIZE=start-end
```

The xdata stack pointer is a segment in itself and must be located in data memory.

Example

```
-Z(DATA)XSP=08-7F
```

Summary

This table summarizes the different stacks:

Stack	Maximum size	Calling convention	Description
Idata	256 bytes	Idata reentrant	Hardware stack. Grows towards higher memory.
Pdata	256 bytes	Pdata reentrant	Emulated stack. Grows towards lower memory.
Xdata	64 Kbytes	Xdata reentrant	Emulated stack. Grows towards lower memory.

Table 16: Summary of stacks

Stack	Maximum size	Calling convention	Description
Extended	64 Kbytes	Extended stack reentrant	Hardware stack. Grows towards higher memory. Only for devices that have a stack in the external data memory space.

Table 16: Summary of stacks (Continued)

The stacks are used in different ways. The idata stack is always used for register spill (or the extended stack if you are using the `--extended_stack` option). It is also used for parameters, local variables, and the return address when you use the idata or extended stack reentrant calling convention. Devices that support xdata memory can store function parameters, local variables and the return address on the pdata or xdata stack by using the pdata reentrant or xdata reentrant calling convention.

Stack size considerations

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

THE HEAP

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

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

- Linker segments used for the heap
- Allocating the heap size, which differs depending on which build interface you are using
- Placing the heap segments in memory.

Note: Only MCS-51 devices with external data memory can have a heap.

Heap segments in DLIB

To access a heap in a specific memory, use the appropriate memory attribute as a prefix to the standard functions `malloc`, `free`, `calloc`, and `realloc`, for example:

```
__xdata_malloc
```

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

Each heap will reside in a segment with the name `_HEAP` prefixed by a memory attribute.

For information about available heaps, see *Heaps*, page 108.

Heap segments in the CLIB runtime environment

The memory allocated to the heap is placed in the segment `HEAP`, which is only included in the application if dynamic memory allocation is actually used.

Heap size allocation in the IAR Embedded Workbench IDE

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

Add the required heap size in the **Heap size** text box.

Heap size allocation from the command line

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

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

```
-D_XDATA_HEAP_SIZE=size
-D_FAR_HEAP_SIZE=size
-D_HUGE_HEAP_SIZE=size
```

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

Specify the appropriate size for your application.

Placement of heap segment

The actual heap segment is allocated in the memory area available for the heap:

```
-Z(DATA)HEAP+_XDATA_HEAP_SIZE=start-end
```

Note: This range does not specify the size of the heap; it specifies the range of the available memory.

Heap size and standard I/O

If you have excluded `FILE` descriptors from the DLIB runtime environment, as in the Normal configuration, there are no input and output buffers at all. Otherwise, as in the Full configuration, be aware that the size of the input and output buffers is set to 512 bytes in the `stdio` library header file. If the heap is too small, I/O will not be buffered, which is considerably slower than when I/O is buffered. If you execute the application using the simulator driver of the IAR C-SPY® Debugger, you are not likely to notice the speed penalty, but it is quite noticeable when the application runs on an MCS-51 microcontroller. If you use the standard I/O library, you should set the heap size to a value which accommodates the needs of the standard I/O buffer.

LOCATED DATA

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

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

Code segments

This section contains descriptions of the segments used for storing code, and the interrupt vector table. For a complete list of all segments, see *Summary of segments*, page 295.

Note: The symbols used in the following examples for describing memory locations are defined in the `lnk_base.xcl` linker command file. Device-specific values for these symbols are defined in the linker command file that you are using.

STARTUP CODE

The segment `CSTART` contains code used during system setup (`cstartup`), runtime initialization (`cmain`), and system termination (`cexit`). The system setup code should be placed at the location where the chip starts executing code after a reset. For the MCS-51 microcontroller, this is at the address `0x0`, the start of the interrupt vector. Located at address zero there is a jump instruction that jumps to the initializing code in `CSTART`. In addition, the segments must be placed into one continuous memory space, which means the `-P` segment directive cannot be used.

In the default linker command file, the following line will place the `CSTART` segment in code memory and the `INTVEC` segment at the address `0x0`. The `INTVEC` segment starts with a jump instruction to the `CSTART` segment:

```
-Z(CODE)INTVEC=0
-Z(CODE)CSTART=_CODE_START-_CODE_END
```

NORMAL CODE

Functions declared without a memory attribute are placed in different segments, depending on which code model you are using. The different segments are: NEAR_CODE, BANKED_CODE, and FAR_CODE.

Near code

Near code—that is, all user-written code when you use the near code model, or functions explicitly typed with the memory attribute __near_func—is placed in the NEAR_CODE segment.

In the linker command file it can look like this:

```
-Z(CODE)NEAR_CODE=_CODE_START-_CODE_END
```

Banked code

When you use the banked code model, all user-written code is located in the BANKED_CODE segment. Here, the -P linker directive is used for allowing XLINK to split up the segments and pack their contents more efficiently. This is useful here, because the memory range is non-consecutive.

In the linker command file it can look like this:

```
-P(CODE)BANKED_CODE=[_CODEBANK_START-_CODEBANK_END]*4+10000
```

Here four code banks are declared. If for example _CODEBANK_START is 4000 and _CODEBANK_END is 7FFF, the following banks are created: 0x4000-0x7FFF, 0x14000-0x17FFF, 0x24000-0x27FFF, 0x34000-0x37FFF.

Far code

Far code—that is, all user-written code when you use the far code model, or functions explicitly typed with the memory attribute __far_func—is placed in the FAR_CODE segment.

In the linker command file it can look like this:

```
-Z(CODE)FAR_CODE=_CODE_START-_CODE_END
-P(CODE)FAR_CODE=[_FAR_CODE_START-_FAR_CODE_END]/10000
```

INTERRUPT VECTORS

The interrupt vectors are typically placed in the segment INTVEC. The location of this segment is chip-dependent. For the MCS-51 microcontrollers, it must be placed at address 0x0:

```
-Z(CODE)INTVEC=0
```

C++ dynamic initialization

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

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

For example:

```
-Z(CONST)DIFUNCT=_CODE_START-_CODE_END
```

For additional information, see *DIFUNCT*, page 306.

Verifying the linked result of code and data placement

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

SEGMENT TOO LONG ERRORS AND RANGE ERRORS

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

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

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

LINKER MAP FILE

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

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

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

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

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

MANAGING MULTIPLE ADDRESS SPACES

Output formats that do not support more than one memory space—like Motorola S-records and Intel hex—may require up to one output file per memory space. This causes no problems if you are only producing output to one memory space (flash memory), but if you also are placing objects in EEPROM or an external ROM in DATA space, the output format cannot represent this, and the linker issues the following error message:

```
Error[e133]: The output format Format cannot handle multiple
address spaces. Use format variants (-y -O) to specify which
address space is wanted.
```

To limit the output to flash, make a copy of the linker command file for the device you are using, and put it in the project directory. Add the following line at the end of the file:

```
-y(CODE)
```

To produce output for the other memory space(s), you must generate one output file per memory space (because the output format you have chosen does not support more than one memory space). Use the XLINK option `-O` for this purpose.

For each additional output file, you have to specify format, XLINK segment type, and file name. For example:

```
-Omotorola,(DATA)=external_rom.a51
-Omotorola,(XDATA)=eeprom.a51
```

Note: As a general rule, an output file is only necessary if you use non-volatile memory. In other words, output from the data space is only necessary if the data space contains external ROM.

The DLIB runtime environment

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

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

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

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

Introduction to the runtime environment

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

This section gives an overview of:

- The runtime environment and its components
- Library selection.

RUNTIME ENVIRONMENT FUNCTIONALITY

The *runtime environment* supports ISO/ANSI C and C++ including the standard template library. The runtime environment consists of the *runtime library*, which contains the functions defined by these standards, and include files that define the library interface.

The runtime library is delivered both as prebuilt libraries and as source files, and you can find them in the product subdirectories `8051\lib` and `8051\src\lib`, respectively.

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

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

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

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

LIBRARY SELECTION

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

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

- Setting library options, for example, for choosing `scanf` input and `printf` output formatters, and for specifying the size of the stack and the heap
- Overriding certain library functions, for example `cstartup.s51`, with your own customized versions
- Choosing the level of support for certain standard library functionality, for example, locale, file descriptors, and multibyte characters, by choosing a *library configuration*: normal or full.

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

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

SITUATIONS THAT REQUIRE LIBRARY BUILDING

Building a customized library is complex. You should therefore carefully consider whether it is really necessary.

You must build your own library when:

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

For information about how to build a customized library, see *Building and using a customized library*, page 95.

LIBRARY CONFIGURATIONS

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

The following DLIB library configurations are available:

Library configuration	Description
Normal DLIB	No locale interface, C locale, no file descriptor support, no multibyte characters in `printf` and `scanf`, and no hex floats in `strtod`.
Customized DLIB	User-defined library settings

Table 17: Library configurations

In addition to these configurations, you can define your own configurations, which means that you must modify the configuration file. Note that the library configuration file describes how a library was built and thus cannot be changed unless you rebuild the library. For further information, see *Building and using a customized library*, page 95.

The prebuilt libraries are based on the default configurations, see *Using a prebuilt library*, page 88. There is also a ready-made library project template that you can use if you want to rebuild the runtime library.

DEBUG SUPPORT IN THE RUNTIME LIBRARY

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

The following table describes the different levels of debugging support:

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

Table 18: Levels of debugging support in runtime libraries

If you build your application project with the XLINK options **With runtime control modules** or **With I/O emulation modules**, certain functions in the library will be replaced by functions that communicate with the IAR C-SPY Debugger. For further information, see *C-SPY Debugger runtime interface*, page 109.

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

Using a prebuilt library

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

- DLIB runtime environment
- Core variant
- Stack location
- Code model

- Data model
- Calling convention
- Constant location
- Number of data pointers
- Data pointer visibility
- Data pointer size
- Data pointer selection method
- Library configuration: Normal.
- Library configuration—Normal and Full.

Note: A DLIB library cannot be built for the Tiny and Small data models.

For the 8051 IAR C/C++ Compiler some of these combinations are delivered as prebuilt runtime libraries. When you need a compiler configuration for which there is no prebuilt library, you can build your own library, see *Building and using a customized library*, page 95.

The names of the libraries are constructed in the following way:

```
<type>-<core><stack>-<code_model><data_model><call_conv><const_lo
c>-<#_of_dptrs><dptr_vis><dptr_size><dptr_select>.r51
```

where

- `<type>` is `dl` for the IAR DLIB library. See the chapter *Library functions*.
- `<core>` is `pl` for the classic 8051 devices, `e1` for the extended1 devices, or `e2` for the extended2 devices. See *Basic project settings for hardware memory configuration*, page 15.
- `<stack>` specifies where the machine stack is located; `i` for an idata stack and `e` for an extended stack.
- `<code_model>` is `n`, `b`, `2`, or `f` for the near, banked, banked_extended2, or far code model, respectively. See *Code models for function storage*, page 47.
- `<data_model>` is `s`, `l`, `g`, or `f` for the small, large, generic, or far data model, respectively. See *Data models*, page 22.
- `<call_conv>` is `d`, `o`, `i`, `p`, `x`, or `e`, representing one of the available calling conventions: data overlay (`d`), idata overlay (`o`), idata reentrant (`i`), pdata reentrant (`p`), xdata reentrant (`x`), or extended stack reentrant (`e`). See *Choosing a calling convention*, page 35.
- `<const_loc>` is the location for constants and strings: `d` or `c` for data or code, respectively. See *Constants and strings*, page 34.
- `<#_of_dptrs>` is a number from 1 to 8 that represents the number of data pointers used. See *Using the DPTR register*, page 17.

- *`<dptr_vis>`*: stands for `DPTR` visibility: `h` (shadowed) or `e` (separate)
- *`<dptr_size>`* is the size of of the used data pointer; either 16 or 24.
- *`<dptr_select>`* shows the DPTR selection method and the selection mask if the XOR selection method is used. In that case the value is `x` followed by the mask in hexadecimal representation, for example `01` for `0x01`, resulting in the selection field `x01`. If the INC selection method is used, the value of the field is `inc`. See *Using the DPTR register*, page 17.

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

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

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

  ```
  dl-pli-nlxd-1e16x01.r51
  ```
- Specify the include paths for the compiler and assembler:

  ```
  -I 8051\inc\dlib
  ```
- Specify the library configuration file for the compiler:

  ```
  --dlib_config C:\...\dl8051Normal.h
  ```

Note: All modules in the library have a name that starts with the character `?` (question mark).

You can find the library object files and the library configuration files in the subdirectory `8051\lib\dlib`.

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

The prebuilt libraries delivered with the 8051 IAR C/C++ Compiler can be used as is. However, it is possible to customize parts of a library without rebuilding it. There are two different methods:

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

The following items can be customized:

Items that can be customized	Described in
Formatters for printf and scanf	*Choosing formatters for printf and scanf*, page 91

Table 19: Customizable items

Items that can be customized	Described in
Startup and termination code	*System startup and termination*, page 96
Low-level input and output	*Standard streams for input and output*, page 100
File input and output	*File input and output*, page 103
Low-level environment functions	*Environment interaction*, page 106
Low-level signal functions	*Signal and raise*, page 107
Low-level time functions	*Time*, page 107
Size of heaps, stacks, and segments	*Placing code and data*, page 67

Table 19: Customizable items (Continued)

For a description about how to override library modules, see *Overriding library modules*, page 93.

Choosing formatters for printf and scanf

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

Note: If you rebuild the library, it is possible to optimize these functions even further, see *Configuration symbols for printf and scanf*, page 102.

CHOOSING PRINTF FORMATTER

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

The following table summarizes the capabilities of the different formatters:

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

Table 20: Formatters for printf

† **Depends on the library configuration that is used.**

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 102.

Specifying the print formatter in the IAR Embedded Workbench IDE

To specify the `printf` formatter in the IAR Embedded Workbench IDE, choose **Project>Options** and select the **General Options** category. Select the appropriate option on the **Library options** page.

Specifying printf formatter from the command line

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

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

CHOOSING SCANF FORMATTER

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

The following table summarizes the capabilities of the different formatters:

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

Table 21: Formatters for scanf

† **Depends on the library configuration that is used.**

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 102.

Specifying scanf formatter in the IAR Embedded Workbench IDE

To specify the printf formatter in the IAR Embedded Workbench IDE, choose **Project>Options** and select the **General Options** category. Select the appropriate option on the **Library options** page.

Specifying scanf formatter from the command line

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

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

Overriding library modules

The library contains modules which you probably need to override with your own customized modules, for example functions for character-based I/O and cstartup. This can be done without rebuilding the entire library. This section describes the procedure for including your version of the module in the application project build process. The library files that you can override with your own versions are located in the 8051\src\lib directory.

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

Overriding library modules using the IAR Embedded Workbench IDE

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

1 Copy the appropriate *`library_module`*`.c` file to your project directory.

2 Make the required additions to the file (or create your own routine, using the default file as a model), and make sure that it has the same *module name* as the original module. The easiest way to achieve this is to save the new file under the same name as the original file.

3 Add the customized file to your project.

4 Rebuild your project.

Overriding library modules from the command line

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

1 Copy the appropriate *`library_module`*`.c` to your project directory.

2 Make the required additions to the file (or create your own routine, using the default file as a model), and make sure that it has the same *module name* as the original module. The easiest way to achieve this is to save the new file under the same name as the original file.

3 Compile the modified file using the same options as for the rest of the project:

```
icc8051 library_module --core=plain --data_model=large
--calling_convention=xr --dptr=2,16,shadowed,xor(0x1)
```

This creates a replacement object module file named *`library_module`*`.r51`.

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

4 Add *`library_module`*`.r51` to the XLINK command line, either directly or by using an extended linker command file, for example:

```
xlink mycode.r51 library_module dl-pli-nlxd-2h16x01.r51
```

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

Run XLINK to rebuild your application.

This will use your version of `library_module.r51`, instead of the one in the library. For information about the XLINK options, see the *IAR Linker and Library Tools Reference Guide*.

Building and using a customized library

In some situations, see *Situations that require library building*, page 87, it is necessary to rebuild the library. In those cases you need to:

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

Information about the build process is described in the *IAR Embedded Workbench® IDE User Guide*.

Note: It is possible to build IAR Embedded Workbench projects from the command line by using the IAR Command Line Build Utility (`iarbuild.exe`). However, no make or batch files for building the library from the command line are provided.

SETTING UP A LIBRARY PROJECT

The IAR Embedded Workbench IDE provides a library project template which can be used for customizing the runtime environment configuration. This library template has Normal library configuration, see Table 17, *Library configurations*, page 87.

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

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

It is easiest to build customized runtime libraries in the 8051 IAR Embedded Workbench. It is however, also possible to build them using the 8051 IAR C/C++ Compiler and IAR XLINK Linker from the command line.

MODIFYING THE LIBRARY FUNCTIONALITY

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

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `Dlib_defaults.h`. This read-only file describes the configuration possibilities. In addition, your library has its own library configuration file `dl8051`*`libraryname`*`.h` (where *`libraryname`* is either `Normal` or `Full`), which sets up that specific library with full library configuration. For more information, see Table 19, *Customizable items*, page 90.

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

Modifying the library configuration file

In your library project, open the file `dl8051`*`libraryname`*`.h` and customize it by setting the values of the configuration symbols according to the application requirements.

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

USING A CUSTOMIZED LIBRARY

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

In the IAR Embedded Workbench IDE you must perform the following steps:

1. Choose **Project>Options** and click the **Library Configuration** tab in the **General Options** category.
2. Choose **Custom DLIB** from the **Library** drop-down menu.
3. In the **Library file** text box, locate your library file.
4. In the **Configuration file** text box, locate your library configuration file.

System startup and termination

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

The code for handling startup and termination is located in the source files `cstartup.s51`, `cmain.s51`, `cexit.s51`, and `low_level_init.c` located in the `8051\src\lib` directory.

SYSTEM STARTUP

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

For the hardware intialization, it looks like this:

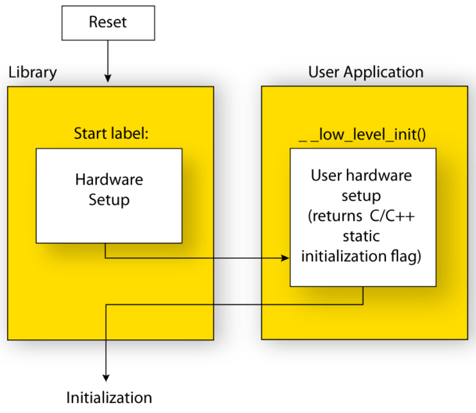


Figure 9: Target hardware initialization phase

- When the CPU is reset it will jump to the program entry label `__program_start` in the system startup code
- The register bank switch register is initialized to the number specified by the symbol `?REGISTER_BANK` in the linker command file
- If the idata stack is used, the stack pointer, `SP`, is initialized to the beginning of the `ISTACK` segment. If the extended stack is used, the extended stack pointer `?ESP:SP` is initialized to the beginning of the `EXT_STACK` segment
- If the xdata reentrant calling convention is available, the xdata stack pointer, `XSP`, is initialized to the end of the `XSTACK` segment
- If the pdata reentrant calling convention is available, the pdata stack pointer, `PSP`, is initialized to the end of the `PSTACK` segment
- If code banking is used, the bank register is initialized to zero
- The `PDATA` page is initialized
- If multiple data pointers are available, the `DPTR` selector register is initialized and the first data pointer (`dptr0`) is set to be the active data pointer
- The function `__low_level_init` is called if you have defined it, giving the application a chance to perform early initializations.

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

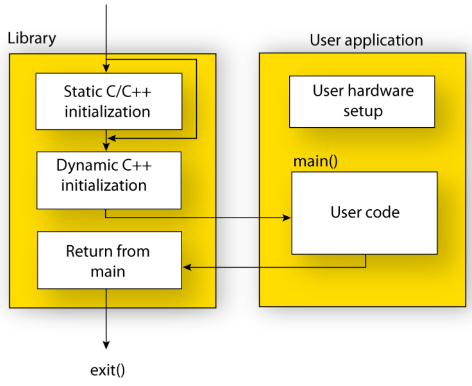


Figure 10: C/C++ initialization phase

- Static variables are initialized; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the rest of the initialized variables depending on the return value of `__low_level_init`. For more details, see *Initialized data*, page 75
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

SYSTEM TERMINATION

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

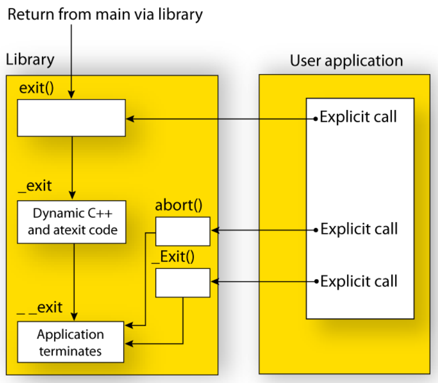


Figure 11: System termination phase

An application can terminate normally in two different ways:

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

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

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

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

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

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

C-SPY interface to system termination

If your project is linked with the XLINK options **With runtime control modules** or **With I/O emulation modules**, the normal `__exit` and `abort` functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see *C-SPY Debugger runtime interface*, page 109.

Customizing system initialization

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

You can do this by providing a customized version of the routine `__low_level_init`, which is called from `cmain.s51` before the data segments are initialized. Modifying the file `cstartup` directly should be avoided.

The code for handling system startup is located in the source files `cstartup.s51` and `low_level_init.c`, located in the `8051\src\lib` directory.

Note: Normally, there is no need for customizing either of the files `cmain.s51` or `cexit.s51`.

If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 95.

Note: Regardless of whether you modify the routine `__low_level_init` or the file `cstartup.s51`, you do not have to rebuild the library.

__LOW_LEVEL_INIT

One skeleton low-level initialization file is supplied with the product—the C source file, `low_level_init.c`. The only limitation using this C source version is that static initialized variables cannot be used within the file, as variable initialization has not been performed at this point.

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

MODIFYING THE FILE CSTARTUP.S51

As noted earlier, you should not modify the file `cstartup.s51` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s51`, we recommend that you follow the general procedure for creating a modified copy of the file and adding it to your project, see *Overriding library modules*, page 93.

Standard streams for input and output

There are three standard communication channels (streams)—`stdin`, `stdout`, and `stderr`—which are defined in `stdio.h`. If any of these streams are used by your application, for example by the functions `printf` and `scanf`, you need to customize the low-level functionality to suit your hardware.

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

IMPLEMENTING LOW-LEVEL CHARACTER INPUT AND OUTPUT

To implement low-level functionality of the `stdin` and `stdout` streams, you must write the functions `__read` and `__write`, respectively. You can find template source code for these functions in the `8051\src\lib` directory.

If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 95. Note that customizing the low-level routines for input and output does not require you to rebuild the library.

Note: If you write your own variants of `__read` or `__write`, special considerations for the C-SPY runtime interface are needed, see *C-SPY Debugger runtime interface*, page 109.

Example of using __write and __read

The code in the following examples use memory-mapped I/O to write to an LCD display:

```
__no_init volatile unsigned char LCD_IO @ address;

size_t __write(int Handle, const unsigned char * Buf,
               size_t Bufsize)
{
  size_t nChars = 0;
  /* Check for stdout and stderr
     (only necessary if FILE descriptors are enabled.) */
  if (Handle != 1 && Handle != 2)
  {
    return -1;
  }
  for (/*Empty */; Bufsize > 0; --Bufsize)
  {
    LCD_IO = * Buf++;
    ++nChars;
  }
  return nChars;
}
```

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

```
__no_init volatile unsigned char KB_IO @ address;

size_t __read(int Handle, unsigned char *Buf, size_t BufSize)
{
  size_t nChars = 0;
  /* Check for stdin
     (only necessary if FILE descriptors are enabled) */
  if (Handle != 0)
  {
    return -1;
  }
  for (/*Empty*/; BufSize > 0; --BufSize)
  {
    unsigned char c = KB_IO;
    if (c == 0)
      break;
    *Buf++ = c;
    ++nChars;
  }
  return nChars;
}
```

For information about the `@` operator, see *Controlling data and function placement in memory*, page 165.

Configuration symbols for printf and scanf

When you set up your application project, you typically need to consider what `printf` and `scanf` formatting capabilities your application requires, see *Choosing formatters for printf and scanf*, page 91.

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

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

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

Printf configuration symbols	Includes support for
`_DLIB_PRINTF_MULTIBYTE`	Multibyte characters
`_DLIB_PRINTF_LONG_LONG`	Long long (`ll` qualifier)
`_DLIB_PRINTF_SPECIFIER_FLOAT`	Floating-point numbers
`_DLIB_PRINTF_SPECIFIER_A`	Hexadecimal floats
`_DLIB_PRINTF_SPECIFIER_N`	Output count (`%n`)
`_DLIB_PRINTF_QUALIFIERS`	Qualifiers `h`, `l`, `L`, `v`, `t`, and `z`
`_DLIB_PRINTF_FLAGS`	Flags `-`, `+`, `#`, and `0`
`_DLIB_PRINTF_WIDTH_AND_PRECISION`	Width and precision

Table 22: Descriptions of printf configuration symbols

Printf configuration symbols	Includes support for
`_DLIB_PRINTF_CHAR_BY_CHAR`	Output char by char or buffered

Table 22: Descriptions of printf configuration symbols (Continued)

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

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

Table 23: Descriptions of scanf configuration symbols

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you need to set up a library project, see *Building and using a customized library*, page 95. Define the configuration symbols according to your application requirements.

File input and output

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

Note that file I/O capability in the library is only supported by libraries with full library configuration, see *Library configurations*, page 87. In other words, file I/O is supported when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is enabled. If not enabled, functions taking a *`FILE *`* argument cannot be used.

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

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

Table 24: Low-level I/O files

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

Note: If you link your library with I/O debugging support, C-SPY variants of the low-level I/O functions will be linked for interaction with C-SPY. For more information, see *Debug support in the runtime library*, page 87.

Locale

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

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

The DLIB library can be used in two main modes:

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

LOCALE SUPPORT IN PREBUILT LIBRARIES

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

- All prebuilt libraries support the C locale only

- Libraries with *normal library configuration* do not have support for the locale interface.

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

CUSTOMIZING THE LOCALE SUPPORT

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

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

Locale configuration symbols

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

```
#define _DLIB_FULL_LOCALE_SUPPORT 1
#define _LOCALE_USE_C        /* C locale */
#define _LOCALE_USE_EN_US    /* US english */
#define _LOCALE_USE_EN_GB    /* UK english */
#define _LOCALE_USE_SV_SE    /* Swedish in Sweden */
```

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

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Building and using a customized library*, page 95.

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

Building a library without support for locale interface

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

Building a library with support for locale interface

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

CHANGING LOCALES AT RUNTIME

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

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

`lang_REGION`

or

`lang_REGION.encoding`

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

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

Example

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

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

Environment interaction

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

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

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

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

```
key=value\0
```

The last string must be empty. Assign the created sequence of strings to the `__environ` variable.

For example:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `8051\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 93.

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

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 95.

Note: If you link your application with support for I/O debugging, the functions `getenv` and `system` will be replaced by C-SPY variants. For further information, see *Debug support in the runtime library*, page 87.

Signal and raise

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

This does not require that you rebuild the library. You can find source templates in the files `signal.c` and `raise.c` in the `8051\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 93.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 95.

Time

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

This does not require that you rebuild the library. You can find source templates in the files `clock.c` and `time.c`, and `getzone.c` in the `8051\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 93.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 95.

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

Note: If you link your application with support for I/O debugging, the functions `clock` and `time` will be replaced by C-SPY variants that return the host clock and time respectively. For further information, see *C-SPY Debugger runtime interface*, page 109.

Strtod

The function `strtod` does not accept hexadecimal floating-point strings in libraries with the normal library configuration. To make a library do so, you need to rebuild the library, see *Building and using a customized library*, page 95. Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.

Assert

If you have linked your application with support for runtime debugging, C-SPY will be notified about failed asserts. If this is not the behavior you require, you must add the source file `xreportassert.c` to your application project. Alternatively, you can rebuild the library. The `__ReportAssert` function generates the assert notification. You can find template code in the `8051\src\lib` directory. For further information, see *Building and using a customized library*, page 95. To turn off assertions, you must define the symbol `NDEBUG`.

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

Heaps

The runtime environment supports heaps in the following memory types:

Memory type	Segment name	Extended keyword	Used by default in data model
Xdata	XDATA_HEAP	__xdata	Large
Far	FAR_HEAP	__far	Far
Huge	HUGE_HEAP	__huge	

Table 25: Heaps and memory types

See *The heap*, page 79 for information about how to set the size for each heap. To use a specific heap, the prefix in the table is the extended keyword to use in front of `malloc`, `free`, `calloc`, and `realloc`, for instance `__far_malloc`. The default functions will use one of the specific heap variants, depending on project settings such as data model. For information about how to use a specific heap in C++, see *New and Delete operators*, page 154.

C-SPY Debugger runtime interface

To include support for runtime and I/O debugging, you must link your application with the XLINK options **With runtime control modules** or **With I/O emulation modules**, see *Debug support in the runtime library*, page 87.

In this case, C-SPY variants of the following library functions will be linked to the application:

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

Table 26: Functions with special meanings when linked with debug info

Function	Description
`system`	Writes a message to the Debug Log window and returns -1
`time`	Returns the time on the host computer
`__write`	`stdin`, `stdout`, and `stderr` will be directed to the Terminal I/O window, all other files will write to the associated host file

Table 26: Functions with special meanings when linked with debug info (Continued)

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

LOW-LEVEL DEBUGGER RUNTIME INTERFACE

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

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

The mechanism used for implementing this feature works as follows:

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

THE DEBUGGER TERMINAL I/O WINDOW

To make the Terminal I/O window available, the application must be linked with support for I/O debugging, see *Debug support in the runtime library*, page 87. This means that when the functions `__read` or `__write` are called to perform I/O operations on the streams `stdin`, `stdout`, or `stderr`, data will be sent to or read from the C-SPY Terminal I/O window.

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

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

Speeding up terminal output

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

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

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

```
-e__write_buffered=__write
```

Checking module consistency

This section introduces the concept of runtime model attributes, a mechanism used by the IAR compiler, assembler, and linker to ensure module consistency.

When developing an application, it is important to ensure that incompatible modules are not used together. For example, in the 8051 IAR C/C++ Compiler, it is possible to write a module that only supports separate registers. If you write a routine that support separate `DPTR` registers, it is possible to check that the routine is not used in an application built for shadowed `DPTR`.

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

RUNTIME MODEL ATTRIBUTES

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

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

Example

In the following table, the object files could (but do not have to) define the two runtime attributes `color` and `taste`. In this case, `file1` cannot be linked with any of the other files, since the runtime attribute `color` does not match. Also, `file4` and `file5` cannot be linked together, because the `taste` runtime attribute does not match.

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

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

Table 27: Example of runtime model attributes

USING RUNTIME MODEL ATTRIBUTES

Runtime model attributes can be specified in your C/C++ source code to ensure module consistency with other object files by using the `#pragma rtmodel` directive. For example:

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

For detailed syntax information, see *rtmodel*, page 270.

Runtime model attributes can also be specified in your assembler source code by using the `RTMODEL` assembler directive. For example:

```
  RTMODEL "color", "red"
```

For detailed syntax information, see the *8051 IAR Assembler Reference Guide*.

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

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

PREDEFINED RUNTIME ATTRIBUTES

The table below shows the predefined runtime model attributes that are available for the 8051 IAR C/C++ Compiler. These can be included in assembler code or in mixed C/C++ and assembler code.

Runtime model attribute	Value	Description
`__calling_convention`	`data_overlay`, `idata_overlay`, `idata_reentrant`, `pdata_reentrant`, `xdata_reentrant` or `ext_stack_reentrant`	Corresponds to the calling convention used in the project.
`__code_model`	`near`, `banked`, `banked_ext2`, or `far`	Corresponds to the code model used in the project.
`__core`	`plain`, `extended1`, or `extended2`	Corresponds to the core variant option used in the project.
`__data_model`	`tiny`, `small`, `large`, `generic`, or `far`	Corresponds to the data model used in the project.
`__dptr_size`	`16` or `24`	Corresponds to the size of the data pointers used in your application.
`__dptr_visibility`	`separate` or `shadowed`	Corresponds to the data pointer visibility.
`__extended_stack`	`enabled` or `disabled`	Corresponds to the extended stack option.
`__location_for_constants`	`code` or `data`	Corresponds to the option for specifying the default location for constants.
`__number_of_dptrs`	a number from `1–8`	Corresponds to the number of data pointers available in your application.

Table 28: Runtime model attributes

Runtime model attribute	Value	Description
`__rt_version`	*n*	This runtime key is always present in all modules generated by the 8051 IAR C/C++ Compiler. If a major change in the runtime characteristics occurs, the value of this key changes.

Table 28: Runtime model attributes (Continued)

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

If you are using assembler routines in the C or C++ code, refer to the chapter *Assembler directives* in the *8051 IAR Assembler Reference Guide*.

Examples

For an example of using the runtime model attribute `__rt_version` for checking module consistency on used calling convention, see *Hints for using a calling convention*, page 132.

The following assembler source code provides a function that counts the number of times it has been called by increasing the register `R4`. The routine assumes that the application does not use `R4` for anything else, that is, the register has been locked for usage. To ensure this, a runtime module attribute, `__reg_r4`, has been defined with a value `counter`. This definition will ensure that this specific module can only be linked with either other modules containing the same definition, or with modules that do not set this attribute. Note that the compiler sets this attribute to `free`, unless the register is locked.

```
            RTMODEL     "__reg_r4", "counter"
            MODULE      myCounter
            PUBLIC      myCounter
            RSEG        CODE:CODE:NOROOT(1)
myCounter:  INC         R4
            RET
            ENDMOD
            END
```

If this module is used in an application that contains modules where the register `R4` has not been locked, an error is issued by the linker:

```
Error[e117]: Incompatible runtime models. Module myCounter
specifies that '__reg_r4' must be 'counter', but module part1
has the value 'free'
```

USER-DEFINED RUNTIME MODEL ATTRIBUTES

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

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

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

The CLIB runtime environment

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

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

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

Note that the CLIB runtime environment can only be used with the C language. DLIB can be used with both the C and the C++ languages. see the chapter *The DLIB runtime environment*, page 85.

For information about migrating from CLIB to DLIB, see the *8051 IAR Embedded Workbench® Migration Guide*.

Runtime environment

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

The 8051 IAR Embedded Workbench comes with a set of prebuilt runtime libraries, which are configured for different combinations of the following features:

- CLIB runtime environment
- Core variant

- Stack location
- Code model
- Data model
- Calling convention
- Constant location
- Number of data pointers
- Data pointer visibility
- Data pointer size
- Data pointer selection method.

The number of possible combinations is high, but not all combinations are equally likely to be useful. For this reason, only a subset of all possible runtime libraries is delivered prebuilt with the product. The larger variants of the prebuilt libraries are also provided for the DLIB library type. These are the libraries using the data models large or far. If you need a library which is not delivered prebuilt, you must build it yourself, see *Building and using a customized library*, page 95.

COMBINATIONS AND DEPENDENCIES

Not all combinations of core variant, code model, data model, and calling convention are possible. The CLIB library type is default, unless you run the compiler in Embedded C++ mode.

The runtime library names are constructed in the following way:

```
<type>-<core><stack>-<code_model><data_model><calling_convention>
<constants_location>-<#_of_dptrs><dptr_visibility><dptr_size>
<dptr_select>.r51
```

where

- `<type>` is `cl` for the IAR CLIB library, or `dl` for the IAR DLIB library, respectively. See the chapter *Library functions*.
- `<core>` is `pl` for the plain classic devices, `e1` for the extended1 devices, or `e2` for the extended2 devices. See *Basic project settings for hardware memory configuration*, page 15.
- `<stack>` specifies where the machine stack is located; `i` for an idata stack and `e` for an extended stack. All non-Maxim (Dallas Semiconductor) 390/400 devices should use `i`, Maxim 390/400 devices can also use `e`.
- `<code_model>` is `n`, `b`, `2`, or `f` for the near, banked, banked_extended2, or far code model, respectively. See *Code models for function storage*, page 47.
- `<data_model>` is `s`, `l`, `g`, or `f` for the small, large, generic, or far data model, respectively. See *Data models*, page 22.

- *`<calling_convention>`* is `d`, `o`, `i`, `p`, `x`, or `e`, representing one of the available calling conventions: data overlay (`d`), idata overlay (`o`), idata reentrant (`i`), pdata reentrant (`p`), xdata reentrant (`x`), or extended stack reentrant (`e`). See *Calling convention*, page 131.
- *`<constants_location>`* is the location for constants and strings: `d` or `c` for data or code, respectively.
- *`<#_of_dptrs>`* is a number from 1 to 8 that represents the number of data pointers used. See *Using the DPTR register*, page 17.
- *`<dptr_visibility>`*:how the DPTR is located in memory; at different locations: `e` (separate) or at the same memory location: `h` (shadowed)
- *`<dptr_size>`* is the size of of the used data pointer; either 16 or 24.
- *`<dptr_select>`* shows the DPTR selection method and the selection mask if the XOR selection method is used. In that case the value is `x` followed by the mask in hexadecimal representation, for example `01` for `0x01`, resulting in the selection field `x01`. If the INC selection method is used, the value of the field is `inc`. See *Using the DPTR register*, page 17.

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

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

```
cl-pli-nsid-1e16x01.r51
```

Building a runtime library

It is easiest to build customized runtime libraries in the 8051 IAR Embedded Workbench IDE. It is however, also possible to build them using the 8051 IAR C/C++ Compiler and IAR XLINK Linker from the command line.

The standard runtime library can be customized using a set of definitions, see the source code for details. For example, the following defines can be used to customize the IAR CLIB C runtime library:

Symbol	Description
`MALLOC_BUFSIZE`	Specifies the size of the heap
`QSORT_MAXSIZE`	Specifies the size of the quick sort buffer
`FRMWRI_BUFSIZE`	Specifies the size of the formatted write buffer

Table 29: Defines used for customizing the runtime library

Input and output

You can customize:

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

CHARACTER-BASED I/O

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

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

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

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

```
__no_init volatile unsigned char DEV_IO @ address;

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

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

For information about how to include your own modified version of putchar and getchar in your project build process, see *Overriding library modules*, page 93.

FORMATTERS USED BY PRINTF AND SPRINTF

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

_medium_write

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

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

`_medium_write` is considerably smaller than `_formatted_write`.

_small_write

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

Specifying the printf formatter in the IAR Embedded Workbench IDE

1 Choose **Project>Options** and select the **General Options** category. Click the **Library options** tab.

2 Select the appropriate **Printf formatter** option, which can be either **Small**, **Medium**, or **Large**.

Specifying the printf formatter from the command line

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

```
-e_small_write=_formatted_write
```

or

```
-e_medium_write=_formatted_write
```

To use the full version, remove the line.

Customizing printf

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

For such applications, a much reduced version of the `printf` function (without `sprintf`) is supplied in source form in the file `intwri.c`. This file can be modified to meet your requirements, and the compiled module inserted into the library in place of the original file; see *Overriding library modules*, page 93.

FORMATTERS USED BY SCANF AND SSCANF

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

_medium_read

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

Specifying the scanf formatter in the IAR Embedded Workbench IDE

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

Specifying the read formatter from the command line

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

```
-e_medium_read=_formatted_read
```

To use the full version, remove the line.

System startup and termination

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

The code for handling startup and termination is located in the source files `cstartup.s51`, `cmain.s51`, `cexit.s51`, and `low_level_init.c` located in the `8051\src\lib` directory.

Note: Normally, there is no need for customizing either of the files `cmain.s51` or `cexit.s51`.

SYSTEM STARTUP

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

For the hardware intialization, it looks like this:

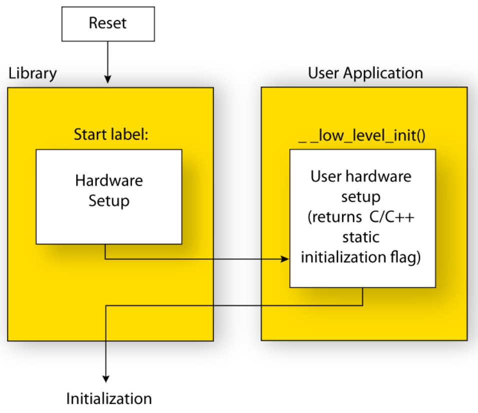


Figure 12: Target hardware initialization phase

- When the cpu is reset a jump will be performed to the program entry label `__program_start` in the system startup code
- If the idata stack is used, the stack pointer, `SP`, is initialized to the beginning of the `ISTACK` segment. If the extended stack is used, the extended stack pointer `?ESP:SP` is initialized to the beginning of the `EXT_STACK` segment
- If the xdata reentrant calling convention is available, the xdata stack pointer, `XSP`, is initialized to the end of the `XSTACK` segment
- If the pdata reentrant calling convention is available, the pdata stack pointer, `PSP`, is initialized to the end of the `PSTACK` segment
- If code banking is used, the bank register is initialized to zero
- The register bank switch register is initialized to the number specified in the linker command file (`?REGISTER_BANK`)
- The `PDATA` page is initialized
- If multiple data pointers are available, the `DPTR` selector register is initialized and the first data pointer (`dptr0`) is set to be the active data pointer
- The custom function `__low_level_init` is called, giving the application a chance to perform early initializations

For the C initialization, it looks like this:

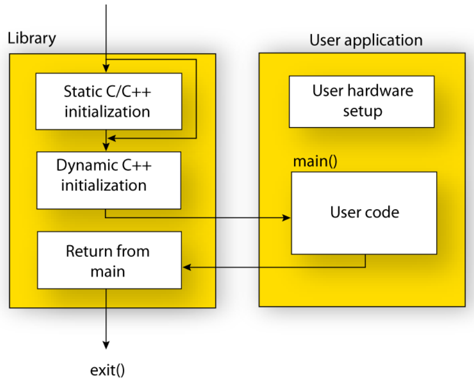


Figure 13: C initialization phase

- Static variables are initialized; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the remaining initialized variables
- The `main` function is called, which starts the application.

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

SYSTEM TERMINATION

An application can terminate normally in two different ways:

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

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

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

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

Overriding default library modules

The IAR CLIB Library contains modules which you probably need to override with your own customized modules, for example for character-based I/O, without rebuilding the entire library. For information about how to override default library modules, see *Overriding library modules*, page 93, in the chapter *The DLIB runtime environment*.

Customizing system initialization

For information about how to customize system initialization, see *Customizing system initialization*, page 99.

C-SPY runtime interface

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

THE DEBUGGER TERMINAL I/O WINDOW

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

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

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

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

TERMINATION

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

Checking module consistency

For information about how to check module consistency, see *Checking module consistency*, page 111.

Assembler language interface

When you develop an application for an embedded system, there may be situations where you will find it necessary to write parts of the code in assembler, for example when using mechanisms in the MCS-51 microcontroller that require precise timing and special instruction sequences.

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

Finally, the chapter covers how functions are called in the different code models, the different memory access methods corresponding to the supported memory types, and how you can implement support for call frame information in your assembler routines for use in the C-SPY® Call Stack window.

Mixing C and assembler

The 8051 IAR C/C++ Compiler provides several ways to mix C or C++ and assembler:

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

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

INTRINSIC FUNCTIONS

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

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

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

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

MIXING C AND ASSEMBLER MODULES

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

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

There will be some overhead in the form of a function call and return instruction sequences, and the compiler will regard some registers as scratch registers. However, the compiler will also assume that all scratch registers are destroyed by an inline assembler instruction. In many cases, the overhead of the extra instructions is compensated by the work of the optimizer.

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

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

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

The first issue is discussed in the section *Calling assembler routines from C*, page 128. The following two are covered in the section *Calling convention*, page 131.

The section on memory access methods, page 142, covers how data in memory is accessed.

The answer to the final question is that the call stack can be displayed when you run assembler code in the debugger. However, the debugger requires information about the *call frame*, which must be supplied as annotations in the assembler source file. For more information, see *Call frame information*, page 144.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 128, and *Calling assembler routines from C++*, page 130, respectively.

INLINE ASSEMBLER

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

```
__no_init __bit bool flag;

void foo(void)
{
  while (!flag)
  {
    asm("MOV C,0x98.0");  /* SCON.R1 */
    asm("MOV flag,C");
  }
}
```

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

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

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

- The compiler's various optimizations will disregard any effects of the inline sequences, which will not be optimized at all
- In general, assembler directives will cause errors or have no meaning. Data definition directives will work as expected
- Auto variables cannot be accessed
- Labels cannot be declared.

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

Calling assembler routines from C

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

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

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

  or

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

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

CREATING SKELETON CODE

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

The following example shows how to create skeleton code to which you can easily add the functional body of the routine. The skeleton source code only needs to declare the variables required and perform simple accesses to them.

In this example, the assembler routine takes an `int` and a `char`, and then returns an `int`:

```
extern int gInt;
extern char gChar;

int func(int arg1, char arg2)
{
  int locInt = arg1;
  gInt = arg1;
  gChar = arg2;
  return locInt;
}

int main()
{
  int locInt = gInt;
  gInt = func(locInt, gChar);
  return 0;
}
```

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

COMPILING THE CODE

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

Use the following options to compile the skeleton code:

```
icc8051 skeleton -lA .
```

The `-lA` option creates an assembler language output file including C or C++ source lines as assembler comments. The `.` (period) specifies that the assembler file should be named in the same way as the C or C++ module (`skeleton`), but with the filename extension `s51`. Also remember to specify the code model, data model, and calling convention you are using as well as a low level of optimization and `-e` for enabling language extensions.

The result is the assembler source output file `skeleton.s51`.

Note: The `-lA` option creates a list file containing call frame information (`CFI`) directives, which can be useful if you intend to study these directives and how they are used. If you only want to study the calling convention, you can exclude the CFI directives from the list file. In the IAR Embedded Workbench IDE, select **Project>Options>C/C++ Compiler>List** and deselect the suboption **Include compiler runtime information**. On the command line, use the option `-lB` instead of `-lA`. Note that CFI information must be included in the source code to make the C-SPY Call Stack window work.

The output file

The output file contains the following important information:

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

The `CFI` directives describe the call frame information needed by the Call Stack window in the IAR C-SPY Debugger.

Calling assembler routines from C++

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

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

However, when using C linkage, the calling convention conforms to the C calling convention. An assembler routine may therefore be called from C++ when declared in the following manner:

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

Memory access layout of non-PODs (“plain old data structures”) is not defined, and may change between compiler versions. Therefore, we do not recommend that you access non-PODs from assembler routines.

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

```
class X;

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

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

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

Note: Support for C++ names from assembler code is extremely limited. This means that:

- Assembler list files resulting from compiling C++ files cannot, in general, be passed through the assembler.
- It is not possible to refer to or define C++ functions that do not have C linkage in assembler.

Calling convention

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

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

This section describes the calling conventions used by the 8051 IAR C/C++ Compiler. The following items are examined:

- Choosing a calling convention
- Function declarations
- C and C++ linkage
- Preserved versus scratch registers

- Function entrance
- Function exit
- Return address handling.

At the end of the section, some examples are shown to describe the calling convention in practice.

CHOOSING A CALLING CONVENTION

The 8051 IAR C/C++ Compiler supports different calling conventions that control how memory is used for parameters and locally declared variables. You can specify a default calling convention or you can explicitly declare the calling convention for each function.

The 8051 IAR C/C++ Compiler supports six different calling conventions:

- Data overlay
- Idata overlay
- Idata reentrant
- Pdata reentrant
- Xdata reentrant
- Extended stack reentrant.

For information about choosing a specific calling convention and when to use a specific calling convention, see *Auto variables—stack and static overlay*, page 35.

Hints for using a calling convention

The calling convention can be very complex and if you intend to use it for your assembler routines, you should create a list file and see how the compiler assigns the different parameters to the available registers. For an example, see *Creating skeleton code*, page 128.

If you intend to use a certain calling convention, you should also specify a value to the runtime model attribute `__rt_version` using the `RTMODEL` assembler directive:

```
RTMODEL "__rt_version"="value"
```

The parameter `value` should have the same value as the one used internally by the compiler. For information about what value to use, see the generated list file. If the calling convention changes in future compiler versions, the runtime model value used internally by the compiler will also change. Using this method gives a module consistency check as the linker will produce an error if there is a mismatch between the values.

For more information about checking module consistency, see *Checking module consistency*, page 111.

FUNCTION DECLARATIONS

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

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

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

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

USING C LINKAGE IN C++ SOURCE CODE

In C++, a function can have either C or C++ linkage. To call assembler routines from C++, it is easiest if you make the C++ function have C linkage.

The following is an example of a declaration of a function with C linkage:

```
extern "C"
{
   int f(int);
}
```

It is often practical to share header files between C and C++. The following is an example of a declaration that declares a function with C linkage in both C and C++:

```
#ifdef __cplusplus
extern "C"
{
#endif

   int f(int);

#ifdef __cplusplus
}
#endif
```

PRESERVED VERSUS SCRATCH REGISTERS

The general MCS-51 CPU registers are divided into three separate sets, which are described in this section.

Scratch registers

Any function is permitted to destroy the contents of a scratch register. If a function needs the register value after a call to another function, it must store it during the call, for example on the stack.

The following registers are scratch registers for all calling conventions: `A`, `B`, `R0`–`R5`, and the `carry` flag.

In addition, the `DPTR` register (or the first `DPTR` register if there are more than one) is a scratch register for all calling conventions except the xdata reentrant calling convention and for banked routines. The `DPTR` register is also considered to be a scratch register if a function is called indirectly, that is, via a function pointer.

Preserved registers

Preserved registers, on the other hand, are preserved across function calls. The called function may use the register for other purposes, but must save the value prior to use and restore it at the exit of the function.

For all calling conventions, all registers that are not scratch registers are preserved registers. These are `R6`, `R7`, all virtual registers (`V0`–`V`*n*) used by the application, the bit register `VB`, and the `DPTR` register in the xdata reentrant calling convention and for banked functions. However, the `DPTR` register (or the first `DPTR` register if there are more than one) is not a preserved register if the function is called indirectly.

Note: If you are using multiple `DPTR` registers, all except the first one are *always* preserved.

Special registers

For some registers there are certain prerequisites that you must consider:

- In the Banked code model, the default bank-switching routine uses the SFR port `P1` as a bank switch register. For more details, see *Bank switching in the Banked code model*, page 63.
- In the Banked extended2 code model, the default bank-switching routine uses the `MEX1` register and the memory extension stack. For more details, see *Bank switching in the Banked extended2 code model*, page 64.

FUNCTION ENTRANCE

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

- Structure types: `struct`, `union`, and classes

- Unnamed parameters to variable length functions; in other words, functions declared as `foo(`*`param1`*`, ...)`, for example `printf`.

Note: Interrupt functions cannot take any parameters.

Hidden parameters

In addition to the parameters visible in a function declaration and definition and independently of the used calling convention there can be hidden parameters:

- If the function returns a structure, the memory location where to store the structure is passed as the last function parameter. The size of the hidden pointer depends on the calling convention used.
- If the function is a non-static Embedded C++ member function, then the `this` pointer is passed as the first parameter. The reason for the requirement that the member function must be non-static is that static member methods do not have a `this` pointer.

Register parameters

Independently of the used calling convention, the five registers `R1`–`R5` are available for register parameters. Each register can contain an 8-bit value and a combination of the registers can contain larger values. Bit parameters are passed in register `B`, starting with `B.0`, `B.1`, etc. If more than eight bit parameters are required, up to eight more are passed in the `VB` register.

The parameters can be passed in the following registers and register combinations:

Parameters	Passed in registers
1-bit values	`B.0`, `B.1`, `B.2`, `B.3`, `B.4`, `B.5`, `B.6`, `B.7`, `VB.0`, `VB.1`, `VB.2`, `VB.3`, `VB.4`, `VB.5`, `VB.6`, or `VB.7`
8-bit values	`R1`, `R2`, `R3`, `R4`, or `R5`
16-bit values	`R3:R2` or `R5:R4`
24-bit values	`R3:R2:R1`
32-bit values	`R5:R4:R3:R2`

Table 30: Registers used for passing parameters

The assignment of registers to parameters is a straightforward process. Traversing the parameters in strict order from left to right, the first parameter is assigned to the first available register or registers. Should there be no suitable register available, the parameter is passed on the stack or overlay frame—depending on used calling convention.

Stack parameters and layout

Stack parameters are stored in memory starting at the location pointed to by the stack pointer specified by the calling convention. The first stack parameter is stored directly after the location pointed to by the stack pointer. The next one is stored directly after the first one, etc.

The idata and extended stacks grow towards a higher address and the pdata and xdata stacks grow towards a lower address.

When the stack parameters have been pushed on the stack, just before the `LCALL` instruction is executed, the stack looks like this:

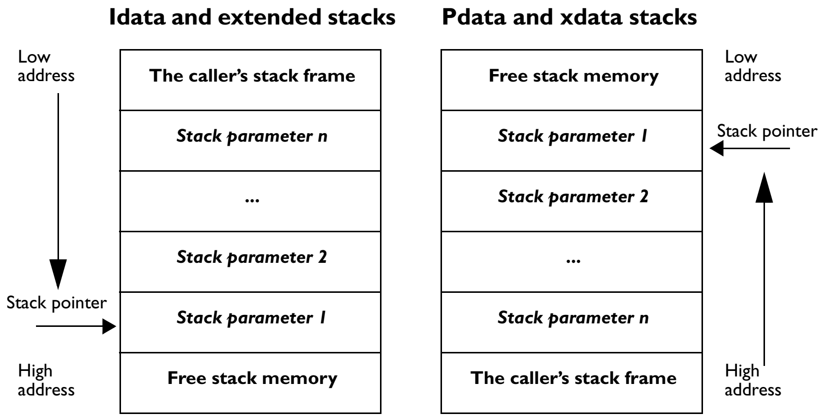


Figure 14: MCS-51 stacks

Note:

- Static overlay functions do not use a stack. Instead non-register parameters are stored on the overlay frame.
- For banked function calls in the Banked extended2 code model, the most significant byte of the return address is pushed on the memory extension stack. The two lower bytes are pushed on the idata stack. For more details, see *Bank switching in the Banked extended2 code model*, page 64.

FUNCTION EXIT

A function can return a value to the function or program that called it, or it can have the return type `void`.

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

Registers used for returning values

For all calling conventions, scalar return values are passed in registers or in the carry bit. The following registers or register combinations are used for the return value:

Return values	Passed in registers
1-bit values	`Carry`
8-bit values	`R1`
16-bit values	`R3:R2`
24-bit values	`R3:R2:R1`
32-bit values	`R5:R4:R3:R2`

Table 31: Registers used for returning values

Returning structures

If a structure is returned, the caller passes a pointer to a location where the called function should store the result. The pointer is passed as an implicit last argument to the function. The called function returns the pointer to the returned value in the same way as for other scalar results.

The location is allocated by the caller on the caller's stack—which depends on the current calling convention—and the called function refers to this location with a default pointer. The default pointer being used depends on the data model. For more information, see *Choosing a calling convention*, page 35.

Stack layout

It is the responsibility of the calling function to clean the stack.

For banked function calls, the return address passed on the stack, can be 3 bytes instead of 2 bytes. For more information, see *Bank switching*, page 62.

Return address handling

A function written in assembler language should, when finished, return to the caller. The location of a function's return address will vary with the calling convention of the function:

Calling convention	Location of return address	Returns using †
Data overlay	The MCS-51 call stack located in idata memory	Assembler-written exit routine

Table 32: Registers used for returning values

Calling convention	Location of return address	Returns using †
Idata overlay	The MCS-51 call stack located in idata memory	Assembler-written exit routine
Idata reentrant	The MCS-51 call stack located in idata memory	Assembler-written exit routine
Pdata reentrant	Moves the return address from the call stack to the emulated pdata stack. However, very simple pdata reentrant routines use the idata stack instead. Interrupt routines always use the idata stack for the return address.	Assembler-written exit routine
Xdata reentrant	Moves the return address from the call stack to the emulated xdata stack. However, very simple xdata reentrant routines use the idata stack instead. Interrupt routines always use the idata stack for the return address.	Assembler-written exit routine
Extended stack reentrant	The extended call stack located in external memory,	Assembler-written exit routine

Table 32: Registers used for returning values (Continued)

† Functions declared `__monitor` return in the same way as normal functions, depending on used calling convention. Interrupt routines always returns using the `RETI` instruction.

EXAMPLES

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

Example 1

Assume that we have the following function declaration:

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

This function takes one parameter in the register pair `R3:R2`, and the return value is passed back to its caller in the register pair.

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

```
MOV     A,R2
ADD     A,#1
MOV     R2,A
MOV     A,R3
ADDC    A,#0
MOV     R3,A
RET
```

Example 2

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

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

The calling function must reserve four bytes on the top of the stack and copy the contents of the `struct` to that location. The character parameter `y` is passed in the register `R1`. The return value is passed back to its caller in the register pair `R3:R2`.

Example 3

The function below will return a `struct`.

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

It is the responsibility of the calling function to allocate a memory location for the return value and pass a pointer to it as a hidden last parameter. The parameter `x` will be passed in the register pair `R3:R2` because `x` is the first parameter and `R3:R2` the first available register pair in the set used for 16-bit register parameters.

The hidden parameter is passed as a second argument to the function. The size of the argument depends on the size of the pointer that refers to the temporary stack location where the return value will be stored. For example, if the function uses the idata or pdata reentrant calling convention, the 8-bit pointer will be located in register `R1`. If instead the xdata reentrant calling convention is used, the 16-bit pointer will be passed in the register pair `R5:R4`. If the extended stack reentrant calling convention is used the pointer is passed on the stack, when some of the registers required to pass a 24-bit value (`R3:R2:R1`) are already occupied.

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

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

In this case, the return value is a scalar, so there is no hidden parameter. The parameter `x` is passed in `R3:R2`, and the return value is returned in `R1`, `R3:R2`, or `R3:R2:R1` depending on the data model used.

FUNCTION DIRECTIVES

The function directives `FUNCTION`, `ARGFRAME`, `LOCFRAME`, and `FUNCALL` are generated by the 8051 IAR C/C++ Compiler to pass information about functions and function calls to the IAR XLINK Linker. These directives can be seen if you use the compiler option **Assembler file** (`-lA`) to create an assembler list file.

Note: These directives are primarily intended to support static overlay.

For reference information about the function directives, see the *8051 IAR Assembler Reference Guide*.

Calling functions

In this section, we describe how functions are called in the different code models.

There are two fundamentally different ways to call functions—directly or via a function pointer. For each code model, we will discuss how both types of calls will be performed.

ASSEMBLER INSTRUCTIONS USED FOR CALLING FUNCTIONS

This section presents the assembler instructions that can be used for calling and returning from functions on the MCS-51 microcontroller.

Normally when calling a function, the `LCALL` assembler instruction is used:

```
LCALL label
```

The location the called function returns to (that is, the location immediately after the `LCALL` instruction) is stored on the idata call stack, unless the extended stack is used in which case the location is stored on the extended stack.

The following sections illustrates how the different code models performs function calls.

Near and Far code model

A direct call using these code models is simply:

```
LCALL function
```

Note: *`function`* is a 16-bit address in the Near code model and a 24-bit address in the Far code model.

When a function returns control to the caller, the `RET` instruction is used.

When a function call is made via a function pointer in the Near code model, the following assembler code is generated:

```
MOV    DPL,#(func&0xFF)       ; Low function address to DPL
MOV    DPH,#((func>>8)&0xFF)  ; High function address to DPH
LCALL  ?CALL_IND              ; Call library function in which
                              ; the function call is made
```

When a function call is made via a function pointer in the Far code model, the following assembler code is generated:

```
MOV    DPL,#(func&0xFF)       ; Low function address to DPL
MOV    DPH,#((func>>8)&0xFF)  ; High function address to DPH
MOV    DPX,#((func>>16)&0xFF) ; Highest function address to DPX
LCALL  ?CALL_IND              ; Call library function in which
                              ; the function call is made
```

Banked code model

In the Banked code model, a direct call translates to the following assembler code:

```
LCALL ??function?relay
```

The call will also generate a relay function `??function?relay` which has a 2-byte address:

```
??function?relay
        LCALL   ?BDISPATCH    ; Call library function in which
                              ; the function call is made
        DATA
        DC24    function      ; Full 3-byte function address
```

A banked indirect function call translates to the following assembler code:

```
MOV    DPL,#(??function?relay&0xFF)
MOV    DPH,#((??function?relay>>8)&0xFF)
LCALL  ?CALL_IND              ; Call library function in which
                              ; the function call is made
```

Banked extended2 code model

When using the Banked extended2 code model, a direct call translates to the following assembler code:

```
MOV     ?MEX1,((function>>16)&0xFF)
LCALL   (function 0xFF)
```

An indirect function call looks like this:

```
MOV    R1,(function&0xFF)
MOV    R2,((function>>8)&0xFF)
MOV    R3,((function>>16)&0xFF)
MOV    DPL,R1
MOV    DPH,R2
LCALL  ?CALL_IND_EXT2         ; Call library function in which
                              ; the function call is made
```

Memory access methods

This section describes the different memory types presented in the chapter *Data storage*. In addition to just presenting the assembler code used for accessing data, it will be used for explaining the reason behind the different memory types.

You should be familiar with the MCS-51 instruction set, in particular the different addressing modes used by the instructions that can access memory.

For each of the access methods described in the following sections, there will be two examples:

- Accessing a global variable
- Accessing memory using a pointer. (This method is used when accessing local data.)

DATA ACCESS METHOD

The data memory is the first 128 bytes of the internal data memory. This memory can be accessed using both direct and indirect addressing modes.

Examples

Accessing the global variable `x`:

```
MOV     A,x
```

Access through a pointer, where `R0` contains a pointer referring to `x`:

```
MOV     A,@R0
```

IDATA ACCESS METHOD

The idata memory consists of the whole internal data memory. This memory can only be accessed using the indirect addressing mode.

Examples

Accessing the global variable `x`:

```
MOV     R0,#x  ; R0 is loaded with the address of x
MOV     A,@R0
```

Access through a pointer, where `R0` contains a pointer referring to `x`:

```
MOV     A,@R0
```

PDATA ACCESS METHOD

The pdata memory consists of a 256-byte block of external memory (xdata memory). The data page that should be used—the high byte of the 2-byte address—can be specified in the linker command file by redefining the symbols _PDATA_START and _PDATA_END. The SFR register that contains the value of the data page can also be specified in the linker command file by changing the definition of the ?PBANK symbol. Pdata accesses can only be made using the indirect addressing mode.

Examples

Accessing the global variable x:

```
MOV     R0,#x
```

Access through a pointer, where R0 contains a pointer referring to x:

```
MOVX    A,@R0
```

XDATA ACCESS METHOD

The xdata memory consists of up to 64 Kbytes of the external memory. Xdata accesses can only be made using the indirect addressing mode with the DPTR register:

```
MOV DPTR,#X
MOV A,@A+DPTR
```

FAR AND HUGE ACCESS METHODS

The far memory and the huge memory consist of up to 16 Mbytes of external memory (extended xdata memory). Far and huge memory accesses are performed using 3-byte data pointers in the same way as the xdata accesses. A far pointer is restricted to a 2-byte offset; this restricts the maximum data object size to 64 Kbytes. The huge pointer has no restrictions; it is a 3-byte pointer with a 3-byte offset.

Examples using the xdata, far, or huge access method

Accessing the global variable x:

```
MOV     DPTR,#x
MOVX    A,@DPTR
```

Access through a pointer, where DPTR contains a pointer referring to x:

```
MOVX    A,@DPTR
```

GENERIC ACCESS METHOD

The generic access method only applies to pointers. If a `__generic` attribute is applied to a non-pointer data object it will be interpreted as`__xdata`. When the generic data model is used, the default pointer is `__generic` but the default data memory attribute is `__xdata`. A generic pointer is 3 bytes in size. The most significant byte reveals if the pointer points to idata, xdata or code memory, and the two least significant bytes specify the address in that memory.

The most significant byte in a generic pointer is coded in the following way:

Data type pointed to	Coding of most significant byte
`idata`	`0x01`
`xdata`	`0x00`
`code`	`0x80`

Table 33: Coding of the most significant byte in a generic pointer

Generic pointers are very flexible and easy to use, but this flexibility comes at the price of reduced execution speed and increased code size. Use generic pointers with caution and only in controlled environments.

The register triplets `R3:R2:R1` and `R6:R5:R4` are used for accessing and manipulating generic pointers.

Accesses to and manipulation of generic pointers are often performed by library routines. If, for example, the register triplet `R3:R2:HR1` contains a generic pointer to the `char` variable `x`, the value of the variable is loaded into register `A` with a call to the library routine `?C_GPRT_LOAD`. This routine decodes the memory that should be accessed and loads the contents into register `A`.

Call frame information

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

If you want the call stack to be available when debugging a routine written in assembler language, you must supply equivalent debug information in your assembler source using the assembler directive `CFI`. This directive is described in detail in the *8051 IAR Assembler Reference Guide*.

The CFI directives will provide C-SPY with information about the state of the calling function(s). Most important of this is the return address, and the value of the stack pointer at the entry of the function or assembler routine. Given this information, C-SPY can reconstruct the state for the calling function, and thereby unwind the stack.

A full description about the calling convention may require extensive call frame information. In many cases, a more limited approach will suffice. The `cstartup` routine and the assembler version of `__low_level_init` both include basic call frame information sufficient to trace the call chain, but do not attempt to trace the values of registers in calling functions. The common definitions for the call frame information used by these routines can be found in the file `iar_cfi.h`, which is provided as source code. These definitions can serve as an introduction and guide to providing call frame information for your own assembler routines.

For an example of a complete implementation of call frame information, you may write a C function and study the assembler language output file. See *Calling assembler routines from C*, page 128.

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

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

The header file `iar_cfi.m51` contains the macros `CFI_names` and `CFI_common` which declare a typical names block and a typical common block. These two macros declare a number of resources, both concrete and virtual. The following are of particular interest:

Resource	**Description**
`A`, `B`, `PSW`, `DPS`	Resources located in SFR memory space
`DPL0`, `DPH0`, `DPX0`	Parts of the ordinary `DPTR` register (`DPX0` only if a 3-byte `DPTR` is used)
`DPL1`–`DPL7`	The low part for additional `DPTR` registers
`DPH1`–`DPH7`	The high part for additional `DPTR` registers
`DPX1`–`DPX7`	The extra part for additional `DPTR` registers if 3-byte `DPTR`s are used
`R0`–`R7`	Register `R0`–`R7`. The locations for these registers might change at runtime

Table 34: Resources for call-frame information

Resource	Description
VB	Virtual register for holding 8-bit variables
V0–V31	Virtual registers located in DATA memory space
SP	Stack pointer to the stack in IDATA memory
ESP	Extended stack pointer to the stack in XDATA memory
ESP16	A concatenation of ESP and SP where SP contains the low byte and ESP the high byte
PSP	Stack pointer to the stack in PDATA memory
XSP	Stack pointer to the stack in XDATA memory
?RET_EXT	Third byte of the return address (for the far code model)
?RET_HIGH	High byte of the return address
?RET_LOW	Low byte of the return address
?RET	A concatenation of ?RET_LOW, ?RET_HIGH and—if the far code model is used—?RET_EXT
C, BR0, BR1, BR2, BR3, BR4, BR5, BR6, BR7, BR8, BR9, BR10, BR11, BR12, BR13, BR14, BR15	Bit registers

Table 34: Resources for call-frame information (Continued)

You should track at least ?RET, so that you can find your way back into the call stack. Track R0–R7, V0–V31, and all available stack pointers to see the values of local variables.

If your application uses more than one register bank, you must track PSW.

Example

The following is an example of an assembler routine that stores a tracked register (R4) on the stack and uses CFI directives to annotate the changes on stack and annotate the changes of locations of R4.

```
#include "iar_cfi.h"

        NAME      cfiexample
        PUBLIC    cfiexample
        FUNCTION cfiexample,0203H

        RSEG NEAR_CODE:CODE:NOROOT(0)
cfiexample:
        CFI BLOCK cfiexample_block USING cfi_common ; Note:
                         cfi_common is defined in iar_cfi.h
```

```
CFI Function cfiexample

MOV       A,R4
CFI       R4 A

PUSH      A
CFI R4 Frame(CFA_MACHINE,3,1)
CFI CFA_MACHINE MACHINE_SP - 3

/* Perform som actions to demonstrate the call stack.*/

MOV       R4,#0xa
MOV       A,#0x0
MOV       R3,A
MOV       A,R4
MOV       R2,A

POP       A
CFI CFA_MACHINE MACHINE_SP - 2
CFI R4 A

MOV       R4,A
CFI R4 SAMEVALUE
RET
CFI EndBlock cfiexample_block
END
```

Using C++

IAR Systems supports two levels of the C++ language: The industry-standard Embedded C++ and IAR Extended Embedded C++. They are described in this chapter.

Overview

Embedded C++ is a subset of the C++ programming language which is intended for embedded systems programming. It was defined by an industry consortium, the Embedded C++ Technical Committee. Performance and portability are particularly important in embedded systems development, which was considered when defining the language.

STANDARD EMBEDDED C++

The following C++ features are supported:

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

C++ features which have been excluded are those that introduce overhead 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 because they represent 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 existing development tools.

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

- Templates
- Multiple and virtual inheritance
- Exception handling

- Runtime type information
- New cast syntax (the operators `dynamic_cast`, `static_cast`, `reinterpret_cast`, and `const_cast`)
- Namespaces
- The `mutable` attribute.

The exclusion of these language features makes the runtime library significantly more efficient. The Embedded C++ library furthermore differs from the full C++ library in that:

- The standard template library (STL) is excluded
- Streams, strings, and complex numbers are supported without the use of templates
- Library features which relate to exception handling and runtime type information (the headers `except`, `stdexcept`, and `typeinfo`) are excluded.

Note: The library is not in the `std` namespace, because Embedded C++ does not support namespaces.

EXTENDED EMBEDDED C++

IAR Systems' Extended EC++ is a slightly larger subset of C++ which adds the following features to the standard EC++:

- Full template support
- Namespace support
- The `mutable` attribute
- The cast operators `static_cast`, `const_cast`, and `reinterpret_cast`.

All these added features conform to the C++ standard.

To support Extended EC++, this product includes a version of the standard template library (STL), in other words, the C++ standard chapters utilities, containers, iterators, algorithms, and some numerics. This STL has been tailored for use with the Extended EC++ language, which means that there are no exceptions, no multiple inheritance, and no support for runtime type information (`rtti`). Moreover, the library is not in the `std` namespace.

Note: A module compiled with Extended EC++ enabled is fully link-compatible with a module compiled without Extended EC++ enabled.

ENABLING C++ SUPPORT

In the 8051 IAR C/C++ Compiler, the default language is C. To be able to compile files written in Embedded C++, you must use the `--ec++` compiler option. See *--ec++*, page 201. You must also use the IAR DLIB runtime library.

To take advantage of *Extended* Embedded C++ features in your source code, you must use the `--eec++` compiler option. See *--eec++*, page 201.

To set the equivalent option in the IAR Embedded Workbench IDE, select **Project>Options>C/C++ Compiler>Language**.

Feature descriptions

When writing C++ source code for the 8051 IAR C/C++ Compiler, there are some benefits and some possible quirks that you need to be aware of when mixing C++ features—such as classes, and class members—with IAR language extensions, such as IAR-specific attributes.

CLASSES

A class type `class` and `struct` in C++ can have static and non-static data members, and static and non-static function members. The non-static function members can be further divided into virtual function members, non-virtual function members, constructors, and destructors. For the static data members, static function members, and non-static non-virtual function members the same rules apply as for statically linked symbols outside of a class. In other words, they can have any applicable IAR-specific type, memory, and object attribute.

The non-static virtual function members can have any applicable IAR-specific type, memory, and object attribute as long as a pointer to the member function can be implicitly converted to the default function pointer type. The constructors, destructors, and non-static data members cannot have any IAR attributes.

The location operator `@` can be used on static data members and on any type of function members.

For further information about attributes, see *Type qualifiers*, page 225.

Example

```
class A {
  public:
   static __xdata __no_init int i @ 60; //Located in xdata at
                                        //address 60
   static __near_func void f(); //Located in code memory
   __near_func void g();        //Located in code memory
   virtual __near_func void h();//Located in code memory
};
virtual void m() const volatile @ "SPECIAL" = 0; //m() placed in
                                                   SPECIAL
```

The this pointer

The `this` pointer used for referring to a class object or calling a member function of a class object will by default have the data memory attribute for the default data pointer type. This means that such a class object can only be defined to reside in memory from which pointers can be implicitly converted to a default data pointer. This restriction may also apply to objects residing on a stack, for example temporary objects and auto objects.

Example

```
class B {
  public:
    void f();
    int i;
};
```

Class memory

To compensate for this limitation, a class can be associated with a *class memory type*. The class memory type changes:

- the `this` pointer type in all member functions, constructors, and destructors into a pointer to class memory
- the default memory for static storage duration variables—that is, not auto variables—of the class type, into the specified class memory
- the pointer type used for pointing to objects of the class type, into a pointer to class memory.

Example

```
class __xdata C {
  public:
    void f();       // Has a this pointer of type C __xdata *
    void f() const; // Has a this pointer of type
                    // C __xdata const *
    C();            // Has a this pointer pointing into xdata
                    // memory
    C(C const &);   // Takes a parameter of type C __xdata
                    // const &
                    // (also true of generated copy constructor)
    int i;
};
C Ca;               // Resides in xdata memory instead of the
                    // default memory
C __pdata Cb;       // Resides in pdata memory, the 'this'
                    // pointer still points into xdata memory
```

```
C __idata Cc;         // Not allowed, __idata pointer can't be
                      // implicitly converted into a __xdata
                      // pointer
void h()
{
  C Cd;               // Resides on the stack
}
C * Cp1;              // Creates a pointer to xdata memory
C __pdata * Cp2;      // Creates a pointer to pdata memory
```

Note: Whenever a class type associated with a class memory type, like `C`, must be declared, the class memory type must be mentioned as well:

```
class __xdata C;
```

Also note that class types associated with different class memories are not compatible types.

There is a built-in operator that returns the class memory type associated with a class, `__memory_of(`*`class`*`)`. For instance, `__memory_of(C)` returns `__xdata`.

When inheriting, the rule is that it must be possible to convert implicitly a pointer to a subclass into a pointer to its base class. This means that a subclass can have a *more* restrictive class memory than its base class, but not a *less* restrictive class memory.

```
class __xdata D : public C { // OK, same class memory
  public:
    void g();
    int j;
};

class __pdata E : public C { // OK, pdata memory is inside xdata
  public:
    void g()  // Has a this pointer pointing into pdata memory
    {
      f();    // Gets a this pointer into xdata memory
    }
    int j;
};

class __idata F : public C { // Not OK, idata memory isn’t
                             // inside xdata memory
  public:
    void g();
    int j;
};
```

```
class G : public C { // OK, will be associated with same class
                     // memory as C
  public:
    void g();
    int j;
};
```

A `new` expression on the class will allocate memory in the heap residing in the class memory. A `delete` expression will naturally deallocate the memory back to the same heap. To override the default `new` and `delete` operator for a class, declare

```
void *operator new(size_t);
void operator delete(void *);
```

as member functions, just like in ordinary C++.

If a pointer to class memory cannot be implicitly casted into a default pointer type, no temporaries can be created for that class, for instance if you have an xdata default pointer, the following example will not work:

```
class __idata Foo {...}
void some_fun (Foo arg) {...}
Foo another_fun (int x) {...}
```

For more information about memory types, see *Memory types*, page 24.

FUNCTIONS

A function with `extern "C"` linkage is compatible with a function that has C++ linkage.

Example

```
extern "C" {
  typedef void (*fpC)(void); // A C function typedef
}
void (*fpCpp)(void);         // A C++ function typedef

fpC f1;
fpCpp f2;
void f(fpC);

f(f1);                       // Always works
f(f2);                       // fpCpp is compatible with fpC
```

NEW AND DELETE OPERATORS

There are operators for `new` and `delete` for each memory that can have a heap, that is, xdata, far, and huge memory.

These examples assume that there is a heap in both far and huge memory.

```
void __huge * operator new __huge(__huge_size_t);
void __far  * operator new __far (__far_size_t);
void operator delete(void __huge *);
void operator delete(void __far  *);
```

And correspondingly for array `new` and `delete` operators:

```
void __huge * operator new[] __huge(__huge_size_t);
void __far  * operator new[] __far (__far_size_t);
void operator delete[](void __huge *);
void operator delete[](void __far  *);
```

Use this syntax if you want to override both global and class-specific `operator new` and `operator delete` for any data memory.

Note that there is a special syntax to name the `operator new` functions for each memory, while the naming for the `operator delete` functions relies on normal overloading.

New and delete expressions

A `new` expression calls the `operator new` function for the memory of the type given. If a `class`, `struct`, or `union` type with a class memory is used, the class memory will determine the `operator new` function called. For example,

```
//Calls operator new __huge(__huge_size_t)
int __huge *p = new __huge int;

//Calls operator new __huge(__huge_size_t)
int __huge *q = new int __huge;

//Calls operator new[] __huge(__huge_size_t)
int __huge *r = new __huge int[10];

//Calls operator new __far(__far_size_t)
class __far S{...};
S *s = new S;
```

A `delete` expression calls the `operator delete` function that corresponds to the argument given. For example,

```
delete p; //Calls operator delete(void __huge *)
delete s; //Calls operator delete(void __far  *)
```

Note that the pointer used in a `delete` expression must have the correct type, that is, the same type as that returned by the `new` expression. If you use a pointer to the wrong memory, the result might be a corrupt heap.

For example,

```
int __huge * t = new __far int;
delete t; //Error: Causes a corrupt heap
```

TEMPLATES

Extended EC++ supports templates according to the C++ standard, except for the support of the `export` keyword. The implementation uses a two-phase lookup which means that the keyword `typename` has to be inserted wherever needed. Furthermore, at each use of a template, the definitions of all possible templates must be visible. This means that the definitions of all templates have to be in include files or in the actual source file.

Templates and data memory attributes

For data memory attributes to work as expected in templates, two elements of the standard C++ template handling have been changed—class template partial specialization matching and function template parameter deduction.

In Extended Embedded C++, the class template partial specialization matching algorithm works like this:

When a pointer or reference type is matched against a pointer or reference to a template parameter type, the template parameter type will be the type pointed to, stripped of any data memory attributes, if the resulting pointer or reference type is the same.

The comments in the example program sequences correspond to the template instantiation.

Example

```
// Here we instantiate four different objects.
template<typename> class Z;
template<typename T> class Z<T *>;
```

Assume that xdata is the default data pointer:

```
Z<int __pdata *> zn;    // T = int __pdata
Z<int __xdata  *> zf;   // T = int __xdata
Z<int        *> zd;     // T = int __xdata
```

Assume that pdata is instead the default pointer:

```
Z<int __xdata *> zh;   // T = int __xdata
```

In Extended Embedded C++, the function template parameter deduction algorithm works like this:

When function template matching is performed and an argument is used for the deduction; if that argument is a pointer to a memory that can be implicitly converted to a default pointer, do the parameter deduction as if it was a default pointer.

When an argument is matched against a reference, do the deduction as if the argument and the parameter were both pointers.

Example

```
template<typename T> void fun(T *);

fun((int __pdata *) 0); // T = int __pdata. The result is
                        // different than the analogous situation
                        // with class template specializations.
fun((int        *) 0); // T = int __xdata
fun((int __xdata  *) 0); // T = int __xdata
```

Assume that pdata is instead the default pointer:

```
fun((int __xdata *) 0); // T = int __xdata
```

For templates that are matched using this modified algorithm, it is impossible to get automatic generation of special code for pointers to *small* memory types. For *large* and “other” memory types (memory that cannot be pointed to by a default pointer) it is possible. In order to make it possible to write templates that are fully memory-aware—in the rare cases where this is useful—use the `#pragma basic_template_matching` directive in front of the template function declaration. That template function will then match without the modifications described above.

Example

```
#pragma basic_template_matching
template<typename T> void fun(T *);
```

Assume that xdata is the default pointer:

```
fun((int __pdata *) 0); // T = int __pdata
```

Non-type template parameters

It is allowed to have a reference to a memory type as a template parameter, even if pointers to that memory type are not allowed.

Example

```
extern int __sfr x;

template<__sfr int &y>
void foo()
{
  y = 17;
}

void bar()
{
  foo<x>();
}
```

The standard template library

The STL (standard template library) delivered with the product is tailored for Extended EC++, as described in *Extended Embedded C++*, page 150.

The containers in the STL, like `vector` and `map`, are memory attribute aware. This means that a container can be declared to reside in a specific memory type which has the following consequences:

- The container itself will reside in the chosen memory
- Allocations of elements in the container will use a heap for the chosen memory
- All references inside it use pointers to the chosen memory.

Example

```
vector<int> d;                 // d placed in default memory, using
                               // the default heap, uses default
                               // pointers
vector<int __xdata> __xdata x; // x placed in xdata memory, heap
                               // allocation from xdata, uses
                               // pointers to xdata memory
vector<int __xdata> __pdata y; // y placed in pdata memory, heap
                               // allocation from xdata, uses
                               // pointers to xdata memory
vector<int __pdata> __xdata z; // Illegal, because the heap
                                 // allocated in pdata uses pdata
                                 // pointers that cannot reach the
                                 // variable z, located in __xdata
                                 // memory
```

Note that `map<key, T>`, `multimap<key, T>`, `hash_map<key, T>`, and `hash_multimap<key, T>` all use the memory of `T`. This means that the `value_type` of these collections will be `pair<key, const T>` *`mem`* where *`mem`* is the memory type of `T`. Supplying a key with a memory type is not useful.

Note that two containers that only differ by the data memory attribute they use cannot be assigned to each other.

Example

```
vector<int __pdata> x;
vector<int __xdata> y;

x = y; // Illegal
y = x; // Illegal
```

However, the templated assign member method will work:

```
x.assign(y.begin(), y.end());
y.assign(x.begin(), x.end());
```

STL and the IAR C-SPY® Debugger

C-SPY has built-in display support for the STL containers.C-SPY has built-in display support for the STL containers. The logical structure of containers is presented in the watch views in a comprehensive way that is easy to understand and follow.

Note: To be able to watch STL containers with many elements in a comprehensive way, the **STL container expansion** option—available by choosing **Tools>Options>Debugger**—is set to display only a small number of items at first.

VARIANTS OF CASTS

In Extended EC++ the following additional C++ cast variants can be used:

```
  const_cast<t2>(t), static_cast<t2>(t), reinterpret_cast<t2>(t).
```

MUTABLE

The `mutable` attribute is supported in Extended EC++. A `mutable` symbol can be changed even though the whole class object is `const`.

NAMESPACE

The namespace feature is only supported in *Extended* EC++. This means that you can use namespaces to partition your code. Note, however, that the library itself is not placed in the `std` namespace.

THE STD NAMESPACE

The `std` namespace is not used in either standard EC++ or in Extended EC++. If you have code that refers to symbols in the `std` namespace, simply define `std` as nothing; for example:

```
#define std  // Nothing here
```

POINTER TO MEMBER FUNCTIONS

A pointer to a member function can only contain a default function pointer, or a function pointer that can implicitly be casted to a default function pointer. To use a pointer to a member function, make sure that all functions that should be pointed to reside in the default memory or a memory contained in the default memory.

Example

```
class X{
public:
  __near_func void f();
};
void (__near_func X::*pmf)(void) = &X::f;
```

USING INTERRUPTS AND EC++ DESTRUCTORS

If interrupts are enabled and the interrupt functions use class objects that have destructors, there may be problems if the program exits either by using `exit` or by returning from `main`. If an interrupt occurs after an object has been destroyed, there is no guarantee that the program will work properly.

To avoid this, make sure that interrupts are disabled when returning from `main` or when calling `exit` or `abort`.

To avoid interrupts, place a call to the intrinsic function `__disable_interrupt` before the call to `_exit`.

C++ language extensions

When you use the compiler in C++ mode and have enabled IAR language extensions, the following C++ language extensions are available in the compiler:

- In a `friend` declaration of a class, the `class` keyword may be omitted, for example:

  ```
  class B;
  class A
  {
    friend B;       //Possible when using IAR language
                    //extensions
    friend class B; //According to standard
  };
  ```

- Constants of a scalar type may be defined within classes, for example:

  ```
  class A {
    const int size = 10;//Possible when using IAR language
                        //extensions
    int a[size];
  };
  ```

  According to the standard, initialized static data members should be used instead.

- In the declaration of a class member, a qualified name may be used, for example:

  ```
  struct A {
    int A::f(); //Possible when using IAR language extensions
    int f();    //According to standard
  };
  ```

- It is permitted to use an implicit type conversion between a pointer to a function with C linkage (`extern "C"`) and a pointer to a function with C++ linkage (`extern "C++"`), for example:

  ```
  extern "C" void f();//Function with C linkage
  void (*pf) ()       //pf points to a function with C++ linkage
                = &f; //Implicit conversion of pointer.
  ```

  According to the standard, the pointer must be explicitly converted.

- If the second or third operands in a construction that contains the `?` operator are string literals or wide string literals (which in C++ are constants), the operands may be implicitly converted to `char *` or `wchar_t *`, for example:

  ```
  char *P = x ? "abc" : "def"; //Possible when using IAR
                               //language extensions
  char const *P = x ? "abc" : "def"; //According to standard
  ```

- Default arguments may be specified for function parameters not only in the top-level function declaration, which is according to the standard, but also in `typedef` declarations, in pointer-to-function function declarations, and in pointer-to-member function declarations.

- In a function that contains a non-static local variable and a class that contains a non-evaluated expression (for example a `sizeof` expression), the expression may reference the non-static local variable. However, a warning is issued.

Note: If you use any of these constructions without first enabling language extensions, errors are issued.

Efficient coding for embedded applications

For embedded systems, the size of the generated code and data is very important, because using smaller external memory or on-chip memory can significantly decrease the cost and power consumption of a system.

The topics discussed are:

- Selecting data types
- Controlling data and function placement in memory
- Controlling compiler optimizations
- Writing efficient code.

As a part of this, the chapter also demonstrates some of the more common mistakes and how to avoid them, and gives a catalog of good coding techniques.

Selecting data types

For efficient treatment of data, you should consider the data types used and the most efficient placement of the variables.

USING EFFICIENT DATA TYPES

The data types you use should be considered carefully, because this can have a large impact on code size and code speed.

- Use small and unsigned data types, (`unsigned char` and `unsigned short`) unless your application really requires signed values.
- Bitfields with sizes other than 1 bit should be avoided because they will result in inefficient code compared to bit operations.
- Declaring a pointer to `const` data tells the calling function that the data pointed to will not change, which opens for better optimizations.

For details about representation of supported data types, pointers, and structures types, see the chapter *Data representation*.

FLOATING-POINT TYPES

The 8051 IAR C/C++ Compiler only supports the 32-bit floating-point format. The 64-bit floating-point format (`double`) is not supported. All types of the 64-bit floating-point format will be treated as 32-bit floats. Also consider replacing code using floating-point operations with code using integers because these are more efficient.

For more information about floating-point types, see *Floating-point types*, page 220.

USING THE BEST POINTER TYPE

The generic pointers can point to all memory spaces, which makes them simple and also tempting to use. However, they carry a cost in that special code is needed before each pointer access to check which memory the pointer points to and performing appropriate actions. Use the smallest pointer type you can, and avoid any generic pointers.

ANONYMOUS STRUCTS AND UNIONS

When a structure or union is declared without a name, it becomes anonymous. The effect is that its members will only be seen in the surrounding scope.

Anonymous structures are part of the C++ language; however, they are not part of the C standard. In the 8051 IAR C/C++ Compiler they can be used in C if language extensions are enabled.

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

Use the `-e` compiler option to enable language extensions. See *-e*, page 200, for additional information.

Example

In the following example, the members in the anonymous `union` can be accessed, in function `f`, without explicitly specifying the `union` name:

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

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

The member names must be unique in the surrounding scope. Having an anonymous `struct` or `union` at file scope, as a global, external, or static variable is also allowed. This could for instance be used for declaring I/O registers, as in the following example:

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

This declares an I/O register byte `IOPORT` at *`address`*. The I/O register has 2 bits declared, `way` and `out`. Note that both the inner structure and the outer union are anonymous.

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

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

Anonymous structures and unions are implemented in terms of objects named after the first field, with a prefix `_A_` to place the name in the implementation part of the namespace. In this example, the anonymous union will be implemented through an object named `_A_IOPORT`.

Controlling data and function placement in memory

The compiler provides different mechanisms for controlling placement of functions and data objects in memory. To use memory efficiently, you should be familiar with these mechanisms to know which one is best suited for different situations. You can use:

- Code and data models

  Use the different compiler options for code and data models, respectively, to take advantage of the different addressing modes available for the microcontroller and thereby also place functions and data objects in different parts of memory. To read more about data and code models, see *Data models*, page 22, and *Code models for function storage*, page 47, respectively.

- Memory attributes

  Use memory attributes to override the default addressing mode and placement of individual functions and data objects. To read more about memory attributes for data and functions, see *Using data memory attributes*, page 30, and *Code models for function storage*, page 47, respectively.

- Calling convention

  The 8051 IAR C/C++ Compiler provides six different *calling conventions* that control how memory is used for parameters and locally declared variables. You can specify a default calling convention or you can explicitly declare the calling convention for each individual function. To read more, see *Choosing a calling convention*, page 35.

- The `@` operator and the `#pragma location` directive for absolute placement

  Use the `@` operator or the `#pragma location` directive to place individual global and static variables at absolute addresses. The variables must be declared either `__no_init` or `const`. This is useful for individual data objects that must be located at a fixed address, for example variables with external requirements, or for populating any hardware tables similar to interrupt vector tables. Note that it is not possible to use this notation for absolute placement of individual functions.

- The `@` operator and the `#pragma location` directive for segment placement

  Use the `@` operator or the `#pragma location` directive to place groups of functions or global and static variables in named segments, without having explicit control of each object. The variables must be declared either `__no_init` or `const`. The segments can, for example, be placed in specific areas of memory, or initialized or copied in controlled ways using the segment begin and end operators. This is also useful if you want an interface between separately linked units, for example an application project and a boot loader project. Use named segments when absolute control over the placement of individual variables is not needed, or not useful.

At compile time, data and functions are placed in different segments as described in *Data segments*, page 72, and *Code segments*, page 81, respectively. At link time, one of the most important functions of the linker is to assign load addresses to the various segments used by the application. All segments, except for the segments holding absolute located data, are automatically allocated to memory according to the specifications of memory ranges in the linker command file, as described in *Placing segments in memory*, page 68.

DATA PLACEMENT AT AN ABSOLUTE LOCATION

The `@` operator, alternatively the `#pragma location` directive, can be used for placing global and static variables at absolute addresses. The variables must be declared using one of the following combinations of keywords:

- `__no_init`
- `__no_init` and `const` (whithout initializers)
- `const` (with initializers).

To place a variable at an absolute address, the argument to the `@` operator and the `#pragma location` directive should be a literal number, representing the actual address.

Note: A variable placed in an absolute location should be defined in an include file, to be included in every module that uses the variable. An unused definition in a module will be ignored. A normal `extern` declaration—one that does not use an absolute placement directive—can refer to a variable at an absolute address; however, optimizations based on the knowledge of the absolute address cannot be performed.

Examples

In this example, a `__no_init` declared variable is placed at an absolute address. This is useful for interfacing between multiple processes, applications, etc:

```
__no_init volatile char alpha @ 0x80;/* OK */
```

In the following examples, there are two `const` declared objects, where the first is not initialized, and the second is initialized to a specific value. Both objects are placed in ROM. This is useful for configuration parameters, which are accessible from an external interface. Note that in the second case, the compiler is not obliged to actually read from the variable, because the value is known.

```
#pragma location=0x90
__no_init const int beta;                /* OK */

const int gamma @ 0xA0 = 3;              /* OK */
```

In the first case, the value is not initialized by the compiler; the value must be set by other means. The typical use is for configurations where the values are loaded to ROM separately, or for special function registers that are read-only.

The following example shows incorrect usage:

```
int delta @ 0xB0;                  /* Error, neither */
                                   /* "__no_init" nor "const".*/
```

C++ considerations

In C++, module scoped `const` variables are static (module local), whereas in C they are global. This means that each module that declares a certain `const` variable will contain a separate variable with this name. If you link an application with several such modules all containing (via a header file), for instance, the declaration:

```
volatile const __no_init int x @ 0x100;        /* Bad in C++ */
```

the linker will report that there are more than one variable located at address `0x100`.

To avoid this problem and make the process the same in C and C++, you should declare these variables `extern`, for example:

```
extern volatile const __no_init int x @ 0x100;  /* the extern
                                        keyword makes x public */
```

Note: C++ static member variables can be placed at an absolute address just like any other static variable.

DATA AND FUNCTION PLACEMENT IN SEGMENTS

It is possible to place data or functions in named segments other than default. The `@` operator, alternatively the `#pragma location` directive, can be used for placing individual variables or individual functions in named segments. The named segment can either be a predefined segment, or a user-defined segment. The variables must be declared either `__no_init` or `const`. If declared `const`, they can have initializers.

C++ static member variables can be placed in named segments just like any other static variable.

If you use your own segments, in addition to the predefined segments, the segments must also be defined in the linker command file using the `-Z` or the `-P` segment control directives.

Note: Take care when explicitly placing a variable or function in a predefined segment other than the one used by default. This is useful in some situations, but incorrect placement can result in anything from error messages during compilation and linking to a malfunctioning application. Carefully consider the circumstances; there might be strict requirements on the declaration and use of the function or variable.

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

Examples of placing variables in named segments

In the following three examples, a data object is placed in a user-defined segment. The segment will be allocated in default memory depending on the used data model.

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

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

const int gamma @ "CONSTANTS" = 3;  /* OK */
```

To override the default segment allocation, you can explicitly specify a memory attribute other than the default:

```
__idata __no_init int alpha @ "NOINIT";/* Placed in idata*/
```

The following example shows incorrect usage:

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

Examples of placing functions in named segments

```
void f(void) @ "FUNCTIONS";

void g(void) @ "FUNCTIONS"
{
}

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

To override the default segment allocation, you can explicitly specify a memory attribute other than the default:

```
__near_func void f(void) @ "FUNCTIONS";
```

Controlling compiler optimizations

The compiler performs many transformations on your application in order to generate the best possible code. Examples of such transformations are storing values in registers instead of memory, removing superfluous code, reordering computations in a more efficient order, and replacing arithmetic operations by cheaper operations.

The linker should also be considered an integral part of the compilation system, because there are some optimizations that are performed by the linker. For instance, all unused functions and variables are removed and not included in the final output.

SCOPE FOR PERFORMED OPTIMIZATIONS

You can decide whether optimizations should be performed on your whole application or on individual files. By default, the same types of optimizations are used for an entire project, but you should consider using different optimization settings for individual files. For example, put code that must execute very quickly into a separate file and compile it for minimal execution time, and the rest of the code for minimal code size. This will give a small program, which is still fast enough where it matters.

In addition, you can exclude individual functions from the performed optimizations. The `#pragma optimize` directive allows you to either lower the optimization level, or specify another type of optimization to be performed. Refer to *optimize*, page 267, for information about the pragma directive.

OPTIMIZATION LEVELS

The 8051 IAR C/C++ Compiler supports different levels of optimizations. The following table lists the optimizations that are performed on each level:

Optimization level	Description
None (Best debug support)	Variables live through their entire scope
Low	Same as above but variables only live for as long as they are needed, not necessarily through their entire scope Dead code elimination Redundant label elimination Redundant branch elimination
Medium	Same as above Live-dead analysis and optimization Code hoisting Register content analysis and optimization Common subexpression elimination
High (Balanced)	Same as above Peephole optimization Cross jumping Cross call (when optimizing for size) Loop unrolling Function inlining Code motion Type-based alias analysis

Table 35: Compiler optimization levels

Note: Some of the performed optimizations can be individually enabled or disabled. For more information about these, see *Fine-tuning enabled transformations*, page 171.

A high level of optimization might result in increased compile time, and will most likely also make debugging more difficult, because it will be less clear how the generated code relates to the source code. For example, at the low, medium, and high optimization levels, variables do not live through their entire scope, which means processor registers used for storing variables can be reused immediately after they were last used. Due to this, the C-SPY Watch window might not be able to display the value of the variable throughout its scope. At any time, if you experience difficulties when debugging your code, try lowering the optimization level.

SPEED VERSUS SIZE

At the high optimization level, the compiler balances between size and speed optimizations. However, it is possible to fine-tune the optimizations explicitly for either size or speed. They only differ in what thresholds that are used; speed will trade size for speed, whereas size will trade speed for size. Note that one optimization sometimes enables other optimizations to be performed, and an application may in some cases become smaller even when optimizing for speed rather than size.

FINE-TUNING ENABLED TRANSFORMATIONS

At each optimization level you can disable some of the transformations individually. To disable a transformation, use either the appropriate option, for instance the command line option `--no_inline`, alternatively its equivalent in the IAR Embedded Workbench IDE **Function inlining**, or the `#pragma optimize` directive. The following transformations can individually be disabled:

- Common subexpression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis.

Common subexpression elimination

Redundant re-evaluation of common subexpressions is by default eliminated at optimization levels **Medium** and **High**. This optimization normally reduces both code size and execution time. However, the resulting code might be difficult to debug.

Note: This option has no effect at optimization levels **None** and **Low**.

To read more about the command line option, see *--no_cse*, page 207.

Loop unrolling

It is possible to duplicate the loop body of a small loop, whose number of iterations can be determined at compile time, to reduce the loop overhead.

This optimization, which can be performed at optimization level **High**, normally reduces execution time, but increases code size. The resulting code might also be difficult to debug.

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed.

Note: This option has no effect at optimization levels **None**, **Low**, and **Medium**.

To read more about the command line option, see *--no_unroll*, page 209.

Function inlining

Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call. This optimization, which is performed at optimization level **High**, normally reduces execution time, but increases code size. The resulting code might also be difficult to debug.

The compiler decides which functions to inline. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed.

Note: This option has no effect at optimization levels **None**, **Low**, and **Medium**.

To read more about the command line option, see *--no_inline*, page 207.

Code motion

Evaluation of loop-invariant expressions and common subexpressions are moved to avoid redundant re-evaluation. This optimization, which is performed at optimization level **High**, normally reduces code size and execution time. The resulting code might however be difficult to debug.

Note: This option has no effect at optimization levels **None**, and **Low**.

Type-based alias analysis

When two or more pointers reference the same memory location, these pointers are said to be *aliases* for each other. The existence of aliases makes optimization more difficult because it is not necessarily known at compile time whether a particular value is being changed.

Type-based alias analysis optimization assumes that all accesses to an object will take place using its declared type or as a `char` type. This assumption lets the compiler detect whether pointers may reference the same memory location or not.

Type-based alias analysis is performed at optimization level **High**. For ISO/ANSI standard-conforming C or C++ application code, this optimization can reduce code size and execution time. However, non-standard-conforming C or C++ code might result in the compiler producing code that leads to unexpected behavior. Therefore, it is possible to turn this optimization off.

Note: This option has no effect at optimization levels **None**, **Low**, and **Medium**.

To read more about the command line option, see *--no_tbaa*, page 208.

Example

```
short f(short * p1, long * p2)
{
  *p2 = 0;
  *p1 = 1;
  return *p2;
}
```

With type-based alias analysis, it is assumed that a write access to the `short` pointed to by `p1` cannot affect the `long` value that `p2` points to. Thus, it is known at compile time that this function returns 0. However, in non-standard-conforming C or C++ code these pointers could overlap each other by being part of the same union. By using explicit casts, you can also force pointers of different pointer types to point to the same memory location.

Writing efficient code

This section contains general programming hints on how to implement functions to make your applications robust, but at the same time facilitate compiler optimizations.

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

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions may modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. There are two main reasons why this is inefficient. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code.

Second, the optimizer can no longer assume that the local variable is unaffected over function calls.

- Module-local variables—variables that are declared static—are preferred over global variables. Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions. This means that instead of calling a function, the compiler inserts the content of the function at the location where the function was called. The result is a faster, but often larger, application. Also, inlining may enable further optimizations. The compiler often inlines small functions declared static. The use of the `#pragma inline` directive and the C++ keyword `inline` gives you fine-grained control, and it is the preferred method compared to the traditional way of using preprocessor macros. This feature can be disabled using the `--no_inline` command line option; see *--no_inline*, page 207.
- Avoid using inline assembler. Instead, try writing the code in C or C++, use intrinsic functions, or write a separate module in assembler language. For more details, see *Mixing C and assembler*, page 125.

SAVING STACK SPACE AND RAM MEMORY

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

- If stack space is limited, avoid long call chains and recursive functions.
- Avoid using large non-scalar types, such as structures, as parameters or return type; in order to save stack space, you should instead pass them as pointers or, in C++, as references.
- Use the smallest possible data type (and signed data types only when necessary)
- Declare variables with a short life span as auto variables. When the life spans for these variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution. Be careful with auto variables, though, as the stack size can exceed its limits.

CALLING CONVENTIONS

The 8051 IAR C/C++ Compiler supports several calling conventions, using different types of stacks. Try to use the smallest possible calling convention. The data overlay, idata overlay, and idata reentrant calling conventions generate the most efficient code. Pdata reentrant and extended stack reentrant functions add some overhead and xdata reentrant functions even more.

Because the xdata stack pointer and the extended stack pointer are larger than 8 bits, they must be updated using two instructions. To make the system interrupt safe, interrupts must be disabled while the stack pointer is updated. This generates an overhead if you are using an xdata or extended stack.

Normally, it is enough to use the default calling convention. However, in some cases it is better to explicitly declare functions of another calling convention, for example:

- Some large and stack-intensive functions do not fit within the limited restrictions of a smaller calling convention. This function can then be declared to be of a larger calling convention
- A large system that uses a limited number of small and important routines can have these declared with a smaller calling convention for efficiency.

Note: Some restrictions apply when you mix different calling conventions. See *Mixing calling conventions*, page 38.

FUNCTION PROTOTYPES

It is possible to declare and define functions using one of two different styles:

- Prototyped
- Kernighan & Ritchie C (K&R C)

Both styles are included in the C standard; however, it is recommended to use the prototyped style, since it makes it easier for the compiler to find problems in the code. In addition, using the prototyped style will make it possible to generate more efficient code, since type promotion (implicit casting) is not needed. The K&R style is only supported for compatibility reasons.

To make the compiler verify that all functions have proper prototypes, use the compiler option **Require prototypes** (`--require_prototypes`).

Prototyped style

In prototyped function declarations, the type for each parameter must be specified.

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

Kernighan & Ritchie style

In K&R style—traditional pre-ISO/ANSI C—it is not possible to declare a function prototyped. Instead, an empty parameter list is used in the function declaration. Also, the definition looks different.

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

INTEGER TYPES AND BIT NEGATION

There are situations when the rules for integer types and their conversion lead to possibly confusing behavior. Things to look out for are assignments or conditionals (test expressions) involving types with different size and logical operations, especially bit negation. Here, *types* also includes types of constants.

In some cases there may be warnings (for example, constant conditional or pointless comparison), in others just a different result than what is expected. Under certain circumstances the compiler might warn only at higher optimizations, for example, if the compiler relies on optimizations to identify some instances of constant conditionals. In the following example an 8-bit character, a 16-bit integer, and two's complement is assumed:

```
void f1(unsigned char c1)
{
  if  (c1 == ~0x80)
  ;
}
```

Here, the test is always false. On the right hand side, `0x80` is `0x0080`, and `~0x0080` becomes `0xFF7F`. On the left hand side, `c1` is an 8-bit unsigned character, so it cannot be larger than 255. It also cannot be negative, which means that the integral promoted value can never have the topmost 8 bits set.

PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

Variables that are accessed asynchronously, for example by interrupt routines or by code executing in separate threads, must be properly marked and have adequate protection. The only exception to this is a variable that is always *read-only.*

To mark a variable properly, use the `volatile` keyword. This informs the compiler, among other things, that the variable can be changed from other threads. The compiler will then avoid optimizing on the variable (for example, keeping track of the variable in registers), will not delay writes to it, and be careful accessing the variable only the number of times given in the source code. To read more about the `volatile` type qualifier, see *Declaring objects volatile*, page 225.

A sequence that accesses a `volatile` declared variable must also not be interrupted. This can be achieved by using the `__monitor` keyword in interruptible code. This must be done for both write *and* read sequences, otherwise you might end up reading a partially updated variable. This is true for all variables of all sizes. Accessing a small-sized variable can be an atomic operation, but this is not guaranteed and you should not rely on it unless you continuously study the compiler output. It is safer to use the `__monitor` keyword to ensure that the sequence is an atomic operation.

Protecting the eeprom write mechanism

A typical example of when it can be necessary to use the `__monitor` keyword is when protecting the eeprom write mechanism, which can be used from two threads (for example, main code and interrupts). Servicing an interrupt during an EEPROM write sequence can in many cases corrupt the written data.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of MCS-51 devices are included in the 8051 IAR C/C++ product installation. The header files are named `io`*`device`*`.h` and define the processor-specific special function registers (SFRs).

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

SFRs with bitfields are declared in the header file; the following example shows how the program status register can be declared:

```
__sfr __no_init volatile union
{
  unsigned char PSW;
  struct
  {
    unsigned char CY: 1;
    unsigned char AC: 1;
    unsigned char FO: 1;
    unsigned char RS: 2;
    unsigned char OV: 1;
    unsigned char   : 1;
    unsigned char P : 1;
  } PSW_bit;
} @ 0xD0;
```

By including the appropriate include file in your code, it is possible to access either the whole register or any individual bit (or bitfields) from C code as follows:

```
// whole register access
PSW = 0x12;
```

```
// Bitfield accesses
PSW_bit.AC = 1;
PSW_bit.RS = 3;
```

You can also use the header files as templates when you create new header files for other MCS-51 devices. For details about the @ operator, see *Located data*, page 81.

NON-INITIALIZED VARIABLES

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

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

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

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

For information about the `__no_init` keyword, see page 253. Note that to use this keyword, language extensions must be enabled; see *-e*, page 200. For information about the `#pragma object_attribute`, see page 267.

Part 2. Reference information

This part of the 8051 IAR C/C++ Compiler Reference Guide contains the following chapters:

External interface details

This chapter provides reference information about how the compiler interacts with its environment. The chapter briefly lists and describes the invocation syntax, methods for passing options to the tools, environment variables, the include file search procedure, and finally the different types of compiler output.

Invocation syntax

You can use the compiler either from the IAR Embedded Workbench IDE or from the command line. Refer to the *IAR Embedded Workbench® IDE User Guide* for information about using the compiler from the IAR Embedded Workbench IDE.

COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
icc8051 [options] [sourcefile] [options]
```

For example, when compiling the source file `prog.c`, use the following command to generate an object file with debug information:

```
icc8051 prog --debug
```

The source file can be a C or C++ file, typically with the filename extension `c` or `cpp`, respectively. If no filename extension is specified, the file to be compiled must have the extension `c`.

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

If you run the compiler from the command line without any arguments, the compiler version number and all available options including brief descriptions are directed to `stdout` and displayed on the screen.

PASSING OPTIONS

There are three different ways of passing options to the compiler:

- Directly from the command line

  Specify the options on the command line after the `icc8051` command, either before or after the source filename; see *Invocation syntax*, page 181.

- Via environment variables

  The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 182.
- Via a text file by using the `-f` option; see *-f*, page 202.

For general guidelines for the option syntax, an options summary, and a detailed description of each option, see the *Compiler options* chapter.

ENVIRONMENT VARIABLES

The following environment variables can be used with the 8051 IAR C/C++ Compiler:

Environment variable	Description
C_INCLUDE	Specifies directories to search for include files; for example: `C_INCLUDE=c:\program files\iar systems\embedded workbench 4.`*n*`\8051\inc;c:\headers`
QCCX51	Specifies command line options; for example: `QCCX51=-lA asm.lst`

Table 36: Compiler environment variables

Include file search procedure

This is a detailed description of the compiler's `#include` file search procedure:

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

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

  it searches the following directories for the file to include:

  1 The directories specified with the `-I` option, in the order that they were specified, see *-I*, page 203.

  2 The directories specified using the `C_INCLUDE` environment variable, if any, see *Environment variables*, page 182.
- If the compiler encounters the name of an `#include` file in double quotes, for example:

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

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

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

For example:

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

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

```
icc8051 ..\src\src.c -I..\include -I..\debugconfig
```

Then the following directories are searched in the order listed below for the file `config.h`, which in this example is located in the `dir\debugconfig` directory:

`dir\include`	Current file is `src.h`.
`dir\src`	File including current file (`src.c`).
`dir\include`	As specified with the first `-I` option.
`dir\debugconfig`	As specified with the second `-I` option.

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

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

Compiler output

The compiler can produce the following output:

- A linkable object file

  The object files produced by the compiler use a proprietary format called UBROF, which stands for Universal Binary Relocatable Object Format. By default, the object file has the filename extension `r51`.
- Optional list files

  Different types of list files can be specified using the compiler option `-l`, see *-l*, page 203. By default, these files will have the filename extension `lst`.
- Optional preprocessor output files

  A preprocessor output file is produced when you use the `--preprocess` option; by default, the file will have the filename extension `i`.
- Diagnostic messages

  Diagnostic messages are directed to `stderr` and displayed on the screen, as well as printed in an optional list file. To read more about diagnostic messages, see *Diagnostics*, page 184.

- Error return codes

  These codes provide status information to the operating system which can be tested in a batch file, see *Error return codes*, page 184.

- Size information

  Information about the generated amount of bytes for functions and data for each memory is directed to `stdout` and displayed on the screen. Some of the bytes might be reported as *shared*.

  Shared objects are functions or data objects that are shared between modules. If any of these occur in more than one module, only one copy will be retained. For example, in some cases inline functions are not inlined, which means that they are marked as shared, because only one instance of each function will be included in the final application. This mechanism is sometimes also used for compiler-generated code or data not directly associated with a particular function or variable, and when only one instance is required in the final application.

Error return codes

The 8051 IAR C/C++ Compiler returns status information to the operating system that 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 and the option `--warnings_affect_exit_code` was used.
2	There were errors.
3	There were fatal errors making the compiler abort.
4	There were internal errors making the compiler abort.

Table 37: Error return codes

Diagnostics

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

MESSAGE FORMAT

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

```
filename,linenumber level[tag]: message
```

with the following elements:

`filename`	The name of the source file in which the issue was encountered
`linenumber`	The line number at which the compiler detected the issue
`level`	The level of seriousness of the issue
`tag`	A unique tag that identifies the diagnostic message
`message`	An explanation, possibly several lines long

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

Use the option `--diagnostics_tables` to list all possible compiler diagnostic messages.

SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

Remark

A diagnostic message that is produced when the compiler finds a source code construction that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see *--remarks*, page 214.

Warning

A diagnostic message that is produced when the compiler finds a programming error or omission which is of concern, but not so severe as to prevent the completion of compilation. Warnings can be disabled by use of the command line option `--no_warnings`, see page 209.

Error

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

Fatal error

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

SETTING THE SEVERITY LEVEL

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

See *Summary of compiler options*, page 190, for a description of the compiler options that are available for setting severity levels.

See the chapter *Pragma directives*, for a description of the pragma directives that are available for setting severity levels.

INTERNAL ERROR

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

```
Internal error: message
```

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

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

Compiler options

This chapter describes the syntax of compiler options and the general syntax rules for specifying option parameters, and gives detailed reference information about each option.

Options syntax

Compiler options are parameters you can specify to change the default behavior of the compiler. You can specify options from the command line—which is described in more detail in this section—and from within the IAR Embedded Workbench IDE.

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

TYPES OF OPTIONS

There are two *types of names* for command line options, *short* names and *long* names. Some options have both.

- A short option name consists of one character, and it may have parameters. You specify it with a single dash, for example `-e`
- A long option name consists of one or several words joined by underscores, and it may have parameters. You specify it with double dashes, for example `--char_is_signed`.

For information about the different methods for passing options, see *Passing options*, page 182.

RULES FOR SPECIFYING PARAMETERS

There are some general syntax rules for specifying option parameters. First, the rules depending on whether the parameter is *optional* or *mandatory*, and whether the option has a short or a long name, are described. Then, the rules for specifying filenames and directories are listed. Finally, the remaining rules are listed.

Rules for optional parameters

For options with a short name and an optional parameter, any parameter should be specified without a preceding space, for example:

`-O` or `-Oh`

For options with a long name and an optional parameter, any parameter should be specified with a preceding equal sign (=), for example:

```
--misrac=n
```

Rules for mandatory parameters

For options with a short name and a mandatory parameter, the parameter can be specified either with or without a preceding space, for example:

`-I..\src` or `-I ..\src\`

For options with a long name and a mandatory parameter, the parameter can be specified either with a preceding equal sign (=) or with a preceding space, for example:

```
--diagnostics_tables=filename
```

or

```
--diagnostics_tables filename
```

Rules for options with both optional and mandatory parameters

For options taking both optional and mandatory parameters, the rules for specifying the parameters are:

- For short options, optional parameters are specified without a preceding space
- For long options, optional parameters are specified with a preceding equal sign (=)
- For short and long options, mandatory parameters are specified with a preceding space.

For example, a short option with an optional parameter followed by a mandatory parameter:

```
-lA filename
```

For example, a long option with an optional parameter followed by a mandatory parameter:

```
--preprocess=n filename
```

Rules for specifying a filename or directory as parameters

The following rules apply for options taking a filename or directory as parameters:

- Options that take a filename as a parameter can optionally also take a path. The path can be relative or absolute. For example, to generate a listing to the file `list.lst` in the directory `..\listings\`:

  ```
  icc8051 prog -l ..\listings\list.lst
  ```

- For options that take a filename as the destination for output, the parameter can be specified as a path without a specified filename. The compiler stores the output in that directory, in a file with an extension according to the option. The filename will be the same as the name of the compiled source file, unless a different name has been specified with the option `-o`, in which case that name will be used. For example:

  ```
  icc8051 prog -l ..\listings\
  ```

  The produced list file will have the default name `..\listings\prog.lst`
- The *current directory* is specified with a period (`.`). For example:

  ```
  icc8051 prog -l .
  ```
- `/` can be used instead of `\` as the directory delimiter.
- By specifying `-`, input files and output files can be redirected to `stdin` and `stdout`, respectively. For example:

  ```
  icc8051 prog -l -
  ```

Additional rules

In addition, the following rules apply:

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

  ```
  icc8051 prog -l ---r
  ```
- For options that accept multiple arguments of the same type, the arguments can be provided as a comma-separated list (without a space), for example:

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

  Alternatively, the option may be repeated for each argument, for example:

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

Summary of compiler options

The following table summarizes the compiler command line options:

Command line option	Description
`--calling_convention`	Specifies the calling convention
`--char_is_signed`	Treats `char` as signed
`--code_model`	Specifies the code model
`--core`	Specifies a CPU core
`-D`	Defines preprocessor symbols
`--data_model`	Specifies the data model
`--debug`	Generates debug information
`--dependencies`	Lists file dependencies
`--diag_error`	Treats these as errors
`--diag_remark`	Treats these as remarks
`--diag_suppress`	Suppresses these diagnostics
`--diag_warning`	Treats these as warnings
`--diagnostics_tables`	Lists all diagnostic messages
`--dlib_config`	Determines the library configuration file
`--dptr`	Enables support for multiple data pointers
`-e`	Enables language extensions
`--ec++`	Enables Embedded C++ syntax
`--eec++`	Enables Extended Embedded C++ syntax
`--enable_multibytes`	Enables support for multibyte characters in source files
`--error_limit`	Specifies the allowed number of errors before compilation stops
`--extended_stack`	Specifies use of an extended stack
`-f`	Extends the command line
`--header_context`	Lists all referred source files and header files
`-I`	Specifies include file path
`-l`	Creates a list file
`--library_module`	Creates a library module
`--migration_preprocessor_extensions`	Extends the preprocessor

Table 38: Compiler options summary

Command line option	Description
`--misrac`	Enables MISRA C-specific error messages
`--misrac_verbose`	Enables verbose logging of MISRA C checking
`--module_name`	Sets the object module name
`--no_code_motion`	Disables code motion optimization
`--no_cse`	Disables common subexpression elimination
`--no_inline`	Disables function inlining
`--no_path_in_file_macros`	Removes the path from the return value of the symbols `__FILE__` and `__BASE_FILE__`
`--no_tbaa`	Disables type-based alias analysis
`--no_typedefs_in_diagnostics`	Disables the use of typedef names in diagnostics
`--no_unroll`	Disables loop unrolling
`--no_warnings`	Disables all warnings
`--no_wrap_diagnostics`	Disables wrapping of diagnostic messages
`--nr_virtual_regs`	Sets the work area size
`-O`	Sets the optimization level
`-o`	Sets the object filename
`--omit_types`	Excludes type information
`--only_stdout`	Uses standard output only
`--output`	Sets the object filename
`--place_constants`	Specifies the location of constants and strings
`--preinclude`	Includes an include file before reading the source file
`--preprocess`	Generates preprocessor output
`--public_equ`	Defines a global named assembler label
`-r`	Generates debug information
`--remarks`	Enables remarks
`--require_prototypes`	Verifies that functions are declared before they are defined
`--rom_mon_bp_padding`	Enables setting breakpoints on all C statements when debugging using the generic ROM-monitor.
`--silent`	Sets silent operation
`--strict_ansi`	Checks for strict compliance with ISO/ANSI C
`--warnings_affect_exit_code`	Warnings affects exit code

Table 38: Compiler options summary (Continued)

Command line option	Description
`--warnings_are_errors`	Warnings are treated as errors

Table 38: Compiler options summary (Continued)

Descriptions of options

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

Note that if you use the options page **Extra Options** to specify specific command line options, the IAR Embedded Workbench IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--calling_convention

Syntax

`--calling_convention=`*`convention`*

Parameters

`data_overlay|do`

`idata_overlay|io`

`idata_reentrant|ir`

`pdata_reentrant|pr`

`xdata_reentrant|xr`

`ext_stack_reentrant|er`

Description

Use this option to specify the default calling convention for a module. All runtime modules in an application must use the same calling convention. However, note that it is possible to override this for individual functions, by using keywords.

See also

Choosing a calling convention, page 35.

Project>Options>General Options>Target>Calling convention

--char_is_signed

Syntax `--char_is_signed`

Description By default, the compiler interprets the `char` type as unsigned. Use this option to make the compiler interpret the `char` type as signed instead. This can be useful when you, for example, want to maintain compatibility with another compiler.

Note: The runtime library is compiled without the `--char_is_signed` option. If you use this option, you may get type mismatch warnings from the linker, because the library uses `unsigned char`.

Project>Options>C/C++ Compiler>Language>Plain 'char' is

--code_model

Syntax `--code_model={near|n|banked|b|banked_ext2|b2|far|f}`

Parameters

`near|n`	Allows for up to 64 Kbytes of ROM; default for the core variant Plain.
`banked|b`	Allows for up to 1 Mbyte of ROM via up to sixteen 64-Kbyte banks and one root bank; supports banked 24-bit calls.
`banked_ext2|b2`	Allows for up to 16 Mbytes of ROM via up to sixteen 1-Mbyte banks; supports banked 24-bit calls. Default for the core variant Extended2.
`far|f`	Allows for up to 16 Mbytes of ROM and supports true 24-bit calls. Default for the core variant Extended1.

Description Use this option to select the code model for which the code is to be generated. If you do not choose a code model option, the compiler uses the default code model. Note that all modules of your application must use the same code model.

See also *Code models for function storage*, page 47.

Project>Options>General Options>Target>Code model

--core

Syntax `--core={plain|pl|extended1|e1|extended2|e2}`

Description Use this option to select the MCS-51 core for which the code is to be generated. If you do not use the option to specify a core, the compiler uses the Plain core as default.

Note that all modules of your application must use the same core. The compiler supports the different MCS-51 microcontroller cores and devices based on these cores. The object code that the compiler generates for one core is not binary compatible with code for the other cores.

See also

Basic project settings for hardware memory configuration, page 15 .

Project>Options>General Options>Target>Core

-D

Syntax

`-D symbol[=value]`

Parameters

symbol	The name of the preprocessor symbol
value	The value of the preprocessor symbol

Description

Use this option to define a preprocessor symbol. If no value is specified, `1` is used. This option can be used one or more times on the command line.

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

```
-Dsymbol
```

is equivalent to:

```
#define symbol 1
```

In order to get the equivalence of:

```
#define FOO
```

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

```
-DFOO=
```

Project>Options>C/C++ Compiler>Preprocessor>Defined symbols

--data_model

Syntax

`--data_model={tiny|t|small|s|large|l|far|f|generic|g}`

Parameters

`tiny|t`	Default memory attribute `__data` Default pointer attribute `__idata`

`small|s`	Default memory attribute `__idata` Default pointer attribute `__idata` Default for the core variant Plain
`large|l`	Default memory attribute `__xdata` Default pointer attribute `__xdata` Default for the core variant Extended2
`far|f`	Default memory attribute `__far` Default pointer attribute `__far` Default for the core variant Extended1
`generic|g`	Default memory attribute `__xdata` Default pointer attribute `__generic`

Description

Use this option to select the data model for which the code is to be generated. If you do not choose a data model option, the compiler uses the default data model. Note that all modules of your application must use the same data model.

See also

Data models, page 22.

Project>Options>General Options>Target>Data model

--debug, -r

Syntax

```
--debug
-r
```

Description

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

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

Project>Options>C/C++ Compiler>Output>Generate debug information

--dependencies

Syntax

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

Parameters

`i` (default)	Lists only the names of files
`m`	Lists in makefile style

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 188.

Description

Use this option to make the compiler list all source and header files opened by the compilation into a file with the default filename extension `i`.

Example

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

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

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

```
foo.r51: c:\iar\product\include\stdio.h
foo.r51: d:\myproject\include\foo.h
```

An example of using `--dependencies` with a popular make utility, such as gmake (GNU make):

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

```
%.r51 : %.c
      $(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

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

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

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

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

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

--diag_error

Syntax

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

Parameters

`tag`	The number of a diagnostic message, for example the message number `Pe117`

Description

Use this option to reclassify certain diagnostic messages as errors. An error indicates a violation of the C or C++ language rules, of such severity that object code will not be generated. The exit code will be non-zero. This option may be used more than once on the command line.

Project>Options>C/C++ Compiler>Diagnostics>Treat these as errors

--diag_remark

Syntax

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

Parameters

tag	The number of a diagnostic message, for example the message number `Pe177`

Description

Use this option to reclassify certain diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a source code construction that may cause strange behavior in the generated code. This option may be used more than once on the command line.

Note: By default, remarks are not displayed; use the `--remarks` option to display them.

Project>Options>C/C++ Compiler>Diagnostics>Treat these as remarks

--diag_suppress

Syntax

`--diag_suppress=`*`tag`*`[,`*`tag,...`*`]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`

Description

Use this option to suppress certain diagnostic messages. These messages will not be displayed. This option may be used more than once on the command line.

Project>Options>C/C++ Compiler>Diagnostics>Suppress these diagnostics

--diag_warning

Syntax `--diag_warning=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe826`

Description Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the compiler to stop before compilation is completed. This option may be used more than once on the command line.

Project>Options>C/C++ Compiler>Diagnostics>Treat these as warnings

--diagnostics_tables

Syntax `--diagnostics_tables {filename|directory}`

Parameters For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 188.

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

This option cannot be given together with other options.

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

--dlib_config

Syntax `--dlib_config filename`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 188.

Description Each runtime library requires a library configuration file. Use this option to specify the library configuration file; choose between `dl8051Normal.h` and `dl8051Full.h` depending on which library configuration you are using.

You can find the library object files and the library configuration files in the directory `8051\lib`. For examples and a list of prebuilt runtime libraries, see *Using a prebuilt library*, page 88.

If you build your own customized runtime library, you should also create a corresponding customized library configuration file, which must be specified to the compiler. For more information, see *Building and using a customized library*, page 95.

Note: This option only applies to the IAR DLIB runtime environment.

See also

Library configurations, page 87.

To set related options, choose:

Project>Options>General Options>Library Configuration

--dptr

Syntax

`--dptr={[size][,number][,visibility][,select]}`

Parameters

`size=16|24`	The pointer size in bits. For the Extended1 core, the default value is 24, for the other cores it is 16.
`number=1|2|3|4|5|6|7|8`	The number of data pointers (`DPTR` registers). For the Extended`1` core, the default value is 2, for all other cores it is `1`.
`visibility=separate|shadowed`	If you are using 2 or more data pointers, the `DPTR0` register can either hide (shadow) the other registers, making them unavailable to the compiler, or they can all be visible in separate special function registers. The default visibility is `separate`.
`select=inc|xor(mask)`	Specifies the method for selecting the active data pointer. `XOR` uses the `ORL` or `ANL` instruction to set the active pointer in the data pointer selection register. The bits used are specified in a bit mask. For example, if four data pointers are used and the selection bits are bit 0 and bit 2, the mask should be `0x05` (`00000101` in binary format). Default (`0x01`) for the Plain core. `INC` increments the bits in the data pointer selection register to select the active data pointer. See *Selecting the active data pointer*, page 19. Default for the Extended`1` core.

Description

Use this option to enable support for more than one data pointer; a feature in many MCS-51 devices. You can specify the number of pointers, the size of the pointers, whether they are visible or not, and the method for switching between them.

To use multiple DPTRs, you must specify the location of the DPTR registers and the data pointer selection register (`?DPS`), either in the linker command file or in the IAR Embedded Workbench IDE.

Example

To use two 16-bit data pointers, use:

```
--dptr=2,16
```

In this case, the default value `separate` is used for DPTR visibility and `xor(0x01)` is used for DPTR selection.

To use four 24-bit pointers, all of them visible in separate registers, to be switched between using the `XOR` method, use:

```
--dptr=24,4,separate,xor(0x05)
```

or

```
--dptr=24 --dptr=4 --dptr=separate --dptr=xor(0x05)
```

See also

Code models for function storage, page 47.

Project>Options>General Options>Target>Data Pointer

-e

Syntax

```
-e
```

Description

In the command line version of the 8051 IAR C/C++ Compiler, language extensions are disabled by default. If you use language extensions such as extended keywords and anonymous structs and unions in your source code, you must enable them by using this option.

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

See also

The chapter *Compiler extensions.*

Project>Options>C/C++ Compiler>Language>Allow IAR extensions

Note: By default, this option is enabled in the IAR Embedded Workbench IDE.

--ec++

Syntax `--ec++`

Description In the 8051 IAR C/C++ Compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.

Project>Options>C/C++ Compiler>Language>Embedded C++

--eec++

Syntax `--eec++`

Description In the 8051 IAR C/C++ Compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++.

See also *Extended Embedded C++*, page 150.

Project>Options>C/C++ Compiler>Language>Extended Embedded C++

--enable_multibytes

Syntax `--enable_multibytes`

Description By default, multibyte characters cannot be used in C or C++ source code. Use this option to make multibyte characters in the source code be interpreted according to the host computer's default setting for multibyte support.

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

Project>Options>C/C++ Compiler>Language>Enable multibyte support

--error_limit

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

Parameters

`n`	The number of errors before the compiler stops the compilation. *`n`* must be a positive integer; `0` indicates no limit.

Description Use the `--error_limit` option to specify the number of errors allowed before the compiler stops the compilation. By default, 100 errors are allowed.

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

--extended_stack

Syntax `--extended_stack`

Description Use this option to enable the extended stack that is available if you use an MCS-51 extended device. This option is set by default if the extended stack reentrant calling convention is used. For all other calling conventions, the extended stack option is not set by default.

Note: The extended stack option cannot be used with the idata or xdata stacks, and by implication, neither with the idata reentrant or xdata reentrant calling conventions.

See also *Code models for function storage*, page 47.

Project>Options>General Options>Target>Do not use extended stack

-f

Syntax `-f `*`filename`*

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 188.

Descriptions Use this option to make the compilerread command line options from the named file, with the default filename extension `xcl`.

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

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

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--header_context

Syntax

```
--header_context
```

Description

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

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

-I

Syntax

`-I` *`path`*

Parameters

`path`	The search path for `#include` files

Description

Use this option to specify the search paths for `#include` files. This option may be used more than once on the command line.

See also

Include file search procedure, page 182.

Project>Options>C/C++ Compiler>Preprocessor>Additional include directories

-l

Syntax

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

Parameters

`a`	Assembler list file
`A`	Assembler list file with C or C++ source as comments

`a`	Assembler list file
`b`	Basic assembler list file. This file has the same contents as a list file produced with `-la`, except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *
`B`	Basic assembler list file. This file has the same contents as a list file produced with `-lA`, except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *
`c`	C or C++ list file
`C` (default)	C or C++ list file with assembler source as comments
`D`	C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values
`N`	No diagnostics in file
`H`	Include source lines from header files in output. Without this option, only source lines from the primary source file are included

*** This makes the list file less useful as input to the assembler, but more useful for reading by a human.**

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 188.

Description

Use this option to generate an assembler or C/C++ listing to a file. Note that this option can be used one or more times on the command line.

To set related options, choose:

Project>Options>C/C++ Compiler>List

--library_module

Syntax

```
--library_module
```

Description

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

Project>Options>C/C++ Compiler>Output>Module type>Library Module

--migration_preprocessor_extensions

Syntax	`--migration_preprocessor_extensions`

Description

If you need to migrate code from an earlier IAR Systems C or C/C++ compiler, you may want to use this option. Use this option to use the following in preprocessor expressions:

- Floating-point expressions
- Basic type names and `sizeof`
- All symbol names (including typedefs and variables).

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

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

Project>Options>C/C++ Compiler>Language>Enable IAR migration preprocessor extensions

--misrac

Syntax	`--misrac[={n,o-p,…|all|required}]`

Parameters

`--misrac=n`	Enables checking for the MISRA C rule with number *n*
`--misrac=o,n`	Enables checking for the MISRA C rules with numbers *o* and *n*
`--misrac=o-p`	Enables checking for all MISRA C rules with numbers from *o* to *p*
`--misrac=m,n,o-p`	Enables checking for MISRA C rules with numbers *m*, *n*, and from *o* to *p*
`--misrac=all`	Enables checking for all MISRA C rules
`--misrac=required`	Enables checking for all MISRA C rules categorized as required

Description

Use this option to enable the compiler to check for deviations from the rules described in the MISRA *Guidelines for the Use of the C Language in Vehicle Based Software* (1998). By using one or more arguments with the option, you can restrict the checking to a specific subset of the MISRA C rules. If the compiler is unable to check for a rule, specifying the option for that rule has no effect. For instance, MISRA C rule 15 is a documentation issue, and the rule is not checked by the compiler. As a consequence, specifying `--misrac=15` has no effect.

To set related options, choose:

Project>Options>General Options>MISRA C or **Project>Options>C/C++ Compiler>MISRA C**

--misrac_verbose

Syntax

`--misrac_verbose`

Description

Use this option to generate a MISRA C log during compilation. This is a list of the rules that are enabled—but not necessarily checked—and a list of rules that are actually checked.

If this option is enabled, the compiler display a text at sign-on that shows both enabled and checked MISRA C rules.

Project>Options>General Options>MISRA C>Log MISRA C Settings

--module_name

Syntax

`--module_name=`*`name`*

Parameters

`name`	An explicit object module 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 explicitly.

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

Project>Options>C/C++ Compiler>Output>Object module name

--no_code_motion

Syntax `--no_code_motion`

Description Use this option to disable code motion optimizations. These optimizations, which are performed at the optimization levels Medium and High, normally reduce code size and execution time. However, the resulting code may be difficult to debug.

Note: This option has no effect at optimization levels below Medium.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Code motion

--no_cse

Syntax `--no_cse`

Description Use this option to disable common subexpression elimination. At the optimization levels Medium and High, the compiler avoids calculating the same expression more than once. This optimization normally reduces both code size and execution time. However, the resulting code may be difficult to debug.

Note: This option has no effect at optimization levels below Medium.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination

--no_inline

Syntax `--no_inline`

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

This optimization, which is performed at optimization level High, normally reduces execution time and increases code size. The resulting code may also be difficult to debug.

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

Note: This option has no effect at optimization levels below High.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Function inlining

--no_path_in_file_macros

Syntax `--no_path_in_file_macros`

Description Use this option to exclude the path from the return value of the predefined preprocessor symbols `__FILE__` and `__BASE_FILE__`.

See also *Descriptions of predefined preprocessor symbols*, page 278.

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

--no_tbaa

Syntax `--no_tbaa`

Description Use this option to disable type-based alias analysis. When this options is not used, the compiler is free to assume that objects are only accessed through the declared type or through `unsigned char`.

See also *Type-based alias analysis*, page 172.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Type-based alias analysis

--no_typedefs_in_diagnostics

Syntax `--no_typedefs_in_diagnostics`

Description Use this option to disable the use of typedef names in diagnostics. Normally, when a type is mentioned in a message from the compiler, most commonly in a diagnostic message of some kind, the typedef names that were used in the original declaration are used whenever they make the resulting text shorter.

Example

```
typedef int (*MyPtr)(char const *);
MyPtr p = "foo";
```

will give an error message like the following:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "MyPtr"
```

If the `--no_typedefs_in_diagnostics` option is used, the error message will be like this:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "int (*)(char const *)"
```

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_unroll

Syntax `--no_unroll`

Description

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

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

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

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

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed and size.

Note: This option has no effect at optimization levels below High.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling

--no_warnings

Syntax `--no_warnings`

Description

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

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

--no_wrap_diagnostics

Syntax `--no_wrap_diagnostics`

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

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

--nr_virtual_regs

Syntax `--nr_virtual_regs=n`

Parameter

n	The size of the work area; a value between 8 and 32.

Description Use this option to specify the size of the work area. The virtual registers are located in data memory.

See also *Virtual registers*, page 44.

Project>Options>General Options>Target>Number of virtual registers

-O

Syntax `-O[n|l|m|h|hs|hz]`

Parameters

`n`	None* (Best debug support)
`l` (default)	Low*
`m`	Medium
`h`	High, balanced
`hs`	High, favoring speed
`hz`	High, favoring size

***The most important difference between None and Low is that at None, all non-static variables will live during their entire scope.**

Description

Use this option to set the optimization level to be used by the compiler when optimizing the code. If no optimization option is specified, the optimization level Low is used by default. If only `-O` is used without any parameter, the optimization level High balanced is used.

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

See also

Controlling compiler optimizations, page 169.

Project>Options>C/C++ Compiler>Optimizations

-o, --output

Syntax

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

Parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 188.

Description

By default, the object code output produced by the compiler is located in a file with the same name as the source file, but with the extension `r51`. Use this option to explicitly specify a different output filename for the object code output.

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

--omit_types

Syntax

```
--omit_types
```

Description

By default, the compiler includes type information about variables and functions in the object output. Use this option if you do not want the compiler to include this type information in the output, which is useful when you build a library that should not contain type information. The object file will then only contain type information that is a part of a symbol's name. This means that the linker cannot check symbol references for type correctness.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--only_stdout

Syntax `--only_stdout`

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

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

--output, -o

Syntax

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

Parameters For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 188.

Description By default, the object code output produced by the compiler is located in a file with the same name as the source file, but with the extension `r51`. Use this option to explicitly specify a different output filename for the object code output.

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

--place_constants

Syntax `--place_constants={data|data_rom|code}`

Parameters

`data` (default)	Copies constants and strings from code memory to data memory. The specific data memory depends on the default data model.
`data_rom`	Places constants and strings in xdata or far memory, depending on the data model, in a range where ROM is located. In the xdata data model the objects are placed in xdata memory and in the far data model they are placed in far memory. In the rest of the data models, the `data_rom` modifier is not allowed.
`code`	Places constants and strings in code memory. In this case, the prebuilt runtime libraries cannot be used as is.

Description Use this option to specify the default location for constants and strings. The default location can be overridden for individual constants and strings by use of keywords.

See also

Constants and strings, page 34.

Project>Options>General Options>Target>Location for constants and strings

--preinclude

Syntax

`--preinclude includefile`

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 188.

Description

Use this option to make the compiler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.

Project>Options>C/C++ Compiler>Preprocessor>Preinclude file

--preprocess

Syntax

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

Parameters

`c`	Preserve comments
`n`	Preprocess only
`l`	Generate `#line` directives

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 188.

Description

Use this option to generate preprocessed output to a named file.

Project>Options>C/C++ Compiler>Preprocessor>Preprocessor output to file

--public_equ

Syntax

`--public_equ symbol[=value]`

Parameters

`symbol`	The name of the assembler symbol to be defined
`value`	An optional value of the defined assembler symbol

Description

This option is equivalent to defining a label in assembler language using the `EQU` directive and exporting it using the `PUBLIC` directive. This option may be used more than once on the command line.

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

-r, --debug

Syntax

```
-r
--debug
```

Description

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

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

Project>Options>C/C++ Compiler>Output>Generate debug information

--remarks

Syntax

```
--remarks
```

Description

The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks.

See also

Severity levels, page 185.

Project>Options>C/C++ Compiler>Diagnostics>Enable remarks

--require_prototypes

Syntax

```
--require_prototypes
```

Description

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

- A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration
- A function definition of a public function with no previous prototype declaration

- An indirect function call through a function pointer with a type that does not include a prototype.

Note: This option only applies to functions in the C standard library.

Project>Options>C/C++ Compiler>Language>Require prototypes

--rom_mon_bp_padding

Syntax

```
--rom_monitor_bp_padding
```

Description

Use this option to enable setting breakpoints on all C statements when using the generic C-SPY ROM-monitor debugger.

When the C-SPY ROM-monitor sets a breakpoint, it replaces the original instruction with the 3-byte instruction `LCALL monitor`. For those cases where the original instruction has a different size than three bytes, the compiler will insert extra `NOP` instructions (pads) to ensure that all jumps to this destination are correctly aligned.

Note: This mechanism is only supported for breakpoints that you set on C-statement level. For breakpoints in assembler code, you have to add pads manually.

Project>Options>C/C++ Compiler>Code>Padding for ROM-monitor breakpoints

--silent

Syntax

```
--silent
```

Description

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

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

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

--strict_ansi

Syntax `--strict_ansi`

Description By default, the compiler accepts a relaxed superset of ISO/ANSI C/C++, see *Minor language extensions*, page 233. Use this option to ensure that the program conforms to the ISO/ANSI C/C++ standard.

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

Project>Options>C/C++ Compiler>Language>Language conformances>Strict ISO/ANSI

--warnings_affect_exit_code

Syntax `--warnings_affect_exit_code`

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

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

--warnings_are_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. Warnings that have been changed into remarks are not treated as errors.

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

See also *diag_warning*, page 264.

Project>Options>C/C++ Compiler>Diagnostics>Treat all warnings as errors

Data representation

This chapter describes the data types, pointers, and structure types supported by the 8051 IAR C/C++ Compiler.

See the chapter *Efficient coding for embedded applications* for information about which data types and pointers provide the most efficient code for your application.

Alignment

Every C data object has an alignment that controls how the object can be stored in memory. Should an object have an alignment of, for example, 4, it must be stored on an address that is divisible by 4.

The reason for the concept of alignment is that some processors have hardware limitations for how the memory can be accessed.

Assume that a processor can read 4 bytes of memory using one instruction, but only when the memory read is placed on an address divisible by 4. Then, 4-byte objects, such as `long` integers, will have alignment 4.

Another processor might only be able to read 2 bytes at a time; in that environment, the alignment for a 4-byte `long` integer might be 2.

A structure type will have the same alignment as the structure member with the most strict alignment. To lower the alignment requirements on the structure and its members, use `#pragma pack`.

All data types must have a size that is a multiple of the alignment. Otherwise, only the first element of an array would be placed in accordance with the alignment requirements.

Note that with the `#pragma data_alignment` directive you can raise the alignment demands on specific variables.

In the following example, the alignment of the structure is 4, under the assumption that `long` has alignment 4. Its size is 8, even though only 5 bytes are effectively used.

```
struct str {
  long a;
  char b;
};
```

In standard C, the size of an object can be determined by using the `sizeof` operator.

ALIGNMENT ON THE MCS-51 MICROCONTROLLER

The MCS-51 microcontrollers have no alignment requirements, thus the alignment in the 8051 IAR C/C++ Compiler is 1.

Basic data types

The compiler supports both all ISO/ANSI C basic data types and some additional types.

INTEGER TYPES

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

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

Table 39: Integer types

Signed variables are represented using the two's complement form.

Bool

The `bool` data type is supported by default in the C++ language. If you have enabled language extensions, the `bool` type can also be used in C source code if you include the file `stdbool.h`. This will also enable the boolean values `false` and `true`.

The enum type

The compiler will use the smallest type required to hold `enum` constants, preferring `signed` rather than `unsigned`.

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

To make the compiler use a larger type than it would automatically use, define an `enum` constant with a large enough value. For example,

```
/* Disables usage of the char type for enum */
enum Cards{Spade1, Spade2,
           DontUseChar=257};
```

The char type

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

The wchar_t type

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The `wchar_t` data type is supported by default in the C++ language. To use the `wchar_t` type also in C source code, you must include the file `stddef.h` from the runtime library.

Note: The IAR CLIB Library has only rudimentary support for `wchar_t`.

Bitfields

In ISO/ANSI C, `int` and `unsigned int` can be used as the base type for integer bitfields. In the 8051 IAR C/C++ Compiler, any integer type can be used as the base type when language extensions are enabled.

Bitfields in expressions will have the same data type as the integer base type. Note that bitfields containing 1-bit fields will be very compact if declared in bdata memory. The fields will also be very efficient to access.

By default, the compiler places bitfield members from the least significant to the most significant bit in the container type.

By using the directive `#pragma bitfields=reversed`, the bitfield members are placed from the most significant to the least significant bit.

FLOATING-POINT TYPES

In the 8051 IAR C/C++ Compiler, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

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

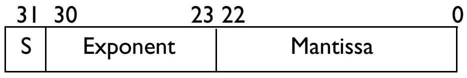
Table 40: Floating-point types

The compiler does not support subnormal numbers. All operations that should produce subnormal numbers will instead generate zero.

Exception flags according to the IEEE 754 standard are not supported.

32-bit floating-point format

The representation of a 32-bit floating-point number as an integer is:



The exponent is 8 bits, and the mantissa is 23 bits.

The value of the number is:

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

The range of the number is:

`±1.18E-38 to ±3.39E+38`

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

Representation of special floating-point numbers

The following list describes the representation of special floating-point numbers:

- 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.

Note: The IAR CLIB Library does not fully support the special cases of floating-point numbers, such as infinity and `NaN`. A library function which gets one of these special cases of floating-point numbers as an argument may behave unexpectedly.

Pointer types

The 8051 IAR C/C++ Compiler has two basic types of pointers: function pointers and data pointers.

FUNCTION POINTERS

Code pointers have two sizes: 16 or 24 bits. The following function pointers are available:

Pointer	Size	Address range	Description
`__near_func`	2 bytes	`0–0xFFFF`	Uses an `LCALL`/`LJMP` instruction to call the function.
`__banked_func`	2 bytes	`0–0xFFFFFF`	Calls a relay function which performs the bank switch and jumps to the banked function. Uses `?BRET` to return from the function. See *Bank switching in the Banked code model*, page 63.
`__banked_func_ext2`	3 bytes	`0–0xFFFFFF`	Uses the `MEX1` register and the memory extension stack. See *Bank switching in the Banked extended2 code model*, page 64.
`__far_func`	3 bytes	`0–0xFFFFFF`	Uses an extended `LCALL`/`LJMP` instruction supporting a 24-bit destination address to call the function. (These instructions are only available in some devices.)

Table 41: Function pointers

DATA POINTERS

Data pointers have three sizes: 8, 16, or 24 bits. The 8-bit pointer is used for data, bdata, idata or pdata memory, the 16-bit pointer is used for xdata or 16-bit code memory, and the 24-bit pointer is used for extended memories and for the generic pointer type.

The following data pointers are available:

Pointer	Address range	Pointer size	Index type	Description
`__idata`	`0–0xFF`	1 byte	signed char	Indirectly accessed data memory, accessed using `MOV A,@Ri`
`__pdata`	`0–0xFF`	1 byte	signed char	Parameter data, accessed using `MOVX A,@Ri`
`__xdata`	`0–0xFFFF`	2 bytes	signed short	Xdata memory, accessed using `MOVX A,@DPTR`
`__generic`	`0–0xFFFF`	3 bytes	signed short	The most significant byte identifies whether the pointer points to code or data memory
`__far`*	`0–0xFFFFFF`	3 bytes	signed short	Far xdata memory, accessed using `MOVX`
`__huge`	`0–0xFFFFFF`	3 bytes	signed long	Huge xdata memory
`__code`	`0–0xFFFF`	2 bytes	signed short	Code memory, accessed using `MOVC`
`__far_code`*	`0–0xFFFFFF`	3 bytes	signed short	Far code memory, accessed using `MOVC`
`__huge_code`	`0–0xFFFFFF`	3 bytes	signed long	Huge code memory, accessed using `MOVC`
`__far_rom`*	`0–0xFFFFFF`	3 bytes	signed short	Far code memory, accessed using `MOVC`
`__huge_rom`	`0–0xFFFFFF`	3 bytes	signed long	Huge code memory, accessed using `MOVC`

Table 42: Data pointers

*** The far pointer types have an index type that is smaller than the pointer size, which means pointer arithmetic will only be performed on the lower 16 bits. This restricts the placement of the object that the pointer points at. That is, the object can only be placed within 64-Kbyte pages.**

Generic pointers

A generic pointer can access objects located in both data and code memory. These pointers are 3 bytes in size. The most significant byte specifies which memory type the object is located in and the two remaining bytes specify the address in that memory. Generic pointers can therefore only address objects located in the first 64 Kbytes of code and data memory. This means that devices that support a larger addressing range will not be able to use generic pointers to access memory beyond the first 64 Kbytes. Copying constant objects to data memory is a better solution for these devices.

The most significant bit in the memory specifier indicates whether data or code memory is referred to (`0`=data, `1`=code). The least significant bit indicates whether internal or external data is referred to (`0`=xdata, `1`=idata).

CASTING

Casts between pointers have the following characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a *value* of an integer type to a pointer of a larger type is performed by zero extension
- Casting a *pointer type* to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed by zero extension
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result
- Casting from a smaller pointer to a larger pointer is performed by truncation
- Casting from a larger pointer to a smaller pointer is performed by zero extension.

size_t

`size_t` is the unsigned integer type required to hold the maximum size of an object. In the 8051 IAR C/C++ Compiler, the size of `size_t` is equal to the size of the unsigned type corresponding to the signed index type of the data pointer in use. The index type of data pointers is described in Table 42, *Data pointers*, page 222.

Note that for the small data model, this is formally a violation of the standard; the size of `size_t` should actually be 16 bits.

ptrdiff_t

`ptrdiff_t` is the type of the signed integer required to hold the difference between two pointers to elements of the same array. In the 8051 IAR C/C++ Compiler, the size of `ptrdiff_t` is equal to the size of the index type of the data pointer in use. The index type of data pointers is described in Table 42, *Data pointers*, page 222. When two pointers of different types are subtracted, the index type of the largest type determines the size.

Note: Subtracting the start address of an object from the end address can yield a negative value, because the object can be larger than what the `ptrdiff_t` can represent. See this example:

```
char buff[34000];            /* Assuming ptrdiff_t is a 16-bit */
char *p1 = buff;             /* signed integer type. */
char *p2 = buff + 34000;
ptrdiff_t diff = p2 - p1;
```

intptr_t

intptr_t is a signed integer type large enough to contain a void *. In the 8051 IAR C/C++ Compiler, the size of intptr_t is 32 bits and the type is signed long.

uintptr_t

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

Structure types

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

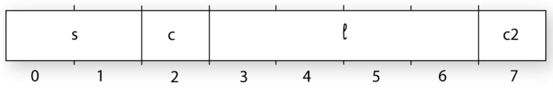
GENERAL LAYOUT

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

Example

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

The following diagram shows the layout in memory:



The alignment of the structure is 1 byte, and its size is 8 bytes.

Type qualifiers

According to the ISO/ANSI C standard, `volatile` and `const` are type qualifiers.

DECLARING OBJECTS VOLATILE

There are three main reasons for declaring an object `volatile`:

- Shared access; the object is shared between several tasks in a multitasking environment
- Trigger access; as for a memory-mapped SFR where the fact that an access occurs has an effect
- Modified access; where the contents of the object can change in ways not known to the compiler.

Definition of access to volatile objects

The ISO/ANSI standard defines an abstract machine, which governs the behavior of accesses to `volatile` declared objects. In general and in accordance to the abstract machine, the compiler:

- Considers each read and write access to an object that has been declared `volatile` as an access
- The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:

  ```
  char volatile a;
  a = 5;   /* A write access */
  a += 6;  /* First a read then a write access */
  ```
- An access to a bitfield is treated as an access to the underlaying type.

However, these rules are not detailed enough to handle the hardware-related requirements. The rules specific to the 8051 IAR C/C++ Compiler are described below.

Rules for accesses

In the 8051 IAR C/C++ Compiler, accesses to `volatile` declared objects are subject to the following rules:

- All accesses are preserved
- All accesses are complete, that is, the whole object is accessed
- All accesses are performed in the same order as given in the abstract machine
- All accesses are atomic, that is, they cannot be interrupted.

The 8051 IAR C/C++ Compiler adheres to these rules for 8-bit accesses of `volatile` declared objects in any data memory (RAM), and 1-bit accesses of `volatile` declared objects located in bit-addressable sfr memory or in bdata memory. For all other combinations, only the rule that states that all accesses are preserved applies.

DECLARING OBJECTS CONST

The `const` type qualifier is used for indicating that a data object, accessed directly or via a pointer, is non-writable. A pointer to `const` declared data can point to both constant and non-constant objects. It is good programming practice to use `const` declared pointers whenever possible because this improves the compiler's possibilities to optimize the generated code and reduces the risk of application failure due to erroneously modified data.

Static and global objects declared `const` and located in memories using the memory attributes `__code`, `__far_code`, and `__huge_code` are allocated in ROM. For all other memory attributes, the objects are allocated in RAM and initialized by the runtime system at startup.

In C++, objects that require runtime initialization cannot be placed in ROM.

Data types in C++

In C++, all plain C data types are represented in the same way as described earlier in this chapter. However, if any Embedded C++ features are used for a type, no assumptions can be made concerning the data representation. This means, for example, that it is not legal to write assembler code that accesses class members.

Compiler extensions

This chapter gives a brief overview of the 8051 IAR C/C++ Compiler extensions to the ISO/ANSI C standard. All extensions can also be used for the C++ programming language. More specifically the chapter describes the available C language extensions.

Compiler extensions overview

The compiler offers the standard features of ISO/ANSI C as well as a wide set of extensions, ranging from features specifically tailored for efficient programming in the embedded industry to the relaxation of some minor standards issues.

You can find the extensions available as:

- C/C++ language extensions

  For a summary of available language extensions, see *C language extensions*, page 228. For reference information about the extended keywords, see the chapter *Extended keywords*. For information about C++, the two levels of support for the language, and C++ language extensions; see the chapter *Using C++*.
- Pragma directives

  The `#pragma` directive is defined by the ISO/ANSI C standard and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

  The compiler provides a set of predefined pragma directives, which can be used for controlling the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages. Most pragma directives are preprocessed, which means that macros are substituted in a pragma directive. The pragma directives are always enabled in the compiler and many of them have an equivalent C/C++ language extensions. For a list of available pragma directives, see the chapter *Pragma directives*.
- Preprocessor extensions

  The preprocessor of the compiler adheres to the ISO/ANSI standard. In addition, the compiler also makes a number of preprocessor-related extensions available to you. For more information, see the chapter *The preprocessor*.
- Intrinsic functions

  The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions. To read more about using intrinsic functions, see *Mixing C and assembler*, page 125. For a list of available functions, see the chapter *Intrinsic functions*.

- Library functions

  The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. In addition, the library also provides some extensions, partly taken from the C99 standard. For more information, see *IAR DLIB Library*, page 287.

Note: Any use of these extensions, except for the pragma directives, makes your application inconsistent with the ISO/ANSI C standard.

ENABLING LANGUAGE EXTENSIONS

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

For information about how to enable and disable language extensions from the command line, see the compiler options *-e*, page 200, and *--strict_ansi*, page 216.

C language extensions

This section gives a brief overview of the C language extensions available in the 8051 IAR C/C++ Compiler. The compiler provides a wide set of extensions, so to help you to find the extensions required by your application, the extensions have been grouped according to their expected usefulness. In short, this means:

- Important language extensions—extensions specifically tailored for efficient embedded programming, typically to meet memory restrictions
- Useful language extensions—features considered useful and typically taken from related standards, such as C99 and C++
- Minor language extensions, that is, the relaxation of some minor standards issues and also some useful but minor syntax extensions.

IMPORTANT LANGUAGE EXTENSIONS

The following language extensions available both in the C and the C++ programming languages are well suited for embedded systems programming:

- Memory attributes, type attributes, and object attributes

  For information about the related concepts, the general syntax rules, and for reference information, see the chapter *Extended keywords*.

- Placement at an absolute address or in a named segment

  The `@` operator or the directive `#pragma location` can be used for placing global and static variables at absolute addresses, or placing a variable or function in a named segment. For more information about using these primitives, see *Controlling data and function placement in memory*, page 165, and *location*, page 266.

- Alignment

  Each data type has its own alignment, for more details, see *Alignment*, page 217. If you want to change the alignment, the `#pragma data_alignment` directive is available. If you want to check the alignment of an object, use the `__ALIGNOF__()` operator.

  The `__ALIGNOF__` operator is used for accessing the alignment of an object. It takes one of two forms:

  - `__ALIGNOF__ (`*`type`*`)`
  - `__ALIGNOF__ (`*`expression`*`)`

  In the second form, the expression is not evaluated.

- Anonymous structs and unions

  C++ includes a feature named anonymous unions. The compiler allows a similar feature for both structs and unions in the C programming language. For more information, see *Anonymous structs and unions*, page 164.

- Bitfields and non-standard types

  In ISO/ANSI C, a bitfield must be of type `int` or `unsigned int`. Using IAR Systems language extensions, any integer type or enumeration may be used. The advantage is that the struct will sometimes be smaller. This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*. For more information, see *Bitfields*, page 219.

- Dedicated segment operators `__segment_begin` and `__segment_end`

  The syntax for these operators is:

  ```
  void * __segment_begin(segment)
  void * __segment_end(segment)
  ```

  These operators return the address of the first byte of the named *`segment`* and the first byte *after* the named *`segment`*, respectively. This can be useful if you have used the `@` operator or the `#pragma location` directive to place a data object or a function in a user-defined segment.

  The named *`segment`* must be a string literal and *`segment`* must have been declared earlier with the `#pragma segment` directive. If the segment was declared with a memory attribute *`memattr`*, the type of the `__segment_begin` operator is a pointer to *`memattr`* `void`. Otherwise, the type is a default pointer to `void`. Note that you must have enabled language extensions to use these operators.

In the following example, the type of the `__segment_begin` operator is `void __huge *`.

```
#pragma segment="MYSEGMENT" __huge
...
segment_start_address = __segment_begin("MYSECTION");
```

See also *segment*, page 271, and *location*, page 266.

USEFUL LANGUAGE EXTENSIONS

This section lists and briefly describes useful extensions, that is, useful features typically taken from related standards, such as C99 and C++:

- Inline functions

  The `#pragma inline` directive, alternatively the `inline` keyword, advises the compiler that the function whose declaration follows immediately after the directive should be inlined. This is similar to the C++ keyword `inline`. For more information, see *inline*, page 265.
- Mixing declarations and statements

  It is possible to mix declarations and statements within the same scope. This feature is part of the C99 standard and C++.
- Declaration in `for` loops

  It is possible to have a declaration in the initialization expression of a `for` loop, for example:

  ```
  for (int i = 0; i < 10; ++i)
  {...}
  ```

  This feature is part of the C99 standard and C++.
- The `bool` data type

  To use the `bool` type in C source code, you must include the file `stdbool.h`. This feature is part of the C99 standard and C++. (The `bool` data type is supported by default in C++.)
- C++ style comments

  C++ style comments are accepted. A C++ style comment starts with the character sequence `//` and continues to the end of the line. For example:

  ```
  // The length of the bar, in centimeters.
  int length;
  ```

  This feature is copied from the C99 standard and C++.

Inline assembler

Inline assembler can be used for inserting assembler instructions in the generated function. This feature is part of the C99 standard and C++.

The asm and __asm extended keywords both insert an assembler instruction. However, when compiling C source code, the asm keyword is not available when the option --strict_ansi is used. The __asm keyword is always available.

Note: Not all assembler directives or operators can be inserted using this keyword.

The syntax is:

```
asm ("string");
```

The string can be a valid assembler instruction or a data definition assembler directive, but not a comment. You can write several consecutive inline assembler instructions, for example:

```
asm ("Label:      nop\n"
     "            lcall Label");
```

where \n (new line) separates each new assembler instruction. Note that you can define and use local labels in inline assembler instructions.

For more information about inline assembler, see *Mixing C and assembler*, page 125.

Compound literals

To create compound literals you can use the following syntax:

```
/* Create a pointer to an anonymous array */
int *p = (int []) {1,2,3};

/* Create a pointer to an anonymous structX */
structX *px = &(structX) {5,6,7};
```

Note:

- A compound literal can be modified unless it is declared const
- Compound literals are not supported in Embedded C++ and Extended EC++.
- This feature is part of the C99 standard.

Incomplete arrays at end of structs

The last element of a struct may be an incomplete array. This is useful because one chunk of memory can be allocated for the struct itself and for the array, regardless of the size of the array.

Note: The array cannot be the only member of the struct. If that was the case, then the size of the struct would be zero, which is not allowed in ISO/ANSI C.

Example

```
struct str
{
  char a;
  unsigned long b[];
};

struct str * GetAStr(int size)
{
  return malloc(sizeof(struct str) +
                sizeof(unsigned long) * size);
}

void UseStr(struct str * s)
{
  s->b[10] = 0;
}
```

The `struct` will inherit the alignment requirements from all elements, including the alignment of the incomplete array. The array itself will not be included in the size of the struct. However, the alignment requirements will ensure that the struct will end exactly at the beginning of the array; this is known as padding.

Hexadecimal floating-point constants

Floating-point constants can be given in hexadecimal style. The syntax is `0x`*`MANT`*`p{+|-}`*`EXP`*, where *`MANT`* is the mantissa in hexadecimal digits, including an optional `.` (decimal point), and *`EXP`* is the exponent with decimal digits, representing an exponent of 2. This feature is part of the C99 standard.

Examples

`0x1p0` is `1`

`0xA.8p2` is `10.5*2^2`

Designated initializers in structures and arrays

Any initialization of either a structure (`struct` or `union`) or an array can have a designation. A designation consists of one or more designators followed by an initializer. A designator for a structure is specified as `.`*`elementname`* and for an array `[`*`constant index expression`*`]`. Using designated initializers is not supported in C++.

Examples

The following definition shows a `struct` and its initialization using designators:

```
struct{
  int i;
  int j;
  int k;
  int l;
  short array[10];
} u = {
  .l = 6,           /* initialize l to 6 */
  .j = 6,           /* initialize j to 6 */
  8,                /* initialize k to 8 */
  .array[7] = 2,    /* initialize element 7 to 2 */
  .array[3] = 2,    /* initialize element 3 to 2 */
  5,                /* array[4] = 5 */
  .k = 4            /* reinitialize k to 4 */
};
```

Note that a designator specifies the destination element of the initialization. Note also that if one element is initialized more than once, it is the last initialization that will be used.

To initialize an element in a union other than the first, do like this:

```
union{
  int i;
  float f;
}y = {.f = 5.0};
```

To set the size of an array by initializing the last element, do like this:

```
char array[] = {[10] = 'a'};
```

MINOR LANGUAGE EXTENSIONS

This section lists and briefly describes minor extensions, that is, the relaxation of some standards issues and also some useful but minor syntax extensions:

- Arrays of incomplete types

  An array may have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).
- Forward declaration of `enum` types

  The IAR Systems language extensions allow that you first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.

- Missing semicolon at end of `struct` or `union` specifier

  A warning is issued if the semicolon at the end of a `struct` or `union` specifier is missing.
- Null and `void`

  In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ISO/ANSI C, some operators allow such things, while others do not allow them.
- Casting pointers to integers in static initializers

  In an initializer, a pointer constant value may be cast to an integral type if the integral type is large enough to contain it. For more information about casting pointers, see *Casting*, page 223.
- Taking the address of a register variable

  In ISO/ANSI C, it is illegal to take the address of a variable specified as a register variable. The compiler allows this, but a warning is issued.
- Duplicated size and sign specifiers

  Should the size or sign specifiers be duplicated (for example, `short short` or `unsigned unsigned`), an error is issued.
- `long float` means `double`

  The type `long float` is accepted as a synonym for `double`.
- Repeated `typedef` declarations

  Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.
- Mixing pointer types

  Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.

  Assignment of a string constant to a pointer to any kind of character is allowed, and no warning will be issued.
- Non-top level `const`

  Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). Comparing and taking the difference of such pointers is also allowed.
- Non-`lvalue` arrays

  A non-`lvalue` array expression is converted to a pointer to the first element of the array when it is used.

- Comments at the end of preprocessor directives

  This extension, which makes it legal to place text after preprocessor directives, is enabled, unless strict ISO/ANSI mode is used. The purpose of this language extension is to support compilation of legacy code; we do *not* recommend that you write new code in this fashion.

- An extra comma at the end of `enum` lists

  Placing an extra comma is allowed at the end of an `enum` list. In strict ISO/ANSI mode, a warning is issued.

- A label preceding a `}`

  In ISO/ANSI C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. In the 8051 IAR C/C++ Compiler, a warning is issued.

Note: This also applies to the labels of `switch` statements.

- Empty declarations

  An empty declaration (a semicolon by itself) is allowed, but a remark is issued (provided that remarks are enabled).

- Single-value initialization

  ISO/ANSI C requires that all initializer expressions of static arrays, `structs`, and `unions` are enclosed in braces.

  Single-value initializers are allowed to appear without braces, but a warning is issued. In the 8051 IAR C/C++ Compiler, the following expression is allowed:

  ```
  struct str
  {
    int a;
  } x = 10;
  ```

- Declarations in other scopes

  External and static declarations in other scopes are visible. In the following example, the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

  ```
  int test(int x)
  {
    if (x)
    {
      extern int y;
      y = 1;
    }

    return y;
  }
  ```

- Expanding function names into strings with the function as context

  Use any of the symbols `__func__` or `__FUNCTION__` inside a function body to make it expand into a string, with the function name as context. Use the symbol `__PRETTY_FUNCTION__` to also include the parameter types and return type. The result might, for example, look like this if you use the `__PRETTY_FUNCTION__` symbol:

  ```
  "void func(char)"
  ```

  These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see *-e*, page 200.

Extended keywords

This chapter describes the extended keywords that support specific features of the MCS-51 microcontroller and the general syntax rules for the keywords. Finally the chapter gives a detailed description of each keyword.

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

General syntax rules for extended keywords

To understand the syntax rules for the extended keywords, it is important to be familiar with some related concepts.

The 8051 IAR C/C++ Compiler provides a set of attributes that can be used on functions or data objects to support specific features of the MCS-51 microcontroller. There are two types of attributes—*type attributes* and *object attributes*:

- Type attributes affect the *external functionality* of the data object or function
- Object attributes affect the *internal functionality* of the data object or function.

The syntax for the keywords differs slightly depending on whether it is a type attribute or an object attribute, and whether it is applied to a data object or a function.

For information about how to use attributes to modify data, see the chapter *Data storage*. For information about how to use attributes to modify functions, see the chapter *Functions*. For detailed information about each attribute, see *Descriptions of extended keywords*, page 242.

Note: The extended keywords are only available when language extensions are enabled in the 8051 IAR C/C++ Compiler.

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

Use the `-e` compiler option to enable language extensions. See *-e*, page 200 for additional information.

TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that if you use a type attribute, it must be specified both when a function or data object is defined and when it is declared.

You can either place the type attributes directly in your source code, or use the pragma directive `#pragma type_attribute`.

Type attributes can be further divided into *memory attributes* and *general type attributes*.

Memory attributes

A memory attribute corresponds to a certain logical or physical memory in the microcontroller.

- Available *function memory attributes*: `__banked_func`, `__far_func`, and `__near_func`
- Available *data memory attributes*: `__bdata`, `__bit`, `__code`, `__data`, `__far`, `__far_code`, `__far_rom`, `__generic`, `__huge`, `__huge_code`, `__huge_rom`, `__idata`, `__ixdata`, `__pdata`, `__sfr`, `__xdata`, and `__xdata_rom`.

Data objects, functions, and destinations of pointers or C++ references always have a memory attribute. If no attribute is explicitly specified in the declaration or by the pragma directive `#pragma type_attribute`, an appropriate default attribute is used. You can only specify one memory attribute for each level of pointer indirection.

General type attributes

The following general type attributes are available:

- *Function type attributes* affect how the function should be called: `__data_overlay`, `__ext_stack_reentrant`, `__idata_overlay`, `__idata_reentrant`, `__interrupt`, `__monitor`, `__pdata_reentrant`, `__task`, and `__xdata_reentrant`
- *Data type attributes*: `const` and `volatile`

You can specify as many type attributes as required for each level of pointer indirection.

To read more about the type qualifiers `const` and `volatile`, see *Type qualifiers*, page 225.

Syntax for type attributes used on data objects

In general, type attributes for data objects follow the same syntax as the type qualifiers `const` and `volatile`.

The following declaration assigns the `__data` type attribute to the variables `i` and `j`; in other words, the variable `i` and `j` is placed in data memory. The variables `k` and `l` behave in the same way:

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

Note that the attribute affects both identifiers.

The following declaration of `i` and `j` is equivalent with the previous one:

```
#pragma type_attribute=__data
int i, j;
```

The advantage of using pragma directives for specifying keywords is that it offers you a method to make sure that the source code is portable. Note that the pragma directive has no effect if a memory attribute is already explicitly declared.

For more examples of using memory attributes, see *More examples*, page 32.

An easier way of specifying storage is to use type definitions. The following two declarations are equivalent:

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

and

```
__data char b;
char __data *bp;
```

Note that `#pragma type_attribute` can be used together with a `typedef` declaration.

Syntax for type attributes on data pointers

The syntax for declaring pointers using type attributes follows the same syntax as the type qualifiers `const` and `volatile`:

`int __data * p;`	The `int` object is located in `__data` memory.
`int * __data p;`	The pointer is located in `__data` memory.
`__data int * p;`	The pointer is located in `__data` memory.

Syntax for type attributes on functions

The syntax for using type attributes on functions, differs slightly from the syntax of type attributes on data objects. For functions, the attribute must be placed either in front of the return type, alternatively in parentheses, for example:

```
__interrupt void my_handler(void);
```

or

```
void (__interrupt my_handler)(void);
```

The following declaration of `my_handler` is equivalent with the previous one:

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

Syntax for type attributes on function pointers

To declare a function pointer, use the following syntax:

```
int (__far_func * fp) (double);
```

After this declaration, the function pointer `fp` points to farfunc memory.

An easier way of specifying storage is to use type definitions:

```
typedef __far_func void FUNC_TYPE(int);
typedef FUNC_TYPE *FUNC_PTR_TYPE;
FUNC_TYPE func();
FUNC_PTR_TYPE funcptr;
```

Note that `#pragma type_attribute` can be used together with a `typedef` declaration.

OBJECT ATTRIBUTES

Object attributes affect the internal functionality of functions and data objects, but not how the function is called or how the data is accessed. This means that an object attribute does not need to be present in the declaration of an object.

The following object attributes are available:

- Object attributes that can be used for variables: `__no_init`
- Object attributes that can be used for functions and variables: `location`, `@`, and `__root`
- Object attributes that can be used for functions: `__intrinsic`, `__noreturn`, and `vector.`

You can specify as many object attributes as required for a specific function or data object.

For more information about `location` and `@`, see *Controlling data and function placement in memory*, page 165. For more information about `vector`, see *vector*, page 272.

Syntax for object attributes

The object attribute must be placed in front of the type. For example, to place `myarray` in memory that is not initialized at startup:

```
__no_init int myarray[10];
```

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

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

Note: Object attributes cannot be used in combination with the `typedef` keyword.

Summary of extended keywords

The following table summarizes the extended keywords:

Extended keyword	Description
`__banked_func`	Controls the storage of functions
`__banked_func_ext2`	Controls the storage of functions
`__bdata`	Controls the storage of data objects
`__bit`	Controls the storage of data objects
`__code`	Controls the storage of data objects
`__data`	Controls the storage of data objects
`__data_overlay`	Controls the storage of auto data objects
`__ext_stack_reentrant`	Controls the storage of auto data objects
`__far`	Controls the storage of data objects
`__far_code`	Controls the storage of constant data objects
`__far_func`	Controls the storage of functions
`__far_rom`	Controls the storage of constant data objects
`__generic`	Pointer type attribute
`__huge`	Controls the storage of data objects
`__huge_code`	Controls the storage of constant data objects
`__huge_rom`	Controls the storage of constant data objects
`__idata`	Controls the storage of data objects
`__idata_overlay`	Controls the storage of auto data objects
`__idata_reentrant`	Controls the storage of auto data objects
`__ixdata`	Controls the storage of data objects
`__interrupt`	Supports interrupt functions
`__intrinsic`	Reserved for compiler internal use only
`__monitor`	Supports atomic execution of a function

Table 43: Extended keywords summary

Extended keyword	Description
`__near_func`	Controls the storage of functions
`__no_init`	Supports non-volatile memory
`__noreturn`	Informs the compiler that the declared function will not return
`__overlay_near_func`	Reserved for compiler internal use only
`__pdata`	Controls the storage of data objects
`__pdata_reentrant`	Controls the storage of auto data objects
`__root`	Ensures that a function or variable is included in the object code even if unused
`__sfr`	Controls the storage of data objects
`__task`	Allows functions to exit without restoring registers
`__xdata`	Controls the storage of data objects
`__xdata_rom`	Controls the storage of constant data objects
`__xdata_reentrant`	Controls the storage of auto data objects

Table 43: Extended keywords summary (Continued)

Descriptions of extended keywords

The following sections give detailed information about each extended keyword.

__banked_func

Syntax

Follows the generic syntax rules for memory type attributes that can be used on functions, see *Type attributes*, page 237.

Description

The `__banked_func` memory attribute overrides the default storage of functions given by the selected code model and places individual functions in memory where they are called using banked 24-bit calls. You can also use the `__banked_func` attribute to create a pointer explicitly pointing to an object located in the banked memory.

Storage information

- Memory space: Code memory space
- Address range: `0-0xFFFFFF`
- Maximum size: 64 Kbytes
- Pointer size: 2 bytes
- Function size: bank size

Note: This keyword is only available when the Banked code model is used, and in this case functions are by default `__banked_func`. There are some exceptions, see *Code that cannot be banked*, page 61. Overlay and extended stack functions cannot be banked. This means that you cannot combine the `__banked_func` keyword with the `__data_overlay` or `__idata_overlay`, and `__ext_stack_reentrant` keywords.

Example

```
__banked_func void myfunction(void);
```

See also

Banked functions, page 55.

__banked_func_ext2

Syntax

Follows the generic syntax rules for memory type attributes that can be used on functions, see *Type attributes*, page 237.

Description

The `__banked_func_ext2` memory attribute overrides the default storage of functions given by the selected code model and places individual functions in memory where they are called using banked 24-bit calls. You can also use the `__banked_func_ext2` attribute to create a pointer explicitly pointing to an object located in the banked memory.

Storage information

- Memory space: Code memory space
- Address range: `0-0xFFFFFF`
- Maximum size: 64 Kbytes
- Pointer size: 3 bytes
- Function size: bank size

Note: This keyword is only available when the Banked extended2 code model is used, and in this case all functions are by default `__banked_func_ext2`. Such functions require the Xdata reentrant calling convention.

Example

```
__banked_func_ext2 void myfunction(void);
```

See also

Banked functions, page 55.

__bdata

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description	The `__bdata` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in bdata memory. **Note:** There are no `__bdata` pointers. Bdata memory is referred to by `__idata` pointers.
Storage information	● Memory space: Internal data memory space ● Address range: `0x20-0x2F` ● Maximum object size: 16 bytes. ● Pointer size: 1 byte, `__idata` pointer
Example	`__bdata int x;`
See also	*Memory types*, page 24.

__bit

Syntax	Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.
Description	The `__bit` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in bit addressable memory. You cannot create a pointer to bit memory.
Storage information	● Memory space: Internal data memory space ● Address range: `0x20-0x2F` ● Maximum object size: 1 bit ● Pointer size: N/A
Example	`__bit int x;`
See also	*Memory types*, page 24.

__code

Syntax	Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description

The `__code` memory attribute overrides the default storage of variables given by the selected data model and places individual constants and strings in code memory. You can also use the `__code` attribute to create a pointer explicitly pointing to an object located in the code memory.

Storage information

- Memory space: Code memory space
- Address range: `0x0-0xFFFF`
- Maximum object size: 64 Kbytes
- Pointer size: 2 bytes

Example

```
__code int x;
```

See also

Memory types, page 24.

__data

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description

The `__data` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in data memory.

Note: There are no `__data` pointers. Data memory is referred to by `__idata` pointers.

Storage information

- Memory space: Internal data memory space
- Address range: `0x0-0x7F`
- Maximum object size: 64 Kbytes
- Pointer size: 1 byte, `__idata` pointer

Example

```
__data int x;
```

See also

Memory types, page 24.

__data_overlay

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 237.

Description

The `__data_overlay` keyword places parameters and auto variables in the data overlay memory area.

Note: This keyword is only available when the Tiny or Small data model is used.

Example

```
__data_overlay void foo(void);
```

See also

Auto variables—stack and static overlay, page 35.

__ext_stack_reentrant

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 237.

Description

The `__ext_stack_reentrant` keyword places parameters and auto variables on the extended stack.

Note: This keyword can only be used when the `--extended_stack` option has been specified.

Example

```
__ext_stack_reentrant void foo(void);
```

See also

Auto variables—stack and static overlay, page 35, *Extended stack*, page 77, and *--extended_stack*, page 202.

__far

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description

The `__far` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in far memory. You can also use the `__far` attribute to create a pointer explicitly pointing to an object located in the far memory.

Note: This memory attribute is only available when the Far data model is used.

Storage information

- Memory space: External data memory space
- Address range: `0-0xFFFFFF`
- Maximum object size: 64 Kbytes. An object cannot cross a 64-Kbyte boundary.
- Pointer size: 3 bytes. Arithmetics is only performed on the two lower bytes, except comparison which is always performed on the entire 24-bit address.

Example

```
__far int x;
```

See also — *Memory types*, page 24.

__far_code

Syntax — Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description — The `__far_code` memory attribute overrides the default storage of variables given by the selected data model and places individual constants and strings in far code memory. You can also use the `__far_code` attribute to create a pointer explicitly pointing to an object located in the far code memory.

Note: This memory attribute is only available when the Far code model is used.

Storage information

- Memory space: Code memory space
- Address range: `0-0xFFFFFF`
- Maximum object size: 64 Kbytes. An object cannot cross a 64-Kbyte boundary.
- Pointer size: 3 bytes. Arithmetics is only performed on the two lower bytes, except comparison which is always performed on the entire 24-bit address.

Example

```
__far_code int x;
```

See also — *Memory types*, page 24.

__far_func

Syntax — Follows the generic syntax rules for memory type attributes that can be used on functions, see *Type attributes*, page 237.

Description — The `__far_func` memory attribute overrides the default storage of functions given by the selected code model and places individual functions in far memory—memory where the function is called using true 24-bit calls. You can also use the `__far_func` attribute to create a pointer explicitly pointing to an object located in the far memory.

Note: This memory attribute is only available when the Far code model is used.

Storage information

- Memory space: Code memory space
- Address range: `0-0xFFFFFF`
- Maximum size: 65535 bytes. A function cannot cross a 64-Kbyte boundary.
- Pointer size: 3 bytes

Example

```
__far_func void myfunction(void);
```

See also

Code models for function storage, page 47.

__far_rom

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description

The `__far_rom` memory attribute overrides the default storage of variables given by the selected data model and places individual constants and strings in ROM located in the far memory range. You can also use the `__far_rom` attribute to create a pointer explicitly pointing to an object located in the far_rom memory.

Note: This memory attribute is only available when the Far data model is used.

Storage information

- Memory space: External data memory space
- Address range: `0-0xFFFFFF`
- Maximum object size: 64 Kbytes. An object cannot cross a 64-Kbyte boundary.
- Pointer size: 2 bytes. Arithmetics is only performed on the two lower bytes, except comparison which is always performed on the entire 24-bit address.

Example

```
__far_rom int x;
```

See also

Memory types, page 24.

__generic

Syntax

Follows the generic syntax rules for type attributes that can be used on data, see *Type attributes*, page 237.

Description

The `__generic` pointer attribute specifies a generic pointer that can access data anywhere in the internal data memory space (idata), external data memory space (xdata), or the code memory space.

If a variable is declared with this keyword, it will be located in the external data memory space (xdata), in the same way as if the `__xdata` attribute had been used.

Note: This memory attribute is not available when the Far data model is used.

Example

```
__int __generic * ptr;
```

See also

Generic pointers, page 222.

__huge

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description

The `__huge` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in huge memory. You can also use the `__huge` attribute to create a pointer explicitly pointing to an object located in the huge memory.

Note: This memory attribute is only available when the Far data model is used.

Storage information

- Memory space: External data memory space
- Address range: `0-0xFFFFFF`
- Maximum object size: 16 Mbytes
- Pointer size: 3 bytes

Example

```
__huge int x;
```

See also

Memory types, page 24.

__huge_code

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description

The `__huge_code` memory attribute overrides the default storage of variables given by the selected data model and places individual constants and strings in huge code memory. You can also use the `__huge_code` attribute to create a pointer explicitly pointing to an object located in the huge code memory.

Note: This memory attribute is only available when the Far code model is used.

Storage information

- Memory space: Code memory space
- Address range: `0-0xFFFFFF`
- Maximum object size: 16 Mbytes
- Pointer size: 3 bytes

Example

```
__huge_code int x;
```

See also

Memory types, page 24.

__huge_rom

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description

The `__huge_rom` memory attribute overrides the default storage of variables given by the selected data model and places individual constants and strings in ROM located in the far memory range. You can also use the `__huge_rom` attribute to create a pointer explicitly pointing to an object located in the huge_rom memory.

Note: This memory attribute is only available when the Far data model is used.

Storage information

- Memory space: External data memory space
- Address range: `0-0xFFFFFF`
- Maximum object size: 16 Mbytes
- Pointer size: 2 bytes. Arithmetics is only performed on the two lower bytes, except comparison which is always performed on the entire 24-bit address.

Example

```
__huge_rom int x;
```

See also

Memory types, page 24.

__idata

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description

The `__idata` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in idata memory. You can also use the `__idata` attribute to create a pointer explicitly pointing to an object located in the idata memory.

Storage information

- Memory space: Internal data memory space
- Address range: `0x0-0xFF`
- Maximum object size: 256 bytes
- Pointer size: 1 byte

Example

```
__idata int x;
```

See also — *Memory types*, page 24.

__idata_overlay

Syntax — Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 237.

Description — The `__idata_overlay` keyword places parameters and auto variables in the idata overlay memory area.

Note: This keyword is only available when the Tiny or Small data model is used.

Example

```
__idata_overlay void foo(void);
```

See also — *Auto variables—stack and static overlay*, page 35.

__idata_reentrant

Syntax — Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 237.

Description — The `__idata_reentrant` keyword places parameters and auto variables on the idata stack.

Note: This keyword can only be used when the `--extended_stack` option has been specified.

Example

```
__idata_reentrant void foo(void);
```

See also — *Auto variables—stack and static overlay*, page 35, *Extended stack*, page 77, and --*extended_stack*, page 202.

__ixdata

Syntax — Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description — The `__ixdata` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in data memory.

The `__ixdata` memory attribute requires a devices that supports on-chip external data (xdata).

Note: There are no `__ixdata` pointers. Data memory is referred to by `__idata` pointers.

Storage information

- Memory space: External data memory space
- Address range: `0x0-0xFFFF`
- Maximum object size: 64 Kbytes
- Pointer size: 1 byte, `__idata` pointer

Example

```
__ixdata int x;
```

See also

Memory types, page 24.

__interrupt

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 237.

Description

The `__interrupt` keyword specifies interrupt functions. To specify one or several interrupt vectors, use the `#pragma vector` directive. The range of the interrupt vectors depends on the device used. It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table.

An interrupt function must have a `void` return type and cannot have any parameters.

The header file `io`*`device`*`.h`, where *`device`* corresponds to the selected device, contains predefined names for the existing interrupt vectors.

Example

```
#pragma vector=0x13
__interrupt void my_interrupt_handler(void);
```

See also

Interrupt functions, page 49, *vector*, page 272, *INTVEC*, page 316.

__intrinsic

Description

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

__monitor

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 237.

Description

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

Example

```
__monitor int get_lock(void);
```

See also

Monitor functions, page 50. Read also about the intrinsic functions *__disable_interrupt*, page 273, *__enable_interrupt*, page 274, *__get_interrupt_state*, page 274, and *__set_interrupt_state*, page 275.

__near_func

Syntax

Follows the generic syntax rules for memory type attributes that can be used on functions, see *Type attributes*, page 237.

Description

The `__near_func` memory attribute overrides the default storage of functions given by the selected code model and places individual functions in near memory.

In the Banked code model, use the `__near_func` attribute to explicitly place a function in the root area.

Storage information

- Memory space: Code memory space
- Address range: `0-0xFFFF`
- Maximum size: 64 Kbytes
- Pointer size: 2 bytes

Example

```
__near_func void myfunction(void);
```

See also

Code models for function storage, page 47.

__no_init

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 240.

Description

Use the `__no_init` keyword to place a data object in non-volatile memory. This means that the initialization of the variable, for example at system startup, is suppressed.

Example

```
__no_init int myarray[10];
```

__noreturn

Syntax — Follows the generic syntax rules for object attributes, see *Object attributes*, page 240.

Description — The `__noreturn` keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are `abort` and `exit`.

Example

```
__noreturn void terminate(void);
```

__overlay_near_func

Description — The `__overlay_near_func` keyword is reserved for compiler internal use only.

__pdata

Syntax — Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description — The `__pdata` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in pdata memory. You can also use the `__pdata` attribute to create a pointer explicitly pointing to an object located in the pdata memory.

Storage information

- Memory space: External data memory space
- Address range: `0x0-0xFF`
- Maximum object size: 256 bytes
- Pointer size: 1 byte

Example

```
__pdata int x;
```

See also — *Memory types*, page 24.

__pdata_reentrant

Syntax — Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 237.

Description — The `__pdata_reentrant` keyword places parameters and auto variables on the pdata stack.

Example	`__pdata_reentrant void foo(void);`
See also	*Auto variables—stack and static overlay*, page 35, *Extended stack*, page 77, and --*extended_stack*, page 202.

__root

Syntax	Follows the generic syntax rules for object attributes, see *Object attributes*, page 240.
Description	A function or variable with the `__root` attribute is kept whether or not it is referenced from the rest of the application, provided its module is included. Program modules are always included and library modules are only included if needed.
Example	`__root int myarray[10];`
See also	To read more about modules, segments, and the link process, see the *IAR Linker and Library Tools Reference Guide.*

__sfr

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description

The `__sfr` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in SFR memory. You cannot create a pointer to an object located in SFR memory.

Storage information

- Memory space: Internal data memory space
- Address range: `0x80-0xFF`, direct addressing
- Maximum object size: 128 bytes
- Pointer size: N/A

Example

```
__sfr int x;
```

See also

Memory types, page 24.

__task

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 237.

Description

This keyword allows functions to exit without restoring registers and it is typically used for the `main` function.

By default, functions save the contents of used non-scratch registers (permanent registers) on the stack upon entry, and restore them at exit. Functions declared `__task` do not save any registers, and therefore require less stack space. Such functions should only be called from assembler routines.

The function `main` may be declared `__task`, unless it is explicitly called from the application. In real-time applications with more than one task, the root function of each task may be declared `__task`.

Example

```
__task void my_handler(void);
```

__xdata

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description

The `__xdata` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in xdata memory. You can also use the `__xdata` attribute to create a pointer explicitly pointing to an object located in the xdata memory.

Storage information

- Memory space: External data memory space
- Address range: `0-0xFFFF` (64 Kbytes)
- Maximum object size: 64 Kbytes
- Pointer size: 2 bytes

Example

```
__xdata int x;
```

See also

Memory types, page 24.

__xdata_reentrant

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 237.

Description

The `__xdata_reentrant` keyword places parameters and auto variables on the xdata stack.

Example

```
__xdata_reentrant void foo(void);
```

See also

Auto variables—stack and static overlay, page 35, *Extended stack*, page 77, and --*extended_stack*, page 202.

__xdata_rom

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description

The `__xdata_rom` memory attribute overrides the default storage of variables given by the selected data model and places individual constants and strings in ROM located in the xdata memory range. You can also use the `__xdata_rom` attribute to create a pointer explicitly pointing to an object located in the xdata rom memory.

Storage information

- Memory space: External data memory space
- Address range: `0-0xFFFF` (64 Kbytes)
- Maximum object size: 64 Kbytes
- Pointer size: 2 bytes

Example

```
__xdata_rom int x;
```

See also

Memory types, page 24.

Pragma directives

This chapter describes the pragma directives of the 8051 IAR C/C++ Compiler.

The #pragma directive is defined by the ISO/ANSI C standard and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

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

The pragma directives are always enabled in the compiler.

Summary of pragma directives

The following table lists the pragma directives of the compiler that can be used either with the `#pragma` preprocessor directive or the `_Pragma()` preprocessor operator:

Pragma directive	Description
`basic_template_matching`	Makes a template function fully memory-attribute aware
`bitfields`	Controls the order of bitfield members
`constseg`	Places constant variables in a named segment
`dataseg`	Places variables in a named segment
`diag_default`	Changes the severity level of diagnostic messages
`diag_error`	Changes the severity level of diagnostic messages
`diag_remark`	Changes the severity level of diagnostic messages
`diag_suppress`	Suppresses diagnostic messages
`diag_warning`	Changes the severity level of diagnostic messages
`include_alias`	Specifies an alias for an include file
`inline`	Inlines a function
`language`	Controls the IAR Systems language extensions
`location`	Specifies the absolute address of a variable, or places groups of functions or variables in named segments

Table 44: Pragma directives summary

Pragma directive	Description
`message`	Prints a message
`object_attribute`	Changes the definition of a variable or a function
`optimize`	Specifies the type and level of an optimization
`__printf_args`	Verifies that a function with a printf-style format string is called with the correct arguments
`register_bank`	Specifies the register bank to be used by an interrupt function
`required`	Ensures that a symbol that is needed by another symbol is included in the linked output
`rtmodel`	Adds a runtime model attribute to the module
`__scanf_args`	Verifies that a function with a scanf-style format string is called with the correct arguments
`segment`	Declares a segment name to be used by intrinsic functions
`type_attribute`	Changes the declaration and definitions of a variable or function
`vector`	Specifies the vector of an interrupt or trap function

Table 44: Pragma directives summary (Continued)

Note: For portability reasons, see also *Recognized pragma directives (6.8.6)*, page 333 and the *8051 IAR Embedded Workbench® Migration Guide*.

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

basic_template_matching

Syntax

```
#pragma basic_template_matching
```

Description

Use this pragma directive in front of a template function declaration to make the function fully memory-attribute aware, in the rare cases where this is useful. That template function will then match the template without the modifications described in *Templates and data memory attributes*, page 156.

Example

```
#pragma basic_template_matching
template<typename T> void fun(T *);

fun((int __huge *) 0); /* Template param T becomes int __huge */
```

bitfields

Syntax

```
#pragma bitfields={reversed|default}
```

Parameters

`reversed`	Bitfield members are placed from the most significant bit to the least significant bit.
`default`	Bitfield members are placed from the least significant bit to the most significant bit.

Description

Use this pragma directive to control the order of bitfield members.

By default, the 8051 IAR C/C++ Compiler places bitfield members from the least significant bit to the most significant bit in the container type. Use the `#pragma bitfields=reversed` directive to place the bitfield members from the most significant to the least significant bit. This setting remains active until you turn it off again with the `#pragma bitfields=default` directive.

See also

Bitfields, page 219.

constseg

Syntax

```
#pragma constseg=[__memoryattribute ]{SEGMENT_NAME|default}
```

Parameters

`__memoryattribute`	An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.
`SEGMENT_NAME`	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.
`default`	Uses the default segment for constants.

Description

Use this pragma directive to place constant variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The setting remains active until you turn it off again with the `#pragma constseg=default` directive.

A constant placed in a named segment with the `#pragma constseg` directive must be located in ROM memory. This is the case when constants are located in `code` or `data_rom`. Otherwise, the memory where the segment should reside must be explicitly specified using the appropriate memory attribute.

Note: Non-initialized constant segments located in data memory can be placed in a named segment with the `#pragma dataseg` directive.

Example

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

dataseg

Syntax

`#pragma dataseg=[__memoryattribute ]{SEGMENT_NAME|default}`

Parameters

`__memoryattribute`	An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.
`SEGMENT_NAME`	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.
`default`	Uses the default segment.

Description

Use this pragma directive to place variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The variable will not be initialized at startup, and can for this reason not have an initializer, which means it must be declared `__no_init`. The setting remains active until you turn it off again with the `#pragma constseg=default` directive.

Example

```
#pragma dataseg=__xdata MY_SEGMENT
__no_init char myBuffer[1000];
#pragma dataseg=default
```

diag_default

Syntax

`#pragma diag_default=tag[,tag,...]`

Parameters

`tag`	The number of a diagnostic message, for example the message number `Pe117`.

Description

Use this pragma directive to change the severity level back to default, or to the severity level defined on the command line by using any of the options `--diag_error`, `--diag_remark`, `--diag_suppress`, or `--diag_warnings`, for the diagnostic messages specified with the tags.

See also *Diagnostics*, page 184.

diag_error

Syntax `#pragma diag_error=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description Use this pragma directive to change the severity level to `error` for the specified diagnostics.

See also *Diagnostics*, page 184.

diag_remark

Syntax `#pragma diag_remark=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe177`.

Description Use this pragma directive to change the severity level to `remark` for the specified diagnostic messages.

See also *Diagnostics*, page 184.

diag_suppress

Syntax `#pragma diag_suppress=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description Use this pragma directive to suppress the specified diagnostic messages.

See also *Diagnostics*, page 184.

diag_warning

Syntax

```
#pragma diag_warning=tag[,tag,...]
```

Parameters

tag	The number of a diagnostic message, for example the message number `Pe826`.

Description

Use this pragma directive to change the severity level to `warning` for the specified diagnostic messages.

See also

Diagnostics, page 184.

include_alias

Syntax

```
#pragma include_alias ("orig_header" , "subst_header")
#pragma include_alias (<orig_header> , <subst_header>)
```

Parameters

orig_header	The name of a header file for which you want to create an alias.
subst_header	The alias for the original header file.

Description

Use this pragma directive to provide an alias for a header file. This is useful for substituting one header file with another, and for specifying an absolute path to a relative file.

This pragma directive must appear before the corresponding `#include` directives and `subst_header` must match its corresponding `#include` directive exactly.

Example

```
#pragma include_alias (<stdio.h> , <C:\MyHeaders\stdio.h>)
#include <stdio.h>
```

This example will substitute the relative file `stdio.h` with a counterpart located according to the specified path.

See also

Include file search procedure, page 182.

inline

Syntax

```
#pragma inline[=forced]
```

Parameters

`forced`	Disables the compiler's heuristics and forces inlining.

Description

Use this pragma directive to advise the compiler that the function whose declaration follows immediately after the directive should be inlined—that is, expanded into the body of the calling function. Whether the inlining actually takes place is subject to the compiler's heuristics.

This is similar to the C++ keyword `inline`, but has the advantage of being available in C code.

Specifying `#pragma inline=forced` disables the compiler's heuristics and forces inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question (like `printf`), an error message is emitted.

Note: Because specifying `#pragma inline=forced` disables the compiler's heuristics, including the inlining heuristics, the function declared immediately after the directive will not be inlined on optimization levels None or Low. No error or warning message will be emitted.

language

Syntax

```
#pragma language={extended|default}
```

Parameters

`extended`	Turns on the IAR Systems language extensions and turns off the `--strict_ansi` command line option.
`default`	Uses the language settings specified by compiler options.

Description

Use this pragma directive to enable the compiler language extensions or for using the language settings specified on the command line.

location

Syntax

```
#pragma location={address|NAME}
```

Parameters

address	The absolute address of the global or static variable for which you want an absolute location.
NAME	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.

Description

Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variable must be declared either `__no_init` or `const`. Alternatively, the directive can take a string specifying a segment for placing either a variable or a function whose declaration follows the pragma directive.

Example

```
#pragma location=0xFF20
__no_init volatile char PORT1; /* PORT1 is located at address
                                  0xFF20 */

#pragma location="foo"
char PORT1; /* PORT1 is located in segment foo */

/* A better way is to use a corresponding mechanism */
#define FLASH _Pragma("location=\"FLASH\"")
...
FLASH int i; /* i is placed in the FLASH segment */
```

See also

Controlling data and function placement in memory, page 165.

message

Syntax

```
#pragma message(message)
```

Parameters

message	The message that you want to direct to `stdout`.

Description

Use this pragma directive to make the compiler print a message to `stdout` when the file is compiled.

Example:

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

object_attribute

Syntax

```
#pragma object_attribute=object_attribute[,object_attribute,...]
```

Parameters

For a list of object attributes that can be used with this pragma directive, see *Object attributes*, page 240.

Description

Use this pragma directive to declare a variable or a function with an object attribute. This directive affects the definition of the identifier that follows immediately after the directive. The object is modified, not its type. Unlike the directive `#pragma type_attribute` that specifies the storing and accessing of a variable or function, it is not necessary to specify an object attribute in declarations.

Example

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

See also

General syntax rules for extended keywords, page 237.

optimize

Syntax

```
#pragma optimize=param[ param...]
```

Parameters

`balanced|size|speed`	Optimizes balanced between speed and size, optimizes for size, or optimizes for speed
`none|low|medium|high`	Specifies the level of optimization
`no_code_motion`	Turns off code motion
`no_cse`	Turns off common subexpression elimination
`no_inline`	Turns off function inlining
`no_tbaa`	Turns off type-based alias analysis
`no_unroll`	Turns off loop unrolling

Description

Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

The parameters `speed`, `size`, and `balanced` only have effect on the `high` optimization level and only one of them can be used as it is not possible to optimize for speed and size at the same time. It is also not possible to use preprocessor macros embedded in this pragma directive. Any such macro will not be expanded by the preprocessor.

Note: If you use the `#pragma optimize` directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.

Example

```
#pragma optimize=speed
int small_and_used_often()
{
    ...
}

#pragma optimize=size no_inline
int big_and_seldom_used()
{
    ...
}
```

__printf_args

Syntax

```
#pragma __printf_args
```

Description

Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example `%d`) is syntactically correct.

Example

```
#pragma __printf_args
int printf(char const *,...);


/* Function call */
printf("%d",x);  /* Compiler checks that x is a double */
```

register_bank

Syntax

```
#pragma register_bank=(0|1|2|3)
```

Parameters

`0|1|2|3`	The number of the register bank to be used.

Description

Use this pragma directive to specify the register bank to be used by the interrupt function declared after the pragma directive.

When a register bank has been specified, the interrupt function switches to the specified register bank. Because of this, registers `R0`–`R7` do not have to be individually saved on the stack. The result is a smaller and faster interrupt prolog and epilog.

The memory occupied by the used register banks cannot be used for other data.

Note: Interrupts that can interrupt each other cannot use the same register bank, because that can result in registers being unintentionally destroyed.

If no register bank is specified, the default bank will be used by the interrupt function.

Example:

```
#pragma register_bank=2
__interrupt void my_handler(void);
```

required

Syntax

`#pragma required=`*`symbol`*

Parameters

symbol	Any statically linked function or variable.

Description

Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol.

Use the directive if the requirement for a symbol is not otherwise visible in the application, for example if a variable is only referenced indirectly through the segment it resides in.

Example

```
const char copyright[] = "Copyright by me";
...
#pragma required=copyright
int main()
{...}
```

Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.

rtmodel

Syntax

```
#pragma rtmodel="key","value"
```

Parameters

`"key"`	A text string that specifies the runtime model attribute.
`"value"`	A text string that specifies the value of the runtime model attribute. Using the special value * is equivalent to not defining the attribute at all.

Description

Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.

This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value `*`. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

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

Example

```
#pragma rtmodel="I2C","ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

See also

Checking module consistency, page 111.

__scanf_args

Syntax

```
#pragma __scanf_args
```

Description

Use this pragma directive on a function with a scanf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example `%d`) is syntactically correct.

Example

```
#pragma __scanf_args
int printf(char const *,...);


/* Function call */
scanf("%d",x);  /* Compiler checks that x is a double */
```

segment

Syntax

```
#pragma segment="NAME" [__memoryattribute] [align]
```

Parameters

"*NAME*"	The name of the segment
`__memoryattribute`	An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used.
`align`	Specifies an alignment for the segment part. The value must be a constant integer expression to the power of two.

Description

Use this pragma directive to define a segment name that can be used by the segment operators `__segment_begin` and `__segment_end`. All segment declarations for a specific segment must have the same memory type attribute and alignment.

If an optional memory attribute is used, the return type of the segment operators `__segment_begin` and `__segment_end` is:

`void __memoryattribute *`.

Example

```
#pragma segment="MYHUGE" __huge 4
```

See also

Important language extensions, page 228. For more information about segments and segment parts, see the chapter *Placing code and data*.

type_attribute

Syntax

```
#pragma type_attribute=type_attribute[,type_attribute,...]
```

Parameters

For a list of type attributes that can be used with this pragma directive, see *Type attributes*, page 237.

Description

Use this pragma directive to specify IAR-specific *type attribute*s, which are not part of the ISO/ANSI C language standard. Note however, that a given type attribute may not be applicable to all kind of objects.

This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.

Example

In the following example, an `int` object with the memory attribute `__xdata` is defined:

```
#pragma type_attribute=__xdata
int x;
```

The following declaration, which uses extended keywords, is equivalent:

```
__xdata int x;
```

See also

See the chapter *Extended keywords* for more details.

vector

Syntax

```
#pragma vector=vector1[, vector2, vector3, ...]
```

Parameters

vector	The vector number(s) of an interrupt or trap function.

Description

Use this pragma directive to specify the vector(s) of an interrupt function whose declaration follows the pragma directive. Note that several vectors can be defined for each function.

Example!

```
#pragma vector=0x13
__interrupt void my_handler(void);
```

Intrinsic functions

This chapter gives reference information about the intrinsic functions, a predefined set of functions available in the compiler.

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

Summary of intrinsic functions

To use intrinsic functions in an application, include the header file `intrinsics.h`.

Note that the intrinsic function names start with double underscores, for example:

`__disable_interrupt`

The following table summarizes the intrinsic functions:

Intrinsic function	**Description**
`__disable_interrupt`	Disables interrupts
`__enable_interrupt`	Enables interrupts
`__get_interrupt_state`	Returns the interrupt state
`__no_operation`	Inserts a `NOP` instruction
`__parity`	Indicates the parity of the argument
`__set_interrupt_state`	Restores the interrupt state
`__tbac`	Atomic read, modify, write instruction

Table 45: Intrinsic functions summary

Descriptions of intrinsic functions

The following section gives reference information about each intrinsic function.

__disable_interrupt

Syntax

`void __disable_interrupt(void);`

Description

Disables interrupts by clearing bit 7 in the interrupt enable (`IE`) register.

__enable_interrupt

Syntax `void __enable_interrupt(void);`

Description Enables interrupts by setting bit `7` in the interrupt enable (IE) register.

__get_interrupt_state

Syntax `__istate_t __get_interrupt_state(void);`

Description Returns the global interrupt state. The return value can be used as an argument to the `__set_interrupt_state` intrinsic function, which will restore the interrupt state.

Example

```
__istate_t s = __get_interrupt_state();
__disable_interrupt();

  /* Do something */

__set_interrupt_state(s);
```

The advantage of using this sequence of code compared to using `__disable_interrupt` and `__enable_interrupt` is that the code in this example will not enable any interrupts disabled.

__no_operation

Syntax `void __no_operation(void);`

Description Inserts a `NOP` instruction.

__parity

Syntax `char __parity(char);`

Description Indicates the parity of the `char` argument; that is, whether the argument contains an even or an odd number of bits set to `1`. If the number is even, `0` is returned and if the number is odd, `1` is returned.

__set_interrupt_state

Syntax `void __set_interrupt_state(__istate_t);`

Descriptions Restores the interrupt state by setting the value returned by the `__get_interrupt_state` function.

For information about the `__istate_t` type, see *__get_interrupt_state*, page 274.

__tbac

Syntax `bool __tbac(bool` *`bitvar`*`);`

Description Use this intrinsic function to create semaphores or similar mutual-exclusion functions. It takes a single bit variable *`bitvar`* and uses the `JBC` assembler instruction to carry out an atomic read, modify, and write instruction (test bit and clear). The function returns the original value of *`bitvar`* (`0` or `1`) and resets *`bitvar`* to `0`.

Note: To use the `bool` type in C source code, see *Bool*, page 218.

The preprocessor

This chapter gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.

Overview of the preprocessor

The preprocessor of the 8051 IAR C/C++ Compiler adheres to the ISO/ANSI standard. The compiler also makes the following preprocessor-related features available to you:

- Predefined preprocessor symbols

  These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. For details, see *Descriptions of predefined preprocessor symbols*, page 278.
- User-defined preprocessor symbols defined using a compiler option

  In addition to defining your own preprocessor symbols using the `#define` directive, you can also use the option `-D`, see *-D*, page 194.
- Preprocessor extensions

  There are several preprocessor extensions, for example many pragma directives; for more information, see the chapter *Pragma directives* in this guide. Read also about the corresponding `_Pragma` operator and the other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 281.
- Preprocessor output

  Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 213.

Some parts listed by the ISO/ANSI standard are implementation-defined, for example the character set used in the preprocessor directives and inclusion of bracketed and quoted filenames. To read more about this, see *Preprocessing directives*, page 332.

Descriptions of predefined preprocessor symbols

The following table describes the predefined preprocessor symbols:

Predefined symbol	Identifies
`__BASE_FILE__`	A string that identifies the name of the base source file (that is, not the header file), being compiled. See also *`__FILE__`*, page 279, and *--no_path_in_file_macros*, page 208.
`__BUILD_NUMBER__`	A unique integer that identifies the build number of the compiler currently in use. The build number does not necessarily increase with a compiler that is released later.
`__CALLING_CONVENTION__`	An integer that identifies the calling convention in use. The value reflects the `--calling_convention` option and is defined to `0` for data overlay, `1` for idata overlay, `2` for idata reentrant, `3` for pdata reentrant, `4` for xdata reentrant, and `5` for extended stack reentrant.
`__CODE_MODEL__`	An integer that identifies the code model in use. The symbol reflects the `--code_model` option and is defined to `1` for Near, `2` for Banked, `3` for Far, and `4` for the Banked extended2 code model.
`__CONSTANT_LOCATION__`	An integer that identifies the default placement of constants and strings. The value reflects the `--place_constants` option and is defined to `0` for Data, `1` for Data ROM, and `2` for Code.
`__CORE__`	An integer that identifies the chip core in use. The symbol reflects the `--core` option and is defined to `0` for Plain, `2` for Extended`1`, and 3 for the Extended`2` core.
`__cplusplus`	An integer which is defined when the compiler runs in any of the C++ modes, otherwise it is undefined. When defined, its value is `199711L`. This symbol can be used with `#ifdef` to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*
`__DATA_MODEL__`	An integer that identifies the data model in use. The symbol reflects the `--data_model` option and can be defined to `0` for Tiny, `1` for Small, `2` for Large, `3` for Generic, and `4` for the Far data model.
`__DATE__`	A string that identifies the date of compilation, which is returned in the form "`Mmm dd yyyy`", for example "`Oct 30 2005`". *

Table 46: Predefined symbols

Predefined symbol	Identifies
`__embedded_cplusplus`	An integer which is defined to `1` when the compiler runs in any of the C++ modes, otherwise the symbol is undefined. This symbol can be used with `#ifdef` to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*
`__EXTENDED_DPTR__`	An integer that is set to `1` when 24-bit data pointers are used. Otherwise, when 16-bit data pointers are used, the symbol is undefined.
`__EXTENDED_STACK__`	An integer that is set to `1` when the extended stack is used. Otherwise, when the extended stack is not used, the symbol is undefined.
`__FILE__`	A string that identifies the name of the file being compiled, which can be the base source file as well as any included header file. See also *`__BASE_FILE__`*, page 278, and *--no_path_in_file_macros*, page 208.*
`__func__`	A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see *-e*, page 200. See also *`__PRETTY_FUNCTION__`*, page 280.
`__FUNCTION__`	A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see *-e*, page 200. See also *`__PRETTY_FUNCTION__`*, page 280.
`__IAR_SYSTEMS_ICC__`	An integer that identifies the IAR compiler platform. The current value is 7. Note that the number could be higher in a future version of the product. This symbol can be tested with `#ifdef` to detect whether the code was compiled by a compiler from IAR Systems.
`__ICC8051__`	An integer that is set to `1` when the code is compiled with the 8051 IAR C/C++ Compiler.
`__INC_DPSEL_SELECT__`	An integer that is set to `1` when the INC method is used for selecting the active data pointer. Otherwise, when the XOR method is used, the symbol is undefined.

Table 46: Predefined symbols (Continued)

Predefined symbol	Identifies
`__LINE__`	An integer that identifies the current source line number of the file being compiled, which can be the base source file as well as any included header file.*
`__NUMBER_OF_DPTRS__`	An integer that identifies to the number of data pointers being used; a value between `1` and `8`.
`__PRETTY_FUNCTION__`	A string that identifies the function name, including parameter types and return type, of the function in which the symbol is used, for example `"void func(char)"`. This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 200. See also *`__func__`*, page 279.
`__STDC__`	An integer that is set to `1`, which means the compiler adheres to the ISO/ANSI C standard. This symbol can be tested with `#ifdef` to detect whether the compiler in use adheres to ISO/ANSI C.*
`__STDC_VERSION__`	An integer that identifies the version of ISO/ANSI C standard in use. The symbols expands to `199409L`. This symbol does not apply in EC++ mode.*
`__SUBVERSION__`	An integer that identifies the version letter of the compiler version number, for example the C in 4.21C, as an ASCII character.
`__TIME__`	A string that identifies the time of compilation in the form `"hh:mm:ss"`.*
`__VER__`	An integer that identifies the version number of the IAR compiler in use. The value of the number is calculated in the following way: `(100 * the major version number + the minor version number)`. For example, for compiler version 3.34, 3 is the major version number and 34 is the minor version number. Hence, the value of `__VER__` is 334.
`__XOR_DPSEL_SELECT__`	An integer that is set to `1` when the XOR method is used for selecting the active data pointer. Otherwise, when the INC method is used, the symbol is undefined.

Table 46: Predefined symbols (Continued)

*** This symbol is required by the ISO/ANSI standard.**

__TID__

Syntax

```
__TID__
```

Description

Expands to the target identifier for the 8051 IAR C/C++ Compiler. The identifier contains the following parts:

- A number unique for each IAR compiler. For the MCS-51 microcontroller, the target identifier is `32` (`0x20`).
- The value of the `--core` option. The value is `0`, `2`, or `3` for the Plain, Extended1, and Extended2 core variant, respectively.
- The value corresponding to the `--data_model` option. The value is `0`, `1`, `2`, `3`, or `4` for the tiny, small, large, generic, and far data model, respectively.
- An intrinsic flag. This flag is set for 8051 because the compiler supports intrinsic functions.

The `__TID__`value is constructed as:

```
((i << 15) | (t << 8) | (c << 4) | m)
```

You can extract the values as follows:

```
i = (__TID__ >> 15) & 0x01;    /* intrinsic flag */

t = (__TID__ >> 8) & 0x7F;     /* target identifier */

c = (__TID__ >> 4) & 0x0F;     /* core option */

m = __TID__ & 0x0F;            /* data model */
```

Note: The use of `__TID__` is not recommended. We recommend that you use the symbols `__ICC8051__` and `__CORE__` instead.

Descriptions of miscellaneous preprocessor extensions

The following section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives, and ISO/ANSI directives.

NDEBUG

Description

This preprocessor symbol determines whether any assert macros you have written in your application shall be included or not in the built application.

If this symbol is not defined, all assert macros are evaluated. If the symbol is defined, all assert macros are excluded from the compilation. In other words, if the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you have written any assert code and build your application, you should define this symbol to exclude the assert code from the final application.

Note that the assert macro is defined in the `assert.h` standard include file.

In the IAR Embedded Workbench IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

_Pragma()

Syntax

`_Pragma("`*`string`*`")`

where *`string`* follows the syntax of the corresponding pragma directive.

Description

This preprocessor operator is part of the C99 standard and can be used, for example, in defines and has the equivalent effect of the `#pragma` directive.

Note: The `-e` option—enable language extensions—does not have to be specified.

Example

```
#if NO_OPTIMIZE
  #define NOOPT _Pragma("optimize=none")
#else
  #define NOOPT
#endif
```

See also

See the chapter *Pragma directives*.

#warning message

Syntax

`#warning` *`message`*

where *`message`* can be any string.

Description

Use this preprocessor directive to produce messages. Typically, this is useful for assertions and other trace utilities, similar to the way the ISO/ANSI standard `#error` directive is used.

__VA_ARGS__

Syntax

```
#define P(...)       __VA_ARGS__
#define P(x,y,...)   x + y + __VA_ARGS__
```

`__VA_ARGS__` will contain all variadic arguments concatenated, including the separating commas.

Description

Variadic macros are the preprocessor macro equivalents of `printf` style functions. `__VA_ARGS__` is part of the C99 standard.

Example

```
#if DEBUG
  #define DEBUG_TRACE(S,...) printf(S,__VA_ARGS__)
#else
  #define DEBUG_TRACE(S,...)
#endif
...
DEBUG_TRACE("The value is:%d\n",value);
```

will result in:

```
printf("The value is:%d\n",value);
```

Library functions

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

For detailed reference information about the library functions, see the online help system.

Introduction

The 8051 IAR C/C++ Compiler provides two different libraries:

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

Note that different customization methods are normally needed for these two libraries. For additional information, see the chapter *The DLIB runtime environment* and *The CLIB runtime environment*, respectively.

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For additional information about library functions, see the chapter *Implementation-defined behavior* in this guide.

HEADER FILES

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

LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic project settings*, page 6. The linker will include only those routines that are required—directly or indirectly—by your application.

REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant.

Most parts of the DLIB library are reentrant, but the following functions and parts are not reentrant as they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, as well as the C++ operators `new` and `delete`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- Multibyte functions—`mbrlen`, `mbrtowc`, `mbsrtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`
- The miscellaneous functions `setlocale`, `rand`, `atexit`, `strerror`, `strtok`
- Functions that use files in some way. This includes `printf`, `scanf`, `getchar`, and `putchar`. The functions `sprintf` and `sscanf` are not included.

For the CLIB library, the `qsort` function is non-reentrant, as well as functions that use files in some way. This includes `printf`, `scanf`, `getchar`, and `putchar`. The functions `sprintf` and `sscanf` are not included.

In addition, some functions share the same storage for `errno`. These functions are not reentrant, since an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it has been read. Among these functions are:

```
exp, exp10, ldexp, log, log10, pow, sqrt, acos, asin, atan2,
cosh, sinh, strtod, strtol, strtoul
```

Remedies for this are:

- Do not use non-reentrant functions in interrupt service routines
- Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of the ISO/ANSI standard for the programming language C. For additional information, see the chapter *Implementation-defined behavior* in this guide.
- Standard C library definitions, for user programs.
- Embedded C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code. It is described in the chapter *The DLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of MCS-51 features. See the chapter *Intrinsic functions* for more information.

In addition, the IAR DLIB Library includes some added C functionality, partly taken from the C99 standard, see *Added C functionality*, page 290.

C HEADER FILES

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

The following table lists the C header files:

Header file	Usage
`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
`inttypes.h`	Defining formatters for all types defined in `stdint.h`
`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

Table 47: Traditional standard C header files—DLIB

Header file	Usage
`stdbool.h`	Adds support for the `bool` data type in C.
`stddef.h`	Defining several useful types and macros
`stdint.h`	Providing integer characteristics
`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
`wchar.h`	Support for wide characters
`wctype.h`	Classifying wide characters

Table 47: Traditional standard C header files—DLIB (Continued)

C++ HEADER FILES

This section lists the C++ header files.

Embedded C++

The following table lists the Embedded C++ header files:

Header file	Usage
`complex`	Defining a class that supports complex arithmetic
`exception`	Defining several functions that control exception handling
`fstream`	Defining several I/O stream classes that manipulate external files
`iomanip`	Declaring several I/O stream 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 stream classes before they are necessarily defined
`iostream`	Declaring the I/O stream 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 stream classes that manipulate string containers
`stdexcept`	Defining several classes useful for reporting exceptions
`streambuf`	Defining classes that buffer I/O stream operations
`string`	Defining a class that implements a string container
`strstream`	Defining several I/O stream classes that manipulate in-memory character sequences

Table 48: Embedded C++ header files

The following table lists additional C++ header files:

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

Table 49: Additional Embedded C++ header files—DLIB

Extended Embedded C++ standard template library

The following table lists the Extended EC++ standard template library (STL) header files:

Header file	Description
`algorithm`	Defines several common operations on sequences
`deque`	A deque sequence container
`functional`	Defines several function objects
`hash_map`	A map associative container, based on a hash algorithm
`hash_set`	A set associative container, based on a hash algorithm
`iterator`	Defines common iterators, and operations on iterators
`list`	A doubly-linked list sequence container
`map`	A map associative container
`memory`	Defines facilities for managing memory
`numeric`	Performs generalized numeric operations on sequences
`queue`	A queue sequence container
`set`	A set associative container
`slist`	A singly-linked list sequence container
`stack`	A stack sequence container
`utility`	Defines several utility components
`vector`	A vector sequence container

Table 50: Standard template library header files

Using standard C libraries in C++

The C++ library works in conjunction with 15 of the header files from the standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, `cassert` and `assert.h`.

The following table shows the new header files:

Header file	Usage
`cassert`	Enforcing assertions when functions execute
`cctype`	Classifying characters
`cerrno`	Testing error codes reported by library functions
`cfloat`	Testing floating-point type properties
`cinttypes`	Defining formatters for all types defined in `stdint.h`
`climits`	Testing integer type properties
`clocale`	Adapting to different cultural conventions
`cmath`	Computing common mathematical functions
`csetjmp`	Executing non-local goto statements
`csignal`	Controlling various exceptional conditions
`cstdarg`	Accessing a varying number of arguments
`cstdbool`	Adds support for the `bool` data type in C.
`cstddef`	Defining several useful types and macros
`cstdint`	Providing integer characteristics
`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
`cwchar`	Support for wide characters
`cwctype`	Classifying wide characters

Table 51: New standard C header files—DLIB

LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

Certain C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call, for example `memcpy`, `memset`, and `strcat`.

ADDED C FUNCTIONALITY

The IAR DLIB Library includes some added C functionality, partly taken from the C99 standard.

The following include files provide these features:

- `ctype.h`
- `inttypes.h`

- `math.h`
- `stdbool.h`
- `stdint.h`
- `stdio.h`
- `stdlib.h`
- `wchar.h`
- `wctype.h`

ctype.h

In `ctype.h`, the C99 function `isblank` is defined.

inttypes.h

This include file defines the formatters for all types defined in `stdint.h` to be used by the functions `printf`, `scanf`, and all their variants.

math.h

In `math.h` all functions exist in a `float` variant and a `long double` variant, suffixed by `f` and `l` respectively. For example, `sinf` and `sinl`.

The following C99 macro symbols are defined:

`HUGE_VALF`, `HUGE_VALL`, `INFINITY`, `NAN`, `FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL`, `FP_ZERO`, `MATH_ERRNO`, `MATH_ERREXCEPT`, `math_errhandling`.

The following C99 macro functions are defined:

`fpclassify`, `signbit`, `isfinite`, `isinf`, `isnan`, `isnormal`, `isgreater`, `isless`, `islessequal`, `islessgreater`, `isunordered`.

The following C99 type definitions are added:

`float_t`, `double_t`.

stdbool.h

This include file makes the `bool` type available if the **Allow IAR extensions** (`-e`) option is used.

stdint.h

This include file provides integer characteristics.

stdio.h

In `stdio.h`, the following C99 functions are defined:

`vscanf`, `vfscanf`, `vsscanf`, `vsnprintf`, `snprintf`

The functions `printf`, `scanf`, and all their variants have added functionality from the C99 standard. For reference information about these functions, see the library reference available from the **Help** menu.

The following functions providing I/O functionality for libraries built without `FILE` support are definded:

`__write_array`	Corresponds to `fwrite` on `stdout`.
`__ungetchar`	Corresponds to `ungetc` on `stdout`.
`__gets`	Corresponds to `fgets` on `stdin`.

stdlib.h

In `stdlib.h`, the following C99 functions are defined:

`_Exit`, `llabs`, `lldiv`, `strtoll`, `strtoull`, `atoll`, `strtof`, `strtold`.

The function `strtod` has added functionality from the C99 standard. For reference information about this functions, see the library reference available from the **Help** menu.

The `__qsortbbl` function is defined; it provides sorting using a bubble sort algorithm. This is useful for applications that have a limited stack.

wchar.h

In `wchar.h`, the following C99 functions are defined:

`vfwscanf`, `vswscanf`, `vwscanf`, `wcstof`, `wcstolb`.

wctype.h

In `wctype.h`, the C99 function `iswblank` is defined.

IAR CLIB Library

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

- Standard C library definitions available for user programs. These are documented in this chapter.

- The system startup code. It is described in the chapter *The CLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of MCS-51 features. See the chapter *Intrinsic functions* for more information.

LIBRARY DEFINITIONS SUMMARY

This following table lists the header files specific to the CLIB library:

Header file	Description
`assert.h`	Assertions
`ctype.h*`	Character handling
`errno.h`	Error return values
`float.h`	Limits and sizes of floating-point types
`iccbutl.h`	Low-level routines
`limits.h`	Limits and sizes of integral types
`math.h`	Mathematics
`setjmp.h`	Non-local jumps
`stdarg.h`	Variable arguments
`stdbool.h`	Adds support for the `bool` data type in C
`stddef.h`	Common definitions including `size_t`, `NULL`, `ptrdiff_t`, and `offsetof`
`stdio.h`	Input/output
`stdlib.h`	General utilities
`string.h`	String handling

Table 52: IAR CLIB Library header files

*** The functions `is`*`xxx`*, `toupper`, and `tolower` declared in the header file `ctype.h` evaluate their argument more than once. This is not according to the ISO/ANSI standard.**

Segment reference

The 8051 IAR C/C++ Compiler places code and data into named segments which are referred to by the IAR XLINK Linker. Details about the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output from the compiler.

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

Summary of segments

The table below lists the segments that are available in the 8051 IAR C/C++ Compiler:

Segment	Description
`BANKED_CODE`	Holds code declared `__banked_func`.
`BANKED_CODE_EXT2_AC`	Holds located constant data, when using the Banked extended2 code model.
`BANKED_CODE_EXT2_AN`	Holds located uninitialized data, when using the Banked extended2 code model.
`BANKED_CODE_EXT2_C`	Holds constant data, when using the Banked extended2 code model.
`BANKED_CODE_EXT2_N`	Holds `__no_init` static and global variables, when using the Banked extended2 code model.
`BANKED_CODE_ INTERRUPTS_EXT2`	Holds `__interrupt` functions when compiling for the extended2 core.
`BANKED_EXT2`	Holds springboard functions when compiling for the extended2 core.
`BANK_RELAYS`	Holds relay functions for bank switching when compiling for the Banked code model.
`BDATA_AN`	Holds `__bdata` located uninitialized data.
`BDATA_I`	Holds `__bdata` static and global initialized variables.
`BDATA_ID`	Holds initial values for `__bdata` static and global variables in `BDATA_I`.
`BDATA_N`	Holds `__no_init __bdata` static and global variables.
`BDATA_Z`	Holds zero-initialized `__bdata` static and global variables.

Table 53: Segment summary

Segment	Description
BIT_N	Holds __no_init __bit static and global variables.
BREG	Holds the compiler's virtual bit register.
CODE_AC	Holds __code located constant data.
CODE_C	Holds __code constant data.
CODE_N	Holds __no_init __code static and global variables.
CSTART	Holds the startup code.
DATA_AN	Holds __data located uninitialized data.
DATA_I	Holds __data static and global initialized variables.
DATA_ID	Holds initial values for __data static and global variables in DATA_I.
DATA_N	Holds __no_init __data static and global variables.
DATA_Z	Holds zero-initialized __data static and global variables.
DIFUNCT	Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before main is called.
DOVERLAY	Holds the static data overlay area.
EXT_STACK	Holds the Maxim (Dallas Semiconductor) 390/400 extended data stack.
FAR_AN	Holds __far located uninitialized data.
FAR_CODE	Holds code declared __far_func.
FAR_CODE_AC	Holds __far_code located constant data.
FAR_CODE_C	Holds __far_code constant data.
FAR_CODE_N	Holds __no_init __far_code static and global variables.
FAR_HEAP	Holds the heap used for dynamically allocated far data.
FAR_I	Holds __far static and global initialized variables.
FAR_ID	Holds initial values for __far static and global variables in FAR_I.
FAR_N	Holds __no_init __far static and global variables.
FAR_ROM_AC	Holds __far_rom located constant data.
FAR_ROM_C	Holds __far_rom constant data.
FAR_Z	Holds zero-initialized __far static and global variables.
HUGE_AN	Holds __huge located uninitialized data.
HUGE_CODE_AC	Holds __huge_code located constant data.
HUGE_CODE_C	Holds __huge_code constant data.
HUGE_CODE_N	Holds __no_init __huge_code static and global variables.

Table 53: Segment summary (Continued)

Segment	Description
HUGE_HEAP	Holds the heap used for dynamically allocated huge data.
HUGE_I	Holds __huge static and global initialized variables.
HUGE_ID	Holds initial values for __huge static and global variables in HUGE_I.
HUGE_N	Holds __no_init __huge static and global variables.
HUGE_ROM_AC	Holds __huge_rom located constant data.
HUGE_ROM_C	Holds __huge_rom constant data.
HUGE_Z	Holds zero-initialized __huge static and global variables.
IDATA_AN	Holds __idata located uninitialized data.
IDATA_I	Holds __idata static and global initialized variables.
IDATA_ID	Holds initial values for __idata static and global variables in IDATA_I.
IDATA_N	Holds __no_init __idata static and global variables.
IDATA_Z	Holds zero-initialized __idata static and global variables.
INTVEC	Contains the reset and interrupt vectors.
INTVEC_EXT2	Contains the reset and interrupt vectors when the core is Extended2.
IOVERLAY	Holds the static idata overlay area.
ISTACK	Holds the internal data stack.
IXDATA_AN	Holds __ixdata located uninitialized data.
IXDATA_I	Holds __ixdata static and global initialized variables.
IXDATA_ID	Holds initial values for __ixdata static and global variables in IXDATA_I.
IXDATA_N	Holds __no_init __ixdata static and global variables.
IXDATA_Z	Holds zero-initialized __ixdata static and global variables.
NEAR_CODE	Holds code declared __near_func.
PDATA_AN	Holds __pdata located uninitialized data.
PDATA_I	Holds __pdata static and global initialized variables.
PDATA_ID	Holds initial values for __pdata static and global variables in PDATA_I.
PDATA_N	Holds __no_init __pdata static and global variables.
PDATA_Z	Holds zero-initialized __pdata static and global variables.
PSP	Holds the stack pointer to the pdata stack.

Table 53: Segment summary (Continued)

Segment	Description
`PSTACK`	Holds the pdata stack.
`RCODE`	Holds code declared `__near_func`.
`SFR_AN`	Holds `__sfr` located uninitialized data.
`VREG`	Contains the compiler's virtual register area.
`XDATA_AN`	Holds `__xdata` located uninitialized data.
`XDATA_HEAP`	Holds the heap used for dynamically allocated data.
`XDATA_I`	Holds `__xdata` static and global initialized variables.
`XDATA_ID`	Holds initial values for `__xdata` static and global variables in `XDATA_I`.
`XDATA_N`	Holds `__no_init __xdata` static and global variables.
`XDATA_ROM_AC`	Holds `__xdata_rom` located constant data.
`XDATA_ROM_C`	Holds `__xdata_rom` constant data.
`XDATA_Z`	Holds zero-initialized `__xdata` static and global variables.
`XSP`	Holds the stack pointer to the xdata stack.
`XSTACK`	Holds the xdata stack.

Table 53: Segment summary (Continued)

Descriptions of segments

This section gives reference information about each segment.

The segments are placed in memory by using the segment placement linker directives `-Z` and `-P`, for sequential and packed placement, respectively. Some segments cannot use packed placement, as their contents must be continuous.

In each description, the segment memory type—`CODE`, `CONST`, or `DATA`—indicates whether the segment should be placed in ROM or RAM memory; see Table 12, *XLINK segment memory types*, page 68.

For information about the `-Z` and the `-P` directives, see the *IAR Linker and Library Tools Reference Guide.*

For information about how to define segments in the linker command file, see *Customizing the linker command file*, page 69.

For detailed information about the extended keywords mentioned here, see the chapter *Extended keywords*.

BANKED_CODE

Description	Holds program code declared `__banked_func`, which is the default in the Banked code model.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space.
Access type	Read-only

BANKED_CODE_EXT2_AC

Description

Holds located constant data, when using the Banked extended2 code model. The segment also holds default-declared initialized located `const` objects if the compiler option `--place_constants=code` has been specified.

Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

BANKED_CODE_EXT2_AN

Description

Holds `__no_init` located data, when using the Banked extended2 code model. Unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified, the segment also holds located non-initialized objects declared `__data const`.

Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

BANKED_CODE_EXT2_C

Description

Holds constant data, when using the Banked extended2 code model.

This segment also holds default-declared non-located constant data and strings if the compiler option `--place_constants=code` has been specified.

Segment memory type	`CODE`
Memory placement	`0F0000-0FFFFF`

Access type	Read/write
See also	--*output*, -*o*, page 212

BANKED_CODE_EXT2_N

Description	Holds static and global `__no_init` variables, when using the Banked extended2 code model. This segment also holds default-declared non-located constant data and strings if the compiler option `--place_constants=code` has been specified.
Segment memory type	`CODE`
Memory placement	`0F0000-0FFFFF`
Access type	Read/write
See also	--*output*, -*o*, page 212

BANKED_CODE_INTERRUPTS_EXT2

Description	Holds `__interrupt` functions when compiling for the extended2 core.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space but must be located in the same bank as the segment `INTVEC_EXT2`.
Access type	Read/write

BANKED_EXT2

Description	Holds the springboard functions, that is functions that need to be copied to every bank when compiling for the extended2 core.
Segment memory type	`CODE`
Memory placement	`10000`
Access type	Read-only

BANK_RELAYS

Description	Holds the relay functions that are used for bank switching when compiling for the extended2 core and the Banked extended2 code model.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere within `0x0–0xFFFF`, but must be located in the root bank.
Access type	Read-only

BDATA_AN

Description	Holds `__no_init __bdata` located data. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

BDATA_I

Description	Holds `__bdata` static and global initialized variables initialized by copying from the segment `BDATA_ID` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`DATA`
Memory placement	`0x20–0x2F`
Access type	Read/write

BDATA_ID

Description	Holds initial values for `__bdata` static and global variables in the `BDATA_I` segment. These values are copied from `BDATA_ID` to `BDATA_I` at application startup.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.

Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space.
Access type	Read-only

BDATA_N

Description	Holds static and global `__no_init __bdata` variables.
Segment memory type	`DATA`
Memory placement	`0x20-0x2F`
Access type	Read-only

BDATA_Z

Description	Holds zero-initialized `__bdata` static and global variables. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`DATA`
Memory placement	`0x20-0x2F`
Access type	Read/write

BIT_N

Description	Holds static and global `__no_init __bit` variables.
Segment memory type	`BIT`
Memory placement	`0x00-0x7F`

Access type	Read-only

BREG

Description	Holds the compiler's virtual bit register.
Segment memory type	`BIT`
Memory placement	`0x00–0x7F`
Access type	Read/write

CODE_AC

Description	Holds `__code` located constant data. The segment also holds `const` objects if the compiler option `--place_constants=code` has been specified. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

CODE_C

Description	Holds `__code` constant data. The segment also holds constant data and strings if the compiler option `--place_constants=code` has been specified.
Segment memory type	`CODE`
Memory placement	`0–0xFFFF`
Access type	Read-only
See also	*--output, -o*, page 212

CODE_N

Description	Holds static and global `__no_init __code` variables. The segment also holds constant data and strings if the compiler option `--place_constants=code` has been specified.
Segment memory type	`CODE`

Memory placement	`0–0xFFFF`
Access type	Read-only
See also	*--output, -o*, page 212

CSTART

Description	Holds the startup code. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`CODE`
Memory placement	This segment must be placed at the address where the chip starts executing after reset, which for the 8051 microcontroller is at the address zero.
Access type	Read-only

DATA_AN

Description	Holds `__no_init __data` located data. Unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified, the segment also holds located non-initialized objects declared `__data const` and, in the Tiny data model, default-declared located non-initialized constant objects. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

DATA_I

Description	Holds `__data` static and global initialized variables initialized by copying from the segment `DATA_ID` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`DATA`

Memory placement	`0x0-0x7F`
Access type	Read/write

DATA_ID

Description	Holds initial values for `__data` static and global variables in the `DATA_I` segment. These values are copied from `DATA_ID` to `DATA_I` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in memory.
Access type	Read-only

DATA_N

Description	Holds static and global `__no_init __data` variables.
Segment memory type	`DATA`
Memory placement	`0x0–0x7F`
Access type	Read/write

DATA_Z

Description	Holds zero-initialized `__data` static and global variables. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`DATA`
Memory placement	`0x0–0x7F`
Access type	Read/write

DIFUNCT

Description	Holds the dynamic initialization vector used by C++.
Segment memory type	`CONST`
Memory placement	In the small data model, this segment must be placed in the first 64 Kbytes of memory. In other data models, this segment can be placed anywhere in memory.
Access type	Read-only

DOVERLAY

Description	Holds the static overlay area for functions called using the data overlay calling convention.
Segment memory type	`DATA`
Memory placement	`0x0–0x7F`
Access type	Read/write

EXT_STACK

Description	Holds the extended data stack.
Segment memory type	`DATA`
Memory placement	`0x0–0x7F`
Access type	Read/write
See also	*The stacks*, page 76.

FAR_AN

Description	Holds `__no_init __far` located data. Unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified, the segment also holds located non-initialized objects declared `__far const` and, in the Far data model, default-declared located non-initialized constant objects. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

FAR_CODE

Description	Holds application code declared `__far_func`.
Segment memory type	`CODE`
Memory placement	This segment must be placed in the code memory space.
Access type	Read-only

FAR_CODE_AC

Description	Holds `__far_code` located constant data. In the Far data model, the segment also holds default-declared initialized located `const` objects, if the compiler option `--place_constants=code` has been specified. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

FAR_CODE_C

Description	Holds `__far_code` constant data. In the Far data model, the segment also holds default-declared non-initialized constant data, if the compiler option `--place_constants=code` has been specified.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space.
Access type	Read-only

See also	-*-output, -o*, page 212

FAR_CODE_N

Description	Holds static and global `__no_init __far_code` variables. In the Far data model, the segment also holds default-declared non-initialized constant data, if the compiler option `--place_constants=code` has been specified.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space.
Access type	Read-only
See also	-*-output, -o*, page 212

FAR_HEAP

Description	Holds the heap used for dynamically allocated data in far memory, in other words data allocated by `far_malloc` and `far_free`, and in C++, `new` and `delete`. **Note:** This segment is only used when you use the DLIB library.
Segment memory type	`XDATA`
Memory placement	This segment can be placed anywhere in external data memory.
Access type	Read/write
See also	*The heap*, page 79 and *New and Delete operators*, page 154.

FAR_I

Description	Holds `__far` static and global initialized variables initialized by copying from the segment `FAR_ID` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`XDATA`

Memory placement	This segment must be placed in the external data memory space.
Access type	Read-only

FAR_ID

Description	Holds initial values for `__far` static and global variables in the `FAR_I` segment. These values are copied from `FAR_ID` to `FAR_I` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space.
Access type	Read-only

FAR_N

Description	Holds static and global `__no_init __far` variables. Unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified, the segment also holds non-initialized objects declared `__far const` and, in the Far data model, default-declared non-initialized constant objects.
Segment memory type	`CONST`
Memory placement	This segment must be placed in the external data memory space.
Access type	Read-only

FAR_ROM_AC

Description	Holds `__far_rom` located constant data. In the Far data model, the segment also holds default-declared initialized located `const` objects if the compiler option `--place_constants=data_rom` has been specified. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

FAR_ROM_C

Description	Holds `__far_rom` constant data. In the Far data model, the segment also holds default-declared non-located constant data and strings if the compiler option `--place_constants=data_rom` has been specified.
Segment memory type	`CONST`
Memory placement	This segment can be placed anywhere in the external data memory space.
Access type	Read-only
See also	*--output, -o*, page 212

FAR_Z

Description	Holds zero-initialized `__far` static and global variables. Unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified, the segment also holds non-initialized or zero-initialized `__far` constants. In the far data model, the segment also holds default-declared zero-initialized constant objects unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`XDATA`
Memory placement	This segment must be placed in the external data memory space.
Access type	Read/write
See also	*--output, -o*, page 212

HUGE_AN

Description	Holds `__no_init __huge` located data. Also holds located non-initialized objects declared `__huge const` unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified.

Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

HUGE_CODE_AC

Description	Holds `__huge_code` located constant data. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

HUGE_CODE_C

Description	Holds `__huge_code` constant data.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space.
Access type	Read-only

HUGE_CODE_N

Description	Holds static and global `__no_init __huge_code` variables.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space.
Access type	Read-only

HUGE_HEAP

Description	Holds the heap used for dynamically allocated data in huge memory, in other words data allocated by `huge_malloc` and `huge_free`, and in C++, `new` and `delete`. **Note:** This segment is only used when you use the DLIB library.
Segment memory type	`XDATA`

Memory placement	This segment can be placed anywhere in the external data memory space.
Access type	Read/write
See also	*The heap*, page 79 and *New and Delete operators*, page 154.

HUGE_I

Description	Holds `__huge` static and global initialized variables initialized by copying from the segment `HUGE_ID` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`XDATA`
Memory placement	This segment can be placed anywhere in the external data memory space.
Access type	Read/write

HUGE_ID

Description	Holds initial values for `__huge` static and global variables in the `HUGE_I` segment. These values are copied from `HUGE_ID` to `HUGE_I` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space.
Access type	Read-only

HUGE_N

Description	Holds static and global `__no_init __huge` variables. Unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified, the segment also holds non-initialized objects declared `__huge const`.
Segment memory type	`XDATA`
Memory placement	This segment can be placed anywhere in the external data memory space.
Access type	Read-only

HUGE_ROM_AC

Description	Holds `__huge_rom` located constant data. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

HUGE_ROM_C

Description	Holds `__huge_rom` constant data.
Segment memory type	`CONST`
Memory placement	This segment can be placed anywhere in the external data memory space.
Access type	Read-only

HUGE_Z

Description	Holds zero-initialized `__huge` static and global variables. Unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified, the segment also holds non-initialized or zero-initialized `__huge` constants. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.

Segment memory type	XDATA
Memory placement	This segment can be placed anywhere in the external data memory space.
Access type	Read/write

IDATA_AN

Description

Holds `__no_init __idata` located data. Unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified, the segment also holds located non-initialized objects declared `__idata const` and, in the small data model, default-declared located non-initialized constant objects.

Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

IDATA_I

Description

Holds `__idata` static and global initialized variables initialized by copying from the segment `IDATA_ID` at application startup.

Unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified, the segment also holds initialized objects declared `__idata const` and, in the small data model, default-declared initialized constant objects.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.

Segment memory type	DATA
Memory placement	0–0xFF
Access type	Read-only

IDATA_ID

Description

Holds initial values for `__idata` static and global variables in the `IDATA_I` segment. These values are copied from `IDATA_ID` to `IDATA_I` at application startup.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.

Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space.
Access type	Read-only

IDATA_N

Description	Holds static and global `__no_init` `__idata` variables.
Segment memory type	`DATA`
Memory placement	`0–0xFF`
Access type	Read/write

IDATA_Z

Description	Holds zero-initialized `__idata` static and global variables. Also holds, in the small data model, default-declared zero-initialized constant objects unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`IDATA`
Memory placement	`0x0–0xFF`
Access type	Read/write

INTVEC

Description	Holds the interrupt vector table generated by the use of the `__interrupt` extended keyword in combination with the `#pragma vector` directive.
Segment memory type	`CODE`
Memory placement	`0x0`
Access type	Read-only

INTVEC_EXT2

Description	When compiling for the extended2 core, this segment holds the interrupt vector table generated by the use of the `__interrupt` extended keyword in combination with the `#pragma vector` directive.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space but must be located in the same bank as the segment `BANKED_CODE_INTERRUPTS_EXT2`.
Access type	Read-only

IOVERLAY

Description	Holds the static overlay area for functions called using the idata overlay calling convention.
Segment memory type	`DATA`
Memory placement	`0x0–0xFF`
Access type	Read/write

ISTACK

Description	Holds the internal data stack.
Segment memory type	`IDATA`

Memory placement	`0x0-0xFF`
Access type	Read/write
See also	*The stacks*, page 76.

IXDATA_AN

Description	Holds `__no_init __ixdata` located data. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

IXDATA_I

Description	Holds `__ixdata` static and global initialized variables initialized by copying from the segment `IXDATA_ID` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`XDATA`
Memory placement	`0–0xFFFF`
Access type	Read-only

IXDATA_ID

Description	Holds initial values for `__ixdata` static and global variables in the `IXDATA_I` segment. These values are copied from `IXDATA_ID` to `IXDATA_I` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space.

Access type	Read-only

IXDATA_N

Description	Holds static and global `__no_init __ixdata` variables.
Segment memory type	`XDATA`
Memory placement	`0–0xFFFF`
Access type	Read/write

IXDATA_Z

Description	Holds zero-initialized `__ixdata` static and global variables. Also holds, in the small data model, default-declared zero-initialized constant objects unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`XDATA`
Memory placement	`0x0–0xFFFF`
Access type	Read-only

NEAR_CODE

Description	Holds program code declared `__near_func`.
Segment memory type	`CODE`
Memory placement	`0–0xFFFF`
Access type	Read-only

PDATA_AN

Description	Holds `__no_init __pdata` located data. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

PDATA_I

Description	Holds `__pdata` static and global initialized variables initialized by copying from the segment `PDATA_ID` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`XDATA`
Memory placement	`0–0x`*`nn`*`FF` (xdata memory) `0–0x`*`nnnn`*`FF` (far memory) This segment must be placed in one 256-byte page of xdata or far memory. Thus, *`nn`* can be anything from `00` to `FF` (xdata) and *`nnnn`* can be anything from `0000` to `FFFF` (far).
Access type	Read/write

PDATA_ID

Description	Holds initial values for `__pdata` static and global variables in the `PDATA_I` segment. These values are copied from `PDATA_ID` to `PDATA_I` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space.
Access type	Read-only

PDATA_N

Description	Holds static and global `__no_init` `__pdata` variables.
Segment memory type	`XDATA`
Memory placement	`0–0x`*`nn`*`FF` (xdata memory) `0–0x`*`nnnn`*`FF` (far memory) This segment must be placed in one 256-byte page of xdata or far memory. Thus, *`nn`* can be anything from `00` to `FF` (xdata) and *`nnnn`* can be anything from `0000` to `FFFF` (far).
Access type	Read/write

PDATA_Z

Description	Holds zero-initialized `__pdata` static and global variables. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`XDATA`
Memory placement	`0–0x`*`nn`*`FF` (xdata memory) `0–0x`*`nnnn`*`FF` (far memory) This segment must be placed in one 256-byte page of xdata or far memory. Thus, *`nn`* can be anything from `00` to `FF` (xdata) and *`nnnn`* can be anything from `0000` to `FFFF` (far).
Access type	Read/write

PSP

Description	Holds the stack pointers to the pdata stack.
Segment memory type	`DATA`
Memory placement	`0–0x7F`
Access type	Read/write
See also	*System startup*, page 97.

PSTACK

Description	Holds the parameter data stack.
Segment memory type	`XDATA`
Memory placement	`0`–`0x`*`nn`*`FF` (xdata memory) `0`–`0x`*`nnnn`*`FF` (far memory) This segment must be placed in one 256-byte page of xdata or far memory. Thus, *`nn`* can be anything from `00` to `FF` (xdata) and *`nnnn`* can be anything from `0000` to `FFFF` (far).
Access type	Read/write
See also	*The stacks*, page 76.

RCODE

Description	Holds assembler-written runtime library code.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space.
Access type	Read-only

SFR_AN

Description	Holds `__no_init __sfr` located data. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

VREG

Description	Holds the compiler's virtual register area.
Segment memory type	`DATA`
Memory placement	`0`–`0x7FF`

Access type	Read/write

XDATA_AN

Description	Holds `__no_init __xdata` located data. Unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified, the segment also holds located non-initialized objects declared `__xdata const` and, in the large data model, default-declared located non-initialized constant objects. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

XDATA_HEAP

Description	Holds the heap used for dynamically allocated data in xdata memory, in other words data allocated by `xdata_malloc` and `xdata_free`, and in C++, `new` and `delete`. **Note:** This segment is only used when you use the DLIB library.
Segment memory type	`XDATA`
Memory placement	This segment can be placed anywhere in the external data memory space.
Access type	Read/write
See also	*The heap*, page 79 and *New and Delete operators*, page 154.

XDATA_I

Description	Holds `__xdata` static and global initialized variables initialized by copying from the segment `XDATA_ID` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`XDATA`
Memory placement	`0–0xFFFF`

Access type	Read/write

XDATA_ID

Description	Holds initial values for `__xdata` static and global variables in the `XDATA_I` segment. These values are copied from `XDATA_ID` to `XDATA_I` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in the code memory space.
Access type	Read-only

XDATA_N

Description	Holds static and global `__no_init __xdata` variables. Unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified, the segment also holds non-initialized objects declared `__xdata const` and, in the Large data model, default-declared non-initialized constant objects.
Segment memory type	`XDATA`
Memory placement	`0–0xFFFF`
Access type	Read/write

XDATA_ROM_AC

Description	Holds `__xdata_rom` located constant data. In the Large data model, the segment also holds default-declared initialized located `const` objects if the compiler option `--place_constants=data_rom` has been specified. See *--place_constants*, page 212. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

XDATA_ROM_C

Description	Holds `__xdata_rom` constant data.
Segment memory type	`CONST`
Memory placement	`0`–`0xFFFF`
Access type	Read-only

XDATA_Z

Description	Holds zero-initialized `__huge` static and global variables. In the Large data model, the segment also holds default-declared zero-initialized constant objects unless the option `--place_constants=code` or `--place_constants=data_rom` has been specified. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`XDATA`
Memory placement	`0`–`0xFFFF`
Access type	Read/write
See also	*--output, -o*, page 212

XSP

Description	Holds the stack pointers to the xdata stack.
Segment memory type	`DATA`
Memory placement	`0`–`0x7F`
Access type	Read/write
See also	*System startup*, page 97.

XSTACK

Description	Holds the xdata stack.
Segment memory type	`XDATA`
Memory placement	`0–0FFFF`
Access type	Read/write
See also	*The stacks*, page 76.

Implementation-defined behavior

This chapter describes how the 8051 IAR C/C++ Compiler 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: The 8051 IAR C/C++ Compiler adheres to a freestanding implementation of the ISO standard for the C programming language. This means that parts of a standard library can be excluded in the implementation.

Descriptions of implementation-defined behavior

This section follows the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

TRANSLATION

Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

```
filename,linenumber level[tag]: message
```

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

ENVIRONMENT

Arguments to main (5.1.2.2.2.1)

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 for the IAR DLIB runtime environment, see *Customizing system initialization*, page 99. To change this behavior for the IAR CLIB runtime environment, see *Customizing system initialization*, page 123.

Interactive devices (5.1.2.3)

The streams `stdin` and `stdout` are treated as interactive devices.

IDENTIFIERS

Significant characters without external linkage (6.1.2)

The number of significant initial characters in an identifier without external linkage is 200.

Significant characters with external linkage (6.1.2)

The number of significant initial characters in an identifier with external linkage is 200.

Case distinctions are significant (6.1.2)

Identifiers with external linkage are treated as case-sensitive.

CHARACTERS

Source and execution character sets (5.2.1)

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set. The IAR CLIB Library does not support multibyte characters.

See *Locale*, page 104.

Bits per character in execution character set (5.2.4.2.1)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

Mapping of characters (6.1.3.4)

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

Unrepresented character constants (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

Character constant with more than one character (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Converting multibyte characters (6.1.3.4)

The IAR C/C++ Compiler has the following levels of locale support:

- If the compiler option `--enable_multibytes` is not used, the 'C' locale is used
- If the compiler option `--enable_multibytes` is used, the host operating system locale is used, but only in strings and comments.

  In addition, the IAR DLIB Library can support other locales than the 'C' locale. You can add support for multibyte characters by tailoring the library.

  The IAR CLIB Library does not support any other locales than the 'C' locale.

See *Locale*, page 104.

Range of 'plain' char (6.2.1.1)

A ‘plain’ `char` has the same range as an `unsigned char`.

INTEGERS

Range of integer values (6.1.2.5)

The representation of integer values are in the two's complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.

See *Basic data types*, page 218, for information about the ranges for the different integer types.

Demotion of integers (6.2.1.2)

Converting an integer to a shorter signed integer is made by truncation. If the value cannot be represented when converting an unsigned integer to a signed integer of equal length, the bit-pattern remains the same. In other words, a large enough value will be converted into a negative value.

Signed bitwise operations (6.3)

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

Sign of the remainder on integer division (6.3.5)

The sign of the remainder on integer division is the same as the sign of the dividend.

Negative valued signed right shifts (6.3.7)

The result of a right-shift of a negative-valued signed integral type preserves the sign-bit. For example, shifting `0xFF00` down one step yields `0xFF80`.

FLOATING POINT

Representation of floating-point values (6.1.2.5)

The representation and sets of the various floating-point numbers adheres to IEEE 854–1987. A typical floating-point number is built up of a sign-bit (`s`), a biased exponent (`e`), and a mantissa (`m`).

See *Floating-point types*, page 220, for information about the ranges and sizes for the different floating-point types: `float` and `double`.

Converting integer values to floating-point values (6.2.1.3)

When an integral number is cast to a floating-point value that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

Demoting floating-point values (6.2.1.4)

When a floating-point value is converted to a floating-point value of narrower type that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

ARRAYS AND POINTERS

size_t (6.3.3.4, 7.1.1)

See *size_t*, page 223, for information about `size_t`.

Conversion from/to pointers (6.3.4)

See *Casting*, page 223, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 223, for information about the `ptrdiff_t`.

REGISTERS

Honoring the register keyword (6.5.1)

User requests for register variables are not honored.

STRUCTURES, UNIONS, ENUMERATIONS, AND BITFIELDS

Improper access to a union (6.3.2.3)

If a union gets its value stored through a member and is then accessed using a member of a different type, the result is solely dependent on the internal storage of the first member.

Padding and alignment of structure members (6.5.2.1)

See the section *Basic data types*, page 218, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)

A 'plain' `int` bitfield is treated as a signed `int` bitfield. All integer types are allowed as bitfields.

Allocation order of bitfields within a unit (6.5.2.1)

Bitfields are allocated within an integer from least-significant to most-significant bit.

Can bitfields straddle a storage-unit boundary (6.5.2.1)

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

Integer type chosen to represent enumeration types (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

QUALIFIERS

Access to volatile objects (6.5.3)

Any reference to an object with volatile qualified type is an access.

DECLARATORS

Maximum numbers of declarators (6.5.4)

The number of declarators is not limited. The number is limited only by the available memory.

STATEMENTS

Maximum number of case statements (6.6.4.2)

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

PREPROCESSING DIRECTIVES

Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a `signed` character.

Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename was enclosed in angle brackets.

Character sequences (6.8.2)

Preprocessor directives use the source character set, 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)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect:

```
alignment
baseaddr
building_runtime
can_instantiate
codeseg
cspy_support
define_type_info
do_not_instantiate
early_dynamic_initialization
function
hdrstop
important_typedef
instantiate
keep_definition
```

```
memory
module_name
no_pch
once
__printf_args
public_equ
__scanf_args
section
STDC
system_include
warnings
```

Default __DATE__ and __TIME__ (6.8.8)

The definitions for `__TIME__` and `__DATE__` are always available.

IAR DLIB LIBRARY FUNCTIONS

The information in this section is valid only if the runtime library configuration you have chosen supports file descriptors. See the chapter *The DLIB runtime environment* for more information about runtime library configurations.

NULL macro (7.1.6)

The `NULL` macro is defined to `0`.

Diagnostic printed by the assert function (7.2)

The `assert()` function prints:

`*filename*:*linenr* *expression* -- assertion failed`

when the parameter evaluates to zero.

Domain errors (7.5.1)

`NaN` (Not a Number) will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets errno to ERANGE (7.5.1)

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

fmod() functionality (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns `NaN`; `errno` is set to `EDOM`.

signal() (7.7.1.1)

The signal part of the library is not supported.

Note: Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 107.

Terminating newline character (7.9.2)

`stdout` stream functions recognize either `newline` or `end of file (EOF)` as the terminating character for a line.

Blank lines (7.9.2)

Space characters written to the `stdout` stream immediately before a newline character are preserved. There is no way to read the line through the `stdin` stream that was written through the `stdout` stream.

Null characters appended to data written to binary streams (7.9.2)

No null characters are appended to data written to binary streams.

Files (7.9.3)

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 103.

remove() (7.9.4.1)

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 103.

rename() (7.9.4.2)

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 103.

%p in printf() (7.9.6.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `void *`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

%p in scanf() (7.9.6.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.

Reading ranges in scanf() (7.9.6.2)

A `-` (dash) character is always treated as a range symbol.

File position errors (7.9.9.1, 7.9.9.4)

On file position errors, the functions `fgetpos` and `ftell` store `EFPOS` in `errno`.

Message generated by perror() (7.9.10.4)

The generated message is:

`usersuppliedprefix``:`*`errormessage`*

Allocating zero bytes of memory (7.10.3)

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`.

Behavior of abort() (7.10.4.1)

The `abort()` function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

Behavior of exit() (7.10.4.3)

The argument passed to the `exit` function will be the return value returned by the `main` function to `cstartup`.

Environment (7.10.4.4)

The set of available environment names and the method for altering the environment list is described in *Environment interaction*, page 106.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 106.

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`*

Table 54: Message returned by strerror()—IAR DLIB library

The time zone (7.12.1)

The local time zone and daylight savings time implementation is described in *Time*, page 107.

clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 107.

IAR CLIB LIBRARY FUNCTIONS

NULL macro (7.1.6)

The `NULL` macro is defined to `(void *) 0.`

Diagnostic printed by the assert function (7.2)

The `assert()` function prints:

`Assertion failed:` *`expression`*`, file` *`Filename`*`, line` *`linenumber`*

when the parameter evaluates to zero.

Domain errors (7.5.1)

HUGE_VAL, the largest representable value in a double floating-point type, will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets errno to ERANGE (7.5.1)

The mathematics functions set the integer expression errno to ERANGE (a macro in errno.h) on underflow range errors.

fmod() functionality (7.5.6.4)

If the second argument to fmod() is zero, the function returns zero (it does not change the integer expression errno).

signal() (7.7.1.1)

The signal part of the library is not supported.

Terminating newline character (7.9.2)

stdout stream functions recognize either newline or end of file (EOF) as the terminating character for a line.

Blank lines (7.9.2)

Space characters written to the stdout stream immediately before a newline character are preserved. There is no way to read the line through the stdin stream that was written through the stdout stream.

Null characters appended to data written to binary streams (7.9.2)

There are no binary streams implemented.

Files (7.9.3)

There are no other streams than stdin and stdout. This means that a file system is not implemented.

remove() (7.9.4.1)

There are no other streams than stdin and stdout. This means that a file system is not implemented.

rename() (7.9.4.2)

There are no other streams than stdin and stdout. This means that a file system is not implemented.

%p in printf() (7.9.6.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `'char *'`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

%p in scanf() (7.9.6.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `'void *'`.

Reading ranges in scanf() (7.9.6.2)

A `-` (dash) character is always treated explicitly as a `-` character.

File position errors (7.9.9.1, 7.9.9.4)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

Message generated by perror() (7.9.10.4)

`perror()` is not supported.

Allocating zero bytes of memory (7.10.3)

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`. For more information, see *Dynamic memory on the heap*, page 42.

Behavior of abort() (7.10.4.1)

The `abort()` function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

Behavior of exit() (7.10.4.3)

The `exit()` function does not return.

Environment (7.10.4.4)

Environments are not supported.

system() (7.10.4.5)

The `system()` function is not supported.

Message returned by strerror() (7.11.6.2)

The messages returned by `strerror()` depending on the argument are:

Argument	Message
`EZERO`	`no error`
`EDOM`	`domain error`
`ERANGE`	`range error`
`<0 || >99`	`unknown error`
all others	`error No.`*`xx`*

Table 55: Message returned by strerror()—IAR CLIB library

The time zone (7.12.1)

The time zone function is not supported.

clock() (7.12.2.1)

The `clock()` function is not supported.