IAR Embedded Workbench

IAR C/C++ Compiler User Guide

for the 8051
Microcontroller Architecture

C8051-7

IAR SYSTEMS

EDITION NOTICE
Seventh edition: March 2017

Part number: C8051-7

This guide applies to version 10.x of IAR Embedded Workbench® for the 8051 microcontroller architecture.

Internal reference: M22, Mym8.0, tut2017.1, csrct2010.1, V_110411, IJOA.

Brief contents

Contents

Tables

Preface

Welcome to the *IAR C/C++ Compiler User Guide for 8051*. The purpose of this guide is to provide you with detailed reference information that can help you to use the 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

Read this guide if you plan to develop an application using the C or C++ language for the 8051 microcontroller and need detailed reference information on how to use the compiler.

REQUIRED KNOWLEDGE

To use the tools in IAR Embedded Workbench, you should have working knowledge of:

- The architecture and instruction set of the 8051 microcontroller (refer to the chip manufacturer's documentation)
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host computer.

For more information about the other development tools incorporated in the IDE, refer to their respective documentation, see *Other documentation*, page 33.

How to use this guide

When you start using the IAR C/C++ Compiler for 8051, 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 this product, we suggest that you first read the guide *Getting Started with IAR Embedded Workbench®* for an overview of the tools and the features that the IDE offers. The tutorials, which you can find in IAR Information Center, will help you get started using IAR Embedded Workbench.

What this guide contains

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

PART 1. USING THE COMPILER

- *Introduction to the IAR build tools* gives an introduction to the IAR build tools, which includes an overview of the tools, the programming languages, the available device support, and extensions provided for supporting specific features of the 8051 microcontroller.
- *Developing embedded applications* gives the information you need to get started developing your embedded software using the IAR build tools.
- *Understanding memory architecture*, page 59 gives an overview of the 8051 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 compiler and the linker.
- *Data storage* describes how to store data in memory.
- *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.
- *Linking overview* describes the linking process using the IAR XLINK Linker and the related concepts.
- *Linking your application* lists aspects that you must consider when linking your application, including using XLINK options and tailoring the linker configuration file.
- *The DLIB runtime environment* describes the DLIB runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization introducing the file `cstartup`, how to use modules for locale, and file I/O.
- *The CLIB runtime environment* gives an overview of the CLIB runtime libraries and how to customize them. The chapter also describes system initialization and introduces the file `cstartup`.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C* gives an overview of the two supported variants of the C language and an overview of the compiler extensions, such as extensions to Standard C.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.

- *Application-related considerations* discusses a selected range of application issues related to using the compiler and linker.
- *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.
- *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 functions to use 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.
- *C/C++ standard 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 for Standard C* describes how the compiler handles the implementation-defined areas of Standard C.
- *Implementation-defined behavior for C89* describes how the compiler handles the implementation-defined areas of the C language standard C89.

Other documentation

User documentation is available as hypertext PDFs and as a context-sensitive online help system in HTML format. You can access the documentation from the Information Center or from the **Help** menu in the IAR Embedded Workbench IDE. The online help system is also available via the F1 key.

USER AND REFERENCE GUIDES

The complete set of IAR Systems development tools is described in a series of guides. Information about:

- System requirements and information about how to install and register the IAR Systems products, is available in the booklet Quick Reference (available in the product box) and the *Installation and Licensing Guide*.
- Getting started using IAR Embedded Workbench and the tools it provides, is available in the guide *Getting Started with IAR Embedded Workbench®*.
- Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide for 8051.*
- Using the IAR C-SPY® Debugger, is available in the *C-SPY® Debugging Guide for 8051.*
- Programming for the IAR C/C++ Compiler for 8051, is available in the *IAR C/C++ Compiler User Guide for 8051.*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, is available in the *IAR Linker and Library Tools Reference Guide*.
- Programming for the IAR Assembler for 8051, is available in the *IAR Assembler User Guide for 8051.*
- Performing a static analysis using C-STAT and the required checks, is available in the *C-STAT® Static Analysis Guide*.
- Developing safety-critical applications using the MISRA C guidelines, is available in the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for 8051, is available in the *IAR Embedded Workbench® Migration Guide*.

Note: Additional documentation might be available depending on your product installation.

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Information about project management, editing, and building in the IDE
- Information about debugging using the IAR C-SPY® Debugger
- Reference information about the menus, windows, and dialog boxes in the IDE
- Compiler reference information
- 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. Note that if you select a function name in the editor window and press F1 while using the CLIB C standard library, you will get reference information for the DLIB C/EC++ standard library.

FURTHER READING

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

- Barr, Michael, and Andy Oram, ed. *Programming Embedded Systems in C and C++*. O'Reilly & Associates.
- Harbison, Samuel P. and Guy L. Steele (contributor). *C: A Reference Manual.* Prentice Hall.
- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C.* R&D Books.
- Mann, Bernhard. *C für Mikrocontroller.* Franzis-Verlag. [Written in German.]
- Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley.
- Meyers, Scott. *More Effective C++.* Addison-Wesley.
- Meyers, Scott. *Effective STL.* Addison-Wesley.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions.* Addison-Wesley.

The web site **isocpp.org** also has a list of recommended books about C++ programming.

WEB SITES

Recommended web sites:

- The Chip manufacturer's web site.
- The IAR Systems web site, **www.iar.com**, that holds application notes and other product information.
- The web site of the C standardization working group, **www.open-std.org/jtc1/sc22/wg14**.
- The web site of the C++ Standards Committee, **www.open-std.org/jtc1/sc22/wg21**.
- The C++ programming language web site, **isocpp.org**.

  This web site also has a list of recommended books about C++ programming.
- The C and C++ reference web site, **en.cppreference.com**.

Document conventions

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

When referring to a directory in your product installation, for example `8051\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench N.n\8051\doc`, where the initial digit of the version number reflects the initial digit of the version number of the IAR Embedded Workbench shared components.

TYPOGRAPHIC CONVENTIONS

The IAR Systems documentation set uses the following typographic conventions:

Style	Used for
`computer`	• Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
`parameter`	A placeholder for an actual value used as a parameter, for example *`filename`*`.h` where *`filename`* represents the name of the file.
[`option`**]**	An optional part of a directive, where **[** and **]** are not part of the actual directive, but any `[`, `]`, `{`, or `}` are part of the directive syntax.
{`option`**}**	A mandatory part of a directive, where **{** and **}** are not part of the actual directive, but any `[`, `]`, `{`, or `}` are part of the directive syntax.
`[option]`	An optional part of a command.
`[a|b|c]`	An optional part of a command with alternatives.
`{a|b|c}`	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
italic	• A cross-reference within this guide or to another guide. • Emphasis.
…	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.

Table 1: Typographic conventions used in this guide

Style	Used for
	Identifies warnings.

Table 1: Typographic conventions used in this guide (Continued)

NAMING CONVENTIONS

The following naming conventions are used for the products and tools from IAR Systems®, when referred to in the documentation:

Brand name	Generic term
IAR Embedded Workbench® for 8051	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for 8051	the IDE
IAR C-SPY® Debugger for 8051	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for 8051	the compiler
IAR Assembler™ for 8051	the assembler
IAR XLINK Linker™	XLINK, the linker
IAR XAR Library Builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Runtime Environment™	the DLIB runtime environment
IAR CLIB Runtime Environment™	the CLIB runtime environment

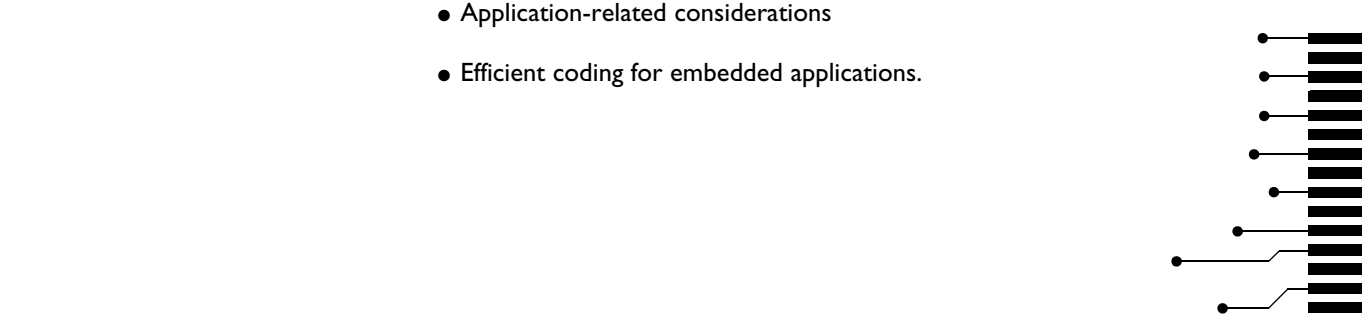
Table 2: Naming conventions used in this guide

Note: In this guide, *8051 microcontroller* refers to all microcontrollers compatible with the 8051 microcontroller architecture.

Part 1. Using the compiler

This part of the *IAR C/C++ Compiler User Guide for 8051* includes these chapters:

- Introduction to the IAR build tools
- Developing embedded applications
- Data storage
- Understanding memory architecture
- Functions
- Banked functions
- Linking overview
- Linking your application
- The DLIB runtime environment
- The CLIB runtime environment
- Assembler language interface
- Using C
- Using C++
- Application-related considerations
- Efficient coding for embedded applications.



Introduction to the IAR build tools

- The IAR build tools—an overview
- IAR language overview
- Device support
- Special support for embedded systems

The IAR build tools—an overview

In the IAR product installation you can find a set of tools, code examples, and user documentation, all suitable for developing software for 8051-based embedded applications. The tools allow you to develop your application in C, C++, or in assembler language.

IAR Embedded Workbench® is a very powerful Integrated Development Environment (IDE) that allows you to develop and manage complete embedded application projects. It provides an easy-to-learn and highly efficient development environment with maximum code inheritance capabilities, comprehensive and specific target support. IAR Embedded Workbench promotes a useful working methodology, and thus a significant reduction of the development time.

For information about the IDE, see the *IDE Project Management and Building Guide for 8051*.

The compiler, assembler, and linker can also be run from a command line environment, if you want to use them as external tools in an already established project environment.

IAR C/C++ COMPILER

The IAR C/C++ Compiler for 8051 is a state-of-the-art compiler that offers the standard features of the C and C++ languages, plus extensions designed to take advantage of the 8051-specific facilities.

IAR ASSEMBLER

The IAR Assembler for 8051 is a powerful relocating macro assembler with a versatile set of directives and expression operators. The assembler features a built-in C language preprocessor and supports conditional assembly.

THE IAR XLINK LINKER

The IAR XLINK Linker is a powerful, flexible software tool for use in the development of embedded controller applications. It is equally well suited for linking small, single-file, absolute assembler programs as it is for linking large, relocatable input, multi-module, C/C++, or mixed C/C++ and assembler programs.

To handle libraries, the library tools XAR and XLIB are included.

EXTERNAL TOOLS

For information about how to extend the tool chain in the IDE, see the *IDE Project Management and Building Guide for 8051*.

IAR language overview

The IAR C/C++ Compiler for 8051 supports:

- C, the most widely used high-level programming language in the embedded systems industry. You can build freestanding applications that follow these standards:
  - Standard C—also known as C99. Hereafter, this standard is referred to as *Standard C* in this guide.
  - C89—also known as C94, C90, C89, and ANSI C. This standard is required when MISRA C is enabled.
- C++, a modern object-oriented programming language with a full-featured library well suited for modular programming. Any of these standards can be used:
  - 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++ (EEC++)—EC++ with additional features such as full template support, namespace support, the new cast operators, as well as the Standard Template Library (STL).

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

For more information about C, see the chapter *Using C*.

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

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

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

Device support

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

SUPPORTED 8051 DEVICES

The IAR C/C++ Compiler for 8051 supports all devices based on the standard 8051 microcontroller architecture.

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 the device you are using, see *Basic project configuration*, page 56.

PRECONFIGURED SUPPORT FILES

The IAR product installation contains preconfigured files for supporting different devices. If you need additional files for device support, they can be created using one of the provided ones as a template.

Header files for I/O

Standard peripheral units are defined in device-specific I/O header files with the filename extension `h`. 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 configuration files

The `8051\config` directory contains ready-made linker configuration files for all supported devices. The files have the filename extension `xcl` and contain the information required by the linker. For more information about the linker configuration file, see *Placing code and data—the linker configuration file*, page 122 as well as the *IAR Linker and Library Tools Reference Guide*.

Device description files

The debugger handles several of the device-specific requirements, such as definitions of available memory areas, peripheral registers and groups of these, by using device description files. These files are located in the `8051\config` directory and they have the filename extension `ddf`. The peripheral registers and groups of these can be defined in separate files (filename extension `sfr`), which in that case are included in the `ddf` file. For more information about these files, see the *C-SPY® Debugging Guide for 8051*.

EXAMPLES FOR GETTING STARTED

Example applications are provided with IAR Embedded Workbench. You can use these examples to get started using the development tools from IAR Systems. You can also use the examples as a starting point for your application project.

You can find the examples in the `8051\examples` directory. The examples are ready to be used as is. They are supplied with ready-made workspace files, together with source code files and all other related files. For information about how to run an example project, see the *IDE Project Management and Building Guide for 8051*.

Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the 8051 microcontroller.

EXTENDED KEYWORDS

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

By default, language extensions are enabled in the IDE.

The command line option `-e` makes the extended keywords available, and reserves them so that they cannot be used as variable names. See, *-e*, page 302 for additional information.

For more information about the extended keywords, see the chapter *Extended keywords*. See also, *Data storage*, page 67 and *Functions*, page 95.

PRAGMA DIRECTIVES

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

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

For more information about 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 more information about the predefined symbols, see the chapter *The preprocessor*.

ACCESSING LOW-LEVEL FEATURES

For hardware-related parts of your application, accessing low-level features is essential. The compiler supports several ways of doing this: intrinsic functions, mixing C and assembler modules, and inline assembler. For information about the different methods, see *Mixing C and assembler*, page 191.

Developing embedded applications

- Developing embedded software using IAR build tools
- The build process—an overview
- Application execution—an overview
- Building applications—an overview
- Basic project configuration

Developing embedded software using IAR build tools

Typically, embedded software written for a dedicated microcontroller is designed as an endless loop waiting for some external events to happen. The software is located in ROM and executes on reset. You must consider several hardware and software factors when you write this kind of software. To your help, you have compiler options, extended keywords, pragma directives, etc.

MAPPING OF MEMORY

Embedded systems typically contain various types of memory, such as on-chip RAM, external DRAM or SRAM, ROM, EEPROM, or flash memory.

As an embedded software developer, you must understand the features of the different types of memory. For example, on-chip RAM is often faster than other types of memories, and variables that are accessed often would in time-critical applications benefit from being placed here. Conversely, some configuration data might be accessed seldom but must maintain their value after power off, so they should be saved in EEPROM or flash memory.

For efficient memory usage, the compiler provides several mechanisms for controlling placement of functions and data objects in memory. For more information, see *Controlling data and function placement in memory*, page 259. The linker places sections of code and data in memory according to the directives you specify in the linker configuration file, see *Placing code and data—the linker configuration file*, page 122.

COMMUNICATION WITH PERIPHERAL UNITS

If external devices are connected to the microcontroller, you might need to initialize and control the signaling interface, for example by using chip select pins, and detect and handle external interrupt signals. Typically, this must be initialized and controlled at runtime. The normal way to do this is to use special function registers (SFR). These are typically available at dedicated addresses, containing bits that control the chip configuration.

Standard peripheral units are defined in device-specific I/O header files with the filename extension `h`. See *Device support*, page 43. For an example, see *Accessing special function registers*, page 273.

EVENT HANDLING

In embedded systems, using *interrupts* is a method for handling external events immediately; for example, detecting that a button was pressed. In general, when an interrupt occurs in the code, the microcontroller immediately stops executing the code it runs, and starts executing an interrupt routine instead.

The compiler provides various primitives for managing hardware and software interrupts, which means that you can write your interrupt routines in C, see *Primitives for interrupts, concurrency, and OS-related programming*, page 97.

SYSTEM STARTUP

In all embedded systems, system startup code is executed to initialize the system—both the hardware and the software system—before the `main` function of the application is called.

As an embedded software developer, you must ensure that the startup code is located at the dedicated memory addresses, or can be accessed using a pointer from the vector table. This means that startup code and the initial vector table must be placed in non-volatile memory, such as ROM, EPROM, or flash.

A C/C++ application further needs to initialize all global variables. This initialization is handled by the linker and the system startup code in conjunction. For more information, see *Application execution—an overview*, page 52.

REAL-TIME OPERATING SYSTEMS

In many cases, the embedded application is the only software running in the system. However, using an RTOS has some advantages.

For example, the timing of high-priority tasks is not affected by other parts of the program which are executed in lower priority tasks. This typically makes a program

more deterministic and can reduce power consumption by using the CPU efficiently and putting the CPU in a lower-power state when idle.

Using an RTOS can make your program easier to read and maintain, and in many cases smaller as well. Application code can be cleanly separated in tasks which are truly independent of each other. This makes teamwork easier, as the development work can be easily split into separate tasks which are handled by one developer or a group of developers.

Finally, using an RTOS reduces the hardware dependence and creates a clean interface to the application, making it easier to port the program to different target hardware.

The build process—an overview

This section gives an overview of the build process; how the various build tools—compiler, assembler, and linker—fit together, going from source code to an executable image.

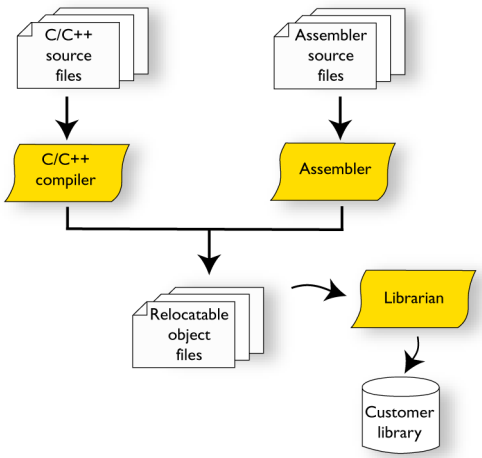
To get familiar with the process in practice, you should run one or more of the tutorials available from the IAR Information Center.

THE TRANSLATION PROCESS

There are two tools in the IDE that translate application source files to intermediary object files. The IAR C/C++ Compiler and the IAR Assembler. Both produce relocatable object files in the IAR UBROF format.

Note: The compiler can also be used for translating C source code into assembler source code. If required, you can modify the assembler source code which then can be assembled into object code. For more information about the IAR Assembler, see the *IAR Assembler User Guide for 8051.*

This illustration shows the translation process:



After the translation, you can choose to pack any number of modules into an archive, or in other words, a library. The important reason you should use libraries is that each module in a library is conditionally linked in the application, or in other words, is only included in the application if the module is used directly or indirectly by a module supplied as an object file. Optionally, you can create a library; then use the IAR XAR Library Builder or the IAR XLIB Librarian.

THE LINKING PROCESS

The relocatable modules, in object files and libraries, produced by the IAR compiler and assembler cannot be executed as is. To become an executable application, they must be *linked*.

Note: Modules produced by a toolset from another vendor can be included in the build as well. Be aware that this also might require a compiler utility library from the same vendor.

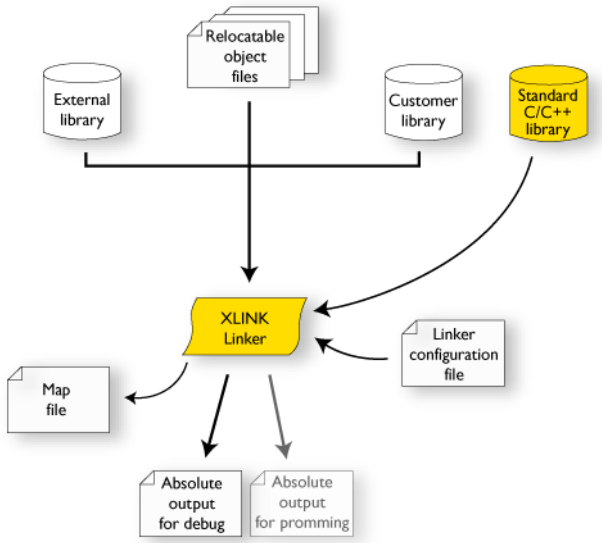
The IAR XLINK Linker (`xlink.exe`) is used for building the final application. Normally, the linker requires the following information as input:

- Several object files and possibly certain libraries
- The standard library containing the runtime environment and the standard language functions
- A program start label (set by default)
- The linker configuration file that describes placement of code and data in the memory of the target system

- Information about the output format.

  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 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.

This illustration shows the linking process:



Note: The Standard C/C++ library contains support routines for the compiler, and the implementation of the C/C++ standard library functions.

During the linking, the linker might produce error messages and logging messages on `stdout` and `stderr`. The log messages are useful for understanding why an application was linked the way it was, for example, why a module was included or a section removed.

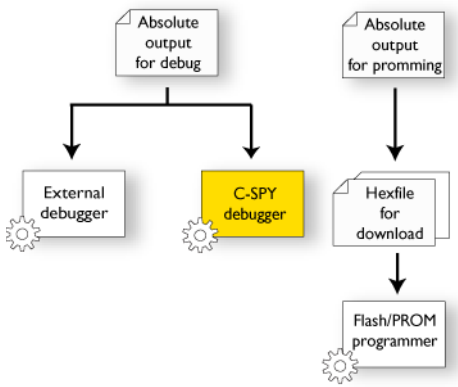
For more information about the procedure performed by the linker, see the *IAR Linker and Library Tools Reference Guide*.

AFTER LINKING

The IAR XLINK Linker produces an absolute object file in the output format you specify. After linking, the produced absolute executable image can be used for:

- Loading into the IAR C-SPY Debugger or any other compatible external debugger that reads UBROF.
- Programming to a flash/PROM using a flash/PROM programmer.

This illustration shows the possible uses of the absolute output files:



Application execution—an overview

This section gives an overview of the execution of an embedded application divided into three phases, the:

- Initialization phase
- Execution phase
- Termination phase.

THE INITIALIZATION PHASE

Initialization is executed when an application is started (the CPU is reset) but before the `main` function is entered. The initialization phase can for simplicity be divided into:

- Hardware initialization, which generally at least initializes the stack pointer.

  The hardware initialization is typically performed in the system startup code `cstartup.s51` and if required, by an extra low-level routine that you provide. It might include resetting/starting the rest of the hardware, setting up the CPU, etc, in preparation for the software C/C++ system initialization.

- Software C/C++ system initialization

  Typically, this includes assuring that every global (statically linked) C/C++ symbol receives its proper initialization value before the `main` function is called.

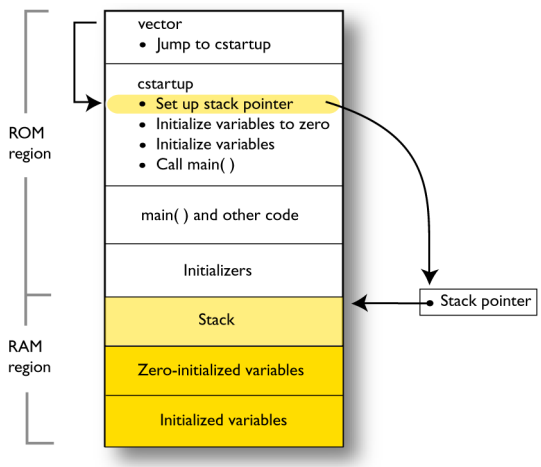
- Application initialization

  This depends entirely on your application. It can include setting up an RTOS kernel and starting initial tasks for an RTOS-driven application. For a bare-bone application, it can include setting up various interrupts, initializing communication, initializing devices, etc.

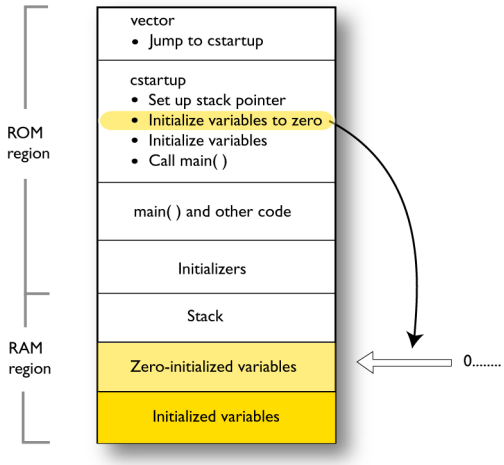
For a ROM/flash-based system, constants and functions are already placed in ROM. All symbols placed in RAM must be initialized before the `main` function is called. The linker has already divided the available RAM into different areas for variables, stack, heap, etc.

The following sequence of illustrations gives a simplified overview of the different stages of the initialization.

I When an application is started, the system startup code first performs hardware initialization, such as initialization of the stack pointer to point at the predefined stack area:

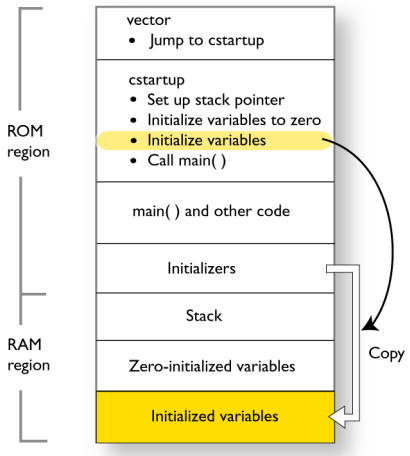


2 Then, memories that should be zero-initialized are cleared, in other words, filled with zeros:

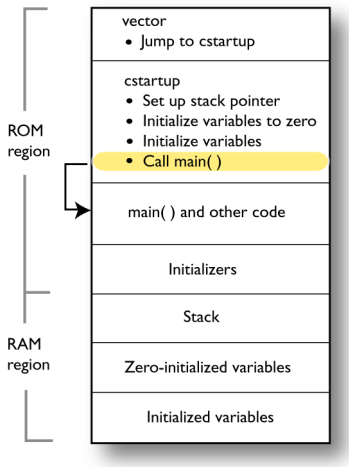


Typically, this is data referred to as *zero-initialized data*; variables declared as, for example, `int i = 0;`

3 For *initialized data*, data declared, for example, like `int i = 6;` the initializers are copied from ROM to RAM:



4 Finally, the `main` function is called:



For more information about each stage, see *System startup and termination*, page 164. For more information about initialization of data, see *Initialization at system startup*, page 123.

THE EXECUTION PHASE

The software of an embedded application is typically implemented as a loop which is either interrupt-driven or uses polling for controlling external interaction or internal events. For an interrupt-driven system, the interrupts are typically initialized at the beginning of the `main` function.

In a system with real-time behavior and where responsiveness is critical, a multi-task system might be required. This means that your application software should be complemented with a real-time operating system. In this case, the RTOS and the different tasks must also be initialized at the beginning of the `main` function.

THE TERMINATION PHASE

Typically, the execution of an embedded application should never end. If it does, you must define a proper end behavior.

To terminate an application in a controlled way, either call one of the Standard C library functions `exit`, `_Exit`, or `abort`, or return from `main`. If you return from `main`, the `exit` function is executed, which means that C++ destructors for static and global variables are called (C++ only) and all open files are closed.

Of course, in case of incorrect program logic, the application might terminate in an uncontrolled and abnormal way—a system crash.

For more information about this, see *System termination*, page 166.

Building applications—an overview

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

```
icc8051 myfile.c
```

You must also specify some critical options, see *Basic project configuration*, page 56.

On the command line, this line can be used for starting the linker:

```
xlink myfile.r51 myfile2.r51 -o a.d51 -f my_configfile.xcl -r
```

In this example, `myfile.r51` and `myfile2.r51` are object files, and `my_configfile.xcl` is the linker configuration file. The option `-o` specifies the name of the output file. The option `-r` is used for specifying the output format UBROF, which can be used for debugging in C-SPY®.

Note: By default, the label where the application starts is `__program_start`. You can use the `-s` command line option to change this.

When building a project, the IAR Embedded Workbench IDE can produce extensive build information in the **Build** messages window. This information can be useful, for example, as a base for producing batch files for building on the command line. You can copy the information and paste it in a text file. To activate extensive build information, right-click in the **Build** messages window and select **All** on the context menu.

Basic project configuration

This section gives an overview of the basic settings for the project setup that are needed to make the compiler and linker generate the best code for the 8051 device you are using. You can specify the options either from the command line interface or in the IDE.

You need to make settings for:

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

- Optimization settings
- Runtime environment, see *Setting up the runtime environment*, page 149
- Customizing the XLINK configuration, see the chapter *Linking your application*.

In addition to these settings, many other options and settings can fine-tune the result even further. For information about how to set options and for a list of all available options, see the chapter *Compiler options* and the *IDE Project Management and Building Guide for 8051*, respectively.

PROCESSOR CONFIGURATION

To make the compiler generate optimum code, you should configure it for the 8051 microcontroller you are using.

Core

The 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.

Use the `--core={plain|extended1|extended2}` option to select the core for which the code will be generated. This option reflects the addressing capability of your target microcontroller. For information about the cores, see *Basic project settings for hardware memory configuration*, page 61.

In the IDE, choose **Project>Options>General Options>Target** and choose the correct core from the **Core** drop-down list. Default options will then be automatically selected, together with device-specific configuration files for the linker and the debugger.

DATA MODEL

One of the characteristics of the 8051 microcontroller is a trade-off in how memory is accessed, ranging 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 compiler option `--data_model` to specify (among other things) the default memory placement of static and global variables, which also implies a default memory access method.

The chapter *Data storage* covers data models in greater detail. The chapter also covers how to override the default access method for individual variables.

CODE MODEL

The compiler supports code models that you can set on file- or function-level to specify the default placement of functions, which also implies the default function calling method.

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

CALLING CONVENTION

Use the compiler option `--calling_convention` 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 198.

DPTR SETUP

Some devices provide up to 8 data pointers. Using them can improve the execution of some library routines.

Use the compiler option `--dptr` 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 configuration file or in the IDE.

OPTIMIZATION FOR SPEED AND SIZE

The compiler's optimizer 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 choose 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 information about compiler optimizations and for more information about efficient coding techniques, see the chapter *Efficient coding for embedded applications*.

Understanding memory architecture

The following pages contain these topics:

- The 8051 microcontroller memory configuration
- Runtime model concepts for memory configuration
- Basic project settings for hardware memory configuration
- Using the DPTR register

The 8051 microcontroller memory configuration

The 8051 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 use 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 information about these files, see *Accessing special function registers*, page 273.

The area between `0x20` and `0x2F` is bit-addressable, as well as `0x80`, `0x88`, `0x90`, `0x98...0xF0`, and `0xF8`. Note that the compiler reserves one byte of the bit-addressable memory for internal use, see *Virtual registers*, page 92.

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.

Runtime model concepts for memory configuration

To use memory efficiently you must be familiar with the basic concepts relating to memory in the compiler and the linker.

COMPILER CONCEPTS

The 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 80 and *Memory types*, page 68, 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 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 configuration file*.

To read more about segments, see *Segments and memory*, page 120 and *Linking considerations*, page 127.

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.

If you make these basic project settings in the IDE, only valid combinations are possible to choose.

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 that the extended stack compiler option is used (`--extended_stack`).

†† Requires that the extended stack compiler option is not 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 the 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|far_generic|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` must be used.

†† Requires that the compiler option `--extended_stack` is not 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 Generic data model also: `__far`, `__far_rom`, `__far22`, `__far22_rom`, `__generic`, `__huge`, `__huge_rom`.

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

Note: Although the 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

The Mentor Graphics M8051W/M8051EW 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 the 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 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 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 301.

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

  The same SFR (DPL 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

  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 exemplify 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
```

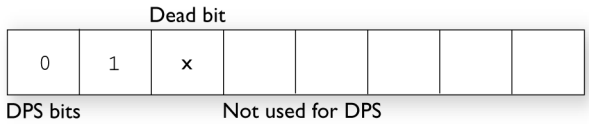
The symbol `?DPS` specifies where the `DPS` register is located. `?DPX` specifies the location of extended byte of the first data pointer. (The low and high address of the first data pointer is always located at the addresses `0x82` and `0x83`, respectively.) `?DPL1`, `?DPH1`, and `?DPX1` specify the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

SELECTING THE ACTIVE DATA POINTER

If you are using more than one DPTR, there are two different ways of switching active data pointers: incremention or XOR.

The incrementation method

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.



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

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 six data pointers are used, the default mask will be `0x07`.

Data storage

- Introduction
- Memory types
- Data models
- Constants and strings
- Storage of auto variables and parameters
- Dynamic memory on the heap
- Virtual registers

Introduction

The 8051 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.

For detailed information about the 8051 microcontroller memory configuration, see *Understanding memory architecture*, page 59.

DIFFERENT WAYS TO STORE DATA

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

- Auto variables

  All variables that are local to a function, except those declared static, are stored 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. For more information, see *Storage of auto variables and parameters*, page 83.

  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 *Storage of auto variables and parameters*, page 83.

- Global variables, module-static 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 80 and *Memory types*, page 68.

- Dynamically allocated data.

  An application can allocate data on the *heap*, where the data 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 90.

Memory types

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

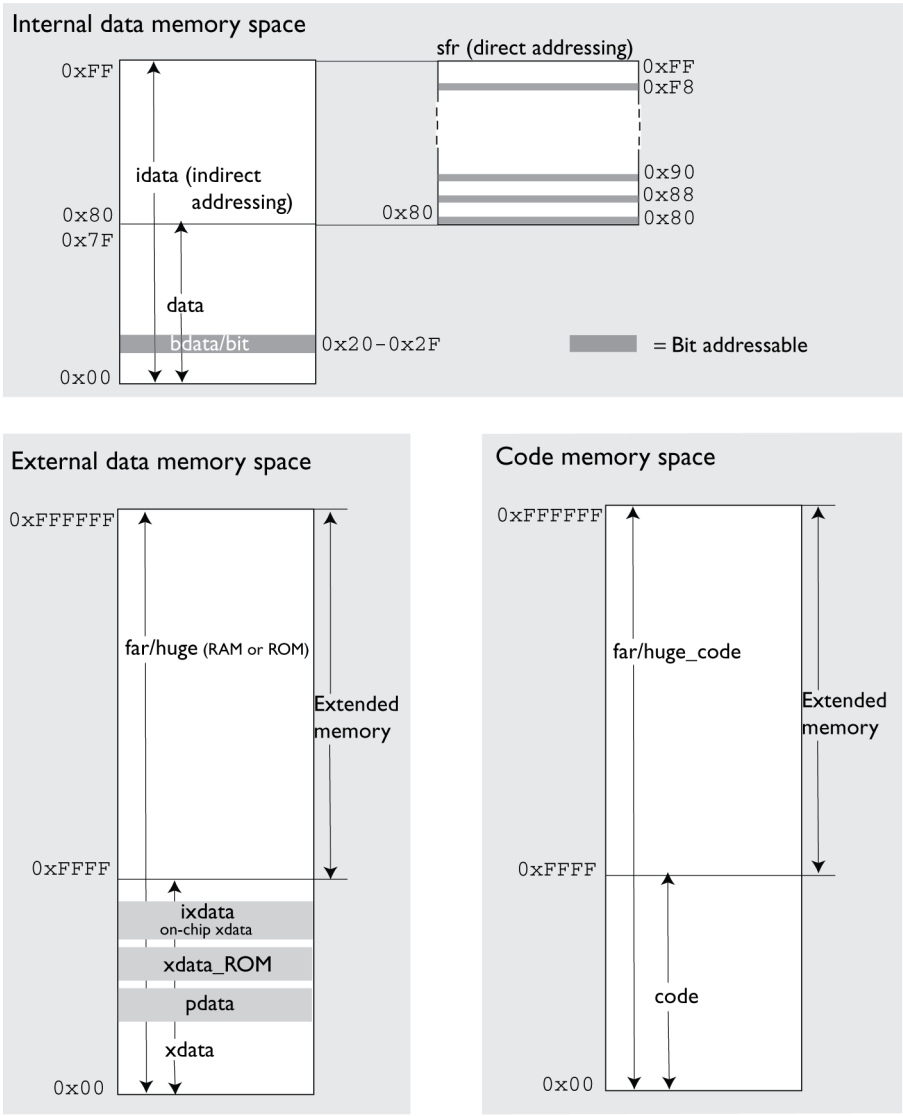
INTRODUCTION TO MEMORY TYPES

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

For example, the memory accessed using the xdata addressing method is called xdata memory.

To choose a default memory type that your application will use, select a *data model*. However, it is possible to specify—for individual variables or pointers—different memory types. 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.

This figure illustrates the different memory types and how they are associated with the different parts of memory:



Below is an overview of the various memory types.

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 *Accessing special function registers*, page 273.

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
- far22
- 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 *Data pointers*, page 331.

far22

The far22 memory type refers to the lower 4 Mbytes (`0x0-0x3FFFFF`) of the external data memory space. Use the `__far22` 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 Generic 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 *Data pointers*, page 331.

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. For these reasons, you should only use huge memory to store objects that do not fit in far or far22 memory.

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 *Data pointers*, page 331.
Far22 ROM	`__far22_rom`	`0x0-0x3FFFFF`	The size of such an object is limited to 64 Kbytes-1. For details about placement restrictions, see *Data pointers*, page 331.

Table 6: Memory types for ROM memory in external data memory space

Memory type	Attribute	Address range	Comments
Huge ROM	`__huge_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 6: Memory types for ROM memory in external data memory space (Continued)

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
- far22 code
- huge code.

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 *Data pointers*, page 331.

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

far22 code

The far22 code memory type refers to the lower 4 Mbytes (`0x0-0x3FFFFF`) of the code memory space. Use the `__far22_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 *Data pointers*, page 331.

This memory type is only available when the Far Generic 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. For these reasons, you should only use huge code memory to store objects that do not fit in far code or far22 code memory.

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

USING DATA MEMORY ATTRIBUTES

The 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 and pointers, which means that you can place data objects in other memory areas than the default memory. This also means that you can fine-tune the access method for each individual data object, which results in smaller code size.

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

Memory type	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`	`0x20–0x2F`	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`
Far22	XDATA	`__far22`	`0x0–0x3FFFFF`	64 Kbytes	`__far22`
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`
Far22 ROM	XDATA	`__far22_rom`	`0x0–0x3FFFFF`	64 Kbytes	`__far22`
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`
Far22 code	CODE	`__far22_code`	`0x0–0x3FFFFF`	64 Kbytes	`__far22`
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 to the internal data memory space, XDATA refers to the external data memory space, CODE refers at the code memory space. See *The 8051 microcontroller memory configuration*, page 59 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 61. For more information about the pointers that can be used, see *Pointer types*, page 330.

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

In the IDE, language extensions are enabled by default.

Use the `-e` compiler option to enable language extensions. See *-e*, page 302 for additional information.

For more information about each keyword, see *Descriptions of extended keywords*, page 342.

Syntax for type attributes used on data objects

Type attributes use almost the same syntax rules as the type qualifiers `const` and `volatile`. For example:

```
__pdata int i;
int __pdata j;
```

Both `i` and `j` are placed in pdata memory.

Unlike `const` and `volatile`, when a type attribute is used before the type specifier in a derived type, the type attribute applies to the object, or typedef itself, except in structure member declarations.

```
int __pdata * p;      /* pointer to integer in pdata memory */

int * __pdata p;      /* pointer in pdata memory */

__pdata int * p;      /* pointer in pdata memory */
```

In all cases, if a memory attribute is not specified, an appropriate default memory type is used, which depends on the data model in use.

Using a type definition can sometimes make the code clearer:

```
typedef __pdata int d16_int;
d16_int *q1;
```

`d16_int` is a typedef for integers in pdata memory. The variable `q1` can point to such integers.

You can also use the `#pragma type_attributes` directive to specify type attributes for a declaration. The type attributes specified in the pragma directive are applied to the data object or typedef being declared.

```
#pragma type_attribute=__pdata
int * q2;
```

The variable `q2` is placed in pdata memory.

For more examples of using memory attributes, see *More examples*, page 78.

Type definitions

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

```
/* Defines via a typedef */
typedef char __idata Byte;
typedef Byte *BytePtr;
Byte aByte;
BytePtr aBytePointer;

/* Defines directly */
__idata char aByte;
char __idata *aBytePointer;
```

POINTERS AND MEMORY TYPES

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

In the compiler, a pointer also points to some type of memory. The memory type is specified using a keyword before the asterisk. For example, a pointer that points to an integer stored in pdata memory is declared by:

```
int __pdata * myPtr;
```

Note that the location of the pointer variable `myPtr` is not affected by the keyword. In the following example, however, the pointer variable `myPtr2` is placed in pdata memory. Like `myPtr`, `myPtr2` points to a character in xdata memory.

```
char __xdata * __pdata myPtr2;
```

Differences between pointer types

A pointer must contain information needed to specify a memory location of a certain memory type. This means that the pointer sizes are different for different memory types. In the IAR C/C++ Compiler for 8051, 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.

For more information about pointers, see *Pointer types*, page 330.

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 pdata memory.

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

__pdata struct MyStruct gamma;
```

This declaration is incorrect:

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

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 pdata memory is declared. The function returns a pointer to an integer

in xdata memory. It makes no difference whether the memory attribute is placed before or after the data type.

`int myA;`	A variable defined in default memory determined by the data model in use.
`int __pdata myB;`	A variable in pdata memory.
`__xdata int myC;`	A variable in xdata memory.
`int * myD;`	A pointer stored in default memory. The pointer points to an integer in default memory.
`int __pdata * myE;`	A pointer stored in default memory. The pointer points to an integer in pdata memory.
`int __pdata * __xdata myF;`	A pointer stored in xdata memory pointing to an integer stored in pdata memory.
`int __xdata * MyFunction(` `  int __pdata *);`	A declaration of a function that takes a parameter which is a pointer to an integer stored in pdata memory. The function returns a pointer to an integer stored in xdata memory.

C++ AND MEMORY TYPES

Instances of C++ classes are placed into a memory (just like all other objects) either implicitly, or explicitly using memory type attributes or other IAR language extensions. Non-static member variables, like structure fields, are part of the larger object and cannot be placed individually into specified memories.

In non-static member functions, the non-static member variables of a C++ object can be referenced via the `this` pointer, explicitly or implicitly. The `this` pointer is of the default data pointer type unless class memory is used, see *Using IAR attributes with Classes*, page 231.

Static member variables can be placed individually into a data memory in the same way as free variables.

All member functions except for constructors and destructors can be placed individually into a code memory in the same way as free functions.

For more information about C++ classes, see *Using IAR attributes with Classes*, page 231.

Data models

Use *data models* to specify in which part of memory the compiler should place static and global variables by default. This means that the data model controls:

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

The data model only specifies the default memory type. It is possible to override this for individual variables and pointers. For information about how to specify a memory type for individual objects, see *Using data memory attributes*, page 74.

The data model you use might restrict the calling conventions and the location of constants and strings. When you compile 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 85.

SPECIFYING A DATA MODEL

There are six data models: Tiny, Small, Large, Generic, Far 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.

Your project can only use one data model at a time, and the same model must be used by all user modules and all library modules. However, you can override the default memory type for individual data objects and pointers by explicitly specifying a memory attribute, see *Using data memory attributes*, page 74.

This 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 Generic	`__far22`	`__generic`	—
Far	`__far`	`__far`	Extended1

Table 8: Data model characteristics

See the *IDE Project Management and Building Guide for 8051* for information about setting options in the IDE.

Use the `--data_model` option to specify the data model for your project; see *--data_model*, page 295.

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 bytes of internal data 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 external data memory by default. This memory can be accessed using 16-bit pointers. The default pointer type passed as a parameter will use two registers 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 bytes of the internal data memory space. The default pointer type passed as a parameter will use three registers or 3 bytes on the stack.

The Far Generic data model

The Far Generic data model uses 4 Mbytes of the code memory space, 4 Mbytes of the external data memory space, and 256 bytes of the internal data memory space. The default pointer type passed as a parameter will use three registers or 3 bytes on the stack.

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

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 registers or 3 bytes on the stack.

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

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`, `__far22_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, Far Generic, 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`, `__far22_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*, page 83.

PLACING CONSTANTS AND STRINGS IN CODE MEMORY

If you want to locate constants and strings in the code memory space, you must compile your project with the option `--place_constants=code`. It is important to note that standard 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. Instead, there are some 8051–specific CLIB library functions declared in `pgmspace.h` that allow access to strings in code memory.

Storage of auto variables and parameters

Variables that are defined inside a function—and not declared static—are named auto variables by the C standard. A few of these variables are placed in processor registers; the rest are placed 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 in the static overlay area. The stack is dynamically allocated at runtime, whereas the static overlay area is statically allocated at link time. For information about choosing between using the stack or the static overlay area for storing variables, see *Choosing a calling convention*, page 83.

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 IAR C/C++ Compiler for 8051 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 Standard C. The overlay calling conventions do not allow recursive functions and they are not reentrant, but they adhere to Standard 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 197.

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 291.

To specify the calling convention in the 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
Far Generic	Generic	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.

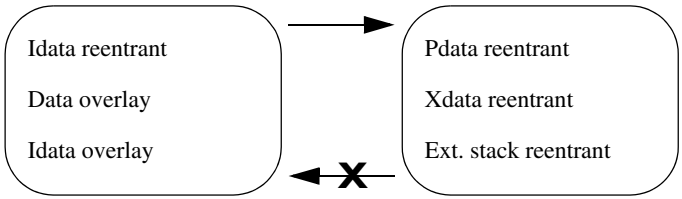
The same reasoning applies to return values of a structure type. The calling function will reserve space on the calling function's stack and pass a hidden parameter to the called function. This hidden parameter is a pointer referring to the location on the caller's stack where the return value should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

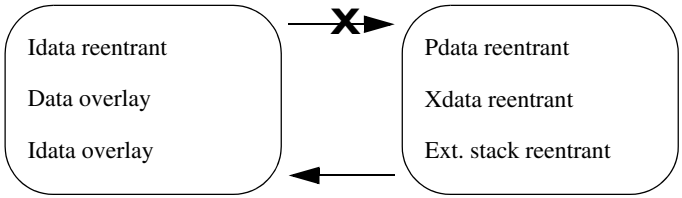
Functions calling other functions

This is how functions can call each other:

Idata default pointer



Xdata/far default pointer



The arrow means that a call is possible. The X means that a call is not possible unless the source code has been explicitly type-declared using function attributes or pragma directives.

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.

See also *Stack considerations*, page 247 and *Setting up stack memory*, page 132.

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 either directly or indirectly—a recursive function—and each invocation can store its own data on the stack.

Potential problems

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

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

Another problem is the risk of running out of stack space. 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 are used.

Stacks used by the 8051 compiler

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 8051 devices this is the idata stack, located in idata memory. On extended 8051 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 data segment used for holding the stack is one of `ISTACK`, `PSTACK`, `XSTACK`, or `EXT_STACK`.

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 8051 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 *IAR Assembler User Guide for 8051*.

Dynamic memory on the heap

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

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

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

The compiler supports heaps in more than one memory type for 8051 devices with external memory:

Memory type	Memory attribute	Secgment name	Used by default in data model
xdata	`__xdata`	`XDATA_HEAP`	Large and Generic
far22	`__far22`	`FAR22_HEAP`	Far Generic
far	`__far`	`FAR_HEAP`	Far
huge (requires the DLIB runtime environment)	`__huge`	`HUGE_HEAP`	—

Table 11: Heaps supported in memory types

In DLIB, to use a specific heap, add the memory attribute in front of `malloc`, `free`, `alloc`, and `realloc`, for example `__pdata_malloc`. The default functions will use of the specific heap variants, depending on project settings such as data model.

For information about how to set up the size for heap memory, see *Setting up heap memory*, page 135 .

POTENTIAL PROBLEMS

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

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

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

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

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

ALTERNATIVE MEMORY ALLOCATION FUNCTIONS

Because the internal memory is very limited on the 8051 microcontroller, the compiler only supports a heap located in the external data memory space. To fully support this

heap when you use the 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 *)
void __far22 *__far22_malloc(unsigned int)
void __far22 *__far22_realloc(void __far22 *, unsigned int)
void __far22 *__far22_calloc(unsigned int, unsigned int)
void __far22_free(void __far22 *)
```

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 Far, Far Generic, Generic, and Large 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 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 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 314.

THE VIRTUAL BIT REGISTER

The 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

- Function-related extensions
- Code models and memory attributes for function storage
- Primitives for interrupts, concurrency, and OS-related programming
- Inlining functions

Function-related extensions

In addition to supporting Standard C, the 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
- Control function inlining
- Facilitate function optimization
- Access hardware features.

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

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

For more information about optimizations, see *Efficient coding for embedded applications*, page 257. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

Code models and memory attributes for function storage

Use *code models* to specify in which part of memory the compiler should place functions by default. Technically, the code models control the following:

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

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.

These 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 107. Functions can be explicitly placed in the root bank by using the `__near_func` memory attribute, see *__near_func*, page 354.
Banked Extended2	Extended2	3 bytes	Supports banked function calls, see *Code models for banked systems*, page 107.
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 12: Code models

See the *IDE Project Management and Building Guide for 8051* for information about specifying a code model in the IDE.

Use the `--code_model` option to specify the code model for your project; see *--code_model*, page 293.

Pointers with function memory attributes have restrictions in implicit and explicit casts between pointers and between pointers and integer types. For information about the restrictions, see *Casting*, page 333.

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

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.

USING FUNCTION MEMORY ATTRIBUTES

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

Function memory attribute	Address range	Pointer size	Default in code model	Description
`__banked_func`	`0-0xFFFFFF`	2 bytes	Banked	The function is callable using banked 24-bit calls.
`__banked_func_ext2`	`0-0xFFFFFF`	3 bytes	Banked extended2	The function is callable using banked 24-bit calls.
`__far_func`	`0-0xFFFFFF`	3 bytes	—	The function is callable using true 24-bit calls.
`__near_func`	`0-0xFFFF`	2 bytes	—	The function is callable from any segment,

Table 13: Function memory attributes

Pointers with function memory attributes have restrictions in implicit and explicit casts between pointers and between pointers and integer types. For information about the restrictions, see *Casting*, page 333.

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

Primitives for interrupts, concurrency, and OS-related programming

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

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

INTERRUPT FUNCTIONS

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

Interrupt service routines

In general, when an interrupt occurs in the code, the microcontroller immediately stops executing the code it runs, and starts executing an interrupt routine instead. It is

important that the environment of the interrupted function is restored after the interrupt is handled (this includes the values of processor registers and the processor status register). This makes it possible to continue the execution of the original code after the code that handled the interrupt was executed.

The 8051 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 8051 microcontroller documentation from the chip manufacturer. If you want to handle several different interrupts using the same interrupt routine, you can specify several interrupt vectors.

Interrupt vectors and the interrupt vector table

The interrupt vector is the offset into the interrupt vector table.

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.

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

Defining an interrupt function—an example

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

```
#pragma vector = IE0_int  /* Symbol defined in I/O header file */
__interrupt void MyInterruptRoutine(void)
{
  /* Do something */
}
```

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

Interrupt and C++ member functions

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

Register banks

The basic 8051 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 more information, see *__monitor*, page 354.

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

Example of implementing a semaphore in C

In the following example, a binary semaphore—that is, a mutex—is implemented using one static variable and two monitor functions. A monitor function works like a critical

region, that is no interrupt can occur and the process itself cannot be swapped out. A semaphore can be locked by one process, and is used for preventing processes from simultaneously using resources that can only be used by one process at a time, for example a USART. The `__monitor` keyword assures that the lock operation is atomic; in other words it cannot be interrupted.

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

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

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

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

    return 0;
  }
}

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

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

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

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

/* An example of using the semaphore. */

void MyProgram(void)
{
  GetLock();

  /* Do something here. */

  ReleaseLock();
}
```

Example of implementing a semaphore in C++

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

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

```
#include <intrinsics.h>

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

    // Disable all interrupts.
    __disable_interrupt();
  }

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

```
private:
  __istate_t mState;
};

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

    // Enter a critical block.
    {
      Mutex m;

      // Get the tick count safely,
      t = smTickCount;
    }
    // and return it.
    return t;
  }

private:
  static volatile long smTickCount;
};

volatile long Tick::smTickCount = 0;

extern void DoStuff();

void MyMain()
{
  static long nextStop = 100;

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

Inlining functions

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

optimization, which is performed at optimization level High, normally reduces execution time, but might increase the code size. The resulting code might become more difficult to debug. Whether the inlining actually occurs is subject to the compiler's heuristics.

The compiler heuristically decides which functions to inline. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed. Normally, code size does not increase when optimizing for size.

C VERSUS C++ SEMANTICS

In C++, all definitions of a specific inline function in separate translation units must be exactly the same. If the function is not inlined in one or more of the translation units, then one of the definitions from these translation units will be used as the function implementation.

In C, you must manually select one translation unit that includes the non-inlined version of an inline function. You do this by explicitly declaring the function as `extern` in that translation unit. If you declare the function as `extern` in more than one translation unit, the linker will issue a *multiple definition* error. In addition, in C, inline functions cannot refer to static variables or functions.

For example:

```
// In a header file.
static int sX;
inline void F(void)
{
  //static int sY; // Cannot refer to statics.
  //sX;            // Cannot refer to statics.
}


// In one source file.
// Declare this F as the non-inlined version to use.
extern inline void F();
```

FEATURES CONTROLLING FUNCTION INLINING

There are several mechanisms for controlling function inlining:

- The `inline` keyword advises the compiler that the function defined immediately after the directive should be inlined.

  If you compile your function in C or C++ mode, the keyword will be interpreted according to its definition in Standard C or Standard C++, respectively.

  The main difference in semantics is that in Standard C you cannot (in general) simply supply an inline definition in a header file. You must supply an external definition in

one of the compilation units, by designating the inline definition as being external in that compilation unit.

- `#pragma inline` is similar to the `inline` keyword, but with the difference that the compiler always uses C++ inline semantics.

  By using the `#pragma inline` directive you can also disable the compiler's heuristics to either force inlining or completely disable inlining. For more information, see *inline*, page 370.

- `--use_c++_inline` forces the compiler to use C++ semantics when compiling a Standard C source code file.

- `--no_inline`, `#pragma optimize=no_inline`, and `#pragma inline=never` all disable function inlining. By default, function inlining is enabled at optimization level High.

The compiler can only inline a function if the definition is known. Normally, this is restricted to the current translation unit. However, when the `--mfc` compiler option for multi-file compilation is used, the compiler can inline definitions from all translation units in the multi-file compilation unit. For more information, see *Multi-file compilation units*, page 264.

For more information about the function inlining optimization, see *Function inlining*, page 267.

Banked functions

The following pages contain these topics:

- Introduction to the banking system
- Writing source code for banked memory
- Bank switching
- Downloading to memory
- Debugging banked applications

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

Introduction to the banking system

If you are using an 8051 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 8051 microcontroller is not sufficient to contain it all.

CODE MODELS FOR BANKED SYSTEMS

The IAR C/C++ Compiler for 8051 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 8051 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 8051 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.

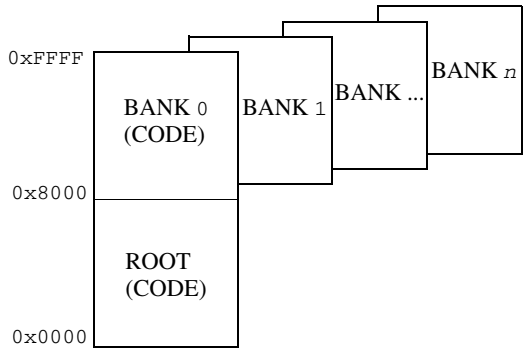
Your hardware must provide these additional physical memory banks, as well as the logic required to decode the high address bits which represent the *bank number*.

Because a basic 8051 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 8051 banked code memory layout:

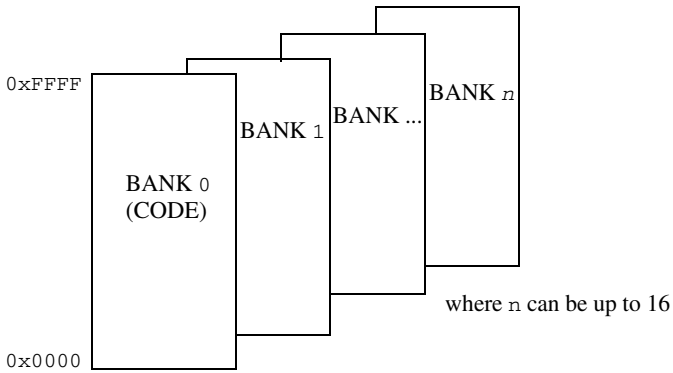


It is practical to place the root area at address 0 and upwards. For information about how to explicitly place functions in the root area, see *Banked versus non-banked function calls*, page 111.

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 8051 microcontroller.

The following illustration shows an example of a 8051 banked code memory layout:



Bank 0 holds the runtime environment code.

SETTING UP THE COMPILER FOR BANKED MODE

To specify the banked code model in the IDE, choose **Project>Options>General Options>Target>Code model**. Choose either **Banked** or, if you are using the **Extended2** core option, **Banked extended2**.

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

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

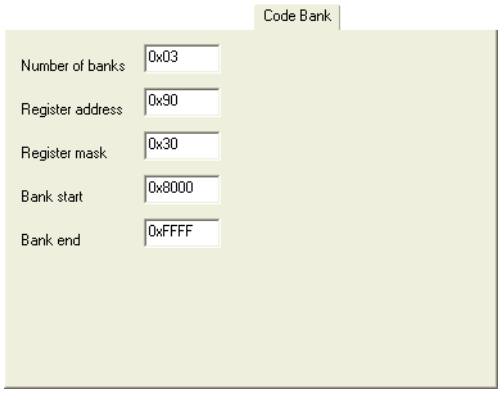
Note: The Banked code model is available when using the core Plain (classic 8051) and the Banked extended 2 code model is available when using the core Extended2 (for the Mentor Graphics M8051EW core).

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 depends 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**.



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 bit mask 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 must adapt them according to your needs.

   As a result, you might 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 source 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 *Controlling data and function placement in memory*, page 259.

For more information about segments, see the chapters *Linking overview* and *Linking your application*.

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 *The memory layout for the banked extended2 code model*, page 108, 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 197. To generate an example of a banked function call, use the technique described in the section *Calling assembler routines from C*, page 194.

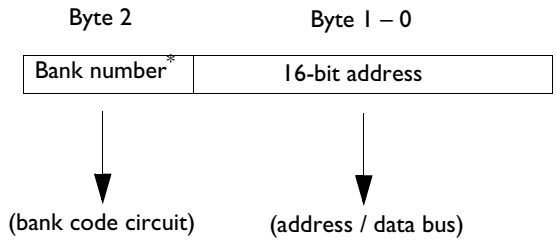
If you are writing banked functions using assembler language, you must also consider the calling convention.

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 current bank, whereas the lower four bits hold the number of the next bank.

For further details of pointer representation, see *Pointer types*, page 330.

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.

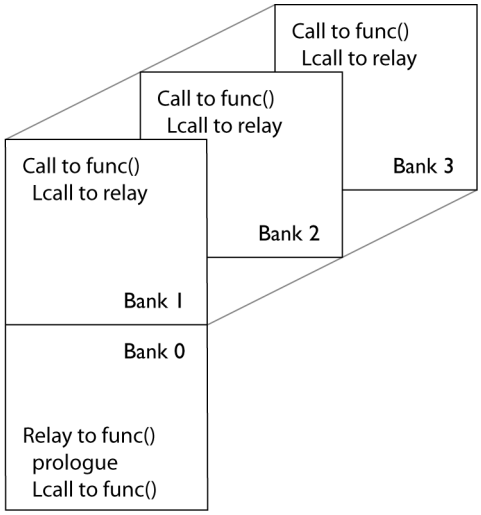
BANK SWITCHING IN THE BANKED CODE MODEL

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.

This figure illustrates the actions:



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 *Accessing banked code*—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.

Linking overview

- Linking—an overview
- Segments and memory
- The linking process in detail
- Placing code and data—the linker configuration file
- Initialization at system startup

Linking—an overview

The IAR XLINK Linker is a powerful, flexible software tool for use in the development of embedded applications. It is equally well suited for linking small, single-file, absolute assembler programs as it is for linking large, relocatable, multi-module, C/C++, or mixed C/C++ and assembler programs.

The linker combines one or more relocatable object files—produced by the IAR Systems compiler or assembler—with required parts of object libraries to produce an executable image containing machine code for the microcontroller you are using. XLINK can generate more than 30 industry-standard loader formats, in addition to the proprietary format UBROF which is used by the C-SPY debugger.

The linker will automatically load only those library modules that are actually needed by the application you are linking. Further, the linker eliminates segment parts that are not required. During linking, the linker performs a full C-level type checking across all modules.

The linker uses a *configuration file* where you can specify separate locations for code and data areas of your target system memory map.

The final output produced by the linker is an absolute, target-executable object file that can be downloaded to the microcontroller, to C-SPY, or to a compatible hardware debugging probe. Optionally, the output file can contain debug information depending on the output format you choose.

To handle libraries, the library tools XAR and XLIB can be used.

Segments and memory

In an embedded system, there might be 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 container for pieces of data or code that should be mapped to a location in physical memory. Each segment consists of one or more *segment parts*. Normally, each function or variable with static storage duration is placed in its own segment part. A segment part is the smallest linkable unit, which allows the linker to include only those segment parts that are referred to. A segment can be placed either in RAM or in ROM. Segments that are placed in RAM generally do not have any content, they only occupy space.

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

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

At compile time, the compiler assigns code and data to the various segments. The IAR XLINK Linker is responsible for placing the segments in the physical memory range, in accordance with the rules specified in the linker configuration file. Ready-made linker configuration files are provided, but, if necessary, they can be 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.

Segment memory type

Each segment always has an associated segment memory type. In some cases, an individual segment has the same name as the segment memory type it belongs to, for example `CODE`. Make sure not to confuse the segment name with the segment memory type in those cases.

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

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

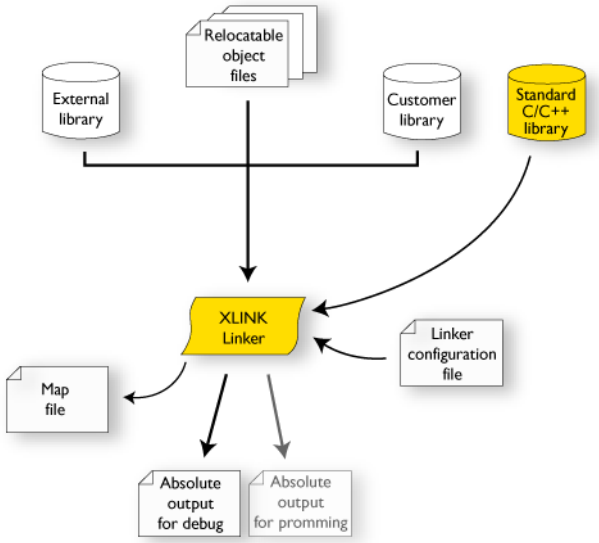
The linking process in detail

The relocatable modules in object files and libraries, produced by the IAR compiler and assembler, cannot be executed as is. To make an application executable, the object files must be *linked*.

The IAR XLINK Linker is used for the link process. It normally performs the following procedure (note that some of the steps can be turned off by command line options or by directives in the linker configuration file):

- Determines which modules to include in the application. Program modules are always included. Library modules are only included if they provide a definition for a global symbol that is referenced from an included module. If the object files containing library modules contain multiple definitions of variables or functions, only the first definition will be included. This means that the linking order of the object files is important.
- Determines which segment parts from the included modules to include in the application. Only those segments that are actually needed by the application are included. There are several ways to determine of which segment parts that are needed, for example, the `__root` object attribute, the `#pragma required` directive, and the `-g` linker option.
- Divides each segment that will be initialized by copying into two segments, one for the ROM part and one for the RAM part. The RAM part contains the label and the ROM part the actual bytes. The bytes are conceptually linked as residing in RAM.
- Determines where to place each segment according to the segment placement directives in the *linker configuration file*.
- Produces an absolute file that contains the executable image and any debug information. The contents of each needed segment in the relocatable input files is calculated using the relocation information supplied in its file and the addresses determined when placing segments. This process can result in one or more range errors if some of the requirements for a particular segment are not met, for instance if placement resulted in the destination address for a `PC`-relative jump instruction being out of range for that instruction.
- Optionally, produces a map file that lists the result of the segment placement, the address of each global symbol, and finally, a summary of memory usage for each module.

This illustration shows the linking process:



During the linking, XLINK might produce error messages and optionally a map file. In the map file you can see the result of the actual linking and is useful for understanding why an application was linked the way it was, for example, why a segment part was included. If a segment part is not included although you expect it to be, the reason is *always* that the segment part was not referenced to from an included part.

Note: To inspect the actual content of the object files, use XLIB. See the *IAR Linker and Library Tools Reference Guide*.

Placing code and data—the linker configuration file

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

In particular, the linker configuration file specifies:

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

- The maximum heap size (only for the IAR DLIB runtime environment).

The file consists of a sequence of linker commands. This means that the linking process will be governed by all commands in sequence.

THE CONTENTS OF THE LINKER CONFIGURATION FILE

Among other things, the linker configuration 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`. Symbols defined using `-D` can also be accessed from your application.
- The placement directives (the largest part of the linker configuration file). Segments can be placed using the `-Z` and `-P` options. The former will place the segment parts in the order they are found, 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 initialized or set to zero at program startup, the `-Z` option is used.

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

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

For more information about the linker configuration file and how to customize it, see *Linking considerations*, page 127.

See also the *IAR Linker and Library Tools Reference Guide*.

Initialization at system startup

In Standard C, all static variables—variables that are allocated at a fixed memory address—must be initialized by the runtime system to a known value at application startup. Static variables can be divided into these categories:

- Variables that are initialized to a non-zero value
- Variables that are initialized to zero
- Variables that are located by use of the `@` operator or the `#pragma location` directive
- Variables that are declared as `const` and therefore can be stored in ROM

- Variables defined with the `__no_init` keyword, meaning that they should not be initialized at all.

STATIC DATA MEMORY SEGMENTS

The compiler generates a specific type of segment for each type of variable initialization.

The names of the segments consist of two parts—the *segment group name* and a *suffix*—for instance, `PDATA_Z`. There is a segment group for each memory type, where each segment in the group holds different categories of declared data. The names of the segment groups are derived from the memory type and the corresponding keyword, for example `PDATA` and `__pdata`. This 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
Far22	FAR22	XDATA	0–3FFFFF
Huge	HUGE	XDATA	0–FFFFFF
Code	CODE	CODE	0-FFFF
Far code	FAR_CODE	CODE	0-FFFFFF
Far22 code	FAR22_CODE	CODE	0-3FFFFF
Huge code	HUGE_CODE	CODE	0-FFFFFF
Xdata ROM	XDATA_ROM	CONST	0–FFFF
Far ROM	FAR_ROM	CONST	0–FFFFFF
Far22 ROM	FAR22_ROM	CONST	0–3FFFFF
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 information about segment memory types, see *Segment memory type*, page 120.

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

Categories of declared data	Segment group	Suffix
Zero-initialized non-located data	BDATA/DATA/IDATA/XDATA/IXDATA/FAR/FAR22/HUGE	Z
Non-zero initialized non-located data	BDATA/DATA/IDATA/XDATA/IXDATA/FAR/FAR22/HUGE	I
Initializers for the above	BDATA/DATA/IDATA/XDATA/IXDATA/FAR/FAR22/HUGE	ID
Initialized located constants	CODE/XDATA_ROM/FAR_CODE/FAR22_CODE/FAR_ROM/FAR22_ROM/HUGE_CODE/HUGE_ROM	AC
Initialized non-located constants	CODE/XDATA_ROM/FAR_CODE/FAR22_CODE/FAR_ROM/FAR22_ROM/HUGE_CODE/HUGE_ROM	C
Non-initialized located data	SFR/CODE/FAR_CODE/FAR22_CODE/HUGE_CODE/BDATA/DATA/IDATA/XDATA/IXDATA/FAR/FAR22/HUGE	AN
Non-initialized non-located data	BIT/CODE/FAR_CODE/FAR22_CODE/HUGE_CODE/BDATA/DATA/IDATA/XDATA/IXDATA/FAR/FAR22/HUGE	N

Table 15: Segment name suffixes

For more information about each segment, see the chapter *Segment reference*.

THE INITIALIZATION PROCESS

Initialization of data is handled by the system startup code. If you add more segments, you must update the system startup code accordingly.

To configure the initialization of variables, you must consider these issues:

- Segments that should be zero-initialized should only be placed in RAM.
- Segments that should be initialized, except for zero-initialized segments:

  The system startup code 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 very 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.
- Segments that contain constants do not need to be initialized; they should only be placed in flash/ROM
- Segments holding `__no_init` declared variables should not be initialized.
- Finally, global C++ object constructors are called.

For more information about and examples of how to configure the initialization, see *Linking considerations*, page 127.

Linking your application

- Linking considerations
- Checking module consistency

Linking considerations

When you set up your project in the IAR Embedded Workbench IDE, a default linker configuration file is automatically used based on your project settings and you can simply link your application. For the majority of all projects it is sufficient to configure the vital parameters that you find in **Project>Options>Linker>Config**.

When you build from the command line, you can use a ready-made linker command file provided with your product package.

The `config` directory contains ready-made linker configuration files for all supported devices (filename extension `xcl`). The files contain the information required by XLINK, and are ready to be used as is. The only change, if any, you will normally have to make to the supplied configuration file is to customize it so it fits the target system memory map. If, for example, your application uses additional external RAM, you must also add details about the external RAM memory area.

The IDE uses device-specific linker configuration files named after this pattern: `lnk51ew_`*`device`*`.xcl`, where *`device`* is the name of the device.

There is also a basic set of linker configuration files to be used on the command line:

- `lnk51.xcl` is a basic, default linker configuration file
- `lnk51b.xcl` supports the banked code model
- `lnk51e.xcl` supports the extended1 core
- `lnk51e2.xcl` supports the extended2 core
- `lnk51eb.xcl` supports the extended1 core and the banked code model
- `lnk51o.xcl` supports the overlay calling conventions.

These files include a device-specific linker configuration file for a generic device. Normally, you do not need to customize this included file.

To edit a linker configuration file, use the editor in the IDE, or any other suitable editor. Do not change the original template file. We recommend that you make a copy in the working directory, and modify the copy instead.

If you find that the default linker configuration file does not meet your requirements, you might want to consider:

- Placing segments
- Placing data
- Setting up stack memory
- Setting up heap memory
- Placing code
- Keeping modules
- Keeping symbols and segments
- Application startup
- Interaction between XLINK and your application
- Producing other output formats than UBROF

PLACING SEGMENTS

The placement of segments in memory is performed by the IAR XLINK Linker.

This section describes the most common linker commands and how to customize the linker configuration file to suit the memory layout of your target system. In demonstrating the methods, fictitious examples are used.

In demonstrating the methods, fictitious examples are used based on this memory layout:

- There is 1 Mbyte addressable memory.
- There is ROM memory in the address ranges `0x0000-0x1FFF`, `0x3000-0x4FFF`, and `0x10000-0x1FFFF`.
- There is RAM memory in the address ranges `0x8000-0xAFFF`, `0xD000-0xFFFF`, and `0x20000-0x27FFF`.
- There are two addressing modes for data, one for pdata memory and one for xdata memory.
- There is one stack and one heap.
- There are two addressing modes for code, one for near_func memory and one for far_func memory.

Note: Even though you have a different memory map, for example if you have additional memory spaces (EEPROM) and additional segments, you can still use the methods described in the following examples.

The ROM can be used for storing `CONST` and `CODE` segment memory types. The RAM memory can contain segments of `DATA` type. The main purpose of customizing the linker

configuration file is to verify that your application code and data do not cross the memory range boundaries, which would lead to application failure.

For the result of each placement directive after linking, inspect the segment map in the list file (created by using the command line option `-x`).

General hints for placing segments

When you consider where in memory you should place your segments, it is typically a good idea to start placing large segments first, then placing small segments.

In addition, you should consider these aspects:

- Start placing the segments that must be placed on a specific address. This is, for example, often the case with the segment holding the reset vector.
- Then consider placing segments that hold content that requires continuous memory addresses, for example the segments for the stack and heap.
- When placing code and data segments for different addressing modes, make sure to place the segments in size order (the smallest memory type first).

Note: Before the linker places any segments in memory, the linker will first place the absolute segments.

Using the -Z command for sequential placement

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

The following illustrates how to use the `-Z` command to place the segment `MYSEGMENTA` followed by the segment `MYSEGMENTB` in `CONST` memory (that is, ROM) in the memory range `0x0000-0x1FFF`.

```
-Z(CONST)MYSEGMENTA,MYSEGMENTB=0000-1FFF
```

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

```
-Z(CONST)MYSEGMENTA=0000-1FFF
-Z(CODE)MYCODE
```

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

```
-Z(CONST)MYSMALLSEGMENT=0000-01FF
-Z(CONST)MYLARGESEGMENT=0000-1FFF
```

Normally, when you place segments sequentially with the `-Z` option, each segment is placed entirely 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 in the available memory.

Using the -P command for packed placement

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

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

```
-P(DATA)MYDATA=8000-AFFF,D000-FFFF
```

If your application has an additional RAM area in the memory range `0x6000-0x67FF`, you can simply add that to the original definition:

```
-P(DATA)MYDATA=8000-AFFF,D000-FFFF,6000-67FF
```

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

Note: Copy initialization segments—*`BASENAME`*`_I` and *`BASENAME`*`_ID`—and dynamic initialization segments 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 109.

PLACING DATA

Static memory is memory that contains variables that are global or declared static.

Placing static memory data segments

Depending on their memory attribute, static data is placed in specific segments. For information about the segments used by the compiler, see *Static data memory segments*, page 124.

For example, these commands can be used to place the static data segments:

```
/* First, the segments to be placed in ROM are defined. */
-Z(CONST)PDATA_C=0000-1FFF,3000-4FFF
-Z(CONST)XDATA_C=0000-1FFF,3000-4FFF,10000-1FFFF
-Z(CONST)PDATA_ID,XDATA_ID=010000-1FFFF

/* Then, the RAM data segments are placed in memory. */
-Z(DATA)PDATA_I,DATA16_Z,PDATA_N=8000-AFFF
-Z(DATA)XDATA_I,XDATA_Z,XDATA_N=20000-27FFF
```

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

Placing located data

A variable that is explicitly placed at an address, for example by using the `#pragma location` directive or the `@` operator, is placed in for example either the `PDATA_AC` or the `PDATA_AN` segment. The former is used for constant-initialized data, and the latter for items declared as `__no_init`. The individual segment part of the segment knows its location in the memory space, and it does not have to be specified in the linker configuration file.

Placing user-defined segments

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

SETTING UP STACK MEMORY

In this example, the data segment for holding the stack is called `CSTACK`. The system startup code initializes the stack pointer to point to the start or the end of the stack segment, depending on the stack type.

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

For more information about stack memory, see *Stacks used by the 8051 compiler*, page 90, and *Stack considerations*, page 247.

Stack size allocation in the IDE

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

Add the required stack size in the dedicated text boxes.

Stack size allocation from the command line

The sizes of the stack segments are defined in the linker configuration file.

The linker configuration file sets up a constant, representing the size of the stack, at the beginning of the file. Specify the appropriate size for your application:

`-D_`*`stackname`*`_SIZE=`*`size`*

where *`stackname`* can be one of `IDATA_STACK`, `XDATA_STACK`, `PDATA_STACK`, or `EXTENDED_STACK`.

Note that the size is written hexadecimally, but not necessarily with the `0x` notation.

In many linker configuration files provided with IAR Embedded Workbench, this line is prefixed with the comment character `//` because the IDE controls the stack size allocation. To make the directive take effect, remove the comment character.

Placing the stack segment

Further down in the linker configuration file, the actual stack segments are 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.
- The = allocates the stack segment at the start of the memory area.

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.

SETTING UP HEAP MEMORY

The heap contains dynamic data allocated by the C function `malloc` (or a corresponding function) or the C++ operator `new`. Only 8051 devices with external data memory can have a heap.

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

- The linker segments used for the heap, see *Dynamic memory on the heap*, page 90
- The steps for allocating the heap size, which differs depending on which build interface you are using
- The steps involved for placing the heap segments in memory.

See also *Heap considerations*, page 247.

In this example, the data segment for holding the heap is called `HEAP`.

Heap size allocation in the IDE

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

Add the required heap size in the dedicated text box.

Heap size allocation from the command line

The size of the `HEAP` segment is defined in the linker configuration file.

The linker configuration file sets up a constant, representing the size of the heap, at the beginning of the file. Specify the appropriate size for your application, in this example 1024 bytes:

```
-D_HEAP_SIZE=400     /* 1024 bytes for heap memory */
```

Note that the size is written hexadecimally, but not necessarily with the `0x` notation.

In many linker configuration files provided with IAR Embedded Workbench, these lines are prefixed with the comment character `//` because the IDE controls the heap size allocation. To make the directive take effect, remove the comment character.

If you use a heap, you should allocate at least 512 bytes for it, to make it work properly.

Placing the heap segment

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

```
-Z(DATA)HEAP+_HEAP_SIZE=8000-AFFF
```

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

PLACING CODE

This section contains descriptions of the segments used for storing code and the interrupt vector table. For information about all segments, see *Summary of segments*, page 405.

Startup code

In this example, the segment `CSTART` contains code used during system startup and termination, see *System startup and termination*, page 164. The system startup code should be placed at the location where the chip starts executing code after a reset. The segment parts must also be placed into one continuous memory space, which means that the `-P` segment directive cannot be used.

This line will place the `CSTART` segment at the address `0x1100`:

```
-Z(CODE)CSTART=1100
```

Normal code

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

For information about segments holding normal code, see the chapter *Segment reference*.

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 configuration 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 vector table contains pointers to interrupt routines, including the reset routine. In this example, the table is placed in the segment `INTVEC`. The linker directive would then look like this:

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

For more information about the interrupt vectors, see *Interrupt vectors and the interrupt vector table*, page 98.

C++ dynamic initialization

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

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

For example:

```
-Z(CONST)DIFUNCT=0000-1FFF,3000-4FFF
```

`DIFUNCT` must be placed using `-Z`. For more information, see *DIFUNCT*, page 416.

KEEPING MODULES

If a module is linked as a program module, it is always kept. That is, it will contribute to the linked application. However, if a module is linked as a library module, it is included only if it is symbolically referred to from other parts of the application that have been included. This is true, even if the library module contains a root symbol. To assure that such a library module is always included, use `-A` to make all modules in the file be treaded as if they were program modules:

```
-A file.r51
```

Use `-C` to makes all modules in the file be treated as if they were library modules:

```
-C file.r51
```

KEEPING SYMBOLS AND SEGMENTS

By default, XLINK removes any segments, segment parts, and global symbols that are not needed by the application. To retain a symbol that does not appear to be needed—or actually, the segment part it is defined in—you can either use the `__root` attribute on the symbol in your C/C++ source code or `ROOT` in your assembler source code, or use the XLINK option `-g`.

For information about included and excluded symbols and segment parts, inspect the map file (created by using the XLINK option `-xm`).

For more information about the linking procedure for keeping symbols and sections, see *The linking process in detail*, page 121.

APPLICATION STARTUP

By default, the point where the application starts execution is defined by the `__program_start` label, which is defined to point at the start of code. The label is also communicated via the debugger information to any debugger.

To change the start point of the application to another label, use the XLINK option `-s`.

INTERACTION BETWEEN XLINK AND YOUR APPLICATION

Use the XLINK option `-D` to define symbols that can be used for controlling your application. You can also use symbols to represent the start and end of a continuous memory area that is defined in the linker configuration file.

To change a reference to one symbol to another symbol, use the XLINK command line option `-e`. This is useful, for example, to redirect a reference from a non-implemented function to a stub function, or to choose one of several different implementations of a certain function.

For information about the addresses and sizes of all global (statically linked) symbols, inspect the entry list in the map file (the XLINK option `-xm`).

PRODUCING OTHER OUTPUT FORMATS THAN UBROF

XLINK can generate more than 30 industry-standard loader formats, in addition to the proprietary format UBROF which is used by the C-SPY debugger. For a complete list, see the *IAR Linker and Library Tools Reference Guide*. To specify a different output format than the default, use the XLINK option `-F`. For example:

```
-F intel-standard
```

Note that it can be useful to use the XLINK `-O` option to produce two output files, one for debugging and one for burning to ROM/flash.

Note also that if you choose to enable debug support using the `-r` option for certain low-level I/O functionality for mechanisms like file operations on the host computer etc, such debug support might have an impact on the performance and responsiveness of your application. In this case, the debug build will differ from your release build due to the debug modules included.

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

Code or data that is placed in a relocatable segment will have its absolute address resolved at link time. Note that it is not known until link time whether all segments will fit in the reserved memory ranges. If the contents of a segment do not fit in the address range defined in the linker configuration 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 more 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 address order
- A module map which lists all segments, local symbols, and entries (public symbols) for every module in the program. All symbols not included in the output can also be listed
- A module summary which lists the contribution (in bytes) from each module
- A symbol list which contains every entry (global symbol) in every module.

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

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

For more information about the listing options and the linker listing, see the *IAR Linker and Library Tools Reference Guide*, and the *IDE Project Management and Building Guide for 8051*.

MANAGING MULTIPLE MEMORY SPACES

Output formats that do not support more than one memory space—like `MOTOROLA` and `INTEL-HEX`—might require up to one output file per memory space. This causes no problems if you are only producing output to one memory space (flash), 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 this 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 memory, make a copy of the linker configuration file for the derivative and memory model you are using, and put it in the project directory. Add this line to 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 chose does not support more than one memory space). Use the XLINK option `-O` for this purpose.

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

```
-Omotorola,(XDATA)=external_rom.a51
-Omotorola,(CODE)=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.

For more information, see the *IAR Linker and Library Tools Reference Guide*.

Checking module consistency

This section introduces the concept of runtime model attributes, a mechanism used by the tools provided by IAR Systems to ensure that modules that are linked into an application are compatible, in other words, are built using compatible settings. The tools use a set of predefined runtime model attributes. You can use these predefined attributes or define your own to ensure that incompatible modules are not used together.

For example, in the compiler, it is possible to write a module that only supports separate registers. If you write a routine that supports separate `DPTR` registers, you can check that the routine is not used in an application built for shadowed `DPTR`.

RUNTIME MODEL ATTRIBUTES

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

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

Example

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

Object file	Color	Taste
`file1`	`blue`	not defined
`file2`	`red`	not defined
`file3`	`red`	`*`

Table 17: Example of runtime model attributes

Object file	Color	Taste
file4	red	spicy
file5	red	lean

Table 17: Example of runtime model attributes (Continued)

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

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

USING RUNTIME MODEL ATTRIBUTES

To ensure module consistency with other object files, use the `#pragma rtmodel` directive to specify runtime model attributes in your C/C++ source code. For example, if you have a UART that can run in two modes, you can specify a runtime model attribute, for example `uart`. For each mode, specify a value, for example `mode1` and `mode2`. Declare this in each module that assumes that the UART is in a particular mode. This is how it could look like in one of the modules:

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

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

```
        rtmodel "uart", "mode1"
```

Note that key names that start with two underscores are reserved by the compiler. For more information about the syntax, see *rtmodel*, page 377 and the *IAR Assembler User Guide for 8051*, respectively.

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 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`, `far_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`, `data_rom`, 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.
`__register_banks`	`*`, `0`, or `x`	Corresponds to the compiler's use of register banks. If register banks are available to the compiler, the value is `*`. If register banks have been disabled, the value is `0`. If a function is using a register bank explicitly, the value is `x`.

Table 18: Runtime model attributes

Runtime model attribute	Value	Description
`__rt_version`	*n*	This runtime key is always present in all modules generated by the compiler. If a major change in the runtime characteristics occurs, the value of this key changes.

Table 18: 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, see the chapter *Assembler directives* in the *IAR Assembler User Guide for 8051.*

Example

For an example of using the runtime model attribute `__rt_version` for checking the module consistency as regards the used calling convention, see *Hints for using the Normal calling convention*, page 198.

The DLIB runtime environment

- Introduction to the runtime environment
- Setting up the runtime environment
- Additional information on the runtime environment

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 more information about CLIB, see the chapter *The CLIB runtime environment*.

Introduction to the runtime environment

A *runtime environment* is the environment in which your application executes.

This section contains information about:

RUNTIME ENVIRONMENT FUNCTIONALITY

The *DLIB runtime environment* supports Standard C and C++ and consists of:

- The *C/C++ standard library*, both its interface (provided in the system header files) and its implementation.
- Startup and exit code.
- Low-level I/O interface for managing input and output (I/O).
- Special compiler support, for instance functions for switch handling or integer arithmetics.
- Support for hardware features:
  - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for interrupt mask handling
  - Peripheral unit registers and interrupt definitions in include files

Runtime environment functions are provided in one or more *runtime libraries*.

The runtime library is delivered both as prebuilt libraries and (depending on your product package) as source files. The prebuilt libraries are available in different *configurations* to meet various needs, see *Runtime library configurations*, page 155. You can find the libraries in the product subdirectories `8051\lib` and `8051\src\lib`, respectively.

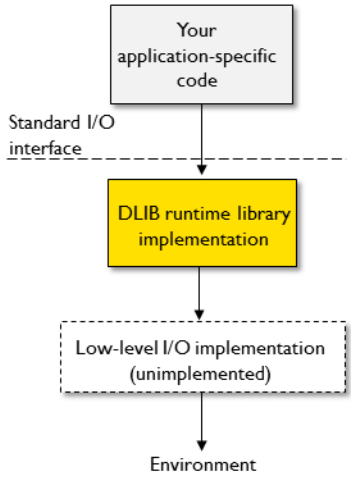
For more information about the library, see the chapter *C/C++ standard library functions*.

BRIEFLY ABOUT INPUT AND OUTPUT (I/O)

Every application must communicate with its environment. The application might for example display information on an LCD, read a value from a sensor, get the current date from the operating system, etc. Typically, your application performs I/O via the C/C++ standard library or some third-party library.

There are many functions in the C/C++ standard library that deal with I/O, including functions for: standard character streams, file system access, time and date, miscellaneous system actions, and termination and assert, This set of functions is referred to as the *standard I/O interface*.

On a desktop computer or a server, the operating system is expected to provide I/O functionality to the application via the standard I/O interface in the runtime environment. However, in an embedded system, the runtime library cannot assume that such functionality is present, or even that there is an operating system at all. Therefore, the low-level part of the standard I/O interface is not completely implemented by default:



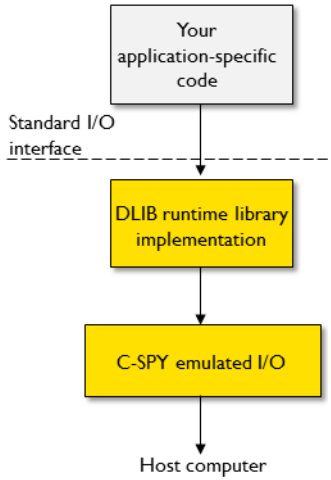
To make the standard I/O interface work, you can:

- Let the C-SPY debugger emulate I/O operations on the host computer, see *Briefly about C-SPY emulated I/O*, page 147
- *Retarget* the standard I/O interface to your target system by providing a suitable implementation of the interface, see *Briefly about retargeting*, page 148.

It is possible to mix these two approaches. You can, for example, let debug printouts and asserts be emulated by the C-SPY debugger, but implement your own file system. The debug printouts and asserts are useful during debugging, but no longer needed when running the application stand-alone (not connected to the C-SPY debugger).

BRIEFLY ABOUT C-SPY EMULATED I/O

C-SPY emulated I/O is a mechanism which lets the runtime environment interact with the C-SPY debugger to emulate I/O actions on the host computer:



For example, when C-SPY emulated I/O is enabled:

- Standard character streams are directed to the C-SPY **Terminal I/O** window
- File system operations are performed on the host computer
- Time and date functions return the time and date of the host computer
- Termination and failed asserts break execution and notify the C-SPY debugger.

This behavior can be valuable during the early development of an application, for example in an application that uses file I/O before any flash file system I/O drivers are implemented, or if you need to debug constructions in your application that use `stdin` and `stdout` without the actual hardware device for input and output being available.

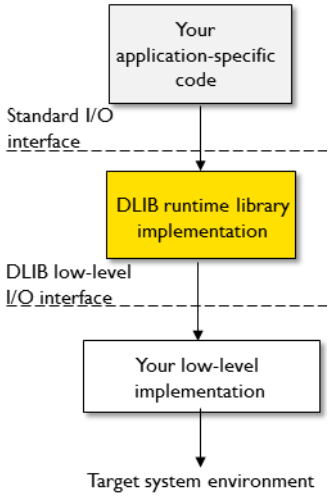
See *Setting up your runtime environment*, page 149 and *The C-SPY emulated I/O mechanism*, page 161.

BRIEFLY ABOUT RETARGETING

Retargeting is the process where you adapt the runtime environment so that your application can execute I/O operations on your target system.

The standard I/O interface is large and complex. To make retargeting easier, the DLIB runtime environment is designed so that it performs all I/O operations through a small set of simple functions, which is referred to as the *DLIB low-level I/O interface*. By default, the functions in the low-level interface lack usable implementations. Some are unimplemented, others have stub implementations that do not perform anything except returning error codes.

To retarget the standard I/O interface, all you have to do is to provide implementations for the functions in the DLIB low-level I/O interface.



For example, if your application calls the functions `printf` and `fputc` in the standard I/O interface, the implementations of those functions both call the low-level function `__write` to output individual characters. To make them work, you just need to provide an implementation of the `__write` function—either by implementing it yourself, or by using a third-party implementation.

For information about how to override library modules with your own implementations, see *Overriding library modules*, page 152. See also *The DLIB low-level I/O interface*, page 168 for information about the functions that are part of the interface.

Setting up the runtime environment

This section contains these tasks:

- *Setting up your runtime environment*, page 149

  A runtime environment with basic project settings to be used during the initial phase of development.
- *Retargeting—Adapting for your target system*, page 151
- *Overriding library modules*, page 152
- *Customizing and building your own runtime library*, page 153

SETTING UP YOUR RUNTIME ENVIRONMENT

You can set up the runtime environment based on some basic project settings. It is also often convenient to let the C-SPY debugger manage things like standard streams, file I/O, and various other system interactions. This basic runtime environment can be used for simulation before you have any target hardware.

To set up the runtime environment:

1. Before you build your project, choose **Project>Options>General Options** to open the **Options** dialog box.
2. On the **Library Configuration** page, verify the following settings:

   - **Library**: choose which *library* and *library configuration* to use. Typically, choose **Tiny DLIB**, **Normal DLIB**, or **Full DLIB**.

     For information about the various library configurations, see *Runtime library configurations*, page 155.

     CLIB or DLIB—for more information about the libraries, see *C/C++ standard library overview*, page 395.
3. On the **Library Options** page, select **Auto** for both **Printf formatter** and **Scanf formatter**. This means that the linker will automatically choose the appropriate formatters based on information from the compiler. For more information about the available formatters and how to choose one manually, see *Formatters for printf*, page 159 and *Formatters for scanf*, page 160, respectively.
4. Choose whether non-static auto variables should be placed on the stack or in a static overlay area.

   The stack is dynamically allocated at runtime, whereas the static overlay area is statically allocated at link time. See *Storage of auto variables and parameters*, page 83.

5 To enable C-SPY emulated I/O, choose **Project>Options>Linker>Output** and select the **Format** option that matches the level of support you need. See *Briefly about C-SPY emulated I/O*, page 147. Choose between:

Linker option in the IDE	Linker command line option	Description
Debug information for C-SPY	`-Fubrof`	Debug support for C-SPY, but without any support for C-SPY emulated I/O.
With runtime control modules	`-r`	The same as `-Fubrof` but also limited support for C-SPY emulated I/O handling program abort, exit, and assertions.
With I/O emulation modules	`-rt`	Full support for C-SPY emulated I/O, which means the same as `-r`, but also support for I/O handling, and accessing files on the host computer during debugging.

Table 19: Debug information and levels of C-SPY emulated I/O

Note: The C-SPY **Terminal I/O** window is not opened automatically; you must open it manually. For more information about this window, see the *C-SPY® Debugging Guide for 8051*.

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

For this reason, a replacement for the `__write` function called `__write_buffered` is included in the runtime 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 in the IDE, choose **Project>Options>Linker>Output** and select the option **Buffered terminal output**.

To enable this function on the command line, add this to the linker command line:

```
-e__write_buffered=__write
```

7 Some math functions are available in different versions: default versions, smaller than the default versions, and larger but more accurate than default versions. Consider which versions you should use.

For more information, see *Math functions*, page 161.

8 When you build your project, a suitable prebuilt library and library configuration file are automatically used based on the project settings you made.

For information about which project settings affect the choice of library file, see *Runtime library configurations*, page 155.

You have now set up a runtime environment that can be used while developing your application source code.

RETARGETING—ADAPTING FOR YOUR TARGET SYSTEM

Before you can run your application on your target system, you must adapt some parts of the runtime environment, typically the system initialization and the DLIB low-level I/O interface functions.

To adapt your runtime environment for your target system:

1 Adapt system initialization.

It is likely that you must adapt the system initialization, for example, your application might need to initialize interrupt handling, I/O handling, watchdog timers, etc. You do this by implementing the routine `__low_level_init`, which is executed before the data segments are initialized. See *System startup and termination*, page 164 and *System initialization*, page 167. Note that you can find device-specific examples on this in the example projects provided in the product installation; see the Information Center.

2 Adapt the runtime library for your target system. To implement such functions, you need a good understanding of the DLIB low-level I/O interface, see *Briefly about retargeting*, page 148.

Typically, you must implement your own functions if your application uses:

- Standard streams for input and output

  If any of these streams are used by your application, for example by the functions `printf` and `scanf`, you must implement your versions of the low-level functions `__read` and `__write`.

  The low-level functions identify I/O streams, such as an open file, with a file handle that is a unique integer. The I/O streams normally associated with `stdin`, `stdout`, and `stderr` have the file handles `0`, `1`, and `2`, respectively. When the handle is `-1`, all streams should be flushed. Streams are defined in `stdio.h`.

- File input and output

  The library contains a large number of powerful functions for file I/O operations, such as `fopen`, `fclose`, `fprintf`, `fputs`, etc. All these functions call a small set of low-level functions, each designed to accomplish one particular task; for example, `__open` opens a file, and `__write` outputs characters. Implement your version of these low-level functions.

- `signal` and `raise`

If the default implementation of these functions does not provide the functionality you need, you can implement your own versions.

- Time and date

  To make the time and date functions work, you must implement the functions `clock`, `__time32` and `__getzone`.

- Assert, see *_ReportAssert*, page 174.
- Environment interaction

  If the default implementation of `system` or `getenv` does not provide the functionality you need, you can implement your own versions.

For more information about the functions, see *The DLIB low-level I/O interface*, page 168.

The library files that you can override with your own versions are located in the `8051\src\lib` directory.

3 When you have implemented your functions of the low-level I/O interface, you must add your version of these functions to your project. For information about this, see *Overriding library modules*, page 152.

Note: If you have implemented a DLIB low-level I/O interface function and added it to a project that you have built with support for C-SPY emulated I/O, your low-level function will be used and not the functions provided with C-SPY emulated I/O. For example, if you implement your own version of `__write`, output to the C-SPY **Terminal I/O** window will not be supported. See *Briefly about C-SPY emulated I/O*, page 147.

4 Before you can execute your application on your target system, you must rebuild your project with a Release build configuration. This means that the linker will not include the C-SPY emulated I/O mechanism and the low-level I/O functions it provides. If your application calls any of the low-level functions of the standard I/O interface, either directly or indirectly, and your project does not contain these, the linker will issue an error for every missing low-level function. Also, note that the `NDEBUG` symbol is defined in a Release build configuration, which means asserts will no longer be generated. For more information, see *_ReportAssert*, page 174.

OVERRIDING LIBRARY MODULES

To override a library function and replace it with your own implementation:

1 Use a template source file—a library source file or another template—and place a copy of it in your project directory.

The library files that you can override with your own versions are located in the `8051\src\lib` directory.

2 Modify the file.

Note: To override the functions in a module, you must provide alternative implementations for all the needed symbols in the overridden module. Otherwise you will get error messages about duplicate definitions.

3 Add the modified file to your project, like any other source file.

Note: If you have implemented a DLIB low-level I/O interface function and added it to a project that you have built with support for C-SPY emulated I/O, your low-level function will be used and not the functions provided with C-SPY emulated I/O. For example, if you implement your own version of `__write`, output to the C-SPY **Terminal I/O** window will not be supported. See *Briefly about C-SPY emulated I/O*, page 147.

You have now finished the process of overriding the library module with your version.

CUSTOMIZING AND BUILDING YOUR OWN RUNTIME LIBRARY

If the prebuilt library configurations do not meet your requirements, you can customize your own library configuration, but that requires that you *rebuild* the library.

Building a customized library is a complex process. Therefore, consider carefully whether it is really necessary. You must build your own runtime library when:

- There is no prebuilt library for the required combination of compiler options or hardware support
- You want to define your own library configuration with support for locale, file descriptors, multibyte characters, etc. This will include or exclude certain parts of the DLIB runtime environment.

In those cases, you must:

- Make sure that you have installed the library source code (`src\lib`). If not already installed, you can install it using the IAR License Manager, see the *Installation and Licensing Guide*.
- Set up a library project
- Make the required library customizations
- Build your customized runtime library
- Finally, make sure your application project will use the customized runtime library.

To set up a library project:

1. In the IDE, choose **Project>Create New Project** and use the library project template which can be used for customizing the runtime environment configuration. There is a library template for the Normal library configuration, see *Runtime library configurations*, page 155

   Note that when you create a new library project from a template, the majority of the files included in the new project are the original installation files. If you are going to modify these files, make copies of them first and replace the original files in the project with these copies.

To customize the library functionality:

1. 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` which you can find in `8051\inc\c`. This read-only file describes the configuration possibilities. Note that you should not modify this file.

   In addition, your custom library has its own *library configuration file* `dl8051`*`libraryname`*`.h`—which you can find in `8051\config\template\project`—and which sets up that specific library with the required library configuration. Customize this file by setting the values of the configuration symbols according to the application requirements. For example, to enable the `ll` qualifier in `printf` format strings, write in your library configuration file:

   ```
   #define _DLIB_PRINTF_LONG_LONG 1
   ```

   For information about configuration symbols that you might want to customize, see:

   - *Configuration symbols for printf and scanf*, page 177
   - *Configuration symbols for file input and output*, page 178
   - *Locale*, page 178
   - *Strtod*, page 180

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

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

To build IAR Embedded Workbench projects from the command line, use the IAR Command Line Build Utility (`iarbuild.exe`). However, no make or batch files for building the library from the command line are provided.For information about the build process and the IAR Command Line Build Utility, see the *IDE Project Management and Building Guide for 8051*.

To use the customized runtime library in your application project:

1. In the IDE, choose **Project>Options>General Options** and click the **Library Configuration** tab.

2 From the **Library** drop-down menu, choose **Custom DLIB**.

3 In the **Library file** text box, locate your library file.

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

Additional information on the runtime environment

This section gives additional information on the runtime environment:

RUNTIME LIBRARY CONFIGURATIONS

The runtime library is provided with different *library configurations*, where each configuration is suitable for different application requirements.

The runtime library configuration is defined in the *library configuration file*. It contains information about what functionality is part of the runtime environment. The less functionality you need in the runtime environment, the smaller the environment becomes.

These predefined library configurations are available:

Library configuration	Description
Normal DLIB (default)	C locale, but no locale interface, no file descriptor support, no multibyte characters in `printf` and `scanf`, and no hexadecimal floating-point numbers in `strtod`.

Table 20: Library configurations

Note: In addition to these predefined library configurations, you can provide your own configuration, see *Customizing and building your own runtime library*, page 153

If you do not specify a library configuration explicitly you will get the default configuration. If you use a prebuilt runtime library, a configuration file that matches the runtime library file will automatically be used. See *Setting up your runtime environment*, page 149.

To override the default library configuration, use one of these methods:

1 Use a prebuilt configuration of your choice—to specify a runtime configuration explicitly:

Choose **Project>Options>General Options>Library Configuration>Library** and change the default setting.

Use the `--dlib_config` compiler option, see *--dlib_config*, page 300.

The prebuilt libraries are based on the default configurations, see *Runtime library configurations*, page 155.

2 If you have built your own customized library, choose **Project>Options>Library Configuration>Library** and choose **Custom** to use your own configuration. For more information, see *Customizing and building your own runtime library*, page 153.

PREBUILT RUNTIME LIBRARIES

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

- Core variant
- Stack location
- Data model
- Code model
- Calling convention
- Constant location
- Number of data pointers
- Data pointer visibility
- Data pointer size
- Data pointer selection method.
- Library configuration—Tiny, Normal, or Full.

To choose a library from the command line:

If you build your application from the command line, make the following settings:

- Specify which library file to use on the XLINK command line, like:
  `dl_libname.r51`
- If you do not specify a library configuration, the default configuration will be used. However, you can specify the library configuration explicitly for the compiler:
  `--dlib_config C:\...\dl_libname.h`

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

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

Library filename syntax

The runtime library names are constructed in this way:
{*lib*}-{*core*}{*stack*}-{*code_mod*}{*data_mod*}{*cc*}{*const_loc*}-{*#dptrs*}{*dptr_vis*}{*dptr_size*}{*dptr_select*}{*lib_config*}.r51

where:

{*lib*}	`dl` for the IAR DLIB runtime environment
{*core*}	Specifies the processor variant: `p1` = the classic 8051 devices `e1` = the extended1 devices `e2` = the extended2 devices
{*stack*}	Specifies the stack location: `i` = idata stack `e` = extended stack
{*code_mod*}	Specifies the code model: `n` = Near `b` = Banked `2` = Banked_extended2 `f` = Far

{*data_mod*}	Specifies the data model: `s` = Small `l` = Large `g` = Generic `j` = Far Generic `f` = Far
{*cc*}	Specifies the calling convention: `d` = data overlay `o` = idata overlay `i` = idata reentrant `p` = pdata reentrant `x` = xdata reentrant `e` = extended stack reentrant
{*const_loc*}	Specifies the location for constants and strings: `d` = data `c` = code
{*#dptrs*}	A number from 1 to 8 that represents the number of data pointers used.
{*dptr_vis*}	Specifies the `DPTR` visibility: `h` = shadowed `e` = separate
{*dptr_size*}	Specifies the size of the data pointer in use: `16` = 16 bits `24` = 24 bits
{*dptr_select*}	Specifies 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`.
{*lib_config*}	Specifies the library configuration: `t` = Tiny `n` = Normal `f` = Full

FORMATTERS FOR PRINTF

The `printf` function uses a formatter called `_Printf`. The full 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. Note that the `wprintf` variants are not affected.

This table summarizes the capabilities of the different formatters:

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

Table 21: Formatters for printf

† `NoMb` means without multibytes.

Note: It is possible to optimize these functions even further, but that requires that you rebuild the library. See *Configuration symbols for printf and scanf*, page 177.

The compiler can automatically detect which formatting capabilities are needed in a direct call to `printf`, if the formatting string is a string literal. This information is passed to the linker, which combines the information from all modules to select a suitable formatter for the application. However, if the formatting string is a variable, or if the call is indirect through a function pointer, the compiler cannot perform the analysis, forcing the linker to select the Full formatter. In this case, you might want to override the automatically selected `printf` formatter.

To override the automatically selected printf formatter in the IDE:

1. Choose **Project>Options>General Options** to open the **Options** dialog box.
2. On the **Library Options** page, select the appropriate formatter.

To override the automatically selected printf formatter from the command line:

1. Add one of these lines in the linker configuration file you are using:

```
-e_PrintfFull=_Printf
-e_PrintfFullNoMb=_Printf
-e_PrintfLarge=_Printf
-e_PrintfLargeNoMb=_Printf
_e_PrintfSmall=_Printf
-e_PrintfSmallNoMb=_Printf
-e_PrintfTiny=_Printf
-e_PrintfTinyNoMb=_Printf
```

FORMATTERS FOR SCANF

In a similar way to the `printf` function, `scanf` uses a common formatter, called `_Scanf`. The full version is quite large, and provides facilities that are not required in many embedded applications. To reduce the memory consumption, two smaller, alternative versions are also provided. Note that the `wscanf` versions are not affected.

This table summarizes the capabilities of the different formatters:

Formatting capabilities	Small/ SmallNoMb†	Large/ LargeNoMb†	Full/ FullNoMb†
Basic specifiers `c`, `d`, `i`, `o`, `p`, `s`, `u`, `X`, `x`, and `%`	Yes	Yes	Yes
Multibyte support	Yes/No	Yes/No	Yes/No
Floating-point specifiers `a`, and `A`	No	No	Yes
Floating-point specifiers `e`, `E`, `f`, `F`, `g`, and `G`	No	No	Yes
Conversion specifier `n`	No	No	Yes
Scan set `[` and `]`	No	Yes	Yes
Assignment suppressing `*`	No	Yes	Yes
`long long` support	No	No	Yes

Table 22: Formatters for scanf

† `NoMb` means without multibytes.

Note: It is possible to optimize these functions even further, but that requires that you rebuild the library. See *Configuration symbols for printf and scanf*, page 177.

The compiler can automatically detect which formatting capabilities are needed in a direct call to `scanf`, if the formatting string is a string literal. This information is passed to the linker, which combines the information from all modules to select a suitable formatter for the application. However, if the formatting string is a variable, or if the call is indirect through a function pointer, the compiler cannot perform the analysis, forcing

the linker to select the full formatter. In this case, you might want to override the automatically selected `scanf` formatter.

To manually specify the scanf formatter in the IDE:

1 Choose **Project>Options>General Options** to open the **Options** dialog box.

2 On the **Library Options** page, select the appropriate formatter.

To manually specify the scanf formatter from the command line:

1 Add one of these lines in the linker configuration file you are using:

```
-e_ScanfFull=_Scanf
-e_ScanfFullNoMb=_Scanf
-e_ScanfLarge=_Scanf
-e_ScanfLargeNoMb=_Scanf
_e_ScanfSmall=_Scanf
_e_ScanfSmallNoMb=_Scanf
```

THE C-SPY EMULATED I/O MECHANISM

The C-SPY emulated I/O mechanism works as follows:

1 The debugger will detect the presence of the function `__DebugBreak`, which will be part of the application if you linked it with the linker option for C-SPY emulated I/O.

2 In this case, the debugger will automatically set a breakpoint at the `__DebugBreak` function.

3 When your application calls a function in the DLIB low-level I/O interface, for example, `open`, the `__DebugBreak` function is called, which will cause the application to stop at the breakpoint and perform the necessary services.

4 The execution will then resume.

See also *Briefly about C-SPY emulated I/O*, page 147.

MATH FUNCTIONS

Some C/C++ standard library math functions are available in different versions:

- The default versions
- Smaller versions (but less accurate)
- More accurate versions (but larger).

Smaller versions

The functions `cos`, `exp`, `log`, `log2`, `log10`, `__iar_Log` (a help function for `log`, `log2`, and `log10`), `pow`, `sin`, `tan`, and `__iar_Sin` (a help function for `sin` and `cos`) exist in additional, smaller versions in the library. They are about 20% smaller and about 20% faster than the default versions. The functions handle INF and NaN values. The drawbacks are that they almost always lose some precision and they do not have the same input range as the default versions.

The names of the functions are constructed like:

`__iar_`*xxx*`_small<f|l>`

where `f` is used for `float` variants, `l` is used for `long double` variants, and no suffix is used for `double` variants.

To specify individual math functions from the command line:

1 Redirect the default function names to these names when linking, using these options:

```
-e__iar_sin_small=sin
-e__iar_cos_small=cos
-e__iar_tan_small=tan
-e__iar_log_small=log
-e__iar_log2_small=log2
-e__iar_log10_small=log10
-e__iar_exp_small=exp
-e__iar_pow_small=pow
-e__iar_Sin_small=__iar_Sin
-e__iar_Log_small=__iar_Log

-e__iar_sin_smallf=sinf
-e__iar_cos_smallf=cosf
-e__iar_tan_smallf=tanf
-e__iar_log_smallf=logf
-e__iar_log2_smallf=log2f
-e__iar_log10_smallf=log10f
-e__iar_exp_smallf=expf
-e__iar_pow_smallf=powf
-e__iar_Sin_smallf=__iar_Sinf
-e__iar_Log_smallf=__iar_Logf
```

```
-e__iar_sin_smalll=sinl
-e__iar_cos_smalll=cosl
-e__iar_tan_smalll=tanl
-e__iar_log_smalll=logl
-e__iar_log2_smalll=log2l
-e__iar_log10_smalll=log10l
-e__iar_exp_smalll=expl
-e__iar_pow_smalll=powl
-e__iar_Sin_smalll=__iar_Sinl
-e__iar_Log_smalll=__iar_Logl
```

Note that if you want to redirect any of the functions `sin`, `cos`, or `__iar_Sin`, you must redirect all three functions.

Note that if you want to redirect any of the functions `log`, `log2`, `log10`, or `__iar_Log`, you must redirect all four functions.

More accurate versions

The functions `cos`, `pow`, `sin`, and `tan`, and the help functions `__iar_Sin` and `__iar_Pow` exist in versions in the library that are more exact and can handle larger argument ranges. The drawback is that they are larger and slower than the default versions.

The names of the functions are constructed like:

`__iar_`*xxx*`_accurate<f|l>`

where `f` is used for `float` variants, `l` is used for `long double` variants, and no suffix is used for `double` variants.

To specify individual math functions from the command line:

1. Redirect the default function names to these names when linking, using these options:

```
-e__iar_sin_accurate=sin
-e__iar_cos_accurate=cos
-e__iar_tan_accurate=tan
-e__iar_pow_accurate=pow
-e__iar_Sin_accurate=__iar_Sin
-e__iar_Pow_accurate=__iar_Pow

-e__iar_sin_accuratef=sinf
-e__iar_cos_accuratef=cosf
-e__iar_tan_accuratef=tanf
-e__iar_pow_accuratef=powf
-e__iar_Sin_accuratef=__iar_Sinf
-e__iar_Pow_accuratef=__iar_Powf
```

```
-e__iar_sin_accuratel=sinl
-e__iar_cos_accuratel=cosl
-e__iar_tan_accuratel=tanl
-e__iar_pow_accuratel=powl
-e__iar_Sin_accuratel=__iar_Sinl
-e__iar_Pow_accuratel=__iar_Powl
```

Note that if you want to redirect any of the functions `sin`, `cos`, or `__iar_Sin`, you must redirect all three functions.

Note that if you want to redirect any of the functions `pow` or `__iar_Pow`, you must redirect both functions.

SYSTEM STARTUP AND TERMINATION

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

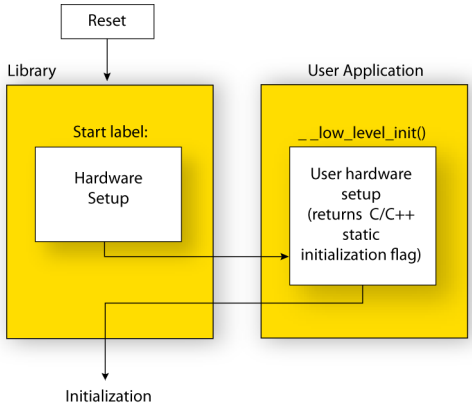
The code for handling startup and termination is located in the source files `cstartup.s51`, `cmain.s51`, `cexit.s51`, and `low_level_init.c` located in the `8051\src\lib` directory.

For information about how to customize the system startup code, see *System initialization*, page 167.

System startup

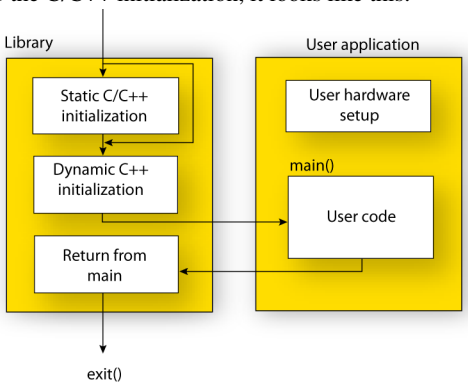
During system startup, an initialization sequence is executed before the `main` function is entered. This sequence performs initializations required for the target hardware and the C/C++ environment.

For the hardware initialization, it looks like this:



- When the CPU is reset it will start executing at 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 configuration 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 defined it, giving the application a chance to perform early initializations.

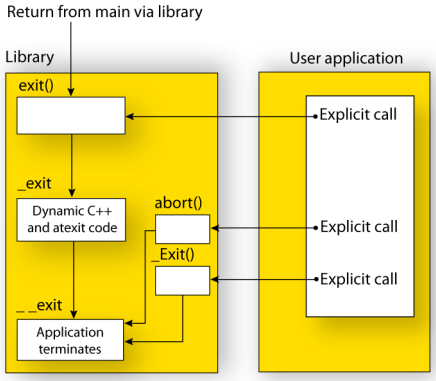
For the C/C++ initialization, it looks like this:



- Static and global variables are initialized. That is, zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This step is skipped if `__low_level_init` returns zero. For more information, see *Initialization at system startup*, page 123.
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

System termination

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



An application can terminate normally in two different ways:

- Return from the `main` function

- Call the `exit` function.

Because the 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:

- 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 function `atexit`.
- Close all open files
- Call `__exit`
- When `__exit` is reached, stop the system.

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

If you want your application to do anything extra at exit, for example resetting the system (and if using `atexit` is not sufficient), you can write your own implementation of the `__exit(int)` function.

The library files that you can override with your own versions are located in the `8051\src\lib` directory. See *Overriding library modules*, page 152.

C-SPY debugging support for system termination

If you have enabled C-SPY emulated I/O during linking, 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 emulate program termination. For more information, see *Briefly about C-SPY emulated I/O*, page 147.

SYSTEM INITIALIZATION

It is likely that you need to adapt the 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 the system startup code.

You can do this by implementing your own version of the routine `__low_level_init`, which is called from the file `cmain.s` before the data segments are initialized. Modifying the file `cstartup.s51` 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, you do not need to customize either of the files `cmain.s51` or `cexit.s51`.

Note: Regardless of whether you implement your own version of `__low_level_init` or the file `cstartup.s51`, you do not have to rebuild the library.

Customizing __low_level_init

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

As noted earlier, you should not modify the file `cstartup.s51` if implementing your own 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 152.

Note that you must make sure that the linker uses the start label used in your version of `cstartup.s51`. For information about how to change the start label used by the linker, read about the `-s` option in the *IAR Linker and Library Tools Reference Guide*.

THE DLIB LOW-LEVEL I/O INTERFACE

The runtime library uses a set of low-level functions—which are referred to as the *DLIB low-level I/O interface*—to communicate with the target system. Most of the low-level functions have no implementation.

For more information about this, see *Briefly about input and output (I/O)*, page 146.

This table lists the functions in the DLIB low-level I/O interface:

Function in DLIB low-level I/O interface	Defined in source file
`abort`	`abort.c`
`clock`	`clock.c`
`__close`	`close.c`
`__exit`	`xxexit.c`
`getenv`	`getenv.c`

Table 23: DLIB low-level I/O interface functions

Function in DLIB low-level I/O interface	Defined in source file
__getzone	getzone.c
__lseek	lseek.c
__open	open.c
raise	raise.c
__read	read.c
remove	remove.c
rename	rename.c
_ReportAssert	xreportassert.c
signal	signal.c
system	system.c
__time32	time.c
__write	write.c

Table 23: DLIB low-level I/O interface functions (Continued)

Note: You should not use the low-level functions prefixed with __ directly in your application. Instead you should use the standard library functions that use these functions. For example, to write to `stdout`, you should use standard library functions like `printf` or `puts`, which in turn calls the low-level function `__write`. If you have forgot to implement a low-level function and your application will call that function via a standard library function, the linker will issues an error when you link in release build configuration.

Note: If you implement your own variants of the functions in this interface, your variants will be used even though you have enabled C-SPY emulated I/O, see *Briefly about C-SPY emulated I/O*, page 147.

abort

Source file	`8051\src\lib\dlib\abort.c`
Description	Standard C library function that aborts execution.
C-SPY debug action	Notifies that the application has called abort.
Default implementation	Calls `__exit(EXIT_FAILURE)`.
See also	*Briefly about retargeting*, page 148 *System termination*, page 166.

clock

Source file	`8051\src\lib\dlib\clock.c`
Description	Standard C library function that accesses the processor time.
C-SPY debug action	Returns the clock on the host computer.
Default implementation	Returns `-1` to indicate that processor time is not available.
See also	*Briefly about retargeting*, page 148.

__close

Source file	`8051\src\lib\dlib\close.c`
Description	Low-level function that closes a file.
C-SPY debug action	Closes the associated host file on the host computer.
Default implementation	None.
See also	*Briefly about retargeting*, page 148.

__exit

Source file	`8051\src\lib\dlib\xxexit.c`
Description	Low-level function that halts execution.
C-SPY debug action	Notifies that the end of the application was reached.
Default implementation	Loops forever.
See also	*Briefly about retargeting*, page 148 *System termination*, page 166.

getenv

Source file	`8051\src\lib\dlib\getenv.c` `8051\src\lib\dlib\environ.c`

C-SPY debug action — Accesses the host environment.

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

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

```
key=value\0
```

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

For example:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function.

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

See also — *Briefly about retargeting*, page 148.

__getzone

Source file — `8051\src\lib\dlib\getzone.c`

Description — Low-level function that returns the current time zone.

C-SPY debug action — Not applicable.

Default implementation — Returns `":"`.

See also — *Briefly about retargeting*, page 148.

__lseek

Source file — `8051\src\lib\dlib\lseek.c`

Description — Low-level function for changing the location of the next access in an open file.

C-SPY debug action	Searches in the associated host file on the host computer.
Default implementation	None.
See also	*Briefly about retargeting*, page 148.

__open

Source file	`8051\src\lib\dlib\open.c`
Description	Low-level function that opens a file.
C-SPY debug action	Opens a file on the host computer.
Default implementation	None.
See also	*Briefly about retargeting*, page 148.

raise

Source file	`8051\src\lib\dlib\raise.c`
Description	Standard C library function that raises a signal.
C-SPY debug action	Not applicable.
Default implementation	Calls the signal handler for the raised signal, or terminates with call to `__exit(EXIT_FAILURE)`.
See also	*Briefly about retargeting*, page 148.

__read

Source file	`8051\src\lib\dlib\read.c`
Description	Low-level function that reads characters from `stdin` and from files.
C-SPY debug action	Directs `stdin` to the **Terminal I/O** window. All other files will read the associated host file.
Default implementation	None.

Example

The code in this example uses memory-mapped I/O to read from a keyboard, whose port is assumed to be located at `0x99`:

```
#include <stddef.h>

__sfr __no_init volatile unsigned char kbIO @ 0x99;

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

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

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

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

  return nChars;
}
```

For information about the handles associated with the streams, see *Retargeting—Adapting for your target system*, page 151.

For information about the `@` operator, see *Controlling data and function placement in memory*, page 259.

See also

Briefly about retargeting, page 148.

remove

Source file

`8051\src\lib\dlib\remove.c`

Description

Standard C library function that removes a file.

C-SPY debug action	Writes a message to the **Debug Log** window and returns `-1`.
Default implementation	Returns `0` to indicate success, but without removing a file.
See also	*Briefly about retargeting*, page 148.

rename

Source file	`8051\src\lib\dlib\rename.c`
Description	Standard C library function that renames a file.
C-SPY debug action	None.
Default implementation	Returns `-1` to indicate failure.
See also	*Briefly about retargeting*, page 148.

_ReportAssert

Source file	`8051\src\lib\dlib\xreportassert.c`
Description	Low-level function that handles a failed assert.
C-SPY debug action	Notifies the C-SPY debugger about the failed assert.
Default implementation	Failed asserts are reported by the function `__ReportAssert`. By default, it prints an error message and calls `abort`. If this is not the behavior you require, you can implement your own version of the function. The assert macro is defined in the header file `assert.h`. To turn off assertions, define the symbol `NDEBUG`.  In the IDE, the 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. See *NDEBUG*, page 393.
See also	*Briefly about retargeting*, page 148.

signal

Source file	`8051\src\lib\dlib\signal.c`

Description	Standard C library function that changes signal handlers.
C-SPY debug action	Not applicable.
Default implementation	As specified by Standard C. You might want to modify this behavior if the environment supports some kind of asynchronous signals.
See also	*Briefly about retargeting*, page 148.

system

Source file	`8051\src\lib\dlib\system.c`
Description	Standard C library function that executes commands.
C-SPY debug action	Notifies the C-SPY debugger that `system` has been called and then returns `-1`.
Default implementation	If you need to use the `system` function, you must implement it yourself. The system function available in the library returns `0` if a null pointer is passed to it to indicate that there is no command processor; otherwise it returns `-1` to indicate failure. If this is not the functionality that you require, you can implement your own version. This does not require that you rebuild the library.
See also	*Briefly about retargeting*, page 148.

__time32

Source file	`8051\src\lib\dlib\time.c`
Description	Low-level the current calendar time.
C-SPY debug action	Returns the time on the host computer.
Default implementation	Returns `-1` to indicate that calendar time is not available.
See also	*Briefly about retargeting*, page 148.

__write

Source file	`8051\src\lib\dlib\write.c`

Description — Low-level function that writes to `stdout`, `stderr`, or a file.

C-SPY debug action — Directs `stdout` and `stderr` to the **Terminal I/O** window. All other files will write to the associated host file.

Default implementation — None.

Example — The code in this example uses memory-mapped I/O to write to an LCD display, whose port is assumed to be located at address `0x99`:

```
#include <stddef.h>

__sfr __no_init volatile unsigned char lcdIO @ 0x99;

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

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

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

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

  return nChars;
}
```

For information about the handles associated with the streams, see *Retargeting—Adapting for your target system*, page 151.

See also — *Briefly about retargeting*, page 148.

CONFIGURATION SYMBOLS FOR PRINTF AND SCANF

If the provided formatters do not meet your requirements (*Formatters for printf*, page 159 and *Formatters for scanf*, page 160), you can customize the Full formatters. Note that this means you must rebuild the runtime library.

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

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

Printf configuration symbols	Includes support for
`_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 floating-point numbers
`_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
`_DLIB_PRINTF_CHAR_BY_CHAR`	Output char by char or buffered

Table 24: Descriptions of printf configuration symbols

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

Scanf configuration symbols	Includes support for
`_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 25: Descriptions of scanf configuration symbols

To customize the formatting capabilities, you must;

1. Define the configuration symbols in the library configuration file according to your application requirements.
2. Rebuild the library, see *Customizing and building your own runtime library*, page 153.

CONFIGURATION SYMBOLS FOR FILE INPUT AND OUTPUT

File I/O is only supported by libraries with the Full library configuration, see *Runtime library configurations*, page 155, or in a customized library when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is defined. If this symbol is not defined, functions taking a `FILE *` argument cannot be used.

To customize your library and rebuild it, see *Customizing and building your own runtime library*, page 153.

LOCALE

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

Depending on which library configuration 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. See *Runtime library configurations*, page 155.

The DLIB runtime library can be used in two main modes:

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

Locale support in prebuilt libraries

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

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

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

Customizing the locale support and rebuilding the library

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

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

Locale configuration symbols

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

```
#define _DLIB_FULL_LOCALE_SUPPORT 1
#define _LOCALE_USE_C         /* C locale */
#define _LOCALE_USE_EN_US     /* American English */
#define _LOCALE_USE_EN_GB     /* British English */
#define _LOCALE_USE_SV_SE     /* Swedish in Sweden */
```

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

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Customizing and building your own runtime library*, page 153.

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");
```

STRTOD

The function `strtod` does not accept hexadecimal floating-point strings in libraries with the normal library configuration. To make `strtod` accept hexadecimal floating-point strings, you must:

1. Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.
2. Rebuild the library, see *Customizing and building your own runtime library*, page 153.

The CLIB runtime environment

- Using a prebuilt runtime library
- Input and output
- System startup and termination
- Overriding default library modules
- Customizing system initialization
- C-SPY emulated I/O

Note that the CLIB runtime environment does not support any C99 functionality. For example, complex numbers and variable length arrays are not supported. Neither does CLIB support C++.

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

Using a prebuilt runtime library

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

- 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 *Customizing and building your own runtime library*, page 153.

CHOOSING A RUNTIME LIBRARY

The IDE includes the correct runtime library based on the options you select. See the *IDE Project Management and Building Guide for 8051* for more information.

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

```
cl_libname.r51
```

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 more information about the runtime libraries, see the chapter *C/C++ standard library functions*.

RUNTIME LIBRARY FILENAME SYNTAX

The runtime library names are constructed in this way:

```
{lib}-{core}{stack}-{code_mod}{data_mod}{cc}{const_loc}-{#dptrs}
{dptr_vis}{dptr_size}{dptr_select}.r51
```

where:

{*lib*}	`cl` for the IAR CLIB runtime environment
{*core*}	Specifies the processor variant: `pl` = the classic 8051 devices `e1` = the extended1 devices `e2` = the extended2 devices
{*stack*}	Specifies the stack location: `i` = idata stack `e` = extended stack
{*code_mod*}	Specifies the code model: `n` = Near `b` = Banked `2` = Banked_extended2 `f` = Far
{*data_mod*}	Specifies the data model: `s` = Small `l` = Large `g` = Generic `j` = Far Generic `f` = Far
{*cc*}	Specifies the calling convention: `d` = data overlay `o` = idata overlay `i` = idata reentrant `p` = pdata reentrant `x` = xdata reentrant `e` = extended stack reentrant
{*const_loc*}	Specifies the location for constants and strings: `d` = data `c` = code
{*#dptrs*}	A number from 1 to 8 that represents the number of data pointers used.

{*dptr_vis*}	Specifies the `DPTR` visibility: `h` = shadowed `e` = separate
{*dptr_size*}	Specifies the size of the data pointer in use: `16` = 16 bits `24` = 24 bits
{*dptr_select*}	Specifies 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`.

Input and output

You can customize:

- The functions related to character-based I/O
- The formatters used by `printf`/`sprintf` and `scanf`/`sscanf`. For information about how to choose a formatter for the 8051-specific functions `printf_P` and `scanf_P`, see 8051-specific CLIB functions.

CHARACTER-BASED I/O

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

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

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

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

```
__no_init volatile unsigned char devIO @ 8;

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

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

For information about how to include your own modified version of `putchar` and `getchar` in your project build process, see *Overriding library modules*, page 152.

FORMATTERS USED BY PRINTF AND SPRINTF

The `printf` and `sprintf` functions use a common formatter, called `_formatted_write`. There are three variants of the formatter:

```
_large_write
_medium_write
_small_write
```

By default, the linker automatically uses the most appropriate formatter for your application.

_large_write

The `_large_write` formatter supports the C89 `printf` format directives.

_medium_write

The `_medium_write` formatter has the same format directives as `_large_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 the large version.

_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 `_medium_write`.

Specifying the printf formatter in the IDE

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

Specifying the printf formatter from the command line

To explicitly specify and override the formatter used by default, add one of the following lines to the linker configuration file:

```
-e_small_write=_formatted_write
-e_medium_write=_formatted_write
-e_large_write=_formatted_write
```

Customizing printf

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

For such applications, a much reduced version of the `printf` function (without `sprintf`) is supplied in source form in the file `intwri.c`. This file can be modified to meet your requirements, and the compiled module inserted into the library in place of the original file; see *Overriding library modules*, page 152.

FORMATTERS USED BY SCANF AND SSCANF

As with the `printf` and `sprintf` functions, `scanf` and `sscanf` use a common formatter, called `_formatted_read`. There are two variants of the formatter:

```
_large_read
_medium_read
```

By default, the linker automatically uses the most appropriate formatter for your application.

_large_read

The `_large_read` formatter supports the C89 `scanf` format directives.

_medium_read

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

Specifying the scanf formatter in the IDE

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

Specifying the read formatter from the command line

To explicitly specify and override the formatter used by default, add one of the following lines to the linker configuration file:

```
-e_medium_read=_formatted_read
-e_large_read=_formatted_read
```

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, you do not need to customize either of the files `cmain.s51` or `cexit.s51`.

SYSTEM STARTUP

When an application is initialized, several steps are performed:

- 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 by the symbol `?REGISTER_BANK` in the linker configuration file.
- 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 defined it, giving the application a chance to perform early initializations.
- Static variables are initialized; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the remaining initialized variables
- The `main` function is called, which starts the application.

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

SYSTEM TERMINATION

An application can terminate normally in two different ways:

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

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

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

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

Overriding default library modules

The IAR CLIB Library contains modules which you probably need to override with your own customized modules, for example for character-based I/O, without rebuilding the entire library. For information about how to override default library modules, see *Overriding library modules*, page 152, in the chapter *The DLIB runtime environment*.

Customizing system initialization

For information about how to customize system initialization, see *System initialization*, page 167.

C-SPY emulated I/O

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

THE DEBUGGER TERMINAL I/O WINDOW

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

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

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

To make the **Terminal I/O** window available, the application must be linked with the XLINK option **With I/O emulation modules** selected. See the *IDE Project Management and Building Guide for 8051*.

TERMINATION

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

Assembler language interface

- Mixing C and assembler
- Calling assembler routines from C
- Calling assembler routines from C++
- Calling convention
- Assembler instructions used for calling functions
- Memory access methods
- Call frame information

Mixing C and assembler

The IAR C/C++ Compiler for 8051 provides several ways to access low-level resources:

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

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

INTRINSIC FUNCTIONS

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

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

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

MIXING C AND ASSEMBLER MODULES

It is possible to write parts of your application in assembler and mix them with your C or C++ modules.

This gives several benefits compared to using inline assembler:

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

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

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

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

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

The first question is discussed in the section *Calling assembler routines from C*, page 194. The following two are covered in the section *Calling convention*, page 197.

For information about how data in memory is accessed, see *Memory access methods*, page 209.

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 211.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 194, and *Calling assembler routines from C++*, page 196, respectively.

INLINE ASSEMBLER

Inline assembler can be used for inserting assembler instructions directly into a C or C++ function.

The `asm` extended keyword and its alias `__asm` both insert assembler instructions. However, when you compile C source code, the `asm` keyword is not available when the option `--strict` is used. The `__asm` keyword is always available.

Note: Not all assembler directives or operators can be inserted using these keywords.

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"
    "sjmp label");
```

where `\n` (new line) separates each new assembler instruction. Note that you can define and use local labels in inline assembler instructions, where the label must be placed in the same `asm()` as all references to this label.

The following example demonstrates the use of the `asm` keyword. 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 might have on the surrounding code are not taken into consideration. If, for example, registers or memory locations are altered, they might have to be restored within the sequence of inline assembler instructions for the rest of the code to work properly.

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

- The compiler's various optimizations will disregard any effects of the inline sequences, which will not be optimized at all
- In general, assembler directives will cause errors or have no meaning. Data definition directives will however work as expected
- Auto variables cannot be accessed.
- Labels cannot be declared.

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

Calling assembler routines from C

An assembler routine that will be called from C must:

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

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

or

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

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

CREATING SKELETON CODE

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

The following example shows how to create skeleton code to which you can easily add the functional body of the routine. The skeleton source code only needs to declare the

variables required and perform simple accesses to them. In this example, the assembler routine takes an `int` and a `char`, and then returns an `int`:

```
extern int gInt;
extern char gChar;

int Func(int arg1, char arg2)
{
  int locInt = arg1;
  gInt = arg1;
  gChar = arg2;
  return locInt;
}

int main()
{
  int locInt = gInt;
  gInt = Func(locInt, gChar);
  return 0;
}
```

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

COMPILING THE SKELETON CODE

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

Use these options to compile the skeleton code:

```
icc8051 skeleton.c -lA . -On -e
```

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`. The `-On` option means that no optimization will be used and `-e` enables language extensions. In addition, make sure to use relevant compiler options, usually the same as you use for other C or C++ source files in your project.

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 IDE, choose **Project>Options>C/C++ Compiler>List** and deselect the suboption **Include call frame information**.

On the command line, use the option `-lB` instead of -lA. Note that `CFI` information must be included in the source code to make the C-SPY **Call Stack** window work.

The output file

The output file contains the following important information:

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

The `CFI` directives describe the call frame information needed by the **Call Stack** window in the debugger. For more information, see *Call frame information*, page 211.

Calling assembler routines from C++

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

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

However, when using C linkage, the calling convention conforms to the C calling convention. An assembler routine can therefore be called from C++ when declared in this manner:

```
extern "C"
{
  int MyRoutine(int);
}
```

In C++, data structures that only use C features are known as PODs ("plain old data structures"), they use the same memory layout as in C. However, we do not recommend that you access non-PODs from assembler routines.

The following example shows how to achieve the equivalent to a non-static member function, which means that the implicit `this` pointer must be made explicit. It is also possible to "wrap" the call to the assembler routine in a member function. Use an inline member function to remove the overhead of the extra call—this assumes that function inlining is enabled:

```
class MyClass;

extern "C"
{
  void DoIt(MyClass *ptr, int arg);
}

class MyClass
{
public:
  inline void DoIt(int arg)
  {
    ::DoIt(this, arg);
  }
};
```

Note: Support for C++ names from assembler code is extremely limited. This means that:

- Assembler list files resulting from compiling C++ files cannot, in general, be passed through the assembler.
- It is not possible to refer to or define C++ functions that do not have C linkage in assembler.

Calling convention

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

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

The 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.

This section describes the calling conventions used by the compiler. These 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

You can choose between these 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 *Storage of auto variables and parameters*, page 83.

In the IDE, choose calling convention on the **Project>General Options>Target** page.

Hints for using the Normal calling convention

The Normal calling convention is 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 194.

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, because the linker produces errors for mismatches between the values.

For more information about checking module consistency, see *Checking module consistency*, page 141.

FUNCTION DECLARATIONS

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

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

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

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

USING C LINKAGE IN C++ SOURCE CODE

In C++, a function can have either C or C++ linkage. To call assembler routines from C++, it is easiest if you make the C++ function have C linkage.

This is an example of a declaration of a function with C linkage:

```
extern "C"
{
  int F(int);
}
```

It is often practical to share header files between C and C++. This is an example of a declaration that declares a function with C linkage in both C and C++:

```
#ifdef __cplusplus
extern "C"
{
#endif

int F(int);

#ifdef __cplusplus
}
#endif
```

PRESERVED VERSUS SCRATCH REGISTERS

The general 8051 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 can use the register for other purposes, but must save the value before using the register and restore it at the exit of the function.

For all calling conventions, all registers that are not scratch registers are preserved registers. These are `R6`, `R7`, all virtual registers (`V0`–`Vn`) 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, you must consider certain prerequisites:

- 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 114.
- 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 115.

FUNCTION ENTRANCE

Parameters can be passed to a function using one of these basic methods:

- In registers
- On the stack
- 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 use registers as much as possible. Only a limited number of registers can be used for passing parameters; when no more registers are available, the remaining parameters are passed on the stack or in the overlay frame. The parameters are also passed on the stack or in the overlay frame—depending on the calling convention—in these cases:

- Structure types: `struct`, `union`, and classes
- Unnamed parameters to variable length (variadic) 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 26: Registers used for passing parameters

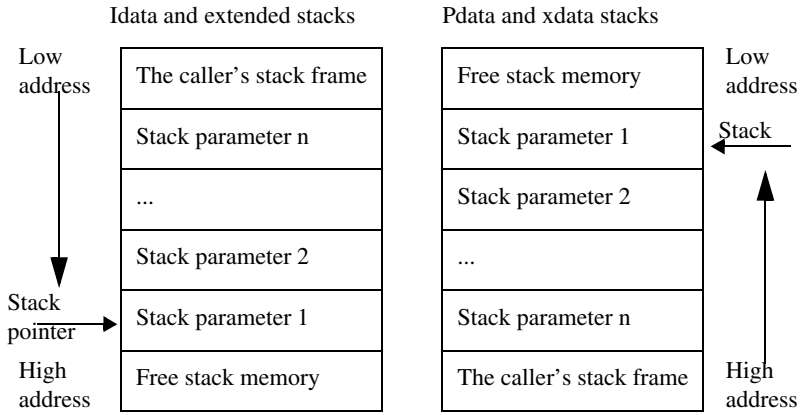
The assignment of registers to parameters is a straightforward process. Traversing the parameters in strict order from left to right, the first parameter is assigned to the first available register or registers. Should there be no suitable register available, the parameter is passed on the stack or overlay 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:



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 115.

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`

Table 27: Registers used for returning values

Return values	Passed in registers
16-bit values	`R3:R2`
24-bit values	`R3:R2:R1`
32-bit values	`R5:R4:R3:R2`

Table 27: Registers used for returning values (Continued)

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 198.

Stack layout at function exit

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 114.

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 8051 call stack located in idata memory	Assembler-written exit routine
Idata overlay	The 8051 call stack located in idata memory	Assembler-written exit routine
Idata reentrant	The 8051 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

Table 28: Registers used for returning values

Calling convention	Location of return address	Returns using †
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 28: 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 toward the end.

Example 1

Assume this 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.

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

```
name    return
rseg    CODE:CODE:NOROOT
mov     A,#1
add     A,R2
mov     R2,A
clr     A
addc    A,R3
mov     R3,A
ret
end
```

Example 2

This example shows how structures are passed on the stack. Assume these declarations:

```
struct MyStruct
{
  short a;
  short b;
  short c;
  short d;
  short e;
};

int MyFunction(struct MyStruct x, int y);
```

The calling function must reserve ten bytes on the top of the stack and copy the contents of the `struct` to that location. The integer parameter `y` is passed in the register `R1`. The return value is passed back to its caller in the register pair `R3:R2`.

Example 3

The function below will return a structure of type `struct MyStruct`.

```
struct MyStruct
{
  int mA[20];
};

struct MyStruct MyFunction(int x);
```

It is the responsibility of the calling function to allocate a memory location for the return value and pass a pointer to it as a hidden 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 was declared to return a pointer to the structure:

```
struct MyStruct *MyFunction(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.

FUNCTION DIRECTIVES

The function directives `FUNCTION`, `ARGFRAME`, `LOCFRAME`, and `FUNCALL` are generated by the compiler to pass information about functions and function calls to the IAR XLINK Linker. These directives can be seen if you use the compiler option **Assembler file** (`-lA`) to create an assembler list file.

For more information about the function directives, see the *IAR Assembler User Guide for 8051*.

Assembler instructions used for calling functions

This section presents the assembler instructions that can be used for calling and returning from functions on the 8051 microcontroller.

Functions can be called in different ways—directly via a function pointer. In this section we will discuss how these types of calls will be performed for each code model.

The normal function calling instruction is the `LCALL` instruction:

```
lcall label
```

The location that the called function should return to (that is, the location immediately after this 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 illustrate how the different code models perform function calls.

CALLING FUNCTIONS IN THE 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
```

CALLING FUNCTIONS IN THE 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
```

CALLING FUNCTIONS IN THE 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&0xFFFF)
```

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 presenting the assembler code used for accessing data, this section will explain the reason behind the different memory types.

You should be familiar with the 8051 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 are three examples:

- Accessing a global variable
- Accessing memory using a pointer.

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 _PDATA0_START and _PDATA0_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
```

FAR22, FAR, AND HUGE ACCESS METHODS

The far22, far, and huge memories consist of up to 16 Mbytes of external memory (extended xdata memory). Memory accesses are performed using 3-byte data pointers in the same way as the xdata accesses. A far22 or 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 METHODS

The generic access methods only apply to pointers.

In the Generic data model:

- a `__generic` attribute applied to a non-pointer data object it will be interpreted as `__xdata`
- the default pointer is `__generic` but the default data memory attribute is `__xdata`.

In the Far Generic data model:

- a `__generic` attribute applied to a non-pointer data object it will be interpreted as `__far22`
- the default pointer is `__generic` but the default data memory attribute is `__far22`.

A generic pointer is 3 bytes in size. The most significant byte reveals if the pointer points to internal data, external data, or code memory.

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 you debug an application using C-SPY, you can view the *call stack*, that is, the chain of functions that called the current function. To make this possible, the compiler

supplies debug information that describes the layout of the call frame, in particular information about where the return address is stored.

If you want the call stack to be available when debugging a routine written in assembler language, you must supply equivalent debug information in your assembler source using the assembler directive `CFI`. This directive is described in detail in the *IAR Assembler User Guide for 8051*.

CFI DIRECTIVES

The `CFI` directives provide C-SPY with information about the state of the calling function(s). Most important of this is the return address, and the value of the stack pointer at the entry of the function or assembler routine. Given this information, C-SPY can reconstruct the state for the calling function, and thereby unwind the stack.

A full description about the calling convention might require extensive call frame information. In many cases, a more limited approach will suffice. 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 194.

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

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

This table lists all the resources defined in the names block used by the compiler:

Resource	Description
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 DPTRs are used
R0–R7	Register R0–R7. The locations for these registers might change at runtime
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 29: Resources for call-frame information

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.

CREATING ASSEMBLER SOURCE WITH CFI SUPPORT

The recommended way to create an assembler language routine that handles call frame information correctly is to start with an assembler language source file created by the compiler.

1. Start with suitable C source code, for example:

```
int F(int);
int cfiExample(int i)
{
  return i + F(i);
}
```

2. Compile the C source code, and make sure to create a list file that contains call frame information—the CFI directives.

On the command line, use the option `-lA`.

In the IDE, choose **Project>Options>C/C++ Compiler>List** and make sure the suboption **Include call frame information** is selected.

For the source code in this example, the list file looks like this:

```
NAME Cfi

RTMODEL "__SystemLibrary", "CLib"
RTMODEL "__calling_convention", "xdata_reentrant"
RTMODEL "__code_model", "near"
RTMODEL "__core", "plain"
RTMODEL "__data_model", "large"
RTMODEL "__dptr_size", "16"
RTMODEL "__extended_stack", "disabled"
RTMODEL "__location_for_constants", "data"
RTMODEL "__number_of_dptrs", "1"
RTMODEL "__rt_version", "1"

RSEG DOVERLAY:DATA:NOROOT(0)
RSEG IOVERLAY:IDATA:NOROOT(0)
RSEG ISTACK:IDATA:NOROOT(0)
RSEG PSTACK:XDATA:NOROOT(0)
RSEG XSTACK:XDATA:NOROOT(0)

EXTERN ?FUNC_ENTER_XDATA
EXTERN ?FUNC_LEAVE_XDATA
EXTERN ?V0

PUBLIC cfiExample
FUNCTION cfiExample,021203H
ARGFRAME XSTACK, 0, STACK
LOCFRAME XSTACK, 9, STACK

CFI Names cfiNames0
CFI StackFrame CFA_SP SP IDATA
CFI StackFrame CFA_PSP16 PSP16 XDATA
CFI StackFrame CFA_XSP16 XSP16 XDATA
CFI StaticOverlayFrame CFA_IOVERLAY IOVERLAY
CFI StaticOverlayFrame CFA_DOVERLAY DOVERLAY
CFI Resource `PSW.CY`:1, `B.BR0`:1, `B.BR1`:1, `B.BR2`:1,
            `B.BR3`:1
CFI Resource `B.BR4`:1, `B.BR5`:1, `B.BR6`:1, `B.BR7`:1,
            `VB.BR8`:1
CFI Resource `VB.BR9`:1, `VB.BR10`:1, `VB.BR11`:1,
            `VB.BR12`:1
CFI Resource `VB.BR13`:1, `VB.BR14`:1, `VB.BR15`:1, VB:8,
            B:8, A:8
CFI Resource PSW:8, DPL0:8, DPH0:8, R0:8, R1:8, R2:8,
            R3:8, R4:8, R5:8
CFI Resource R6:8, R7:8, V0:8, V1:8, V2:8, V3:8, V4:8,
            V5:8, V6:8, V7:8
```

```
CFI Resource V8:8, V9:8, V10:8, V11:8, V12:8, V13:8,
            V14:8, V15:8
CFI Resource V16:8, V17:8, V18:8, V19:8, V20:8, V21:8,
            V22:8, V23:8
CFI Resource SP:8, PSPH:8, PSPL:8, PSP16:16, XSPH:8,
            XSPL:8, XSP16:16
CFI VirtualResource ?RET:16, ?RET_HIGH:8, ?RET_LOW:8
CFI ResourceParts PSP16 PSPH, PSPL
CFI ResourceParts XSP16 XSPH, XSPL
CFI ResourceParts ?RET ?RET_HIGH, ?RET_LOW
CFI EndNames cfiNames0

CFI Common cfiCommon0 Using cfiNames0
CFI CodeAlign 1
CFI DataAlign -1
CFI ReturnAddress ?RET CODE
CFI CFA_DOVERLAY Used
CFI CFA_IOVERLAY Used
CFI CFA_SP SP+-2
CFI CFA_PSP16 PSP16+0
CFI CFA_XSP16 XSP16+0
CFI `PSW.CY` SameValue
CFI `B.BR0` SameValue
CFI `B.BR1` SameValue
CFI `B.BR2` SameValue
CFI `B.BR3` SameValue
CFI `B.BR4` SameValue
CFI `B.BR5` SameValue
CFI `B.BR6` SameValue
CFI `B.BR7` SameValue
CFI `VB.BR8` SameValue
CFI `VB.BR9` SameValue
CFI `VB.BR10` SameValue
CFI `VB.BR11` SameValue
CFI `VB.BR12` SameValue
CFI `VB.BR13` SameValue
CFI `VB.BR14` SameValue
CFI `VB.BR15` SameValue
CFI VB SameValue
CFI B Undefined
CFI A Undefined
CFI PSW SameValue
CFI DPL0 SameValue
CFI DPH0 SameValue
CFI R0 Undefined
CFI R1 Undefined
CFI R2 Undefined
```

```
CFI R3 Undefined
CFI R4 Undefined
CFI R5 Undefined
CFI R6 SameValue
CFI R7 SameValue
CFI V0 SameValue
CFI V1 SameValue
CFI V2 SameValue
CFI V3 SameValue
CFI V4 SameValue
CFI V5 SameValue
CFI V6 SameValue
CFI V7 SameValue
CFI V8 SameValue
CFI V9 SameValue
CFI V10 SameValue
CFI V11 SameValue
CFI V12 SameValue
CFI V13 SameValue
CFI V14 SameValue
CFI V15 SameValue
CFI V16 SameValue
CFI V17 SameValue
CFI V18 SameValue
CFI V19 SameValue
CFI V20 SameValue
CFI V21 SameValue
CFI V22 SameValue
CFI V23 SameValue
CFI PSPH Undefined
CFI PSPL Undefined
CFI XSPH Undefined
CFI XSPL Undefined
CFI ?RET Concat
CFI ?RET_HIGH Frame(CFA_SP, 2)
CFI ?RET_LOW Frame(CFA_SP, 1)
CFI EndCommon cfiCommon0

EXTERN F
FUNCTION F,0202H
ARGFRAME ISTACK, 0, STACK
ARGFRAME PSTACK, 0, STACK
ARGFRAME XSTACK, 9, STACK
ARGFRAME IOVERLAY, 0, STATIC
ARGFRAME DOVERLAY, 0, STATIC

RSEG NEAR_CODE:CODE:NOROOT(0)
```

```
cfiExample:
        CFI Block cfiBlock0 Using cfiCommon0
        CFI Function cfiExample
        CODE

        FUNCALL cfiExample, F
        LOCFRAME ISTACK, 0, STACK
        LOCFRAME PSTACK, 0, STACK
        LOCFRAME XSTACK, 9, STACK
        LOCFRAME IOVERLAY, 0, STATIC
        LOCFRAME DOVERLAY, 0, STATIC
        ARGFRAME ISTACK, 0, STACK
        ARGFRAME PSTACK, 0, STACK
        ARGFRAME XSTACK, 9, STACK
        ARGFRAME IOVERLAY, 0, STATIC
        ARGFRAME DOVERLAY, 0, STATIC
        MOV     A,#-0x9
        LCALL   ?FUNC_ENTER_XDATA
        CFI DPH0 load(1, XDATA, add(CFA_XSP16, literal(-1)))
        CFI DPL0 load(1, XDATA, add(CFA_XSP16, literal(-2)))
        CFI ?RET_HIGH load(1, XDATA, add(CFA_XSP16, literal(-3)))
        CFI ?RET_LOW load(1, XDATA, add(CFA_XSP16, literal(-4)))
        CFI R7 load(1, XDATA, add(CFA_XSP16, literal(-5)))
        CFI V1 load(1, XDATA, add(CFA_XSP16, literal(-6)))
        CFI V0 load(1, XDATA, add(CFA_XSP16, literal(-7)))
        CFI VB load(1, XDATA, add(CFA_XSP16, literal(-8)))
        CFI R6 load(1, XDATA, add(CFA_XSP16, literal(-9)))
        CFI CFA_SP SP+0
        CFI CFA_XSP16 add(XSP16, 9)

        MOV     A,R2
        MOV     R6,A
        MOV     A,R3
        MOV     R7,A
        LCALL   F
        MOV     ?V0,R2
        MOV     ?V1,R3
        MOV     A,R6
        ADD     A,?V0
        MOV     R2,A
        MOV     A,R7
        ADDC    A,?V1
        MOV     R3,A
        MOV     R7,#0x2
        LJMP    ?FUNC_LEAVE_XDATA

        CFI EndBlock cfiBlock0
```

```
        END
```

Note: The header file `iar_cfi.m51` contains the macros `XCFI_NAMES` and `XCFI_COMMON`, which declare a typical names block and a typical common block. These two macros declare several resources, both concrete and virtual.

Using C

- C language overview
- Extensions overview
- IAR C language extensions

C language overview

The IAR C/C++ Compiler for 8051 supports the ISO/IEC 9899:1999 standard (including up to technical corrigendum No.3), also known as C99. In this guide, this standard is referred to as *Standard C* and is the default standard used in the compiler. This standard is stricter than C89.

In addition, the compiler also supports the ISO 9899:1990 standard (including all technical corrigenda and addenda), also known as C94, C90, C89, and ANSI C. In this guide, this standard is referred to as *C89*. Use the `--c89` compiler option to enable this standard.

The C99 standard is derived from C89, but adds features like these:

- The `inline` keyword advises the compiler that the function defined immediately after the keyword should be inlined
- Declarations and statements can be mixed within the same scope
- A declaration in the initialization expression of a `for` loop
- The `bool` data type
- The `long long` data type
- The `complex` floating-point type
- C++ style comments
- Compound literals
- Incomplete arrays at the end of structs
- Hexadecimal floating-point constants
- Designated initializers in structures and arrays
- The preprocessor operator `_Pragma()`
- Variadic macros, which are the preprocessor macro equivalents of `printf` style functions
- VLA (variable length arrays) must be explicitly enabled with the compiler option `--vla`

- Inline assembler using the `asm` or the `__asm` keyword, see *Inline assembler*, page 193.

Note: Even though it is a C99 feature, the IAR C/C++ Compiler for 8051 does not support UCNs (universal character names).

Note: CLIB does not support any C99 functionality. For example, complex numbers and variable length arrays are not supported.

Extensions overview

The compiler offers the features of Standard C and a wide set of extensions, ranging from features specifically tailored for efficient programming in the embedded industry to the relaxation of some minor standards issues.

This is an overview of the available extensions:

- IAR C language extensions

  For information about available language extensions, see *IAR C language extensions*, page 223. For more 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 Standard C 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. For several of them there is also a corresponding C/C++ language extension. For information about available pragma directives, see the chapter *Pragma directives*.

- Preprocessor extensions

  The preprocessor of the compiler adheres to Standard C. The compiler also makes several preprocessor-related extensions available to you. For more information, see the chapter *The preprocessor*.

- Intrinsic functions

  The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions. For more information about using intrinsic functions, see *Mixing C and*

assembler, page 191. For information about available functions, see the chapter *Intrinsic functions*.

- Library functions

  The DLIB runtime environment provides the C and C++ library definitions in the C/C++ standard library that apply to embedded systems. For more information, see *DLIB runtime environment—implementation details*, page 397.

Note: Any use of these extensions, except for the pragma directives, makes your source code inconsistent with Standard C.

ENABLING LANGUAGE EXTENSIONS

You can choose different levels of language conformance by means of project options:

Command line	IDE*	Description
`--strict`	**Strict**	All IAR C language extensions are disabled; errors are issued for anything that is not part of Standard C.
None	**Standard**	All *extensions to Standard C* are enabled, but no *extensions for embedded systems programming*. For information about extensions, see *IAR C language extensions*, page 223.
`-e`	**Standard with IAR extensions**	All *IAR C language extensions* are enabled.

Table 30: Language extensions

* In the IDE, choose **Project>Options>C/C++ Compiler>Language 1>Language conformance** and select the appropriate option. Note that language extensions are enabled by default.

IAR C language extensions

The compiler provides a wide set of C language extensions. To help you to find the extensions required by your application, they are grouped like this in this section:

- *Extensions for embedded systems programming*—extensions specifically tailored for efficient embedded programming for the specific microcontroller you are using, typically to meet memory restrictions
- *Relaxations to Standard C*—that is, the relaxation of some minor Standard C issues and also some useful but minor syntax extensions, see *Relaxations to Standard C*, page 226.

EXTENSIONS FOR EMBEDDED SYSTEMS PROGRAMMING

The following language extensions are available both in the C and the C++ programming languages and they 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 features, see *Controlling data and function placement in memory*, page 259, and *location*, page 372.

- Alignment control

  Each data type has its own alignment; for more information, see *Alignment*, page 325. 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 called 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 258.

- Bitfields and non-standard types

  In Standard C, a bitfield must be of the type `int` or `unsigned int`. Using IAR C language extensions, any integer type or enumeration can be used. The advantage is that the struct will sometimes be smaller. For more information, see *Bitfields*, page 327.

- `static_assert()`

  The construction `static_assert(const-expression,"message");` can be used in C/C++. The construction will be evaluated at compile time and if *`const-expression`* is false, a message will be issued including the *`message`* string.

- Parameters in variadic macros

  Variadic macros are the preprocessor macro equivalents of `printf` style functions. The preprocessor accepts variadic macros with no arguments, which means if no parameter matches the `...` parameter, the comma is then deleted in the `", ##__VA_ARGS__"` macro definition. According to Standard C, the `...` parameter must be matched with at least one argument.

Dedicated segment operators

The compiler supports getting the start address, end address, and size for a segment with these built-in segment operators:

`__segment_begin`	Returns the address of the first byte of the named segment.
`__segment_end`	Returns the address of the first byte *after* the named segment.
`__segment_size`	Returns the size of the named segment in bytes.

Note: The aliases `__segment_begin`/`__sfb`, `__segment_end`/`__sfe`, and `__segment_size`/`__sfs` can also be used.

The operators can be used on named segments defined in the linker configuration file.

These operators behave syntactically as if declared like:

```
void * __segment_begin(char const * segment)
void * __segment_end(char const * segment)
size_t __segment_size(char const * segment)
```

When you use the `@` operator or the `#pragma location` directive to place a data object or a function in a user-defined segment in the linker configuration file, the segment operators can be used for getting the start and end address of the memory range where the segments were placed.

The named *`segment`* must be a string literal and it must have been declared earlier with the `#pragma` *`segment`* directive. If the segment was declared with a memory attribute *`mematttr`*, the type of the `__segment_begin` operator is a pointer to *`mematttr`* `void`. Otherwise, the type is a default pointer to `void`. Note that you must enable language extensions to use these operators.

Example

In this example, the type of the `__segment_begin` operator is `void __pdata *`.

```
#pragma segment="MYSEGMENT" __pdata
...
segment_start_address = __segment_begin("MYSEGMENT");
```

See also *segment*, page 378, and *location*, page 372.

RELAXATIONS TO STANDARD C

This section lists and briefly describes the relaxation of some Standard C issues and also some useful but minor syntax extensions:

- Arrays of incomplete types

  An array can have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).

- Forward declaration of `enum` types

  The extensions allow you to first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.

- Accepting missing semicolon at the end of a `struct` or `union` specifier

  A warning—instead of an error—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 Standard C, some operators allow this kind of behavior, while others do not allow it.

- Casting pointers to integers in static initializers

  In an initializer, a pointer constant value can be cast to an integral type if the integral type is large enough to contain it. For more information about casting pointers, see *Casting*, page 333.

- Taking the address of a register variable

  In Standard 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.

- `long float` means `double`

  The type `long float` is accepted as a synonym for `double`.

- Repeated `typedef` declarations

  Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.

- Mixing pointer types

  Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.

  Assignment of a string constant to a pointer to any kind of character is allowed, and no warning is issued.

- Non-top level `const`

  Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). Comparing and taking the difference of such pointers is also allowed.

- Non-`lvalue` arrays

  A non-`lvalue` array expression is converted to a pointer to the first element of the array when it is used.

- Comments at the end of preprocessor directives

  This extension, which makes it legal to place text after preprocessor directives, is enabled unless the strict Standard C 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 Standard C mode, a warning is issued.

- A label preceding a `}`

  In Standard C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. The compiler allows this, but issues a warning.

  Note that 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

  Standard C requires that all initializer expressions of static arrays, structs, and unions are enclosed in braces.

  Single-value initializers are allowed to appear without braces, but a warning is issued. The compiler accepts this expression:

  ```
  struct str
  {
    int a;
  } x = 10;
  ```

- Declarations in other scopes

  External and static declarations in other scopes are visible. In the following example, the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

```
int test(int x)
{
  if (x)
  {
    extern int y;
    y = 1;
  }

  return y;
}
```

- Expanding function names into strings with the function as context

  Use any of the symbols `__func__` or `__FUNCTION__` inside a function body to make the symbol expand into a string that contains the name of the current function. 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 302.

- Static functions in function and block scopes

  Static functions may be declared in function and block scopes. Their declarations are moved to the file scope.

- Numbers scanned according to the syntax for numbers

  Numbers are scanned according to the syntax for numbers rather than the `pp-number` syntax. Thus, `0x123e+1` is scanned as three tokens instead of one valid token. (If the `--strict` option is used, the `pp-number` syntax is used instead.)

Using C++

- Overview—EC++ and EEC++
- Enabling support for C++
- EC++ feature descriptions
- EEC++ feature description
- C++ language extensions

Overview—EC++ and EEC++

IAR Systems supports the C++ language. You can choose between the industry-standard Embedded C++ and Extended Embedded C++. *Using C++* describes what you need to consider when using the C++ language.

Embedded C++ is a proper 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. EC++ offers the same object-oriented benefits as C++, but without some features that can increase code size and execution time in ways that are hard to predict.

EMBEDDED C++

These C++ features are supported:

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

C++ features that are excluded are those that introduce overhead in execution time or code size that are beyond the control of the programmer. Also excluded are features added very late before Standard C++ was defined. Embedded C++ thus offers a subset of C++ which is efficient and fully supported by existing development tools.

Embedded C++ lacks these features of C++:

- Templates
- Multiple and virtual inheritance
- Exception handling
- Runtime type information
- New cast syntax (the operators `dynamic_cast`, `static_cast`, `reinterpret_cast`, and `const_cast`)
- Namespaces
- The `mutable` attribute.

The exclusion of these language features makes the runtime library significantly more efficient. The Embedded C++ library furthermore differs from the full C++ library in that:

- The standard template library (STL) is excluded
- Streams, strings, and complex numbers are supported without the use of templates
- Library features which relate to exception handling and runtime type information (the headers `except`, `stdexcept`, and `typeinfo`) are excluded.

Note: The library is not in the `std` namespace, because Embedded C++ does not support namespaces.

EXTENDED EMBEDDED C++

IAR Systems' Extended EC++ is a slightly larger subset of C++ which adds these features to the standard EC++:

- Full template support
- Namespace support
- The `mutable` attribute
- The cast operators `static_cast`, `const_cast`, and `reinterpret_cast`.

All these added features conform to the C++ standard.

To support Extended EC++, this product includes a version of the standard template library (STL), in other words, the C++ standard chapters utilities, containers, iterators, algorithms, and some numerics. This STL is tailored for use with the Extended EC++

language, which means no exceptions and no support for runtime type information (`rtti`). Moreover, the library is not in the `std` namespace.

Note: A module compiled with Extended EC++ enabled is fully link-compatible with a module compiled without Extended EC++ enabled.

Enabling support for C++

In the compiler, the default language is C.

To compile files written in Embedded C++, use the `--ec++` compiler option. See *--ec++*, page 303.

To take advantage of *Extended* Embedded C++ features in your source code, use the `--eec++` compiler option. See *--eec++*, page 303.

For EC++, and EEC++, you must also use the IAR DLIB runtime library.

To enable EC++ or EEC++ in the IDE, choose **Project>Options>C/C++ Compiler>Language 1** and select the appropriate standard.

EC++ feature descriptions

When you write C++ source code for the IAR C/C++ Compiler for 8051, you must be aware of some benefits and some possible quirks when mixing C++ features—such as classes, and class members—with IAR language extensions, such as IAR-specific attributes.

USING IAR ATTRIBUTES WITH CLASSES

Static data members of C++ classes are treated the same way global variables are, and can have any applicable IAR type, memory, and object attribute.

Member functions are in general treated the same way free functions are, and can have any applicable IAR type, memory, and object attributes. Virtual member functions can only have attributes that are compatible with default function pointers, and constructors and destructors cannot have any such attributes.

The location operator `@` and the `#pragma location` directive can be used on static data members and with all member functions.

Example of using attributes with classes

```
class MyClass
{
public:
  // Locate a static variable in xdata memory at address 60
  static __xdata __no_init int mI @ 60;

  // Locate a static function in __near_func (code) memory
  static __near_func void F();

  // Locate a function in __near_func memory
  __near_func void G();

  // Locate a virtual function in __near_func memory
  virtual __near_func void H();

  // Locate a virtual function into SPECIAL
  virtual void M() const volatile @ "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 might also apply to objects residing on a stack, for example temporary objects and auto objects.

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 MyF();       // Has a this pointer of type C __xdata *
  void MyF() 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 mI;
};

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

void MyH()
{
  C cd;             // Resides on the stack
}

C *Cp1;             // Creates a pointer to xdata memory
C __pdata *Cp2;     // Creates a pointer to pdata memory
```

Note: To place the C class in xdata memory is not allowed because a huge pointer cannot be implicitly converted into a `__xdata` pointer.

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.

A built-in operator 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 MyG();
  int mJ;
};

class __pdata E : public C
{ // OK, pdata memory is inside xdata
public:
  void MyG() // Has a this pointer pointing into pdata memory
  {
    MyF();    // Gets a this pointer into xdata memory
  }
  int mJ;
};

class F : public C
{ // OK, will be associated with same class memory as C
public:
  void MyG();
  int mJ;
};
```

Note that the following is not allowed because huge is not inside far memory:

```
class __huge G:public C
{
};
```

A `new` expression on the class will allocate memory in the heap associated with the class memory. A `delete` expression will naturally deallocate the memory back to the same heap. To override the default `new` and `delete` operator for a class, declare

```
void *operator new(size_t);
void operator delete(void *);
```

as member functions, just like in ordinary C++.

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 68.

FUNCTION TYPES

A function type with `extern "C"` linkage is compatible with a function that has C++ linkage.

Example

```
extern "C"
{
  typedef void (*FpC)(void);     // A C function typedef
}

typedef void (*FpCpp)(void);     // A C++ function typedef

FpC F1;
FpCpp F2;
void MyF(FpC);

void MyG()
{
  MyF(F1);                       // Always works
  MyF(F2);                       // FpCpp is compatible with FpC
}
```

NEW AND DELETE OPERATORS

There are operators for `new` and `delete` for each memory that can have a heap, that is, xdata, far, and huge memory.

```
#include <stddef.h>

// Assumes that there is a heap in both __xdata and __far memory
#if __DATA_MODEL__ >= 4
void __far *operator new __far(__far_size_t);
void operator delete(void __far *);
#else
void __xdata  *operator new __xdata (__xdata_size_t);
void operator delete(void __xdata  *);
#endif
// And correspondingly for array new and delete operators
#if __DATA_MODEL__ >= __DM_FAR__
void __far *operator new[] __far(__far_size_t);
void operator delete[](void __far *);
#else
void __xdata  *operator new[] __xdata (__xdata_size_t);
void operator delete[](void __xdata  *);
#endif
```

Use this syntax if you want to override both global and class-specific `operator new` and `operator delete` for any data memory.

Note that there is a special syntax to name the `operator new` functions for each memory, while the naming for the `operator delete` functions relies on normal overloading.

New and delete expressions

A `new` expression calls the `operator new` function for the memory of the type given. If a `class`, `struct`, or `union` type with a class memory is used, the class memory will determine the `operator new` function called. For example,

```
void MyF()
{
  // Calls operator new __far(__far_size_t)
  int __far *p = new __far int;

  // Calls operator new __far(__far_size_t)
  int __far *q = new int __far;

  // Calls operator new[] __far(__far_size_t)
  int __far *r = new __far int[10];

  // Calls operator new __huge(__huge_size_t)
  class __huge S
  {
  };
  S *s = new S;

  // Calls operator delete(void __far *)
  delete p;
  // Calls operator delete(void __huge  *)
  delete s;

  int __huge *t = new __far int;
  delete t; // Error: Causes a corrupt heap
}
```

Note that the pointer used in a `delete` expression must have the correct type, that is, the same type as that returned by the `new` expression. If you use a pointer to the wrong memory, the result might be a corrupt heap.

USING STATIC CLASS OBJECTS IN INTERRUPTS

If interrupt functions use static class objects that need to be constructed (using constructors) or destroyed (using destructors), your application will not work properly if the interrupt occurs before the objects are constructed, or, during or after the objects are destroyed.

To avoid this, make sure that these interrupts are not enabled until the static objects have been constructed, and are disabled when returning from `main` or calling `exit`. For information about system startup, see *System startup and termination*, page 164.

Function local static class objects are constructed the first time execution passes through their declaration, and are destroyed when returning from `main` or when calling `exit`.

USING NEW HANDLERS

To handle memory exhaustion, you can use the `set_new_handler` function.

New handlers in Embedded C++

If you do not call `set_new_handler`, or call it with a NULL new handler, and `operator new` fails to allocate enough memory, it will call `abort`. The `nothrow` variant of the new operator will instead return NULL.

If you call `set_new_handler` with a non-NULL new handler, the provided new handler will be called by `operator new` if `operator new` fails to allocate memory. The new handler must then make more memory available and return, or abort execution in some manner. The `nothrow` variant of `operator new` will never return NULL in the presence of a new handler.

TEMPLATES

Extended EC++ supports templates according to the C++ standard, but not the `export` keyword. The implementation uses a two-phase lookup which means that the keyword `typename` must be inserted wherever needed. Furthermore, at each use of a template, the definitions of all possible templates must be visible. This means that the definitions of all templates must be in include files or in the actual source file.

DEBUG SUPPORT IN C-SPY

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.

For more information about this, see the *C-SPY® Debugging Guide for 8051*.

EEC++ feature description

This section describes features that distinguish Extended EC++ from EC++.

TEMPLATES

The compiler supports templates with the syntax and semantics as defined by Standard C++. However, note that the STL (standard template library) delivered with the product is tailored for Extended EC++, see *Extended Embedded C++*, page 230.

Templates and data memory attributes

For data memory attributes to work as expected in templates, two elements of the standard C++ template handling were changed—class template partial specialization matching and function template parameter deduction.

In Extended Embedded C++, the class template partial specialization matching algorithm works like this:

When a pointer or reference type is matched against a pointer or reference to a template parameter type, the template parameter type will be the type pointed to, stripped of any data memory attributes, if the resulting pointer or reference type is the same.

Example

```
// We assume that __far is the memory type of the default
// pointer.
template<typename> class Z {};
template<typename T> class Z<T *> {};

Z<int __pdata *> Zn;   // T = int __pdata
Z<int __far   *> Zf;   // T = int
Z<int         *> Zd;   // T = int
Z<int __huge  *> Zh;   // T = int __huge
```

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

```
// We assume that __far is the memory type of the default
// pointer.
template<typename T> void fun(T *);

void MyF()
{
  fun((int __pdata *) 0); // T = int. The result is different
                          // than the analoguous situation with
                          // class template specializations.
  fun((int         *) 0); // T = int
  fun((int __far   *) 0); // T = int
  fun((int __huge  *) 0); // T = int __huge
}
```

For templates that are matched using this modified algorithm, it is impossible to get automatic generation of special code for pointers to “small” memory types. For “large” and “other” memory types (memory that cannot be pointed to by a default pointer) it is possible. To make it possible to write templates that are fully memory-aware—in the rare cases where this is useful—use the `#pragma basic_template_matching` directive in front of the template function declaration. That template function will then match without the modifications described above.

Example

```
// We assume that __pdata is the memory type of the default
// pointer.
#pragma basic_template_matching
template<typename T> void fun(T *);

void MyF()
{
  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  __no_init __sfr int X @0xf2;

template<__sfr int &y>
void Foo()
{
  y = 17;
}

void Bar()
{
  Foo<X>();
}
```

The standard template library

The containers in the STL, like `vector` and `map`, are memory attribute aware. This means that a container can be declared to reside in a specific memory type which has the following consequences:

- The container itself will reside in the chosen memory
- Allocations of elements in the container will use a heap for the chosen memory
- All references inside it use pointers to the chosen memory.

Example

```
#include <vector>

vector<int> D;                        // D placed in default
memory,
                                      // using the default heap,
                                      // uses default pointers
vector<int __far22> __far22 X;        // X placed in far22 memory,
                                      // heap allocation from
                                      // far22, uses pointers to
                                      // far22 memory
vector<int __huge> __far22 Y;         // Y placed in far22 memory,
                                      // heap allocation from
                                      // Huge, uses pointers to
                                      // Huge memory
```

Note that this is illegal:

```
vector<int __far22> __huge Z;
```

Note also that `map<key, T>`, `multimap<key, T>`, `hash_map<key, T>`, and `hash_multimap<key, T>` all use the memory of `T`. This means that the `value_type`

of these collections will be `pair<key, const T>` *mem* where *mem* is the memory type of `T`. Supplying a key with a memory type is not useful.

Example

Note that two containers that only differ by the data memory attribute they use cannot be assigned to each other. Instead, the templated assign member method must be used.

```
#include <vector>

vector<int __far> X;
vector<int __huge> Y;

void MyF()
{
  // The templated assign member method will work
  X.assign(Y.begin(), Y.end());
  Y.assign(X.begin(), X.end());
}
```

VARIANTS OF CAST OPERATORS

In Extended EC++ these additional variants of C++ cast operators can be used:

```
const_cast<to>(from)
static_cast<to>(from)
reinterpret_cast<to>(from)
```

MUTABLE

The `mutable` attribute is supported in Extended EC++. A `mutable` symbol can be changed even though the whole class object is `const`.

NAMESPACE

The namespace feature is only supported in *Extended* EC++. This means that you can use namespaces to partition your code. Note, however, that the library itself is not placed in the `std` namespace.

THE STD NAMESPACE

The `std` namespace is not used in either standard EC++ or in Extended EC++. If you have code that refers to symbols in the `std` namespace, simply define `std` as nothing; for example:

```
#define std
```

You must make sure that identifiers in your application do not interfere with identifiers in the runtime library.

POINTER TO MEMBER FUNCTIONS

A pointer to a member function can only contain a default function pointer, or a function pointer that can implicitly be casted to a default function pointer. To use a pointer to a member function, make sure that all functions that should be pointed to reside in the default memory or a memory contained in the default memory.

Example

```
class X
{
public:
  __near_func1 void F();
};

void (__near_func1 X::*PMF)(void) = &X::F;
```

C++ language extensions

When you use the compiler in any C++ mode and enable IAR language extensions, the following C++ language extensions are available in the compiler:

- In a `friend` declaration of a class, the `class` keyword can be omitted, for example:

  ```
  class B;
  class A
  {
    friend B;       //Possible when using IAR language
                    //extensions
    friend class B; //According to the standard
  };
  ```

- Constants of a scalar type can be defined within classes, for example:

  ```
  class A
  {
    const int mSize = 10; //Possible when using IAR language
                          //extensions
    int mArr[mSize];
  };
  ```

  According to the standard, initialized static data members should be used instead.

- In the declaration of a class member, a qualified name can be used, for example:

  ```
  struct A
  {
    int A::F(); // Possible when using IAR language extensions
    int G();    // According to the standard
  };
  ```

- It is permitted to use an implicit type conversion between a pointer to a function with C linkage (`extern "C"`) and a pointer to a function with C++ linkage (`extern "C++"`), for example:

```
extern "C" void F(); // Function with C linkage
void (*PF)()       // PF points to a function with C++ linkage
             = &F; // Implicit conversion of function pointer.
```

  According to the standard, the pointer must be explicitly converted.

- If the second or third operands in a construction that contains the `?` operator are string literals or wide string literals (which in C++ are constants), the operands can be implicitly converted to `char *` or `wchar_t *`, for example:

```
bool X;

char *P1 = X ? "abc" : "def";       //Possible when using IAR
                                    //language extensions
char const *P2 = X ? "abc" : "def";//According to the standard
```

- Default arguments can be specified for function parameters not only in the top-level function declaration, which is according to the standard, but also in `typedef` declarations, in pointer-to-function function declarations, and in pointer-to-member function declarations.

- In a function that contains a non-static local variable and a class that contains a non-evaluated expression (for example a `sizeof` expression), the expression can reference the non-static local variable. However, a warning is issued.

- An anonymous union can be introduced into a containing class by a `typedef` name. It is not necessary to first declare the union. For example:

```
typedef union
{
  int i,j;
} U;  // U identifies a reusable anonymous union.

class A
{
public:
  U;  // OK -- references to A::i and A::j are allowed.
};
```

  In addition, this extension also permits *anonymous classes* and *anonymous structs*, as long as they have no C++ features (for example, no static data members or member

functions, and no non-public members) and have no nested types other than other anonymous classes, structs, or unions. For example:

```
struct A
{
  struct
  {
    int i,j;
  }; // OK -- references to A::i and A::j are allowed.
};
```

- The friend class syntax allows nonclass types as well as class types expressed through a `typedef` without an elaborated type name. For example:

```
typedef struct S ST;

class C
{
public:
  friend S;  // Okay (requires S to be in scope)
  friend ST; // Okay (same as "friend S;")
  // friend S const; // Error, cv-qualifiers cannot
                     // appear directly
};
```

- It is allowed to specify an array with no size or size 0 as the last member of a struct. For example:

```
typedef struct
{
  int i;
  char ir[0]; // Zero-length array
};

typedef struct
{
  int i;
  char ir[];  // Zero-length array
};
```

Note: If you use any of these constructions without first enabling language extensions, errors are issued.

Application-related considerations

- Stack considerations
- Heap considerations
- Interaction between the tools and your application
- Checksum calculation for verifying image integrity

Stack considerations

To make your application use stack memory efficiently, there are some considerations to be made.

STACK SIZE CONSIDERATIONS

The required stack size depends heavily on the application's behavior. If the given stack size is too large, RAM will be wasted. If the given stack size is too small, one of two things can happen, depending on where in memory you located your stack:

- Variable storage will be overwritten, leading to undefined behavior
- The stack will fall outside of the memory area, leading to an abnormal termination of your application.

Both alternatives are likely to result in application failure. Because the second alternative is easier to detect, you should consider placing your stack so that it grows toward the end of the RAM memory.

For more information about the stack size, see *Setting up stack memory*, page 132, and *Saving stack space and RAM memory*, page 270. See also the chip manufacturer's documentation for details about stack size.

Heap considerations

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

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

- Linker segments used for the heap
- Allocating the heap size, see *Setting up heap memory*, page 135.

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:

```
__pdata_malloc
```

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

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

For information about available heaps, see *Dynamic memory on the heap*, page 90.

HEAP SEGMENTS IN CLIB

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 AND STANDARD I/O

If you 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 8051 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.

Interaction between the tools and your application

The linking process and the application can interact symbolically in four ways:

- Creating a symbol by using the linker command line option `-D`. The linker will create a public absolute constant symbol that the application can use as a label, as a size, as setup for a debugger, etc.
- Using the compiler operators `__segment_begin`, `__segment_end`, or `__segment_size`, or the assembler operators `SFB`, `SFE`, or `SIZEOF` on a named

segment. These operators provide access to the start address, end address, and size of a contiguous sequence of segments with the same name

- The command line option `-s` informs the linker about the start label of the application. It is used by the linker as a root symbol and to inform the debugger where to start execution.

The following lines illustrate how to use `-D` to create a symbol. If you need to use this mechanism, add these options to your command line like this:

```
-Dmy_symbol=A
-DMY_HEAP_SIZE=400
```

The linker configuration file can look like this:

```
-Z(DATA)MyHeap+MY_HEAP_SIZE20000-2FFFF
```

Add these lines to your application source code:

```
#include <stdlib.h>

/* Use symbol defined by an XLINK option to dynamically allocate
an array of elements with specified size. The value takes the
form of a label.
 */
extern int NrOfElements;

typedef char Elements;
Elements *GetElementArray()
{
  return malloc(sizeof(Elements) * (long) &NrOfElements);
}

/* Use a symbol defined by an XLINK option, a symbol that in the
 * configuration file was made available to the application.
 */
extern char HeapSize;
```

```
/* Declare the section that contains the heap. */
#pragma segment = "MYHEAP"

char *MyHeap()
{
  /* First get start of statically allocated section, */
  char *p = __segment_begin("MYHEAP");

  /* ...then we zero it, using the imported size. */
  for (int i = 0; i < (int) &HeapSize; ++i)
  {
    p[i] = 0;
  }
  return p;
}
```

Checksum calculation for verifying image integrity

This section contains information about checksum calculation:

- *Briefly about checksum calculation*, page 250
- *Calculating and verifying a checksum*, page 252
- *Troubleshooting checksum calculation*, page 256

BRIEFLY ABOUT CHECKSUM CALCULATION

You can use a checksum to verify that the image is the same at runtime as when the image's original checksum was generated. In other words, to verify that the image has not been corrupted.

This works as follows:

- You need an initial checksum.

  You can either use the linker to generate an initial checksum or you might have a third-party checksum available.
- You must generate a second checksum during runtime.

  You can either add specific code to your application source code for calculating a checksum during runtime or you can use some dedicated hardware on your device for calculating a checksum during runtime.
- You must add specific code to your application source code for comparing the two checksums and take an appropriate action if they differ.

  If the two checksums have been calculated in the same way, and if there are no errors in the image, the checksums should be identical. If not, you should first suspect that the two checksums were not generated in the same way.

No matter which solutions you use for generating the two checksum, you must make sure that both checksums are calculated *in the exact same way*. If you use the linker for the initial checksum and use a software-based calculation during runtime, you have full control of the generation for both checksums. However, if you are using a third-party checksum for the initial checksum or some hardware support for the checksum calculation during runtime, there might be additional requirements that you must consider.

For the two checksums, there are some choices that you must always consider and there are some choices to make only if there are additional requirements. Still, all of the details must be the same for both checksums.

Details always to consider:

- *Checksum range*

  The memory range (or ranges) that you want to verify by means of checksums. Typically, you might want to calculate a checksum for all ROM memory. However, you might want to calculate a checksum only for specific ranges. Remember that:

  - It is OK to have several ranges for one checksum.
  - Typically, the checksum must be calculated from the lowest to the highest address for every memory range.
  - Each memory range must be verified in the same order as defined (for example, `0x100-0x1FF,0x400-0x4FF` is not the same as `0x400-0x4FF,0x100-0x1FF`).
  - If several checksums are used, you should place them in sections with unique names and use unique symbol names.
  - A checksum should never be calculated on a memory range that contains a checksum or a software breakpoint.

- *Algorithm and size of checksum*

  You should consider which algorithm is most suitable in your case. There are two basic choices, Sum (a simple arithmetic algorithm) or CRC (which is the most commonly used algorithm). For CRC there are different sizes to choose for the checksum, 2 or 4 bytes where the predefined polynomials are wide enough to suit the size, for more error detecting power. The predefined polynomials work well for most, but possibly not for all data sets. If not, you can specify your own polynomial. If you just want a decent error detecting mechanism, use the predefined CRC algorithm for your checksum size, typically CRC16 or CRC32.

  Note that for an n-bit polynomial, the n:th bit is always considered to be set. For a 16-bit polynomial (for example, CRC16) this means that `0x11021` is the same as `0x1021`.

  For more information about selecting an appropriate polynomial for data sets with non-uniform distribution, see for example section 3.5.3 in *Tannenbaum, A.S., "Computer Networks," Prentice Hall 1981, ISBN: 0131646990.*

- *Fill*

  Every byte in the checksum range must have a well-defined value before the checksum can be calculated. Typically, bytes with unknown values are *pad bytes* that have been added for alignment. This means that you must specify which fill pattern to be used during calculation, typically `0xFF` or `0x00`.

- *Initial value*

  The checksum must always have an explicit initial value.

In addition to these mandatory details, there might be other details to consider. Typically, this might happen when you have a third-party checksum, you want the checksum be compliant with the Rocksoft™ checksum model, or when you use hardware support for generating a checksum during runtime. The linker provides support for also controlling alignment, complement, bit order, and checksum unit size.

CALCULATING AND VERIFYING A CHECKSUM

In this example procedure, a checksum is calculated for ROM memory from `0x8002` up to `0x8FFF` and the 2-byte calculated checksum is placed at `0x8000`.

1. The `CHECKSUM` segment will only be included in your application if the segment appears to be needed. If the checksum is not needed by the application itself, use the linker option `-g__checksum` to force the segment to be included.
2. When configuring the linker to calculate a checksum, there are some basic choices to make:

   - Checksum algorithm

     Choose which checksum algorithm you want to use. In this example, the CRC16 algorithm is used.

   - Memory range

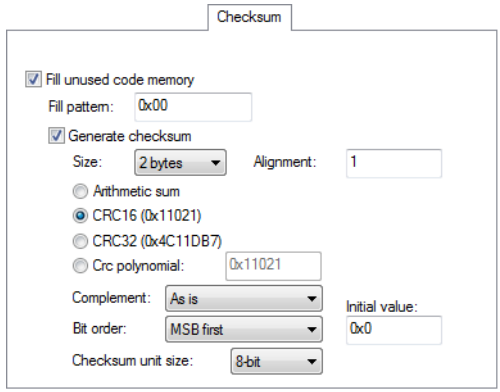
     Using the IDE, the checksum will by default be calculated for all placement directives (specified in the linker configuration file) for ROM-based memory. From the command line, you can specify any ranges.

   - Fill pattern

     Specify a fill pattern—typically `0xFF` or `0x00`—for bytes with unknown values. The fill pattern will be used in all checksum ranges.

   For more information, see *Briefly about checksum calculation*, page 250.

To run the linker from the IDE, choose **Project>Options>Linker>Checksum** and make your settings, for example:



In the simplest case, you can ignore (or leave with default settings) these options: **Complement**, **Bit order**, and **Checksum unit size**.

To make the linker create a checksum from the command line, use the `-J` linker option, for example like this:

```
-J2,crc16,,__checksum,CHECKSUM,1=0x8002-0x8FFF
```

The checksum will be created when you build your project and will be placed in the automatically generated segment `CHECKSUM`. If you are using your own linker configuration file or if you explicitly want to control the placement of the `CHECKSUM` segment, you must update your linker configuration file with placement information accordingly. In that case, make sure to place the segment so that it is not part of the application's checksum calculation.

3 You can specify several ranges instead of only one range.

If you are using the IDE, perform these steps:

- Choose **Project>Options>Linker>Checksum** and make sure to deselect **Fill unused code memory**.
- Choose **Project>Options>Linker>Extra Opions** and specify the ranges, for example like this:

```
-h(CODE)0-3FF,8002-8FFF
-J2,crc16,,,1=0-3FF,8002-8FFF
```

If you are using the command line, use the `-J` option and specify the ranges. for example like this:

```
-h(CODE)0-3FF,8002-8FFF
-J2,crc16,,,1=0-3FF,8002-8FFF
```

4 Add a function for checksum calculation to your source code. Make sure that the function uses the same algorithm and settings as for the checksum calculated by the linker. For example, a slow variant of the crc16 algorithm but with small memory footprint (in contrast to the fast variant that uses more memory):

```
unsigned short SmallCrc16(uint16_t
 sum,
                          unsigned char *p,
                          unsigned int len)
{
  while (len--)
  {
    int i;
    unsigned char byte = *(p++);

    for (i = 0; i < 8; ++i)
    {
      unsigned long oSum = sum;
      sum <<= 1;
      if (byte & 0x80)
        sum |= 1;
      if (oSum & 0x8000)
        sum ^= 0x1021;
      byte <<= 1;
    }
  }
  return sum;
}
```

You can find the source code for this checksum algorithm in the `8051\src\linker` directory of your product installation.

5 Make sure that your application also contains a call to the function that calculates the checksum, compares the two checksums, and takes appropriate action if the checksum values do not match. This code gives an example of how the checksum can be calculated for your application and to be compared with the linker generated checksum:

```
/* The calculated checksum */

/* Linker generated symbols */
extern unsigned short const __checksum;
extern int __checksum_begin;
extern int __checksum_end;


void TestChecksum()
{
  unsigned short calc = 0;
  unsigned char  zeros[2] = {0, 0};

  /* Run the checksum algorithm */
  calc = SmallCrc16(0,
                    (unsigned char *) &__checksum_begin,
                    ((unsigned char *) &__checksum_end -
                    ((unsigned char *) &__checksum_begin)+1));

  /* Fill the end of the byte sequence with zeros. */
  calc = SmallCrc16(calc, zeros, 2);

  /* Test the checksum */
  if (calc != __checksum)
  {
    printf("Incorrect checksum!\n");
    abort();   /* Failure */
  }

  /* Checksum is correct */
}
```

Note: Make sure to define the symbols `__checksum_begin` and `__checksum_end` in your linker configuration file.

6 Build your application project and download it.

During the build, the linker creates a checksum and places it in the specified symbol `__checksum` in the segment `CHECKSUM`.

7 Choose **Download and Debug** to start the C-SPY debugger.

During execution, the checksum calculated by the linker and the checksum calculated by your application should be identical.

TROUBLESHOOTING CHECKSUM CALCULATION

If the two checksums do not match, there are several possible causes. These are some troubleshooting hints:

- If possible, start with a small example when trying to get the checksums to match.
- Verify that the exact same memory range or ranges are used in both checksum calculations.

  To help you do this, The linker produces useful information in the map file about the exact addresses that were used and the order in which they were accessed.
- Make sure that all checksum symbols are excluded from all checksum calculations.

  In the map file, notice the checksum symbol and its size, and for information about its placement, check the module map or the entry list. Compare that placement with the checksum range.
- Verify that the checksum calculations use the same polynomial.
- Verify that the bits in the bytes are processed in the same order in both checksum calculations, from the least to the most significant bit or the other way around. You control this with the **Bit order** option (or from the command line, the `-m` parameter of the `--checksum` option).
- If you are using the small variant of CRC, check whether you need to feed additional bytes into the algorithm.

  The number of zeros to add at the end of the byte sequence must match the size of the checksum, in other words, one zero for a 1-byte checksum, two zeros for a 2-byte checksum, and four zeros for a 4-byte checksum.
- Carefully consider the number of bits in pointers when crossing 64-Kbyte boundaries.

  For example, if the pointer `0xFFFF` is incremented by one, this typically results in the pointer `0x0` (and not `0x10000`) if the pointer is 16 bits.
- Any breakpoints in flash memory change the content of the flash. This means that the checksum which is calculated by your application will no longer match the initial checksum calculated by the linker. To make the two checksums match again, you must disable all your breakpoints in flash and any breakpoints set in flash by C-SPY internally. The stack plugin and the debugger option **Run to** both require C-SPY to set breakpoints. Read more about possible breakpoint consumers in the *C-SPY® Debugging Guide for 8051*.

Efficient coding for embedded applications

- Selecting data types
- Controlling data and function placement in memory
- Controlling compiler optimizations
- Facilitating good code generation

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 parameter to point to `const` data might open for better optimizations in the calling function.

For information about representation of supported data types, pointers, and structures types, see the chapter *Data representation*.

FLOATING-POINT TYPES

Using floating-point types on a microprocessor without a math coprocessor is very inefficient, both in terms of code size and execution speed. Thus, you should consider replacing code that uses floating-point operations with code that uses integers, because these are more efficient.

The compiler only supports the 32-bit floating-point format. The 64-bit floating-point format is not supported. The `double` type will be treated as a `float` type.

For more information about floating-point types, see *Basic data types—floating-point types*, page 329.

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 a pointer points to and taking the appropriate actions. Use the smallest pointer type that you can, and avoid any generic pointers unless necessary.

ANONYMOUS STRUCTS AND UNIONS

When a structure or union is declared without a name, it becomes anonymous. The effect is that its members will only be seen in the surrounding scope.

Anonymous structures are part of the C++ language; however, they are not part of the C standard. In the IAR C/C++ Compiler for 8051 they can be used in C if language extensions are enabled.

In the IDE, language extensions are enabled by default.

Use the `-e` compiler option to enable language extensions. See *-e*, page 302, for additional information.

Example

In this example, the members in the anonymous `union` can be accessed, in function `F`, without explicitly specifying the `union` name:

```
struct S
{
  char mTag;
  union
  {
    long mL;
    float mF;
  };
} St;

void F(void)
{
  St.mL = 5;
}
```

The member names must be unique in the surrounding scope. Having an anonymous `struct` or `union` at file scope, as a global, external, or static variable is also allowed. This could for instance be used for declaring I/O registers, as in this example:

```
__no_init __sfr volatile
union
{
  unsigned char IOPORT;
  struct
  {
    unsigned char way: 1;
    unsigned char out: 1;
  };
} @ 0x90;



/* The variables are used here. */
void Test(void)
{
  IOPORT = 0;
  way = 1;
  out = 1;
}
```

This declares an I/O register byte `IOPORT` at address `0x90`. The I/O register has 2 bits declared, `way` and `out`. Note that both the inner structure and the outer union are anonymous.

Anonymous structures and unions are implemented in terms of objects named after the first field, with a prefix `_A_` to place the name in the implementation part of the namespace. In this example, the anonymous union will be implemented through an object named `_A_IOPORT`.

Controlling data and function placement in memory

The 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 and know which one is best suited for different situations. You can use:

- Code models

  By selecting a code model, you can control the default memory placement of functions. For more information, see *Code models and memory attributes for function storage*, page 95.

- Data models

  By selecting a data model, you can control the default memory placement of variables and constants. For more information, see *Data models*, page 80.

- Memory attributes

  Using IAR-specific keywords or pragma directives, you can override the default addressing mode, and the default placement of functions and data objects. For more information, see *Using function memory attributes*, page 97 and *Using data memory attributes*, page 74, respectively.

- Calling convention

  The 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 198.

- Virtual registers

  In larger data models, fine-tuning the number of virtual registers can result in more efficient code; see *Virtual registers*, page 92.

- The `@` operator and the `#pragma location` directive for absolute placement.

  Using the `@` operator or the `#pragma location` directive, you can place individual global and static variables at absolute addresses. Note that it is not possible to use this notation for absolute placement of individual functions. For more information, see *Data placement at an absolute location*, page 260.

- The `@` operator and the `#pragma location` directive for segment placement.

  Using the `@` operator or the `#pragma location` directive, you can place individual functions, variables, and constants in named segments. The placement of these segments can then be controlled by linker directives. For more information, see *Data and function placement in segments*, page 262.

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 the `const` keyword (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: All declarations of variables placed at an absolute address are *tentative definitions*. Tentatively defined variables are only kept in the output from the compiler if they are needed in the module being compiled. Such variables will be defined in all modules in which they are used, which will work as long as they are defined in the same

way. The recommendation is to place all such declarations in header files that are included in all modules that use the variables.

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 @ 0xFF20;/* OK */
```

The next example contains two `const` declared objects. The first one is not initialized, and the second one is initialized to a specific value. Both objects are placed in ROM. This is useful for configuration parameters, which are accessible from an external interface. Note that in the second case, the compiler is not obliged to actually read from the variable, because the value is known.

```
#pragma location=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.

This 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 more than one variable is located at address `0x100`.

To avoid this problem and make the process the same in C and C++, you should declare these variables `extern`, for example:

```
/* The extern keyword makes x public. */
extern volatile const __no_init int x @ 0x100;
```

Note: C++ static member variables can be placed at an absolute address just like any other static variable.

DATA AND FUNCTION PLACEMENT IN SEGMENTS

The following method can be used for placing data or functions in named segments other than default:

- The `@` operator, alternatively the `#pragma location` directive, can be used for placing individual variables or individual functions in named segments other than the default one. The named segment can either be a predefined segment, or a user-defined segment.

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 configuration file using the `-Z` or the `-P` segment control directives.

Note: Take care when explicitly placing a variable or function in a predefined segment other than the one used by default. This is useful in some situations, but incorrect placement can result in anything from error messages during compilation and linking to a malfunctioning application. Carefully consider the circumstances; there might be strict requirements on the declaration and use of the function or variable.

The location of the segments can be controlled from the linker configuration file.

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

Examples of placing variables in named segments

In the following examples, a data object is placed in a user-defined segment. If no memory attribute is specified, the variable will, like any other variable, be treated as if it is located in the default memory. Note that you must as always ensure that the segment is placed in the appropriate memory area when linking.

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

#pragma location="MY_CONSTANTS"
const int beta = 42;               /* OK */

const int gamma @ "MY_CONSTANTS" = 17;/* OK */
int theta @ "MY_ZEROS";             /* OK */
int phi @ "MY_INITED" = 4711;       /* OK */
```

The compiler will warn that segments that contain zero-initialized and initialized data must be handled manually. To do this, you must use the linker option `-Q` to separate the initializers into one separate segment and the symbols to be initialized to a different segment. You must then write source code that copies the initializer segment to the initialized segment, and zero-initialized symbols must be cleared before they are used.

As usual, you can use memory attributes to select a memory for the variable. Note that you must as always ensure that the segment is placed in the appropriate memory area when linking.

```
__pdata __no_init int alpha @ "MY_PDATA_NOINIT";/* Placed in
                                                    pdata*/
```

This example shows incorrect usage:

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

Examples of placing functions in named segments

```
void f(void) @ "MY_FUNCTIONS";

void g(void) @ "MY_FUNCTIONS"
{
}

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

Specify a memory attribute to direct the function to a specific memory, and then modify the segment placement in the linker configuration file accordingly:

```
__ void f(void) @ "MY_NEAR_FUNC_FUNCTIONS";
```

Controlling compiler optimizations

The compiler performs many transformations on your application to generate the best possible code. Examples of such transformations are storing values in registers instead of memory, removing superfluous code, reordering computations in a more efficient order, and replacing arithmetic operations by cheaper operations.

The linker should also be considered an integral part of the compilation system, because some optimizations are performed by the linker. For instance, all unused functions and variables are removed and not included in the final output.

SCOPE FOR PERFORMED OPTIMIZATIONS

You can decide whether optimizations should be performed on your whole application or on individual files. By default, the same types of optimizations are used for an entire project, but you should consider using different optimization settings for individual files. For example, put code that must execute very quickly into a separate file and compile it for minimal execution time, and the rest of the code for minimal code size. This will give a small program, which is still fast enough where it matters.

You can also exclude individual functions from the performed optimizations. The `#pragma optimize` directive allows you to either lower the optimization level, or specify another type of optimization to be performed. See *optimize*, page 374, for information about the pragma directive.

MULTI-FILE COMPILATION UNITS

In addition to applying different optimizations to different source files or even functions, you can also decide what a compilation unit consists of—one or several source code files.

By default, a compilation unit consists of one source file, but you can also use multi-file compilation to make several source files in a compilation unit. The advantage is that interprocedural optimizations such as inlining, cross call, and cross jump have more source code to work on. Ideally, the whole application should be compiled as one compilation unit. However, for large applications this is not practical because of resource restrictions on the host computer. For more information, see *--mfc*, page 308.

Note: Only one object file is generated, and thus all symbols will be part of that object file.

If the whole application is compiled as one compilation unit, it is very useful to make the compiler also discard unused public functions and variables before the interprocedural optimizations are performed. Doing this limits the scope of the optimizations to functions and variables that are actually used. For more information, see *--discard_unused_publics*, page 299.

Multi-file compilation should not be used with any of the banked code models; see *Writing source code for banked memory*, page 111.

OPTIMIZATION LEVELS

The compiler supports different levels of optimizations. This table lists optimizations that are typically performed on each level:

Optimization level	Description
None (Best debug support)	Variables live through their entire scope Disabled register banks (needs to be enabled manually)
Low	Same as above but variables only live for as long as they are needed, not necessarily through their entire scope, and: Dead code elimination Redundant label elimination Redundant branch elimination
Medium	Same as above, and: Live-dead analysis and optimization Code hoisting Register content analysis and optimization Common subexpression elimination
High (Balanced)	Same as above, and: Peephole optimization Cross jumping Cross call (when optimizing for size or balanced) Loop unrolling Function inlining Code motion Type-based alias analysis

Table 31: 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 266.

A high level of optimization might result in increased compile time, and will most likely also make debugging more difficult, because it is less clear how the generated code relates to the source code. For example, at the low, medium, and high optimization levels, variables do not live through their entire scope, which means processor registers used for storing variables can be reused immediately after they were last used. Due to this, the C-SPY **Watch** window might not be able to display the value of the variable throughout its scope. At any time, if you experience difficulties when debugging your code, try lowering the optimization level.

SPEED VERSUS SIZE

At the high optimization level, the compiler balances between size and speed optimizations. However, it is possible to fine-tune the optimizations explicitly for either

size or speed. They only differ in what thresholds that are used; speed will trade size for speed, whereas size will trade speed for size. Note that one optimization sometimes enables other optimizations to be performed, and an application might in some cases become smaller even when optimizing for speed rather than size.

If you use the optimization level High speed, the `--no_size_constraints` compiler option relaxes the normal restrictions for code size expansion and enables more aggressive optimizations.

You can choose an optimization goal for each module, or even individual functions, using command line options and pragma directives (see *-O*, page 314 and *optimize*, page 374). For a small embedded application, this makes it possible to achieve acceptable speed performance while minimizing the code size: Typically, only a few places in the application need to be fast, such as the most frequently executed inner loops, or the interrupt handlers.

Rather than compiling the whole application with High (Balanced) optimization, you can use High (Size) in general, but override this to get High (Speed) optimization only for those functions where the application needs to be fast.

Because of the unpredictable way in which different optimizations interact, where one optimization can enable other optimizations, sometimes a function becomes smaller when compiled with High (Speed) optimization than if High (Size) is used. Also, using multi-file compilation (see *--mfc*, page 308) can enable many optimizations to improve both speed and size performance. It is recommended that you experiment with different optimization settings so that you can pick the best ones for your project.

FINE-TUNING ENABLED TRANSFORMATIONS

At each optimization level you can disable some of the transformations individually. To disable a transformation, use either the appropriate option, for instance the command line option `--no_inline`, alternatively its equivalent in the IDE **Function inlining**, or the `#pragma optimize` directive. These transformations can be enabled/disabled individually:

- Common subexpression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis
- Cross call
- Disabled register banks

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.

For more information about the command line option, see *--no_cse*, page 310.

Loop unrolling

Loop unrolling means that the code body of a loop, whose number of iterations can be determined at compile time, is duplicated. Loop unrolling reduces the loop overhead by amortizing it over several iterations.

This optimization is most efficient for smaller loops, where the loop overhead can be a substantial part of the total loop body.

Loop unrolling, which can be performed at optimization level High, normally reduces execution time, but increases code size. The resulting code might also be difficult to debug.

The compiler heuristically decides which loops to unroll. Only relatively small loops where the loop overhead reduction is noticeable will be unrolled. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed.

Note: This option has no effect at optimization levels None, Low, and Medium.

To disable loop unrolling, use the command line option `--no_unroll`, see *--no_unroll*, page 313.

Function inlining

Function inlining means that a 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 normally reduces execution time, but might increase the code size.

For more information, see *Inlining functions*, page 103.

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 Medium and above, 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 below Medium.

For more information about the command line option, see *--no_code_motion*, page 309.

Type-based alias analysis

When two or more pointers reference the same memory location, these pointers are said to be *aliases* for each other. The existence of aliases makes optimization more difficult because it is not necessarily known at compile time whether a particular value is being changed.

Type-based alias analysis optimization assumes that all accesses to an object are performed using its declared type or as a `char` type. This assumption lets the compiler detect whether pointers can reference the same memory location or not.

Type-based alias analysis is performed at optimization level High. For application code conforming to standard C or C++ application code, this optimization can reduce code size and execution time. However, non-standard 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.

For more information about the command line option, see *--no_tbaa*, page 312.

Example

```
short F(short *p1, long *p2)
{
  *p2 = 0;
  *p1 = 1;
  return *p2;
}
```

With type-based alias analysis, it is assumed that a write access to the `short` pointed to by `p1` cannot affect the `long` value that `p2` points to. Thus, it is known at compile time that this function returns 0. However, in non-standard-conforming C or C++ code these pointers could overlap each other by being part of the same union. If you use explicit casts, you can also force pointers of different pointer types to point to the same memory location.

Cross call

Common code sequences are extracted to local subroutines. This optimization, which is performed at optimization level High, can reduce code size, sometimes dramatically, on behalf of execution time and stack size. The resulting code might however be difficult to debug.

For more information about related command line options, see *--no_cross_call*, page 309.

Disabled register banks

If your application does not use register banks, you can disable the compiler's support for them, and open up for further optimizations. This transformation is never used by default, and is not automatically used when you set the appropriate optimization level. It must always be turned on explicitly.

Program modules compiled with register banks disabled cannot be linked together with modules that use register banks. The runtime attribute `__register_banks` helps you avoid mixing such modules: see *Predefined runtime attributes*, page 143.

For information about the command line option, see *--disable_register_banks*, page 299.

Facilitating good code generation

This section contains hints on how to help the compiler generate good code:

WRITING OPTIMIZATION-FRIENDLY SOURCE CODE

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 can modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. This is inefficient for two main reasons. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.

- Module-local variables—variables that are declared static—are preferred over global variables (non-static). Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions, see *Function inlining*, page 267. To maximize the effect of the inlining transformation, it is good practice to place the definitions of small functions called from more than one module in the header file rather than in the implementation file. Alternatively, you can use multi-file compilation. For more information, see *Multi-file compilation units*, page 264.
- Avoid using inline assembler. Instead, try writing the code in C/C++, use intrinsic functions, or write a separate module in assembler language. For more information, see *Mixing C and assembler*, page 191.

SAVING STACK SPACE AND RAM MEMORY

The following is a list of programming techniques that save memory and stack space:

- If stack space is limited, avoid long call chains and recursive functions.
- Avoid using large non-scalar types, such as structures, as parameters or return type. To save stack space, you should instead pass them as pointers or, in C++, as references.
- 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 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 86.

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 valid C, however it is strongly recommended to use the prototyped style, and provide a prototype declaration for each public function in a header that is included both in the compilation unit defining the function and in all compilation units using it.

The compiler will not perform type checking on parameters passed to functions declared using K&R style. Using prototype declarations will also result in more efficient code in some cases, as there is no need for type promotion for these functions.

To make the compiler require that all function definitions use the prototyped style, and that all public functions have been declared before being defined, use the **Project>Options>C/C++ Compiler>Language 1>Require prototypes** compiler option (`--require_prototypes`).

Prototyped style

In prototyped function declarations, the type for each parameter must be specified.

```
int Test(char, int); /* Declaration */

int Test(char ch, int i) /* Definition */
{
  return i + ch;
}
```

Kernighan & Ritchie style

In K&R style—pre-Standard C—it is not possible to declare a function prototyped. Instead, an empty parameter list is used in the function declaration. Also, the definition looks different.

For example:

```
int Test();     /* Declaration */

int Test(ch, i) /* Definition */
char ch;
int i;
{
  return i + ch;
}
```

INTEGER TYPES AND BIT NEGATION

In some situations, the rules for integer types and their conversion lead to possibly confusing behavior. Things to look out for are assignments or conditionals (test expressions) involving types with different size, and logical operations, especially bit negation. Here, *types* also includes types of constants.

In some cases there might be warnings (for example, for constant conditional or pointless comparison), in others just a different result than what is expected. Under certain circumstances the compiler might warn only at higher optimizations, for example, if the compiler relies on optimizations to identify some instances of constant conditionals. In this example an 8-bit character, a 16-bit integer, and two's complement is assumed:

```
void F1(unsigned char c1)
{
  if (c1 == ~0x80)
    ;
}
```

Here, the test is always false. On the right hand side, `0x80` is `0x0080`, and `~0x0080` becomes `0xFF7F`. On the left hand side, `c1` is an 8-bit unsigned character, so it cannot be larger than 255. It also cannot be negative, which means that the integral promoted value can never have the topmost 8 bits set.

PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

Variables that are accessed asynchronously, for example by interrupt routines or by code executing in separate threads, must be properly marked and have adequate protection. The only exception to this is a variable that is always *read-only*.

To mark a variable properly, use the `volatile` keyword. This informs the compiler, among other things, that the variable can be changed from other threads. The compiler will then avoid optimizing on the variable (for example, keeping track of the variable in registers), will not delay writes to it, and be careful accessing the variable only the number of times given in the source code.

For sequences of accesses to variables that you do not want to be interrupted, use the `__monitor` keyword. This must be done for both write *and* read sequences, otherwise you might end up reading a partially updated variable. Accessing a small-sized `volatile` variable can be an atomic operation, but 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. For more information, see *__monitor*, page 354.

For more information about the `volatile` type qualifier and the rules for accessing volatile objects, see *Declaring objects volatile*, page 334.

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 several 8051 devices are included in the IAR 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. This example is from `io8051.h`:

```
__no_init volatile union
{
  unsigned char PSW;
  struct
  {
    unsigned char P : 1;
    unsigned char F1 : 1;
    unsigned char OV : 1;
    unsigned char RS0 : 1;
    unsigned char RS1 : 1;
    unsigned char F0 : 1;
    unsigned char AC : 1;
    unsigned char CY : 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.
 */

void Test()
{
  /* Whole register access */
  PSW = 0x12;

  /* Bitfield accesses */
  PSW_bit.AC  = 1;
  PSW_bit.RS0 = 0;
}
```

You can also use the header files as templates when you create new header files for other 8051 devices. For information about the @ operator, see *Placing located data*, page 132.

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 a separate segment, according to the specified memory keyword. See the chapter *Linking overview* 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 more information, see *__no_init*, page 356. Note that to use this keyword, language extensions must be enabled; see *-e*, page 302. For more information, see also *object_attribute*, page 373.

Part 2. Reference information

This part of the *IAR C/C++ Compiler User Guide for 8051* contains these chapters:

- External interface details
- Compiler options
- Data representation
- Extended keywords
- Pragma directives
- Intrinsic functions
- The preprocessor
- C/C++ standard library functions
- Segment reference
- Implementation-defined behavior for Standard C
- Implementation-defined behavior for C89.

External interface details

- Invocation syntax
- Include file search procedure
- Compiler output
- Diagnostics

Invocation syntax

You can use the compiler either from the IDE or from the command line. See the *IDE Project Management and Building Guide for 8051* for information about using the compiler from the 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 this command to generate an object file with debug information:

```
icc8051 prog.c --debug
```

The source file can be a C or C++ file, typically with the filename extension `c` or `cpp`, respectively. If no filename extension is specified, the file to be compiled must have the extension `c`.

Generally, the order of options on the command line, both relative to each other and to the source filename, is not significant. There is, however, one exception: when you use the `-I` option, the directories are searched in the same order as 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 279.

- Via environment variables

  The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 280.

- Via a text file, using the `-f` option; see *-f*, page 305.

For general guidelines for the option syntax, an options summary, and a detailed description of each option, see the chapter *Compiler options*.

ENVIRONMENT VARIABLES

These environment variables can be used with the compiler:

Environment variable	Description
`C_INCLUDE`	Specifies directories to search for include files; for example: `C_INCLUDE=c:\program files\iar systems\embedded workbench 8.`*n*`\8051\inc;c:\headers`
`QCCX51`	Specifies command line options; for example: `QCCX51=-lA asm.lst`

Table 32: 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 specified in angle brackets or double quotes, that file is opened.
- If the compiler encounters the name of an `#include` file in angle brackets, such as:

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

  it searches these directories for the file to include:

  1 The directories specified with the `-I` option, in the order that they were specified, see *-I*, page 306.

  2 The directories specified using the `C_INCLUDE` environment variable, if any; see *Environment variables*, page 280.

  3 The automatically set up library system include directories. See *--clib*, page 292, *--dlib*, page 300, and *--dlib_config*, page 300.

- If the compiler encounters the name of an `#include` file in double quotes, for example:

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

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

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

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

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

```
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.

For more information, see *Overview of the preprocessor*, page 387.

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

  Various kinds of list files can be specified using the compiler option `-l`, see *-l*, page 306. By default, these files will have the filename extension `lst`.

- Optional preprocessor output files

  A preprocessor output file is produced when you use the `--preprocess` option; by default, the file will have the filename extension `i`.

- Diagnostic messages

  Diagnostic messages are directed to the standard error stream and displayed on the screen, and printed in an optional list file. For more information about diagnostic messages, see *Diagnostics*, page 283.

- 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 282.

- Size information

  Information about the generated amount of bytes for functions and data for each memory is directed to the standard output stream and displayed on the screen. Some of the bytes might be reported as *shared*.

  Shared objects are functions or data objects that are shared between modules. If any of these occur in more than one module, only one copy is retained. For example, in some cases inline functions are not inlined, which means that they are marked as shared, because only one instance of each function will be included in the final application. This mechanism is sometimes also used for compiler-generated code or data not directly associated with a particular function or variable, and when only one instance is required in the final application.

ERROR RETURN CODES

The compiler returns status information to the operating system that can be tested in a batch file.

These command line error codes are supported:

Code	Description
0	Compilation successful, but there might have been warnings.
1	Warnings were produced and the option `--warnings_affect_exit_code` was used.
2	Errors occurred.
3	Fatal errors occurred, making the compiler abort.
4	Internal errors occurred, making the compiler abort.

Table 33: Error return codes

Diagnostics

This section describes the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.

MESSAGE FORMAT

All diagnostic messages are issued as complete, self-explanatory messages. A typical diagnostic message from the compiler is produced in the form:

filename,*linenumber* *level*[*tag*]: *message*

with these elements:

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 construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see *--remarks*, page 319.

Warning

A diagnostic message that is produced when the compiler finds a potential programming error or omission which is of concern, but which does not prevent completion of the compilation. Warnings can be disabled by use of the command line option `--no_warnings`, see *--no_warnings*, page 313.

Error

A diagnostic message that is produced when the compiler finds a construct which clearly violates the C or C++ language rules, such that code cannot be produced. An error will produce a non-zero exit code.

Fatal error

A diagnostic message that is produced when the compiler finds a condition that not only prevents code generation, but which makes further processing of the source code pointless. After the message is issued, compilation terminates. A fatal error will produce a non-zero exit code.

SETTING THE SEVERITY LEVEL

The diagnostic messages can be suppressed or the severity level can be changed for all diagnostics messages, except for fatal errors and some of the regular errors.

See the chapter *Compiler options*, for information about the compiler options that are available for setting severity levels.

See the chapter *Pragma directives*, for information about the pragma directives that are available for setting severity levels.

INTERNAL ERROR

An internal error is a diagnostic message that signals that there was a serious and unexpected failure due to a fault in the compiler. It is produced using this form:

```
Internal error: message
```

where *message* is an explanatory message. If internal errors occur, they should be reported to your software distributor or IAR Systems Technical Support. Include enough information to reproduce the problem, typically:

- The product name
- The version number of the compiler, which can be seen in the header of the list files generated by the compiler
- Your license number
- The exact internal error message text
- The source file of the application that generated the internal error
- A list of the options that were used when the internal error occurred.

Compiler options

- Options syntax
- Summary of compiler options
- Descriptions of compiler options

Options syntax

Compiler options are parameters you can specify to change the default behavior of the compiler. You can specify options from the command line—which is described in more detail in this section—and from within the IDE.

See the online help system for information about the compiler options available in the IDE and how to set them.

TYPES OF OPTIONS

There are two *types of names* for command line options, *short* names and *long* names. Some options have both.

- A short option name consists of one character, and it can have parameters. You specify it with a single dash, for example `-e`
- A long option name consists of one or several words joined by underscores, and it can have parameters. You specify it with double dashes, for example `--char_is_signed`.

For information about the different methods for passing options, see *Passing options*, page 280.

RULES FOR SPECIFYING PARAMETERS

There are some general syntax rules for specifying option parameters. First, the rules depending on whether the parameter is *optional* or *mandatory*, and whether the option has a short or a long name, are described. Then, the rules for specifying filenames and directories are listed. Finally, the remaining rules are listed.

Rules for optional parameters

For options with a short name and an optional parameter, any parameter should be specified without a preceding space, for example:

`-O` or `-Oh`

For options with a long name and an optional parameter, any parameter should be specified with a preceding equal sign (=), for example:

```
--misrac2004=n
```

Rules for mandatory parameters

For options with a short name and a mandatory parameter, the parameter can be specified either with or without a preceding space, for example:

`-I..\src` or `-I ..\src\`

For options with a long name and a mandatory parameter, the parameter can be specified either with a preceding equal sign (=) or with a preceding space, for example:

```
--diagnostics_tables=MyDiagnostics.lst
```

or

```
--diagnostics_tables MyDiagnostics.lst
```

Rules for options with both optional and mandatory parameters

For options taking both optional and mandatory parameters, the rules for specifying the parameters are:

- For short options, optional parameters are specified without a preceding space
- For long options, optional parameters are specified with a preceding equal sign (=)
- For short and long options, mandatory parameters are specified with a preceding space.

For example, a short option with an optional parameter followed by a mandatory parameter:

```
-lA MyList.lst
```

For example, a long option with an optional parameter followed by a mandatory parameter:

```
--preprocess=n PreprocOutput.lst
```

Rules for specifying a filename or directory as parameters

These rules apply for options taking a filename or directory as parameters:

- Options that take a filename as a parameter can optionally take a file 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.c -l ..\listings\List.lst
  ```

- For options that take a filename as the destination for output, the parameter can be specified as a path without a specified filename. The compiler stores the output in that directory, in a file with an extension according to the option. The filename will be the same as the name of the compiled source file, unless a different name was specified with the option `-o`, in which case that name is used. For example:

  ```
  icc8051 prog.c -l ..\listings\
  ```

  The produced list file will have the default name `..\listings\prog.lst`
- The *current directory* is specified with a period (`.`). For example:

  ```
  icc8051 prog.c -l .
  ```
- `/` can be used instead of `\` as the directory delimiter.
- By specifying `-`, input files and output files can be redirected to the standard input and output stream, respectively. For example:

  ```
  icc8051 prog.c -l -
  ```

Additional rules

These rules also apply:

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

  ```
  icc8051 prog.c -l ---r
  ```
- For options that accept multiple arguments of the same type, the arguments can be provided as a comma-separated list (without a space), for example:

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

  Alternatively, the option can be repeated for each argument, for example:

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

Summary of compiler options

This table summarizes the compiler command line options:

Command line option	Description
`--c89`	Specifies the C89 dialect
`--calling_convention`	Specifies the calling convention
`--char_is_signed`	Treats `char` as signed
`--char_is_unsigned`	Treats `char` as unsigned
`--clib`	Uses the system include files for the CLIB library

Table 34: Compiler options summary

Command line option	Description
`--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
`--disable_register_banks`	Disables the compiler's use of register banks
`--discard_unused_publics`	Discards unused public symbols
`--dlib`	Uses the system include files for the DLIB library
`--dlib_config`	Uses the system include files for the DLIB library and determines which configuration of the library to use
`--dptr`	Enables support for multiple data pointer
`-e`	Enables language extensions
`--ec++`	Specifies Embedded C++
`--eec++`	Specifies Extended Embedded C++
`--enable_multibytes`	Enables support for multibyte characters in source files
`--enable_restrict`	Enables the Standard C keyword `restrict`
`--error_limit`	Specifies the allowed number of errors before compilation stops
`--extended_stack`	Specifies the use of an extended stack
`-f`	Extends the command line
`--guard_calls`	Enables guards for function static variable initialization
`--has_cobank`	Informs the compiler that the device has COBANK bits in the bank selection register for constants

Table 34: Compiler options summary (Continued)

Command line option	Description
--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
--macro_positions_in_diagnostics	Obtains positions inside macros in diagnostic messages
--mfc	Enables multi-file compilation
--misrac1998	Enables error messages specific to MISRA-C:1998. See the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.
--misrac2004	Enables error messages specific to MISRA-C:2004. See the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
--misrac_verbose	*IAR Embedded Workbench® MISRA C:1998 Reference Guide* or the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
--module_name	Sets the object module name
--no_call_frame_info	Disables output of call frame information
--no_code_motion	Disables code motion optimization
--no_cross_call	Disables cross-call 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_size_constraints	Relaxes the normal restrictions for code size expansion when optimizing for speed.
--no_static_destruction	Disables destruction of C++ static variables at program exit
--no_system_include	Disables the automatic search for system include files
--no_tbaa	Disables type-based alias analysis
--no_typedefs_in_diagnostics	Disables the use of typedef names in diagnostics
--no_ubrof_messages	Excludes messages from UBROF files
--no_unroll	Disables loop unrolling

Table 34: Compiler options summary (Continued)

Command line option	Description
`--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. Alias for `--output`.
`--omit_types`	Excludes type information
`--only_stdout`	Uses standard output only
`--output`	Sets the object filename
`--pending_instantiations`	Sets the maximum number of instantiations of a given C++ template.
`--place_constants`	Specifies the location of constants and strings
`--predef_macros`	Lists the predefined symbols.
`--preinclude`	Includes an include file before reading the source file
`--preprocess`	Generates preprocessor output
`--public_equ`	Defines a global named assembler label
`-r`	Generates debug information. Alias for `--debug`.
`--relaxed_fp`	Relaxes the rules for optimizing floating-point expressions
`--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`	Checks for strict compliance with Standard C/C++
`--system_include_dir`	Specifies the path for system include files
`--use_c++_inline`	Uses C++ inline semantics in C99
`--version`	Sends compiler output to the console and then exits.
`--vla`	Enables C99 VLA support
`--warn_about_c_style_casts`	Makes the compiler warn when C-style casts are used in C++ source code

Table 34: Compiler options summary (Continued)

Command line option	Description
`--warnings_affect_exit_code`	Warnings affect exit code
`--warnings_are_errors`	Warnings are treated as errors

Table 34: Compiler options summary (Continued)

Descriptions of compiler options

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

Note that if you use the options page **Extra Options** to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--c89

Syntax

```
--c89
```

Description

Use this option to enable the C89 C dialect instead of Standard C.

Note: This option is mandatory when the MISRA C checking is enabled.

See also

C language overview, page 221.

Project>Options>C/C++ Compiler>Language 1>C dialect>C89

--calling_convention

Syntax

```
--calling_convention=convention
```

Parameters

convention is one of:

```
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

Calling convention, page 197.

Project>Options>General Options>Target>Calling convention

--char_is_signed

Syntax

```
--char_is_signed
```

Description

By default, the compiler interprets the plain `char` type as unsigned. Use this option to make the compiler interpret the plain `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 and cannot be used with code that is compiled with this option.

Project>Options>C/C++ Compiler>Language 2>Plain 'char' is

--char_is_unsigned

Syntax

```
--char_is_unsigned
```

Description

Use this option to make the compiler interpret the plain `char` type as unsigned. This is the default interpretation of the plain `char` type.

Project>Options>C/C++ Compiler>Language 2>Plain 'char' is

--clib

Syntax

```
--clib
```

Description

Use this option to use the system header files for the CLIB library; the compiler will automatically locate the files and use them when compiling.

Note: The CLIB library is used by default for CLIB projects. To use the DLIB library, use the `--dlib` or the `--dlib_config` option instead.

See also

--dlib, page 300 and *--dlib_config*, page 300.

To set related options, choose:

Project>Options>General Options>Library Configuration

--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 generated. If you do not select a code model, the compiler uses the default code model. Note that all modules of your application must use the same code model.

See also

Code models and memory attributes for function storage, page 95.

Project>Options>General Options>Target>Code model

--core

Syntax

```
--core={plain|pl|extended1|e1|extended2|e2}
```

Parameters

`plain|pl`	Corresponds to a classic 8051 core with a 64-Kbyte address area of ROM or a classic 8051/8052 core with code memory expanded with up to 256 banks of additional ROM.
`extended1|e1`	Corresponds to a core with up to 16 Mbytes of external continuous data and code memory.

`extended2|e2`	Corresponds to a core with an extended addressing mechanism that can extend code memory with up to sixteen 64-Kbytes banks.

Description

Use this option to select the processor core for which the code will be generated. If you do not use the option to specify a core, the compiler uses the Plain core as default. Note that all modules of your application must use the same core.

The compiler supports the different 8051 microcontroller cores and devices based on these cores. The object code that the compiler generates for the different cores is not binary compatible.

See also

Understanding memory architecture, page 59.

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
```

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|far_generic|fg|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
`far_generic|fg`	Default memory attribute `__far22`
	Default pointer attribute `__generic`
`generic|g`	Default memory attribute `__xdata`
	Default pointer attribute `__generic`

Description

Use this option to select the data model, which means a default placement of data objects. If you do not select a data model, 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 80.

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|n][s]] {filename|directory|+}
```

Parameters

`i` (default)	Lists only the names of files
`m`	Lists in makefile style (multiple rules)
`n`	Lists in makefile style (one rule)
`s`	Suppresses system files
`+`	Gives the same output as `-o`, but with the filename extension `d`

See also *Rules for specifying a filename or directory as parameters*, page 286.

Description

Use this option to make the compiler list the names of all source and header files opened for input into a file with the default filename extension `i`.

Example

If `--dependencies` or `--dependencies=i` is used, the name of each opened input 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 is in makefile style. For each input 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 an input 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 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

See *Rules for specifying a filename or directory as parameters*, page 286.

Description

Use this option to list all possible diagnostic messages to 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.

Typically, this option cannot be given together with other options.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--disable_register_banks

Syntax

```
--disable_register_banks
```

Description

Use this option to disable the compiler's use of register banks.

See also

Disabled register banks, page 269.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Disabled register banks

--discard_unused_publics

Syntax

```
--discard_unused_publics
```

Description

Use this option to discard unused public functions and variables when compiling with the `--mfc` compiler option.

Note: Do not use this option only on parts of the application, as necessary symbols might be removed from the generated output. Use the object attribute `__root` to keep symbols that are used from outside the compilation unit, for example interrupt handlers. If the symbol does not have the `__root` attribute and is defined in the library, the library definition will be used instead.

See also

--mfc, page 308 and *Multi-file compilation units*, page 264.

Project>Options>C/C++ Compiler>Discard unused publics

--dlib

Syntax

```
--dlib
```

Description

Use this option to use the system header files for the DLIB library; the compiler will automatically locate the files and use them when compiling.

Note: The DLIB library is used by default: To use the CLIB library, use the `--clib` option instead.

See also

--dlib_config, page 300, *--no_system_include*, page 311, *--system_include_dir*, page 321, and *--clib*, page 292.

To set related options, choose:

Project>Options>General Options>Library Configuration

--dlib_config

Syntax

```
--dlib_config filename.h|config
```

Parameters

filename	A DLIB configuration header file, see *Rules for specifying a filename or directory as parameters*, page 286.
config	The default configuration file for the specified configuration will be used. Choose between: `none`, no configuration will be used `tiny`, the tiny library configuration will be used `normal`, the normal library configuration will be used (default) `full`, the full library configuration will be used.

Description

Use this option to specify which library configuration to use, either by specifying an explicit file or by specifying a library configuration—in which case the default file for that library configuration will be used. Make sure that you specify a configuration that corresponds to the library you are using. If you do not specify this option, the default library configuration file will be used.

All prebuilt runtime libraries are delivered with corresponding configuration files. You can find the library object files and the library configuration files in the directory

8051\lib. For examples and information about prebuilt runtime libraries, see *Prebuilt runtime libraries*, page 156.

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 *Customizing and building your own runtime library*, page 153.

Note: This option only applies to the IAR DLIB runtime environment.

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 Extended1 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 65. Default for the Extended1 core.

Description

Use this option to enable support for more than one data pointer; a feature in many 8051 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 and memory attributes for function storage, page 95.

Project>Options>General Options>Target>Data Pointer

-e

Syntax

```
-e
```

Description

In the command line version of the compiler, language extensions are disabled by default. If you use language extensions such as extended keywords and anonymous structs and unions in your source code, you must use this option to enable them.

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

See also

Enabling language extensions, page 223.

Project>Options>C/C++ Compiler>Language 1>Standard with IAR extensions

Note: By default, this option is selected in the IDE.

--ec++

Syntax

```
--ec++
```

Description

In the compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.

Project>Options>C/C++ Compiler>Language 1>C++

and

Project>Options>C/C++ Compiler>Language 1>C++ dialect>Embedded C++

--eec++

Syntax

```
--eec++
```

Description

In the compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++.

See also

Extended Embedded C++, page 230.

Project>Options>C/C++ Compiler>Language 1>C++

and

Project>Options>C/C++ Compiler>Language 1>C++ dialect>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 2>Enable multibyte support

--enable_restrict

Syntax

```
--enable_restrict
```

Description

Enables the Standard C keyword `restrict`. This option can be useful for improving analysis precision during optimization.

To set this option, use **Project>Options>C/C++ Compiler>Extra options**

--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 IDE.

--extended_stack

Syntax

```
--extended_stack
```

Description

Use this option to enable the extended stack that is available if you use an 8051 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 and memory attributes for function storage, page 95.

Project>Options>General Options>Target>Do not use extended stack

-f

Syntax

```
-f filename
```

Parameters

See *Rules for specifying a filename or directory as parameters*, page 286.

Description

Use this option to make the compiler read 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**.

--guard_calls

Syntax

```
--guard_calls
```

Description

Use this option to enable guards for function static variable initialization. This option should be used in a threaded C++ environment.

Note: This option requires a threaded C++ environment, which is not supported in the IAR C/C++ Compiler for 8051.

This option is not available in the IDE.

--has_cobank

Syntax

```
--has_cobank
```

Description

Use this option to inform the compiler that the device you are using has COBANK bits in the bank selection register for constants. In particular, this is important if it is a Silicon Laboratories C8051F120 device.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--header_context

Syntax

```
--header_context
```

Description

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

This option is not available in the IDE.

-I

Syntax

```
-I path
```

Parameters

path	The search path for `#include` files

Description

Use this option to specify the search paths for `#include` files. This option can be used more than once on the command line.

See also

Include file search procedure, page 280.

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` (default)	Assembler list file
`A`	Assembler list file with C or C++ source as comments
`b`	Basic assembler list file. This file has the same contents as a list file produced with `-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.

See also *Rules for specifying a filename or directory as parameters*, page 286.

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

--macro_positions_in_diagnostics

Syntax

```
--macro_positions_in_diagnostics
```

Description

Use this option to obtain position references inside macros in diagnostic messages. This is useful for detecting incorrect source code constructs in macros.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--mfc

Syntax

```
--mfc
```

Description

Use this option to enable *multi-file compilation*. This means that the compiler compiles one or several source files specified on the command line as one unit, which enhances interprocedural optimizations.

Note: The compiler will generate one object file per input source code file, where the first object file contains all relevant data and the other ones are empty. If you want only the first file to be produced, use the `-o` compiler option and specify a certain output file.

Example

```
icc8051 myfile1.c myfile2.c myfile3.c --mfc
```

See also

--discard_unused_publics, page 299, *--output, -o*, page 315, and *Multi-file compilation units*, page 264.

Project>Options>C/C++ Compiler>Multi-file compilation

--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_call_frame_info

Syntax

```
--no_call_frame_info
```

Description

Normally, the compiler always generates call frame information in the output, to enable the debugger to display the call stack even in code from modules with no debug information. Use this option to disable the generation of call frame information.

See also

Call frame information, page 211.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_code_motion

Syntax

```
--no_code_motion
```

Description

Use this option to disable code motion optimizations.

Note: This option has no effect at optimization levels below Medium.

See also

Code motion, page 267.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Code motion

--no_cross_call

Syntax

```
--no_cross_call
```

Description

Use this option to disable the cross-call optimization.

Note: This option has no effect at optimization levels below High.

See also

Cross call, page 268.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Cross call

--no_cse

Syntax

```
--no_cse
```

Description

Use this option to disable common subexpression elimination.

Note: This option has no effect at optimization levels below Medium.

See also

Common subexpression elimination, page 267.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination

--no_inline

Syntax

```
--no_inline
```

Description

Use this option to disable function inlining.

See also

Inlining functions, page 103.

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

Description of predefined preprocessor symbols, page 388.

This option is not available in the IDE.

--no_size_constraints

Syntax

```
--no_size_constraints
```

Description

Use this option to relax the normal restrictions for code size expansion when optimizing for high speed.

Note: This option has no effect unless used with `-Ohs`.

See also

Speed versus size, page 265.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>No size constraints

--no_static_destruction

Syntax

```
--no_static_destruction
```

Description

Normally, the compiler emits code to destroy C++ static variables that require destruction at program exit. Sometimes, such destruction is not needed.

Use this option to suppress the emission of such code.

See also

System termination, page 166.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_system_include

Syntax

```
--no_system_include
```

Description

By default, the compiler automatically locates the system include files. Use this option to disable the automatic search for system include files. In this case, you might need to set up the search path by using the `-I` compiler option.

See also

--dlib, page 300, *--dlib_config*, page 300, and *--system_include_dir*, page 321.

Project>Options>C/C++ Compiler>Preprocessor>Ignore standard include directories

--no_tbaa

Syntax

```
--no_tbaa
```

Description

Use this option to disable type-based alias analysis.

Note: This option has no effect at optimization levels below High.

See also

Type-based alias analysis, page 268.

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 = "My text string";
```

will give an error message like this:

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

If the `--no_typedefs_in_diagnostics` option is used, the error message will be like this:

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

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_ubrof_messages

Syntax

```
--no_ubrof_messages
```

Description

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

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.

Note: This option has no effect at optimization levels below High.

See also

Loop unrolling, page 267.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling

--no_warnings

Syntax

```
--no_warnings
```

Description

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

This option is not available in the IDE.

--no_wrap_diagnostics

Syntax

```
--no_wrap_diagnostics
```

Description

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

This option is not available in the IDE.

--nr_virtual_regs

Syntax

```
--nr_virtual_regs=n
```

Parameter

n	The size of the work area (number of virtual registers); a value between 8 and 32.

Description

Use this option to specify the number of virtual registers, which determines the size of the work area. The virtual registers are located in data memory.

See also

Virtual registers, page 92.

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 263.

Project>Options>C/C++ Compiler>Optimizations

--omit_types

Syntax

```
--omit_types
```

Description

By default, the compiler includes type information about variables and functions in the object output. Use this option if you do not want the compiler to include this type information in the output, which is useful when you build a library that should not contain type information. The object file will then only contain type information that is a part of a symbol's name. This means that the linker cannot check symbol references for type correctness.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--only_stdout

Syntax

```
--only_stdout
```

Description

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

This option is not available in the IDE.

--output, -o

Syntax

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

Parameters

See *Rules for specifying a filename or directory as parameters*, page 286.

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 IDE.

--pending_instantiations

Syntax

```
--pending_instantiations number
```

Parameters

number	An integer that specifies the limit, where `64` is default. If `0` is used, there is no limit.

Description

Use this option to specify the maximum number of instantiations of a given C++ template that is allowed to be in process of being instantiated at a given time. This is used for detecting recursive instantiations.

Project>Options>C/C++ Compiler>Extra Options

--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 Large 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.

If you locate constants and strings in code memory, you might want to use some 8051–specific CLIB library function variants that allow access to strings in code memory.

See also

Constants and strings, page 82 and *8051-specific CLIB functions*, page 404.

Project>Options>General Options>Target>Location for constants and strings

Project>Options>Linker>Output>Output file

--predef_macros

Syntax

```
--predef_macros {filename|directory}
```

Parameters

See *Rules for specifying a filename or directory as parameters*, page 286.

Description

Use this option to list the predefined symbols. When using this option, make sure to also use the same options as for the rest of your project.

If a filename is specified, the compiler stores the output in that file. If a directory is specified, the compiler stores the output in that directory, in a file with the `predef` filename extension.

Note that this option requires that you specify a source file on the command line.

This option is not available in the IDE.

--preinclude

Syntax

```
--preinclude includefile
```

Parameters

See *Rules for specifying a filename or directory as parameters*, page 286.

Description

Use this option to make the compiler read 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

See also *Rules for specifying a filename or directory as parameters*, page 286.

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 can be used more than once on the command line.

This option is not available in the IDE.

--relaxed_fp

Syntax

```
--relaxed_fp
```

Description

Use this option to allow the compiler to relax the language rules and perform more aggressive optimization of floating-point expressions. This option improves performance for floating-point expressions that fulfill these conditions:

- The expression consists of both single- and double-precision values
- The double-precision values can be converted to single precision without loss of accuracy
- The result of the expression is converted to single precision.

Note that performing the calculation in single precision instead of double precision might cause a loss of accuracy.

Example

```
float F(float a, float b)
{
  return a + b * 3.0;
}
```

The C standard states that `3.0` in this example has the type `double` and therefore the whole expression should be evaluated in `double` precision. However, when the

`--relaxed_fp` option is used, 3.0 will be converted to `float` and the whole expression can be evaluated in `float` precision.

To set related options, choose:

Project>Options>C/C++ Compiler>Language 2>Floating-point semantics

--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 283.

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.

Project>Options>C/C++ Compiler>Language 1>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 IDE.

--strict

Syntax

```
--strict
```

Description

By default, the compiler accepts a relaxed superset of Standard C and C++. Use this option to ensure that the source code of your application instead conforms to strict Standard C and C++.

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

See also

Enabling language extensions, page 223.

Project>Options>C/C++ Compiler>Language 1>Language conformance>Strict

--system_include_dir

Syntax

```
--system_include_dir path
```

Parameters

path	The path to the system include files, see *Rules for specifying a filename or directory as parameters*, page 286.

Description

By default, the compiler automatically locates the system include files. Use this option to explicitly specify a different path to the system include files. This might be useful if you have not installed IAR Embedded Workbench in the default location.

See also

--dlib_config, page 300, and *--no_system_include*, page 311.

This option is not available in the IDE.

--use_c++_inline

Syntax

```
--use_c++_inline
```

Description

Standard C uses slightly different semantics for the `inline` keyword than C++ does. Use this option if you want C++ semantics when you are using C.

See also

Inlining functions, page 103

Project>Options>C/C++ Compiler>Language 1>C dialect>C99>C++ inline semantics

--version

Syntax

```
--version
```

Description

Use this option to make the compiler send version information to the console and then exit.

This option is not available in the IDE.

--vla

Syntax

```
--vla
```

Description

Use this option to enable support for C99 variable length arrays. Such arrays are located on the heap. This option requires Standard C and cannot be used together with the `--c89` compiler option.

Note: `--vla` should not be used together with the `longjmp` library function, as that can lead to memory leakages.

See also

C language overview, page 221.

Project>Options>C/C++ Compiler>Language 1>C dialect>Allow VLA

--warn_about_c_style_casts

Syntax

```
--warn_about_c_style_casts
```

Description

Use this option to make the compiler warn when C-style casts are used in C++ source code.

This option is not available in the IDE.

--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 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 298.

Project>Options>C/C++ Compiler>Diagnostics>Treat all warnings as errors

Data representation

- Alignment
- Basic data types—integer types
- Basic data types—floating-point types
- Pointer types
- Structure types
- Type qualifiers
- Data types in C++

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.

All data types must have a size that is a multiple of their alignment. Otherwise, only the first element of an array would be guaranteed to be placed in accordance with the alignment requirements. This means that the compiler might add pad bytes at the end of the structure.

Note that with the `#pragma data_alignment` directive you can increase the alignment demands on specific variables.

ALIGNMENT ON THE 8051 MICROCONTROLLER

The 8051 microcontroller does not have any alignment restrictions.

Basic data types—integer types

The compiler supports both all Standard C basic data types and some additional types.

INTEGER TYPES—AN OVERVIEW

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

Data type	Size	Range	Alignment
`bool`	8 bits	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
`signed long long`	32 bits	-2^{31} to $2^{31}-1$	1
`unsigned long long`	32 bits	0 to $2^{32}-1$	1

Table 35: 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`, `unsigned long`, `long long`, or `unsigned long long`.

To make the compiler use a larger type than it would automatically use, define an `enum` constant with a large enough value. For example:

```
/* Disables usage of the char type for enum */
enum Cards{Spade1, Spade2,
           DontUseChar=257};
```

THE CHAR TYPE

The `char` type is by default unsigned in the 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 Standard C, `int`, `signed int`, and `unsigned int` can be used as the base type for integer bitfields. In standard C++, and in C when language extensions are enabled in the compiler, any integer or enumeration type can be used as the base type. It is implementation defined whether a plain integer type (`char`, `short`, `int`, etc) results in a signed or unsigned bitfield.

In the IAR C/C++ Compiler for 8051, plain integer types are treated as signed.

Bitfields in expressions are treated as `int` if `int` can represent all values of the bitfield. Otherwise, they are treated as the bitfield 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.

Each bitfield is placed into a container of its base type from the least significant bit to the most significant bit. If the last container is of the same type and has enough bits available, the bitfield is placed into this container, otherwise a new container is allocated. This allocation scheme is referred to as the disjoint type bitfield allocation.

If you use the directive `#pragma bitfields=reversed`, bitfields are placed from the most significant bit to the least significant bit in each container. See *bitfields*, page 363.

Assume this example:

```
struct BitfieldExample
{
  uint32_t a : 12;
  uint16_t b : 3;
  uint16_t c : 7;
  uint8_t  d;
};
```

The example in the disjoint types bitfield allocation strategy

To place the first bitfield, `a`, the compiler allocates a 32-bit container at offset `0` and puts `a` into the least significant 12 bits of the container.

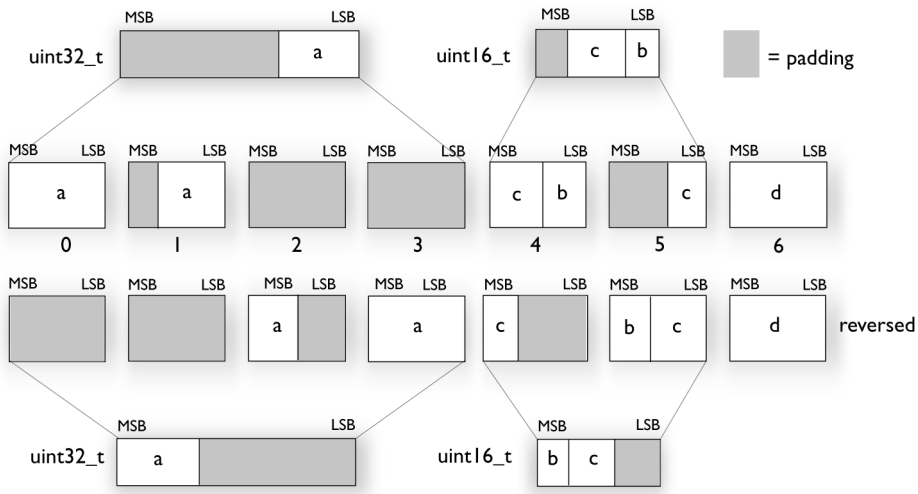
To place the second bitfield, `b`, a new container is allocated at offset `4`, because the type of the bitfield is not the same as that of the previous one. `b` is placed into the least significant three bits of this container.

The third bitfield, `c`, has the same type as `b` and fits into the same container.

The fourth member, `d`, is allocated into the byte at offset `6`. `d` cannot be placed into the same container as `b` and `c` because it is not a bitfield, it is not of the same type, and it would not fit.

When using reverse order, each bitfield is instead placed starting from the most significant bit of its container.

This is the layout of `bitfield_example` :



Basic data types—floating-point types

In the IAR C/C++ Compiler for 8051, 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 36: Floating-point types

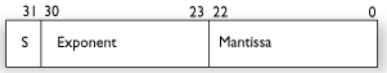
The compiler does not support subnormal numbers. All operations that should produce subnormal numbers will instead generate zero.

FLOATING-POINT ENVIRONMENT

Exception flags are not supported. The `feraiseexcept` function does not raise any exceptions.

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

The range of the number is at least:

```
±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

This 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, `NaN`. A library function which gets one of these special cases of floating-point numbers as an argument might behave unexpectedly.

Pointer types

The compiler has two basic types of pointers: function pointers and data pointers.

FUNCTION POINTERS

Code pointers have two sizes: 16 or 24 bits. These function pointers are available:

Pointer	Size	Address range	Description
`__near_func`	2 bytes	`0–0xFFFF`	Uses an `LCALL/LJMP` instruction to call the function.

Table 37: Function pointers

Pointer	Size	Address range	Description
__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 114.
__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 115.
__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 37: Function pointers (Continued)

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. These 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
__far22*	0–0x3FFFFF	3 bytes	signed short	Far22 xdata memory, accessed using MOVX
__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

Table 38: Data pointers

Pointer	Address range	Pointer size	Index type	Description
__far22_code*	0–0x3FFFFF	3 bytes	signed short	Far22 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
__far22_rom*	0–0x3FFFFF	3 bytes	signed short	Far22 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 38: Data pointers (Continued)

* The far22 and 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 reveals which memory type the object is located in and the remaining bits specify the address in that memory.

Segmented data pointer comparison

Note that the result of using relational operators (<, <=, <, <=) on data pointers is only defined if the pointers point into the same object. For segmented data pointers, only the offset part of the pointer is compared when using these operators. For example:

```
void MyFunc(char __far * p, char __far * q)
{
  if (p == q)  /* Compares the entire pointers. */
    ...
  if (p < q) /* Compares only the low 16 bits of the pointers. */
    ...
}
```

CASTING

Casts between pointers have these characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a value of an integer type to a pointer of a larger type is performed by zero extension
- Casting a pointer type to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed by zero extension
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result
- Casting from a smaller pointer to a larger pointer is performed by zero extension
- Casting from a larger pointer to a smaller pointer is performed by truncation.

size_t

`size_t` is the unsigned integer type of the result of the `sizeof` operator. In the IAR C/C++ Compiler for 8051, the type used for `size_t`

Note that some data memory types might be able to accommodate larger, or only smaller, objects than the memory pointed to by default pointers. In this case, the type of the result of the `sizeof` operator could be a larger or smaller unsigned integer type. There exists a corresponding `size_t` typedef for each memory type, named after the memory type. In other words, `__pdata_size_t` for `__pdata` memory.

ptrdiff_t

`ptrdiff_t` is the signed integer type of the result of subtracting two pointers. In the IAR C/C++ Compiler for 8051, the type used for `ptrdiff_t` is the signed integer variant of the `size_t` type.

Note that subtracting pointers other than default pointers could result in a smaller or larger integer type. In each case, this integer type is the signed integer variant of the corresponding `size_t` type.

Note: It is sometimes possible to create an object that is so large that the result of subtracting two pointers in that object is negative. See this example:

intptr_t

`intptr_t` is a signed integer type large enough to contain a `void *`. In the IAR C/C++ Compiler for 8051, the type used for `intptr_t` is `signed long`.

uintptr_t

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

Structure types

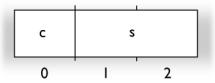
The members of a `struct` are stored sequentially in the order in which they are declared: the first member has the lowest memory address.

GENERAL LAYOUT

Members of a `struct` are always allocated in the order specified in the declaration. Each member is placed in the `struct` according to the specified alignment (offsets).

```
struct First
{
  char c;
  short s;
} s;
```

This diagram shows the layout in memory:



The alignment of the structure is 1 byte, and the size is 3 bytes.

Type qualifiers

According to the C standard, `volatile` and `const` are type qualifiers.

DECLARING OBJECTS VOLATILE

By declaring an object `volatile`, the compiler is informed that the value of the object can change beyond the compiler's control. The compiler must also assume that any accesses can have side effects—thus all accesses to the `volatile` object must be preserved.

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 C standard defines an abstract machine, which governs the behavior of accesses to `volatile` declared objects. In general and in accordance to the abstract machine:

- The compiler considers each read and write access to an object declared `volatile` as an access
- The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:

```
char volatile a;
a = 5;   /* A write access */
a += 6;  /* First a read then a write access */
```

- An access to a bitfield is treated as an access to the underlying type
- Adding a `const` qualifier to a `volatile` object will make write accesses to the object impossible. However, the object will be placed in RAM as specified by the C standard.

However, these rules are not detailed enough to handle the hardware-related requirements. The rules specific to the IAR C/C++ Compiler for 8051 are described below.

Rules for accesses

In the IAR C/C++ Compiler for 8051, accesses to `volatile` declared objects are subject to these rules:

- All accesses are preserved
- All accesses are complete, that is, the whole object is accessed
- All accesses are performed in the same order as given in the abstract machine
- All accesses are atomic, that is, they cannot be interrupted.

The 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 combinations of object types not listed, only the rule that states that all accesses are preserved applies.

DECLARING OBJECTS VOLATILE AND CONST

If you declare a `volatile` object `const`, it will be write-protected but it will still be stored in RAM memory as the C standard specifies.

To store the object in read-only memory instead, but still make it possible to access it as a `const volatile` object, declare it with one of these memory attributes: `__code`, `__far_code`, `__far_rom`, `__far22_code`, `__far22_rom`, `__huge_code`, `__huge_rom`, or `__xdata_rom`.

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 the 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 supported to write assembler code that accesses class members.

Extended keywords

- General syntax rules for extended keywords
- Summary of extended keywords
- Descriptions of extended keywords

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

General syntax rules for extended keywords

The compiler provides a set of attributes that can be used on functions or data objects to support specific features of the 8051 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 more information about each attribute, see *Descriptions of extended keywords*, page 342. 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*.

Note: The extended keywords are only available when language extensions are enabled in the compiler.

In the IDE, language extensions are enabled by default.

Use the `-e` compiler option to enable language extensions. See *-e*, page 302.

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 explicitly in your declarations, or use the pragma directive `#pragma type_attribute`.

Type attributes can be further divided into *memory type attributes* and *general type attributes*. Memory type attributes are referred to as simply *memory attributes* in the rest of the documentation.

Memory attributes

A memory attribute corresponds to a certain logical or physical memory in the microcontroller.

Available *function memory attributes*:

`__near_func`, `__far_func`, `__banked_func`

Available *data memory attributes*:

`__pdata`, `__xdata`, `__bdata`, `__bit`, `__code`, `__data`, `__far`, `__far_code`, `__far_rom`, `__far22`, `__far22_code`, `__far22_rom`, `__generic`, `__huge`, `__huge_code`, `__huge_rom`, `__idata`, `__ixdata`, `__sfr`, 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 implicitly used by the compiler. You can specify one memory attribute for each level of pointer indirection.

General type attributes

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`

You can specify as many type attributes as required for each level of pointer indirection.

Syntax for type attributes used on data objects

Type attributes use almost the same syntax rules as the type qualifiers `const` and `volatile`. For example:

```
__pdata int i;
int __pdata j;
```

Both `i` and `j` are placed in pdata memory.

Unlike `const` and `volatile`, when a type attribute is used before the type specifier in a derived type, the type attribute applies to the object, or typedef itself, except in structure member declarations.

```
int __pdata * p;       /* pointer to integer in pdata memory */

int * __pdata p;       /* pointer in pdata memory */

__pdata int * p;       /* pointer in pdata memory */
```

In all cases, if a memory attribute is not specified, an appropriate default memory type is used, which depends on the data model in use.

Using a type definition can sometimes make the code clearer:

```
typedef __pdata int d16_int;
d16_int *q1;
```

`d16_int` is a typedef for integers in pdata memory. The variable `q1` can point to such integers.

You can also use the `#pragma type_attributes` directive to specify type attributes for a declaration. The type attributes specified in the pragma directive are applied to the data object or typedef being declared.

```
#pragma type_attribute=__pdata
int * q2;
```

The variable `q2` is placed in pdata memory.

For more examples of using memory attributes, see *More examples*, page 78.

Syntax for type attributes used on functions

The syntax for using type attributes on functions differs slightly from the syntax of type attributes on data objects. For functions, the attribute must be placed either in front of the return type, or in parentheses, for example:

```
__interrupt void my_handler(void);
```

or

```
void (__interrupt my_handler)(void);
```

This declaration of `my_handler` is equivalent with the previous one:

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

To declare a function pointer, use this syntax:

```
int (__near_func * fp) (double);
```

After this declaration, the function pointer `fp` points to pdata memory.

An easier way of specifying storage is to use type definitions:

```
typedef __near_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

These object attributes are available:

- Object attributes that can be used for variables:

  `__no_alloc`, `__no_alloc16`, `__no_alloc_str`, `__no_alloc_str16`, `__no_init`, `__ro_placement`
- Object attributes that can be used for functions and variables:

  `location`, `@`, `__root`
- Object attributes that can be used for functions:

  `__intrinsic`, `__noreturn`, `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 259. For more information about `vector`, see *vector*, page 381.

Syntax for object attributes

The object attribute must be placed in front of the type. For example, to place `myarray` in memory that is not initialized at startup:

```
__no_init int myarray[10];
```

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

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

Note: Object attributes cannot be used in combination with the `typedef` keyword.

Summary of extended keywords

This table summarizes the extended keywords:

Extended keyword	Description
__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 auto 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
__far22	Controls the storage of data objects
__far22_code	Controls the storage of constant data objects
__far22_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	Specifies interrupt functions
__intrinsic	Reserved for compiler internal use only
__monitor	Specifies atomic execution of a function
__near_func	Controls the storage of functions
__no_alloc, __no_alloc16	Makes a constant available in the execution file

Table 39: Extended keywords summary

Extended keyword	Description
`__no_alloc_str`, `__no_alloc_str16`	Makes a string literal available in the execution file
`__no_init`	Places a data object in non-volatile memory
`__noreturn`	Informs the compiler that the 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
`__ro_placement`	Places `const volatile` data in read-only memory.
`__sfr`	Controls the storage of data objects
`__xdata`	Controls the storage of data objects
`__xdata_reentrant`	Controls the storage of auto data objects
`__xdata_rom`	Controls the storage of constant data objects

Table 39: Extended keywords summary (Continued)

Descriptions of extended keywords

This section gives detailed information about each extended keyword.

__banked_func

Syntax

See *Syntax for type attributes used on functions*, page 339.

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 banked memory.

Storage information

- Memory space: Code memory space
- Address range: `0-0xFFFFFF`
- Maximum size: 64 Kbytes
- Pointer size: 2 bytes

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 113. 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 107.

__banked_func_ext2

Syntax

See *Syntax for type attributes used on functions*, page 339.

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 banked memory.

Storage information

- Memory space: Code memory space
- Address range: `0-0xFFFFFF`
- Maximum size: 64 Kbytes
- Pointer size: 3 bytes

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 107.

__bdata

Syntax

See *Syntax for type attributes used on data objects*, page 338.

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 68.

__bit

Syntax See *Syntax for type attributes used on data objects*, page 338.

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
```
__no_init __bit bool x;
```

See also *Memory types*, page 68.

__code

Syntax See *Syntax for type attributes used on data objects*, page 338.

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 const int x;
```

See also — *Memory types*, page 68.

__data

Syntax — See *Syntax for type attributes used on data objects*, page 338.

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 68.

__data_overlay

Syntax — See *Syntax for type attributes used on functions*, page 339.

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 myfunction(void);
```

See also — *Storage of auto variables and parameters*, page 83.

__ext_stack_reentrant

Syntax — See *Syntax for type attributes used on functions*, page 339.

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 myfunction(void);
```

See also

Storage of auto variables and parameters, page 83, *Extended stack*, page 133, and *--extended_stack*, page 304.

__far

Syntax

See *Syntax for type attributes used on data objects*, page 338.

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 68.

__far_code

Syntax

See *Syntax for type attributes used on data objects*, page 338.

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 const int x;
```

See also

Memory types, page 68.

__far_func

Syntax

See *Syntax for type attributes used on functions*, page 339.

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 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: 64 Kbytes. A function cannot cross a 64-Kbyte boundary.
- Pointer size: 3 bytes

Example

```
__far_func void myfunction(void);
```

See also

Code models and memory attributes for function storage, page 95.

__far_rom

Syntax

See *Syntax for type attributes used on data objects*, page 338.

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: 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_rom const int x;
```

See also *Memory types*, page 68.

__far22

Syntax See *Syntax for type attributes used on data objects*, page 338.

Description The `__far22` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in far22 memory.

You can also use the `__far22` attribute to create a pointer explicitly pointing to an object located in the far22 memory.

Note: This memory attribute is only available when the Far Generic data model is used.

Storage information
- Memory space: External data memory space
- Address range: `0-0x3FFFFF`
- 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 22-bit address.

Example
```
__far22 int x;
```

See also *Memory types*, page 68.

__far22_code

Syntax See *Syntax for type attributes used on data objects*, page 338.

Description The `__far22_code` memory attribute overrides the default storage of variables given by the selected data model and places individual constants and strings in far22 code memory.

You can also use the `__far22_code` attribute to create a pointer explicitly pointing to an object located in the far22 code memory.

Note: This memory attribute is only available when the Far Generic code model is used.

Storage information

- Memory space: Code memory space
- Address range: `0-0x3FFFFF`
- 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 22-bit address.

Example

```
__far22_code const int x;
```

See also

Memory types, page 68.

__far22_rom

Syntax

See *Syntax for type attributes used on data objects*, page 338.

Description

The `__far22_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 far22 memory range.

You can also use the `__far22_rom` attribute to create a pointer explicitly pointing to an object located in the far22_rom memory.

Note: This memory attribute is only available when the Far Generic data model is used.

Storage information

- Memory space: External data memory space
- Address range: `0-0x3FFFFF`
- 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 22-bit address.

Example

```
__far22_rom const int x;
```

See also

Memory types, page 68.

__generic

Syntax

See *Syntax for type attributes used on data objects*, page 338.

Description

The `__generic` pointer attribute specifies a generic pointer that can access data in the internal data memory space, external data memory space, or the code memory space.

If a variable is declared with this keyword, it will be located in the external data memory space.

Note: This memory attribute is not available when the Far data model is used.

Example

```
int __generic * ptr;
```

See also

Generic pointers, page 332.

__huge

Syntax

See *Syntax for type attributes used on data objects*, page 338.

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 68.

__huge_code

Syntax

See *Syntax for type attributes used on data objects*, page 338.

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 const int x;
```

See also

Memory types, page 68.

__huge_rom

Syntax

See *Syntax for type attributes used on data objects*, page 338.

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 huge 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: 3 bytes

Example

```
__huge_rom const int x;
```

See also

Memory types, page 68.

__idata

Syntax

See *Syntax for type attributes used on data objects*, page 338.

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 68.

__idata_overlay

Syntax

See *Syntax for type attributes used on functions*, page 339.

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 myfunction(void);
```

See also

Storage of auto variables and parameters, page 83.

__idata_reentrant

Syntax

See *Syntax for type attributes used on functions*, page 339.

Description

The `__idata_reentrant` keyword places parameters and auto variables on the idata stack.

Example

```
__idata_reentrant void myfunction(void);
```

See also

Storage of auto variables and parameters, page 83, *Idata stack*, page 133

__ixdata

Syntax

See *Syntax for type attributes used on data objects*, page 338.

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 `__xdata` pointers.

Storage information

- Memory space: External data memory space
- Address range: `0x0-0xFFFF`
- Maximum object size: 64 Kbytes
- Pointer size: 2 bytes, `__xdata` pointer

Example

```
__ixdata int x;
```

See also

Memory types, page 68.

__interrupt

Syntax

See *Syntax for type attributes used on functions*, page 339.

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.

To make sure that the interrupt handler executes as fast as possible, you should compile it with `-Ohs`, or use `#pragma optimize=speed` if the module is compiled with another optimization goal.

Example

```
__interrupt void my_interrupt_handler(void);
```

See also

Interrupt functions, page 97, *vector*, page 381, and *INTVEC*, page 430.

__intrinsic

Description

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

__monitor

Syntax

See *Syntax for type attributes used on functions*, page 339.

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 99. For information about related intrinsic functions, see *__disable_interrupt*, page 383, *__enable_interrupt*, page 384, *__get_interrupt_state*, page 384, and *__set_interrupt_state*, page 385, respectively.

__near_func

Syntax

See *Syntax for type attributes used on functions*, page 339.

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 and memory attributes for function storage, page 95.

__no_alloc, __no_alloc16

Syntax

See *Syntax for object attributes*, page 340.

Description

Use the `__no_alloc` or `__no_alloc16` object attribute on a constant to make the constant available in the executable file without occupying any space in the linked application.

You cannot access the contents of such a constant from your application. You can take its address, which is an integer offset to the segment of the constant. The type of the offset is `unsigned long` when `__no_alloc` is used, and `unsigned short` when `__no_alloc16` is used.

Example

```
__no_alloc const struct MyData my_data @ "XXX" = {...};
```

See also

__no_alloc_str, __no_alloc_str16, page 355.

__no_alloc_str, __no_alloc_str16

Syntax

`__no_alloc_str(`*`string_literal`* `@` *`segment`*`)`

and

`__no_alloc_str16(`*`string_literal`* `@` *`segment`*`)`

where

`string_literal`	The string literal that you want to make available in the executable file.
`segment`	The name of the segment to place the string literal in.

Description

Use the `__no_alloc_str` or `__no_alloc_str16` operators to make string literals available in the executable file without occupying any space in the linked application.

The value of the expression is the offset of the string literal in the segment. For `__no_alloc_str`, the type of the offset is `unsigned long`. For `__no_alloc_str16`, the type of the offset is `unsigned short`.

Example

```
#define MYSEG "YYY"
#define X(str) __no_alloc_str(str @ MYSEG)

extern void dbg_printf(unsigned long fmt, ...)

#define DBGPRINTF(fmt, ...) dbg_printf(X(fmt), __VA_ARGS__)

void
foo(int i, double d)
{
  DBGPRINTF("The value of i is: %d, the value of d is: %f",i,d);
}
```

Depending on your debugger and the runtime support, this could produce trace output on the host computer. Note that there is no such runtime support in C-SPY, unless you use an external plugin module.

See also

__no_alloc, __no_alloc16, page 355.

__no_init

Syntax

See *Syntax for object attributes*, page 340.

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];
```

See also

Non-initialized variables, page 274.

__noreturn

Syntax

See *Syntax for object attributes*, page 340.

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`.

Note: At optimization levels medium or high, the `__noreturn` keyword might cause incorrect call stack debug information at any point where it can be determined that the current function cannot return.

Example

```
__noreturn void terminate(void);
```

__overlay_near_func

Description — The `__overlay_near_func` keyword is reserved for compiler internal use only.

__pdata

Syntax — See *Syntax for type attributes used on data objects*, page 338.

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 68.

__pdata_reentrant

Syntax — See *Syntax for type attributes used on functions*, page 339.

Description — The `__pdata_reentrant` keyword places parameters and auto variables on the pdata stack.

Example
```
__pdata_reentrant void myfunction(void);
```

See also — *Storage of auto variables and parameters*, page 83, *Pdata stack*, page 134

__root

Syntax — See *Syntax for object attributes*, page 340.

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
For more information about modules, segments, and the link process, see the *IAR Linker and Library Tools Reference Guide.*

__ro_placement

Syntax
See *Syntax for object attributes*, page 340.

Description
The `__ro_placement` attribute specifies that a data object should be placed in read-only memory. There are two cases where you might want to use this object attribute:

- Data objects declared `const volatile` are by default placed in read-write memory. Use the `__ro_placement` object attribute to place the data object in read-only memory instead.
- In C++, a data object declared `const` and that needs dynamic initialization is placed in read-write memory and initialized at system startup. If you use the `__ro_placement` object attribute, the compiler will give an error message if the data object needs dynamic initialization.

You can only use the `__ro_placement` object attribute on `const` objects.

In some cases (primarily involving simple constructors), the compiler will be able to optimize C++ dynamic initialization of a data object into static initialization. In that case no error message will be issued for the object.

Example
```
__ro_placement const volatile int x = 10;
```

__sfr

Syntax
See *Syntax for type attributes used on data objects*, page 338.

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 the SFR memory.

Storage information
- Memory space: Internal data memory space
- Address range: `0x80-0xFF`
- Maximum object size: 128 bytes
- Pointer size: N/a

Example
```
__sfr int x;
```

See also
Memory types, page 68.

__task

Syntax
See *Syntax for type attributes used on functions*, page 339.

Description
This keyword allows functions to relax the rules for preserving registers. Typically, the keyword is used on the start function for a task in an RTOS.

By default, functions save the contents of used preserved registers on the stack upon entry, and restore them at exit. Functions that are declared `__task` do not save all registers, and therefore require less stack space.

Because a function declared `__task` can corrupt registers that are needed by the calling function, you should only use `__task` on functions that do not return or call such a function from assembler code.

The function `main` can 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 can be declared `__task`.

Example
```
__task void my_handler(void);
```

__xdata

Syntax
See *Syntax for type attributes used on data objects*, page 338.

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: `0x0-0xFFFF`
- Maximum object size: 64 Kbytes
- Pointer size: 2 bytes

Example
```
__xdata int x;
```

See also
Memory types, page 68.

__xdata_reentrant

Syntax

See *Syntax for type attributes used on functions*, page 339.

Description

The `__xdata_reentrant` keyword places parameters and auto variables on the xdata stack.

Example

```
__xdata_reentrant void myfunction(void);
```

See also

Storage of auto variables and parameters, page 83, *Xdata stack*, page 134

__xdata_rom

Syntax

See *Syntax for type attributes used on data objects*, page 338.

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`
- Maximum object size: 64 Kbytes
- Pointer size: 2 bytes

Example

```
__xdata_rom const int x;
```

See also

Memory types, page 68.

Pragma directives

- Summary of pragma directives
- Descriptions of pragma directives

Summary of pragma directives

The #pragma directive is defined by Standard C 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.

This table lists the pragma directives of the compiler that can be used either with the `#pragma` preprocessor directive or the `_Pragma()` preprocessor operator:

Pragma directive	Description
`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.
`cstat_disable`	See the *C-STAT® Static Analysis Guide*.
`cstat_enable`	See the *C-STAT® Static Analysis Guide*.
`cstat_restore`	See the *C-STAT® Static Analysis Guide*.
`cstat_suppress`	See the *C-STAT® Static Analysis Guide*.
`data_alignment`	Gives a variable a higher (more strict) alignment.
`dataseg`	Places variables in a named segment.
`default_function_attributes`	Sets default type and object attributes for declarations and definitions of functions.
`default_variable_attributes`	Sets default type and object attributes for declarations and definitions of variables.
`diag_default`	Changes the severity level of diagnostic messages.
`diag_error`	Changes the severity level of diagnostic messages.

Table 40: Pragma directives summary

Pragma directive	Description
diag_remark	Changes the severity level of diagnostic messages.
diag_suppress	Suppresses diagnostic messages.
diag_warning	Changes the severity level of diagnostic messages.
error	Signals an error while parsing.
include_alias	Specifies an alias for an include file.
inline	Controls inlining of 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.
message	Prints a message.
object_attribute	Adds object attributes to the declaration or definition of a variable or 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.
public_equ	Defines a public assembler label and gives it a value.
register_bank	Sets the register bank for 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.
section	This directive is an alias for #pragma segment.
segment	Declares a segment name to be used by intrinsic functions.
STDC CX_LIMITED_RANGE	Specifies whether the compiler can use normal complex mathematical formulas or not.
STDC FENV_ACCESS	Specifies whether your source code accesses the floating-point environment or not.
STDC FP_CONTRACT	Specifies whether the compiler is allowed to contract floating-point expressions or not.
vector	Specifies the vector of an interrupt or trap function.
weak	Makes a definition a weak definition, or creates a weak alias for a function or a variable.

Table 40: Pragma directives summary (Continued)

Pragma directive	Description
`type_attribute`	Adds type attributes to a declaration or to definitions.

Table 40: Pragma directives summary (Continued)

Note: For portability reasons, see also *Recognized pragma directives (6.10.6)*, page 449.

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, see *Templates and data memory attributes*, page 239.

Example

```
#pragma basic_template_matching
template<typename T> void fun(T *);

void MyF()
{
  fun((int __pdata *) 0); // T = int __pdata
}
```

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.

Example

```
#pragma bitfields=reversed
/* Structure that uses reversed bitfields. */
struct S
{
  unsigned char  error : 1;
  unsigned char  size  : 4;
  unsigned short code  : 10;
};
#pragma bitfields=default /* Restores to default setting. */
```

See also

Bitfields, page 327.

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
```

data_alignment

Syntax

```
#pragma data_alignment=expression
```

Parameters

expression	A constant which must be a power of two (1, 2, 4, etc.).

Description

Use this pragma directive to give the immediately following variable a higher (more strict) alignment of the start address than it would otherwise have. This directive can be used on variables with static and automatic storage duration.

When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.

Note: Normally, the size of a variable is a multiple of its alignment. The `data_alignment` directive only affects the alignment of the variable's start address, and not its size, and can thus be used for creating situations where the size is not a multiple of the alignment.

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 dataseg=default` directive.

Example

```
#pragma dataseg=__pdata MY_SEGMENT
__no_init char myBuffer[1000];
#pragma dataseg=default
```

default_function_attributes

Syntax

```
#pragma default_function_attributes=[ attribute...]
```

where *attribute* can be:

```
type_attribute
object_attribute
@ segment_name
```

Parameters

type_attribute	See *Type attributes*, page 337.
object_attribute	See *Object attributes*, page 340.
@ *segment_name*	See *Data and function placement in segments*, page 262.

Description

Use this pragma directive to set default segment placement, type attributes, and object attributes for function declarations and definitions. The default settings are only used for declarations and definitions that do not specify type or object attributes or location in some other way.

Specifying a `default_function_attributes` pragma directive with no attributes, restores the initial state where no such defaults have been applied to function declarations and definitions.

Example

```
/* Place following functions in segment MYSEG" */
#pragma default_function_attributes = @ "MYSEG"
int fun1(int x) { return x + 1; }
int fun2(int x) { return x - 1;
/* Stop placing functions into MYSEG */
#pragma default_function_attributes =
```

has the same effect as:

```
int fun1(int x) @ "MYSEG" { return x + 1; }
int fun2(int x) @ "MYSEG" { return x - 1; }
```

See also

location, page 372

object_attribute, page 373

type_attribute, page 380

default_variable_attributes

Syntax

```
#pragma default_variable_attributes=[ attribute...]
```

where *attribute* can be:

```
type_attribute
object_attribute
@ segment_name
```

Parameters

type_attribute	See *Type attributes*, page 337.
object_attributes	See *Object attributes*, page 340.
@ *segment_name*	See *Data and function placement in segments*, page 262.

Description

Use this pragma directive to set default segment placement, type attributes, and object attributes for declarations and definitions of variables with static storage duration. The default settings are only used for declarations and definitions that do not specify type or object attributes or location in some other way.

Specifying a `default_variable_attributes` pragma directive with no attributes restores the initial state of no such defaults being applied to variables with static storage duration.

Example

```
/* Place following variables in segment MYSEG" */
#pragma default_variable_attributes = @ "MYSEG"
int var1 = 42;
int var2 = 17;
/* Stop placing variables into MYSEG */
#pragma default_variable_attributes =
```

has the same effect as:

```
int var1 @ "MYSEG" = 42;
int var2 @ "MYSEG" = 17;
```

See also

location, page 372

object_attribute, page 373

type_attribute, page 380

diag_default

Syntax

```
#pragma diag_default=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 back to the default, or to the severity level defined on the command line by any of the options `--diag_error`, `--diag_remark`, `--diag_suppress`, or `--diag_warnings`, for the diagnostic messages specified with the tags. This level remains in effect until changed by another diagnostic-level pragma directive.

See also

Diagnostics, page 283.

diag_error

Syntax

```
#pragma diag_error=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 `error` for the specified diagnostics. This level remains in effect until changed by another diagnostic-level pragma directive.

See also

Diagnostics, page 283.

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. This level remains in effect until changed by another diagnostic-level pragma directive.

See also — *Diagnostics*, page 283.

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. This level remains in effect until changed by another diagnostic-level pragma directive.

See also — *Diagnostics*, page 283.

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. This level remains in effect until changed by another diagnostic-level pragma directive.

See also — *Diagnostics*, page 283.

error

Syntax — `#pragma error message`

Parameters

message	A string that represents the error message.

Description — Use this pragma directive to cause an error message when it is parsed. This mechanism is different from the preprocessor directive `#error`, because the `#pragma error` directive can be included in a preprocessor macro using the `_Pragma` form of the directive and only causes an error if the macro is used.

Example

```
#if FOO_AVAILABLE
#define FOO ...
#else
#define FOO _Pragma("error\"Foo is not available\"")
#endif
```

If `FOO_AVAILABLE` is zero, an error will be signaled if the `FOO` macro is used in actual source code.

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 280.

inline

Syntax

```
#pragma inline[=forced|=never]
```

Parameters

No parameter	Has the same effect as the `inline` keyword.
`forced`	Disables the compiler's heuristics and forces inlining.

`never`	Disables the compiler's heuristics and makes sure that the function will not be inlined.

Description

Use `#pragma inline` to advise the compiler that the function defined immediately after the directive should be inlined according to C++ inline semantics.

Specifying `#pragma inline=forced` will always inline the defined function. If the compiler fails to inline the function for some reason, for example due to recursion, a warning message is emitted.

Inlining is normally performed only on the High optimization level. Specifying `#pragma inline=forced` will enable inlining of the function also on the Medium optimization level.

See also

Inlining functions, page 103.

language

Syntax

```
#pragma language={extended|default|save|restore}
```

Parameters

`extended`	Enables the IAR Systems language extensions from the first use of the pragma directive and onward.
`default`	From the first use of the pragma directive and onward, restores the settings for the IAR Systems language extensions to whatever that was specified by compiler options.
`save|restore`	Saves and restores, respectively, the IAR Systems language extensions setting around a piece of source code. Each use of `save` must be followed by a matching `restore` in the same file without any intervening `#include` directive.

Description

Use this pragma directive to control the use of language extensions.

Example

At the top of a file that needs to be compiled with IAR Systems extensions enabled:

```
#pragma language=extended
/* The rest of the file. */
```

Around a particular part of the source code that needs to be compiled with IAR Systems extensions enabled, but where the state before the sequence cannot be assumed to be the same as that specified by the compiler options in use:

```
#pragma language=save
#pragma language=extended
/* Part of source code. */
#pragma language=restore
```

See also

-*e*, page 302 and --*strict*, page 320.

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. Do not place variables that would normally be in different segments (for example, variables declared as `__no_init` and variables declared as `const`) in the same named segment.

Example

```
#pragma location=0xFF20
__no_init volatile char PORT1; /* PORT1 is located at address
                                  0xFF20 */

#pragma segment="FLASH"
#pragma location="FLASH"
__no_init char PORT2; /* PORT2 is located in segment FLASH */

/* A better way is to use a corresponding mechanism */
#define FLASH _Pragma("location=\"FLASH\"")
/* ... */
FLASH __no_init int i; /* i is placed in the FLASH segment */
```

See also

Controlling data and function placement in memory, page 259 and *Placing user-defined segments*, page 132.

message

Syntax

```
#pragma message(message)
```

Parameters

message	The message that you want to direct to the standard output stream.

Description

Use this pragma directive to make the compiler print a message to the standard output stream when the file is compiled.

Example

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

object_attribute

Syntax

```
#pragma object_attribute=object_attribute[ object_attribute...]
```

Parameters

For information about object attributes that can be used with this pragma directive, see *Object attributes*, page 340.

Description

Use this pragma directive to add one or more IAR-specific object attributes to the declaration or definition of a variable or function. Object attributes affect the actual variable or function and not its type. When you define a variable or function, the union of the object attributes from all declarations including the definition, is used.

Example

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

is equivalent to:

```
__no_init char bar;
```

See also

General syntax rules for extended keywords, page 337.

optimize

Syntax

```
#pragma optimize=[goal][level][no_optimization...]
```

Parameters

goal	Choose between: `size`, optimizes for size `balanced`, optimizes balanced between speed and size `speed`, optimizes for speed. `no_size_constraints`, optimizes for speed, but relaxes the normal restrictions for code size expansion.
level	Specifies the level of optimization; choose between `none`, `low`, `medium`, or `high`.
`no_`*optimization*	Disables one or several optimizations; choose between: `no_code_motion`, disables code motion `no_crosscall`, disables interprocedural cross call `no_cse`, disables common subexpression elimination `no_inline`, disables function inlining `no_tbaa`, disables type-based alias analysis `no_unroll`, disables 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 `size`, `balanced`, `speed`, and `no_size_constraints` 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 SmallAndUsedOften()
{
  /* Do something here. */
}

#pragma optimize=size
int BigAndSeldomUsed()
{
  /* Do something here. */
}
```

See also

Fine-tuning enabled transformations, page 266.

__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 *,...);

void PrintNumbers(unsigned short x)
{
  printf("%d", x);  /* Compiler checks that x is an integer */
}
```

public_equ

Syntax

```
#pragma public_equ="symbol",value
```

Parameters

symbol	The name of the assembler symbol to be defined (string).
value	The value of the defined assembler symbol (integer constant expression).

Description

Use this pragma directive to define a public assembler label and give it a value.

Example

```
#pragma public_equ="MY_SYMBOL",0x123456
```

See also --*public_equ*, page 318.

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()
{
  /* Do something here. */
}
```

Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.

See also

Inline assembler, page 193

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. 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 141.

__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 scanf(char const *,...);

int GetNumber()
{
  int nr;
  scanf("%d", &nr);  /* Compiler checks that
                        the argument is a
                        pointer to an integer */

  return nr;
}
```

segment

Syntax

```
#pragma segment="NAME" [__memoryattribute] [align]
```

alias

```
#pragma section="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. 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`, `__segment_end`, and `__segment_size`. All segment declarations for a specific segment must have the same memory type attribute and alignment.

The *align* and the*__memoryattribute* parameters are only relevant when used together with the segment operators `__segment_begin`, `__segment_end`, and

`__segment_size`. If you consider using *`align`* on an individual variable to achieve a higher alignment, you must instead use the `#pragma data_alignment` directive.

If an optional memory attribute is used, the return type of the segment operators `__segment_begin` and `__segment_end` is:

`void __memoryattribute *.`

Note: To place variables or functions in a specific segment, use the `#pragma location` directive or the `@` operator.

Example

```
#pragma segment="MYPDATA" __pdata 4
```

See also

Dedicated segment operators, page 225 and the chapters *Linking overview* and *Linking your application*.

STDC CX_LIMITED_RANGE

Syntax

```
#pragma STDC CX_LIMITED_RANGE {ON|OFF|DEFAULT}
```

Parameters

`ON`	Normal complex mathematic formulas can be used.
`OFF`	Normal complex mathematic formulas cannot be used.
`DEFAULT`	Sets the default behavior, that is `OFF`.

Description

Use this pragma directive to specify that the compiler can use the normal complex mathematic formulas for `*` (multiplication), `/` (division), and `abs`.

Note: This directive is required by Standard C. The directive is recognized but has no effect in the compiler.

STDC FENV_ACCESS

Syntax

```
#pragma STDC FENV_ACCESS {ON|OFF|DEFAULT}
```

Parameters

`ON`	Source code accesses the floating-point environment. Note that this argument is not supported by the compiler.
`OFF`	Source code does not access the floating-point environment.
`DEFAULT`	Sets the default behavior, that is `OFF`.

Description

Use this pragma directive to specify whether your source code accesses the floating-point environment or not.

Note: This directive is required by Standard C.

STDC FP_CONTRACT

Syntax

```
#pragma STDC FP_CONTRACT {ON|OFF|DEFAULT}
```

Parameters

`ON`	The compiler is allowed to contract floating-point expressions.
`OFF`	The compiler is not allowed to contract floating-point expressions. Note that this argument is not supported by the compiler.
`DEFAULT`	Sets the default behavior, that is `ON`.

Description

Use this pragma directive to specify whether the compiler is allowed to contract floating-point expressions or not. This directive is required by Standard C.

Example

```
#pragma STDC FP_CONTRACT=ON
```

type_attribute

Syntax

```
#pragma type_attribute=type_attr[ type_attr...]
```

Parameters

For information about type attributes that can be used with this pragma directive, see *Type attributes*, page 337.

Description

Use this pragma directive to specify IAR-specific *type attribute*s, which are not part of Standard C. Note however, that a given type attribute might 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 this example, an `int` object with the memory attribute `__pdata` is defined:

```
#pragma type_attribute=__pdata
int x;
```

This declaration, which uses extended keywords, is equivalent:

```
__pdata int x;
```

See also

The chapter *Extended keywords*.

vector

Syntax

```
#pragma vector=vector1[, vector2, vector3, ...]
```

Parameters

vectorN	The vector number(s) of an interrupt function.

Description

Use this pragma directive to specify the vector(s) of an function whose declaration follows the pragma directive. Note that several vectors can be defined for each function.

Example

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

weak

Syntax

```
#pragma weak symbol1[=symbol2]
```

Parameters

symbol1	A function or variable with external linkage.
symbol2	A defined function or variable.

Description

This pragma directive can be used in one of two ways:

- To make the definition of a function or variable with external linkage a weak definition.
- To create a weak alias for another function or variable. You can make more than one alias for the same function or variable.

Example

To make the definition of `foo` a weak definition, write:

```
#pragma weak foo
```

To make `NMI_Handler` a weak alias for `Default_Handler`, write:

```
#pragma weak NMI_Handler=Default_Handler
```

If `NMI_Handler` is not defined elsewhere in the program, all references to `NMI_Handler` will refer to `Default_Handler`.

Intrinsic functions

- Summary of intrinsic functions
- Descriptions of intrinsic functions

Summary of 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 use intrinsic functions in an application, include the header file `intrinsics.h`.

Note that the intrinsic function names start with double underscores, for example:

```
__disable_interrupt
```

This table summarizes the intrinsic functions:

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 41: Intrinsic functions summary

Descriptions of intrinsic functions

This 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

```
#include "intrinsics.h"

void CriticalFn()
{
  __istate_t s = __get_interrupt_state();
  __disable_interrupt();

  /* Do something here. */

  __set_interrupt_state(s);
}
```

The advantage of using this sequence of code compared to using `__disable_interrupt` and `__enable_interrupt` is that the code in this example will not enable any interrupts disabled before the call of `__get_interrupt_state`.

__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);
```

Description

Restores the interrupt state to a value previously returned by the `__get_interrupt_state` function.

For information about the `__istate_t` type, see *__get_interrupt_state*, page 384.

__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 326.

The preprocessor

- Overview of the preprocessor
- Description of predefined preprocessor symbols
- Descriptions of miscellaneous preprocessor extensions

Overview of the preprocessor

The preprocessor of the IAR C/C++ Compiler for 8051 adheres to Standard C. The compiler also makes these preprocessor-related features available to you:

- Predefined preprocessor symbols

  These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. For more information, see *Description of predefined preprocessor symbols*, page 388.
- 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 294.
- Preprocessor extensions

  There are several preprocessor extensions, for example many pragma directives; for more information, see the chapter *Pragma directives*. For information about the corresponding `_Pragma` operator and the other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 393.
- Preprocessor output

  Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 317.

To specify a path for an include file, use forward slashes:

```
#include "mydirectory/myfile"
```

In source code, use forward slashes:

```
file = fopen("mydirectory/myfile","rt");
```

Note that backslashes can also be used. In this case, use one in include file paths and two in source code strings.

Description of predefined preprocessor symbols

This section lists and describes the preprocessor symbols.

Note: To list the predefined preprocessor symbols, use the compiler option `--predef_macros`. See *--predef_macros*, page 317.

__BASE_FILE__

Description

A string that identifies the name of the base source file (that is, not the header file), being compiled.

See also

__FILE__, page 390, and *--no_path_in_file_macros*, page 310.

__BUILD_NUMBER__

Description

A unique integer that identifies the build number of the compiler currently in use.

__CALLING_CONVENTION__

Description

An integer that identifies the calling convention in use. The symbol 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. These symbolic names can be used when testing the `__CALLING_CONVENTION__` symbol: `__CC_DO__`, `__CC_IO__`, `__CC_IR__`, `__CC_PR__`, `__CC_XR__`, or `__CC_ER__`.

__CODE_MODEL__

Description

An integer that identifies the code model in use. The value reflects the setting of the `--code_model` option and is defined to `1` for Near, `2` for Banked, `3` for Far, and `4` for Banked extended2 code model. These symbolic names can be used when testing the `__CODE_MODEL__` symbol: `__CM_NEAR__`, `__CM_BANKED__`, `__CM_FAR__`, or `__CM_BANKED_EXT2__`.

__CONSTANT_LOCATION__

Description

An integer that identifies the default placement of constants and strings. The symbol reflects the setting of the `--place_constants` option and is defined to `0` for Data, `1` for Data ROM, and `2` for Code.

__CORE__

Description

An integer that identifies the chip core in use. The value reflects the setting of the `--core` option and is defined to `1` for Plain, `2` for Extended1, and `3` for Extended2 core. These symbolic names can be used when testing the `__CORE__` symbol: `__CORE_PLAIN__`, `__CORE_EXTENDED1__`, or `__CORE_EXTENDED2__`..

__COUNTER__

Description

A macro that expands to a new integer each time it is expanded, starting at zero (0) and counting up.

__cplusplus

Description

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.

This symbol is required by Standard C.

__DATA_MODEL__

Description

An integer that identifies the data model in use. The value reflects the setting of the `--data_model` option and is defined to `0` for Tiny, `1` for Small, `2` for Large, `3` for Generic, `4` for Far, and `5` for the Far Generic data model. These symbolic names can be used when testing the `__DATA_MODEL__` symbol: `__DM_TINY__`, `__DM_SMALL__`, `__DM_LARGE__`, `__DM_GENERIC__`, `__DM_FAR__`, or `__DM_FAR_GENERIC__`..

__DATE__

Description

A string that identifies the date of compilation, which is returned in the form "`Mmm dd yyyy`", for example "`Oct 30 2014`"

This symbol is required by Standard C.

__embedded_cplusplus

Description

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.

This symbol is required by Standard C.

__EXTENDED_DPTR__

Description

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__

Description

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__

Description

A string that identifies the name of the file being compiled, which can be both the base source file and any included header file.

This symbol is required by Standard C.

See also

__BASE_FILE__, page 388, and *--no_path_in_file_macros*, page 310.

__func__

Description

A predefined string identifier that is initialized with 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.

This symbol is required by Standard C.

See also

-e, page 302 and *__PRETTY_FUNCTION__*, page 391.

__FUNCTION__

Description

A predefined string identifier that is initialized with 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 also

-e, page 302 and *__PRETTY_FUNCTION__*, page 391.

__IAR_SYSTEMS_ICC__

Description

An integer that identifies the IAR compiler platform. The current value is 8. 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__

Description

An integer that is set to `1` when the code is compiled with the IAR C/C++ Compiler for 8051.

__INC_DPSEL_SELECT__

Description

An integer that is set to `1` when the INC method is used for selecting he active data pointer. Otherwise, when the XOR method is used, the symbol is undefined.

__LINE__

Description

An integer that identifies the current source line number of the file being compiled, which can be both the base source file and any included header file.

This symbol is required by Standard C.

__NUMBER_OF_DPTRS__

Description

An integer that identifies the number of data pointers being used; a value between `1` and `8`.

__PRETTY_FUNCTION__

Description

A predefined string identifier that is initialized with 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 also

-*e*, page 302 and__*func*__, page 390.

__STDC__

Description

An integer that is set to `1`, which means the compiler adheres to Standard C. This symbol can be tested with `#ifdef` to detect whether the compiler in use adheres to Standard C.*

This symbol is required by Standard C.

__STDC_VERSION__

Description

An integer that identifies the version of the C standard in use. The symbol expands to `199901L`,unless the `--c89` compiler option is used in which case the symbol expands to `199409L`. This symbol does not apply in EC++ mode.

This symbol is required by Standard C.

__SUBVERSION__

Description

An integer that identifies the subversion number of the compiler version number, for example 3 in 1.2.3.4.

__TIME__

Description

A string that identifies the time of compilation in the form `"hh:mm:ss"`.

This symbol is required by Standard C.

__TIMESTAMP__

Description

A string constant that identifies the date and time of the last modification of the current source file. The format of the string is the same as that used by the `asctime` standard function (in other words, `"Tue Sep 16 13:03:52 2014"`).

__VER__

Description

An integer that identifies the version number of the IAR compiler in use. The value of the number is calculated in this way: `(100 * the major version number + the minor version number)`. For example, for compiler version 3.34, 3 is the major version number and 34 is the minor version number. Hence, the value of `__VER__` is 334.

__XOR_DPSEL_SELECT__

Description

An integer that is set to `1` when the XOR method is used for selecting he active data pointer. Otherwise, when the INC method is used, the symbol is undefined.

Descriptions of miscellaneous preprocessor extensions

This section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives, and Standard C directives.

NDEBUG

Description

This preprocessor symbol determines whether any assert macros you have written in your application shall be included or not in the built application.

If this symbol is not defined, all assert macros are evaluated. If the symbol is defined, all assert macros are excluded from the compilation. In other words, if the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you write any assert code and build your application, you should define this symbol to exclude the assert code from the final application.

Note that the assert macro is defined in the `assert.h` standard include file.

In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

See also

_ReportAssert, page 174.

#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 Standard C `#error` directive is used. This directive is not recognized when the `--strict` compiler option is used.

C/C++ standard library functions

- C/C++ standard library overview
- DLIB runtime environment—implementation details
- CLIB runtime environment—implementation details
- 8051-specific CLIB functions

For detailed reference information about the library functions, see the online help system.

C/C++ standard library overview

The compiler comes with two different implementations of the C/C++ standard library:

The IAR DLIB Runtime Environment is a complete implementation of the C/C++ standard library, compliant with Standard 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, etc.

The IAR CLIB Runtime Environment is a light-weight implementation of the C standard library, which is not fully compliant with Standard C. Neither does it fully support floating-point numbers in IEEE 754 format and it does not support C++.

For more information about customization, 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 more information about library functions, see the chapter *Implementation-defined behavior for Standard C* in this guide.

HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into several

different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic project configuration*, page 56 . The linker will include only those routines that are required—directly or indirectly—by your application.

See also *Overriding library modules*, page 152 for information about how you can override library modules with your own versions.

ALTERNATIVE MORE ACCURATE LIBRARY FUNCTIONS

The default implementation of `cos`, `sin`, `tan`, and `pow` is designed to be fast and small. As an alternative, there are versions designed to provide better accuracy. They are named `__iar_xxx_accuratef` for `float` variants of the functions and `__iar_xxx_accuratel` for `long double` variants of the functions, and where *xxx* is `cos`, `sin`, etc.

To use any of these more accurate versions, use the `-e` linker option.

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 runtime environment are reentrant, but the following functions and parts are not reentrant because they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, etc. and the C++ operators `new` and `delete`
- Locale functions—`localeconv`, `setlocale`
- Multibyte functions—`mblen`, `mbrlen`, `mbrtowc`, `mbsrtowc`, `mbtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`
- Rand functions—`rand`, `srand`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- The miscellaneous functions `atexit`, `perror`, `strerror`, `strtok`

- Functions that use files or the heap in some way. This includes `scanf`, `sscanf`, `getchar`, `getwchar`, `putchar`, and `putwchar`. In addition, if you are using the options `--enable_multibyte` and `--dlib_config=Full`, the `printf` and `sprintf` functions (or any variants) can also use the heap.

For the CLIB library, the `qsort` function and functions that use files in some way are non-reentrant. This includes `printf`, `scanf`, `getchar`, and `putchar`. However, the functions `sprintf` and `sscanf` are reentrant.

Functions that can set `errno` are not reentrant, because an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it is read. This applies to math and string conversion functions, among others.

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.

THE LONGJMP FUNCTION

A `longjmp` is in effect a jump to a previously defined `setjmp`. Any variable length arrays or C++ objects residing on the stack during stack unwinding will not be destroyed. This can lead to resource leaks or incorrect application behavior.

DLIB runtime environment—implementation details

The DLIB runtime environment provides most of the important C and C++ standard library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of Standard C. The library supports most of the hosted functionality, but you must implement some of its base functionality. For additional information, see the chapter *Implementation-defined behavior for Standard C* in this guide.
- Standard C library definitions, for user programs.
- C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code, see 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 8051 features. See the chapter *Intrinsic functions* for more information.

In addition, the DLIB runtime environment includes some added C functionality, see *Added C functionality*, page 401.

C HEADER FILES

This section lists the C header files specific to the DLIB runtime environment. Header files may additionally contain target-specific definitions; these are documented in the chapter *Using C*.

This table lists the C header files:

Header file	Usage
`assert.h`	Enforcing assertions when functions execute
`complex.h`	Computing common complex mathematical functions
`ctype.h`	Classifying characters
`errno.h`	Testing error codes reported by library functions
`fenv.h`	Floating-point exception flags
`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
`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
`tgmath.h`	Type-generic mathematical functions
`time.h`	Converting between various time and date formats
`uchar.h`	Unicode functionality (IAR extension to Standard C)
`wchar.h`	Support for wide characters
`wctype.h`	Classifying wide characters

Table 42: Traditional Standard C header files—DLIB

C++ HEADER FILES

This section lists the C++ header files:

- The C++ library header files
  The header files that constitute the Embedded C++ library.
- The C++ standard template library (STL) header files
  The header files that constitute STL for the Extended Embedded C++ library.
- The C++ C header files
  The C++ header files that provide the resources from the C library.

The C++ library header files

This table lists the header files that can be used in Embedded C++:

Header file	Usage
`complex`	Defining a class that supports complex arithmetic
`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
`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 43: C++ header files

The C++ standard template library (STL) header files

The following table lists the standard template library (STL) header files that can be used in Extended Embedded C++:

Header file	Description
`algorithm`	Defines several common operations on sequences

Table 44: Standard template library header files

Header file	Description
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 44: Standard template library header files (Continued)

Using Standard C libraries in C++

The C++ library works in conjunction with some 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`.

This 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

Table 45: New Standard C header files—DLIB

Header file	Usage
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 45: New Standard C header files—DLIB (Continued)

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 DLIB runtime environment includes some added C functionality.

The following include files provide these features:

- `fenv.h`
- `stdio.h`
- `stdlib.h`
- `string.h`

fenv.h

In `fenv.h`, trap handling support for floating-point numbers is defined with the functions `fegettrapenable` and `fegettrapdisable`.

stdio.h

These functions provide additional I/O functionality:

`fdopen`	Opens a file based on a low-level file descriptor.

`fileno`	Gets the low-level file descriptor from the file descriptor (`FILE*`).
`__gets`	Corresponds to `fgets` on `stdin`.
`getw`	Gets a `wchar_t` character from `stdin`.
`putw`	Puts a `wchar_t` character to `stdout`.
`__ungetchar`	Corresponds to `ungetc` on `stdout`.
`__write_array`	Corresponds to `fwrite` on `stdout`.

string.h

These are the additional functions defined in `string.h`:

`strdup`	Duplicates a string on the heap.
`strcasecmp`	Compares strings case-insensitive.
`strncasecmp`	Compares strings case-insensitive and bounded.
`strnlen`	Bounded string length.

SYMBOLS USED INTERNALLY BY THE LIBRARY

The following symbols are used by the library, which means that they are visible in library source files, etc:

`__assignment_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__constrange()`

Determines the allowed range for a parameter to an intrinsic function and that the parameter must be of type `const`.

`__construction_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__has_constructor`, `__has_destructor`

These symbols determine properties for class objects and they function like the `sizeof` operator. The symbols are true when a class, base class, or member (recursively) has a user-defined constructor or destructor, respectively.

`__memory_of`

Determines the class memory. A class memory determines which memory a class object can reside in. This symbol can only occur in class definitions as a class memory.

Note: The symbols are reserved and should only be used by the library.

Use the compiler option `--predef_macros` to determine the value for any predefined symbols.

CLIB runtime environment—implementation details

The CLIB runtime environment provides most of the important C standard 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; see 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 8051 features. See the chapter *Intrinsic functions* for more information.

LIBRARY DEFINITIONS SUMMARY

This table lists the C header files specific to the CLIB runtime environment:

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`

Table 46: CLIB runtime environment header files

Header file	Description
stdio.h	Input/output
stdlib.h	General utilities
string.h	String handling

Table 46: CLIB runtime environment header files (Continued)

* The functions isxxx, toupper, and tolower declared in the header file ctype.h evaluate their argument more than once. This is not according to the ISO/ANSI standard.

8051-specific CLIB functions

This section lists the 8051–specific CLIB library functions declared in pgmspace.h that allow access to strings in code memory.

SPECIFYING READ AND WRITE FORMATTERS

You can override default formatters for the functions printf_P and scanf_P by editing the linker configuration file. Note that it is not possible to use the IDE for overriding the default formatter for the 8051-specific library routines.

To override the default printf_P formatter, type any of the following lines in your linker command file:

```
-e_small_write_P=_formatted_write_P
-e_medium_write_P=_formatted_write_P
```

To override the default scanf_P formatter, type the following line in your linker command file:

```
-e_medium_read_P=_formatted_read_P
```

Note: In the descriptions below, PGM_VOID_P is a symbol that expands to void const __code * or void const __far_code *, depending on the data model; and PGM_P is a symbol that expands to char const __code * or char const __far_code *, depending on the data model.

Segment reference

- Summary of segments
- Descriptions of segments

For more information about placement of segments, see the chapter *Linking your application*.

Summary of segments

The 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.

The table below lists the segments that are available in the 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.

Table 47: Segment summary

Segment	Description
BDATA_Z	Holds zero-initialized __bdata static and global variables.
BIT_N	Holds __no_init __bit static and global variables.
BREG	Holds the compiler's virtual bit register.
CHECKSUM	Holds the checksum generated by the linker.
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 data in far memory.
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.
FAR22_AN	Holds __far22 located uninitialized data.
FAR22_CODE	Holds code declared __far22_code.
FAR22_CODE_AC	Holds __far22_code located constant data.

Table 47: Segment summary (Continued)

Segment	Description
FAR22_CODE_C	Holds __far22_code constant data.
FAR22_CODE_N	Holds __no_init __far22_code static and global variables.
FAR22_HEAP	Holds the heap used for dynamically allocated data in far22 memory.
FAR22_I	Holds __far22 static and global initialized variables.
FAR22_ID	Holds initial values for __far22 static and global variables in FAR22_I.
FAR22_N	Holds __no_init __far22 static and global variables.
FAR22_ROM_AC	Holds __far22_rom located constant data.
FAR22_ROM_C	Holds __far22_rom constant data.
FAR22_Z	Holds zero-initialized __far22 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.
HUGE_HEAP	Holds the heap used for dynamically allocated data in huge memory.
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.

Table 47: Segment summary (Continued)

Segment	Description
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.
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 47: Segment summary (Continued)

Descriptions of segments

This section gives reference information about each segment.

The segments are placed in memory by the segment placement linker directives -Z and -P, for sequential and packed placement, respectively. Some segments cannot use packed placement, as their contents must be continuous. For information about these

directives, see *Using the -Z command for sequential placement*, page 129 and *Using the -P command for packed placement*, page 130, respectively.

For each segment, the segment memory type is specified, which indicates in which type of memory the segment should be placed; see *Segment memory type*, page 120.

For information about how to define segments in the linker configuration file, see *Linking your application*, page 127.

For more 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. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

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`. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

BANKED_CODE_EXT2_C

Description	Holds constant data, when using the Banked extended2 code model. This can include constant variables, string and aggregate literals, etc. 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 315.

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 315.

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	`0xFFF0-0xFFFF` in each 64-Kbyte block.
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. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

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 configuration 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 configuration 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. The contents of this segment is declared by the system 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 configuration 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

CHECKSUM

Description	Holds the checksum bytes generated by the linker. This segment also holds the `__checksum` symbol. Note that the size of this segment is affected by the linker option `-J`.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in ROM memory.
Access type	Read-only

CODE_AC

Description	Holds `__code` located constant data. The segment also holds `const` objects if the compiler option `--place_constants=code` has been specified.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

CODE_C

Description	Holds `__code` constant data. This can include constant variables, string and aggregate literals, etc. 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 315.

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 315.

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 configuration file, the `-Z` directive must be used.
Segment memory type	`CODE`

Memory placement — This segment must be placed at the address where the microcontroller starts executing after reset, which for the 8051 microcontroller is at the address `0x0`.

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.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

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 configuration 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 configuration 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. The contents of this segment is declared by the system 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 configuration 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	`XDATA`
Memory placement	This segment must be placed in External memory space.
Access type	Read-write
See also	For information about how to define this segment and its length in the linker configuration file, see *Setting up stack memory*, page 132.

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. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

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. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

FAR_CODE_C

Description	Holds `__far_code` constant data. This can include constant variables, string and aggregate literals, etc. 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 315.

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 315.

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`.

Segment memory type — `XDATA`

Memory placement — This segment can be placed anywhere in external data memory.

Access type — Read-write

See also — For information about how to define this segment and its length in the linker configuration file and for information about using the `new` and `delete` operators for a heap in far memory, see *Setting up heap memory*, page 135 and *New and Delete operators*, page 236.

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 configuration 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 configuration 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. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

FAR_ROM_C

Description	Holds `__far_rom` constant data. This can include constant variables, string and aggregate literals, etc. 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 315.

FAR_Z

Description Holds zero-initialized `__far` static and global variables. The contents of this segment is declared by the system startup code. 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 configuration 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 315.

FAR22_AN

Description Holds `__no_init __far22` located data.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

FAR22_CODE

Description Holds application code declared `__far22_code`.

Segment memory type `CODE`

Memory placement `0x0-0x3FFFFF`

Access type Read-only

FAR22_CODE_AC

Description	Holds `__far22_code` located constant data. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

FAR22_CODE_C

Description	Holds `__far22_code` constant data. This can include constant variables, string and aggregate literals, etc.
Segment memory type	`CODE`
Memory placement	`0x0-0x3FFFFF`
Access type	Read-only
See also	*--output, -o*, page 315.

FAR22_CODE_N

Description	Holds static and global `__no_init __far22_code` variables.
Segment memory type	`CODE`
Memory placement	`0x0-0x3FFFFF`
Access type	Read-only
See also	

FAR22_HEAP

Description	Holds the heap used for dynamically allocated data in far22 memory, in other words data allocated by `far22_malloc` and `far22_free`, and in C++, `new` and `delete`.
Segment memory type	`XDATA`
Memory placement	`0x0-0x3FFFFF`

Access type	Read-write
See also	For information about how to define this segment and its length in the linker configuration file and for information about using the `new` and `delete` operators for a heap in far memory, see *Setting up heap memory*, page 135 and *New and Delete operators*, page 236.

FAR22_I

Description	Holds `__far22` static and global initialized variables initialized by copying from the segment `FAR22_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 configuration file, the `-Z` directive must be used.
Segment memory type	`XDATA`
Memory placement	`0x0-0x3FFFF`
Access type	Read-only

FAR22_ID

Description	Holds initial values for `__far22` static and global variables in the `FAR22_I` segment. These values are copied from `FAR22_ID` to `FAR22_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 configuration 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

FAR22_N

Description	Holds static and global `__no_init __far22` variables.
Segment memory type	`CONST`

Memory placement	`0x0-0x3FFFFF`
Access type	Read-only

FAR22_ROM_AC

Description	Holds `__far22_rom` located constant data. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

FAR22_ROM_C

Description	Holds `__far22_rom` constant data. This can include constant variables, string and aggregate literals, etc.
Segment memory type	`CONST`
Memory placement	`0x0-0x3FFFFF`
Access type	Read-only
See also	*--output, -o*, page 315.

FAR22_Z

Description	Holds zero-initialized `__far22` static and global variables. The contents of this segment is declared by the system 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 configuration file, the `-Z` directive must be used.
Segment memory type	`XDATA`
Memory placement	`0x0-0x3FFFFF`
Access type	Read-write
See also	*--output, -o*, page 315.

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.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

HUGE_CODE_AC

Description

Holds `__huge_code` located constant data.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

HUGE_CODE_C

Description

Holds `__huge_code` constant data. This can include constant variables, string and aggregate literals, etc.

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`.
Segment memory type	`XDATA`
Memory placement	This segment can be placed anywhere in the external data memory space.
Access type	Read-write
See also	For information about how to define this segment and its length in the linker configuration file and for information about using the `new` and `delete` operators for a heap in huge memory, see *Setting up heap memory*, page 135 and *New and Delete operators*, page 236.

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 configuration 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 configuration 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. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

HUGE_ROM_C

Description	Holds `__huge_rom` constant data. This can include constant variables, string and aggregate literals, etc.
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. The contents of this segment is declared by the system startup code. 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 configuration 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. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

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 configuration 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 configuration 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. The contents of this segment is declared by the system startup code. 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 configuration file, the `-Z` directive must be used.
Segment memory type	`DATA`
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	This segment can be placed anywhere in memory.
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	`DATA`

Memory placement `0x0-0xFF`

Access type Read-write

See also For information about how to define this segment and its length in the linker configuration file, see *Setting up stack memory*, page 132.

IXDATA_AN

Description Holds `__no_init __ixdata` located data.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

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 configuration 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 configuration 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. The contents of this segment is declared by the system startup code. 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 configuration 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.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

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 configuration 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 configuration 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. The contents of this segment is declared by the system 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 configuration 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 — For information about setting up stack pointers, see *System startup*, page 164.

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 — For information about how to define this segment and its length in the linker configuration file, see *Setting up stack memory*, page 132.

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.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

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. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

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`.
Segment memory type	`XDATA`
Memory placement	This segment can be placed anywhere in the external data memory space.
Access type	Read-write
See also	For information about how to define this segment and its length in the linker configuration file and for information about using the `new` and `delete` operators for a heap in xdata memory, see *Setting up heap memory*, page 135 and *New and Delete operators*, page 236.

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 configuration 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 configuration 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 316.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

XDATA_ROM_C

Description

Holds `__xdata_rom` constant data. This can include constant variables, string and aggregate literals, etc.

Segment memory type

`CONST`

Memory placement

`0–0xFFFF`

Access type

Read-only

XDATA_Z

Description

Holds zero-initialized `__huge` static and global variables. The contents of this segment is declared by the system startup code. 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 configuration file, the `-Z` directive must be used.

Segment memory type

`XDATA`

Memory placement

`0–0xFFFF`

Access type

Read-write

See also

--output, -o, page 315.

XSP

Description	Holds the stack pointers to the xdata stack.
Segment memory type	`DATA`
Memory placement	`0–0x7F`
Access type	Read-write
See also	For information about setting up stack pointers, see *System startup*, page 164.

XSTACK

Description	Holds the xdata stack.
Segment memory type	`XDATA`
Memory placement	`0–0FFFF`
Access type	Read-write
See also	For information about how to define this segment and its length in the linker configuration file, see *Setting up stack memory*, page 132.

Implementation-defined behavior for Standard C

- Descriptions of implementation-defined behavior

If you are using C89 instead of Standard C, see *Implementation-defined behavior for C89*, page 457. For a short overview of the differences between Standard C and C89, see *C language overview*, page 221.

The text in this chapter applies to the DLIB runtime environment. Because the CLIB runtime environment does not follow Standard C, its implementation-defined behavior is not documented. See also *The CLIB runtime environment*, page 181.

Descriptions of implementation-defined behavior

This section follows the same order as the C standard. Each item includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

Note: The IAR Systems implementation adheres to a freestanding implementation of Standard C. This means that parts of a standard library can be excluded in the implementation.

J.3.1 TRANSLATION

Diagnostics (3.10, 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.

White-space characters (5.1.1.2)

At translation phase three, each non-empty sequence of white-space characters is retained.

J.3.2 ENVIRONMENT

The character set (5.1.1.2)

The source character set is the same as the physical source file multibyte character set. By default, the standard ASCII character set is used. However, if you use the `--enable_multibytes` compiler option, the host character set is used instead.

Main (5.1.2.1)

The function called at program startup is called `main`. No prototype is declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the DLIB runtime environment, see *System initialization*, page 167.

The effect of program termination (5.1.2.1)

Terminating the application returns the execution to the startup code (just after the call to `main`).

Alternative ways to define main (5.1.2.2.1)

There is no alternative ways to define the `main` function.

The argv argument to main (5.1.2.2.1)

The `argv` argument is not supported.

Streams as interactive devices (5.1.2.3)

The streams `stdin`, `stdout`, and `stderr` are treated as interactive devices.

Signals, their semantics, and the default handling (7.14)

In the DLIB runtime environment, the set of supported signals is the same as in Standard C. A raised signal will do nothing, unless the `signal` function is customized to fit the application.

Signal values for computational exceptions (7.14.1.1)

In the DLIB runtime environment, there are no implementation-defined values that correspond to a computational exception.

Signals at system startup (7.14.1.1)

In the DLIB runtime environment, there are no implementation-defined signals that are executed at system startup.

Environment names (7.20.4.5)

In the DLIB runtime environment, there are no implementation-defined environment names that are used by the `getenv` function.

The system function (7.20.4.6)

The `system` function is not supported.

J.3.3 IDENTIFIERS

Multibyte characters in identifiers (6.4.2)

Additional multibyte characters may not appear in identifiers.

Significant characters in identifiers (5.2.4.1, 6.1.2)

The number of significant initial characters in an identifier with or without external linkage is guaranteed to be no less than 200.

J.3.4 CHARACTERS

Number of bits in a byte (3.6)

A byte contains 8 bits.

Execution character set member values (5.2.1)

The values of the members of the execution character set are the values of the ASCII character set, which can be augmented by the values of the extra characters in the host character set.

Alphabetic escape sequences (5.2.2)

The standard alphabetic escape sequences have the values `\a-7`, `\b-8`, `\f-12`, `\n-10`, `\r-13`, `\t-9`, and `\v-11`.

Characters outside of the basic executive character set (6.2.5)

A character outside of the basic executive character set that is stored in a `char` is not transformed.

Plain char (6.2.5, 6.3.1.1)

A plain `char` is treated as an `unsigned char`.

Source and execution character sets (6.4.4.4, 5.1.1.2)

The source character set is the set of legal characters that can appear in source files. By default, the 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. By default, the 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 DLIB runtime environment needs a multibyte character scanner to support a multibyte execution character set. See *Locale*, page 178.

Integer character constants with more than one character (6.4.4.4)

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.

Wide character constants with more than one character (6.4.4.4)

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Locale used for wide character constants (6.4.4.4)

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

Locale used for wide string literals (6.4.5)

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

Source characters as executive characters (6.4.5)

All source characters can be represented as executive characters.

J.3.5 INTEGERS

Extended integer types (6.2.5)

There are no extended integer types.

Range of integer values (6.2.6.2)

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.

For information about the ranges for the different integer types, see *Basic data types—integer types*, page 326.

The rank of extended integer types (6.3.1.1)

There are no extended integer types.

Signals when converting to a signed integer type (6.3.1.3)

No signal is raised when an integer is converted to a signed integer type.

Signed bitwise operations (6.5)

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, except for the operator >> which will behave as an arithmetic right shift.

J.3.6 FLOATING POINT

Accuracy of floating-point operations (5.2.4.2.2)

The accuracy of floating-point operations is unknown.

Rounding behaviors (5.2.4.2.2)

There are no non-standard values of `FLT_ROUNDS`.

Evaluation methods (5.2.4.2.2)

There are no non-standard values of `FLT_EVAL_METHOD`.

Converting integer values to floating-point values (6.3.1.4)

When an integral value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Converting floating-point values to floating-point values (6.3.1.5)

When a floating-point value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Denoting the value of floating-point constants (6.4.4.2)

The round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Contraction of floating-point values (6.5)

Floating-point values are contracted. However, there is no loss in precision and because signaling is not supported, this does not matter.

Default state of FENV_ACCESS (7.6.1)

The default state of the pragma directive `FENV_ACCESS` is `OFF`.

Additional floating-point mechanisms (7.6, 7.12)

There are no additional floating-point exceptions, rounding-modes, environments, and classifications.

Default state of FP_CONTRACT (7.12.2)

The default state of the pragma directive `FP_CONTRACT` is `OFF`.

J.3.7 ARRAYS AND POINTERS

Conversion from/to pointers (6.3.2.3)

For information about casting of data pointers and function pointers, see *Casting*, page 333.

ptrdiff_t (6.5.6)

For information about `ptrdiff_t`, see *ptrdiff_t*, page 333.

J.3.8 HINTS

Honoring the register keyword (6.7.1)

User requests for register variables are not honored.

Inlining functions (6.7.4)

User requests for inlining functions increases the chance, but does not make it certain, that the function will actually be inlined into another function. See *Inlining functions*, page 103.

J.3.9 STRUCTURES, UNIONS, ENUMERATIONS, AND BITFIELDS

Sign of 'plain' bitfields (6.7.2, 6.7.2.1)

For information about how a 'plain' `int` bitfield is treated, see *Bitfields*, page 327.

Possible types for bitfields (6.7.2.1)

All integer types can be used as bitfields in the compiler's extended mode, see *-e*, page 302.

Bitfields straddling a storage-unit boundary (6.7.2.1)

A bitfield is always placed in one—and one only—storage unit, which means that the bitfield cannot straddle a storage-unit boundary.

Allocation order of bitfields within a unit (6.7.2.1)

For information about how bitfields are allocated within a storage unit, see *Bitfields*, page 327.

Alignment of non-bitfield structure members (6.7.2.1)

The alignment of non-bitfield members of structures is the same as for the member types, see *Alignment*, page 325.

Integer type used for representing enumeration types (6.7.2.2)

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.

J.3.10 QUALIFIERS

Access to volatile objects (6.7.3)

Any reference to an object with `volatile` qualified type is an access, see *Declaring objects volatile*, page 334.

J.3.11 PREPROCESSING DIRECTIVES

Mapping of header names (6.4.7)

Sequences in header names are mapped to source file names verbatim. A backslash '\' is not treated as an escape sequence. See *Overview of the preprocessor*, page 387.

Character constants in constant expressions (6.10.1)

A character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set.

The value of a single-character constant (6.10.1)

A single-character constant may only have a negative value if a plain character (`char`) is treated as a signed character, see *--char_is_signed*, page 292.

Including bracketed filenames (6.10.2)

For information about the search algorithm used for file specifications in angle brackets <>, see *Include file search procedure*, page 280.

Including quoted filenames (6.10.2)

For information about the search algorithm used for file specifications enclosed in quotes, see *Include file search procedure*, page 280.

Preprocessing tokens in #include directives (6.10.2)

Preprocessing tokens in an `#include` directive are combined in the same way as outside an `#include` directive.

Nesting limits for #include directives (6.10.2)

There is no explicit nesting limit for `#include` processing.

Universal character names (6.10.3.2)

Universal character names (UCN) are not supported.

Recognized pragma directives (6.10.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect. If a pragma directive is listed both in the chapter *Pragma directives* and here, the information provided in the chapter *Pragma directives* overrides the information here.

```
alignment

baseaddr

building_runtime

can_instantiate

codeseg

cspy_support

define_type_info

do_not_instantiate

early_dynamic_initialization

function

function_effects

hdrstop

important_typedef

instantiate

keep_definition

library_default_requirements

library_provides

library_requirement_override

memory

module_name

no_pch

once

system_include

warnings
```

Default __DATE__ and __TIME__ (6.10.8)

The definitions for `__TIME__` and `__DATE__` are always available.

J.3.12 LIBRARY FUNCTIONS

Additional library facilities (5.1.2.1)

Most of the standard library facilities are supported. Some of them—the ones that need an operating system—require a low-level implementation in the application. For more information, see *The DLIB runtime environment*, page 145.

Diagnostic printed by the assert function (7.2.1.1)

The `assert()` function prints:

`filename``:`*`linenr expression`* `-- assertion failed`

when the parameter evaluates to zero.

Representation of the floating-point status flags (7.6.2.2)

For information about the floating-point status flags, see *fenv.h*, page 401.

Feraiseexcept raising floating-point exception (7.6.2.3)

For information about the `feraiseexcept` function raising floating-point exceptions, see *Floating-point environment*, page 329.

Strings passed to the setlocale function (7.11.1.1)

For information about strings passed to the `setlocale` function, see *Locale*, page 178.

Types defined for float_t and double_t (7.12)

The `FLT_EVAL_METHOD` macro can only have the value `0`.

Domain errors (7.12.1)

No function generates other domain errors than what the standard requires.

Return values on domain errors (7.12.1)

Mathematic functions return a floating-point `NaN` (not a number) for domain errors.

Underflow errors (7.12.1)

Mathematic functions set `errno` to the macro `ERANGE` (a macro in `errno.h`) and return zero for underflow errors.

fmod return value (7.12.10.1)

The `fmod` function returns a floating-point `NaN` when the second argument is zero.

The magnitude of remquo (7.12.10.3)

The magnitude is congruent modulo `INT_MAX`.

signal() (7.14.1.1)

The signal part of the library is not supported.

Note: The default implementation of `signal` does not perform anything. Use the template source code to implement application-specific signal handling. See *signal*, page 174 and *raise*, page 172, respectively.

NULL macro (7.17)

The `NULL` macro is defined to `0`.

Terminating newline character (7.19.2)

`stdout` stream functions recognize either `newline` or `end of file (EOF)` as the terminating character for a line.

Space characters before a newline character (7.19.2)

Space characters written to a stream immediately before a newline character are preserved.

Null characters appended to data written to binary streams (7.19.2)

No null characters are appended to data written to binary streams.

File position in append mode (7.19.3)

The file position is initially placed at the beginning of the file when it is opened in append-mode.

Truncation of files (7.19.3)

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 *Briefly about input and output (I/O)*, page 146.

File buffering (7.19.3)

An open file can be either block-buffered, line-buffered, or unbuffered.

A zero-length file (7.19.3)

Whether a zero-length file exists depends on the application-specific implementation of the low-level file routines.

Legal file names (7.19.3)

The legality of a filename depends on the application-specific implementation of the low-level file routines.

Number of times a file can be opened (7.19.3)

Whether a file can be opened more than once depends on the application-specific implementation of the low-level file routines.

Multibyte characters in a file (7.19.3)

The encoding of multibyte characters in a file depends on the application-specific implementation of the low-level file routines.

remove() (7.19.4.1)

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *Briefly about input and output (I/O)*, page 146.

rename() (7.19.4.2)

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *Briefly about input and output (I/O)*, page 146.

Removal of open temporary files (7.19.4.3)

Whether an open temporary file is removed depends on the application-specific implementation of the low-level file routines.

Mode changing (7.19.5.4)

`freopen` closes the named stream, then reopens it in the new mode. The streams `stdin`, `stdout`, and `stderr` can be reopened in any new mode.

Style for printing infinity or NaN (7.19.6.1, 7.24.2.1)

The style used for printing infinity or `NaN` for a floating-point constant is `inf` and `nan` (`INF` and `NAN` for the `F` conversion specifier), respectively. The n-char-sequence is not used for `nan`.

%p in printf() (7.19.6.1, 7.24.2.1)

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.

Reading ranges in scanf (7.19.6.2, 7.24.2.1)

A - (dash) character is always treated as a range symbol.

%p in scanf (7.19.6.2, 7.24.2.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.

File position errors (7.19.9.1, 7.19.9.3, 7.19.9.4)

On file position errors, the functions `fgetpos`, `ftell`, and `fsetpos` store `EFPOS` in `errno`.

An n-char-sequence after nan (7.20.1.3, 7.24.4.1.1)

An n-char-sequence after a NaN is read and ignored.

errno value at underflow (7.20.1.3, 7.24.4.1.1)

`errno` is set to `ERANGE` if an underflow is encountered.

Zero-sized heap objects (7.20.3)

A request for a zero-sized heap object will return a valid pointer and not a null pointer.

Behavior of abort and exit (7.20.4.1, 7.20.4.4)

A call to `abort()` or `_Exit()` will not flush stream buffers, not close open streams, and not remove temporary files.

Termination status (7.20.4.1, 7.20.4.3, 7.20.4.4)

The termination status will be propagated to `__exit()` as a parameter. `exit()` and `_Exit()` use the input parameter, whereas abort uses `EXIT_FAILURE`.

The system function return value (7.20.4.6)

The `system` function is not supported.

The time zone (7.23.1)

The local time zone and daylight savings time must be defined by the application. For more information, see *__time32*, page 175.

Range and precision of time (7.23)

The *time* interface supports years from 1900 up to 2035 and uses a 32-bit integer for `time_t`. The application must supply the actual implementation for the functions `time` and `clock`. See *__time32*, page 175 and *clock*, page 170, respectively.

clock() (7.23.2.1)

The application must supply an implementation of the `clock` function. See *clock*, page 170.

%Z replacement string (7.23.3.5, 7.24.5.1)

By default, ":" is used as a replacement for `%Z`. Your application should implement the time zone handling. See *__time32*, page 175.

Math functions rounding mode (F.9)

The functions in `math.h` honor the rounding direction mode in `FLT-ROUNDS`.

J.3.13 ARCHITECTURE

Values and expressions assigned to some macros (5.2.4.2, 7.18.2, 7.18.3)

There are always 8 bits in a byte.

`MB_LEN_MAX` is at the most 6 bytes depending on the library configuration that is used.

For information about sizes, ranges, etc for all basic types, see *Data representation*, page 325.

The limit macros for the exact-width, minimum-width, and fastest minimum-width integer types defined in `stdint.h` have the same ranges as `char`, `short`, `int`, `long`, and `long long`.

The floating-point constant `FLT_ROUNDS` has the value 1 (to nearest) and the floating-point constant `FLT_EVAL_METHOD` has the value 0 (treat as is).

The number, order, and encoding of bytes (6.2.6.1)

See *Data representation*, page 325.

The value of the result of the sizeof operator (6.5.3.4)

See *Data representation*, page 325.

J.4 LOCALE

Members of the source and execution character set (5.2.1)

By default, the compiler accepts all one-byte characters in the host's default character set. If the compiler option `--enable_multibytes` is used, the host multibyte characters are accepted in comments and string literals as well.

The meaning of the additional character set (5.2.1.2)

Any multibyte characters in the extended source character set is translated verbatim into the extended execution character set. It is up to your application with the support of the library configuration to handle the characters correctly.

Shift states for encoding multibyte characters (5.2.1.2)

Using the compiler option `--enable_multibytes` enables the use of the host's default multibyte characters as extended source characters.

Direction of successive printing characters (5.2.2)

The application defines the characteristics of a display device.

The decimal point character (7.1.1)

The default decimal-point character is a '`.`'. You can redefine it by defining the library configuration symbol `_LOCALE_DECIMAL_POINT`.

Printing characters (7.4, 7.25.2)

The set of printing characters is determined by the chosen locale.

Control characters (7.4, 7.25.2)

The set of control characters is determined by the chosen locale.

Characters tested for (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.5.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11)

The sets of characters tested are determined by the chosen locale.

The native environment (7.1.1.1)

The native environment is the same as the "C" locale.

Subject sequences for numeric conversion functions (7.20.1, 7.24.4.1)

There are no additional subject sequences that can be accepted by the numeric conversion functions.

The collation of the execution character set (7.21.4.3, 7.24.4.4.2)

The collation of the execution character set is determined by the chosen locale.

Message returned by strerror (7.21.6.2)

The messages returned by the `strerror` function 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 48: Message returned by strerror()—DLIB runtime environment

Implementation-defined behavior for C89

- Descriptions of implementation-defined behavior

If you are using Standard C instead of C89, see *Implementation-defined behavior for Standard C*, page 441. For a short overview of the differences between Standard C and C89, see *C language overview*, page 221.

Descriptions of implementation-defined behavior

The descriptions follow the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

TRANSLATION

Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

```
filename,linenumber level[tag]: message
```

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

ENVIRONMENT

Arguments to main (5.1.2.2.1)

The function called at program startup is called `main`. No prototype was declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the DLIB runtime environment, see *System initialization*, page 167. To change this behavior for the CLIB runtime environment, see *Customizing system initialization*, page 188.

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 DLIB runtime environment needs a multibyte character scanner to support a multibyte execution character set. The CLIB runtime environment does not support multibyte characters.

See *Locale*, page 178.

Bits per character in execution character set (5.2.4.2.1)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

Mapping of characters (6.1.3.4)

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the

same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

Unrepresented character constants (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

Character constant with more than one character (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Converting multibyte characters (6.1.3.4)

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compiler—is the 'C' locale. If you use the command line option `--enable_multibytes`, the DLIB runtime environment will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library. The CLIB runtime environment does not support multibyte characters.

See *Locale*, page 178.

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—integer types*, page 326, 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, except for the operator >> which will behave as an arithmetic right shift.

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 *Basic data types—floating-point types*, page 329, 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 333, for information about `size_t`.

Conversion from/to pointers (6.3.4)

See *Casting*, page 333, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 333, 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—integer types*, page 326, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)

A 'plain' `int` bitfield is treated as a `signed int` bitfield. All integer types are allowed as bitfields.

Allocation order of bitfields within a unit (6.5.2.1)

Bitfields are allocated within an integer from least-significant to most-significant bit.

Can bitfields straddle a storage-unit boundary (6.5.2.1)

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

Integer type chosen to represent enumeration types (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

QUALIFIERS

Access to volatile objects (6.5.3)

Any reference to an object with volatile qualified type is an access.

DECLARATORS

Maximum numbers of declarators (6.5.4)

The number of declarators is not limited. The number is limited only by the available memory.

STATEMENTS

Maximum number of case statements (6.6.4.2)

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

PREPROCESSING DIRECTIVES

Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a `signed` character.

Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source

file currently being processed. If there is no grandparent file and the file is not found, the search continues as if the filename was enclosed in angle brackets.

Character sequences (6.8.2)

Preprocessor directives use the source character set, except for escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile","rt");
```

Recognized pragma directives (6.8.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect. If a pragma directive is listed both in the chapter *Pragma directives* and here, the information provided in the chapter *Pragma directives* overrides the information here.

```
alignment

baseaddr

building_runtime

can_instantiate

codeseg

cspy_support

define_type_info

do_not_instantiate

early_dynamic_initialization

function

function_effects

hdrstop

important_typedef

instantiate

keep_definition

library_default_requirements

library_provides
```

```
library_requirement_override

memory

module_name

no_pch

once

system_include

warnings
```

Default __DATE__ and __TIME__ (6.8.8)

The definitions for `__TIME__` and `__DATE__` are always available.

LIBRARY FUNCTIONS FOR THE IAR DLIB RUNTIME ENVIRONMENT

Note that some items in this list only apply when file descriptors are supported by the library configuration. 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: The default implementation of `signal` does not perform anything. Use the template source code to implement application-specific signal handling. See *signal*, page 174 and *raise*, page 172, respectively.

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 the file position indicator of an append-mode stream is initially positioned at the beginning or the end of the file, depends on the application-specific implementation of the low-level file routines.

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 *Briefly about input and output (I/O)*, page 146.

The characteristics of the file buffering is that the implementation supports files that are unbuffered, line buffered, or fully buffered.

Whether a zero-length file actually exists depends on the application-specific implementation of the low-level file routines.

Rules for composing valid file names depends on the application-specific implementation of the low-level file routines.

Whether the same file can be simultaneously open multiple times depends on the application-specific implementation of the low-level file routines.

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 *Briefly about input and output (I/O)*, page 146.

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 *Briefly about input and output (I/O)*, page 146.

%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 *getenv*, page 170.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *system*, page 175.

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 49: Message returned by strerror()—DLIB runtime environment

The time zone (7.12.1)

The local time zone and daylight savings time implementation is described in *__time32*, page 175.

clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *clock*, page 170.

LIBRARY FUNCTIONS FOR THE CLIB RUNTIME ENVIRONMENT

NULL macro (7.1.6)

The `NULL` macro is defined to `(void *) 0`.

Diagnostic printed by the assert function (7.2)

The `assert()` function prints:

`Assertion failed:` *`expression`*`, file` *`Filename`*`, line` *`linenumber`*

when the parameter evaluates to zero.

Domain errors (7.5.1)

`HUGE_VAL`, the largest representable value in a double floating-point type, will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets errno to ERANGE (7.5.1)

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

fmod() functionality (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns zero (it does not change the integer expression `errno`).

signal() (7.7.1.1)

The signal part of the library is not supported.

Terminating newline character (7.9.2)

`stdout` stream functions recognize either `newline` or `end of file (EOF)` as the terminating character for a line.

Blank lines (7.9.2)

Space characters written to the `stdout` stream immediately before a newline character are preserved. There is no way to read the line through the `stdin` stream that was written through the `stdout` stream.

Null characters appended to data written to binary streams (7.9.2)

There are no binary streams implemented.

Files (7.9.3)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

remove() (7.9.4.1)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

rename() (7.9.4.2)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

%p in printf() (7.9.6.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `'char *'`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

%p in scanf() (7.9.6.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `'void *'`.

Reading ranges in scanf() (7.9.6.2)

A - (dash) character is always treated explicitly as a - character.

File position errors (7.9.9.1, 7.9.9.4)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

Message generated by perror() (7.9.10.4)

`perror()` is not supported.

Allocating zero bytes of memory (7.10.3)

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`.

Behavior of abort() (7.10.4.1)

The `abort()` function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

Behavior of exit() (7.10.4.3)

The `exit()` function does not return.

Environment (7.10.4.4)

Environments are not supported.

system() (7.10.4.5)

The `system()` function is not supported.

Message returned by strerror() (7.11.6.2)

The messages returned by `strerror()` depending on the argument are:

Argument	Message
`EZERO`	`no error`
`EDOM`	`domain error`
`ERANGE`	`range error`
`<0 || >99`	`unknown error`
all others	`error No.`*`xx`*

Table 50: Message returned by strerror()—CLIB runtime environment

The time zone (7.12.1)

The time zone function is not supported.

clock() (7.12.2.1)

The `clock()` function is not supported.

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Symbols

Numerics