IAR C/C++ Compiler User Guide

for the 8051 Microcontroller Architecture

IAR C/C++ 编译器用户指南

对于8051 微控制器架构

C8051-7

DISCLAIMER

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

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

TRADEMARKS

IAR Systems, IAR Embedded Workbench, IAR Connect, C-SPY, C-RUN, C-STAT, visualSTATE, IAR KickStart Kit, IAR Experiment!, I-jet, I-jet Trace, I-scope, IAR Academy, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation. Intel® is a registered trademark of Intel Corporation.

Adobe and Acrobat Reader are registered trademarks of Adobe Systems Incorporated.

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

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.

版权声明

免责声明

本文档中的信息如有更改，恕不另行通知，并不代表 IAR Systems 任何部分的承诺。尽管此处包含的信息被认 为是准确的，但 IAR Systems 对任何错误或遗漏不承担任何责任。

在任何情况下，IAR Systems、其员工、其承包商或本文档的作者均不对特殊、直接、间接或后果性损害、损失、成本、收费、索赔、要求、利润损失、费用或任何性质或种类的费用。

商标

IAR Systems, IAR Embedded Workbench, IAR Connect, C-SPY, C-RUN, C-STAT, visualSTATE, IAR KickStart Kit, IAR Experiment!, I-jet, I-jet Trace, I-scope, IAR Academy, IAR,和 IAR Systems 的标识是 IAR Systems AB 拥有的商标或注册商标。

Microsoft 和 Windows 是 微软公司的注册商标。

INTEL® 是英特尔公司的注册商标。

Adobe 和 Acrobat Reader 是 Adobe Systems Incorporated 的注册商标。

所有其他产品名称均为其各自的商标或注册商标拥有者。

版本通知

第七版：2017 年 3 月

元件编号：C8051-7

本指南适用于 8051 微控制器架构的 IAR Embedded Workbench® 10.x 版本。

内部参考：M22、Mym8.0、tut2017.1、csrct2010.1、V_110411、IJOA。

简要内容Brief contents

前言

欢迎阅读 8051 的 IAR C/C++ 编译器用户指南。本指南是为您提供可以帮助您的详细参考信息，使用编译器以最适合您的应用程序要求。所以本指南为您提供有关编码技术的建议，以便您开发效率最高的应用。

本指南的阅读对象

如果您计划使用 C 或 C++ 语言开发应用程序,需要了解有关如何使用编译器的详细参考信息，请阅读本 8051 微控制器编译器用户指南。

所需知识

要使用 IAR Embedded Workbench 中的工具，您应该具备以下工作知识：

- 8051 单片机的架构和指令集（参考芯片制造商的文件）
- C 或 C++ 编程语言
- 嵌入式系统应用开发
- 主机的操作系统。

有关集成在 IDE 中的其他开发工具的更多信息，请参阅请参阅其他文档，第 33 页。

如何使用本指南

当您开始使用 8051 的 IAR C/C++ 编译器时，您应该阅读第 1 部分使用本指南中的编译器。
当您熟悉编译器并已配置您的项目时，您可以更多地关注第 2 部分参考信息。
如果您不熟悉使用此产品，我们建议您首先阅读 IAR Embedded Workbench® 入门指南，
以了解 IDE 提供的工具和功能的概述。
您可以在 IAR 信息中心找到这些教程，它们将帮助您开始使用 IAR Embedded Workbench。

页码:31

本指南包含的内容

以下是本指南各章的简要大纲和摘要。

第 1 部分. 使用编译器

● IAR 构建工具介绍介绍 IAR 构建工具，其中包括工具概述、编程语言、可用的设备支持，以及为支持 8051 的特定功能而提供的扩展微控制器。

● 开发嵌入式应用程序为您提供入门所需的信息使用 IAR 构建工具开发您的嵌入式软件。

● 了解内存架构，第 59 页概述了 8051 根据不同存储空间的微控制器内存配置可用的。本章还概述了与内存相关的概念在编译器和链接器中可用。

● 数据存储介绍如何将数据存储在内存中。

● Functions 简要概述了与函数相关的扩展--控制功能--并更详细地描述了其中一些机制。

● 分页功能介绍分页技术；何时使用它，它做什么，以及这个怎么运作。

● 链接概述描述了使用 IAR XLINK 链接器的链接过程和相关的概念。

● 链接您的应用程序列出了链接您的应用程序时必须考虑的方面应用程序，包括使用 XLINK 选项和定制链接器配置文件。

● DLIB 运行环境描述了 DLIB 运行环境，其中到应用程序执行。它涵盖了如何通过设置选项来修改它，覆盖默认库模块，或构建您自己的库。所以这一章描述系统初始化介绍文件 cstartup，如何使用模块用于语言环境和文件 I/O。

● CLIB 运行时环境概述了 CLIB 运行时库和如何自定义它们。本章还介绍了系统初始化和介绍文件 cstartup。

● 汇编语言界面包含当部分应用程序是用汇编语言编写的。这包括调用约定。

● 使用 C 概述了 C 语言的两个受支持变体和一个编译器扩展的概述，例如标准 C 的扩展。

● 使用 C++ 概述了两个级别的 C++ 支持：行业标准 EC++ 和 IAR 扩展 EC++。

● 与应用程序相关的注意事项讨论了选定范围的应用程序问题与使用编译器和链接器有关。

● 嵌入式应用程序的高效编码提供了有关如何编写代码的提示编译为嵌入式应用程序的有效代码。

第 2 部分. 参考信息

● 外部接口详细信息提供有关如何编译器的参考信息与其环境交互--调用语法、传递选项的方法到编译器、环境变量、包含文件搜索过程和不同类型的编译器输出。本章还描述了编译器如何诊断系统工作。

● 编译器选项说明如何设置选项，提供选项摘要，以及包含每个编译器选项的详细参考信息。

● 数据表示描述了可用的数据类型、指针和结构类型。本章还提供有关类型和对象属性的信息。

● 扩展关键字提供有关每个 8051 特定的参考信息关键字是标准 C/C++ 语言的扩展。

● Pragma 预编译指令提供有关 Pragma 预编译指令的参考信息。

● 内在函数提供有关用于访问的函数的参考信息 8051 特定的低级功能。

● 预处理器提供了预处理器的简要概述，包括参考有关不同预处理器指令、符号和其他相关的信息。

● C/C++ 标准库函数介绍了 C 或 C++ 库函数，并总结了头文件。

● 段引用提供有关编译器使用的参考信息段。

● 标准 C 的实现定义行为描述了编译器如何处理标准 C 的实现定义区域。

● C89 的实现定义的行为描述了编译器如何处理 C 语言标准 C89 的实现定义区域。

其他文件

用户文档以超文本 PDF 形式提供，并且作为上下文相关在线提供 HTML 格式的帮助系统。
您可以从信息中访问文档中心或从 IAR Embedded Workbench IDE 中的帮助菜单。
也可通过 F1 键使用在线帮助系统。

页码:33

用户和参考指南

一系列指南中描述了完整的 IAR Systems 开发工具集。

关于：

● 系统要求和有关如何安装和注册 IAR 的信息系统产品，可在小册子快速参考（可在产品盒）和安装和许可指南。

● IAR Embedded Workbench 及其提供的工具入门 IAR Embedded Workbench® 入门指南中提供。

● 使用 IDE 进行项目管理和构建，在 IDE Project 中可用 8051 管理和建筑指南。

● 使用 IAR C-SPY® 调试器，可在 C-SPY® 调试指南中找到为 8051。

● IAR C/C++ Compiler for 8051 的编程，在 IAR C/C++ 中可用 8051 编译器用户指南。

● 使用 IAR XLINK 链接器、IAR XARLibrary Builder 和 IAR XLIBLibrarian，可在 IAR 链接器和库工具参考指南中找到。

● 8051 的 IAR 汇编器编程，在 IAR 汇编器中可用 8051 用户指南。

● 使用 C-STAT 执行静态分析，所需的检查可在 C-STAT® 静态分析指南。

● 可以使用 MISRA C 指南开发安全关键型应用程序在 IAR Embedded Workbench® MISRA C:2004 参考指南或 IAR Embedded Workbench® MISRA C:1998 参考指南。

● 移植使用先前版本的 IAR 创建的应用程序代码和项目用于 8051 的嵌入式工作台，可在 IAR Embedded Workbench® 中使用迁移指南。

注意：可能会提供其他文档，具体取决于您的产品安装。

在线帮助系统

上下文相关的联机帮助包含：

● IDE 中有关项目管理、编辑和构建的信息

● 有关使用 IAR C-SPY® 调试器进行调试的信息

● IDE 中的菜单、窗口和对话框的参考信息

● 编译器参考信息

● DLILibrary 函数的关键字参考信息。获取参考

功能信息，在编辑器窗口中选择功能名称并按 F1 请注意，如果您在编辑器窗口中选择一个函数名称并在按下 F1 的同时使用 CLIB C 标准库，您将获得 DLIB 的参考信息 C/EC++ 标准库。

进一步阅读

在使用 IAR Systems 开发工具时，您可能会对这些书籍感兴趣：

- Barr、Michael 和 Andy Oram 编辑，用 C 和 C++。 O'Reilly & Associates。
- Harbison、Samuel P. 和 Guy L. Steele（撰稿人）。 C：参考手册。普伦蒂斯霍尔。
- Labrosse, Jean J. 嵌入式系统构建模块：完整且即用型 C.R&D 书籍中的模块。
- 伙计，伯恩哈德。 C 代表微控制器。Franzis-Verlag。 [用英文写的。]
- 迈耶斯、斯科特。有效的 C++：改进程序的 50 种特定方法设计。艾迪生-韦斯利。
- 迈耶斯、斯科特。更有效的 C++。艾迪生-韦斯利。
- 迈耶斯、斯科特。有效的 STL。艾迪生-韦斯利。
- 萨特，赫伯。出色的 C++：47 个工程难题、编程问题、和解决方案。艾迪生-韦斯利。

网站 isocpp.org 也有一份关于 C++ 编程的推荐书籍列表。

网站

推荐网站：

- 芯片制造商的网站。
- IAR Systems 网站 www.iar.com，其中包含应用说明和其他产品信息。
- C 标准化工作组网站，www.open-std.org/jtc1/sc22/wg14。
- C++ 标准委员会的网站，www.open-std.org/jtc1/sc22/wg21。
- C++ 编程语言网站 isocpp.org。该网站还提供了有关 C++ 编程的推荐书籍列表。
- C 和 C++ 参考网站 en.cppreference.com。

页码:35

文档约定

在 IAR Systems 文档中，当我们提到编程语言 C 时，text 也适用于 C++，除非另有说明。当引用产品安装中的目录时，例如 8051\doc，假定位置的完整路径，例如 C:\Program Files\IAR Systems\Embedded Workbench N.n\8051\doc，其中版本的起始数字 number 反映了 IAR Embedded Workbench 版本号的初始数字共享组件。

印刷约定

IAR Systems 文档集使用以下印刷约定：

风格	用于
computer	• 源代码示例和文件路径。 • 命令行上的文本。 • 二进制、十六进制和八进制数。
parameter	用作参数的实际值的占位符，例如 filename.h 其中 filename 表示文件的名称。
[option]	指令的可选部分，其中［和 ］不是实际指令的一部分指令，但任何［、］、{ 或 } 都是指令语法的一部分。
{option}	指令的强制部分，其中 { 和 } 不是实际指令的一部分，但任何［、］、{ 或 } 都是指令语法的一部分
[option]	命令的可选部分。
[a|b|c]	带有替代项的命令的可选部分。
{a|b|c}	带有替代项的命令的强制性部分。
bold	出现在屏幕上的菜单、菜单命令、按钮和对话框的名称。
italic	• 本指南或其他指南中的交叉引用。 • 强调
…	省略号表示前一项可以任意重复次数。
	识别特定于 IAR Embedded Workbench® IDE 的指令界面
	标识特定于命令行界面的指令。
	确定有用的提示和编程提示。
	标识警告。

表 1：本指南中使用的排版约定

命名约定

以下命名约定用于 IAR 的产品和工具 Systems®，在文档中提及时：

品牌名称	总称
IAR Embedded Workbench® for 8051	用于 8051 的 IAR Embedded Workbench®
IAR Embedded Workbench® IDE for 8051	用于 8051 的 IAR Embedded Workbench® IDE
IAR C-SPY® Debugger for 8051	用于 8051 的 IAR C-SPY® 调试器
IAR C-SPY® Simulator	IAR C-SPY® 模拟器
IAR C/C++ Compiler™ for 8051	用于 8051 的 IAR C/C++ 编译器™
IAR Assembler™ for 8051	用于 8051 的 IAR 汇编器™
IAR XLINK Linker™	IAR XLINK 链接器™
IAR XAR Library Builder™	IAR XAR 库生成器™
IAR XLIB Librarian™	IAR XLIB 库管理员™
IAR DLIB Runtime Environment™	IAR DLIB 运行时环境™
IAR CLIB Runtime Environment™	IAR CLIB 运行时环境™

表 2：本指南中使用的命名约定

注意：在本指南中，8051 微控制器是指所有兼容 8051 微控制器架构。

（空白页）
页码:38

第 1 部分：使用编译器

IAR C/C++编译器 8051 用户指南的这一部分包括以下章节：

●IAR 构建工具简介
●开发嵌入式应用程序
●数据存储
●了解内存体系结构
●功能
●分页功能
●链接概述
●链接应用程序
●DLIB 运行时环境
●CLIB 运行时环境
●汇编语言接口
●使用 C
●使用 C++
●与应用程序相关的注意事项
●嵌入式应用程序的高效编码。

（空白页）
页码:40

IAR 构建工具简介

●IAR 构建工具概述
●IAR 语言概述
●设备支持
●对嵌入式系统的特殊支持

IAR 构建工具概述

在 IAR 产品安装中，您可以找到一组工具、代码示例和用户文档，这些都适合为基于 8051 的嵌入式应用程序开发软件。这些工具允许您使用 C、C++或汇编语言开发应用程序。

IAR Embedded Workbench®是一个非常强大的集成开发环境（IDE），允许您开发和管理完整的嵌入式应用程序项目。

它提供了一个易于学习和高效的开发环境，具有最大的代码继承能力、全面和特定的目标支持。IAR Embedded Workbench 促进了一种有用的工作方法，从而大大缩短了开发时间。

有关 IDE 的信息，请参阅 8051 的 IDE 项目管理和构建指南。

如果您想在已经建立的项目环境中将编译器、汇编器和链接器用作外部工具，那么也可以从命令行环境中运行它们。

IAR C/C++编译器
用于 8051 的 IAR C/C++编译器是一种最先进的编译器，它提供了 C 和 C++语言的标准功能，以及旨在利用 8051 特定功能的扩展。

IAR 汇编程序

用于 8051 的 IAR 汇编程序是一个功能强大的重新定位宏汇编程序，具有一组多功能的指令和表达式运算符。汇编程序具有内置的 C 语言预处理器，并支持条件汇编。

IAR XLINK 链接器

IAR XLINK 链接器是一种功能强大、灵活的软件工具，用于开发嵌入式控制器应用程序。它同样适用于连接小型、单文件、绝对汇编程序，用于链接大型、可重定位输入、多模块、C/C++或混合 C/C++和汇编程序。

为了处理库，包含了库工具 XAR 和 XLIB。

外部工具

有关如何在 IDE 中扩展工具链的信息，请参阅针对 8051 的 IDE 项目管理和构建指南。

IAR 语言概述

针对 8051 的 IAR C/C++编译器支持：

1 C，嵌入式系统行业中使用最广泛的高级编程语言。您可以构建遵循以下标准的独立应用程序：

- 标准 C-也称为 C99。本指南中本标准以下简称 标准 C。
- C89 也称为 C94、C90、C89 和 ANSI C。启用 MISRA C 时需要此标准。

2 C++，一种现代面向对象编程语言，具有功能齐全的库，非常适合模块化编程。可以使用以下任何标准：

- 嵌入式 C++（EC++）-C++编程标准的子集，用于嵌入式系统编程。它是由一个行业联盟嵌入式 C++技术委员会定义的。请参阅使用 C++一章。
- IAR Extended Embedded C++（EEC++）-EC++，具有其他功能，如完全模板支持、命名空间支持、新的 cast 操作符以及标准模板库（STL）。

每种受支持的语言都可以在严格或宽松模式下使用，也可以在启用 IAR 扩展的情况下轻松使用。严格模式符合标准，而放松模式允许一些与标准的常见偏差。

有关 C 的更多信息，请参阅使用 C 一章。
有关嵌入式 C++ 和扩展嵌入式 C++ 的更多信息，请参阅使用 C++ 一章。
有关编译器如何处理语言的实现定义区域的信息，请参阅标准 C 的实现定义的行为一章。
也可以用汇编语言实现部分应用程序或整个应用程序。请参阅 8051 的 IAR 汇编器用户指南。

设备支持

为了顺利开始您的产品开发，IAR 产品安装附带了广泛的设备特定支持。

支持的 8051 设备

用于 8051 的 IAR C/C++ 编译器支持所有基于标准 8051 微控制器架构的设备。
还支持以下扩展：
•多数据指针 (DPTR)。代码生成器中集成了多达 8 个数据指针的支持
• 扩展代码存储器，最大 16 MB。（例如在 Maxim (Dallas 半导体) DS80C390/DS80C400 器件中使用。）
• 扩展数据存储器，最大 16 MB。（例如用于 Analog Devices ADuC812 器件和 Maxim DS80C390/DS80C400 器件。）
•Maxim DS80C390/DS80C400 器件和类似器件，包括对扩展指令集、多数据指针和扩展堆栈（位于 xdata 存储器中的调用堆栈）的支持
•Mentor Graphics M8051W/M8051EW 内核和基于此的设备，包括对分组 LCALL 指令、分组 MOVC A、@A+DPTR 的支持，以及将中断服务例程放置在分组内存中。
要了解有关如何根据您使用的设备设置项目的更多信息，请参阅基本项目配置，第 56 页。

预配置的支持文件

IAR 产品安装包含用于支持不同设备的预配置文件。如果您需要其他文件来支持设备，可以使用提供的文件之一作为模板来创建它们。

I/O 的头文件

标准外围单元在设备特定的 I/O 头文件中定义文件扩展名 h.您可以在 8051\inc 目录中找到这些文件。确保在应用程序源文件中包含适当的包含文件。如果你需要附加的 I/O 头文件，它们可以使用提供的一个作为模板。

链接器配置文件

8051\config 目录包含所有现成的链接器配置文件支持的设备。这些文件的文件扩展名为 xcl 并包含链接器所需的信息。有关链接器配置的更多信息文件，请参阅放置代码和数据--链接器配置文件，第 122 页以及 IAR 链接器和库工具参考指南。

设备描述文件

调试器处理几个特定于设备的要求，例如定义可用存储区、外围寄存器和这些的组，通过使用设备描述文件。这些文件位于 8051\config 目录中，它们有文件扩展名 ddf。可以定义这些外设寄存器和组在单独的文件（文件扩展名 sfr）中，在这种情况下，这些文件包含在 ddf 文件中。有关这些文件的更多信息，请参阅 8051 的 C-SPY® 调试指南。

入门示例

IAR Embedded Workbench 提供了示例应用程序。你可以使用这些开始使用 IAR Systems 开发工具的示例。这样你就可以使用这些示例作为您的应用程序项目的起点。您可以在 8051\examples 目录中找到示例。示例已准备就绪按原样使用。它们提供了现成的工作区文件，以及源代码文件和所有其他相关文件。有关如何运行示例的信息项目，请参阅 8051 的 IDE 项目管理和构建指南。

对嵌入式系统的特殊支持

本节简要介绍编译器提供的扩展以支持 8051 微控制器的特定功能。

扩展关键词

编译器提供了一组关键字，可用于配置代码的方式生成。例如，有用于控制内存类型的关键字单个变量以及用于声明特殊函数类型。默认情况下，在 IDE 中启用语言扩展。命令行选项 -e 使扩展关键字可用，并保留它们这样它们就不能用作变量名。参见，-e，第 302 页了解更多信息信息。有关扩展关键字的更多信息，请参阅扩展关键字一章。另请参见第 67 页的数据存储和第 95 页的功能。

Pragma 预编译指令

Pragma 预编译指令控制编译器的行为，例如它如何分配内存，是否允许扩展关键字，是否发出警告信息。编译指示总是在编译器中启用。它们与标准 C，当您想确保源代码是便携的。有关 Pragma 预编译指令的更多信息，请参阅 Pragma 预编译指令一章。

预定义符号

使用预定义的预处理器符号，您可以检查编译时环境，例如代码和数据模式。有关预定义符号的更多信息，请参阅预处理器一章。

访问低级功能

对于应用程序的硬件相关部分，访问低级功能是必不可少的。编译器支持几种方法：内在函数、混合 C 和汇编器模块和内联汇编器。有关不同方法的信息，参见混合 C 和汇编程序，第 191 页。

页码:45

（空白页）
页码:46

开发嵌入式应用

- 使用 IAR 构建工具开发嵌入式软件
- 构建过程--概述
- 应用程序执行--概述
- 构建应用程序--概述
- 基本项目配置

使用 IAR 构建工具开发嵌入式软件

通常，为专用微控制器编写的嵌入式软件被设计为无限循环等待一些外部事件发生。该软件位于 ROM 并在复位时执行。您必须考虑几个硬件和软件因素当您编写此类软件时。为了您的帮助，您有编译器选项，扩展关键字、Pragma 预编译指令等。

内存映射

嵌入式系统通常包含各种类型的存储器，例如片上 RAM、外部 DRAM 或 SRAM、ROM、EEPROM 或闪存。作为嵌入式软件开发人员，您必须了解不同的功能记忆的种类。例如，片上 RAM 通常比其他类型的在时间关键型应用程序中经常访问的内存和变量受益于被安置在这里。相反，可能会访问一些配置数据很少但必须在断电后保持其值，因此应保存在 EEPROM 或闪存。为了有效地使用内存，编译器提供了几种控制机制在内存中放置函数和数据对象。有关详细信息，请参阅控制内存中的数据和函数放置，第 259 页。链接器放置根据您在链接器中指定的指令，内存中的代码和数据部分配置文件，请参见放置代码和数据--链接器配置文件，第 122 页。

与外围设备的通信

如果外部设备连接到微控制器，您可能需要初始化并控制信令接口，例如通过使用芯片选择引脚，并检测和处理外部中断信号。通常，这必须初始化并控制在运行。执行此操作的常规方法是使用特殊功能寄存器 (SFR)。这些都是通常在专用地址可用，包含控制芯片的位配置。标准外围单元在设备特定的 I/O 头文件中定义文件扩展名 h.请参阅设备支持，第 43 页。有关示例，请参阅访问特殊功能寄存器，第 273 页。事件处理在嵌入式系统中，使用中断是一种处理外部事件的方法立即地;例如，检测到按钮被按下。一般来说，开启时代码中发生中断，单片机立即停止执行代码它运行，并开始执行中断程序。编译器提供各种用于管理硬件和软件的原语中断，这意味着您可以用 C 编写中断例程，请参阅 Primitives 中断、并发和与操作系统相关的编程，第 97 页。

系统启动

在所有嵌入式系统中，都会执行系统启动代码来初始化系统--两者都是硬件和软件系统--在应用程序的主要功能是之前调用。

作为嵌入式软件开发人员，您必须确保启动代码位于专用内存地址，或者可以使用向量中的指针访问桌子。这意味着启动代码和初始向量表必须放在非易失性存储器，例如 ROM、EPROM 或闪存。C/C++ 应用程序还需要初始化所有全局变量。这个初始化是由链接器和系统启动代码共同处理。了解更多信息，见应用程序执行--概述，第 52 页。

实时操作系统

在许多情况下，嵌入式应用程序是系统中运行的唯一软件。但是，使用 RTOS 有一些优势。例如，高优先级任务的时序不受其他部分的影响在较低优先级任务中执行的程序。

这通常会使程序更具确定性，并且可以通过有效地使用 CPU 来降低功耗，并且在空闲时将 CPU 置于低功耗状态。

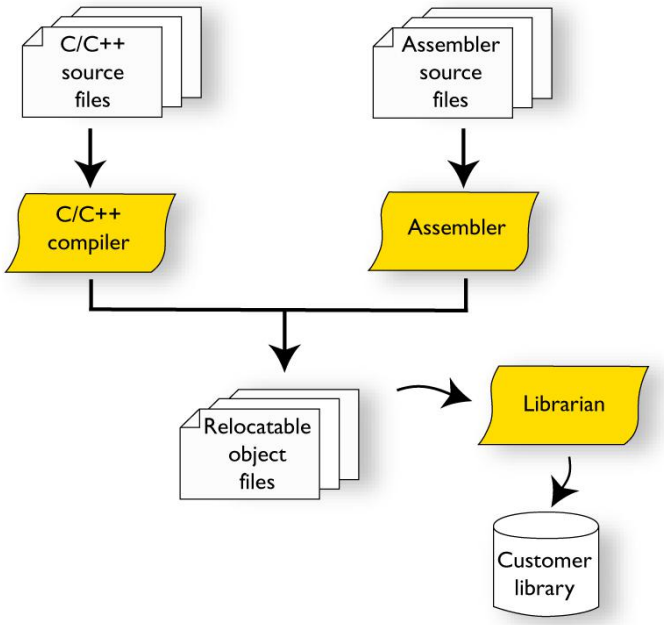
使用 RTOS 可以使您的程序更易于阅读和维护，而且在许多情况下也更小。应用程序代码可以在真正的任务中干净地分离彼此独立。这使得团队合作更容易，因为开发工作可以很容易地分成单独的任务，由一个开发人员或一组开发商。最后，使用 RTOS 减少了对硬件的依赖并创建了一个干净的界面到应用程序，更容易将程序移植到不同的目标硬件。

构建过程--概述

本节概述了构建过程；各种构建工具如何--编译器、汇编器和链接器--结合在一起，从源代码到可执行文件图片。要在实践中熟悉该过程，您应该运行一个或多个教程可从 IAR 信息中心获得。

编译过程

IDE 中有两个工具可以将应用程序源文件转换为中间文件目标文件。 IAR C/C++ 编译器和 IAR 汇编器。两者都产生 IAR UBROF 格式的可重定位目标文件。注意：编译器也可用于将 C 源代码翻译成汇编源代码代码。如果需要，您可以修改汇编源代码，然后可以组装成目标代码。有关 IAR 汇编程序的更多信息，请参阅 IAR8051 汇编器用户指南。



编译后，您可以选择将任意数量的模块打包到一个存档中，或者换句话说，一个库中。您应该使用库的重要原因是库中的每个模块在应用程序中都有条件链接，或者换句话说，只有在作为目标文件提供的模块直接或间接使用该模块时才包含在应用程序中。或者，您可以创建一个库；然后使用 IAR XAR Library Builder 或 IAR XLIB Librarian。

链接过程

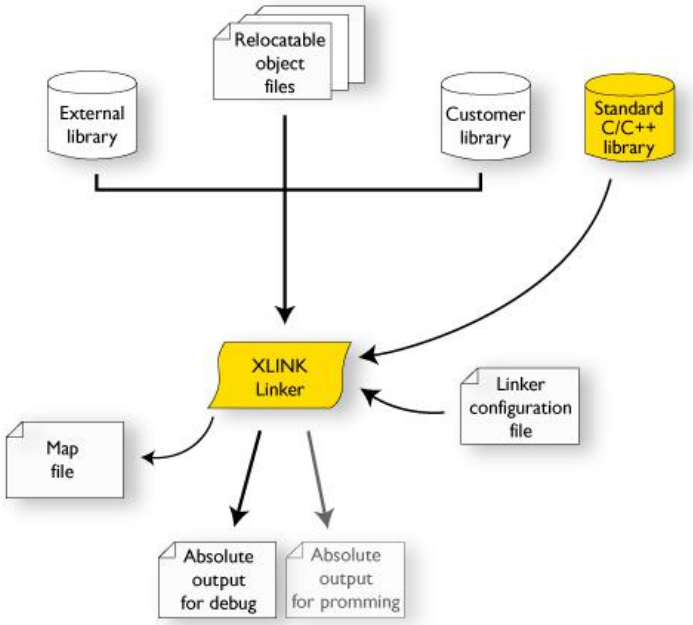
IAR 编译器和汇编器生成的目标文件和库中的可重定位模块不能按原样执行。要成为一个可执行的应用程序，它们必须被链接起来。

注意：由其他供应商的工具集生成的模块也可以包含在构建中。请注意，这也可能需要来自同一供应商的编译器实用程序库。

IAR XLINK 链接器 (xlink.exe) 用于构建最终应用程序。通常，链接器需要以下信息作为输入：

•几个目标文件和可能的某些库
•包含运行环境和标准语言函数的标准库
•程序开始标签（默认设置）
• 描述目标系统内存中代码和数据放置的链接器配置文件

2 关于输出格式的信息。
IAR XLINK 链接器根据您的规范生成输出。 选择适合您目的的输出格式。 您可能希望将输出加载到调试器--这意味着您需要带有调试信息的输出。 或者，您可能希望将输出加载到闪存加载程序或 PROM 编程器--在这种情况下，您需要没有调试信息的输出，例如 Intel hex 或 Motorola S- 记录。 选项 -F 可用于指定输出格式。
此图显示了链接过程：



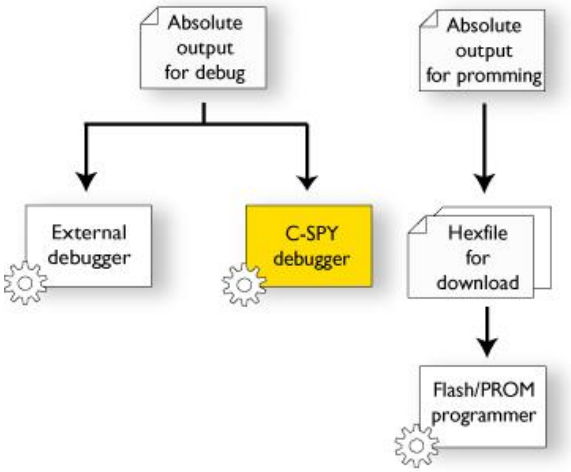
注意：标准 C/C++ 库包含编译器的支持例程，以及 C/C++ 标准库函数的实现。在链接期间，链接器可能会在 stdout 和 stderr 上生成错误消息和日志消息。 日志消息有助于理解为什么应用程序以原来的方式链接，例如，为什么包含模块或删除部分。有关链接器执行的过程的更多信息，请参阅 IAR 链接器和库工具参考指南。

链接后

IAR XLINK 链接器以您指定的输出格式生成一个绝对目标文件。 链接后，生成的绝对可执行映像可用于：

●加载到 IAR C-SPY 调试器或任何其他兼容的读取 UBROF 的外部调试器。

●使用闪存/PROM 编程器对闪存/PROM 进行编程。 此图显示了绝对输出文件的可能用途：



应用程序执行--概述

本节概述了嵌入式应用程序的执行，分为以下几部分：

三个阶段，即：

- 初始化阶段
- 执行阶段
- 终止阶段

初始化阶段

初始化在应用程序启动（CPU 复位）但在进入主函数。 为简单起见，初始化阶段可以分为：

- 硬件初始化，一般至少初始化堆栈指针。

硬件初始化通常在系统启动代码中执行 cstartup.s51and 如果需要，通过您提供的额外低级例程。 它可能包括重置/启动其余硬件、设置 CPU 等，在准备软件 C/C++ 系统初始化。

● 软件 C/C++系统初始化

通常，这包括确保每个全局（静态链接的）C/C++ 符号在调用 main 函数之前接收其正确的初始化值。

● 应用程序初始化

这完全取决于您的应用程序。它可以包括设置 RTOS 内核并为 RTOS 驱动的应用程序启动初始任务。

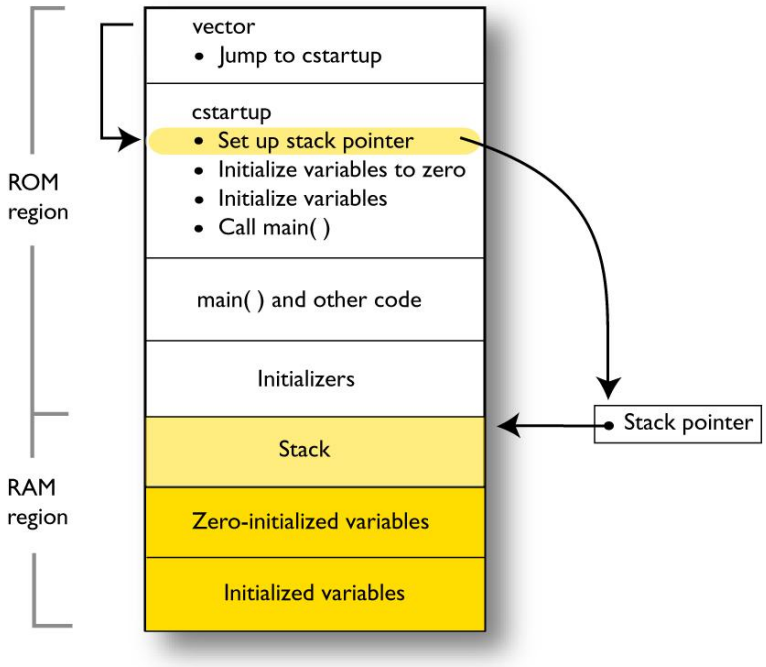
对于准系统应用程序，它可以包括设置各种中断、初始化通信、初始化设备等。

对于基于 ROM/flash 的系统，常量和函数已经放置在 ROM 中。

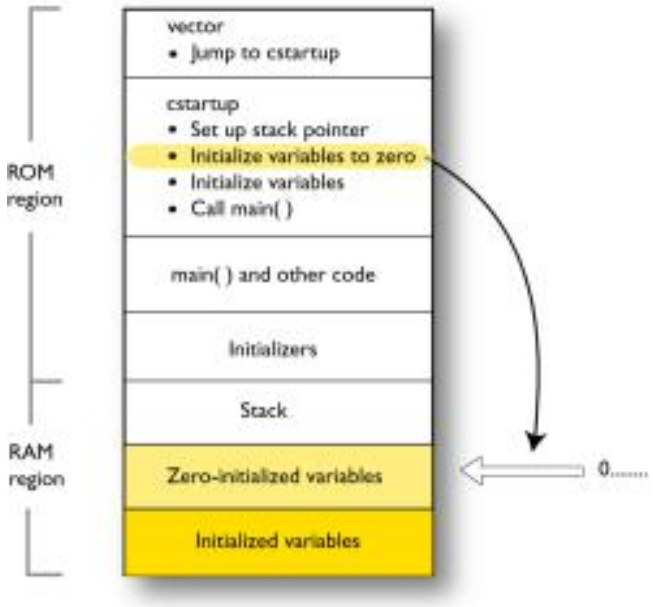
必须在调用 main 函数之前初始化放置在 RAM 中的符号。这链接器已经将可用 RAM 划分为变量、堆栈、堆等。

以下插图序列简要概述了不同的初始化阶段。

1 应用程序启动时，系统启动代码首先执行硬件初始化，比如初始化栈指针指向预定义的栈区域：

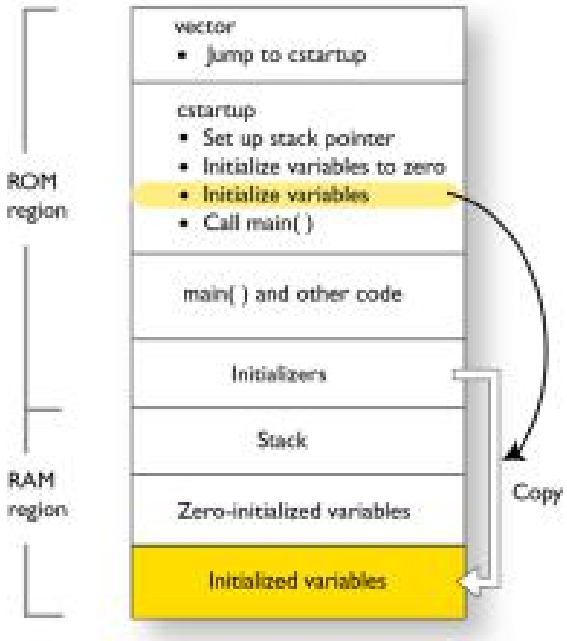


2 然后，清零应该初始化的内存，也就是说，填满零：



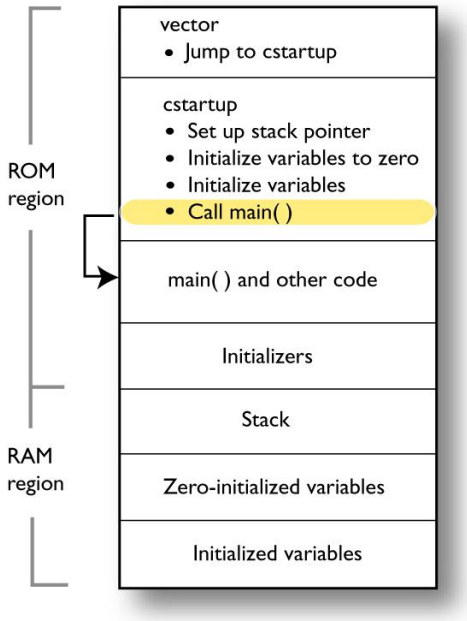
通常，此数据称为零初始化数据； 声明为的变量，对于
例如，int i = 0；

3 对于已初始化的数据，声明的数据，例如 int i = 6；初始化器是从 ROM 复制到 RAM。



页码:54

4 最后调用 main 函数：



有关每个阶段的详细信息，请参阅系统启动和终止，第 164 页。
有关数据初始化的详细信息，请参阅系统启动时的初始化，第 123 页。

执行阶段

嵌入式应用程序的软件通常实现为一个循环，该循环要么由中断驱动，要么使用轮询来控制外部交互或内部事件。对于中断驱动的系统，中断通常在 main 函数的开头进行初始化。在具有实时行为且响应能力至关重要的系统中，可能需要多任务系统。这意味着你的应用软件应该有一个实时操作系统来补充，在这种情况下，RTOS 和不同的任务也必须在 main 函数开始时进行初始化。

终止阶段

通常，嵌入式应用程序的执行永远不会结束。如果是这样，您必须定义正确的结束行为。要以受控方式终止应用程序，可以调用标准 C 库函数 exit、_Exit 或 abort 之一，或者从 main 返回。如果从 main 返回，则执行 exit 函数，这意味着调用静态和全局变量的 C++ 析构函数（仅限 C++）并关闭所有打开的文件。

当然，如果程序逻辑不正确，应用程序可能会在不受控制和异常的方式--系统崩溃。有关这方面的更多信息，请参阅系统终止，第 166 页。

页码:55

构建应用程序--概述

在命令行界面中，此行使用默认设置将源文件 myfile.c 编译为目标文件 myfile.r51：

```
icc8051 myfile.c
```

您还必须指定一些关键选项，请参见基本项目配置，第 56 页。

在命令行上，此行可用于启动链接器：

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

在本例中，myfile.r51 和 myfile2.r51 是目标文件，并且 my_configfile.xcli 是链接器配置文件。选项 -o 指定名称的输出文件。选项 -ris 用于指定输出格式 UBROF，它可用于在 C-SPY® 中进行调试。

注意：默认情况下，应用程序启动的标签是__program_start。你可以使用 -s 命令行选项来改变它。

在构建项目时，IAR Embedded Workbench IDE 可以生成大量的在构建消息窗口中构建信息。此信息可用于例如，作为在命令行上生成批处理文件的基础。你可以复制信息并将其粘贴到文本文件中。要激活广泛的构建信息，在 Buildmessages 窗口中单击鼠标右键，然后在上下文菜单中选择 Allon。

基本项目配置

本节概述了所需的项目设置的基本设置使编译器和链接器为您正在使用的 8051 设备生成最佳代码。

您可以从命令行界面或 IDE 中指定选项。

您需要进行以下设置：

- 核心
- 数据模式
- 代码模型
- 调用约定
- DPTR 设置
- 优化设置
- 运行时环境，请参见设置运行时环境，第 149 页
- 自定义 XLINK 配置，请参阅链接应用程序一章。

页码:56

除了这些设置之外，还有许多其他选项和设置可以微调结果更深入。
有关如何设置选项的信息以及所有可用的列表选项，请参阅编译器选项和 IDE 项目管理一章和分别为 8051 的构建指南。

处理器配置

为了使编译器生成最佳代码，您应该为 8051 配置它您正在使用的微控制器。

内核

该编译器支持经典的 Intel 8051 微控制器内核，Maxim (Dallas 半导体）DS80C390 内核，Mentor Graphics M8051W/M8051EW 内核，如以及类似的设备。使用 --core={plain|extended1|extended2} 选项选择核心将生成哪个代码。此选项反映了您的寻址能力目标微控制器。有关内核的信息，请参阅基本项目设置硬件内存配置，第 61 页。

在 IDE 中，选择 Project>Options>General Options>Target 并选择正确的从核心下拉列表中选择核心。然后将自动选择默认选项，以及链接器和调试器的设备特定配置文件。

数据模式

8051 微控制器的特性之一是在内存的存储方式上进行权衡访问，从廉价访问到小内存区域，到更昂贵的访问可以访问外部数据的访问方法。更多信息参见章节了解内存架构。使用编译器选项 --data_model 指定（除其他外）默认值静态和全局变量的内存放置，这也意味着默认内存访问方法。数据存储一章更详细地介绍了数据模式。本章还涵盖如何覆盖单个变量的默认访问方法。

页码:57

代码模型

编译器支持您可以在文件或函数级别设置的代码模型以指定函数的默认放置,这也意味着默认的函数调用方法。有关代码模型的详细信息，请参阅函数一章。

调用惯例

使用编译器选项 --calling_convention 来控制编译器是否通过默认应该对本地数据使用堆栈模型或覆盖模型，其中内存堆栈或覆盖框架应放置。换言之，无论是本地变量和参数应该放在堆栈上或通过覆盖内存区域，以及堆栈/内存区域应放置在内存中的哪个位置。了解更多信息，请参见选择调用约定，第 198 页。

DPTR 设置

有些设备提供多达 8 个数据指针，使用它们可以提高一些库例程。使用编译器选项 --dptr 指定要使用的数据指针的数量、大小数据指针的位置，以及数据指针的位置。记忆数据指针的地址在链接器配置文件或 IDE 中指定。

速度和尺寸优化

除其他外，编译器的优化器执行死代码消除，常量传播、内联、公共子表达式消除和精度减少。它还执行循环优化，例如展开和归纳变量消除。您可以在多个优化级别之间进行选择，对于最高级别，您可以在不同的优化目标（大小、速度或平衡）之间进行选择。必须优化将使应用程序更小更快。然而，当这是并非如此，编译器使用选定的优化目标来决定如何执行优化。可以为整个应用程序指定优化级别和目标文件，并用于各个功能。此外，一些个别的优化，如函数内联，可以禁用。有关编译器优化的信息以及有关高效的更多信息编码技术，请参阅嵌入式应用程序的高效编码一章。

了解内存架构

以下页面包含这些主题：

- 8051 单片机内存配置
- 内存配置的运行时模型概念
- 硬件内存配置的基本项目设置
- 使用 DPTR 寄存器

8051 微控制器内存配置

8051 微控制器具有三个独立的存储空间：代码存储器、内部数据存储器和外部数据存储器。不同的内存空间以不同的方式访问。内部数据存储器总是可以被有效地访问，而外部数据存储器和代码存储器不能总是以同样有效的方式被访问。

代码存储空间

在经典的 8051 中，代码存储空间是 ROM 存储器的 64 KB 地址区域，用于存储程序代码，包括所有函数和库例程，也包括常量。根据您使用的设备和硬件设计，代码存储器可以是内部（片上）、外部或两者兼有。

对于经典的 8051/8052 器件，代码存储器可以扩展至多达 256 组额外的 ROM 存储器。编译器使用 2 字节指针来访问不同的存储区。 Silabs C8051F12x 和 Texas Instruments CC2430 就是这种器件类型的示例。此外，一些器件具有扩展代码存储器，这意味着它们可以具有高达 16 MB 的线性代码存储器（例如用于 Maxim DS80C390/DS80C400 器件）。编译器使用 3 字节指针访问该区域。基于 Mentor Graphics M8051W/M8051EW 内核的设备将其代码存储器划分为 16 个存储库，每个存储库为 64 KB。编译器使用 3 字节指针来访问不同的存储区。

内部数据存储空间

根据器件的不同，内部数据存储器由最多 256 字节的片上读写存储器组成，用于存储数据，通常是常用变量。在这个区域，内存访问要么使用直接寻址方式，要么使用 MOV 指令的间接寻址方式。但是，在上部区域 (0x80 - 0xFF)，直接寻址访问专用 SFR 区域，而间接寻址访问 IDATA 区域。SFR 区域用于内存映射寄存器，例如 DPTR、A、串行端口目标寄存器 SBUF 和用户端口 P0。标准外围单元在特定于设备的 I/O 头文件中定义，文件扩展名为 h。有关这些文件的更多信息，请参见访问特殊功能寄存器，第 273 页。0x20 和 0x2Fi 之间的区域是可位寻址的，以及 0x80、0x88、0x90、0x98...0xF0 和 0xF8。请注意，编译器会保留一个字节的位寻址存储器供内部使用，请参阅第 92 页的虚拟寄存器。

外部数据存储空间

外部数据最多可包含 64 KB 的读写存储器，只能通过 MOVX 指令间接寻址。

因此，外部存储器比内部存储器慢。

许多现代设备在外部数据存储空间中提供片上 XDATA（外部数据）。例如，德州仪器 CC2430 器件具有 8 KB 的片上 XDATA。

一些器件将外部数据存储空间扩展到 16 MB。在这种情况下，编译器使用 3 字节指针来访问该区域。

内存配置的运行时模型概念

要有效地使用内存，您必须熟悉编译器和链接器中与内存相关的基本概念。

编译器概念

编译器将内存区域的每个部分与内存类型相关联。通过将不同的内存（或部分内存）映射到内存类型，编译器可以生成可以有效访问数据或函数的代码。

对于每种内存类型，编译器都提供了一个关键字（内存属性），您可以直接在源代码中使用它。这些属性使您可以显式指定单个对象的内存类型，这意味着该对象将驻留在该内存中。

页码:60

您可以通过选择数据模式来指定编译器要使用的默认内存类型。
可以通过使用内存属性来覆盖单个变量和指针。相反，您可以为代码的默认放置选择代码模型。有关可用数据模式、内存类型和相应内存属性的详细信息，请分别参见第 80 页的数据模式和第 68 页的内存类型。

链接器概念

编译器生成许多段。段是包含应该映射到内存中物理位置的一段数据或代码的逻辑实体。
编译器有许多用于不同目的的预定义段。每个段都有一个描述段内容的名称，以及一个表示内容类型的段内存类型。除了预定义的段之外，您还可以定义自己的段。
在编译时，编译器将每个段分配给它的内容。
例如，段 DATA_Z 保存零初始化的静态和全局变量。
IAR XLINK 链接器负责根据链接器配置文件中指定的规则将段放置在物理内存范围内。
要了解有关段的更多信息，请参阅段和内存，第 120 页和链接注意事项，第 127 页。
硬件内存配置的基本项目设置 根据您使用的设备及其内存配置，以下项目需要基本项目设置：

- 核心
- 代码模型
- 数据模式
- 调用约定
- DPTR 设置。

并非所有这些组合都是可能的。以下段落为您提供了有关哪些设置可用于特定类型设备的信息。
如果您在 IDE 中进行这些基本项目设置，则只能选择有效的组合。

经典 8051/8052 设备

对于经典的 8051/8052 设备，硬件内存配置主要分为三类：

- 无外部 RAM（单芯片，带或不带外部 ROM）
- 存在外部 RAM（从 0 到 64 KB）
- 分页模式（超过 64 KB 的 ROM）。

可以使用以下编译器设置的组合：

--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

表 3：核心平原 的编译器选项的可能组合

要求使用扩展堆栈编译器选项(--extended_stack).

要求不使用扩展堆栈编译器选项(--extended_stack).

如果您使用 分页 代码模式，则可以使用 __near_func 内存属性将各个函数显式放置在根库中。

您可以使用 __code 内存属性将常量和字符串放置在代码内存空间中。

以下内存属性可用于将单个数据对象放置在与默认值不同的数据内存中：__sfr, __bit, __bdata, __data, __idata, __pdata, __xdata, __xdata_rom, __ixdata, __generic.

MAXIM (DALLAS SEMICONDUCTOR) 390 及类似产品设备

这种类型的器件可以将存储器扩展到 16 MB 的外部连续数据存储器和代码存储器。

可以使用以下编译器设置的组合：

--core	--code_model	--data_model	--calling_convention	--dptr
extended1	far	far|far_generic|large	pr|er†|xr††	24
extended1	far	tiny|small	ir	24

表 4：核心扩展 1 的编译器选项的可能组合

† 需要扩展堆栈，这意味着必须使用编译器选项 --extended_stack。
†† 要求不使用编译器选项 --extended_stack。
您可以使用 __far_code 和 __huge_code 内存属性将常量和字符串放置在代码内存空间中。
以下内存属性可用于将单个数据对象放置在非默认数据内存中：__sfr、__bit、__bdata、__data、__idata、__pdata。
对于 Tiny 和 Large 数据模式还有：__xdata、__xdata_rom、__ixdata、__generic。
对于 Far Generic 数据模式，还有：__far、__far_rom、__far22、__far22_rom、__generic、__huge、__huge_rom。
对于 Far 数据模式还有：__far、__far_rom、__huge、__huge_rom。
注：虽然编译器支持 Maxim (Dallas Semiconductor) DS80C390 器件的代码和数据存储器扩展模型，但不支持器件的 40 位累加器。
基于 Mentor Graphics M8051W/M8051EW 的设备核
Mentor Graphics M8051W/M8051EW 内核和基于它的设备提供了扩展寻址机制，这意味着您可以将代码内存扩展至多达 16 个存储区，每个存储区为 64 KB。
可以使用以下编译器设置组合：

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

表 5：核心扩展 2 的编译器选项的可能组合

以下内存属性可用于将单个数据对象放置在与默认值不同的数据内存中：__sfr、__bit、__bdata、__data、__idata、__pdata。
对于 Tiny 和 Large 数据模式还有：__xdata、__xdata_rom、__ixdata、__generic。

使用 DPTR 寄存器

一些设备有多达 8 个数据指针，可用于加速内存访问。 支持扩展内存的设备必须使用 24 位数据指针，而不支持扩展内存的经典设备使用 16 位数据指针。

使用 DPTR 寄存器，编译器最多支持 8 个数据指针。
在某些情况下，使用 DPTR 寄存器可以生成更紧凑、更高效的代码。在许多应用程序中，数据指针是一种频繁使用的资源，通常必须保存在堆栈或寄存器中，然后再恢复。如果应用程序可以使用多个数据指针，则可以大大减少开销。
如果您使用 DPTR 寄存器，您必须指定：

- 要使用的数据指针的数量
- 数据指针的大小
- 数据指针所在的位置
- 如何选择数据指针。

要在 IDE 中设置 DPTR 寄存器的选项，请选择 Project>Options>General Options>Data Pointer。
在命令行上，使用编译器选项 --dptr 指定必要的选项，
参见 --dptr，第 301 页。

内存位置
不同的 8051 器件以不同的方式表示 DPTR 寄存器。
两种方法之一用于在内存中定位数据指针寄存器。
使用您正在使用的设备支持的设备。

- 阴影可见度

所有数据指针使用相同的 SFR（DPL 和 DPH）地址；数据指针选择寄存器 (DPS) 指定哪个数据指针在该地址可见。

- 单独的可见性

不同的位置 DPL0、DPL1 等和 DPH0、DPH1 等用于不同的数据指针。
如果使用此方法，DPS 寄存器指定当前活动的数据指针。
如果数据指针在内存中有不同的位置，则必须单独指定这些内存位置。
对于大多数设备，这些地址由 IAR Embedded Workbench 自动设置。
如果您的设备缺少此信息，您可以轻松指定这些地址。
指定内存中的位置
用于数据指针的内存地址在链接器命令文件中指定。

以下行举例说明了使用位于不同地址的两个 24 位数据指针的设备的设置：

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

符号 ?DPS 指定 DPS 寄存器的位置。 ?DPX 指定第一个数据指针的扩展字节的位置。（第一个数据指针的低地址和高地址总是分别位于地址 0x82 和 0x83。） ?DPL1、?DPH1 和?DPX1 指定第二个数据指针的位置。

选择活动数据指针

如果您使用多个 DPTR，则有两种不同的方式来切换活动数据指针：递增或 XOR。

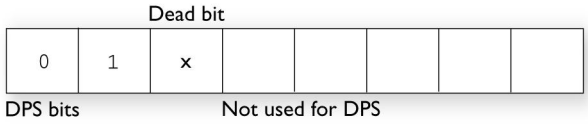
增量法

增量法（INC）是更有效的方法，但并非所有 8051 器件都支持。

使用这种方法，可以递增 DPS 寄存器中的位以选择活动数据指针。

这只有在与数据指针选择无关的 DPS 寄存器的内容可以在切换操作期间被破坏时才有可能，或者如果不能被破坏的位被一个死位保护，防止开关操作溢出到它们中。

DPS 寄存器中的选择位也必须按顺序排列，并从最低有效位开始。



可以与 INC 方法一起使用的 DPTR 的数量取决于死位的位置。

例如，如果有四个数据指针可用，并且 DPS 寄存器中的位 0 和 1 用于数据指针选择，位 2 是死位，

INC 方法只能在使用所有四个数据指针时使用。 如果只使用四个数据指针中的两个，则必须改用 XOR 选择方法。

另一方面，如果位 0 和 2 用于数据指针选择并且位 1 是死位，

使用两个数据指针时可以使用 INC 方法，但如果使用所有四个数据指针，则必须使用 XOR 方法。

异或法

XOR 方法并不总是那么有效，但它总是可以使用。 只有用于数据指针选择的位会受到 XOR 选择操作的影响。

这些位在位掩码中指定，如果使用此方法，则必须指定该位掩码。

选择位在位掩码中标记为设置位，例如，如果使用四个数据指针，选择位为位 0 和位 2，则选择位掩码应为 0x05（二进制格式为 00000101）。

因此可以始终使用 XOR 数据指针选择方法，而不管任何死位、哪些位用于选择或 DPS 寄存器中使用的其他位。

注意：INC 是 Extended1 内核的默认切换方法。

对于其他内核，XOR 是默认方法。 默认掩码取决于指定的数据指针的数量。

此外，假设使用了最低有效位，例如，如果使用六个数据指针，则默认掩码将为 0x07。

数据存储

- 简介
- 内存类型
- 数据模式
- 常量和字符串
- 自动变量和参数的存储
- 堆上的动态内存
- 虚拟寄存器

====

介绍

8051 微控制器具有三个独立的存储空间：代码存储器、内部数据存储器和外部数据存储器。不同的内存空间以不同的方式访问。内部数据存储器总是可以被有效地访问，而外部数据存储器和代码存储器不能总是以同样有效的方式被访问。

有关 8051 微控制器存储器配置的详细信息，请参阅了解存储器架构，第 59 页。

存储数据的不同方式

在典型的应用程序中，数据可以通过三种不同的方式存储在内存中：

- 自动变量

除声明为静态的变量外，函数的所有局部变量都存储在寄存器中或每个函数的局部框架中。只要函数执行，就可以使用这些变量。当函数返回给它的调用者时，内存空间不再有效。有关详细信息，请参阅自动变量和参数的存储，第 83 页。

本地帧既可以在运行时从堆栈（堆栈帧）分配，也可以在链接时静态分配（静态覆盖帧）。为其框架使用静态覆盖的函数通常不可重入，也不支持递归调用。

有关详细信息，请参阅自动变量和参数的存储，第 83 页。

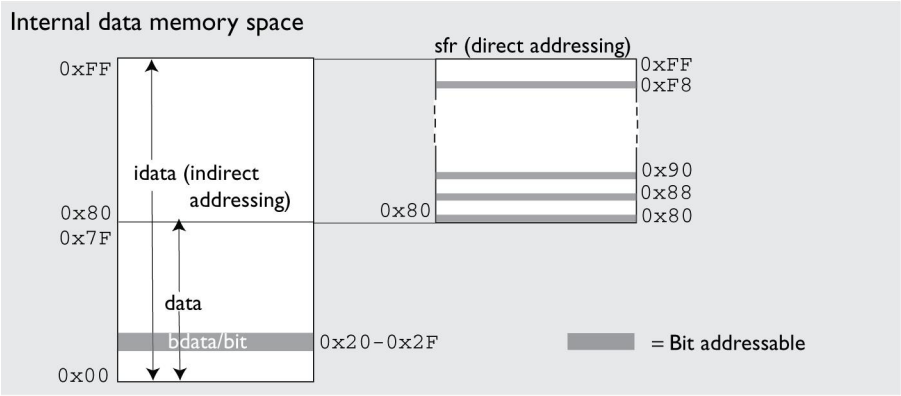
● 全局变量、模块静态变量和局部变量声明为静态的，这种情况下，内存是一次性分配的，这里的静态一词是指为这种变量分配的内存量在应用时不会改变在跑。有关详细信息，请参阅第 80 页的数据模式和第 68 页的内存类型。
● 动态分配数据。
应用程序可以在堆上分配数据，其中数据保持有效，直到应用程序显式释放回系统。这种类型的记忆是当应用程序执行之前不知道对象的数量时很有用。请注意，在内存有限的系统或预计会长时间运行的系统中使用动态分配的数据存在潜在风险。有关详细信息，请参阅堆上的动态内存，第 90 页。

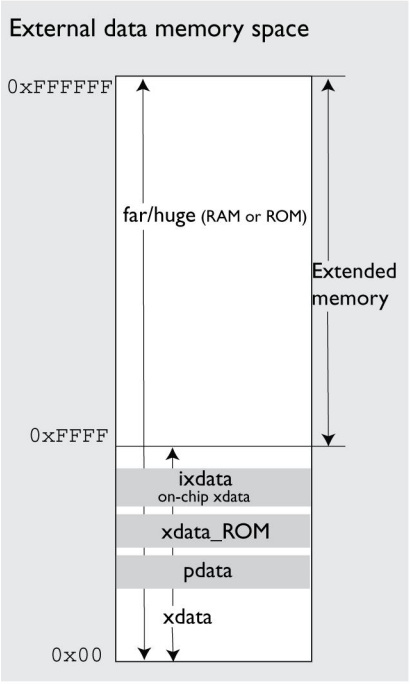
内存类型
本节介绍编译器用于访问数据的内存类型的概念。它还讨论了存在多种内存类型时的指针。对于每种内存类型，都讨论了功能和限制。

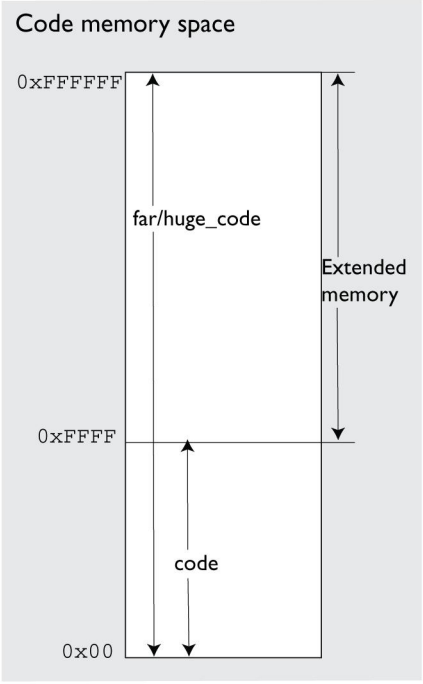
内存类型简介
编译器使用不同的内存类型来访问位于内存不同区域的数据。到达内存区域有不同的方法，它们在代码空间、执行速度和寄存器使用方面有不同的成本。这访问方法的范围从可以访问完整内存空间的通用但昂贵的方法到可以访问有限内存区域的廉价方法。每种内存类型对应一种内存访问方法。如果您将不同的内存（或部分内存）映射到内存类型，编译器可以生成可以有效访问数据的代码。
例如，使用 xdata 寻址方法访问的内存称为 xdata 内存。
要选择应用程序将使用的默认内存类型，请选择数据模式。
但是，可以为单个变量或指针指定不同的内存类型。这使得创建一个可以包含大量数据的应用程序成为可能，同时确保将经常使用的变量放在可以有效访问的内存中。

页码:68

此图说明了不同的内存类型以及它们如何与内存的不同部分相关联：







下面是各种内存类型的概述。

内部数据存储器空间的存储器类型

在内部数据存储空间中，可以使用以下存储类型：

- data
- idata
- bit/bdata
- sfr

data

数据存储器覆盖内部数据存储器的前 128 个字节 (0x0-0x7F)
空间。要在此内存中放置变量，请使用 __datamemory 属性。
这表示该变量将使用最紧凑的 MOV A,10 直接寻址可以访问变量。
此类对象的大小限制为 128 字节-8 字节（注册区域）。
此内存在 Tiny 数据模式中是默认的。

Idata

idata 存储器覆盖所有 256 字节的内部数据存储空间 (0x0-0xFF)。
上 idata 对象可以放置在这个范围内的任何位置，并且这样的对象的大小是限制为 256 个字节 - 8（寄存器区域）。要在此内存中放置一个变量，请使用__idatamemory 属性。
这样的对象将通过间接寻址来访问使用以下构造 MOV R0,H10 和 MOV A,@R0，速度较慢与直接在数据存储器中访问的对象相比。
Idata 内存默认为小数据模式。

bit/bdata

位/bdata 存储器覆盖 32 字节位可寻址存储器区域（0x20-0x2F 以及内部数据存储空间中从 0x0 和 0x08 开始的所有 SFR 地址。
__bitmemory 属性可以访问该区域中的各个位，而__bdataattribute 可以使用相同的指令访问 8 位。

sfr

sfr 存储器覆盖内部数据存储器的 128 个高字节 (0x80-0xFF)空间，并使用直接寻址进行访问。标准外围单元定义在文件扩展名为 h 的设备特定 I/O 头文件。有关更多详细信息这些文件，请参见访问特殊功能寄存器，第 273 页。

使用 __sfrmemory 属性定义您自己的 SFR 定义。

页码:70

外部数据存储器空间的存储器类型

在外部数据存储空间中，可以使用以下存储类型：

- xdata
- pdata
- ixdata
- far
- far22
- huge
- 外部数据存储空间中 ROM 存储器的存储器类型。

xdata

xdata 内存类型是指内存中的 64 KB 内存区域 (0x0 – 0xFFFF)外部数据存储空间。
使用 __xdatamemory 属性将对象放入这个内存区域，将使用 MOVX A，@DPTR 访问。
这种内存类型是大型和通用数据模式中的默认值。

pdata

pdata 内存类型是指放置在内存范围内任意位置的 256 字节区域外部数据存储空间的 0x0 – 0xFFFF。
使用 __pdatamemory 属性在这个内存区域放置一个对象。
将使用 MOVX A 访问的对象，@Ri，与使用 xdata 内存类型相比效率更高。

ixdata

一些设备提供片上 xdata（外部数据）存储器，其访问速度比正常的外部数据存储器。
如果可用，此片上数据存储器可放置在存储器范围 0x0 – 中的任何位置 0xFFFF 外部数据存储空间。 使用 __ixdatamemory 属性访问该片上存储器中的对象。如果使用，则在系统启动代码 (cstartup.s51) 中启用此内存。你应该验证它是否根据您的要求进行设置。

far

far 内存类型是指整个 16 MB 内存区域 (0x0 – 0xFFFFFF)外部数据存储空间。
使用 __farmemory 属性将对象放入这个内存区域，将使用 MOVX 访问。
这种内存类型仅使用 Far 数据模式时可用，在这种情况下默认使用它。
请注意，此类对象的大小限制为 64 KB-1。
有关详细信息放置限制，请参见数据指针，第 331 页。

far22

far22 内存类型是指外部的低 4 MB (0x0 – 0x3FFFFF)数据存储空间。使用 __far22memory 属性将对象放置在此内存区域，将使用 MOVX 访问。

此内存类型仅可用当使用 Far Generic 数据模式时，默认情况下使用它。

请注意，此类对象的大小限制为 64 KB-1。有关详细信息放置限制，请参见数据指针，第 331 页。

huge

huge 内存类型是指整个 16-Mbyte 内存区域 (0x0 – 0xFFFFFF)外部数据存储空间。

使用 __hugememory 属性将对象放入这个内存区域，将使用 MOVX 访问。

huge 内存类型的缺点是生成访问内存的代码比任何其他内存类型更大且更慢。

此外，代码使用更多的处理器寄存器，这可能会强制将局部变量存储在堆栈上而不是在寄存器中分配。

由于这些原因，你应该只使用 huge 内存来存储不适合 far 或 far22 内存的对象。

外部数据存储器中 ROM 存储器的存储器类型空间一些设备在访问的外部数据存储空间中提供 ROM 存储器以与其他外部数据存储器相同的方式。

取决于 16 MB 的位置此 ROM 内存可用的地址空间，不同的内存类型相关联与 ROM 存储器：

内存类型	属性	地址范围	解释
Xdata ROM	__xdata_rom	0x0-0xFFFF	此类对象的大小限制为 64Kbytes-1，它不能跨越 64 KB 的物理段边界。
Far ROM	__far_rom	0x0-0xFFFFFF	此类对象的大小限制为 64 KB-1。 有关放置限制的详细信息，请参见数据指针，第 331 页。
Far22 ROM	__far22_rom	0x0-0x3FFFFF	此类对象的大小限制为 64 KB-1。 有关放置限制的详细信息，请参见数据指针，第 331 页。
Huge ROM	__huge_rom	0x0-0xFFFFFF	为访问内存而生成的代码比任何其他内存类型的代码更大且速度更慢。此外，代码使用了更多的处理器寄存器，这可能会强制将局部变量存储在堆栈中，而不是分配到寄存器中。

表 6：外部数据存储空间中 ROM 存储器的存储器类型

将此内存用于常量和字符串，因为此内存只能读取而不能写入。

代码存储空间的存储类型

在代码数据存储空间中，可以使用以下存储类型：

- code
- far code
- far22 code
- huge code

Code

代码内存类型是指代码中的 64-Kbyte 内存区域 (0x0 - 0xFFFF)内存空间。
使用 __codememory 属性将常量和字符串放置在此内存区域，将使用 MOVC 访问。

far code

远代码内存类型是指整个 16 MB 内存区域 (0x0 - 0xFFFFFF)在代码存储空间中。
使用 __far_codememory 属性放置常量和此内存区域中的字符串，将使用 MOVC 访问。
请注意，此类对象的大小限制为 64 KB-1。
有关详细信息放置限制，请参见数据指针，第 331 页。
此内存类型仅在使用 Far 数据模式时可用。

far22 code

far22 code 内存类型是指代码的低 4 MB (0x0 - 0x3FFFFF)内存空间。
使用 __far22_codememory 属性放置常量和字符串在这个内存区域中，将使用 MOVC 访问。

请注意，此类对象的大小限制为 64 KB-1。有关详细信息放置限制，请参见数据指针，第 331 页。
此内存类型仅在使用 Far Generic 数据模式时可用。

页码:73

huge code

huge code 内存类型是指整个 16-Mbyte 的内存区域（0x0-0xFFFFFF）在外部数据存储空间。使用 __huge_codememory 属性在这个内存区域中放置一个对象，将使用 MOVC 访问该对象。巨大内存类型的缺点是生成访问内存的代码比任何其他内存类型更大且更慢。

此外，代码使用更多的处理器寄存器，这可能会强制将局部变量存储在堆栈上而不是在寄存器中分配。 由于这些原因，您应该只使用巨大的代码内存来存储不适合远代码或远代码内存的对象。

此内存类型仅在使用 Far 数据模式时可用。

使用数据内存属性

编译器提供了一组扩展关键字，可以作为数据存储器使用属性。这些关键字允许您覆盖单个数据的默认内存类型对象和指针，这意味着您可以将数据对象放置在其他内存区域中比默认内存。这也意味着您可以微调访问方法每个单独的数据对象，这会导致更小的代码大小。

下表总结了可用的内存类型及其对应的关键字：

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

表 7：内存类型和指针类型属性

* 在此表中，术语 IDATA 是指内部数据存储空间，XDATA 是指到外部数据存储空间，CODE 指代码存储空间。

见 8051 微控制器内存配置，第 59 页了解有关内存空间。

并非所有内存类型都始终可用；有关详细信息，请参阅基本项目硬件内存配置的设置，第 61 页。有关可以使用的指针，请参见指针类型，第 330 页。

只有在编译器中启用了语言扩展时，这些关键字才可用。

在 IDE 中，默认启用语言扩展。

使用 -e 编译器选项启用语言扩展。 有关其他信息，请参见 -e，第 302 页。
有关每个关键字的详细信息，请参阅扩展关键字的描述，第 342 页。
用于数据对象的类型属性的语法
类型属性使用与类型限定符 const 和 volatile 几乎相同的语法规则。 例如：

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

i 和 j 都放在 pdata 内存中。
与 const 和 volatile 不同，当派生类型中在类型说明符之前使用类型属性时，类型属性适用于对象或 typedef 本身，结构成员声明除外。

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

在所有情况下，如果未指定内存属性，则使用适当的默认内存类型，这取决于使用的数据模式。
使用类型定义有时可以使代码更清晰：

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

d16_int 是 pdata 内存中整数的 typedef。 变量 q1 可以指向这样的整数。
您还可以使用#pragma type_attributes 指令为声明指定类型属性。 在 Pragma 预编译指令中指定的类型属性应用于正在声明的数据对象或 typedef。

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

变量 q2 放置在 pdata 内存中。
有关使用内存属性的更多示例，请参阅更多示例，第 78 页。

类型定义

也可以使用类型定义来指定存储。 这两个声明是相等的：

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

指针和内存类型

指针用于引用数据的位置。通常，指针具有类型。

例如，int * 类型的指针指向一个整数。

在编译器中，指针也指向某种类型的内存。内存类型使用星号前的关键字指定。例如，指向存储在 pdata 内存中的整数的指针通过以下方式声明：

```
int __pdata * myPtr;
```

请注意，指针变量 myPtr 的位置不受关键字影响。然而，在下面的示例中，指针变量 myPtr2 被放置在 pdata 内存中。

与 myPtr 一样，myPtr2 指向 xdata 内存中的一个字符。

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

指针类型之间的差异

指针必须包含指定特定内存类型的内存位置所需的信息。这意味着指针大小对于不同的内存类型是不同的。

在用于 8051 的 IAR C/C++ 编译器中，如果较小的指针都指向相同类型的内存，则它们可以隐式转换为较大类型的指针。例如，可以将 __pdata 指针隐式转换为 __xdata 指针。指针不能隐式转换为指向不兼容内存区域的指针（即代码指针不能隐式转换为数据指针，反之亦然），较大的指针不能隐式转换为较小的指针，如果这意味着这种精确性正在丧失。

只要有可能，就应该在没有内存属性的情况下声明指针。例如，标准库中的函数都是在没有显式内存类型的情况下声明的。

有关指针的详细信息，请参阅指针类型，第 330 页。

结构和内存类型

对于结构，整个对象被放置在相同的内存类型中。不可能将单个结构成员放置在不同的内存类型中。

在下面的示例中，变量 gamma 是放置在 pdata 内存中的结构。

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

此声明不正确：

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

更多示例

以下是一系列带有说明的示例。 首先定义了一些整型变量，然后引入了指针变量。 最后，声明了一个接受指向 pdata 内存中整数的指针的函数。 该函数返回一个指向 xdata 内存中整数的指针。 内存属性放在数据类型之前还是之后没有区别。

int myA;　　默认内存中定义的变量，由正在使用的数据模式确定。
int __pdata myB;　　pdata 内存中的变量。
__xdata int myC;　　xdata 内存中的变量。
int * myD;　　存储在默认内存中的指针。 指针指向默认内存中的一个整数。
int __pdata * myE;　　存储在默认内存中的指针。指针指向 pdata 内存中的一个整数。
int __pdata * __xdata myF;存储在 xdata 内存中的指针，指向存储在 pdata 内存中的整数。
int __xdata * MyFunction(int __pdata *); 一个函数的声明，它接受一个参数，该参数是一个指向存储在 pdata 内存中的整数的指针。 该函数返回一个指向存储在 xdata 内存中的整数的指针。

C++ 和内存类型

C++ 类的实例被隐式地或使用内存类型属性或其他 IAR 语言扩展显式地放入内存中（就像所有其他对象一样）。
非静态成员变量，如结构字段，是较大对象的一部分，不能单独放入指定的内存中。
在非静态成员函数中，C++ 对象的非静态成员变量可以通过 this 指针显式或隐式引用。this 指针 是默认数据指针类型，除非使用类内存，请参阅将 IAR 属性与类一起使用，第 231 页。
静态成员变量可以像自由变量一样单独放入数据存储器中。
除了构造函数和析构函数之外的所有成员函数都可以像自由函数一样单独放入代码内存中。
有关 C++ 类的更多信息，请参阅将 IAR 属性与类一起使用，第 231 页。

数据模式

使用数据模式指定默认情况下编译器应将静态变量和全局变量放置在内存的哪个部分。这意味着数据模式控制：

- 默认内存类型
- 静态和全局变量以及常量字面量的默认位置
- 动态分配的数据，例如使用 malloc 分配的数据，或者在 C++ 中，操作符 new
- 默认指针类型

数据模式只指定了默认的内存类型。可以为单个变量和指针覆盖它。有关如何为单个对象指定内存类型的信息，请参阅使用数据内存属性，第 74 页。

您使用的数据模式可能会限制调用约定以及常量和字符串的位置。当您编译非类型化 ANSI C 或 C++ 代码（包括标准 C 库）时，所选数据模式中的默认指针必须能够访问所有默认数据对象。因此，默认指针必须能够访问位于堆栈上的变量以及常量和字符串对象。因此，并非数据模式、调用约定和常量位置的所有组合都是允许的，请参阅调用约定和匹配数据模式，第 85 页。

指定数据模式

有六种数据模式：Tiny、Small、Large、Generic、Far Generic 和 Far。数据模式的范围从适用于小于 128 字节数据的应用程序的 Tiny 到支持高达 16 MB 数据的 Far。这些模型由 --data_model 选项控制。每个模型都有一个默认的内存类型和一个默认的指针大小。

您的项目一次只能使用一种数据模式，并且所有用户模块和所有库模块都必须使用相同的模型。但是，您可以通过显式指定内存属性来覆盖单个数据对象和指针的默认内存类型，请参阅使用数据内存属性，第 74 页。

下表总结了不同的数据模式：

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

表 8：数据模式特征

有关在 IDE 中设置选项的信息，请参阅《8051 IDE 项目管理和构建指南》。

使用--data__model 选项指定项目的数据模式；请参见--data__model，第 295 页。

微型数据模式

默认情况下，微型数据模式使用微型内存，该内存位于内部数据内存空间的前 128 字节。可以使用直接寻址访问此内存。优点是指针存储只需要 8 位。作为参数传递的默认指针类型将使用堆栈上的一个寄存器或一个字节。

小型数据模式

默认情况下，小数据模式使用前 256 字节的内部数据内存。可以使用 8 位指针访问此内存。优点是指针存储只需要 8 位。作为参数传递的默认指针类型将使用堆栈上的一个寄存器或一个字节。

大型数据模式

默认情况下，大数据模式使用前 64 KB 的外部数据内存。可以使用 16 位指针访问此内存。作为参数传递的默认指针类型将使用堆栈上的两个寄存器或两个字节。

通用数据模式

通用数据模式使用 64 KB 的代码内存空间、64 KB 的外部数据内存空间和 256 字节的内部数据内存空间。作为参数传递的默认指针类型将使用堆栈上的三个寄存器或 3 个字节。

Far 通用数据模式

Far 通用数据模式使用 4 MB 的代码内存空间、4 MB 的外部数据内存空间和 256 字节的内部数据内存空间。作为参数传递的默认指针类型将使用堆栈上的三个寄存器或 3 个字节。

要求使用 24 位数据指针，使用--dptr 编译器选项显式设置。

Far 数据模式

Far 数据模式使用 16 MB 的外部数据存储空间。这是唯一可以使用 24 位指针访问的内存。作为参数传递的默认指针类型将使用堆栈上的三个寄存器或 3 个字节。

要求使用 24 位数据指针，使用--dptr 编译器选项显式设置。

常量和字符串

常量和字符串的放置可以通过以下三种方式之一进行处理：

● 系统初始化时，常量和字符串从 ROM（非易失性存储器）复制到 RAM

字符串和常量的处理方式与初始化变量相同。这是默认行为，产品附带的所有预构建库都使用此方法。如果应用程序仅使用少量常量和字符串，并且微控制器在外部数据存储空间中没有非易失性存储器，则应选择此方法。请注意，此方法需要在非易失性内存和易失性内存中为常量和字符串留出空间。

● 常量放在位于外部数据存储器空间的 ROM（非易失性存储器）中

常量和字符串使用与普通外部数据内存相同的访问方法进行访问。这是最有效的方法，但只有在微控制器在外部数据存储空间中具有非易失性存储器。要使用此方法，应使用内存属性__xdata__rom、__far22__rom、__far__rom 或__maging__rom 显式声明常量或字符串。这也是如果使用了--place__constants=data__rom 选项，则为默认行为。

请注意，如果使用了 Far、Far Generic、Generic 或 Large 数据模式之一，则可以使用此选项。

● 常量和字符串位于代码内存中，不会复制到数据内存中。

此方法占用外部数据内存空间中的内存。但是，位于代码内存中的常量和字符串只能通过指针__code，__far22_code，__far_code，__huge_code,或 __generic. 进行访问。

只有在使用大量字符串和常量且其他两种方法都不合适时，才应考虑使用此方法。将运行时库与此方法一起使用时会出现一些复杂情况，请参阅将常量和字符串放置在代码内存中，第 83 页。

在代码内存中放置常量和字符串

如果要在代码内存空间中定位常量和字符串，则必须使用选项--place__constants=code 来编译项目。需要注意的是,采用常量参数作为输入的标准运行时库函数,如 sscanf 和 sprintf，仍然希望在数据内存而不是代码内存中找到这些参数。相反，pgmspace 中声明了一些特定于 8051 的 CLIB 库函数。h 允许访问代码内存中的字符串。

自动变量和参数的存储

C 标准将在函数中定义且未声明为静态的变量命名为自动变量。其中一些变量被放入处理器寄存器中；其余部分放置在堆栈上或静态覆盖区域中。从语义的角度来看，这是等效的。主要区别在于，访问寄存器的速度更快，与变量位于堆栈或静态覆盖区域时相比，需要的内存更少。堆栈在运行时动态分配，而静态覆盖区域在链接时静态分配。有关在使用堆栈或静态覆盖区域存储变量之间进行选择的信息，请参阅选择调用约定，第 83 页。

自动变量只能在函数执行时存在；当函数返回时，释放堆栈上分配的内存。

选择调用约定

适用于 8051 的 IAR C/C++编译器提供了六种不同的调用约定，用于控制如何将内存用于参数和本地声明的变量。您可以指定默认调用约定，也可以显式声明每个函数的调用约定。

下表列出了可用的调用约定：

Calling convention	Function attribute	Stack pointer	Description
Data overlay	Idata overlay	Idata reentrant	
__data_overlay	--		数据存储器中的覆盖帧用于本地数据和参数。
__idata_overlay	--		idata 内存中的覆盖帧用于本地数据和参数。
__idata_reentrant		SP	idata 堆栈用于本地数据和参数。
Pdata reentrant	__pdata_reentrant	PSP	位于 pdata 内存中的模拟堆栈用于本地数据和参数。
Xdata reentrant	__xdata_reentrant	XSP	位于 xdata 内存中的模拟堆栈用于本地数据和参数。
Extended stack reentrant	__ext_stack_reentrant	ESP:SP	扩展堆栈用于本地数据和参数。

表 9：调用约定

只有可重入模型严格遵守标准 C。覆盖调用约定不允许递归函数并且它们不是可重入的，但它们在所有其他方面都遵守标准 C。

选择调用约定也会影响一个函数如何调用另一个函数。编译器会自动处理这个问题，但是如果你用汇编语言编写函数，你必须知道在哪里以及如何找到它的参数以及如何返回正确。有关详细信息，请参阅调用约定，第 197 页。

指定默认调用约定

您可以使用命令行选项 --calling_convention=convention 选择默认调用约定，请参见 --calling_convention，第 291 页。

要在 IDE 中指定调用约定，请选择 Project>Options>General 选项>目标>调用约定。

为单个函数指定调用约定

在您的源代码中，您可以使用 __idata_reentrant 函数属性声明要使用的各个函数，例如 Idata 可重入调用约定，例如：

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

页码:84

调用约定和匹配数据模式

调用约定的选择与应用程序使用的默认指针（由数据模式指定）密切相关。只有当默认指针可以访问所选调用约定使用的堆栈或静态覆盖区域时，在没有特定内存属性或 pragma 指令的情况下编写的应用程序才能正常工作。

下表显示了无需显式键入 declare 指针和参数即可组合哪些数据模式和调用约定：

数据模式	默认指针	调用约定	堆栈或覆盖帧内存
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

表 10：数据模式和调用约定

编译器不会允许不一致的默认模型，但如果任何函数已显式声明为使用非默认调用约定，则可能必须在声明指针时显式指定指针类型。

页码:85

例子

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

如果使用小型数据模式编译上述示例，编译将成功。在本例中，指针 xp 将是默认的指针类型，即 idata。

idata 指针可以由位于 idata 堆栈上的局部变量 x 的地址实例化。

另一方面，如果应用程序是使用大数据模式编译的，则编译将失败。变量 x 仍将位于 idata 堆栈上，但默认类型的指针 xp 将是与 idata 地址不兼容的扩展数据指针。编译器将生成以下错误消息：

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

这里，int * 指针是默认类型，即实际上是 int __xdata *。

但是，如果将指针 xp 显式声明为 __idata 指针，则可以使用大数据模式成功编译应用程序。 源代码将如下所示：

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

混合调用约定

并非所有调用约定都可以在同一个应用程序中共存，并且如果您使用多个调用约定，则只能以有限的方式执行传递本地指针和返回结构。

只能同时使用一个内部堆栈--例如，PUSH 和 CALL 指令使用的堆栈。 大多数 8051 设备仅支持位于 IDATA 中的内部堆栈。

一个值得注意的例外是扩展设备，如果您使用扩展堆栈选项，它允许内部堆栈位于 XDATA 中。这意味着 idata 可重入和扩展堆栈可重入调用约定不能组合在同一个应用程序中。 此外，xdata 可重入调用约定不能与扩展堆栈可重入调用约定相结合，因为同一时间只能有一个堆栈位于 XDATA 中。

混合调用约定还可以对参数和返回值进行限制。

这些限制仅适用于作为参数传递的本地声明指针以及返回结构声明变量时。

如果默认指针（由数据模式指定）无法引用正在使用的调用约定所隐含的堆栈，则会出现问题。如果默认指针可以引用位于堆栈上的对象，则调用不受限制。

例子

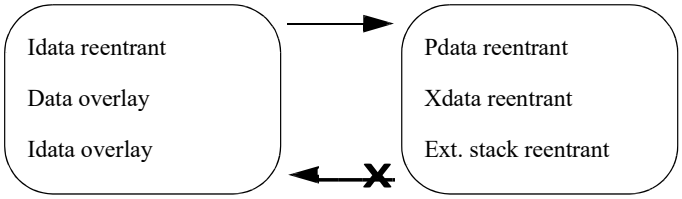
```
__xdata_reentrant void foo(int *ptr)
{
  *ptr = 20;
}
__idata_reentrant void bar(void)
{
  int value;
  foo(&value);
}
```

如果使用 Small 数据模式，则应用程序将成功编译，并且变量值将位于 idata 堆栈上。引用变量值的实际参数 &value 在引用位于 idata 堆栈上的对象时将成为 __idata * 指针。函数 foo 的形式参数将是默认指针类型，即 __idata，它与调用函数时使用的实际参数类型相同。

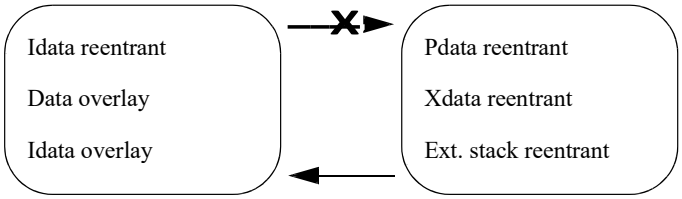
同样的道理也适用于结构类型的返回值。调用函数将在调用函数的堆栈上保留空间并将隐藏参数传递给被调用函数。这个隐藏参数是一个指针，指向调用者堆栈上返回值应该位于的位置。

因此，应用程序在使用不同调用约定的函数如何相互调用方面受到限制。使用可通过默认指针类型访问的堆栈的函数可以调用所有其他函数，而不管被调用函数使用的调用约定。

调用其他函数的函数
这是函数可以相互调用的方式：
Idata 默认指针



Xdata/far 默认指针



箭头表示可以调用。X 表示除非使用函数属性或杂注指令显式声明源代码，否则无法进行调用。
因此，如果应用程序使用 Small 数据模式编译，则__idata_reentrant 函数可以调用__xdata_reentrant 函数。但是__xdata_reentrant 函数不能调用__idata_reentrant 函数。但是，所有应用程序都可以显式地声明类型，以便它们按照预期的方式工作。
使用扩展堆栈排除了使用 idata 和 xdata 堆栈的可能性;
扩展的堆栈可重入函数只能与 pdata 可重入函数共存。扩展堆栈位于 xdata 内存中，可以在调用约定兼容性上下文中以与 xdata 堆栈相同的方式查看。

堆栈
堆栈可以包含：
- 不存储在寄存器中的局部变量和参数
- 表达式的临时结果
- 函数的返回值（除非在寄存器中传递）
- 中断期间的处理器状态
- 函数返回前应恢复的处理器寄存器(调用保存寄存器)
- 由调用函数保存的寄存器(caller-save 寄存器)

堆栈是一个固定的内存块，分为两部分。第一部分包含调用当前函数的函数所使用的已分配内存，以及调用当前函数的函数所使用的内存，等等。第二部分包含可分配的空闲内存。这两个区域之间的边界称为堆栈顶部，由堆栈指针表示，对于 idata 和扩展堆栈来说，堆栈指针是专用的处理器寄存器。通过移动堆栈指针在堆栈上分配内存。

函数永远不应该引用堆栈区域中包含空闲内存的内存。原因是，如果发生中断，被调用的中断函数可以分配、修改，当然还有释放堆栈上的内存。

请参见第 247 页的堆栈注意事项和第 132 页的建立堆栈内存。

优势

堆栈的主要优点是，位于程序不同部分的函数可以使用相同的内存空间来存储它们的数据。与堆不同，堆栈永远不会成为碎片或遭受内存泄漏。

函数可以直接或间接地调用自己(递归函数)，每次调用都可以在堆栈上存储自己的数据。

潜在的问题

堆栈的工作方式使得不可能存储函数返回后应该存在的数据。下面的函数演示了一个常见的编程错误。它返回一个指向变量 x 的指针，该变量在函数返回时不复存在。

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

另一个问题是堆栈空间耗尽的风险。这将会发生在一个函数调用另一个，它反过来调用第三个，等等，以及堆栈使用的总和每个函数都大于堆栈的大小。如果是大型数据对象，则风险更高存储在堆栈上，或者当使用递归函数时。

8051 编译器使用的堆栈

您可以将编译器配置为使用三个不同的堆栈；两个硬件堆栈中的一个(由处理器内部支持，并由，例如，PUSH、POP、CALL、和 RET 指令)和两个模拟的堆栈。

微控制器一次只支持一个硬件堆栈。标准这是 idata 堆栈，位于 idata 内存中。扩展的 8051 设备有一个选项是使用位于 x 数据内存中的扩展硬件堆栈。

模拟的 xdata 和 pdata 堆栈在硬件上不受支持；他们是相反由编译器软件支持。

用于保存堆栈的数据段是其中之一 ISTACK, PSTACK, XSTACK, 或 EXT_STACK.

静态覆盖

静态覆盖是一种系统，其中本地数据和功能参数存储在内存中的静态位置。每个函数都与一个覆盖框架相关联，该框架具有固定的大小，并且包含用于局部变量、函数参数和临时数据的空间。

静态覆盖可以在 8051 微控制器上产生非常有效的代码，因为它对直接寻址有很好的支持。然而，可直接访问的内存量是有限的，因此静态覆盖系统只适用于小型应用程序。

然而，静态覆盖系统存在一个问题；很难支持递归和可重入的应用程序。在可重入和递归系统中，同一个函数的几个实例可以同时活动，所以每个函数只有一个覆盖框架是不够的；相反，编译器必须能够为同一个函数处理多个覆盖帧。因此，静态覆盖系统受到限制，不支持递归或可重入。

有关静态覆盖系统使用的功能指令的信息，请参阅 8051 的 IAR 汇编程序用户指南。

堆上的动态内存

在对象被显式释放之前，堆上分配的对象内存将一直存在。这种类型的内存存储对于直到运行时才知道数据量的应用程序非常有用。

在 C 语言中，内存是使用标准库函数 malloc 或者相关函数 calloc 和 realloc 来分配的。使用 free 再次释放内存。

在 C++中，一个特殊的关键字 new 分配内存并运行构造函数。用 new 分配的内存必须用关键字 delete 释放。

编译器支持具有外部存储器的 8051 设备的多个存储器类型的堆：

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	--

表 11：内存类型中支持的堆

在 DLIB 中，要使用特定的堆，请在 malloc、空闲、allaloc 前面添加内存属性，例如 __pdata_malloc。默认函数将使用特定的堆变量，这取决于项目设置，如数据模式。
有关如何设置堆内存大小的信息，请参阅设置堆内存，第 135 页。

潜在问题

使用堆分配的数据对象的应用程序必须非常仔细地设计，因为很容易出现无法在堆上分配对象的情况。
如果应用程序使用了太多内存，堆可能会变得耗尽。如果没有释放不再使用的内存，它也可以变满。
对于每个分配的内存块，需要几个字节的数据用于管理目的。对于分配了大量小块的应用程序，这种管理开销可能会很大。
还有碎片化的问题；这意味着在堆中，一小部分空闲内存被分配对象使用的内存分隔。如果没有足够大的空闲内存适合该对象，则不可能分配一个新对象，即使空闲内存的大小之和超过了该对象的大小。
不幸的是，随着内存的分配和释放，碎片化往往会增加。因此，被设计为长时间运行的应用程序应该尽量避免使用分配在堆上的内存。

可替代的内存分配函数

由于 8051 微控制器上的内部内存非常有限，因此编译器只支持位于外部数据内存空间中的堆。
要在使用 CLIB 运行时库时完全支持这个堆，库中还包含其他函数：

```
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 *)
```

建议使用这些替代函数来代替标准 C 库函数 malloc、calloc、realloc 和 free。函数的 __xdata 版本在使用 16 位数据指针 (DPTR) 时可用，而 __far 版本在使用 24 位数据指针时可用。

标准函数可与 Far、Far Generic、Generic 和 Large 数据模式一起使用；他们将调用相应的 __xdata 或 __far 替代函数，具体取决于您使用的 DPTR 的大小。但是，如果使用 Tiny 或 Small 数据模式，则根本无法使用标准的 malloc、calloc、realloc 和 free 函数，在这些情况下，您必须显式使用相应的替代函数来使用外部内存中的堆。

可选的 __xdata 和 __far 内存分配函数与标准函数之间的区别在于返回值的指针类型和参数的指针类型。函数 malloc、calloc 和 realloc 返回指向已分配内存区域的指针，而 free 和 realloc 函数将指针参数指向先前分配的区域。这些指针必须是与堆所在的内存类型相同的指针，与默认内存和指针属性无关。

注意：相应的功能也可以在 DLIB 运行时环境中使用。

虚拟寄存器

编译器使用一组虚拟寄存器(位于数据内存中)像使用其他寄存器一样使用。编译器至少需要 8 个虚拟寄存器，但最多可以使用 32 个。

一组更大的虚拟寄存器使得编译器可以分配更多的变量到寄存器中。然而，更大的虚拟寄存器集也需要更大的数据存储区域。

在 Large 数据模式中，您可能应该使用更多的虚拟寄存器，例如 32 个，以帮助编译器生成更好的代码。

在 IDE 选择菜单： Project>Options>General Options>Target>Number of virtual registers.

在命令行上,使用选项 --nr_virtual_regs 指定虚拟寄存器的数量。见--nr_virtual_regs，第 314 页。

虚拟位寄存器

编译器保留一个字节的位可寻址内存供内部使用，用于存储本地声明的 bool 变量。虚拟位寄存器可以位于位可寻址存储器区域（0x20 - 0x2F）中的任何位置。建议您在此区域中使用第一个或最后一个字节。

通过定义 ？VB 为适当的值，例如：

-D?VB=2F

（空白页）
页码:94

函数

- 函数相关的扩展
- 函数存储的代码模型和内存属性
- 中断、并发和操作系统相关编程的原语
- 内联函数

函数相关的扩展

除了支持标准 C 之外，编译器还提供了几个用于在 C 中编写函数的扩展。使用这些，您可以：

- 控制函数在内存中的存储
- 使用原语进行中断、并发和与操作系统相关的编程
- 控制函数内联
- 方便函数优化
- 访问硬件函数。

编译器使用编译器选项、扩展关键字、Pragma 预编译指令和内部函数来支持这一点。

有关分页函数相关扩展的详细信息，请参见分页函数一章。

有关优化的更多信息，请参阅嵌入式应用程序的高效编码，第 257 页。有关用于访问硬件操作的可用内部函数的信息，请参阅内部函数一章。

函数存储的代码模型和内存属性

使用代码模型指定默认情况下编译器应将函数放置在内存的哪个部分。从技术上讲，代码模型控制以下内容：

- 存储函数的默认内存范围，即默认内存属性
- 最大模块尺寸
- 最大应用程序大小

您的项目一次只能使用一个代码模型，所有用户模块和所有库模块都必须使用相同的模型。
这些代码模型可用:

代码模式	默认核心	指针大小	说明
Near Plain		2 bytes	支持高达 64kbytes 的可编译代码，可以访问整个 16 位的地址空间
Banked --		2 bytes	支持分页函数调用,参见分页系统的代码模型，第 107 页。函数可以通过使用 near _ func memory 属性显式地放置在根分页中,请参阅 near _ func， 第 354 页。
Banked Extended2 Extended2		3 bytes	支持分页函数调用，请参阅分页系统的代码模式，第 107 页。
Far Extended1		3 bytes	具有扩展代码存储器和真正的 24 位指令的设备

表 12：代码模式

有关在 ide 中指定代码模型的信息，请参见 8051 的 IDE 项目管理和构建指南。
使用- code_model 选项指定项目的代码模型；参见第 293 页的代码模型。
具有函数内存属性的指针在指针之间以及指针和整数类型之间的隐式和显式强制转换中有限制。有关限制的信息，请参见第 333 页的“ Casting”。
有关语法信息和每个属性的更多信息，请参阅扩展关键字一章。
在“汇编语言接口”一章中，当我们描述如何从汇编语言调用 C 函数时，会更详细地研究生成的代码，反之亦然。

使用函数记忆属性
可以覆盖单个函数的默认位置。使用适当的函数内存属性来指定此属性。这些属性可用：

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,

表 13：函数内存属性

具有函数内存属性的指针在指针之间以及指针和整数类型之间的隐式和显式强制转换方面存在限制。有关限制的信息，请参阅强制转换，第 333 页。有关语法信息和有关每个属性的详细信息，请参阅扩展关键字一章。

中断、并发和操作系统相关编程的原语
8051 的 IAR C/ C++编译器提供了以下与编写中断函数、并发函数和操作系统相关函数相关的原语:

- 扩展关键字：__interrupt、__task、__monitor
- Pragma 预编译指令#pragma vector
- 内部函数：__enable_interrupt、__disable_interrupt、__get_interrupt_state、__set_interrupt_state。

中断函数
在嵌入式系统中，使用中断是一种立即处理外部事件的方法;例如，检测按钮被按下。

中断服务例程
通常，当代码中发生中断时，微控制器立即停止执行它运行的代码，转而开始执行中断例程。

重要的是，在处理完中断后，被中断函数的环境被恢复了（这包括处理器寄存器和处理器状态寄存器的值）。这使得在处理中断的代码被执行后，有可能继续执行原来的代码。
8051 微控制器支持许多中断源。对于每个中断源，可以编写一个中断例程。每个中断例程都与一个向量号相关联，这个向量号在芯片制造商的 8051 微控制器文件中指定。如果你想用同一个中断例程来处理几个不同的中断，你可以指定几个中断向量。

中断向量和中断向量表

中断向量是进入中断向量表的偏移量。
如果在中断函数的定义中指定了一个向量，处理器的中断向量表就会被填充。
也可以定义一个没有向量的中断函数。如果应用程序能够在运行时填充或改变中断向量表，这就很有用。
头文件 iodevice.h，其中 device 对应于选定的设备，包含了现有中断向量的预定义名称。

定义一个中断函数--一个例子

为了定义一个中断函数，可以使用__interrupt 关键字和#pragma vector 指令。比如说

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

注意：一个中断函数必须有返回类型 void，而且不能指定任何参数。

中断和 C++成员函数

只有静态成员函数可以成为中断函数。当一个非静态成员函数被调用时，它必须被应用于一个对象。当一个中断发生并调用中断函数时，没有对象可以应用该成员函数。

寄存器分页

基本的 8051 微控制器支持四个寄存器分页。如果你指定了一个由中断使用的寄存器分页，当中断函数进入时，寄存器 R0-R7 不会被保存在堆栈中。相反，应用程序会切换到你指定的分页，然后在中断功能退出时再切换回来。这是一个有用的方法，可以加快重要中断的速度。

的符号 REGISTER_BANK 的值来改变运行时系统使用的默认寄存器页。其他的寄存器分页可以通过使用#pragma register_bank 指令与中断例程相关联。
注意：编译器的运行系统所使用的分页不能用于中断，不同的中断可以互相中断，不能使用同一个寄存器分页。

例子

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

在链接器命令文件中，它可以看起来像这样。

```
-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
```

如果你将一个寄存器分页与中断例程一起使用，寄存器分页占用的空间不能用于其他数据。

监控函数

一个监控函数会使中断在函数的执行过程中被禁用。在函数进入时，状态寄存器被保存，中断被禁用。
在函数退出时，原来的状态寄存器被恢复，从而在函数调用前存在的中断状态也被恢复。
为了定义一个监控函数，你可以使用__monitor 关键字。更多信息，请参见__monitor，第 354 页。
避免在大型函数上使用__monitor 关键字，因为否则中断会被关闭太长时间。

在 C 语言中实现信号灯的例子

在下面的例子中,使用一个静态变量和两个监控函数实现了一个二进制信号--也就是一个突发事件。
一个监控函数的工作原理类似于一个关键区域，即不能发生中断，进程本身也不能被换出。
信号标可以被一个进程锁定，用于防止进程同时使用一次只能被一个进程使用的资源，例如 USART。__monitor 关键字保证了锁操作是原子性的。
换句话说，它不能被中断。

```
/* 这就是锁变量。当非零时，有人拥有它。 */
静态易失性无符号整数 sTheLock = 0;
测试锁是否打开的函数，如果是则取它。
成功返回 1，失败返回 0。*/
__monitor int TryGetLock(void)
{
if (sTheLock == 0)
{
  /* 成功，没有人锁定。 */
  sTheLock = 1;
  return 1;
}
else
{
  /* 失败了，别人有了锁。 */
  return 0;
}
}
/* 函数以解锁锁定。
* 它只能由拥有锁的人调用。
*/
__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();
}
```

在 C++中实现信号灯的例子

在 C++中，实现小方法的目的是为了让它们被内联，这很常见。然而，编译器并不支持对使用__monitor 关键字声明的函数和方法进行内联。

在下面这个 C++的例子中，一个自动对象被用来控制监控块，它使用内在的函数而不是__monitor 关键字。

```
#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();
  }
}
```

内联函数

函数内联是指一个在编译时就知道定义的函数被集成到其调用者的主体中,以消除函数调用的开销。

这种优化是在高优化级别上进行的，通常会减少执行时间，但可能会增加代码的大小。由此产生的代码可能变得更加难以调试。内联是否真的发生，取决于编译器的启发式方法。
编译器启发式地决定哪些函数需要内联。在对速度、大小进行优化时，或者在大小和速度之间进行平衡时，会使用不同的启发式方法。
通常情况下，在对大小进行优化时，代码大小不会增加。

C 与 C++的语义

在 C++中，一个特定的内联函数在不同的翻译单元中的所有定义必须完全相同。如果该函数在一个或多个翻译单元中没有被内联，那么这些翻译单元中的一个定义将被作为函数的实现。在 C 语言中，你必须手动选择一个翻译单元，其中包括内联函数的非内联版本。你可以通过在该翻译单元中明确地将该函数声明为 extern 来做到这一点。如果你在一个以上的翻译单元中将该函数声明为 extern。
链接器会发出一个多重定义的错误。此外，在 C 语言中，内联函数不能引用静态变量或函数。例如：

```
//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();
```

控制函数内联的特征

有几种机制可以控制函数的内联。
inline 关键字告知编译器，紧随指令之后定义的函数应该被内联。
如果你在 C 或 C++模式下编译你的函数，该关键字将分别根据其在标准 C 或标准 C++中的定义进行解释。

语义上的主要区别是，在标准 C 中，你不能（一般来说）简单地在头文件中提供一个内联定义。你必须在其中一个编译单元中提供一个外部定义，方法是将内联定义指定为该编译单元中的外部定义。

页码:104

该编译单元中的外部定义

● #pragma inline 类似于 inline 关键字，但不同的是，编译器总是使用 C++内联语义。通过使用#pragma inline 指令，你也可以禁用编译器的启发式方法，强制内联或完全禁用内联。更多信息，见 inline，第 370 页。

● --use_C++_inline 强制编译器在编译标准 C 源代码文件时使用 C++语义。

● --no_inline、#pragma optimize=no_inline 和#pragma inline=never 都禁用函数内联。默认情况下，函数内联在优化级别高时被启用。

编译器只有在已知定义的情况下才能内联一个函数。通常情况下，这被限制在当前的翻译单元。然而，当使用多文件编译的-mfc 编译器选项时，编译器可以内联多文件编译单元中所有翻译单元的定义。欲了解更多信息，请参见多文件编译单元，第 264 页。关于函数内联优化的更多信息，参见函数内联。第 267 页。

（空白页）
页码:106

函数分页

以下页面包含这些主题：

●分页系统介绍
●编写存储库的源代码
●分页切换
●下载到内存
●调试分页应用程序

请注意，当您阅读本章时，您应该已经熟悉第 1 部分中描述的其他概念。使用编译器。

分页系统简介

如果您使用 8051 微控制器，其自然地址范围为 64 KB 内存，则它具有 16 位寻址能力。Banking 是一种技术，用于扩展处理器可以访问的内存量，使其超出处理器自然寻址方案大小设置的限制。换句话说，可以访问更多的代码。
存储区内存用于需要大量可执行代码的项目，以至于基本 8051 微控制器的自然 64 KB 地址范围不足以包含所有代码。

分页系统的代码模式

用于 8051 的 IAR C/C++ 编译器提供了两种代码模型，用于存储您的代码，允许高达 16 MB 的 ROM 内存：
●分页代码模型
允许将 8051 微控制器的代码存储区扩展至多达 256 组额外的 ROM 存储器。每个存储库最大为 64 KB，减去根存储库的大小。

●分页扩展 2 代码模型

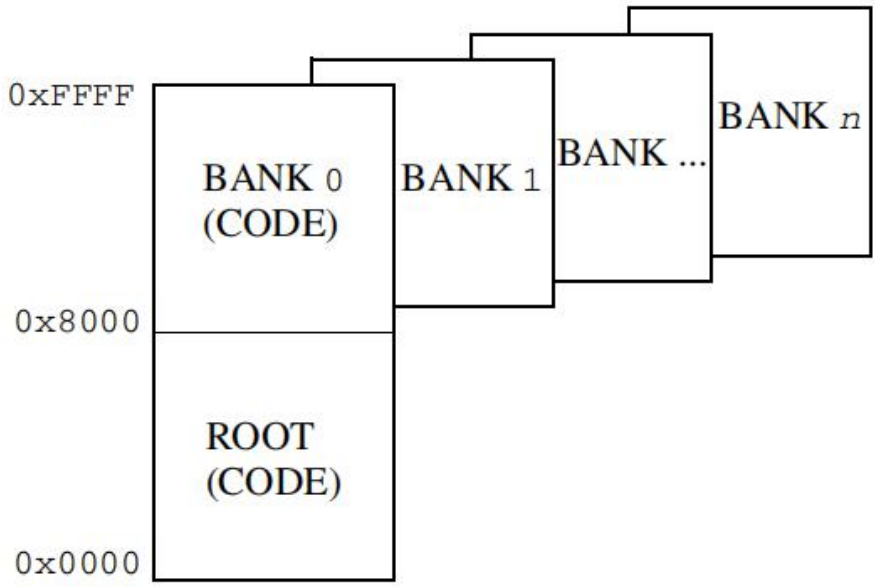
允许 8051 微控制器的代码存储区扩展到 16 组额外的 ROM 存储器。每个存储体可以达到 64k 字节。你不需要为传统的根分页预留空间。

您的硬件必须提供这些额外的物理内存库，以及解码代表内存库编号的高地址位所需的逻辑。因为一个基本的 8051 微控制器在任何时候都不能处理超过 64k 字节的存储器，所以由分页业引入的扩展存储器范围意味着必须特别小心。CPU 一次只能看到一个存储体，其余的存储体必须进入 64k 字节的地址范围才能使用。

组合代码模型的内存布局

你可以在代码内存中的任何地方放置内存块区域，但是必须有一个根区域来保存运行时环境代码和内存块开关的中继函数。

下图显示了一个 8051 分页代码存储器布局的示例:



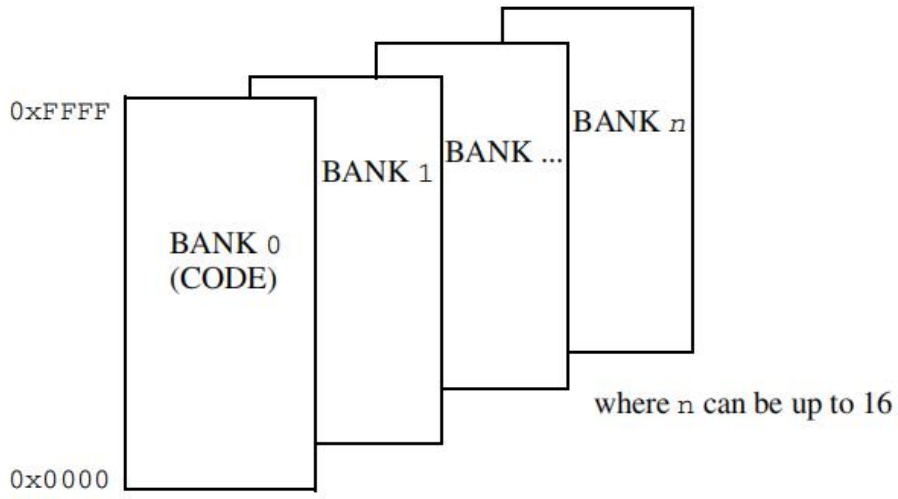
将根区域放置在地址 0 及以上是可行的。有关如何将函数显式放置在根区域的信息，请参见第 111 页的“组合与非组合函数调用”。

分页扩展 2 代码模型的存储器布局

基于 Mentor Graphics M8051EW 内核的设备--这意味着您正在使用- core=extended2 编译器选项--使用与经典 8051 微控制器不同的分页机制。

页码:108

下图显示了 8051 分页代码内存布局的示例:



Bank 0 保存执行期函式库代码。

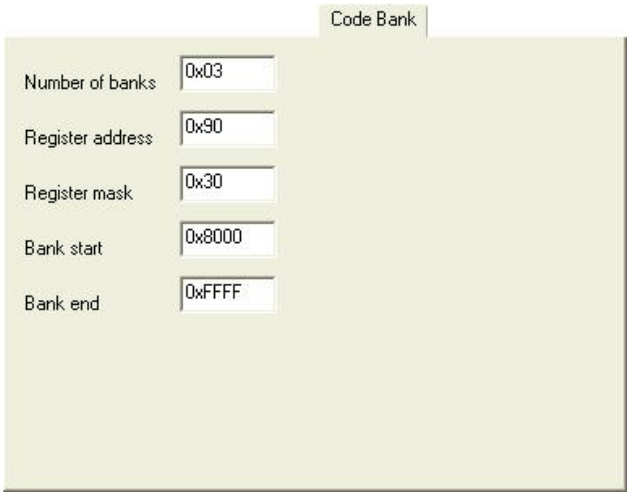
设置分页模式的编译器在 ide 中指定分页代码模型，选择 project > options > general options > target > code model。选择 banked 或者，
如果您正在使用 extded2 核心选项，Banked extended2.

编译您的模块为 bank 模式，使用带有适当参数的编译器选项:
--code_model={banked|banked_ext2}

注意: 使用 core plain (经典 8051)时可以使用 分页 代码模式，使用 core extded2(用于指导图形 m8051ew core)时可以使用 分页 扩展 2 代码模式。

为分页模式设置链接器
当连接一个分页模式的应用程序时，你必须把你的代码段放入与你的硬件中可用的物理库相对应的分页中。然而，物理分页的大小和位置取决于你的根分页的大小，而根分页的大小又取决于你的源代码。
为链接器设置代码库的最简单方法是使用 IAR Embedded Workbench IDE。然而，如果你的硬件内存配置很不寻常，你可以使用链接器命令文件来为链接器设置分页系统。

在 iar 嵌入式工作台端设置代码库选项；选择：
Project>Options>General Options>Code Bank.



根据内存配置，使用文本框设置分页业务。

1 在“分页数量”文本框中指定分页数量。
2 组交换例程基于用于组交换的 SFR 端口。
默认情况下，使用 P1。要设置不同的 SFR 端口，请在寄存器地址文本框中添加所需的端口。
3 如果整个 SFR 未用于选择活动存储组，请在寄存器掩码文本框中指定适当的位掩码，以指示使用的位。例如，如果仅使用位 0 和 1，则指定 0x03。
4 在分页起始文本框中，键入分页区域的起始地址。相应地，在分页结束文本框中键入结束地址。

在预定义的链接器命令文件中，预定义了一组符号：

```
-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 */
```

如果值不合适，您可以简单地更改它们以匹配您的硬件内存配置。但是，如果这些符号不适合您的配置，那么您必须根据您的需要来调整它们。
因此，您可能需要通过链接过程进行一些尝试，以确定最佳的内存配置设置。

页码:110

为分页内存编写源代码

编写用于分页区的源代码与编写标准内存的代码没有太大区别，但有一些问题需要注意。这些主要涉及将您的代码划分为函数和源模块，以便链接器可以最有效地将它们放入分页中，以及分页代码与非分页代码之间的区别。

C/C++ 语言注意事项

从 C/C++ 语言的角度来看，任何 C/C++ 程序都可以编译为存储区内存。唯一的限制是函数的大小，因为它不能大于分页的大小。

分页大小和代码大小

编译的每个分页 C/C++源函数将生成一个单独的段部分，并且从分页函数生成的所有段部分将位于分页-代码段中。段部分中包含的代码是一个不可分割的单元，也就是说，链接器不能分解段部分并将其一部分放在一个分页中，将其一部分放在另一个分页中。因此，分页函数生成的代码必须始终小于分页大小。

但是，一些优化要求从同一模块（源文件）中的分组函数生成的所有段部分必须作为一个单元链接在一起。在这种情况下，同一模块中所有 分页函数的组合大小必须小于分页尺寸。这意味着您必须考虑每个 C/C++ 源文件的大小，以便生成的代码适合您的分页。如果您的任何代码段大于指定的分页 大小，链接器将发出错误。
如果您需要单独指定任何代码的位置,您可以将每个函数的代码段重命名为一个不同的名称，以便您单独引用它。

要将函数分配给特定段，请使用 @ 运算符或 #pragma location 指令。请参见控制内存中的数据和函数放置，第 259 页。有关段的更多信息，请参阅链接概述和链接应用程序章节。

分页与非分页功能调用

在 分页代码模式中,区分 non-分页和分页函数调用很重要,因为 non-分页 函数调用比 分页 函数调用更快并且占用更少的代码空间。因此，熟悉导致非分页函数调用的函数声明类型很有用。

注意：在分页 扩展 2 代码模式中，所有代码都是分页的，这意味着不需要区分非分页函数调用和分页函数调用。在此上下文中，函数调用是编译器在 C/C++源代码中的函数调用另一个 C/C++函数或库例程时生成的机器指令序列。这还包括保存返回地址，然后将新的执行地址发送到硬件。

假设您使用的是分页代码模式，则有两个函数调用序列：
● 非分页函数调用：返回地址和新执行地址是 16 位值。声明为__near_func 的函数将具有非分页函数调用。

● 分页函数调用：返回地址和新执行地址是 24 位（3 字节）值（分页代码模式中的默认值）在分页代码模式中，所有非类型化函数都将位于分页内存中。

但是，可以通过使用__near_func memory 属性显式声明函数为非分页函数。此类函数不会生成分页调用，将位于 NEAR_CODE 段而不是 BANKED_CODE 段中。当不产生分页调用时，NEAR_CODE 段必须位于根分页中。

将频繁调用的函数放在根分页中通常是一个好主意，以减少分页调用和返回的开销。

例子
在这个例子中，你可以看到组合函数 f1 是如何调用非组合函数的
f2:

```
__near_func void f2(void) /* Non-banked function */
{
/ * code here … */
}
void f1(void) /* Banked function in the Banked code model */
{
f2();
}
```

从 f1 中实际调用 f2 的方式与普通调用相同函数调用（LCALL）。
注意：有一个 __banked_func 内存属性可用，但您不需要明确使用它。 该属性仅供编译器内部使用。

无法分页的代码

在分页代码模型中，代码分页是通过将用于程序代码的地址空间分为两部分来实现的：非分页代码和分页代码。

在第 108 页的 banked extended2 代码模型的内存布局中，包含 non-banked 代码的部分称为根分页 k。必须始终存在一定数量的 non-banked 代码。

例如，中断服务程序必须始终可用，因为中断随时可能发生。

以下选定部分必须位于非分页区内存中：

- cstartup 例程，位于 CSTART 段中
- 中断向量，位于 INTVEC 段
- 中断服务程序，位于 NEAR_CODE 段
- 包含常量的段。这些都是以_AC、_C 和_N 结尾的段，见章节参考
- 包含初始化变量初始值的段可以位于根分页中，也可以位于 bank 0 中但不位于其他分页中，这些都是以_ID 结尾的段，参见章节参考
- 包含在运行库中的汇编程序编写的例程。它们位于 RCODE 段中
- 中继函数，位于 BANK_RELAYS 段
- 分页切换 例程，即将执行调用并从分页例程返回的例程。它们位于 CSTART 段中。

使用覆盖调用约定之一（__data_overlay 或 __idata_overlay）的函数或远函数（__far_func)不支持分页业务,因为只有函数类型属性 __far_func 和 __banked_func 之一可以同时用作系统中的关键字。

位于非分区内存中的代码将始终可供处理器使用，并且始终位于同一地址。

注意：即使中断函数不能被分页，中断例程本身也可以调用分页函数。

从汇编语言调用分页函数

在汇编语言程序中，调用位于另一个分页中的 C/C++ 函数需要使用与编译器相同的调用约定。有关此调用约定的信息，请参阅第 197 页的调用约定。要生成分页函数调用的示例，请使用第 194 页的从 C 调用汇编程序例程一节中描述的技术。

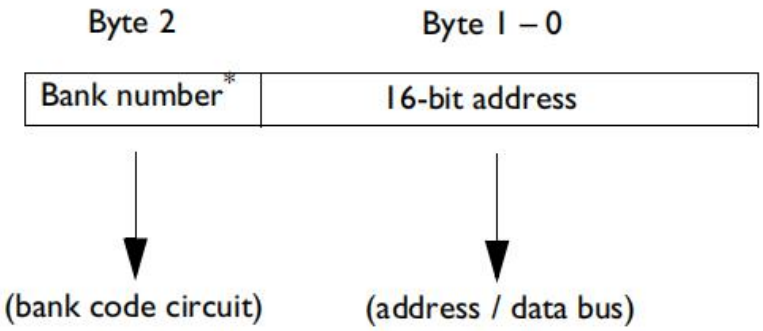
如果使用汇编语言编写分页函数，则还必须考虑调用约定。

分页切换

用 C 或 C++ 编写的分页系统的分页切换，通常不需要考虑分页切换机制，因为编译器会为您处理这个问题。但是，如果您用汇编语言编写分页函数，则可能需要注意汇编函数中的分页切换机制。此外，如果你想使用一个完全不同的分页切换解决方案，你必须实现你自己的分页切换例程。

访问分页代码

访问代码驻留在一个内存分页，编译器通过维护一个 3 字节的指针来跟踪一个分页函数的分页号码，它的形式如下:



*对于分页扩展 2 代码模式，高四位保存当前分页的编号，而低四位保存下一个分页的编号。

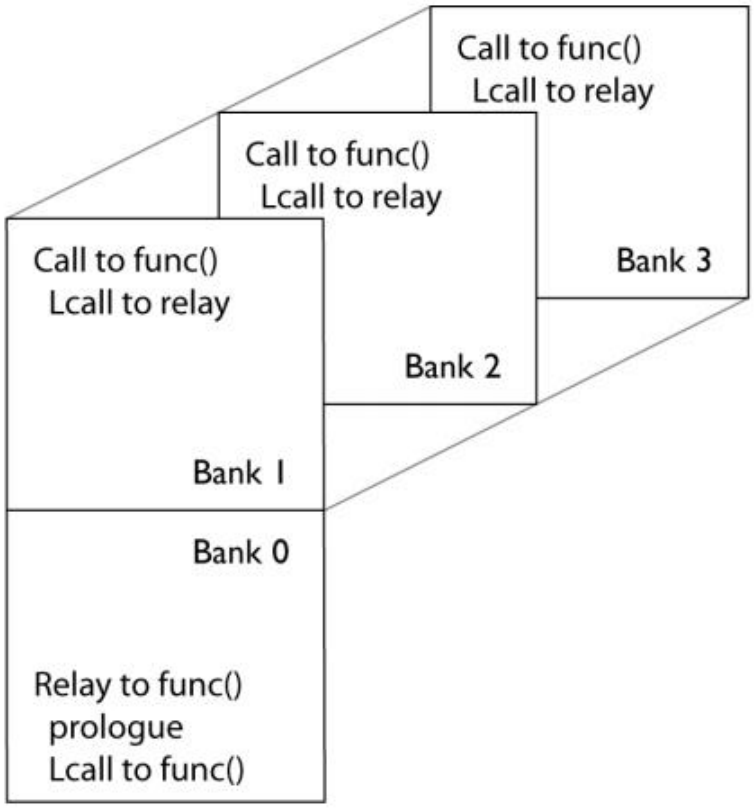
有关指针表示的更多细节，请参见第 330 页的指针类型。
默认的分页切换代码可以在提供的汇编语言源文件 iar_banked_code_support.s51 中找到，您可以在目录 8051\src\lib 中找到它。
两种代码模式之间的分页切换机制不同。

分页代码模式中的分页切换

默认的分页切换例程基于用于分页切换的 SFR 端口(P1)。当一个函数(调用者)调用另一个函数(被调用者)时，就会执行分页切换。编译器不知道链接器将把一个函数放在哪个分页中。

页码:114

更确切地说，是执行以下动作。
调用函数对位于根分页的中继函数执行 LCALL。
返回地址--当前的 PC（程序计数器）--被推到 idata 栈上。
中继函数向一个调度函数执行 LCALL。中继函数包含被调用者的地址。
对调度函数的调用意味着当前的分页号（调用者的分页号）被保存在 idata 堆栈中。下一个分页号码被保存在 P1 中作为当前分页。被调用者的地址被移到 PC 上，这意味着执行转移到被调用者函数上。换句话说，已经执行了一个分页的切换。
这个图说明了这些动作。



当被调用函数执行完毕后，它将执行一个?LCALL 到?BRET。在 BRET 中，调用者函数的库号被从 idata 堆栈中弹出，然后执行一条 RET 指令。现在的执行又回到了调用函数中。
分页扩展 2 代码模型中的分页切换
默认的分页切换使用 MEX 寄存器，内存扩展堆栈，以及 MEXSP 寄存器作为堆栈的指针。

页码:115

当一个函数(调用者)调用另一个函数(被调用者)时，就会执行分页切换。在分页切换之前，在调用程序功能中执行以下操作:

●3 字节指针的高字节--在访问存储代码中描述--放在 MEX 中。NB 寄存器(下一个分页寄存器)。

●执行 LCALL，这意味着在硬件中执行以下步骤:

●MEX 登记簿。CB(当前分页寄存器)放在内存扩展堆栈上。

●返回地址，即当前 PC，放在 idata 堆栈上。

●下一个 MEX 分页。NB 复制到当前分页 MEX.CB。

●3 字节指针的两个低位字节被复制到 PC，这意味着执行移动到被调用函数。换句话说，已经执行了分页切换。

当被调用函数执行后，RET 指令以相反的顺序执行存储体切换过程，这意味着执行回到调用函数中。

注意:内存扩展堆栈限制为 128 字节，这意味着函数调用深度不能超过这个限制。

修改默认分页切换例程

默认的分页切换代码可以在提供的汇编语言源文件 iar_banked_code_support.s51 中找到，您可以在目录 8051\src\lib 中找到它。

分页切换例程基于用于分页切换的 SFR 端口。

默认情况下，使用 P1。SFR 端口在链接器命令文件中由以下行定义:

-D?CBANK=PORTADDRESS

只要您使用这个解决方案，您必须做的唯一事情就是定义适当的 SFR 端口地址。

修改文件后，重新组装它，并替换正在使用的运行时分页中的目标模块。只需将它包含在您的应用程序项目中，它将代替库中的标准模块使用。

下载到存储器

有多种设备和内存可供使用，并且根据您的分页模式系统的外观，可能需要不同的操作。例如,存储器可能是容量超过 64 KB 的单个存储器电路,或者是几个较小的存储器电路。例如，存储器电路可以是 EPROM 或闪存。

默认情况下，链接器以 Intel 扩展格式生成输出，但您可以轻松更改此设置以使用您正在使用的工具所需的任何可用格式。
如果您使用的是 IAR Embedded Workbench IDE，则默认输出格式取决于使用的构建配置 - 发布或调试。
当您将代码下载到物理内存中时，可能需要特别注意。
例如，假设一个存储系统具有两个 32 KB 的 ROM 存储区，从 0x8000 开始。如果存储代码的大小超过 32 KB，当您链接项目时，结果将是单个输出文件，其中存储代码从 0x8000 开始并超过存储库上限。
现代 EPROM 程序员不需要一次将一个文件下载到一个 EPROM 中。它通过拆分文件并下载它来自动处理下载。但是，旧类型的程序员并不总是支持重定位，或者无法忽略 3 字节地址的高字节。
这意味着您必须手动编辑文件以将每个地址的高字节设置为 0，以便程序员可以正确定位它们。

调试分页应用程序

对于分页代码模型，C-SPY 调试器支持分页模式调试。要在 IAR Embedded Workbench IDE 中设置存储模式调试选项，请选择
Project>Options，选择 General Options 类别，然后单击 Code Bank 选项卡。
为以下选项键入适当的值：

- 寄存器地址指定用作分页寄存器的 SFR 地址
- Bank start 指定分页起始地址
- Bank end 指定分页结束地址
- 分页掩码指定用于选择活动分页的位
- 分页数指定硬件上可用的分页数。

使用其他调试器进行 分页 模式调试
如果您的仿真器不支持 分页 模式，一种常见的技术是将源代码分成不超过分页大小的较小部分，然后您可以使用 Near 或 Far 代码模型编译、链接和调试每个部分。
对各种功能组重复此过程。然后，当您在目标硬件上实际测试最终的分页系统时，许多与 C/C++ 编程相关的问题已经得到解决。

页码:117

（空白页）
页码:118

链接概述

●链接-概述
●分段和记忆
●链接过程的细节
●放置代码和数据--链接器配置文件
●系统启动时的初始化

链接-概述

IAR XLINK 链接器是一个强大、灵活的软件工具，用于开发嵌入式应用。它同样适用于连接小型、单文件、绝对汇编程序，也适用于连接大型、可重定位、多模块、C/C++或 C/C++和汇编混合程序。

链接器将一个或多个由 IAR 系统编译器或汇编器产生的可重定位的对象文件与对象库的必要部分结合起来，产生一个包含机器代码的可执行图像，供你使用的微控制器使用。

XLINK 可以生成 30 多种行业标准的加载器格式，此外还有 C-SPY 调试器所使用的专有格式 UBROF。

链接器将只自动加载那些你正在链接的应用程序实际需要的库模块。此外，链接器消除了不需要的分段部分。在链接过程中，链接器对所有模块进行全面的 C 级类型检查。

链接器使用一个配置文件，你可以为你的目标系统内存图的代码和数据区域指定单独的位置。

链接器产生的最终输出是一个绝对的、目标可执行的对象文件，可以下载到微控制器、C-SPY 或兼容的硬件调试探头。根据你选择的输出格式，输出文件可以选择包含调试信息。

为了处理库，可以使用库工具 XAR 和 XLIB。

分段和内存

在一个嵌入式系统中，可能有许多不同类型的物理内存。而且，你的代码和数据的部分在物理内存中的位置往往很关键。

出于这个原因，开发工具满足这些要求是很重要的。

什么是段？

段落是数据或代码片段的容器，应该被映射到物理内存的某个位置。每个段由一个或多个段部分组成。通常，每个具有静态存储时间的函数或变量都被放置在它自己的段部分。

段落部分是最小的可链接单元，它允许链接器只包括那些被提及的段落部分。一个段可以放在 RAM 中，也可以放在 ROM 中。放在 RAM 中的段一般没有任何内容，它们只占用空间。

注意：这里，ROM 存储器是指所有类型的只读存储器，包括闪存。

编译器为不同的目的使用了几个预定义的段。每个段都有一个通常描述该段内容的名称，并有一个表示内容类型的段内存类型来识别。

除了预定义的段之外，你也可以定义你自己的段。

在编译时，编译器会将代码和数据分配给各个段。IAR XLINK 链接器负责按照链接器配置文件中指定的规则，将这些段放在物理内存范围内。

提供了现成的链接器配置文件，但是，如果有必要，可以根据你的目标系统和应用程序的要求来修改它们。重要的是要记住，从链接器的角度来看，所有的段都是平等的；它们只是被命名为内存的一部分。

分段内存类型

每个段总是有一个相关的段内存类型。在某些情况下，单个段的名称与它所属的段内存类型相同，例如 CODE。在这些情况下，请确保不要将段名与段内存类型混淆。

XLINK 支持更多的段存储器类型，而不是上面描述的那些。然而，它们的存在是为了支持其他类型的微控制器。

关于各个段的更多信息，请参见段参考一章。

页码:120

详细的链接过程

由 IAR 编译器和汇编器生成的目标文件和库中的可重定位模块不能按原样执行。要使应用程序可执行，必须链接目标文件。

IAR XLINK Linker 用于链接过程。它通常执行以下过程（请注意，某些步骤可以通过命令行选项或链接器配置文件中的指令关闭）：

● 确定要包含在应用程序中的模块。始终包含程序模块。仅当库模块为从包含的模块引用的全局符号提供定义时，它们才会被包含在内。如果包含库模块的目标文件包含变量或函数的多个定义，则仅包含第一个定义。这意味着目标文件的链接顺序很重要。

● 确定要包含在应用程序中的所包含模块中的哪些段部分。仅包含应用程序实际需要的那些段。有几种方法可以确定需要哪些段部分，例如，__root 对象属性、#pragma required 指令和 -g 链接器选项。

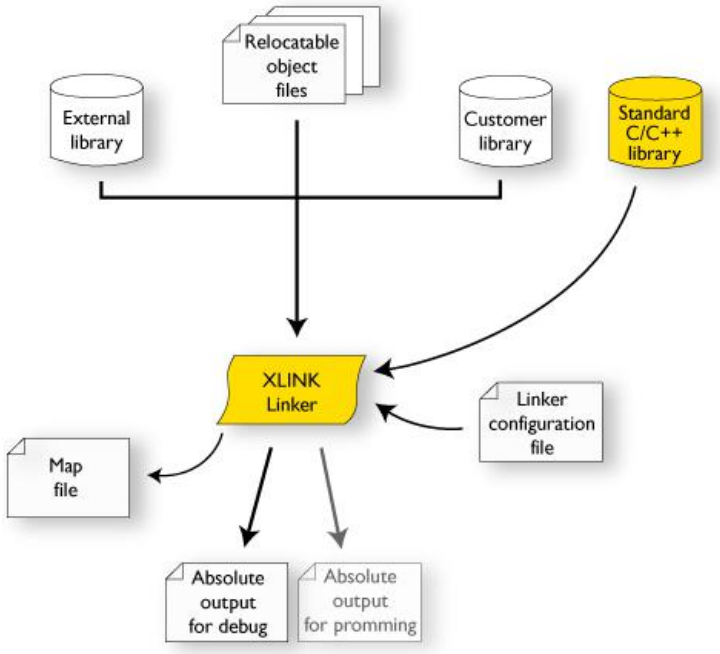
● 将通过复制初始化的每个段分成两段，一段用于 ROM 部分，另一段用于 RAM 部分。 RAM 部分包含标签，ROM 部分包含实际字节。字节在概念上被链接为驻留在 RAM 中。

● 根据链接器配置文件中的段放置指令确定每个段的放置位置。

● 生成包含可执行映像和任何调试信息的绝对文件。可重定位输入文件中每个所需段的内容是使用其文件中提供的重定位信息和放置段时确定的地址来计算的。如果不满足特定段的某些要求，例如，如果放置导致 PC 相关跳转指令的目标地址超出该指令的范围，则此过程可能导致一个或多个范围错误。

● （可选）生成一个映射文件，其中列出了段放置的结果、每个全局符号的地址，最后是每个模块的内存使用摘要。

下图显示了链接过程:



在链接期间，xlink 可能会产生错误消息和可选的映射文件。在映射文件中，您可以看到实际链接的结果，这对于理解为什么应用程序以原来的方式链接是有用的，例如，为什么包括段部分。如果一个段部分没有被包含，尽管您希望它被包含，原因总是没有从包含的部分.
备注 引用该段部分中: 要检查对象文件的实际内容，使用 xlib。参见 iar 链接器和库工具参考指南。

放置代码和数据

链接器配置文件在内存中段的放置是由 iar xlink 链接器执行的。它使用一个包含命令行选项的链接器配置文件，命令行选项指定可以放置段的位置，从而确保您的应用程序适合目标微控制器。要在不同的设备上使用相同的源代码，只需使用适当的链接器配置文件重新构建代码。特别是，链接器配置文件指定:

●段在内存中的位置
●最大堆栈大小
●最大堆大小（仅适用于 IAR DLIB 运行环境）。

该文件由一连串的链接器命令组成。这意味着链接过程将由所有命令依次支配。

页码:122

链接器配置文件的内容
除其他外，链接器配置文件包含三种不同类型的 XLINK 命令行选项：
● 使用的 CPU：
-cx51
这将指定目标微控制器。

● 文件中使用的常量的定义。这些是使用 XLINK 选项-D 定义的。也可以从应用程序访问使用-D 定义的符号。

● 放置指令（链接器配置文件的最大部分）。可以使用-Z 和-P 选项放置线段。前者将按找到的顺序放置片段部分，而后者将尝试重新排列它们以更好地利用内存。当应该放置段的内存不连续时，-P 选项很有用。但是，如果程序启动时需要初始化段或将其设置为零，则使用-Z 选项。

在链接器配置文件中，数字通常以十六进制格式指定。

但是，不必使用前缀 0x 或后缀 h。
注意：提供的链接器配置文件包含解释内容的注释。
有关链接器配置文件以及如何自定义它的更多信息，请参阅链接注意事项，第 127 页。
另请参阅《IAR 链接器和库工具参考指南》。

系统启动时初始化
在标准 C 中,运行时系统必须在应用程序启动时将分配在固定内存地址的所有静态变量初始化为已知值。静态变量可分为以下几类：

● 初始化为非零值的变量
● 初始化为零的变量
● 使用@运算符或#pragma location 指令定位的变量
● 声明为常量的变量，因此可以存储在 ROM 中
● 用__no_init 关键字定义的变量，这意味着它们根本不应该初始化。

静态数据存储段

编译器为每种类型的变量初始化生成特定类型的段。

段的名称由两部分组成—段组名称和后缀—例如 PDATA_Z。每种内存类型都有一个段组，其中每个段保存不同类别的声明数据。

段组的名称来源于内存类型和相应的关键字，例如 PDATA 和__pdata。下表总结了内存类型和相应的段组:

内存类型	段组	段内存类型	地址范围
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

表 14:对应内存组的内存类型

一些声明的数据被放置在非易失性内存中，例如 ROM，而一些数据被放置在 RAM 中。
因此，了解每个段的 XLINK 段内存类型也很重要。有关段内存类型的更多信息，请参阅段内存类型，第 120 页。

页码:124

这个表总结了不同的后缀,它们是哪个 XLINK 段内存类型,以及它们表示的声明数据的类别:

声明数据的类别	段组	后缀
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

表 15：节段名称后缀

关于每个段的更多信息，参见段参考章节。

初始化过程

数据的初始化由系统启动代码处理。如果添加了更多的段,则必须相应地更新系统启动代码。
要配置变量的初始化，必须考虑以下问题:
应该是零初始化的段只应该放在 RAM 中。
应该初始化的段，除了初始化为零的段:

系统启动代码通过将 ROM 块复制到变量在 RAM 中的位置来初始化非零变量。这意味着带有后缀 ID 的 ROM 段中的数据被复制到相应的 I 段中。

当两个段都被放置在连续内存中时，这是有效的。然而，如果其中一个片段被分成更小的片段，这是非常重要的:

●另一个部分以完全相同的方式划分
●读写代表序列中间隙的内存是合法的。
●包含常数的段不需要初始化；它们只能放在闪存/ROM 中
●保存__no_init 声明变量的段不应初始化。
●最后调用全局 C++对象构造函数。
有关如何配置初始化的更多信息和示例，请参见第 127 页的链接注意事项。

页码:126

链接您的应用程序

●链接注意事项
●检查模块一致性
====

链接注意事项

在 IAR Embedded Workbench IDE 中设置项目时，会根据您的项目设置自动使用默认链接器配置文件，您可以简单地链接您的应用程序。 对于大多数项目，配置您在 Project>Options>Linker>Config 中找到的重要参数就足够了。

当您从命令行构建时，可以使用随产品包提供的现成链接器命令文件。

config 目录包含所有受支持设备的现成链接器配置文件（文件扩展名为 xcl）。这些文件包含 XLINK 所需的信息，可以按原样使用。
您通常必须对提供的配置文件进行的唯一更改（如果有的话）是对其进行自定义，使其适合目标系统内存映射。
例如，如果您的应用程序使用额外的外部 RAM，您还必须添加有关外部 RAM 内存区域的详细信息。
IDE 使用以此模式命名的特定于设备的链接器配置文件：
lnk51ew_device.xcl，其中 device 是设备的名称。
还有一组基本的链接器配置文件要在命令行上使用：

- lnk51.xcl 是一个基本的默认链接器配置文件
- lnk51b.xcl 支持分页代码模型
- lnk51e.xcl 支持 extended1 核心
- lnk51e2.xcl 支持 extended2 核心
- lnk51eb.xcl 支持 extended1 核心和 banked 代码模型
- lnk51o.xcl 支持覆盖调用约定。

这些文件包括通用设备的特定于设备的链接器配置文件。
通常，您不需要自定义此包含文件。

要编辑链接器配置文件，请使用 IDE 中的编辑器或任何其他合适的编辑器。
不要更改原始模板文件。我们建议您在工作目录中制作一个副本，然后修改该副本。

如果您发现默认的链接器配置文件不符合您的要求，您可能需要考虑：

- 放置段
- 放置数据
- 设置堆栈内存
- 设置堆内存
- 放置代码
- 保留模块
- 保留符号和段
- 应用程序启动
- XLINK 与您的应用程序之间的交互
- 制作 UBROF 以外的其他输出格式

放置段

段在内存中的放置由 IAR XLINK 链接器执行。

本节介绍最常用的链接器命令以及如何自定义链接器配置文件以适应目标系统的内存布局。

在演示方法时，使用了虚构示例。

在演示这些方法时，使用了基于此内存布局的虚构示例：

- 有 1 MB 可寻址内存。
- 地址范围 0x0000-0x1FFF、0x3000-0x4FFF、0x10000-0x1FFFF 有 ROM 存储器。
- 在地址范围 0x8000 – 0xAFFF、0xD000 – 0xFFFF 和 0x20000 – 0x27FFF 中有 RAM 存储器。
- 数据有两种寻址方式，一种用于 pdata 内存，一种用于 xdata 内存。
- 一栈一堆。
- 代码有两种寻址方式，一种为 near_func 内存，一种为 far_func 内存。

注意：即使您有不同的内存映射，例如，如果您有额外的内存空间 (EEPROM) 和额外的段，您仍然可以使用以下示例中描述的方法。

ROM 可用于存储 CONST 和 CODE 段存储器类型。 RAM 存储器可以包含 DATA 类型的段。

自定义链接器配置文件的主要目的是验证您的应用程序代码和数据不会跨越内存范围边界，

否则会导致应用程序失败。

对于链接后每个放置指令的结果，检查列表文件中的段映射（使用命令行选项 -x 创建）。

放置段的一般提示

当您考虑应该将段放置在内存中的什么位置时，通常最好先开始放置大段，然后再放置小段。此外，您应该考虑以下方面：

- 开始放置必须放在特定地址的段。例如，保存复位向量的段通常就是这种情况。
- 然后考虑放置包含需要连续内存地址的内容的段，例如用于堆栈和堆的段。
- 放置不同寻址模式的代码段和数据段时，请确保按大小顺序放置段（最小的内存类型在前）。

注意：在链接器将任何段放入内存之前，链接器将首先放置绝对段。

使用 -Z 命令进行顺序放置

当您必须将一个段保持在一个连续的块中时，当您必须在一个段中保留段部分的顺序时，或者当您必须按特定顺序放置段时，使用 -Z 命令的可能性更小。

下面说明如何使用 -Z 命令将段 MYSEGMENTA 后跟段 MYSEGMENTB 放入 CONST 内存（即 ROM）中，内存范围为 0x0000-0x1FFF。

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

要将两个不同类型的段连续放置在同一内存区域中，请不要为第二个段指定范围。在以下示例中，MYSEGMENTA 段首先位于内存中。

然后，MYCODE 可以使用剩余的内存范围。

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

两个内存范围可以重叠。这允许具有不同放置要求的段共享部分内存空间；例如：

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

通常，当你用-Z 选项顺序放置段时，每个段都被完全放置在你所指定的一个地址范围内。如果你使用修饰符 SPLIT-，段的每一部分都会按顺序单独放置，允许不同段部分之间有地址间隙，例如。

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

在大多数情况下，使用打包段放置（-P）会更好。唯一使用-Z(SPLIT-type)比较好的情况是当段的开始和结束地址很重要时，例如，当整个段在程序启动时必须初始化为零，并且单个段的部分不能被放置在任意的地址。

即使不是严格要求，也要确保总是指定每个内存范围的结束。如果你这样做了，IAR XLINK 链接器就会在你的段不能容纳在可用的内存中时提醒你。

使用-P 命令进行打包放置

-P 命令与-Z 不同，它不一定按顺序放置段（或段部分）。使用-P，可以将片段部分放入先前放置时留下的孔中。

下面的例子说明了如何使用 XLINK -P 选项来有效地利用内存区域。这个命令将把数据段 MYDATA 放在 DATA 内存（也就是 RAM 中）的一个虚构的内存区域中。

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

如果你的应用程序在 0x6000-0x67FF 的内存范围内有一个额外的 RAM 区域，你可以简单地将其添加到原始定义中。

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

然后链接器可以将 MYDATA 段的某些部分放在第一个范围内，某些部分放在第二个范围内。如果你使用了-Z 命令，链接器将不得不把所有段的部分放在同一范围内。

注意：复制初始化段-BASENAME_I 和 BASENAME_ID 以及动态初始化段必须使用-Z 来放置。

页码:130

使用-P 命令进行库位放置

-P 命令对分页段放置很有用，也就是说，代码应该被分成几个不同的内存分页。例如，如果你的分页代码使用 ROM 内存区域 0x8000-0x9FFF，链接器指令将看起来像这样：

```
//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
```

这个例子把这个段分成四个段，它们分别位于这些地址:

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

有关这些符号以及如何为分页系统配置的详细信息，请参阅为分页模式设置链接器，第 109 页。

放置数据

静态内存是包含全局变量或声明为静态变量的内存。

放置静态内存数据段

根据它们的内存属性，静态数据被放置在特定的段中。 有关编译器使用的段的信息，请参见静态数据存储器段，第 124 页。

例如，这些命令可用于放置静态数据段：

```
/* 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
```

所有的数据段都被放置在片上 RAM 使用的区域。

放置定位数据

例如,通过使用#pragma location 指令或@运算符显式放置在地址上的变量被放置在 PDATA\u AC 或 PDATA\u an 段中。前者用于常量初始化数据，后者用于声明为\uu no\u init 的项。段的单个段部分知道其在内存空间中的位置，并且不必在链接器配置文件中指定。

放置用户定义的线段

如果使用#pragma location 指令或@运算符创建自己的段，则还必须在链接器配置文件中使用-Z 或-P 段控制指令定义这些段。

设置堆栈内存

在本例中，用于保存堆栈的数据段称为 CSTACK。系统启动代码初始化堆栈指针，使其指向堆栈段的开始或结束，具体取决于堆栈类型。

使用命令行界面时，为堆栈分配内存区域的执行方式与使用 IDE 时不同。

有关堆栈内存的更多信息，请参阅 8051 编译器使用的堆栈，第 90 页和堆栈注意事项，第 247 页。

IDE 中的堆栈大小分配

选择 Project>Options。在 General Options 类别中，单击 Stack/Heap 选项卡。

在专用文本框中添加所需的堆栈大小。

从命令行分配堆栈大小

堆栈段的大小在链接器配置文件中定义。

链接器配置文件在文件的开头设置一个常量,表示堆栈的大小。为应用程序指定适当的大小：
-D_*stackname*_SIZE=*size*
其中 stackname 可以是 IDATA_STACK、XDATA_STACK、PDATA_STACK、或 extended_stack。注意，大小是按十六进制写的，但不一定要用 0x 符号。

在 IAR Embedded Workbench 提供的许多链接器配置文件中，这一行的前缀是注释字符//，因为 IDE 控制堆栈大小。前缀为注释字符//，因为 IDE 控制堆栈大小的 分配。要使该指令生效，请删除注释字符。

放置堆栈段

在链接器配置文件的下方，实际的堆栈段在堆栈可用的内存区域中定义：
-Z(IDATA)ISTACK+_IDATA_STACK_SIZE=*start-end*
-Z(XDATA)XSTACK+_XDATA_STACK_SIZE=*start-end*
注意：
- 此范围不指定堆栈的大小；它指定可用内存的范围。
- = 在内存区域的开头分配堆栈段。

idata 数据堆栈
idata 堆栈由硬件寄存器 SP 指向。堆栈向更高地址增长，cstartup 将 SP 初始化到 idata 堆栈段的开头。
注意： idata 堆栈和扩展堆栈不能同时存在。
例子
-Z(IDATA)ISTACK+_IDATA_STACK_SIZE=*start-end*

扩展堆栈
在某些设备上，您可以使用选项 --extended_stack 来选择扩展的
堆栈在 xdata 内存中，而不是 idata 堆栈中。扩展堆栈由寄存器对?ESP:SP 指向。?ESP 寄存器在链接器命令文件中定义。堆栈向更高地址增长，cstartup 初始化寄存器对
?ESP:SP 到扩展堆栈段的开头。
注意：扩展堆栈不能与 idata 堆栈或 xdata 堆栈同时存在。
但是，可以同时使用扩展堆栈和 pdata 堆栈。
例子
-Z(XDATA)EXT_STACK+_EXTENDED_STACK_SIZE=*start-end*

Pdata 堆栈

pdata 堆栈由一个 8 位模拟的堆栈指针 PSP 指向，堆栈向低地址增长。pdata 堆栈必须位于 xdata 内存的 pdata 范围内。cstartup 模块将 PSP 初始化为堆栈段的末端。
注意：pdata 堆栈可以与所有其他类型的堆栈并行存在。
例子
-Z(XDATA)PSTACK+_PDATA_STACK_SIZE=*start-end*
pdata 堆栈指针本身就是一个段，必须位于数据存储器中。

例子
-Z(DATA)PSP=08-7F

扩展数据堆栈

扩展数据堆栈由一个 16 位模拟堆栈指针 XSP 指向，堆栈向较低的地址增长。扩展数据堆栈必须位于扩展数据内存中。cstartup 模块将 XSP 初始化到堆栈段的末尾。

注意：扩展数据堆栈和扩展堆栈不能同时存在。

例子
-Z(XDATA)XSTACK+_XDATA_STACK_SIZE=*start-end*
xdata 堆栈指针本身是一个段，必须位于数据内存中。

例子
-Z(DATA)XSP=08-7F

总结
此表总结了不同的堆栈：

堆栈	最大尺寸	调用约定	描述
Idata	256 bytes	Idata 可重入硬件堆栈	向更高的内存增长。
Pdata	256 bytes	Pdata 可重入仿真堆栈	向更低的内存增长。
Xdata	64 Kbytes	Xdata 可重入仿真堆栈	向更低的内存增长。
Extended	64 Kbytes	扩展堆栈可重入硬件堆栈	向更高的内存增长。 仅适用于在外部数据内存空间中具有堆栈的设备。

表 16：堆栈总结

页码:134

堆栈以不同的方式使用。 idata 堆栈始终用于寄存器溢出（或扩展堆栈，如果您使用 --extended_stack 选项）。
当您使用 idata 或扩展堆栈可重入调用约定时，它还用于参数、局部变量和返回地址。支持 xdata 内存的设备可以使用 pdata 可重入或 xdata 可重入调用约定将函数参数、局部变量和返回地址存储在 pdata 或 xdata 堆栈上。

设置堆内存

堆包含由 C 函数 malloc（或相应的函数）或 C++ 运算符 new 分配的动态数据。只有带有外部数据存储器的 8051 设备可以有一个堆。
如果您的应用程序使用动态内存分配，您应该熟悉：

- 用于堆的链接器段，请参见堆上的动态内存，第 90 页
- 分配堆大小的步骤，根据您使用的构建接口而有所不同
- 将堆段放入内存的步骤。

另请参见堆注意事项，第 247 页。
在此示例中，用于保存堆的数据段称为 堆。

IDE 中的堆大小分配

选择 Project>Options。在常规选项类别中，单击 Stack/Heap 选项卡。
在专用文本框中添加所需的堆大小。

从命令行分配堆大小

HEAP 段的大小在链接器配置文件中定义。
链接器配置文件在文件开头设置一个常量,表示堆的大小。为您的应用程序指定适当的大小，在本例中为 1024 字节：

```
-D_HEAP_SIZE=400 /* 1024 bytes 堆内存 */
```

请注意，大小是以十六进制写的，但不一定用 0x 符号。

在 IAR Embedded Workbench 提供的许多链接器配置文件中，这些行的前缀是注释字符//，因为 IDE 控制堆的大小分配。要使该指令生效，请删除注释字符。
如果你使用一个堆，你应该至少为它分配 512 字节，以使它正常工作。

放置堆段
实际的堆段分配在可用于堆的内存区域中：
-Z(DATA)HEAP+_HEAP_SIZE=8000-AFFF

注意：这个范围并不指定堆的大小，它指定的是可用内存的范围。

放置代码
本节包含用于存储代码和中断向量表的段的描述。关于所有段的信息，请参见段的概要，第 405 页。

启动代码
在本例中，段 CSTART 包含了系统启动和终止时使用的代码，参见系统启动和终止，第 164 页。系统启动代码
应该放在芯片复位后开始执行代码的位置。段落部分也必须放在一个连续的内存空间中，这这一行将把 CSTART 段放在地址 0x1100 处。
-Z(CODE)CSTART=1100

普通代码
没有内存类型属性声明的函数被放置在不同的段中，具体取决于您使用的代码模型。 这些段是：NEAR_CODE、BANKED_CODE 和 FAR_CODE。
有关包含普通代码的段的信息，请参阅“段参考”一章。

近代码
Near 代码（即，当您使用 Near 代码模型时，所有用户编写的代码，或使用内存属性 __near_func 显式键入的函数）都放置在 NEAR_CODE 段中。

页码:136

在链接器命令文件中，它可能如下所示：
-Z(CODE)NEAR_CODE=_CODE_START-_CODE_END

分页代码

使用分页代码模型时，所有用户编写的代码都位于 BANKED_CODE 段。在这里，-P 链接器指令用于允许 XLINK 拆分段并更有效地打包其内容。
这在这里很有用，因为内存范围是不连续的。
在链接器命令文件中，它可能如下所示：
-P(CODE)BANKED_CODE=[_CODEBANK_START-_CODEBANK_END]*4+10000
这里声明了四个代码库。例如，如果 _CODEBANK_START 为 4000，_CODEBANK_END 为 7FFF，则将创建以下分页：0x4000-0x7FFF、0x14000-0x17FFF、0x24000-0x27FFF、0x34000-0x37FFF。

远代码

远代码--也就是说，当你使用远代码模型时，所有用户编写的代码，或者用内存属性 __far_func 明确输入的函数--都放在 FAR_CODE 段中。
在链接器的配置文件中，它看起来像这样。
-Z(CODE)FAR_CODE=_CODE_START-_CODE_END
-P(CODE)FAR_CODE=[_FAR_CODE_START-_FAR_CODE_END]/10000

中断向量

中断向量表包含指向中断例程（包括重置例程）的指针。在此示例中，该表放置在段 INTVEC 中。然后，链接器指令将如下所示：
-Z(CODE)INTVEC=0

关于中断向量的更多信息，请参见中断向量和中断向量表，第 98 页。

C++动态初始化

在 C++中，所有的全局对象都是在主函数被调用之前创建的。对象的创建可以涉及构造函数的执行。
DIFUNCT 段包含一个指向初始化代码的地址向量。
当系统被初始化时，向量中的所有条目都被调用。

比如说：
-Z(CONST)DIFUNCT=0000-1FFF,3000-4FFF
DIFUNCT 必须使用-Z 参数。欲了解更多信息，请参见 DIFUNCT，第 416 页。

保存模块

如果一个模块作为一个程序模块被链接，它总是被保留。也就是说，它将对链接的应用程序作出贡献。然而，如果一个模块作为一个库模块被链接，那么只有当它被应用程序的其他部分以符号方式引用时，它才会被包括在内。这是真的，即使该库模块包含一个根符号。为了保证这样的库模块总是被包括在内，使用-A 来使文件中的所有模块被当作程序模块来处理。
-A file.r51
使用 -C 使文件中的所有模块都被视为库模块：
-C file.r51

保留符号和部分

默认情况下，XLINK 会删除应用程序不需要的任何段、段部分和全局符号。要保留似乎不需要的符号(或者实际上是定义它的段部分),您可以在 C/C++ 源代码中的符号上使用 __root 属性或在汇编器源代码中使用 ROOT，或者使用 XLINK 选项 -g。
有关包含和排除的符号和段部分的信息，请检查映射文件（使用 XLINK 选项 -xm 创建）。
有关保留符号和节的链接过程的更多信息，请参阅详细的链接过程，第 121 页。

应用程序启动

默认情况下，应用程序开始执行的点由 __program_start 标签定义，该标签被定义为指向代码的开头。标签也通过调试器信息传递给任何调试器。
要将应用程序的起点更改为另一个标签，请使用 XLINK 选项 -s。

XLINK 与您的应用程序之间的交互

使用 XLINK 选项 -D 定义可用于控制应用程序的符号。您还可以使用符号来表示链接器配置文件中定义的连续内存区域的开始和结束。

要将一个符号的引用更改为另一个符号，请使用 XLINK 命令行选项 -e。 这很有用，例如，将引用从未实现的函数重定向到存根函数，或者选择某个函数的几个不同实现之一。
有关所有全局（静态链接）符号的地址和大小的信息，请检查映射文件中的条目列表（XLINK 选项 -xm）。

生成 UBROF 以外的其他输出格式
除了 C-SPY 调试器使用的专有格式 UBROF 之外，XLINK 还可以生成 30 多种行业标准的加载器格式。 如需完整列表，
请参阅 IAR 链接器和库工具参考指南。 要指定不同于默认的输出格式，请使用 XLINK 选项 -F。 例如：
-F intel-standard
注意，使用 XLINK -O 选项产生两个输出文件可能很有用，一个用于调试，一个用于刻录到 ROM/flash。

还要注意的是，如果你选择使用-r 选项启用某些低级 I/O 功能的调试支持，如主机上的文件操作等机制，这种调试支持可能会对你的应用程序的性能和响应性产生影响。在这种情况下，由于所包含的调试模块，调试构建将与你的发布构建不同。

验证代码和数据放置的链接结果
链接器有几个功能可以帮助你管理代码和数据的放置，例如，链接时的信息和链接器映射文件。
段太长错误和范围错误
放在可重定位段中的代码或数据将在链接时解决其绝对地址。注意，直到链接时才知道是否所有的段都适合保留的内存范围。如果一个段的内容不适合链接器配置文件中定义的地址范围，XLINK 将发出一个段太长的错误。
有些指令不工作，除非在链接后某个条件成立，例如，分支必须在某个距离内，或者地址必须是偶数。
XLINK 在链接文件时，会验证这些条件是否成立。如果某个条件不满足，XLINK 会产生一个范围错误或警告，并打印出错误的描述。
关于这些类型的错误的更多信息，请参阅《IAR 链接器和库工具参考指南》。

页码:139

链接器映射文件

XLINK 可以生成广泛的交叉引用列表，其中可以选择包含以下信息：

- 按地址顺序列出所有段的段映射
- 列出程序中每个模块的所有段、本地符号和条目（公共符号）的模块图。还可以列出输出中未包含的所有符号
- 模块摘要，列出每个模块的贡献（以字节为单位）
- 包含每个模块中的每个条目（全局符号）的符号列表。

使用 IDE 中的选项生成链接器列表，或命令行中的选项-x 及其子选项之一生成链接器列表。

通常，如果链接过程中发生任何错误，例如范围错误，XLINK 将不会生成输出文件。即使遇到非致命错误，也可以在 IDE 中使用选项 Always generate output，或在命令行中使用选项-B 来生成输出文件。

有关列表选项和链接器列表的更多信息，请参阅 IAR 链接器和库工具参考指南，以及 8051 的 IDE 项目管理和构建指南。

管理多个内存空间

不支持多个内存空间的输出格式（如 MOTOROLA 和 INTEL HEX）可能需要每个内存空间最多一个输出文件。如果您只向一个内存空间(闪存)生成输出，但如果您还将对象放置在 EEPROM 或数据空间中的外部 ROM 中，则输出格式不能表示这一点，链接器会发出以下错误消息：
错误【e133】：输出格式无法处理多个地址空间。使用格式变量（-y-O）指定所需的地址空间。
(Error[e133]: The output format *Format* cannot handle multiple address spaces. Use format variants (-y -O) to specify which address space is wanted.)

要将输出限制为闪存，请为您使用的衍生工具和内存模型制作链接器配置文件的副本，并将其放在项目目录中。将此行添加到文件：

```
-y(CODE)
```

要为其他内存空间生成输出，必须为每个内存空间生成一个输出文件（因为您选择的输出格式不支持多个内存空间）。为此，请使用 XLINK 选项-O。

对于每个附加输出文件，必须指定格式、XLINK 段类型和文件名。例如：

-Omotorola,(XDATA)=external_rom.a51
-Omotorola,(CODE)=eeprom.a51
注： 作为一般规则，仅当使用非易失性内存时，才需要输出文件。
换句话说，仅当数据空间包含外部只读存储器。
有关详细信息，请参阅 IAR 链接器和库工具参考指南。

检查模块一致性

本节介绍运行时模型属性的概念，这是 IAR Systems 提供的工具使用的一种机制，用于确保链接到应用程序的模块是兼容的，换句话说，是使用兼容的设置构建的。这些工具使用一组预定义的运行时模型属性。您可以使用这些预定义属性或定义自己的属性，以确保不兼容的模块不会一起使用。
例如，在编译器中，可以编写一个只支持单独寄存器的模块。如果您编写一个支持单独 DPTR 寄存器的例程，您可以检查该例程是否未在为影子 DPTR 构建的应用程序中使用。

运行时模型属性

运行时属性是由命名键及其对应值组成的对。两个模块只有在它们定义的每个键的值相同时才能链接在一起。
有一个例外：如果一个属性的值为 *，那么该属性匹配任何值。
这样做的原因是您可以在模块中指定它以表明您已经考虑了一致性属性，这可以确保模块不依赖该属性。
例子
在此表中，目标文件可以（但不是必须）定义两个运行时属性颜色（color）和按钮（taste）：
Object file Color Taste
file1 blue not defined
file2 red not defined
file3 red *
file4 red spicy
file5 red lean
====
表 17：运行时模型属性示例

在这种情况下，file1 无法与任何其他文件链接，因为运行时属性颜色不匹配。 此外，file4 和 file5 不能链接在一起，因为按钮运行时属性不匹配。
另一方面，file2 和 file3 可以相互链接，并且可以与 file4 或 file5 链接，但不能同时与两者链接。

使用运行时模型属性
要确保模块与其他目标文件的一致性，请使用 #pragma rtmodel 指令在 C/C++ 源代码中指定运行时模型属性。 例如，如果您有一个可以在两种模式下运行的 UART，您可以指定一个运行时模型属性，例如 uart。
对于每种模式，指定一个值，例如 model 和 mode2。 在假设 UART 处于特定模式的每个模块中声明这一点。
这就是它在其中一个模块中的样子：
#pragma rtmodel="uart", "model"
或者，也可以使用 rtmodel 汇编程序指令在汇编程序源代码中指定运行时模型属性。例如：
rtmodel "uart", "model"
注意，以两个下划线开头的键名被编译器保留。关于语法的更多信息，请分别参阅 rtmodel，第 377 页和 8051 的 IAR 汇编程序用户指南。
注意：预定义的运行时属性都以两个下划线开始。你自己指定的任何属性名称都不应该在名称中包含两个首字母下划线，以消除它们与未来 IAR 运行时属性名称冲突的任何风险。
在链接时，IAR XLINK 链接器检查模块的一致性，确保具有冲突的运行时属性的模块不会被一起使用。如果检测到冲突，就会发出错误。

预定义的运行时属性

下表显示了可用于编译器的预定义运行时模型属性。这些可以包含在汇编程序代码中，或包含在 C/C++和汇编程序混合代码中。

运行时模型属性	值	说明
__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_const	ants 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.
__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.	

表 18：运行时模式属性

找到 RTMODEL 指令正确设置的最简单方法是编译 C 或 C++ 模块以生成汇编程序文件，然后检查该文件。如果您在 C 或 C++ 代码中使用汇编程序例程，请参阅 8051 的 IAR 汇编程序用户指南中的汇编程序指令一章。

例子

有关使用运行时模型属性 __rt_version 来检查与使用的调用约定相关的模块一致性的示例，请参阅使用 Normal 调用约定的提示，第 198 页。

DLIB 运行时环境

- 运行环境介绍
- 设置运行时环境
- 关于运行时环境的附加信息

DLIB 可用于 C 和 C++ 语言。另一方面，CLIB 只能与 C 语言一起使用。
有关 CLIB 的更多信息，请参阅 CLIB 运行时环境一章。

运行时环境介绍

运行时环境是应用程序执行的环境。
本节包含以下信息：

- 运行时环境功能，第 145 页
- 关于输入和输出 (I/O) 的简要说明，第 146 页
- 简要介绍 C-SPY 模拟 I/O，第 147 页
- 关于重新定位的简要说明，第 148 页

运行时环境功能

DLIB 运行时环境支持标准 C 和 C++，包括：

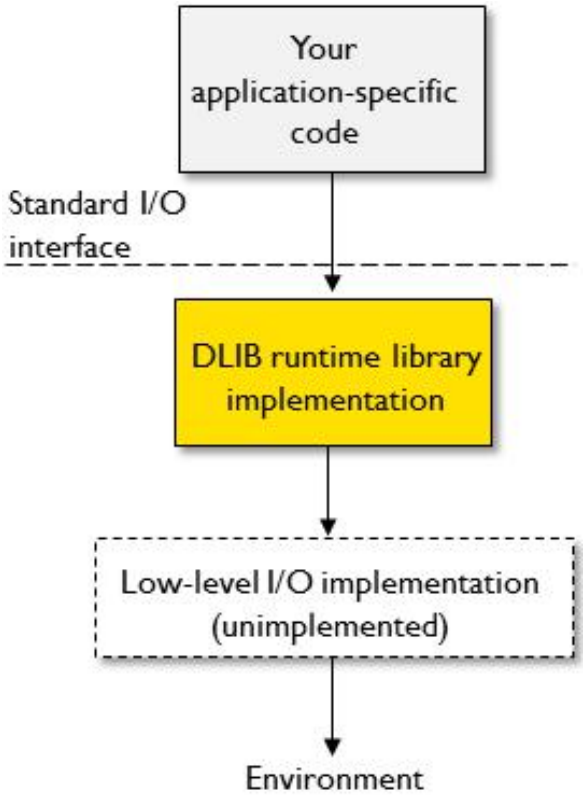
- C/C++ 标准库，包括它的接口（在系统头文件中提供）和它的实现。
- 启动和退出代码。
- 用于管理输入和输出(I/O) 的低级 I/O 接口。
- 特殊编译器支持，例如用于开关处理或整数运算的函数。
- 支持硬件特性：
- 通过内部函数直接访问低级处理器操作，例如中断屏蔽处理函数
- 包含文件中的外围单元寄存器和中断定义

运行时环境函数在一个或多个运行时库中提供。

运行时库作为预构建库和（取决于您的产品包）作为源文件提供。预构建的库以不同的配置提供以满足各种需求，请参阅运行时库配置，第 155 页。
您可以分别在产品子目录 8051\lib 和 8051\src\lib 中找到这些库。
有关该库的更多信息，请参阅 C/C++ 标准库函数一章。

简要介绍输入和输出（I/O）

每个应用程序都必须与其环境进行通信。例如，应用程序可能会在 LCD 上显示信息、从传感器读取值、从操作系统获取当前日期等。通常，您的应用程序通过 C/C++ 标准库或某些第三方库执行 I/O .
C/C++ 标准库中有许多处理 I/O 的函数，包括用于：标准字符流、文件系统访问、时间和日期、杂项系统操作以及终止和断言的函数，这组函数被称为作为标准 I/O 接口。
在台式计算机或服务器上，操作系统应通过运行时环境中的标准 I/O 接口向应用程序提供 I/O 功能。但是，在嵌入式系统中，运行时库不能假设存在这样的功能，甚至根本就没有操作系统，因此，标准 I/O 接口的低级部分默认没有完全实现：



页码:146

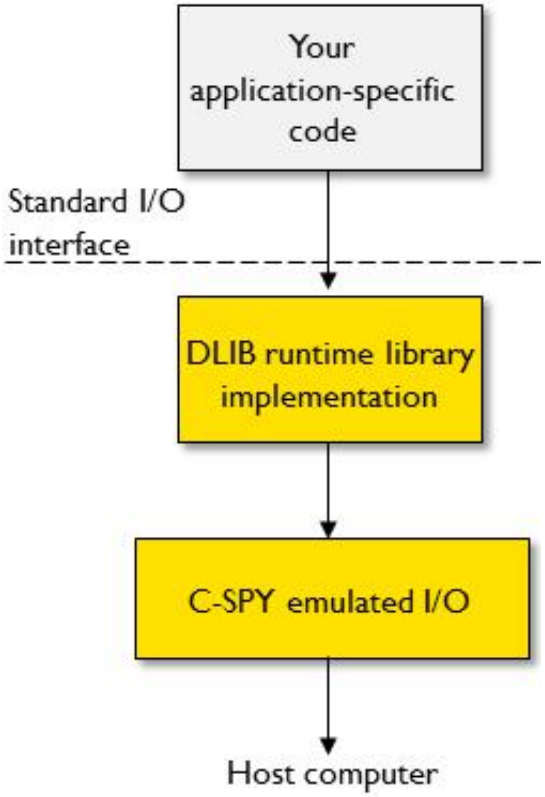
要使标准 I/O 接口正常工作，您可以：

● 让 C-SPY 调试器在主机上模拟 I/O 操作,请参阅简要介绍 C-SPY 模拟 I/O,第 147 页
● 通过提供合适的接口实现，将标准 I/O 接口重新定位到目标系统，请参阅第 148 页的关于重新定位的简要说明。

可以混合使用这两种方法。例如，您可以让 C-SPY 调试器模拟调试打印输出和断言，但实现您自己的文件系统。调试打印输出和断言在调试期间很有用，但在独立运行应用程序时不再需要（不是连接到 C-SPY 调试器）。

简要介绍 C-SPY 模拟 I/O

C-SPY 模拟 I/O 是一种机制,它允许运行时环境与 C-SPY 调试器交互以模拟主机上的 I/O 操作：



例如，当启用 C-SPY 模拟 I/O 时：

● 标准字符流被定向到 C-SPY 终端 I/O 窗口
● 文件系统操作在主机上进行
● 时间和日期功能返回主机的时间和日期
● 终止和失败断言中断执行并通知 C-SPY 调试器。

这种行为在应用程序的早期开发过程中可能很有价值，例如在实现任何闪存文件系统 I/O 驱动程序之前使用文件 I/O 的应用程序中，或者如果您需要在应用程序中调试使用 stdin 和没有用于输入和输出的实际硬件设备可用的标准输出。

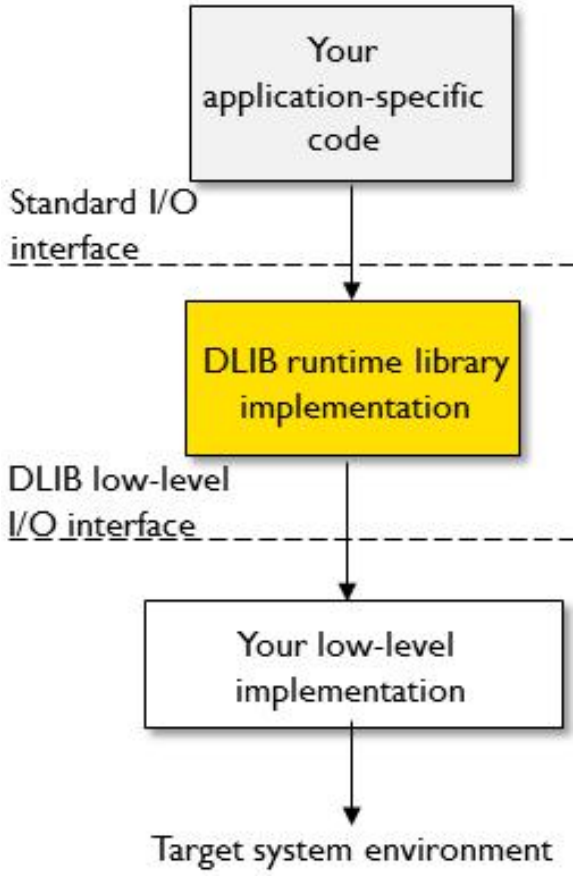
页码:147

请参阅设置运行时环境，第 149 页和 C-SPY 模拟 I/O 机制，第 161 页。

重新定位简介

重定向是调整运行时环境以便应用程序可以在目标系统上执行 I/O 操作的过程。

标准 I/O 接口大而复杂。为了使重定向更容易，DLIB 运行时环境被设计成通过一小组简单的函数执行所有 I/O 操作，称为 DLIB 低级 I/O 接口。默认情况下，低级接口中的函数缺少可用的实现。有些是未实现的，有些是存根实现，除了返回错误代码之外不执行任何操作。要重新定位标准 I/O 接口，您所要做的就是为 DLIB 低级 I/O 接口中的功能提供实现。



例如，如果您的应用程序在标准 I/O 接口中调用函数 printf 和 fputc，那么这些函数的实现都会调用低级函数 __write 来输出单个字符。为了让它们工作，你只需要提供一个 __write 函数的实现--要么自己实现，要么使用第三方实现。

有关如何使用您自己的实现覆盖库模块的信息，请参阅第 152 页的覆盖库模块。有关作为接口一部分的函数的信息，另请参阅第 168 页的 DLIB 低级 I/O 接口。

页码:148

设置运行时环境

本节包含以下任务：

- 设置运行时环境，第 149 页

在开发的初始阶段使用具有基本项目设置的运行时环境。

- 重定向—适应您的目标系统，第 151 页
- 覆盖库模块，第 152 页
- 自定义和构建您自己的运行时库，第 153 页

设置您的运行时环境

您可以根据一些基本的项目设置来设置运行时环境。让 C-SPY 调试器管理诸如标准流、文件 I/O 和各种其他系统交互之类的东西通常也很方便。在您拥有任何目标硬件之前，可以使用此基本运行时环境进行仿真。

要设置运行时环境：

1 在构建项目之前，选择 Project>Options>General Options 打开 Options 对话框。

2 在库配置页面上，验证以下设置：

- 库：选择要使用的库和库配置。通常，选择 Tiny DLIB、Normal DLIB 或 Full DLIB。

有关各种库配置的信息，请参阅运行时库配置，第 155 页。

CLIB 或 DLIB — 有关库的更多信息，请参阅 C/C++ 标准库概述，第 395 页。

3 在库选项页面上，为 Printf 格式化程序和 Scanf 格式化程序选择自动。这意味着链接器将根据来自编译器的信息自动选择适当的格式化程序。有关可用格式化程序以及如何手动选择一种的详细信息，请分别参见 printf 格式化程序，第 159 页和 scanf 格式化程序，第 160 页。

4 选择将非静态自动变量放置在堆栈上还是放置在静态覆盖区域中。

堆栈是在运行时动态分配的，而静态覆盖区域是在链接时静态分配的。请参见自动变量和参数的存储，第 83 页。

5 要启用 C-SPY 仿真 I/O,请选择 Project>Options>Linker>Output 并选择与您需要的支持级别相匹配的格式选项。 简要了解 C-SPY 模拟 I/O，第 147 页。选择：
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.
====
表 19：C-SPY 仿真 I/O 的调试信息和级别

注意：C-SPY 终端 I/O 窗口不会自动打开；您必须手动打开它。有关此窗口的更多信息，请参阅适用于 8051 的 C-SPY® 调试指南。

6 在某些系统上，终端输出可能很慢，因为主机和目标系统必须针对每个字符进行通信。因此，运行时库中包含一个名为 __write_buffered 的 __write 函数的替代品。该模块缓冲输出并一次一行地将其发送到调试器,从而加快输出速度。请注意,此函数使用大约 80 字节的 RAM 内存。
要在 IDE 中使用此功能，请选择 Project>Options>Linker>Output 并选择 Buffered terminal output 选项。
要在命令行上启用此功能，请将其添加到链接器命令行：
-e__write_buffered=__write

7 一些数学函数在不同版本中可用：默认版本、小于默认版本、更大但比默认版本更准确。考虑您应该使用哪些版本。有关详细信息，请参阅数学函数，第 161 页。

8 构建项目时，会根据您所做的项目设置自动使用合适的预构建库和库配置文件。

页码:150

有关哪些项目设置影响库文件选择的信息，请参阅运行时库配置，第 155 页。
您现在已经设置了一个运行时环境，可以在开发应用程序源代码时使用该环境。

重新定位--适应您的目标系统

在您可以在目标系统上运行应用程序之前，您必须调整运行时环境的某些部分，通常是系统初始化和 DLIB 低级 I/O 接口函数。
为您的目标系统调整您的运行时环境：
1 调整系统初始化。
您可能必须调整系统初始化，例如，您的应用程序可能需要初始化中断处理、I/O 处理、看门狗定时器等。
您可以通过实现例程 __low_level_init 来做到这一点，该例程在数据段初始化之前执行。请参见系统启动和终止，第 164 页和系统初始化，第 167 页。请注意，您可以在产品安装中提供的示例项目中找到有关此设备的特定示例；见信息中心。

2 为您的目标系统调整运行时库。要实现这些功能，您需要对 DLIB 低级 I/O 接口有一个很好的理解，请参阅第 148 页关于重定向的简要介绍。
通常，如果您的应用程序使用以下功能，您必须实现自己的函数：
● 输入和输出的标准流
如果您的应用程序使用了这些流中的任何一个，例如函数 printf 和 scanf，则必须实现低级函数 __read 和 __write 的版本。
低级函数使用唯一整数的文件句柄标识 I/O 流,例如打开的文件。通常与 stdin、stdout 和 stderr 关联的 I/O 流分别具有文件句柄 0、1 和 2。当句柄为 -1 时，应刷新所有流。流在 stdio.h 中定义。
● 文件输入输出
该库包含大量用于文件 I/O 操作的强大函数，例如 fopen、fclose、fprintf、fputs 等。所有这些函数都调用一小组低级函数，每个低级函数旨在完成一项特定任务；例如，__open 打开一个文件，__write 输出字符。实现这些低级函数的版本。

● 发出信号并加注
如果这些功能的默认实现没有提供您需要的功能，您可以实现自己的版本。
● 时间和日期
要使时间和日期函数正常工作，您必须实现函数 clock、__time32 和__getzone。
● 断言，请参见 _ReportAssert，第 174 页。
● 环境互动
如果 system 或 getenv 的默认实现没有提供您需要的功能，您可以实现自己的版本。
有关函数的更多信息，请参阅 DLIB 低级 I/O 接口，第 168 页。
您可以使用自己的版本覆盖的库文件位于 8051\src\lib 目录中。

3 当您实现了低级 I/O 接口的功能后，您必须将这些功能的版本添加到您的项目中。有关这方面的信息，请参阅
覆盖库模块，第 152 页。
注意：如果您实现了 DLIB 低级 I/O 接口函数并将其添加到您构建的支持 C-SPY 模拟 I/O 的项目中，则将使用您的低级函数而不是提供的函数使用 C-SPY 模拟 I/O。例如，如果您实现自己的 __write 版本，则输出到 C-SPY
不支持终端 I/O 窗口。请参阅简要介绍 C-SPY 模拟 I/O，第 147 页。

4 在您可以在目标系统上执行应用程序之前，您必须使用 Release 构建配置重新构建您的项目。这意味着链接器将不包含 C-SPY 模拟 I/O 机制及其提供的低级 I/O 功能。如果您的应用程序调用标准 I/O 接口的任何低级函数，要么
直接或间接，并且您的项目不包含这些，链接器将为每个缺少的低级函数发出错误。另外，请注意 NDEBUG 符号是在 Release 构建配置中定义的，这意味着将不再生成断言。有关详细信息，请参阅 _ReportAssert，第 174 页。

覆盖库模块
要覆盖库函数并将其替换为您自己的实现：
1 使用模板源文件（库源文件或其他模板）并将其副本放在您的项目目录中。
您可以使用自己的版本覆盖的库文件位于 8051\src\lib 目录中。

页码:152

2 修改文件。
注意：要覆盖模块中的函数，您必须为被覆盖模块中所有需要的符号提供替代实现。否则，您将收到有关重复定义的错误消息。

3 将修改后的文件添加到您的项目中，就像任何其他源文件一样。
注意:如果您实现了 DLIB 低级 I/O 接口函数并将其添加到您构建的支持 C-SPY 模拟 I/O 的项目中，则将使用您的低级函数而不是提供的函数使用 C-SPY 模拟 I/O。例如，如果您实现自己的 __write 版本,将不支持输出到 C-SPY 终端 I/O 窗口。请参阅简要介绍 C-SPY 模拟 I/O，第 147 页。
您现在已经完成了用您的版本覆盖库模块的过程。

自定义和构建您自己的运行时库

如果预构建的库配置不符合您的要求，您可以自定义您自己的库配置，但这需要您重新构建库。
构建自定义库是一个复杂的过程。因此，请仔细考虑是否真的有必要。在以下情况下，您必须构建自己的运行时库：
- 没有用于编译器选项或硬件支持所需组合的预构建库
- 您想定义自己的库配置，支持语言环境、文件描述符、多字节字符等。这将包括或排除 DLIB 运行时环境的某些部分。

在这些情况下，您必须：
- 确保您已安装库源代码 (scr\lib)。如果尚未安装,您可以使用 IAR License Manager 安装它，请参阅安装和许可指南。
- 建立图书馆项目
- 进行所需的库自定义
- 构建您的自定义运行时库
- 最后，确保您的应用程序项目将使用定制的运行时库。

要设置库项目：

1 在 IDE 中，选择 Project>Create New Project 并使用可用于自定义运行时环境配置的库项目模板。 Normal 库配置有一个库模板，请参阅运行时库配置，第 155 页

请注意，当您从模板创建新库项目时，新项目中包含的大部分文件都是原始安装文件。如果您要修改这些文件，请先复制它们，然后用这些副本替换项目中的原始文件。

要自定义库功能：

1 库功能由一组配置符号决定。这些符号的默认值在文件 DLib_Defaults.h 中定义，您可以在 8051\inc\c 中找到该文件。这个只读文件描述了配置的可能性。请注意，您不应修改此文件。

此外，您的自定义库有自己的库配置文件 dl8051libraryname.h（您可以在 8051\config\template\project 中找到该文件），它使用所需的库配置设置该特定库。根据应用需求，通过设置配置符号的值来自定义这个文件。例如，要在 printf 格式字符串中启用 ll 限定符，请在库配置文件中写入：

```
#define _DLIB_PRINTF_LONG_LONG 1
```

有关您可能要自定义的配置符号的信息，请参阅：

- printf 和 scanf 的配置符号，第 177 页
- 文件输入和输出的配置符号，第 178 页
- 语言环境，第 178 页
- Strtod，第 180 页

2 完成后，使用适当的项目选项构建您的库项目。

构建库后，您必须确保在应用程序项目中使用它。

要从命令行构建 IAR Embedded Workbench 项目，请使用 IAR 命令行构建实用程序 (iarbuild.exe)。但是，没有提供用于从命令行构建库的 make 或批处理文件。有关构建过程和 IAR 命令行构建实用程序的信息，请参阅 8051 的 IDE 项目管理和构建指南。

要在您的应用程序项目中使用自定义运行时库：

1 在 IDE 中，选择 Project>Options>General Options，然后单击 Library Configuration 选项卡。

2 从库下拉菜单中，选择自定义 DLIB。
3 在库文件文本框中，找到您的库文件。
4 在配置文件文本框中，找到您的库配置文件。

有关运行时环境的其他信息

本节提供有关运行时环境的其他信息：

运行时库配置

运行时库提供了不同的库配置，其中每种配置适用于不同的应用程序需求。
运行时库配置在库配置文件中定义。它包含有关哪些功能是运行时环境的一部分的信息。您在运行时环境中需要的功能越少，环境就会变得越小。
这些预定义的库配置可用：

库配置	说明
正常 DLIB（默认）	C locale，但没有 locale 接口，不支持文件描述符，printf 和 scanf 中没有多字节字符，strtod 中没有十六进制浮点数。

表 20：库的配置

注意：除了这些预定义的库配置之外，您还可以提供自己的配置，请参阅自定义和构建您自己的运行时库，第 153 页
如果您没有明确指定库配置，您将获得默认配置。如果您使用预构建的运行时库，将自动使用与运行时库文件匹配的配置文件。
请参阅设置运行时环境，第 149 页。

要覆盖默认库配置，请使用以下方法之一：
1 使用您选择的预构建配置 — 指定运行时配置：
选择项目>选项>常规选项>库配置>库并更改默认设置。
使用 --dlib_config 编译器选项，请参见 --dlib_config，第 300 页。
预建库基于默认配置，请参见运行时库配置，第 155 页。

2 如果您已经构建了自己的自定义库，请选择 Project>Options>Library Configuration>Library 并选择 Custom 以使用您自己的配置。有关详细信息，请参阅自定义和构建您自己的运行时库，第 153 页。

预建的运行时库
为这些选项的不同组合配置了预构建的运行时库：
- 核心变体
- 堆栈位置
- 数据模式
- 代码模型
- 调用约定
- 固定位置
- 数据指针数量
- 数据指针可见性
- 数据指针大小
- 数据指针选择方法。
- 库配置—Tiny、Normal 或 Full。

页码:156

从命令行选择库：
如果您从命令行构建应用程序，请进行以下设置：
● 在 XLINK 命令行上指定要使用的库文件，例如：
dl_libname.r51
● 如果不指定库配置，将使用默认配置。
但是，您可以为编译器显式指定库配置：
--dlib_config C:\...\dl_libname.h
注意：库中的所有模块都有一个以字符开头的名称？ （问号）。
您可以在子目录 8051\lib\dlib 中找到库文件,在 8051\inc\dlib 子目录中找到库配置文件。
库文件名语法
运行时库名称以这种方式构造：
{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:
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.
{*lib_config*} Specifies the library configuration:
t = Tiny
n = Normal
f = Full

PRINTF 格式

printf 函数使用名为 _Printf 的格式化程序。 完整版相当大，并提供许多嵌入式应用程序不需要的功能。 为了减少内存消耗，还提供了三个较小的替代版本。 请注意， wprintf 变体不受影响。

下表总结了不同格式化程序的功能：

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

表 21：PRINTF 格式

† NoMb 意味着没有多字节

注意：可以进一步优化这些函数，但这需要您重建库。请参阅 printf 和 scanf 的配置符号，第 177 页。

如果格式设置字符串是字符串文本,则编译器可以自动检测在直接调用 printf 时需要哪些格式设置功能。

此信息将传递到链接器，链接器将来自所有模块的信息组合在一起，以便为应用程序选择合适的格式化程序。但是，如果格式设置字符串是变量，或者如果调用是通过函数指针间接进行的，则编译器无法执行分析，从而强制链接器选择完整格式化程序。

在这种情况下，您可能需要覆盖自动选择的 printf 格式化程序。

要覆盖 IDE 中自动选择的 printf 格式化程序，请执行以下操作：

1 选择“Project>Options>General Options ”以打开“Option”对话框。

2 在“ Library Options”页上，选择适当的格式化程序。

要从命令行覆盖自动选择的 printf 格式化程序，请执行以下操作：
1 在您正在使用的链接器配置文件中添加以下行之一：
-e_PrintfFull=_Printf
-e_PrintfFullNoMb=_Printf
-e_PrintfLarge=_Printf
-e_PrintfLargeNoMb=_Printf
_e_PrintfSmall=_Printf
-e_PrintfSmallNoMb=_Printf
-e_PrintfTiny=_Printf
-e_PrintfTinyNoMb=_Printf

用于 SCANF 的格式化程序

与 printf 函数类似，scanf 使用一种称为 _Scanf 的通用格式化程序。完整版相当大，并且提供了许多嵌入式应用程序中不需要的功能。为了减少内存消耗，还提供了两个较小的替代版本。请注意，wscanf 版本不受影响。
下表总结了不同格式化程序的功能：

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

表 22：scanf 的格式设置
† NoMb 意味着没有多字节。

注意：有可能进一步优化这些函数，但这需要你重建库。参见 printf 和 scanf 的配置符号，第 177 页。
如果格式化字符串是一个字符串字面，编译器可以自动检测在直接调用 scanf 时需要哪些格式化功能。

这个信息被传递给链接器，链接器结合所有模块的信息，为应用程序选择一个合适的格式化器。但是，如果格式化字符串是一个变量，或者通过一个函数指针间接调用，编译器就不能进行分析，迫使链接器选择完整的格式化器。
在这种情况下，你可能想覆盖自动选择的 scanf 格式化。

页码:160

要在 IDE 中手动指定 scanf 格式器

1 选择 Project>Options>General Options，打开 Options 对话框。

2 在 "库选项 "页面，选择适当的格式。

要从命令行中手动指定 scanf 格式。

1 在你使用的链接器配置文件中添加以下一行。

-e_ScanfFull=_Scanf

-e_ScanfFullNoMb=_Scanf

-e_ScanfLarge=_Scanf

-e_ScanfLargeNoMb=_Scanf

_e_ScanfSmall=_Scanf

_e_ScanfSmallNoMb=_Scanf

C-SPY 模拟 I/O 机制

C-SPY 模拟 I/O 机制的工作原理如下：

1 调试器将检测函数__DebugBreak 是否存在，如果将其与 C-SPY 模拟 I/O 的链接器选项链接在一起，则该函数将成为应用程序的一部分。

2 在这种情况下，调试器将自动在__DebugBreak 函数处设置断点。

3 当应用程序调用 DLIB 低级 I/O 接口中的函数（例如，打开）时，将调用__DebugBreak 函数，这将导致应用程序在断点处停止并执行必要的服务。

4 然后恢复执行。

另请参见简要介绍 C-SPY 模拟 I/O，第 147 页。

数学函数

一些 C/C++ 标准库数学函数有不同的版本：

- 默认版本
- 较小的版本（但精度较低）
- 更准确的版本（但更大）

较小版本

函数 cos、exp、log、log2、log10、__iar_Log（log、log2 和 log10 的帮助函数）、pow、sin、tan 和 __iar_Sin（sin 和 cos 的帮助函数）存在于库中的其他较小版本中。它们比默认版本小约 20%，快约 20%。这些函数处理 INF 和 NaN 值。缺点是它们几乎总是会丢失一些精度，并且它们没有与默认版本相同的输入范围。
函数的名称构造如下：
__iar_xxx_small<f|l>
其中 f 用于浮点变量，l 用于长双变体，没有后缀用于双变体。

从命令行指定各个数学函数：
1 使用以下选项在链接时将默认函数名称重定向到这些名称：

```
-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
```

注意，如果你想重定向任何一个函数 sin，cos，或 __iar_Sin，你必须重定向所有三个函数。
注意，如果你想重定向任何一个函数 log，log2，log10，或__iar_Log，你必须重定向所有四个函数。

更精确的版本
函数 cos，pow，sin，和 tan，以及帮助函数 __iar_Sin 和 __iar_Pow 在库中有更精确的版本，可以处理更大的参数范围。缺点是它们比默认版本更大、更慢。
这些函数的名称是这样构成的。
__iar_xxx_accurate<f|l>，其中 f 用于浮点运算。
其中 f 用于浮点数变体，l 用于长双数变体，没有后缀用于双数变体。
要从命令行中指定单个数学函数。
l 在链接时，使用这些选项，将默认的函数名称重定向到这些名称。

```
-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
```

注意，如果要重定向任何函数 sin、cos 或 __iar_Sin，则必须重定向所有三个函数。
注意，如果要重定向任何函数 pow 或 __iar_Pow，则必须重定向这两个函数。

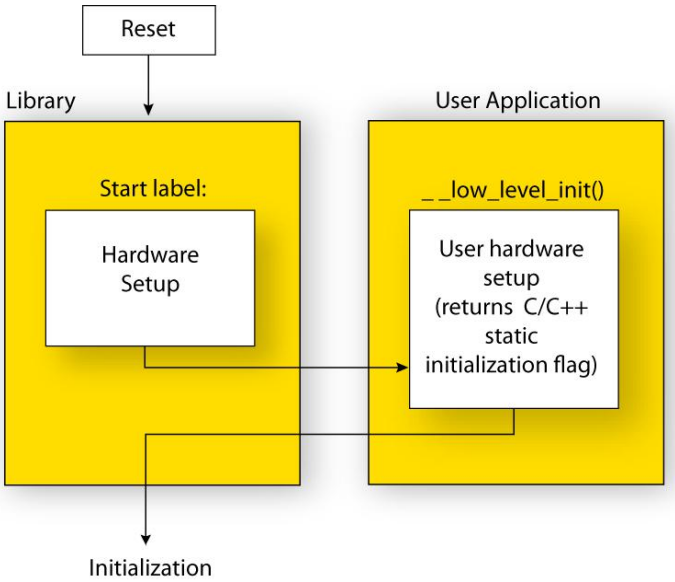
系统启动和终止

本节介绍在应用程序启动和终止期间执行的运行时环境操作。
处理启动和终止的代码位于 8051\src\lib 目录下的源文件 cstartup.s51、cmain.s51、cexit.s51 和 low_level_init.c 中。
有关如何自定义系统启动代码的信息，请参阅系统初始化，第 167 页。

系统启动

在系统启动过程中，在进入主函数之前会执行一个初始化序列。 此序列执行目标硬件和 C/C++ 环境所需的初始化。

对于硬件初始化，它看起来像这样：



● 当 CPU 复位时，它将在程序入口标签处开始执行 __program_start 在系统启动代码中。

● 寄存器组切换寄存器初始化为符号指定的编号 ?REGISTER_BANK 在链接器配置文件中。

● 如果使用 idata 堆栈,堆栈指针 SP 被初始化到 ISTACK 段的开头。如果使用扩展堆栈，扩展堆栈指针 ?ESP:SP 被初始化为 EXT_STACK 段的开头。

● 如果xdata 可重入调用约定可用,则xdata 堆栈指针XSP 被初始化到XSTACK 段的末尾。

● 如果 pdata 可重入调用约定可用,则 pdata 堆栈指针 PSP 被初始化到 PSTACK 段的末尾。

● 如果使用代码库，则库寄存器初始化为零。

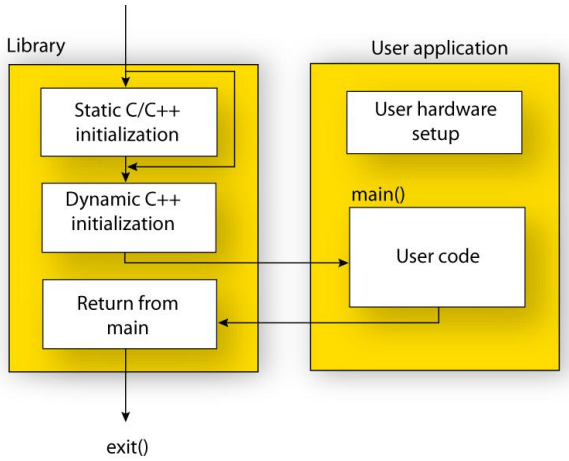
● PDATA 页面被初始化。

● 如果有多个数据指针可用，则初始化 DPTR 选择器寄存器并将第一个数据指针 (dptr0) 设置为活动数据指针。

● 如果您定义了函数__low_level_init，它就会被调用，从而使应用程序有机会执行早期初始化。

页码:165

对于 C/C ＋ ＋ 初始化，如下所示：



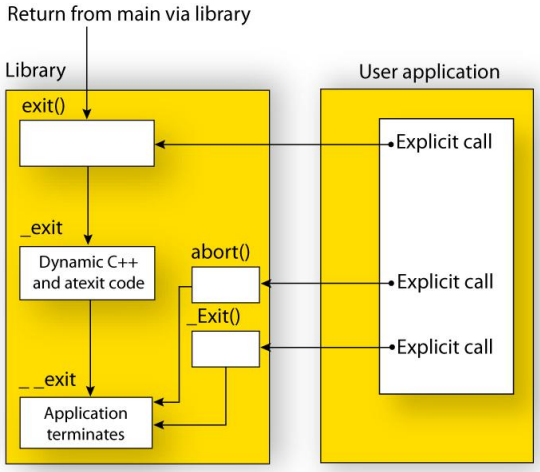
● 静态和全局变量被初始化。 即清零初始化变量，将其他初始化变量的值从 ROM 复制到 RAM 内存。 如果 __low_level_init 返回零，则跳过此步骤。 有关详细信息，请参阅系统启动时的初始化，第 123 页。

● 构造静态 C++ 对象

● 调用 main 函数，启动应用程序。

系统终止

此图显示了嵌入式应用程序可以以受控方式终止的不同方式：



应用程序可以通过两种不同的方式正常终止：

● 从主函数返回

● 调用 exit 函数

页码:166

因为 C 标准规定这两种方法应该是等价的，所以如果 main 返回，系统启动代码将调用 exit 函数。传递给 exit 函数的参数是 main 的返回值。

默认退出函数是用 C 编写的。它调用一个小的汇编函数 _exit，它将：
- 在应用程序结束时调用注册执行的函数。这包括用于静态和全局变量的 C++ 析构函数，以及使用标准函数 atexit 注册的函数。
- 关闭所有打开的文件
- 调用__exit
- 到达__exit 时，停止系统。

应用程序也可以通过调用 abort 或 _Exit 函数退出。 abort 函数只是调用 __exit 来停止系统，并且不执行任何类型的清理。
_Exit 函数等价于 abort 函数，除了 _Exit 需要一个参数来传递退出状态信息。
如果您希望您的应用程序在退出时做任何额外的事情，例如重置系统（如果使用 atexit 还不够），您可以编写自己的 __exit(int) 函数实现。
您可以使用自己的版本覆盖的库文件位于 8051\src\lib 目录中。请参见覆盖库模块，第 152 页。

对系统终止的 C-SPY 调试支持
如果您在链接期间启用了 C-SPY 模拟 I/O,则正常的 __exit 和 abort 函数将替换为特殊函数。然后，C-SPY 将识别何时调用这些函数，并可以采取适当的措施来模拟程序终止。
有关详细信息，请参阅 C-SPY 模拟 I/O 简介，第 147 页。

系统初始化
您可能需要调整系统初始化。例如，您的应用程序可能需要初始化内存映射的特殊功能寄存器 (SFR)，或者忽略由系统启动代码执行的数据段的默认初始化。
您可以通过实现您自己版本的例程 __low_level_init 来做到这一点,该例程在初始化数据段之前从文件 cmain.s 中调用。
应避免直接修改文件 cstartup.s51。
处理系统启动的代码位于 8051\src\lib 目录下的源文件 cstartup.s51 和 low_level_init.c 中。

注意：通常，您不需要自定义文件 cmain.s51 或 cexit.s51。

注意：无论您是实现自己的 __low_level_init 版本还是文件 cstartup.s51，您都不必重新构建库。

自定义 __low_level_init

产品提供了一个骨架低级初始化文件：

low_level_init.c。请注意，静态初始化变量不能在文件中使用，因为此时尚未执行变量初始化。

__low_level_init 返回的值决定了数据段是否应该由系统启动代码初始化。如果函数返回 0，则数据段不会被初始化。

修改 cstartup 文件

如前所述，如果实现您自己的 __low_level_init 版本足以满足您的需要，则不应修改文件 cstartup.s51。但是，如果您确实需要修改 cstartup.s51 文件，我们建议您按照一般程序创建文件的修改副本并将其添加到您的项目中，请参阅覆盖库模块，第 152 页。

请注意，您必须确保链接器使用您的 cstartup.s51 版本中使用的开始标签。有关如何更改链接器使用的起始标签的信息，请阅读 IAR 链接器和库工具参考指南中的 -s 选项。

DLIB 低级 I/O 接口

运行时库使用一组低级函数（称为 DLIB 低级 I/O 接口）与目标系统进行通信，大多数低级函数没有实现(执行)。

有关这方面的更多信息，请参阅简要介绍输入和输出 (I/O)，第 146 页。

下表列出了 DLIB 低级 I/O 中函数的接口：

DLIB 低级输入/输出函数接口	定义源文件
abort	abort.c
clock	clock.c
__close	close.c
__exit	xxexit.c
getenv	getenv.c
__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

表 23：DLIB 低级输入/输出函数接口

注意：您不应在应用程序中直接使用以 __为前缀的低级函数。 相反，您应该使用使用这些函数的标准库函数。 例如，要写入标准输出，您应该使用标准库函数，如 printf 或 puts，它们依次调用低级函数 __write。 如果您忘记执行低级函数并且您的应用程序将通过标准库函数调用该函数，则链接器将在您在发布构建配置中链接时发出错误。

注意：如果你在这个接口中执行了你自己的函数变体，你的变体将被使用，即使你已经启用了 C-SPY 仿真 I/O，见第 147 页关于 C-SPY 仿真 I/O 的简介。

Abort
源文件 8051/src/lib/dlib/abort.c
描述 标准 C 库函数，中止执行。
C-SPY debug action 通知应用程序已经调用 abort。
默认实现调用__exit(EXIT_FAILURE)。
参见关于重定向的简介，第 148 页
系统终止，第 166 页

clock
源文件 8051\src\lib\dlib\clock.c
描述 访问处理器时间的标准 C 库函数。
C-SPY 调试操作 返回主机上的时钟。
默认实现 返回 -1 表示处理器时间不可用。
另请参阅第 148 页的关于重新定位的简要介绍。

__close
源文件 8051\src\lib\dlib\close.c
描述 关闭文件的低级函数。
C-SPY 调试操作 关闭主机上的关联主机文件。
默认实现无。
另请参阅第 148 页的关于重新定位的简要介绍。

__exit
源文件 8051\src\lib\dlib\xxexit.c
描述 暂停执行的低级函数。
C-SPY 调试操作 通知已到达应用程序的末尾。
默认实现永远循环。
另请参阅关于重新定位的简要说明，第 148 页
系统终止，第 166 页。

getenv
源文件 8051\src\lib\dlib\getenv.c
8051\src\lib\dlib\environ.c

C-SPY 调试操作 访问主机环境。
默认实现 库中的 getenv 函数在全局变量 __environ 指向的字符串中搜索作为参数传递的键。如果找到键，则返回它的值，否则返回 0（零）。默认情况下，字符串为空。
要在字符串中创建或编辑键，您必须创建一个以 null 结尾的字符串序列，其中每个字符串具有以下格式：
键=值\0
以一个额外的空字符结束字符串（如果您使用 C 字符串，则会自动添加）。将创建的字符串序列分配给 __environ 变量。
例如：

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

如果你需要一个更复杂的环境来处理变量，你应该实现自己的 getenv，可能还有 putenv 函数。
注意：标准不要求 putenv 函数，库不提供它的实现。
另请参阅关于重新定位的简要说明，第 148 页

__getzone
源文件 8051\src\lib\dlib\getzone.c
描述 返回当前时区的低级函数。
C-SPY 调试操作 不适用。
默认实现返回“:”。
另请参阅第 148 页的关于重新定位的简要介绍。

__lseek
源文件 8051\src\lib\dlib\lseek.c
描述 用于更改打开文件中下一次访问位置的低级函数。
C-SPY 调试操作 在主机计算机上的关联主机文件中进行搜索。
默认实现无。
另请参阅第 148 页的关于重新定位的简要介绍。

__open
源文件 8051\src\lib\dlib\open.c
描述 打开文件的低级函数。
C-SPY 调试操作 打开主机上的文件。
默认实现无。
另请参阅第 148 页的关于重新定位的简要介绍。

raise
源文件 8051\src\lib\dlib\raise.c
说明 引发信号的标准 C 库函数。
C-SPY 调试操作 不适用。
默认实现 调用引发信号的信号处理程序，或以调用终止__exit(EXIT_FAILURE)。
另请参阅第 148 页的关于重新定位的简要介绍。

__read
源文件 8051\src\lib\dlib\read.c
描述 从标准输入和文件中读取字符的低级函数。
C-SPY 调试操作 将标准输入定向到终端 I/O 窗口。所有其他文件将读取关联的主机文件。
默认实现无。

示例
本示例中的代码使用内存映射 I/O 从键盘读取，其端口假定位于 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;
}
```

有关与流关联的句柄的信息，请参阅重定向 - 适应您的目标系统，第 151 页。
有关 @ 运算符的信息，请参阅控制数据和函数在内存中的放置，第 259 页。
另请参阅关于重新定位的简要说明，第 148 页

Remove
源文件 8051\src\lib\dlib\remove.c
描述 删除文件的标准 C 库函数。

C-SPY 调试操作将消息写入调试日志窗口，并返回-1。
默认实现返回 0 以表示成功，但不删除文件。
参见关于重定位，第 148 页。

rename

源文件 8051\src\lib\dlib\rename.c
描述可重命名文件的标准 C 库函数。
C-SPY 调试操作无。
默认实现返回-1，以表示失败。
参见关于重定位，第 148 页。

_ReportAssert

源文件 8051\src\lib\dlib\xreportassert.c
描述 处理失败的断言的低级函数。
D-SPY 调试操作 通知 C-SPY 调试器有关失败信息。

默认实现失败的断言由函数__ReportAssert 报告。默认情况下，它会打印一条错误消息并调用中止。
如果这不是您所需要的行为，那么您可以实现您自己的函数版本。
断言宏在头文件 asslert.h 中定义。若要关闭断言，请定义符号 NDEBUG。
在 IDE 中，符号 NDEBUG 默认是在发布项目中定义的，而不是在调试项目中定义的。
如果从命令行构建，则必须根据需要显式定义符号。参见 NDEBUG，第 393 页。参见关于重定位，第 148 页。

signal

源文件 8051\src\lib\dlib\signal.c

描述 标准的 C 库函数，改变信号处理程序。
C-SPY 调试动作 不适用。
如果环境支持某种异步信号，你可能想修改这个行为。
参见第 148 页关于重定向的简要介绍。

system
源文件 8051/src\lib\dlib\system.c
描述 执行命令的标准 C 库函数。
C-SPY 调试动作 通知 C-SPY 调试器系统已被调用，然后返回-1。
默认实现 如果你需要使用 system 函数，你必须自己实现它。
库中提供的 system 函数如果传递给它一个空指针，则返回 0，表示没有命令处理器；否则返回-1，表示失败。
如果这不是你需要的功能，你可以实现自己的版本。这并不要求你重建这个库。
另见关于重定向的简介，第 148 页。

__time32
源文件 8051\src\lib\dlib\time.c
描述 当前日历时间的低级。
C-SPY 调试操作 返回主机上的时间。
默认实现 返回 -1 表示日历时间不可用。
另请参阅第 148 页的关于重新定位的简要介绍。

__write
源文件 8051\src\lib\dlib\write.c

描述 写入标准输出、标准错误或文件的低级函数。
C-SPY 调试操作 将 stdout 和 stderr 定向到终端 I/O 窗口。 所有其他文件将写入关联的主机文件。
默认实现无。
示例 本示例中的代码使用内存映射 I/O 写入 LCD 显示器，假设其端口位于地址 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;
}
```

有关与流关联的句柄的信息，请参阅重定向 - 适应您的目标系统，第 151 页。
另请参阅第 148 页的关于重新定位的简要介绍。

PRINTF 和 SCANF 的配置符号

如果提供的格式化程序不符合您的要求（printf 格式化程序，第 159 页和 scanf 格式化程序，第 160 页），您可以自定义完整格式化程序。 请注意，这意味着您必须重新构建运行时库。

printf 和 scanf 格式化程序的默认行为由文件 DLib_Defaults.h 中的配置符号定义。这些配置符号决定了函数 printf 应该具备的能力：

Printf 配置符号	包括支持
_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

表 24：printf 配置符号的说明

当你建立一个库时，这些配置决定了函数 scanf 应该具有的功能。

Scanf 配置符号	包括支持
_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 ([*])

表 25: scanf 配置符号的描述

要自定义格式化功能，你必须：

1 根据你的应用要求，在库的配置文件中定义配置符号。

2 重新构建库，见自定义和构建你自己的运行库，第 153 页。

文件输入和输出的配置符号

只有具有完整库配置的库，见运行时库配置，第 155 页，或者在定义了配置符号 __DLIB_FILE_DESCRIPTOR 的定制库中，才支持文件输入输出。如果没有定义这个符号，就不能使用带有 FILE * 参数的函数。

要定制你的库并重建它，请参见定制和构建你自己的运行时库，第 153 页。

LOCALE

LOCALE 是 C 语言的一部分，它允许对几个领域进行语言和国家的特定设置，如货币符号、日期和时间以及多字节字符编码。

根据你所使用的库配置，你会得到不同程度的 locale 支持。然而，支持的语言越多，你的代码就会越大。因此，有必要考虑你的应用程序需要什么级别的支持。见运行时库配置，第 155 页。

DLIB 运行时库可以在两种主要模式下使用。

●有地方语言接口，这使得在运行时在不同的地方语言之间切换成为可能

●没有地区性接口，其中一个选定的地区性被硬连接到应用程序中。

预建库中的语言支持

预置库中的语言支持水平取决于库的配置。

所有的预置库只支持 C 语言。

● 所有具有完整库配置的库都有对 locale 接口的支持。对于带有 locale 接口的预建库，默认情况下只支持在运行时切换多字节的字符编码。

● 具有普通库配置的库没有对 locale 接口的支持。

如果你的应用程序需要一个不同的 locale 支持，你必须重建库。

自定义语言环境支持并重建库

如果您决定重建库，您可以在以下语言环境中进行选择：

- C 语言环境
- POSIX 语言环境
- 广泛的欧洲语言环境。

语言环境配置符号

库配置文件中定义的配置符号 _DLIB_FULL_LOCALE_SUPPORT 确定库是否支持语言环境接口。语言环境配置符号 _LOCALE_USE_LANG_REGION 和 _ENCODING_USE_ENCODING 定义了所有支持的语言环境和编码：

```
#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 */
```

有关受支持的区域环境和编码设置的列表，请参见 DLib_Defaults.h。

如果要自定义区域设置支持,则只需定义应用程序所需的区域设置配置符号。有关更多信息，请参见自定义和构建您自己的运行时库，第 153 页。

注意:如果在 C 或汇编程序源代码中使用多字节字符,请确保选择了正确的区域设置符号(本地主机区域设置）。

构建一个不支持区域设置接口的库

如果使用配置符号，则不包括区域设置接口

_DLIB_FULL_LOCALE_SUPPORT 被设置为 0（零）。这意味着使用硬连接的区域设置—默认情况下使用标准 C 区域设置—但您可以选择受支持的区域设置配置符号之一。设置区域设置函数不可用，因此不能用于在运行时更改区域设置。

构建一个支持语言区域设置接口的库

如果将配置符号_DLIB_FULL_LOCALE_SUPPORT 设置为 1，则会获得对区域设置界面的支持。默认情况下，使用标准 C 区域设置，但是您可以根据需要定义多个配置符号。因为设置语言设置函数在应用程序中可用，所以可以在运行时切换语言区域设置。

在运行时更改语言环境

标准库函数 setlocale 用于在应用程序运行时选择应用程序区域设置的适当部分。

setlocale 函数有两个参数。第一个是在模式 LC_CATEGORY 之后构造的语言环境类别。第二个参数是一个描述语言环境的字符串。 它可以是先前由 setlocale 返回的字符串，也可以是在模式之后构造的字符串：

long_REGION

或者

lang_REGION.encoding

lang 部分指定语言代码，REGION 部分指定区域限定符，encoding 指定应使用的多字节字符编码。

lang_REGION 部分匹配可以在库配置文件中指定的 _LOCALE_USE_LANG_REGION 预处理器符号。

例子

此示例设置语言环境配置:

```
setlocale (LC_ALL, "sv_FI.Utf8");
```

STROD

函数 strtod 不接受具有正常库配置的库中的十六进制浮点字符串。 要使 strtod 接受十六进制浮点字符串，您必须：

1 在库配置文件中启用配置符号 _DLIB_STRTOD_HEX_FLOAT。

2 重建库，请参见自定义和构建您自己的运行时库，第 153 页。

CLIB 运行时环境

- 使用预构建的运行时库
- 输入和输出
- 系统启动和终止
- 覆盖默认库模块
- 自定义系统初始化
- C-SPY 模拟 I/O

请注意，CLIB 运行时环境不支持任何 C99 功能。例如，不支持复数和可变长度数组。CLIB 也不支持 C++。

提供旧版 CLIB 运行时环境是为了向后兼容，不应将其用于新的应用程序项目。

使用预构建的运行时库

预构建的运行时库配置为这些的不同组合

特征：

- 核心变体
- 堆栈位置
- 代码模型
- 数据模式
- 调用约定
- 固定位置
- 数据指针数量
- 数据指针可见性
- 数据指针大小
- 数据指针选择方法

可能的组合数量很多，但并非所有组合都同样可能有用。出于这个原因，只有所有可能的运行时库的一个子集是与产品一起预构建的。还为 DLIB 库类型提供了更大的预建库变体。这些是使用数据模式 Large 或 Far 的库。如果您需要一个未交付预构建的库，您必须自己构建它，请参阅自定义和构建您自己的运行时库，第 153 页。

选择运行时库

IDE 包含基于您选择的选项的正确运行时库。有关详细信息，请参阅 8051 的 IDE 项目管理和构建指南。

指定在 XLINK 命令行上使用哪个运行时库文件，例如：

cl_libname.r51

CLIB 运行时环境包括 C 标准库。链接器将仅包含应用程序直接或间接需要的那些例程。有关运行时库的更多信息，请参阅 C/C++ 标准库函数一章。

运行时库文件名语法

运行时库名称以这种方式构造：

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

页码:182

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} 如果使用 XOR 选择方法，则指定 DPTR 选择方法和选择掩码。 在这种情况下，值是 x 后跟十六进制表示的掩码，例如 01 表示 0x01，从而产生选择字段 x01。 如果使用 INC 选择方法，则该字段的值为 inc。

输入输出

您可以自定义：

- 与基于字符的 I/O 相关的函数
- printf/sprintf 和 scanf/sscanf 使用的格式化程序。 有关如何为 8051 特定函数 printf_P 和 scanf_P 选择格式化程序的信息，请参阅 8051 特定 CLIB 函数。

基于字符的 I/O

函数 putchar 和 getchar 是基于字符的 I/O 的基本 C 函数。 对于任何可用的基于字符的 I/O，您必须使用硬件环境提供的任何工具为这两个函数提供定义。

新 I/O 例程的创建基于以下文件：

- putchar.c，作为 printf 等函数的底层部分
- getchar.c，用作 scanf 等函数的低级部分。

下面的代码例子显示了如何使用内存映射的 I/O 来写到内存映射的 I/O 设备:

```
__no_init volatile unsigned char devIO @ 8;
int putchar(int outChar)
{
devIO = outChar;
return outChar;
}
```

确切的地址是一个设计决策。例如，它可以依赖于选定的处理器变体。

有关如何在项目构建过程中包含自己的图形和图形版本的信息，请参见覆盖库模块，第 152 页。

printf 和 sprintf 使用的格式化程序

printf 和 sprintf 函数使用一个通用的格式化程序，称为_formatted_write。该格式化程序有三种变体：

_large_write

_medium_write

_small_write

默认情况下，链接器会自动为应用程序使用最合适的格式化程序。

_large_write

_large_write 格式化程序支持 C89 printf 格式指令。

_medium_write

_medium_write 格式化程序具有与_large_write 相同的格式指令，只是不支持浮点数。任何使用 %f、%g、%G、%e 或 %E 说明符的尝试都将产生运行时错误：

浮点？ 安装点了错误的格式化程序！

_medium_write 比大版本小得多。

(FLOATS? wrong formatter installed!

_medium_write is considerably smaller than the large version.)

_small_write

它只支持整数对象的%%, %d, %o, %c, %s, 和 %x 参数，而不支持字段宽度和精度参数。

_small_write 的大小是_medium_write 的 10-15%。

在 IDE 中指定 printf 格式化程序
1 选择 Project>Options 并选择常规选项类别。 单击 Library Options 选项卡。
2 选择适当的 Printf 格式化程序选项，可以是 Auto、Small、Medium 或 Large。
从命令行指定 printf 格式化程序
要显式指定和覆盖默认使用的格式化程序，请将以下行之一添加到链接器配置文件：
-e_small_write=_formatted_write
-e_medium_write=_formatted_write
-e_large_write=_formatted_write

自定义 printf
对于许多嵌入式应用程序，sprintf 不是必需的，考虑到它消耗的内存量，即使是带有 _small_write 的 printf 也提供了比合理的更多的功能。 或者，可能需要自定义输出例程来支持特定的格式化需求或非标准输出设备。
对于此类应用程序，在 intwri.c 文件中以源代码形式提供了 printf 函数的简化版本（没有 sprintf）。 该文件可以修改以满足您的要求，并将编译后的模块插入到库中以代替原始文件； 请参见覆盖库模块，第 152 页。
SCANF 和 SSCANF 使用的格式化程序
与 printf 和 sprintf 函数一样，scanf 和 sscanf 使用一个通用的格式化程序，称为 _formatted_read。 格式化程序有两种变体：
_large_read
_medium_read
默认情况下，链接器会自动为您的应用程序使用最合适的格式化程序。

_large_read
_large_read 格式化程序支持 C89 scanf 格式指令。

_medium_read
_medium_read 格式化程序具有与大版本相同的格式指令，但不支持浮点数。 _medium_read 比大版本小得多。

在 IDE 中指定 scanf 格式化程序
1 选择“Project>Options”，然后选择“General Options”类别。单击 Library Options 选项卡。
2 选择适当的 Scanf 格式化程序选项，可以是“自动”、“中”或“大”。
从命令行指定读取格式化程序
若要显式指定并重写默认使用的格式化程序，请将以下行之一添加到链接器配置文件中：
-e_medium_read=_formatted_read
-e_large_read=_formatted_read

系统启动和终止

本节介绍运行时环境在应用程序启动和终止期间执行的操作。
处理启动和终止的代码位于 8051\src\lib 目录下的源文件 cstartup.s51、cmain.s51、cexit.s51 和 low_level_init.c 中。
注意：通常，您不需要自定义文件 cmain.s51 或 cexit.s51。

系统启动

初始化应用程序时，会执行几个步骤：
● 当 CPU 复位时，将跳转到系统启动代码中的程序入口标签__program_start。
● 如果使用 idata 堆栈,堆栈指针 SP 被初始化到 ISTACK 段的开头。如果使用扩展堆栈，扩展堆栈指针 ?ESP:SP 被初始化为 EXT_STACK 段的开头。
● 如果 xdata 可重入调用约定可用,则 xdata 堆栈指针 XSP 被初始化到 XSTACK 段的末尾。
● 如果 pdata 可重入调用约定可用,则 pdata 堆栈指针 PSP 被初始化到 PSTACK 段的末尾。
● 如果使用代码库，则库寄存器初始化为零。
● 寄存器组切换寄存器初始化为符号指定的编号
?REGISTER_BANK 在链接器配置文件中。
● PDATA 页面被初始化。

● 如果有多个数据指针可用，则初始化 DPTR 选择器寄存器，并将第一个数据指针（dptr0）设置为活动数据指针。
● 如果定义了函数__low_level_init，则会调用该函数，使应用程序有机会执行早期初始化。
● 初始化静态变量；这包括清除零初始化内存和复制剩余初始化变量 RAM 内存的 ROM 映像
● 调用 main 函数，启动应用程序。
请注意，系统启动代码包含的步骤比此处描述的步骤更多。其他步骤适用于 DLIB 运行时环境。

系统终止
应用程序可以通过两种不同的方式正常终止：
● 从主功能返回
● 调用 exit 函数。
因为 C 标准规定这两个方法应该是等效的，所以如果 main 返回，cstartup 代码将调用 exit 函数。传递给 exit 函数的参数是 main 的返回值。默认的退出函数是在汇编程序中编写的。当应用程序以调试模式构建时，C-SPY 在到达特殊代码标签时停止?C_EXIT。
应用程序也可以通过调用 abort 函数退出。默认函数只调用__exit 来停止系统，而不执行任何类型的清理。

重写默认库模块
IAR CLIB 库包含您可能需要使用自己的自定义模块覆盖的模块，例如用于基于字符的 I/O，而无需重建整个库。有关如何重写默认库模块的信息，请参阅 DLIB 运行时环境一章中的重写库模块，第 152 页。
自定义系统初始化
有关如何自定义系统初始化的信息，请参阅系统初始化，第 167 页。

C-SPY 模拟 I/O

低级 I/O 接口用于被调试的应用程序和调试器本身之间的通信。接口很简单：C-SPY 会在应用程序的某些汇编标签上放置断点。当位于特殊标签的代码即将被执行时，C-SPY 将收到通知并可以执行操作。

调试器终端 I/O 窗口

当标签 ?C_PUTCHAR 和 ?C_GETCHAR 处的代码被执行时，数据将被发送到调试器窗口或从调试器窗口读取。

对于 ?C_PUTCHAR 例程，从输出流中取出一个字符并写入。

如果一切顺利，则返回字符本身，否则返回-1。

当达到标签 ?C_GETCHAR 时，C-SPY 在输入字段中返回下一个字符。如果没有给出输入，C-SPY 会一直等待，直到用户输入一些输入并按下 Return 键。

要使终端 I/O 窗口可用，应用程序必须与选择了 I/O 仿真模块的 XLINK 选项链接。请参阅 8051 的 IDE 项目管理和构建指南。

终止

调试器在到达特殊标签 ?C_EXIT 时停止执行。

（空白页）
页码:190

汇编语言接口

●混合使用 C 语言和汇编语言
●从 C 语言中调用汇编程序
●从 C++中调用汇编程序
●调用惯例
●调用函数时使用的汇编器指令
●内存访问方法
●调用框架信息

混合 C 和汇编程序

用于 8051 的 IAR C/C++编译器提供了几种访问底层资源的方法。
●完全用汇编语言编写的模块
●内在函数（C 语言的替代）。
●内联汇编器。
使用简单的内联汇编器可能是很诱人的。然而，你应该仔细选择使用哪种方法。

内在函数

编译器提供了一些预定义的函数，允许直接访问低级别的处理器操作，而不需要使用汇编语言。这些函数被称为内在函数。它们在例如时间紧迫的例程中可能非常有用。
内在函数看起来像一个普通的函数调用，但它实际上是一个编译器识别的内置函数。内在函数编译成内联代码，可以是一条指令，也可以是一个简短的指令序列。
关于可用的内在函数的更多信息，见内在函数一章。

页码:191

混合 C 和汇编模块

可以用汇编程序编写应用程序的一部分并将它们与 C 或 C++ 模块混合。

与使用内联汇编器相比，这有几个好处：

- 函数调用机制定义明确
- 代码将易于阅读
- 优化器可以使用 C 或 C++ 函数。

这会以函数调用和返回指令的形式产生一些开销，编译器会将某些寄存器视为暂存寄存器。但是，编译器还将假定所有寄存器都被内联汇编指令破坏。在许多情况下，优化器可以消除额外指令的开销。

一个重要的优势是您将在编译器生成的内容和您在汇编程序中编写的内容之间拥有一个定义良好的接口。使用内联汇编程序时，您无法保证您的内联汇编程序行不会干扰编译器生成的代码。

当应用程序部分用汇编语言编写，部分用 C 或 C++ 编写时，您将面临几个问题：

- 应如何编写汇编代码以便可以从 C 中调用？
- 汇编代码在哪里找到它的参数，返回值是如何传回给调用者的？
- 汇编代码应该如何调用 C 编写的函数？
- 用汇编语言编写的代码如何访问全局 C 变量？
- 为什么调试器在调试汇编代码时不显示调用堆栈？

第一个问题在第 194 页从 C 调用汇编程序部分讨论。以下两个在第 197 页调用约定部分讨论。

有关如何访问内存中数据的信息，请参阅内存访问方法，第 209 页。

最后一个问题的答案是，在调试器中运行汇编代码时可以显示调用堆栈。但是，调试器需要有关调用帧的信息，这些信息必须作为汇编源文件中的注释提供。

有关详细信息，请参阅呼叫框架信息，第 211 页。

混合 C 或 C++ 和汇编程序模块的推荐方法分别在第 194 页从 C 调用汇编程序例程和从 C++ 调用汇编程序例程第 196 页中描述

页码:192

内联汇编器

内联汇编器可用于将汇编器指令直接插入 C 或 C++ 函数。

asm 扩展关键字及其别名__asm 都插入汇编指令。

但是，在编译 C 源代码时，使用 --strict 选项时 asm 关键字不可用。 __asm 关键字始终可用。

注意：并非所有的汇编指令或运算符都可以使用这些关键字插入。

语法是：

```
asm ("string");
```

但不是评论。 您可以编写几个连续的内联汇编指令，例如：

```
asm("label:nop\n"
"sjmp label");
```

其中 \n （新行）分隔每个新的汇编指令。 请注意，您可以在内联汇编程序指令中定义和使用局部标号，其中标号必须与对该标号的所有引用放在同一个 asm() 中。

以下示例演示了 asm 关键字的使用。 此示例还显示了使用内联汇编程序的风险。

```
__no_init __bit bool flag;
void Foo(void)
{
while (!flag)
{
asm("MOV C,0x98.0"); /* SCON.R1 */
asm("MOV flag,C");
}
}
```

在此示例中，编译器没有注意到标志的赋值，这意味着不能期望周围的代码依赖于内联汇编程序语句。

内联汇编程序指令将简单地插入到程序流中的给定位置。插入可能对周围代码产生的后果或副作用不被考虑在内。

例如，如果寄存器或内存位置被更改，则可能必须在内联汇编程序指令序列中还原它们，以使其余代码正常工作。

页码:193

内联汇编程序序列与由 C 或 C++代码生成的周围代码没有定义良好的接口。这使得内联汇编程序代码变得脆弱，如果您将来升级编译器，也可能成为一个维护问题。
使用内联汇编程序也有几个限制：
●编译器的各种优化将忽略内联序列的任何影响，这将根本不会被优化
●一般来说，汇编程序指令会导致错误或没有任何意义。但是，数据定义指令将按预期进行工作
●无法访问 Auto 变量。
●标签无法声明
因此，通常最好避免使用内联汇编器。如果没有合适的内在函数，我们建议您使用用汇编程序语言编写的模块，而不是内联汇编程序，因为对汇编程序例程的函数调用通常会导致较少的性能降低。

从 C 语言中调用汇编程序例程
一个将从 C 中调用的汇编程序例程必须：
●符合调用约定
●有一个公共入口点标签
●在任何调用之前声明为外部，允许类型检查和可选的参数升级，如如下例所示：

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

或

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

满足这些需求的一种方法是在 C 语言中创建骨架代码，编译它，并研究汇编程序列表文件。
创建骨架代码
创建具有正确接口的汇编语言例程的建议方法是从由 C 编译器创建的汇编语言源文件开始。请注意，您必须为每个函数原型创建骨架代码。
下面的示例展示了如何创建可以轻松地添加例程的功能主体的骨架代码。基本源代码只需要声明所需的变量，并执行对它们的简单访问。
在本例中，汇编程序例程接受一个 int 和一个字符，然后返回一个 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;
}
```

注意：在本例中，我们在编译代码时使用较低优化级别来显示本地和全局变量访问。如果使用更高级的优化，可以在优化过程中删除所需的局部变量的引用。实际的函数声明不会因优化级别而改变。

编译骨架代码

在 IDE 中，指定文件级别上的列表选项。在工作区窗口中选择该文件。

然后选择 **Project>Options**。在 C/C++编译器类别中，选择 **Override inherited settings.**。在 **List** 页上，取消选择“**Output list file**”，而是选择“**Output assembler file** ”选项及其子选项“**Include source**”。此外，请确保指定一个较低级别的优化。

使用以下选项可以编译基本代码：

```
icc8051 skeleton.c -lA . -On -e
```

-lA 选项创建一个汇编语言输出文件，包括 C 或 C++ 源代码行作为汇编注释。 这 。（句点）指定汇编文件的命名方式应与 C 或 C++ 模块（骨架）相同，但文件扩展名为 s51。 -On 选项意味着不使用优化，并且 -e 启用语言扩展。 此外，请确保使用相关的编译器选项，通常与您在项目中用于其他 C 或 C++ 源文件的选项相同。

结果是汇编源输出文件 skeleton.s51。

注意：-lA 选项创建一个包含调用框架信息 (CFI) 指令的列表文件，如果您打算研究这些指令以及它们的使用方式，这将很有用。

如果只想研究调用约定，可以从列表文件中排除 CFI 指令。

在 IDE 中，选择 Project>Options>C/C++ Compiler>List 并取消选择 Include call frame information 子选项。

在命令行上，使用选项 -lB 而不是 -lA。请注意，源代码中必须包含 CFI 信息才能使 C-SPY 调用堆栈窗口工作。

输出文件

输出文件包含以下重要信息：

- 调用约定
- 返回值
- 全局变量
- 功能参数
- 如何在堆栈上创建空间（自动变量）
- 呼叫帧信息（CFI）。

CFI 指令描述了调试器中调用堆栈窗口所需的调用帧信息。有关详细信息，请参阅呼叫框架信息，第 211 页。

从 C++ 调用汇编程序例程

C 调用约定不适用于 C++ 函数。最重要的是，函数名不足以识别 C++ 函数。函数的范围和类型也需要保证类型安全的链接，并解决重载问题。

另一个区别是非静态成员函数有一个额外的隐藏参数，即 this 指针。

但是，当使用 C 链接时，调用约定符合 C 调用约定。因此，当以这种方式声明时，可以从 C++ 调用汇编程序例程：

```
extern "C"
{
  int MyRoutine(int);
}
```

页码:196

在 C++中，只使用 C 特性的数据结构称为 POD（“普通旧数据结构”），它们使用与 C 中相同的内存布局。但是，我们不建议您从汇编程序例程访问非 POD。

下面的示例演示如何实现与非静态成员函数的等效，这意味着隐式 this 指针必须显式。也可以在成员函数中“包装”对汇编程序例程的调用。使用内联成员函数消除额外调用的开销这假设启用了函数内联：

```
class MyClass;
extern "C"
{
void DoIt(MyClass *ptr, int arg);
}
class MyClass
{
public:
inline void DoIt(int arg)
{
::DoIt(this, arg);
}
};
```

注意：对汇编程序代码中 C++名称的支持非常有限。这意味着：

- 汇编程序列表文件，编译 C++文件通常不能通过汇编程序传递。
- 无法引用或定义在汇编程序中没有 C 链接 C++函数。

调用约定

调用约定是程序中的函数调用另一个函数的方式。编译器会自动处理此情况，但是，如果函数是用汇编语言编写的，则必须知道在何处以及如何找到其参数，如何从调用它的位置返回到程序位置，以及如何返回结果值。

了解汇编程序级例程必须保留哪些寄存器也很重要。如果程序保留了太多寄存器，则该程序可能无效。如果它保留的寄存器太少，结果将是程序不正确。

编译器支持不同的调用约定，这些约定控制如何将内存用于参数和本地声明的变量。您可以指定默认调用约定，也可以显式声明每个函数的调用约定。
本节介绍编译器使用的调用约定。检查这些项目：
- 选择调用约定
- 函数声明
- C 与 C++联动
- 保留寄存器与临时寄存器
- 功能入口
- 功能出口
- 退货地址处理

在本节的最后，展示了一些示例来描述实践中的调用约定。
选择调用约定
您可以在以下调用约定之间进行选择：
- data 叠加
- Idata 叠加
- Idata 可重入
- Pdata 可重入
- Xdata 可重入
- 扩展堆栈可重入。

有关选择特定调用约定以及何时使用特定调用约定的信息，请参阅自动变量和参数的存储，第 83 页。
在 IDE 中，选择 Project>General Options>Target 页面上的调用约定。

使用正常调用约定的提示
正常的调用约定非常复杂，如果您打算将它用于您的汇编程序例程，您应该创建一个列表文件并查看编译器如何将不同的参数分配给可用的寄存器。有关示例，请参见创建框架代码，第 194 页。
如果您打算使用某种调用约定，您还应该使用 RTMODEL 汇编器指令为运行时模型属性__rt_version 指定一个值：

RTMODEL "__rt_version", "*value*"

参数值应该与编译器内部使用的参数值相同。关于使用什么值的信息，请看生成的列表文件。如果调用惯例在未来的编译器版本中发生变化，编译器内部使用的运行时模型值也将发生变化。使用这种方法可以进行模块一致性检查，因为链接器会对数值之间的不匹配产生错误。关于检查模块一致性的更多信息，请参见检查模块的 一致性，第 141 页。

函数声明

在 C 语言中，为了让编译器知道如何调用函数，必须声明函数。声明可以如下所示：

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

这意味着该函数接受两个参数：一个整数和一个指向字符的指针。该函数返回一个值，即整数。

在一般情况下，这是编译器关于函数的唯一知识。

因此，它必须能够从该信息推断出调用约定。

在 C++源代码中使用 C 链接

在 C++中，函数可以具有 C 或 C++链接。要从 C++调用汇编程序例程，最简单的方法是使 C++函数具有 C 链接。

这是一个使用 C 链接声明函数的示例：

```
extern "C"
{
  int F(int);
}
```

在 C 和 C++之间共享头文件通常是实际的。这是一个声明的例子，它声明了一个在 C 和 C++中都具有 C 链接的函数：

```
#ifdef __cplusplus
extern "C"
{
  #endif
  int F(int);
  #ifdef __cplusplus
}
#endif
```

页码:199

保留与暂存寄存器

一般的 8051 CPU 寄存器分为三个独立的组，在本节中进行介绍。

暂存寄存器

允许任何函数破坏临时寄存器的内容。如果一个函数在调用另一个函数后需要寄存器值，它必须在调用过程中存储它，例如在堆栈上。

以下寄存器是所有调用约定的暂存寄存器：A、B、R0-R5 和进位标志。

此外，DPTR 寄存器（或第一个 DPTR 寄存器，如果有多个）是除 xdata 可重入调用约定和分组例程之外的所有调用约定的临时寄存器。如果函数被间接调用，即通过函数指针，DPTR 寄存器也被认为是临时寄存器。

保留的寄存器

另一方面，保留的寄存器在函数调用中被保留。被调用函数可以将寄存器用于其他用途，但必须在使用寄存器前保存值，并在函数退出时恢复。

对于所有调用约定，所有不是临时寄存器的寄存器都是保留寄存器。它们是 R6、R7、应用程序使用的所有虚拟寄存器 (V0 – Vn)、位寄存器 VB 和 xdata 可重入调用约定中的 DPTR 寄存器以及用于分组函数。但是，如果间接调用函数，则 DPTR 寄存器（或第一个 DPTR 寄存器，如果有多个）不是保留寄存器。

注意：如果您使用多个 DPTR 寄存器，则始终保留除第一个之外的所有寄存器。

专用寄存器

对于某些寄存器，您必须考虑某些先决条件：

● 在分页代码模型中，默认的分页切换 例程使用 SFR 端口 P1 作为 bank 切换寄存器。有关更多详细信息，请参阅 分页代码模型中的 Bank 切换，第 114 页。

● 在分页扩展 2 代码模型中，默认的分页切换 例程使用 MEX1 寄存器和内存扩展堆栈。有关更多详细信息，请参阅 分页扩展 2 代码模型中的 分页切换，第 115 页。

功能入口

可以使用以下基本方法之一将参数传递给函数：

- 在寄存器中
- 在堆栈上
- 在覆盖框中

使用寄存器比绕道内存要高效得多，因此所有调用约定都旨在尽可能多地使用寄存器。只能使用有限数量的寄存器来传递参数；当没有更多的寄存器可用时，剩余的参数在堆栈或覆盖帧中传递。在这些情况下，参数也会在堆栈或覆盖框架中传递（取决于调用约定）：

- 结构类型：struct、union 和 classes
- 可变长度（可变参数）函数的未命名参数；换句话说，声明为 foo(paraml, ...) 的函数，例如 printf。

注意：中断函数不能接受任何参数。

隐藏参数

除了在函数声明和定义中可见且独立于使用的调用约定的参数之外，还可以有隐藏参数：

- 如果函数返回结构，存储结构的内存位置将作为最后一个函数参数传递。隐藏指针的大小取决于使用的调用约定。
- 如果函数是非静态嵌入式 C++ 成员函数，则 this 指针作为第一个参数传递。要求成员函数必须是非静态的原因是静态成员方法没有 this 指针。

寄存器参数

独立于使用的调用约定,五个寄存器 R1-R5 可用于寄存器参数。每个寄存器可以包含一个 8 位值，并且寄存器的组合可以包含更大的值。位参数在寄存器 B 中传递，从 B.0、B.1 等开始。如果需要八个以上的位参数，则在 VB 寄存器中最多再传递八个。

页码:201

这些参数可以在以下寄存器和寄存器组合中传递：

参数	已传入寄存器
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

表 26：用于传递参数的寄存器

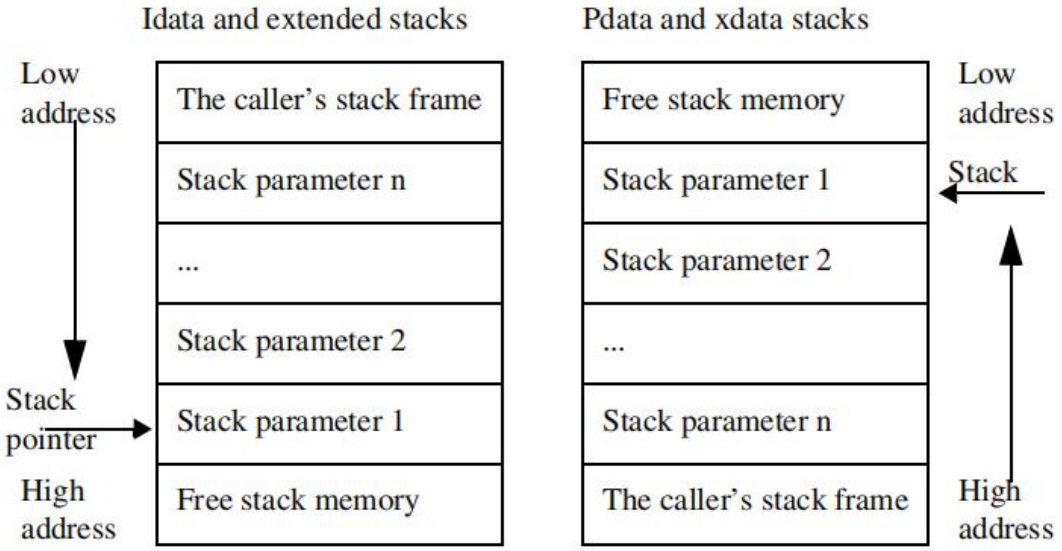
将寄存器分配给参数是一个简单的过程。 严格按照从左到右的顺序遍历参数，将第一个参数分配给第一个可用的寄存器或多个寄存器。 如果没有合适的寄存器可用，则参数将在堆栈或覆盖帧上传递--取决于使用的调用约定。

堆栈参数和布局

堆栈参数存储在内存中，从调用约定指定的堆栈指针指向的位置开始。 第一个堆栈参数直接存储在堆栈指针指向的位置之后。 下一个直接存储在第一个之后，依此类推。
idata 和扩展堆栈向更高地址增长，pdata 和 xdata 堆栈向更低地址增长。

页码:202

当堆栈参数被推送到堆栈上时，就在执行 LCALL 指令之前，堆栈如下所示：



注意：

- 静态叠加函数不使用堆栈。相反，非寄存器参数存储在叠加帧上。
- 对于 Banked extended2 代码模型中的存储函数调用，返回地址的最高有效字节被推送到内存扩展堆栈上。两个较低的字节被推送到 idata 堆栈上。有关更多详细信息，请参阅 Banked 扩展 2 代码模型中的 Bank 切换，第 115 页。

函数退出

函数可以向调用它的函数或程序返回值，也可以具有返回类型 void。

函数的返回值（如果有）可以是标量（如整数和指针）、浮点数或结构。

用于返回值的寄存器

对于所有调用约定，标量返回值在寄存器或携带位中传递。

以下寄存器或寄存器组合用于返回值：

返回值	传入寄存器
1-bit values	Carry
8-bit values	R1
16-bit values	R3:R2
24-bit values	R3:R2:R1
32-bit values	R5:R4:R3:R2

表 27：用于返回值的寄存器

页码:203

返回结构

如果返回结构，调用方将传递一个指针，指向被调用函数应该存储结果的位置。指针作为隐式的最后一个参数传递给函数。被调用的函数以与其他标量结果相同的方式返回指向返回值的指针。

该位置由调用者在调用者堆栈上分配，这取决于当前的调用约定，被调用函数使用默认指针引用该位置。使用的默认指针取决于数据模式。有关更多信息，请参阅选择调用约定，第 198 页。

函数出口处的堆栈布局

调用函数负责清理堆栈。

对于分页函数调用，在堆栈上传递的返回地址可以是 3 个字节，而不是 2 个字节。有关更多信息，请参阅分页交换，第 114 页。

返回地址处理

用汇编语言编写的函数应该在完成后返回给调用者。函数返回地址的位置将随函数的调用约定而变化：

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.	
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

表 28：用于返回值的寄存器

页码:204

† 声明为 __monitor 的函数以与普通函数相同的方式返回，具体取决于使用的调用约定。
中断例程总是使用 RETI 指令返回。
例子
以下部分显示了一系列声明示例和相应的调用约定。示例的复杂性随着时间的推移而增加。
示例 1
假设这个函数声明：

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

此函数在寄存器对 R3:R2 中获取一个参数，并将返回值传递回寄存器对中的调用者。
这个汇编程序与声明兼容； 它将返回一个比其参数值高一个数字的值：

```
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
```


示例 2

这个例子显示了结构是如何在堆栈中传递的。假设有这些声明:

```
struct MyStruct
{
short a;
short b;
short c;
short d;
short e;
};
int MyFunction(struct MyStruct x, int y);
```

调用函数必须在堆栈顶部保留十个字节，并将结构的内容复制到该位置。整数参数 y 在寄存器 R1 中传递。返回值在寄存器对 R3:R2 中传回给调用者。

示例 3

下面的函数将返回一个 struct MyStruct 类型的结构。

```
{
int mA[20];
};
struct MyStruct MyFunction(int x);
```

调用函数的职责是为返回值分配一个内存位置,并将其作为一个隐藏的最后一个参数传递给它的指针。参数 x 将在寄存器对 R3：R2 中传递，因为 x 是第一个参数，而 R3：R2 是用于 16 位寄存器参数的集合中的第一个可用的寄存器对。

隐藏的参数作为第二个参数传递给函数。参数的大小取决于引用将存储返回值的临时堆栈位置的指针的大小。例如，如果该函数使用 idata 或 pdata 重入调用约定，则 8 位指针将位于寄存器 R1 中。如果使用 xdata 重入式调用约定，则 16 位指针将在寄存器对 R5：R4 中传递。如果使用了扩展的堆栈重入式调用约定，那么当一些寄存器需要传递一个 24 位的值时，指针将在堆栈上传递（R3：R2：R1）已占用。

假设该函数被声明为返回一个指向该结构的指针：

```
struct MyStruct *MyFunction(int x);
```

页码:206

在这种情况下，返回值是标量，因此不存在隐藏参数。参数 x 在 R3:R2 中传递，返回值在 R1、R3:R2 或 R3:R2:R1 中返回，具体取决于数据模式。

功能指令

编译器生成函数指令 function、ARGFRAME、LOCFRAME 和 FUNCALL，以将有关函数和函数调用的信息传递给 IAR XLINK 链接器。如果使用编译器选项汇编程序文件（-lA）创建汇编程序列表文件，则可以看到这些指令。

有关函数指令的更多信息，请参阅 8051 的 IAR 汇编程序用户指南。

用于调用函数的汇编指令

本节介绍可用于调用和从 8051 微控制器上的函数返回的汇编指令。
函数可以以不同的方式调用--直接通过函数指针。 在本节中，我们将讨论如何为每个代码模型执行这些类型的调用。
正常的函数调用指令是 LCALL 指令：

```
lcall label
```

被调用函数应返回的位置（即，紧接此指令之后的位置）存储在 idata 调用堆栈上，除非使用扩展堆栈，在这种情况下，该位置存储在扩展堆栈上。

以下各节说明了不同的代码模型如何执行函数调用。

在远近代码模型中调用函数

使用这些代码模型的直接调用很简单：

```
lcall function
```

注：函数在近代码模型中为 16 位地址，在远代码模型中为 24 位地址。
当函数将控制权返回给调用方时，将使用 RET 指令。

当在 Near 代码模型中通过函数指针进行函数调用时，会产生以下汇编代码:

```
mov DPL,#(func&0xFF)          ; Low function address to DPL
mov DPH,#((func>>8)&0xFF)     ; High function address to DPH
lcall ?CALL_IND               ; 进行函数调用的调用库函数
```

当通过 Far 代码模型中的函数指针进行函数调用时，会产生以下汇编代码:

```
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               ; 进行函数调用的调用库函数
```

在分页代码模型中调用函数

在分页代码模型中，直接调用可转化为以下汇编代码：

```
lcall ??function?relay
```

该调用还将生成一个中继函数 ??function?relay ，它有一个 2 字节的地址：

```
??function?relay
lcall ?BDISPATCH ; Call library function in which the function call is made
data
dc24 function     ; Full 3-byte function address
```

分页间接函数调用转换为以下汇编代码：

```
mov DPL,#(??function?relay&0xFF)
mov DPH,#((??function?relay>>8)&0xFF)
lcall ?CALL_IND ; Call library function in which the function call is made
```

分页扩展 2 代码模型中的调用函数

使用 分页扩展 2 代码模型时，直接调用将转换为以下汇编代码：

```
mov ?MEX1,((function>>16)&0xFF)
lcall (function&0xFFFF)
```

间接函数调用如下所示：

```
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
```

内存访问方法

本节介绍数据存储一章中介绍的不同内存类型。

除了介绍用于访问数据的汇编代码之外，本节还将解释不同内存类型背后的原因。

您应该熟悉 8051 指令集，尤其是可以访问内存的指令所使用的不同寻址模式。

对于以下部分中描述的每种访问方法，都有三个示例：

- 访问全局变量
- 使用指针访问内存。

DATA 访问方法

数据存储器是内部数据存储器的前 128 个字节。可以使用直接和间接寻址模式访问该存储器。

例子

```
Accessing the global variable x:
MOV A,x
Access through a pointer, where R0 contains a pointer referring to x:
MOV A,@R0
```

IDATA 访问方法

idata 内存由整个内部数据内存组成。只能使用间接寻址模式访问此内存。

页码:209

例子

```
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 访问方法

pdata 存储器由一个 256 字节的外部存储器块（xdata 存储器）组成。
应该使用的数据页--2 字节地址的高字节--可以通过重新定义符号_PDATA0_START 和 _PDATA0_END 在链接器命令文件中指定。 包含数据页值的 SFR 寄存器也可以通过更改 ?PBANK 符号的定义在链接器命令文件中指定。
只能使用间接寻址模式进行 Pdata 访问。

例子

```
Accessing the global variable x:
MOV R0,#x
Access through a pointer, where R0 contains a pointer referring to x:
MOVX A,@R0
```

XDATA 访问方法

xdata 存储器由多达 64 KB 的外部存储器组成。 Xdata 访问只能使用带有 DPTR 寄存器的间接寻址模式进行：

```
MOV DPTR,#X
MOV A,@A+DPTR
```

FAR22、FAR 和 HUGE 的访问方法

far22、far 和 huge 内存由高达 16 MB 的外部内存（扩展的 xdata 内存）组成。 使用 3 字节数据指针以与 xdata 访问相同的方式执行内存访问。 far22 或 far 指针被限制为 2 字节偏移； 这将最大数据对象大小限制为 64 KB。 巨大的指针没有限制； 它是一个带有 3 字节偏移量的 3 字节指针。

使用 xdata、far 或 huge 访问方法的示例

```
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
```

通用访问方法

通用访问方法仅适用于指针。

在通用数据模式中：

- 应用于非指针数据对象的__generic 属性将被解释为__xdata
- 默认指针是__generic，但默认数据存储器属性是__xdata。

在 Far Generic 数据模式中：

- 应用于非指针数据对象的__generic 属性将被解释为__far22
- 默认指针为__generic，但默认数据存储器属性为__far22。

通用指针大小为 3 个字节。最高有效字节显示指针是指向内部数据、外部数据还是代码存储器。

通用指针非常灵活且易于使用，但这种灵活性是以降低执行速度和增加代码大小为代价的。谨慎使用通用指针，并且只能在受控环境中使用。

寄存器三元组 R3:R2:R1 和 R6:R5:R4 用于访问和操作通用指针。

对通用指针的访问和操作通常由库例程执行。例如，如果寄存器三元组 R3:R2:HR1 包含指向 char 变量 x 的通用指针，则该变量的值通过调用库例程 ?C_GPRT_LOAD 加载到寄存器 A。该例程解码应该访问的内存并将内容加载到寄存器 A。

调用帧信息

当您使用 C-SPY 调试应用程序时，您可以查看调用堆栈，即调用当前函数的函数链。 为了实现这一点，编译器提供了描述调用框架布局的调试信息，特别是有关返回地址存储位置的信息。

如果您希望在调试以汇编语言编写的例程时调用堆栈可用，则必须使用汇编指令 CFI 在汇编源代码中提供等效的调试信息。
该指令在 8051 的 IAR 汇编器用户指南中有详细描述。
CFI 指令
CFI 指令为 C-SPY 提供有关调用函数状态的信息。其中最重要的是返回地址，以及函数或汇编程序入口处的堆栈指针的值。给定这些信息，C-SPY 可以重建调用函数的状态，从而展开堆栈。
关于调用约定的完整描述可能需要大量的调用框架信息。在许多情况下，更有限的方法就足够了。 cstartuproutine 和 __low_level_init 的汇编程序版本都包含足以跟踪调用链的基本调用帧信息，但不尝试跟踪调用函数中的寄存器值。这些例程使用的调用帧信息的通用定义可以在文件 iar_cfi.h 中找到，该文件作为源代码提供。这些定义可以作为为您自己的汇编程序提供调用帧信息的介绍和指南。
对于调用帧信息的完整实现示例，您可以编写一个 C 函数并研究汇编语言输出文件。请参见从 C 调用汇编程序例程，第 194 页。

在描述调用帧信息时，必须存在以下三个组件：
- 描述要跟踪的可用资源的名称块
- 对应调用约定的公共块
- 描述在调用帧上执行的更改的数据块。这通常包括有关何时更改堆栈指针以及何时在堆栈上存储或恢复永久寄存器的信息。

下表列出了编译器使用的名称块中定义的所有资源：

资源	描述
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	

表 29: 调用框架信息的资源

你应该至少跟踪?RET，这样你就可以找到回到调用栈的方法。
追踪 R0-R7、V0-V31 和所有可用的堆栈指针以查看局部变量的值。
如果你的应用程序使用一个以上的寄存器库，你必须跟踪 PSW。

页码:213

使用 CFI 支持创建汇编程序源

创建正确处理调用帧信息的汇编语言例程的推荐方法是从编译器创建的汇编语言源文件开始。

1 从合适的 C 源代码开始，例如：

```
int F(int);
int cfiExample(int i)
{
  return i + F(i);
}
```

2 编译 C 源代码，并确保创建一个包含调用框架信息的列表文件 — CFI 指令。

在命令行上，使用选项 -lA。

在 IDE 中，选择 Project>Options>C/C++ Compiler>List 并确保选中了 Include call frame information 子选项。

页码:214

对于本例中的源代码，列表文件如下所示：

```
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
```

页码:217

```
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
```

页码:218

```
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
```

注意：头文件 iar_cfi.m51 包含宏 XCFI_NAMES 和 XCFI_COMMON，它们声明一个典型的名称块和一个典型的公共块。这两个宏声明了多个资源，包括具体资源和虚拟资源。

（空白页）
页码:220

使用 C

●C 语言概述
●扩展概述
●IAR C 语言扩展

C 语言概述

用于 8051 的 IAR C/C++ 编译器支持 ISO/IEC 9899:1999 标准(包括最高技术勘误 No.3)，也称为 C99。在本指南中，此标准称为标准 C，是编译器中使用的默认标准。这个标准比 C89 更严格。
此外，编译器还支持 ISO 9899:1990 标准（包括所有技术勘误和附录），也称为 C94、C90、C89 和 ANSI C。在本指南中，此标准称为 C89。使用 --c89 编译器选项启用此标准。
C99 标准源自 C89，但增加了如下特性：

- inline 关键字建议编译器紧跟在关键字后面定义的函数应该被内联
- 声明和声明可以在同一范围内混合使用
- for 循环初始化表达式中的声明
- bool 数据类型
- long long 数据类型
- 复数浮点型
- C++ 风格的注释
- 复合文字
- 结构末尾的不完整数组
- 十六进制浮点常量
- 结构和数组中的指定初始化器
- 预处理运算符_Pragma()
- 可变参数宏，它是 printf 风格函数的预处理宏等价物
- VLA（可变长度数组）必须通过编译器选项显式启用--vla

● 使用 asm 或 __asm 关键字的内联汇编程序，请参见内联汇编程序，第 193 页。
注意： 即使它是 C99 功能，用于 8051 的 IAR C/C++ 编译器也不支持 UCN（通用字符名称）。
注意：CLIB 不支持任何 C99 功能。例如，不支持复数和可变长度数组。

扩展概述

该编译器提供标准 C 的特性和广泛的扩展集，从专为嵌入式行业高效编程定制的特性到放宽一些次要标准问题。
这是可用扩展的概述：

● IAR C 语言扩展
有关可用语言扩展的信息，请参阅 IAR C 语言扩展，第 223 页。有关扩展关键字的更多信息，请参阅扩展关键字一章。有关 C++、对语言的两个级别的支持以及 C++ 语言扩展的信息；请参阅使用 C++ 一章。

●Pragma 预编译指令
#Pragma 预编译指令由标准 C 定义，是一种以受控方式使用供应商特定扩展以确保源代码仍然可移植的机制。
编译器提供了一组预定义的 Pragma 预编译指令，可用于控制编译器的行为，例如它如何分配内存、是否允许扩展关键字以及是否输出警告消息。
大多数 Pragma 预编译指令都经过预处理，这意味着宏在 Pragma 预编译指令中被替换。编译指示总是在编译器中启用。其中一些还有相应的 C/C++ 语言扩展。有关可用 Pragma 预编译指令的信息，请参阅 Pragma 预编译指令一章。

● 预处理器扩展
编译器还为您提供了几个与预处理器相关的扩展。有关详细信息，请参阅预处理器一章。

● 内在函数
内在函数提供对低级处理器操作的直接访问，并且在时间要求严格的例程中非常有用。内在函数编译成内联代码，或者作为单个指令或作为短指令序列。有关使用内部函数的更多信息，请参阅混合 C 和汇编器，第 191 页。

页码:222

有关可用功能的信息，请参见章节内在功能。

●库函数

DLIB 运行时环境在适用于嵌入式系统的 C/C++ 标准库中提供 C 和 C++ 库定义。有关详细信息，请参阅 DLIB 运行时环境实施详细信息，第 397 页。

注意：对这些扩展的任何使用，除了 Pragma 预编译指令，都会使您的源代码与标准 C 不一致。

启用语言扩展

你可以通过项目选项来选择不同级别的语言一致性:

命令行	IDE*	描述
--strict	Strict	所有 IAR C 语言扩展均已禁用;对于不属于标准 C 的任何内容，都会发出错误。
None	Standard	标准 C 的所有扩展均已启用，但嵌入式系统编程的扩展不启用。有关扩展的信息，请参阅 IAR C 语言扩展，第 223 页。
-e	Standard with IAR extensions	所有 IAR C 语言扩展均已启用。

表 30：语言扩展

*在 IDE 中，选择“Project>Options>C/C++ Compiler>Language 1>Language conformance，然后选择适当的选项。请注意，语言扩展是默认情况下启用。

IAR C 语言扩展

编译器提供了一组广泛的 C 语言扩展。为了帮助您找到应用程序所需的扩展，它们在本节中按如下方式分组：

● 嵌入式系统编程扩展 — 专为您正在使用的特定微控制器的高效嵌入式编程而定制的扩展，通常是为了满足内存限制

●放宽标准 C，即放宽一些次要的标准 C 问题以及一些有用但次要的语法扩展，请参阅放宽标准 C，第 226 页。

页码:223

嵌入式系统编程的扩展

在 C 和 C++编程语言中都有以下语言扩展，它们非常适用于嵌入式系统编程。

●内存属性、类型属性和对象属性

关于相关的概念、一般的语法规则和参考信息，请参见扩展关键字一章。

● 放置在绝对地址或命名段中

@操作符或指令#pragma location 可用于将全局变量和静态变量放置在绝对地址上，或将变量或函数放置在一个命名的段中。关于使用这些功能的更多信息，请参见控制数据和函数在内存中的位置，第 259 页，以及位置，第 372 页。

●对齐控制

每种数据类型都有自己的对齐方式，更多信息请参见对齐方式，第 325 页。如果你想改变对齐方式，可以使用#pragma data_alignment 指令。如果你想检查一个对象的对齐方式，可以使用 __ALIGNOF__() 操作符。

操作符__ALIGNOF__是用来访问一个对象的对齐方式的。它采用两种形式之一。

● __ALIGNOF__ (类型)

● __ALIGNOF__ (表达式)

在第二种形式中，表达式不被评估。

●匿名结构和联合体

C++包括一个叫做匿名联盟的特性。编译器允许在 C 编程语言中的结构体和联合体有类似的功能。欲了解更多信息，请参见匿名结构体和联合体，第 258 页。

●位域和非标准类型

在标准 C 语言中，位域必须是 int 或无符号 int 类型。使用 IAR C 语言扩展，可以使用任何整数类型或枚举。其优点是，结构体有时会更小。更多信息，请参见 Bitfields，第 327 页。

●static_assert()

结构 static_assert(const-expression, "message");可以在 C/C++中使用。该结构将在编译时被评估，如果 const-expression 为 false，将发出一条包括消息字符串的消息。

页码:224

● 可变宏中的参数

可变宏是 printf 风格函数的预处理器宏等价物。

预处理器接受不带参数的可变宏，这意味着如果没有参数匹配。。。参数， 则在 ", ##__VA_ARGS__"宏定义中删除逗号。根据标准 C，在。。。参数必须至少与一个参数匹配。

专用分段运算符

编译器支持使用以下内置段运算符获取段的起始地址、结束地址和大小：

__segment_begin　返回命名段的第一个字节的地址。

__segment_end　　返回命名段后第一个字节的地址。

__segment_size　　返回命名段的大小（字节）。

注意：别名__segment_begin/__sfb, __segment_end/__sfe, 与 __segment_size/__sfs 也可以使用。

这些运算符可用于链接器配置文件中定义的命名段。

这些操作符在语法上的行为就像声明的那样：

```
void * __segment_begin(char const * segment)
void * __segment_end(char const * segment)
size_t __segment_size(char const * segment)
```

使用@运算符或#pragma location 指令在链接器配置文件中的用户定义段中放置数据对象或函数时，段运算符可用于获取放置段的内存范围的起始地址和结束地址。

命名段必须是字符串文字，并且必须在前面使用#pragma segment 指令声明。如果段是用内存属性 memattr 声明的，\uu segment\u begin 运算符的类型是指向 memattr void 的指针。

否则，该类型是指向 void 的默认指针。请注意，必须启用语言扩展才能使用这些运算符。

例子

在此例子中 __segment_begin operator is void __pdata *.

```
#pragma segment="MYSEGMENT" __pdata
...
segment_start_address = __segment_begin("MYSEGMENT");
```

另见第 378 页 "段 "和第 372 页 "位置"。

放宽到标准 C

本节列出并简要描述了一些标准 C 问题的放宽以及一些有用但较小的语法扩展：

● 不完整类型的数组

数组的元素类型可以是不完整的结构、联合或枚举类型。

类型必须在使用数组之前完成（如果是），或者在编译单元结束时（如果不是）。

● 枚举类型的前向声明

扩展允许您首先声明枚举的名称，然后通过指定大括号括起来的列表来解析它。

● 接受结构或联合说明符末尾缺少的分号 如果结构或联合说明符末尾的分号丢失,则会发出警告而不是错误。

● 无效和无效

在指针操作中，如果需要，指向 void 的指针总是隐式转换为另一种类型，如果需要，空指针常量总是隐式转换为正确类型的空指针。在标准 C 中，一些运算符允许这种行为，而另一些则不允许。

● 在静态初始化器中将指针转换为整数

在初始化程序中,如果整数类型足够大以包含它,则可以将指针常量值强制转换为整数类型。有关转换指针的更多信息，请参见转换，第 333 页。

● 取寄存器变量的地址

在标准 C 中，将指定的变量地址作为寄存器变量是非法的。编译器允许这样做，但会发出警告。

● long float 表示双倍

long float 类型被接受为 double 的同义词。

● 重复的 typedef 声明

允许在同一范围内重新声明 typedef，但会发出警告。

● 混合指针类型

指向可互换但不相同的类型的指针之间允许赋值和指针差异；例如，unsigned char * 和 char *。这包括指向相同大小的整数类型的指针。发出警告。

允许将字符串常量分配给指向任何类型字符的指针，并且不会发出警告。

● 非顶级常量

如果目标类型添加了不在顶层的类型限定符（例如，int ** 到 int const **），则允许分配指针。

也允许比较和获取这些指针的差异。

● 非左值数组

非左值数组表达式在使用时会转换为指向数组第一个元素的指针。

● 预处理器指令末尾的注释

除非使用严格的标准 C 模式，否则启用此扩展，它使得在预处理器指令之后放置文本是合法的。这个语言扩展的目的是支持遗留代码的编译；我们不建议您以这种方式编写新代码。

● 枚举列表末尾的额外逗号

允许在枚举列表的末尾放置一个额外的逗号。在严格的标准 C 模式下，会发出警告。

● } 前的标签

在标准 C 中，标签后面必须至少有一个语句。因此，将标签放在块的末尾是非法的。编译器允许这样做，但会发出警告。

请注意，这也适用于 switch 语句的标签。

● 空声明

允许使用空声明（单独使用分号），但会发出备注（前提是启用备注）。

● 单值初始化

标准 C 要求静态数组、结构和联合的所有初始化表达式都用大括号括起来。

单值初始化器可以不带大括号出现，但会发出警告。编译器接受这个表达式：

```
struct str
{
  int a;
} x = 10;
```

页码:227

● 其他范围的声明

其他作用域中的外部声明和静态声明是可见的。在下面的示例中，变量 y 可以在函数的末尾使用，即使它应该只在 if 语句的主体中可见。发出警告。

```
int test(int x)
{
  if (x)
  {
    extern int y;
    y = 1;
  }
  return y;
}
```

● 以函数为上下文，将函数名扩展为字符串

在函数体内使用任何符号 __func__ 或 __FUNCTION__ 使符号扩展为包含当前函数名称的字符串。

使用 symbol __PRETTY_FUNCTION__ 也包括参数类型和返回类型。例如，如果你使用 __PRETTY_FUNCTION__ symbol，结果可能是这样的。"void func(char)"oid func(char)"

这些符号对断言和其他跟踪实用程序很有用,它们需要启用语言扩展,请参见 -e,第 302 页。

● 函数和块作用域中的静态函数

静态函数可以在函数和块范围内声明。他们的声明被移动到文件范围。

● 根据数字语法扫描的数字

根据数字的语法而不是 pp-number 语法扫描数字。因此，0x123e+1 被扫描为三个令牌而不是一个有效令牌。 （如果使用 --strict 选项，则使用 pp-number 语法。）

使用 C++

- 概述--EC++ 和 EEC++
- 启用对 C++ 的支持
- EC++ 功能说明
- EEC++ 功能说明
- C++ 语言扩展

概述--EC++ 和 EEC++

IAR Systems 支持 C++ 语言。
您可以选择行业标准嵌入式 C++ 和扩展嵌入式 C++。
使用 C++ 描述使用 C++ 语言时需要考虑什么。
嵌入式 C++ 是 C++ 编程语言的适当子集，旨在用于嵌入式系统编程。
它是由一个行业联盟定义的，嵌入式 C++ 技术委员会。
性能和便携性尤为突出在嵌入式系统开发中很重要，在定义语言。
EC++ 提供与 C++ 相同的面向对象的好处,但没有一些可以以难以预测的方式增加代码大小和执行时间的功能。

嵌入式 C++

支持以下 C++ 功能：

- 类，是用户定义的类型，包含数据结构和行为；继承的基本特征允许数据结构和行为在类之间共享
- 多态性，这意味着一个操作可以在不同的情况下表现不同类，由虚函数提供
- 运算符和函数名的重载，允许多个运算符或具有相同名称的函数，前提是它们的参数列表足够不同的
- 使用操作符 new 和 delete 的类型安全内存管理
- 内联函数，特别适合内联扩展。

被排除在外的 C++ 特性是那些在执行时间中引入开销的特性或超出程序员控制的代码大小。还排除了功能在标准 C++ 定义之前很晚才添加。
因此，嵌入式 C++ 提供了一个子集的 C++，它是有效的，并且由现有的开发工具完全支持。
嵌入式 C++ 缺少 C++ 的这些特性：

- 模板
- 多重和虚拟继承
- 异常处理
- 运行时类型信息
- 新的转换语法(运算符 dynamic_cast、static_cast、reinterpret_cast 和 const_cast）
- 命名空间
- 可变属性。

这些语言特性的排除使得运行时库显着增加高效的。嵌入式 C++ 库也不同于完整的 C++ 库：

- 标准模板库（STL）被排除在外
- 无需使用模板即可支持流、字符串和复数
- 与异常处理和运行时类型信息相关的库功能

（头文件 except、stdexcept 和 typeinfo）被排除在外。
注意：该库不在 std 命名空间中，因为 Embedded C++ 不支持命名空间。

扩展嵌入式 C++

IAR Systems 的扩展 EC++ 是 C++ 的一个稍大的子集，它添加了这些
标准 EC++ 的功能：

- 完整的模板支持
- 命名空间支持
- 可变属性
- 转换运算符 static_cast、const_cast 和 reinterpret_cast。

所有这些添加的功能都符合 C++ 标准。
为了支持 Extended EC++，本产品包含一个标准模板版本库 (STL)，换句话说，C++ 标准章节实用程序、容器、迭代器、算法和一些数字。

页码:230

此 STL 专为与 Extended EC++ 一起使用而定制语言，这意味着没有异常并且不支持运行时类型信息(rtti)。此外，该库不在 std 命名空间中。
注意：启用扩展 EC++ 编译的模块与在未启用扩展 EC++ 的情况下编译的模块。

启用对 C++ 的支持
在编译器中，默认语言是 C。
要编译用嵌入式 C++ 编写的文件，请使用 --eC++ 编译器选项。
参见--eC++，第 303 页。
要利用源代码中的扩展嵌入式 C++ 功能，请使用--eeC++ 编译器选项。参见--eeC++，第 303 页。
对于 EC++ 和 EEC++，您还必须使用 IAR DLIB 运行时库。
要在 IDE 中启用 EC++ 或 EEC++，请选择 **Project>Options>C/C++ Compiler>Language 1** 并选择适当的标准。

EC++ 功能描述
当您为 8051 的 IAR C/C++ 编译器编写 C++ 源代码时，您必须意识到混合 C++ 功能时的一些好处和一些可能的怪癖--
例如类和类成员--具有 IAR 语言扩展，例如 IAR 特定属性。

在类中使用 IAR 属性
C++ 类的静态数据成员的处理方式与全局变量相同，并且可以有任何适用的 IAR 类型、内存和对象属性。

成员函数的处理方式通常与自由函数相同，并且可以具有任何适用的 IAR 类型、内存和对象属性。虚成员函数可以仅具有与默认函数指针和构造函数兼容的属性析构函数不能有任何这样的属性。

位置运算符 @ 和 #pragma location 指令可用于静态数据成员和所有成员函数。

将属性与类一起使用的示例

```
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";
};
```

指针

指针用于引用一个类对象或调用一个类的成员函数默认情况下,类对象将具有默认数据指针的数据内存属性类型。

这意味着这样的类对象只能被定义为驻留在内存中哪些指针可以隐式转换为默认数据指针。这个限制也可能适用于驻留在堆栈上的对象，例如临时对象和自动对象。

类记忆

为了弥补这个限制，一个类可以与一个类内存类型相关联。

类内存类型改变：

- 将 this 指针类型在所有成员函数、构造函数和析构函数中转换为指向类内存的指针
- 静态存储持续时间变量的默认内存--也就是说，不是自动的变量--类类型，到指定的类内存中
- 指针类型用于指向类类型的对象，变成类指针记忆。

页码:232

示例

```
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
```

注意：不允许将 C 类放在 xdata 内存中，因为指针很大不能隐式转换为 __xdata 指针。
每当必须声明与类内存类型关联的类类型（如 C）时，还必须提到类内存类型：
类 __xdata C;
另请注意，与不同类记忆关联的类类型不兼容类型。
内置运算符返回与类关联的类内存类型，
__memory_of(class). For instance, __memory_of(C) returns __xdata

继承时，规则是必须可以将指向子类的指针隐式转换为指向其基类的指针。
这意味着子类可以具有比其基类更严格的类内存，但不能具有更少限制的类内存。

页码:233

```
class __xdata D : public C
{ // OK，相同类的内存
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
{ // 好的，将与与 C 相同的类内存相关联
public:
void MyG();
int mJ;
};
```

请注意，不允许执行以下操作，因为 huge 不在远内存中：

```
class __huge G:public C
{
};
```

类上的新表达式将在与类内存关联的堆中分配内存。 删除表达式自然会将内存释放回同一个堆。 要覆盖类的默认 new 和 delete 运算符，请声明

```
void *operator new(size_t);
void operator delete(void *);
```

作为成员函数，就像在普通 C++ 中一样。
如果无法将指向类内存的指针隐式转换为默认指针类型,则无法为该类创建临时对象,例如,如果您有 xdata 默认指针，则以下示例将不起作用：

```
class __idata Foo {...}
void some_fun (Foo arg) {...}
Foo another_fun (int x) {...}
```

页码:234

有关内存类型的更多信息，请参阅内存类型，第 68 页。

功能类型

具有外部“C”链接的函数类型与具有 C++ 链接的函数兼容。

示例

```
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
}
```

新的和删除的操作员

对于每个可以有堆的内存，即 xdata、far 和 huge 内存，都有 new 和 delete 的操作符。

```
#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
```

如果您想为任何数据存储器覆盖全局和特定于类的 operator new 和 operator delete，请使用此语法。

请注意，有一种特殊的语法来命名每个内存的操作符 newfunctions，而操作符 deletefunctions 的命名依赖于正常的重载。

页码:236

新建和删除表达式

一个新的表达式为给定类型的内存调用 operator new 函数。 如果使用带有类内存的类、结构或联合类型，则类内存将确定调用的运算符 newfunction。 例如，

```
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
}
```

请注意，删除表达式中使用的指针必须具有正确的类型，即与新表达式返回的类型相同。 如果您使用指向错误内存的指针，则结果可能是损坏的堆。

在中断中使用静态类对象

如果中断函数使用需要构造的静态类对象（使用构造函数）或销毁（使用析构函数），如果中断发生在对象被构造之前，或者在对象被销毁期间或之后，您的应用程序将无法正常工作。为避免这种情况，请确保在构造静态对象之前不启用这些中断，并在从 main 或调用 exit 返回时禁用这些中断。 有关系统启动的信息，请参阅系统启动和终止，第 164 页。

页码:237

函数局部静态类对象在第一次执行通过时构造它们的声明，并在从 main 或调用 exit 时返回时被销毁。

使用新的处理程序

要处理内存耗尽，您可以使用 set_new_handler 函数。

嵌入式 C++ 中的新处理程序

如果您不调用 set_new_handler，或者使用 NULL 新处理程序调用它，并且 operator new 未能分配足够的内存，它会调用 abort。诺特罗 new 运算符的变体将改为返回 NULL。

如果您使用非 NULL 新处理程序调用 set_new_handler，则提供的新处理程序处理程序将由 operator newif operator newfails 调用以分配内存。这然后新的处理程序必须提供更多可用内存并返回或中止执行一些礼仪。运算符 new 的 nothrowvariant 永远不会在新处理程序的存在。

模板

扩展 EC++ 支持根据 C++ 标准的模板，但不支持导出关键字。该实现使用两阶段查找，这意味着关键字 typename 必须在需要的地方插入。此外，在每次使用模板时，所有可能模板的定义必须是可见的。这意味着定义所有模板必须在包含文件或实际源文件中。

C-SPY 中的调试支持

C-SPY® 具有对 STL 容器的内置显示支持。逻辑结构容器在手表视图中以一种易于理解的综合方式呈现了解并遵循。有关这方面的更多信息，请参阅 8051 的 C-SPY® 调试指南。

EEC++ 功能说明

本节介绍区分扩展 EC++ 和 EC++ 的功能。

模板

编译器支持具有标准定义的语法和语义的模板
C++。 但是，请注意产品随附的 STL（标准模板库）
为扩展 EC++ 量身定制，请参阅扩展嵌入式 C++，第 230 页。

页码:238

模板和数据存储器属性

为了使数据存储器属性在模板中按预期工作，标准 C++ 模板处理已更改--类模板部分特化匹配和函数模板参数推导。

在扩展嵌入式 C++ 中，类模板偏特化匹配算法是这样工作的：
当指针或引用类型与指针或对模板的引用匹配时参数类型，模板参数类型将是指向的类型，去掉任何数据存储器属性，如果结果指针或引用类型相同。

例子

```
// 我们假设 __far 是默认指针的内存类型。
// 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
```

Extended Embedded C++中，函数模板参数推导算法
像这样工作：
当执行函数模板匹配并且参数用于
扣除；如果该参数是指向可以隐式转换为的内存的指针
一个默认指针，做参数推导就好像它是一个默认指针。
当一个参数与一个引用匹配时，像参数一样进行推导
并且参数都是指针。

例子

```
// 我们假设 __far 是默认指针的内存类型。
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
}
```

对于使用这种修改后的算法匹配的模板，不可能为指向“小”内存类型的指针自动生成特殊代码。
对于“大”和“其他”内存类型（默认指针无法指向的内存），这是可能的。
为了使编写完全内存感知的模板成为可能--在极少数情况下这很有用--在模板函数声明之前使用#pragma basic_template_matching 指令。然后，该模板函数将在没有上述修改的情况下匹配。

例子

```
// 我们假设 __far 是默认指针的内存类型。
#pragma basic_template_matching
template<typename T> void fun(T *);
void MyF()
{
  fun((int __pdata *) 0); // T = int __pdata
}
```

非类型模板参数

允许引用内存类型作为模板参数，即使不允许指向该内存类型的指针。

例子

```
extern __no_init __sfr int X @0xf2;
template<__sfr int &y>
void Foo()
{
y = 17;
}
void Bar()
{
Foo<X>();
}
```

标准模板库

STL 中的容器，如 vector 和 map，是内存属性感知的。 这意味着可以将容器声明为驻留在特定的内存类型中，这会产生以下后果：

- 容器本身将驻留在所选内存中
- 容器中的元素分配将使用堆来存储所选内存
- 它内部的所有引用都使用指向所选内存的指针

例子

```
#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
```

请注意，这是非法的：

```
vector<int __far22> __huge Z;
```

另请注意，map<key, T>、multimap<key, T>、hash_map<key, T> 和 hash_multimap<key, T> 都使用 T 的内存。这意味着这些集合的 value_type 将是 pair< key, const T> mem 其中 mem 是 T 的内存类型。提供具有内存类型的键是没有用的。

页码:241

例子

请注意，仅在它们使用的数据存储器属性上不同的两个容器不能相互分配。 相反，必须使用模板分配成员方法。

```
#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());
}
```

强制转换操作符的变体

在 Extended EC++中，可以使用 C++强制转换操作符的其他变体：

```
const_cast<to>(from)
static_cast<to>(from)
reinterpret_cast<to>(from)
```

可变的

扩展 EC++ 支持 mutable 属性。 即使整个类对象是 const，也可以更改可变符号。

命名空间

命名空间功能仅在扩展 EC++ 中受支持。 这意味着您可以使用命名空间来划分代码。 但是请注意，库本身并未放置在 std 命名空间中。

标准命名空间

标准 EC++ 或扩展 EC++ 中均未使用 std 命名空间。 如果您的代码引用了 std 命名空间中的符号，只需将 std 定义为空；

例如：

```
#define std
```

您必须确保应用程序中的标识符不会干扰运行时库中的标识符。

页码:242

指向成员函数的指针

指向成员函数的指针只能包含默认函数指针，或者可以隐式转换为默认函数指针的函数指针。要使用指向成员函数的指针，请确保应指向的所有函数都驻留在默认内存或默认内存中包含的内存中。

例子

```
class X
{
public:
__near_func1 void F();
};
void (__near_func1 X::*PMF)(void) = &X::F;
```

C++ 语言扩展

当您在任何 C++ 模式下使用编译器并启用 IAR 语言扩展时，编译器中会提供以下 C++ 语言扩展：

● 在类的友元声明中，class 关键字可以省略，例如：

```
class B;
class A
{
friend B; //Possible when using IAR language
//extensions
friend class B; //According to the standard
};
```

● 标量类型的常量可以在类中定义，例如：

```
class A
{
const int mSize = 10; //Possible when using IAR language
//extensions
int mArr[mSize];
};
```

根据标准，应该使用初始化的静态数据成员。

● 在类成员的声明中，可以使用限定名，例如：

```
struct A
{
int A::F(); // Possible when using IAR language extensions
int G(); // According to the standard
};
```

页码:243

● 允许在指向具有 C 链接的函数的指针 (extern "C") 和指向具有 C++ 链接的函数的指针 (extern "C++") 之间使用隐式类型转换，例如：

```
extern "C" void F(); // Function with C linkage
void (*PF)() // PF points to a function with C++ linkage
= &F; // Implicit conversion of function pointer.
```

根据标准，必须显式转换指针。

● 如果构造中的第二个或第三个操作数包含？ 运算符是字符串文字或宽字符串文字（在 C++ 中是常量），操作数可以隐式转换为 char * 或 wchar_t *，例如：

```
bool X;
char *P1 = X ? "abc" : "def"; //Possible when using IAR
//language extensions
char const *P2 = X ? "abc" : "def";//According to the standard
```

● 函数参数的默认参数不仅可以在顶级函数声明中指定,这符合标准,而且可以在 typedef 声明、指向函数的函数声明和指向成员的函数声明中指定 .

● 在包含非静态局部变量的函数和包含非求值表达式（例如 sizeof 表达式）的类中，表达式可以引用非静态局部变量。 但是，会发出警告。

● 可以通过 typedef 名称将匿名联合引入包含类。

没有必要先声明联合。 例如：

```
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.
};
```

此外，此扩展还允许匿名类和匿名结构，只要它们没有 C++ 特性（例如，没有静态数据成员或成员函数，也没有非公共成员）并且除了其他匿名类之外没有嵌套类型 、结构或联合。例如：

页码:244

```
struct A
{
struct
{
int i,j;
}; // OK -- references to A::i and A::j are allowed.
};
```

友好类的语法允许非类的类型以及通过类型定义表达的类的类型，而没有详细的类型名称。
比如说:

```
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
};
```

● 允许指定一个没有大小或大小为 0 的数组作为结构的最后一个成员。
例如：

```
typedef struct
{
int i;
char ir[0]; // Zero-length array
};
typedef struct
{
int i;
char ir[]; // Zero-length array
};
```

注意：如果在没有首先启用语言扩展的情况下使用这些构造中的任何一种，都会出现错误。

页码:245

（空白页）
页码:246

与应用程序相关的注意事项

- 堆栈注意事项
- 堆的考虑
- 工具和应用程序之间的交互
- 校验和计算，以验证图像的完整性

堆栈注意事项

为了使应用程序高效地使用堆栈内存，需要考虑一些问题。

堆栈大小注意事项

所需的堆栈大小在很大程度上取决于应用程序的行为。如果给定的堆栈大小太大，RAM 将被浪费。如果给定的堆栈大小太小，可能会发生两件事中的一件，这取决于您的堆栈在内存中的位置：

- 变量存储将被覆盖，从而导致未定义的行为
- 堆栈将落在内存区域之外，导致应用程序的异常终止。

这两种替代方案都有可能导致应用程序失败。因为第二种选择更容易检测，所以您应该考虑放置堆栈，使其增长到 RAM 内存的末尾。

有关堆栈大小的更多信息，请参见设置堆栈存储器，第 132 页，和节省堆栈空间和 RAM 存储器，第 270 页。有关堆栈大小的详细信息，还请参见芯片制造商的文档。

堆的考虑

堆包含通过使用 C 函数 malloc（或相应的函数）或 C++操作符 new 来分配的动态数据。

如果您的应用程序使用动态内存分配，您应该熟悉：

- 用于堆的链接器段
- 分配堆大小，请参见设置堆内存，第 135 页。

DLIB 中的堆段

要访问特定内存中的堆，请使用适当的内存属性作为标准函数 malloc、free、calloc 和 realloc 的前缀，例如：

__pdata_malloc

如果您使用任何不带前缀的标准函数，该函数将被映射到默认的内存类型 pdata。

每个堆将驻留在一个名为 _HEAP 的段中，该段以内存属性为前缀，例如 PDATA_HEAP。

有关可用堆的信息，请参阅堆上的动态内存，第 90 页。

CLIB 中的堆段

分配给堆的内存放在段 HEAP 中，只有在实际使用动态内存分配时才会包含在应用程序中。

堆大小和标准 I/O

如果您从 DLIB 运行时环境中排除 FILE 描述符，就像在正常配置中一样，则根本没有输入和输出缓冲区。

否则，与完整配置一样，请注意输入和输出缓冲区的大小在 stdio 库头文件中设置为 512 字节。

如果堆太小，则不会缓冲 I/O，这比缓冲 I/O 时要慢得多。如果您使用 IAR C-SPY® 调试器的模拟器驱动程序执行应用程序，您可能不会注意到速度损失，但当应用程序在 8051 微控制器上运行时，它会非常明显。

如果使用标准 I/O 库，则应将堆大小设置为适应标准 I/O 缓冲区需要的值。

工具与您的应用程序之间的交互

链接过程和应用程序可以通过四种方式进行象征性交互：

- 使用链接器命令行选项-D 创建符号。 链接器将创建一个公共绝对常量符号，应用程序可以将其用作标签、大小、调试器设置等。
- 使用编译器运算符 __segment_begin、__segment_end 或__segment_size 或命名段上的汇编器运算符 SFB、SFE 或 SIZEOF。 这些运算符提供对具有相同名称的连续段序列的起始地址、结束地址和大小的访问。

● 命令行选项-s 通知链接器应用程序的开始标签。 它被链接器用作根符号并通知调试器从哪里开始执行。

以下几行说明了如何使用 -D 创建符号。 如果您需要使用此机制，请将这些选项添加到您的命令行中，如下所示：

```
-Dmy_symbol=A
-DMY_HEAP_SIZE=400
```

链接器配置文件可能如下所示：

```
-Z(DATA)MyHeap+MY_HEAP_SIZE20000-2FFFF
```

将这些行添加到您的应用程序源代码中：

```
#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;
}
```

页码:249

用于验证图像完整性的校验和计算

本节包含有关校验和计算的信息：

- 关于校验和计算的简要说明，第 250 页
- 计算和验证校验和，第 252 页
- 校验和计算故障排除，第 256 页

关于校验和计算的简要说明

您可以使用校验和来验证图像在运行时是否与生成图像的原始校验和时相同。换句话说，验证图像没有损坏。

这工作如下：

- 您需要一个初始校验和。

您可以使用链接器生成初始校验和，也可以使用第三方校验和。

- 您必须在运行期间生成第二个校验和。

您可以在应用程序源代码中添加特定代码以在运行时计算校验和，也可以使用设备上的一些专用硬件在运行时计算校验和。

- 您必须将特定代码添加到您的应用程序源代码以比较两个校验和，并在它们不同时采取适当的措施。

如果两个校验和以相同的方式计算，并且图像中没有错误，则校验和应该相同。如果不是，您首先应该怀疑这两个校验和不是以相同的方式生成的。

页码:250

无论您使用哪种解决方案来生成两个校验和，都必须确保两个校验和以完全相同的方式计算。如果您将链接器用于初始校验和并在运行时使用基于软件的计算，则您可以完全控制两个校验和的生成。

但是，如果您将第三方校验和用于初始校验和，或者在运行时对校验和计算使用某些硬件支持，则可能需要考虑其他要求。

对于这两个校验和，您必须始终考虑一些选择，并且只有在有其他要求时才能做出一些选择。尽管如此，两个校验和的所有细节都必须相同。

始终需要考虑的细节：

● 校验和范围

要通过校验和验证的内存范围（或范围）。

通常，您可能想要计算所有 ROM 内存的校验和。但是，您可能只想计算特定范围的校验和。请记住：

● 一个校验和可以有多个范围。

● 通常，必须从每个内存范围的最低地址到最高地址计算校验和。

● 每个内存范围必须按照定义的相同顺序进行验证（例如，0x100 - 0x1FF,0x400 - 0x4FF 与 0x400 - 0x4FF,0x100 - 0x1FF 不同）。

● 如果使用多个校验和，则应将它们放在具有唯一名称的部分中，并使用唯一的符号名称。

● 绝不应在包含校验和或软件断点的内存范围内计算校验和。

● 校验和的算法和大小

您应该考虑哪种算法最适合您的情况。有两种基本选择，Sum（一种简单的算术算法）或 CRC（最常用的算法）。

对于 CRC，校验和有不同的大小可供选择，2 或 4 个字节，其中预定义的多项式足够宽以适应大小，以获得更大的错误检测能力。预定义多项式适用于大多数数据集，但可能不适用于所有数据集。如果没有，您可以指定自己的多项式。如果您只是想要一个体面的错误检测机制，请使用预定义的 CRC 算法来满足您的校验和大小，通常是 CRC16 或 CRC32。

请注意，对于 n 位多项式，始终认为第 n 位已设置。对于 16 位多项式（例如 CRC16），这意味着 0x11021 与 0x1021 相同。

有关为具有非均匀分布的数据集选择适当多项式的更多信息，请参见例如 Tannenbaum, A.S. 中的第 3.5.3 节，

“计算机网络”，Prentice Hall 1981，ISBN：0131646990。

● 填充

校验和范围内的每个字节都必须具有明确定义的值，然后才能计算校验和。通常，具有未知值的字节是为对齐而添加的填充字节。这意味着您必须指定在计算期间要使用的填充模式，通常为 0xFF 或 0x00。

● 初始值

校验和必须始终具有明确的初始值。

除了这些强制性细节之外，可能还有其他细节需要考虑。

通常，当您有第三方校验和、您希望校验和与 Rocksoft™ 校验和模型兼容时，或者当您使用硬件支持在运行时生成校验和时，可能会发生这种情况。链接器还支持控制对齐、补码、位顺序和校验和单元大小。

计算和验证校验和

在此示例过程中，为 ROM 存储器从 0x8002 到 0x8FFF 计算校验和，并将计算得到的 2 字节校验和放置在 0x8000。

1 CHECKSUM 段仅在您需要该段时才会包含在您的应用程序中。如果应用程序本身不需要校验和，请使用链接器选项 -g__checksum 强制包含该段。

2 配置链接器以计算校验和时，需要做出一些基本选择：

● 校验和算法

选择您要使用的校验和算法。在本例中，使用了 CRC16 算法。

● 内存范围

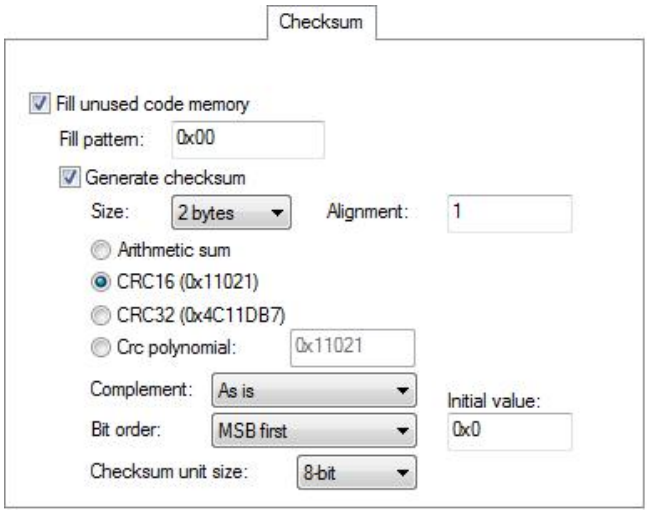
使用 IDE，默认情况下将为基于 ROM 的存储器的所有放置指令（在链接器配置文件中指定）计算校验和。从命令行，您可以指定任何范围。

● 填充图案

为具有未知值的字节指定填充模式（通常为 0xFF 或 0x00）。填充模式将用于所有校验和范围。

有关详细信息，请参阅第 250 页的关于校验和计算的简要说明。

若要从 IDE 运行链接器，请选择“Project>Options>Linker>Checksum”，然后选择进行设置，例如：



在最简单的情况下，您可以忽略（或保留默认设置）这些选项：
补码、位顺序和校验和单元大小。
要使链接器从命令行创建校验和，请使用 -J 链接器选项，例如：
-J2,crc16,,__checksum,CHECKSUM,1=0x8002 - 0x8FFF
校验和将在您构建项目时创建，并将放置在自动生成的段 CHECKSUM 中。如果您使用自己的链接器配置文件，或者如果您明确想要控制 CHECKSUM 段的放置，则必须相应地使用放置信息更新链接器配置文件。在这种情况下，请确保放置段，使其不是应用程序校验和计算的一部分。

3 您可以指定多个范围，而不仅仅是一个范围。
如果您使用的是 IDE，请执行以下步骤：
● 选择 Project>Options>Linker>Checksum 并确保取消选择 Fill used code memory。
● 选择 Project>Options>Linker>Extra Opions 并指定范围，例如：
-h(CODE)0-3FF,8002-8FFF
-J2,crc16,,,1=0-3FF,8002-8FFF

如果您使用的是命令行，请使用 -J 选项并指定范围。 例如像这样：
-h(CODE)0-3FF,8002-8FFF
-J2,crc16,,,1=0-3FF,8002-8FFF

4 在源代码中添加校验和计算函数。 确保函数使用与链接器计算的校验和相同的算法和设置。 例如，crc16 算法的慢速变体，但内存占用很小（与使用更多内存的快速变体相反）：

```
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;
}
```

您可以在产品安装目录 8051\src\linker 中找到此校验和算法的源代码。

5 确保您的应用程序还包含对计算校验和、比较两个校验和并在校验和值不匹配时采取适当措施的函数的调用。 此代码提供了一个示例，说明如何为您的应用程序计算校验和并与链接器生成的校验和进行比较：

```
/* 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 */
}
```

注意：确保在链接器配置文件中定义符号 __checksum_begin 和 __checksum_end。

6 构建您的应用程序项目并下载它。
在构建期间，链接器创建一个校验和并将其放置在 CHECKSUM 段中的指定符号 __checksum 中。

7 选择下载和调试以启动 C-SPY 调试器。

页码:255

在执行期间，链接器计算的校验和与您的应用程序计算的校验和应该相同。

故障排除校验和计算

如果两个校验和不匹配，有几个可能的原因。以下是一些故障排除提示：

- 如果可能，在尝试使校验和匹配时从一个小示例开始。
- 确认两个校验和计算中使用了完全相同的内存范围。

为了帮助您做到这一点，链接器会在映射文件中生成有用的信息，这些信息与所使用的确切地址和访问顺序有关。

- 确保从所有校验和计算中排除所有校验和符号。

在映射文件中，注意校验和符号及其大小，有关其位置的信息，请检查模块映射或条目列表。将该位置与校验和范围进行比较。

- 验证校验和计算是否使用相同的多项式。
- 验证字节中的位在两次校验和计算中的处理顺序是否相同，从最低有效位到最高有效位或相反。您可以使用 Bit order 选项（或从命令行， --checksum 选项的 -m 参数）来控制它。
- 如果您使用 CRC 的小变体，请检查您是否需要向算法中输入额外的字节。

在字节序列末尾添加的零的数量必须与校验和的大小相匹配，换句话说，一个 0 用于 1 字节校验和，两个零用于 2 字节校验和，四个零用于 4-字节校验和。

- 跨越 64 KB 边界时，请仔细考虑指针中的位数。

例如,如果指针 0xFFFF 加 1,如果指针是 16 位,这通常会导致指针 0x0(而不是 0x10000)。

- 闪存中的任何断点都会改变闪存的内容。这意味着您的应用程序计算的校验和将不再匹配链接器计算的初始校验和。要使两个校验和再次匹配，您必须禁用闪存中的所有断点以及 C-SPY 内部在闪存中设置的任何断点。堆栈插件和调试器选项 Run 都需要 C-SPY 设置断点。在 8051 的 C-SPY® 调试指南中阅读有关可能的断点使用者的更多信息。

嵌入式应用程序的高效编码

- 选择数据类型
- 控制内存中的数据和函数放置
- 控制编译器优化
- 促进良好的代码生成

选择数据类型

为了有效地处理数据，您应该考虑使用的数据类型和最有效的变量放置。

使用有效的数据类型

应仔细考虑您使用的数据类型，因为这会对代码大小和代码速度产生很大影响。

- 使用小型和无符号数据类型（无符号字符和无符号短），除非您的应用程序确实需要有符号值。
- 应避免使用大小不是 1 位的位域，因为与位操作相比，它们会导致代码效率低下。
- 声明一个指向常量数据的指针参数可能会为调用函数提供更好的优化。

有关支持的数据类型、指针和结构类型的表示的信息，请参阅数据表示一章。

浮点类型

在没有数学协处理器的微处理器上使用浮点类型在代码大小和执行速度方面都非常低效。因此，您应该考虑将使用浮点运算的代码替换为使用整数的代码，因为这样更有效。

编译器仅支持 32 位浮点格式。不支持 64 位浮点格式。 double 类型将被视为 float 类型。

有关浮点类型的详细信息，请参阅基本数据类型 - 浮点类型，第 329 页。

使用最佳指针类型

通用指针可以指向所有内存空间，这使得它们简单且易于使用。

但是,它们的代价是在每次指针访问之前需要特殊代码来检查指针指向的内存并采取适当的行动。尽可能使用最小的指针类型，除非必要，否则避免使用任何通用指针。

匿名结构和联合

当一个结构或联合声明没有名字时,它变成匿名的。效果是它的成员只能在周围范围内看到。匿名结构是 C++ 语言的一部分；但是，它们不是 C 标准的一部分。在适用于 8051 的 IAR C/C++ 编译器中，如果启用了语言扩展，它们可以在 C 中使用。

在 IDE 中，默认启用语言扩展。

使用 -e 编译器选项启用语言扩展。有关其他信息，请参见 -e，第 302 页。

例如

在此示例中，可以在函数 F 中访问匿名联合中的成员，而无需显式指定联合名称：

```
struct S
{
char mTag;
union
{
long mL;
float mF;
};
} St;
void F(void)
{
St.mL = 5;
}
```

页码:258

成员名称在周围范围内必须是唯一的。 还允许在文件范围内使用匿名结构或联合，作为全局、外部或静态变量。
例如，这可以用于声明 I/O 寄存器，如下例所示：

```
__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;
}
```

这在地址 0x90 处声明了一个 I/O 寄存器字节 IOPORT。 I/O 寄存器有 2 位声明，way 和 out。 请注意，内部结构和外部联合都是匿名的。
匿名结构和联合是根据以第一个字段命名的对象来实现的，并带有前缀 _A_ 以将名称放在命名空间的实现部分中。 在这个例子中，匿名联合将通过一个名为 _A_IOPORT 的对象来实现。

控制数据和函数内存中的放置

编译器提供了不同的机制来控制函数和数据对象在内存中的放置。 为了有效地使用内存，您应该熟悉这些机制并知道哪一种最适合不同的情况。 您可以使用：

- 代码模型

通过选择代码模型，您可以控制函数的默认内存布局。 有关详细信息，请参阅函数存储的代码模型和内存属性，第 95 页。

● 数据模式

通过选择数据模式，您可以控制变量和常量的默认内存位置。有关详细信息，请参阅数据模式，第 80 页。

● 内存属性

使用 IAR 特定的关键字或 Pragma 预编译指令，您可以覆盖默认寻址模式以及函数和数据对象的默认放置。有关详细信息，请分别参见第 97 页的使用函数存储器属性和第 74 页的使用数据存储器属性。

● 调用约定

编译器提供了六种不同的调用约定来控制内存如何用于参数和本地声明的变量。您可以指定默认调用约定，也可以为每个单独的函数显式声明调用约定。要了解更多信息，请参阅选择调用约定，第 198 页。

● 虚拟寄存器

在较大的数据模式中，微调虚拟寄存器的数量可以产生更高效的代码；参见虚拟寄存器，第 92 页。

● 绝对放置的@ 运算符和#pragma location 指令。

使用@ 运算符或#pragma location 指令，您可以将单独的全局变量和静态变量放置在绝对地址上。请注意，不能将此符号用于各个功能的绝对位置。有关详细信息，请参阅绝对位置的数据放置，第 260 页。

● 用于段放置的@ 运算符和#pragma location 指令。

使用@ 运算符或#pragma location 指令，您可以将单个函数、变量和常量放置在命名段中。然后可以通过链接器指令控制这些段的放置。有关详细信息，请参阅段中的数据和函数放置，第 262 页。

绝对位置的数据放置

@ 运算符，或者#pragma location 指令，可用于将全局变量和静态变量放置在绝对地址。必须使用 const 关键字（带有初始化程序）声明变量。

要将变量放置在绝对地址，@ 运算符和#pragma location 指令的参数应该是一个文字数字，表示实际地址。

注意：放置在绝对地址的所有变量声明都是暂定定义。临时定义的变量只保留在编译器的输出中如果正在编译的模块中需要它们。这些变量将在使用它们的所有模块中定义，只要它们以相同的方式定义，它们就会起作用。

建议将所有此类声明放在包含在所有使用变量的模块中的头文件中。

页码:260

例子

在这个例子中，一个 __no_init 声明的变量被放置在一个绝对地址上。 这对于多个进程、应用程序等之间的接口很有用：

```
__no_init volatile char alpha @ 0xFF20;/* OK */
```

下一个示例包含两个 const 声明的对象。 第一个未初始化，第二个初始化为特定值。 两个对象都放置在 ROM 中。

这对于可从外部接口访问的配置参数很有用。 请注意，在第二种情况下，编译器没有义务实际从变量中读取，因为该值是已知的。

```
#pragma location=0x90
__no_init const int beta; /* OK */
const int gamma @ 0xA0 = 3; /* OK */
```

在第一种情况下，编译器没有初始化值； 该值必须通过其他方式设置。 典型用途是用于将值单独加载到 ROM 的配置，或用于只读的特殊功能寄存器。

这显示了不正确的用法：

```
int delta @ 0xB0; /* Error, neither */
/* "__no_init" nor "const".*/
```

C++ 注意事项

在 C++ 中，模块范围的 const 变量是静态的（模块本地的），而在 C 中它们是全局的。 这意味着每个声明某个 const 变量的模块都将包含一个具有此名称的单独变量。如果您将应用程序与多个包含（通过头文件）的此类模块链接，例如，声明：

```
volatile const __no_init int x @ 0x100; /* Bad in C++ */
```

链接器将报告不止一个变量位于地址 0x100。

为了避免这个问题并使 C 和 C++ 中的过程相同，您应该将这些变量声明为 extern，例如：

```
/* The extern keyword makes x public. */
extern volatile const __no_init int x @ 0x100;
```

注意：C++静态成员变量可以像任何其他静态变量一样放置在绝对地址。

页码:261

分段中的数据和功能放置

以下方法可用于将数据或函数放置在默认以外的命名段中：

● @ 运算符，或者#pragma location 指令，可用于将单个变量或单个函数放置在默认段以外的命名段中。命名段可以是预定义的段，也可以是用户定义的段。

C++ 静态成员变量可以像任何其他静态变量一样放置在命名段中。

如果您使用自己的段，除了预定义的段之外，还必须使用 -Z 或 -P 段控制指令在链接器配置文件中定义这些段。

注意：在将变量或函数显式放置在默认使用的段以外的预定义段中时要小心。这在某些情况下很有用，但不正确的放置可能会导致编译期间的错误消息和链接到故障应用程序。仔细考虑情况；对函数或变量的声明和使用可能有严格的要求。

可以从链接器配置文件控制段的位置。

有关分段的更多信息，请参阅分段参考一章。

在命名段中放置变量的示例

在以下示例中，数据对象放置在用户定义的段中。如果未指定内存属性，则该变量将像任何其他变量一样被视为位于默认内存中。请注意，您必须始终确保在链接时将段放置在适当的内存区域中。

```
__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 */
```

编译器将警告必须手动处理包含零初始化和初始化数据的段。 为此，您必须使用链接器选项 -Q 将初始化程序分隔到一个单独的段中，并将要初始化的符号分隔到不同的段中。 然后，您必须编写将初始化程序段复制到已初始化段的源代码，并且必须在使用零初始化符号之前清除它们。

像往常一样，您可以使用内存属性为变量选择内存。 请注意，您必须始终确保在链接时将段放置在适当的内存区域中。

```
__pdata __no_init int alpha @ "MY_PDATA_NOINIT";  /* 放在 pdata 中*/
```

此例子显示了不正确的用法：

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

将函数放置在命名段中的示例

```
void f(void) @ "MY_FUNCTIONS";
void g(void) @ "MY_FUNCTIONS"
{
}
#pragma location="MY_FUNCTIONS"
void h(void);
```

指定内存属性以将函数定向到特定内存，然后相应地修改链接器配置文件中的段位置：

```
__ void f(void) @ "MY_NEAR_FUNC_FUNCTIONS";
```

控制编译器的优化

编译器对你的应用程序进行了许多转换，以生成尽可能好的代码。这类转换的例子有：将数值存储在寄存器中而不是内存中，删除多余的代码，以更有效的顺序重新排列计算，以及用更便宜的操作代替算术操作。

链接器也应被视为编译系统的一个组成部分，因为有些优化是由链接器执行的。例如，所有未使用的函数和变量都被删除，并且不包括在最终输出中。

执行优化的范围

你可以决定是对你的整个应用程序还是对单个文件进行优化。默认情况下，整个项目使用相同类型的优化，但你应该考虑对单个文件使用不同的优化设置。

例如，把必须快速执行的代码放到一个单独的文件中，并以最小的执行时间来编译它，其余的代码以最小的代码大小来编译。这将得到一个小程序，在重要的地方仍然足够快。

页码:263

您还可以从执行的优化中排除单个功能。 #pragma optimize 指令允许您降低优化级别，或指定要执行的另一种优化类型。有关 Pragma 预编译指令的信息，请参见第 374 页的优化。

多文件编译单元

除了对不同的源文件甚至函数应用不同的优化之外，您还可以决定一个编译单元由什么组成--一个或多个源代码文件。

默认情况下，一个编译单元由一个源文件组成，但您也可以使用多文件编译在一个编译单元中制作多个源文件。优点是内联、交叉调用和交叉跳转等过程间优化有更多的源代码可以处理。

理想情况下，应将整个应用程序编译为一个编译单元。但是，对于大型应用程序，由于主机上的资源限制，这是不切实际的。有关详细信息，请参阅 --mfc，第 308 页。

注意：只生成一个目标文件，因此所有符号都将成为该目标文件的一部分。

如果整个应用程序被编译为一个编译单元，那么让编译器在执行过程间优化之前也丢弃未使用的公共函数和变量是非常有用的。这样做会将优化范围限制为实际使用的函数和变量。有关详细信息，请参阅 --discard_unused_publics，第 299 页。

多文件编译不应该与任何分页代码模型一起使用；请参阅为存储的内存编写源代码，第 111 页。

页码:264

优化级别

编译器支持不同级别的优化。此表列出了通常在每个级别上执行的优化：

优化级别	描述
无（最佳调试支持）	变量贯穿其整个范围 禁用的寄存器组（需要手动启用）
低的	与上面相同，但变量仅在需要时才存在，不一定在其整个范围内，并且： 死码消除 消除冗余标签 冗余分支消除
中等的	与上述相同，并且：活死分析和优化 代码提升 注册内容分析与优化 公共子表达式消除
高（平衡）	与上述相同，并且： 窥孔优化 跨跳 交叉调用（优化大小或平衡时） 循环展开 函数内联 代码运动 基于类型的别名分析

表 31: 编译器优化级别

注意：一些已执行的优化可以被单独启用或禁用。关于这些的更多信息，请参见微调启用的转换，第 266 页。

高水平的优化可能会导致编译时间的增加，而且很可能也会使调试更加困难，因为生成的代码与源代码的关系不太清楚。例如，在低、中、高优化水平下，变量不会贯穿其整个范围，这意味着用于存储变量的处理器寄存器可以在最后一次使用后立即重复使用。由于这个原因，C-SPY 观察窗口可能无法在整个范围内显示变量的值。在任何时候，如果你在调试代码时遇到困难，试着降低优化级别。

速度与大小

在高优化级别，编译器在大小和速度优化之间进行平衡。然而，可以明确地对大小或速度的优化进行微调。

页码:265

它们仅在使用的阈值上有所不同；速度将以尺寸换取速度，而尺寸将以速度换取尺寸。请注意，有时一种优化可以执行其他优化，并且在某些情况下应用程序可能会变得更小，即使在优化速度而不是大小时也是如此。

如果您使用优化级别 High speed，--no_size_constraints 编译器选项会放宽对代码大小扩展的正常限制并启用更积极的优化。

您可以使用命令行选项和 Pragma 预编译指令为每个模块甚至单个函数选择优化目标(参见 -O，第 314 页和优化，第 374 页）。对于小型嵌入式应用程序，这可以在最小化代码大小的同时实现可接受的速度性能：通常，应用程序中只有少数地方需要快速，例如最频繁执行的内部循环或中断处理程序。
一般情况下，您可以使用 High（Size）而不是使用 High（Balanced）优化来编译整个应用程序，但覆盖它以仅针对应用程序需要快速的那些功能获得 High（Speed）优化。
由于不同优化交互的方式不可预测，其中一种优化可以启用其他优化，有时使用高速（高速）优化编译的函数比使用高速（大小）优化时要小。此外，使用多文件编译（参见 --mfc，第 308 页）可以启用许多优化以提高速度和大小性能。建议您尝试不同的优化设置，以便为您的项目选择最佳设置。

微调启用的转换
在每个优化级别，您可以单独禁用一些转换。要禁用转换，请使用适当的选项，例如命令行选项 --no_inline，或者它在 IDE 函数内联中的等效选项，或 #pragma optimize 指令。这些转换可以单独启用/禁用：
- 公共子表达式消除
- 循环展开
- 函数内联
- 代码运动
- 基于类型的别名分析
- 交叉调用
- 禁用的寄存器分页

页码:266

公共子表达式消除

默认情况下，在中等和高优化级别消除了对公共子表达式的冗余重新评估。这种优化通常会减少代码大小和执行时间。但是，生成的代码可能难以调试。

注意：此选项对优化级别 None 和 Low 无效。

有关命令行选项的更多信息，请参见 --no_cse，第 310 页。

循环展开

循环展开意味着循环的代码体（其迭代次数可以在编译时确定）被复制。循环展开通过在多次迭代中摊销循环开销来减少循环开销。

这种优化对于较小的循环最有效，其中循环开销可能是整个循环体的很大一部分。

可以在优化级别 High 执行的循环展开通常会减少执行时间，但会增加代码大小。生成的代码也可能难以调试。

编译器启发式地决定展开哪些循环。只有循环开销减少明显的相对较小的循环才会被展开。在优化速度、大小或在大小和速度之间进行平衡时，会使用不同的启发式方法。

注意：此选项在优化级别无、低和中时无效。

要禁用循环展开，请使用命令行选项 --no_unroll，请参见 --no_unroll，第 313 页。

函数内联

函数内联意味着一个函数，其定义在编译时已知，被集成到其调用者的主体中，以消除调用的开销。这种优化通常会减少执行时间，但可能会增加代码大小。

有关详细信息，请参阅内联函数，第 103 页。

代码运动

移动循环不变表达式和公共子表达式的求值以避免多余的重新求值。这种优化在中等及以上优化级别执行，通常会减少代码大小和执行时间。然而，生成的代码可能难以调试。

注意：此选项对低于中等的优化级别无效。

有关命令行选项的更多信息，请参阅 --no_code_motion，第 309 页。

基于类型的别名分析

当两个或多个指针引用相同的内存位置时，这些指针被称为彼此的别名。别名的存在使优化更加困难，因为在编译时不一定知道特定值是否正在更改。

基于类型的别名分析优化假定对对象的所有访问都是使用其声明的类型或作为 char 类型执行的。这个假设让编译器检测指针是否可以引用相同的内存位置。

基于类型的别名分析在优化级别高执行。对于符合标准 C 或 C++ 应用程序代码的应用程序代码，这种优化可以减少代码大小和执行时间。但是，非标准 C 或 C++ 代码可能会导致编译器生成导致意外行为的代码。因此，可以关闭此优化。

注意：此选项在优化级别无、低和中时无效。

有关命令行选项的更多信息，请参见 --no_tbaa，第 312 页。

例子

```
short F(short *p1, long *p2)
{
*p2 = 0;
*p1 = 1;
return *p2;
}
```

使用基于类型的别名分析，假定对 p1 所指向的短路的写入访问不会影响 p2 指向的长整型值。因此，在编译时知道此函数返回 0。但是，在不符合标准的 C 或 C++ 代码中，这些指针可以通过成为同一联合的一部分而相互重叠。如果使用显式强制转换，还可以强制不同指针类型的指针指向同一内存位置。

交叉调用

将通用代码序列提取到本地子例程。这种优化是在优化级别“高”执行的，可以代表执行时间和堆栈大小来减小代码大小，有时甚至会显著减小代码大小。但是，生成的代码可能难以调试。

有关相关命令行选项的详细信息，请参阅 --no_cross_call，第 309 页。

禁用的寄存器分页

如果应用程序不使用寄存器分页，则可以禁用编译器对它们的支持，并进行进一步优化。默认情况下不会使用此转换，并且在设置适当的优化级别时不会自动使用此转换。
它必须始终显式打开。
禁用寄存器组编译的程序模块不能与使用寄存器组的模块链接在一起。运行时属性__register_banks 可帮助您避免混合此类模块：请参阅第 143 页的预定义运行时属性。
有关命令行选项的信息，请参阅--disable_register_banks，第 299 页。

促进良好的代码生成

本节包含如何帮助编译器生成好代码的提示：

- 编写优化友好的源代码，第 269 页
- 节省堆栈空间和 RAM 内存，第 270 页
- 调用约定，第 270 页
- 功能原型，第 271 页
- 整数类型和位求反，第 272 页
- 保护同时访问的变量，第 272 页
- 访问特殊功能寄存器，第 273 页
- 未初始化变量，第 274 页

编写优化友好的源代码

下面列出了一些编程技术，遵循这些技术可以使编译器更好地优化应用程序。

- 局部变量自动变量和参数优先于静态或全局变量。原因是优化器必须假设，例如，被调用的函数可以修改非局部变量。当局部变量的生命周期结束时，可以重用以前占用的内存。全局声明的变量将在整个程序执行期间占用数据内存。
- 避免使用&运算符获取局部变量的地址。这是低效的，主要有两个原因。首先，变量必须放在内存中，因此不能放在处理器寄存器中。这会导致更大和更慢的代码。其次，优化器不能再假设局部变量不受函数调用的影响。

页码:269

●模块-局部变量-声明为静态的变量比全局变量（非静态变量）更可取。还要避免使用经常访问的静态变量的地址。
●该编译器能够内联函数，请参见函数内联，第 267 页。为了最大限度地提高内联转换的效果，最好将从多个模块调用的小函数的定义放在头文件中，而不是放在实现文件中。或者，您也可以使用多文件编译。有关详细信息，请参见多文件编译单元，第 264 页。
●避免使用内联汇编程序。相反，尝试用 C/C++编写代码，使用内在函数，或者用汇编语言编写一个单独的模块。有关更多信息，请参见混合 C 和汇编器，第 191 页。
节省了堆栈空间和 RAM 内存
以下是一个节省内存和堆栈空间的编程技术列表：
●如果堆栈空间有限，则避免长调用链和递归函数。
●避免使用大型非标量类型，如结构，作为参数或返回类型。
为了保存堆栈空间，您应该将它们作为指针或在 C++中作为引用传递。
●使用尽可能小的数据类型（只有在必要时才使用签名数据类型）
●将使用寿命较短的变量声明为自动变量。当这些变量的寿命结束时，就可以重用以前占用的内存。全局声明的变量将在整个程序执行过程中占用数据内存。
但是，要小心自动变量，因为堆栈的大小可能会超过其限制。

调用约定

该编译器支持使用不同类型的堆栈的几种调用约定。尝试使用尽可能小的调用约定。数据覆盖、idata 覆盖和 idata 重入调用约定生成最有效的代码。Pdata 重入和扩展堆栈重入函数增加了一些开销，xdata 重入函数甚至多。
由于 xdata 堆栈指针和扩展堆栈指针大于 8 位，因此必须使用两个指令来更新它们。为了使系统中断安全，必须在更新堆栈指针时禁用中断。如果您正在使用 xdata 或扩展堆栈，则会产生开销。

页码:270

通常，使用默认调用约定就足够了。但是，在某些情况下，最好显式声明另一个调用约定的函数，例如：

- 一些大型和堆栈密集型函数不适合较小调用约定的有限限制。然后可以将该函数声明为具有更大的调用约定
- 使用有限数量的小而重要的例程的大型系统可以使用更小的调用约定来声明这些以提高效率。

注意：当您混合不同的调用约定时，会有一些限制。请参阅混合调用约定，第 86 页。

功能原型

可以使用两种不同样式中的一种来声明和定义函数：

- 原型
- Kernighan & Ritchie C (K&R C)

两种样式都是有效的 C，但是强烈建议使用原型样式，并在包含在定义函数的编译单元和所有使用它的编译单元中的头文件中为每个公共函数提供原型声明。

编译器不会对传递给使用 K&R 样式声明的函数的参数执行类型检查。在某些情况下，使用原型声明也会产生更高效的代码，因为这些函数不需要类型提升。

要使编译器要求所有函数定义都使用原型样式，并且所有公共函数在定义之前都已声明，请使用

Project>Options>C/C++ Compiler>Language 1>Require prototypes compiler option (--require_prototypes).

原型风格

在原型函数声明中，必须指定每个参数的类型。

```
int Test(char, int); /* Declaration */
int Test(char ch, int i) /* Definition */
{
return i + ch;
}
```

Kernighan & Ritchie 风格

在 K&R 风格（标准 C 之前）中，不可能将函数声明为原型。

相反，在函数声明中使用了一个空参数列表。 此外，定义看起来不同。

例如：

```
int Test(); /* Declaration */
int Test(ch, i) /* Definition */
char ch;
int i;
{
  return i + ch;
}
```

整数类型和位求反

在某些情况下，整数类型的规则及其转换可能会导致混乱的行为。要注意的是涉及不同大小的类型的赋值或条件（测试表达式），以及逻辑运算，特别是位否定。这里，类型还包括常量的类型。

在某些情况下，可能会有警告（例如，对于常量条件或无意义的比较），在其他情况下，只是一个与预期不同的结果。在某些情况下，编译器可能仅在较高的优化级别发出警告，例如，如果编译器依赖优化来识别常量条件的某些实例。在此示例中，假设为 8 位字符、16 位整数和 2 的补码：

```
void F1(unsigned char c1)
{
if (c1 == ~0x80)
;
}
```

在这里，这个测试总是错误的。在右边，0x80 是 0x0080，而~0x0080 变成了 0xFF7F。在左边，c1 是一个 8 位的无符号字符，所以它不能大于 255。它也不能是负的，这意味着积分提升值永远不能设置最上面的 8 位。

保护同时访问的变量

异步访问的变量，例如通过中断例程或在单独线程中执行的代码，必须正确标记并具有足够的保护。

唯一的例外是一个总是只读的变量。

要正确地标记一个变量，请使用挥发性关键字。这告诉编译器可以从其他线程更改变量。编译器将避免对变量进行优化（例如，在寄存器中跟踪变量），不会延迟对它的写入，并且只按照源代码中给出的次数仔细访问变量。

对于不希望中断的变量访问序列，请使用__monitor 关键字。必须对写入和读取序列都执行此操作，否则最终可能会读取部分更新的变量。访问小尺寸的 volatile 变量可能是一个原子操作，但除非您不断研究编译器的输出，否则不应该依赖它。使用__monitor 关键字来确保序列是原子操作更安全。有关更多信息，请参阅__monitor，第 354 页。

有关 volatile 类型限定符和访问 volatile 对象的规则的更多信息，请参阅声明对象 volatile，第 334 页。

保护 eeprom 写入机制

需要使用__monitor 关键字的一个典型示例是保护 eeprom 写入机制，该机制可从两个线程（例如，主代码和中断）使用。在 EEPROM 写入序列期间维护中断在许多情况下会损坏写入的数据。

访问特殊功能寄存器

IAR 产品安装中包含多个 8051 设备的特定头文件。头文件名为 iodevice.h。定义处理器特定的特殊功能寄存器（SFR）。

注意：每个头文件包含编译器使用的一个部分和汇编程序使用的一个部分。

带有位域的 SFR 在头文件中声明。 这个例子来自 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;
```

/* 通过在代码中包含适当的包含文件，可以从 C 代码访问整个寄存器或任何单个位（或位域），如下所示。 */

```
void Test()
{
/* Whole register access */
PSW = 0x12;
/* Bitfield accesses */
PSW_bit.AC = 1;
PSW_bit.RS0 = 0;
}
```

当您为其他 8051 设备创建新的头文件时，您也可以将头文件用作模板。 有关 @ 运算符的信息，请参阅放置定位的数据，第 132 页。

未初始化的变量

通常，运行时环境会在应用程序启动时初始化所有全局变量和静态变量。

编译器支持声明不会被初始化的变量，使用 __no_init 类型修饰符。 它们可以指定为关键字或使用#pragma object_attribute 指令。 编译器根据指定的内存关键字将这些变量放在一个单独的段中。 有关详细信息，请参阅链接概述一章。

页码:274

对于__no_init，const 关键字意味着对象是只读的，而不是对象存储在只读内存中。不可能给__no_init 对象赋予初始值。

例如，使用__no_init 关键字声明的变量可以是大的输入缓冲区，或者映射到特殊的 RAM，即使在应用程序关闭时也能保持其内容。

更多信息，请参见__no_init，第 356 页。请注意，要使用该关键字，必须启用语言扩展。参见 e，第 302 页。更多信息，请参见第 373 页的对象属性。

页码:275

（空白页）
页码:276

第二部分。参考信息

这一部分包含 IAR C/C + + 8051 编译器用户指南的以下章节:

- 外部接口细节
- 编译器选项
- 数据表示
- 扩展关键字
- 实用指令
- 内部函数
- 预处理器
- C/C++ 标准库函数
- 分段参考
- 标准 C 的实现定义行为
- C89 的实现定义行为。

页码:277

（空白页）
页码:278

外部接口细节

- 调用语法
- 包含文件搜索程序
- 编译器输出
- 诊断

调用语法

您可以从 IDE 或命令行使用编译器。有关从 IDE 使用编译器的信息，请参阅 8051 的 IDE 项目管理和构建指南。

编译器调用语法

编译器的调用语法是：

icc8051 [选项] [源文件] [选项]

例如，在编译源文件 prog.c 时，使用此命令生成带有调试信息的目标文件：

icc8051 prog.c --debug

源文件可以是 C 或 C++ 文件，通常分别具有文件扩展名 c 或 cpp。如果未指定文件扩展名，则要编译的文件必须具有扩展名 c。

通常，命令行上选项的顺序，无论是相对于彼此还是相对于源文件名，都不重要。但是，有一个例外：当您使用 -I 选项时，搜索目录的顺序与它们在命令行中指定的顺序相同。

如果您从命令行不带任何参数运行编译器,则编译器版本号和所有可用选项(包括简要说明)将定向到标准输出并显示在屏幕上。

通过选项

向编译器传递选项有三种不同的方式：

●直接从命令行

指定 icc8051 命令后命令行上的选项,无论是在源文件名之前还是在源文件名之后;参见《调用语法》，第 279 页。

●通过环境变量

编译器自动将环境变量的值附加到每个命令行中；参见环境变量，第 280 页.

●通过文本文件，使用-f 选项；见-f，第 305 页。

对于选项语法的一般指导原则，进行了选项总结，并对每个选项进行了详细的描述，参见编译器选项章节。

环境变量

这些环境变量可以与编译器一起使用：

环境变量	描述
C_INCLUDE	指定目录来搜索包含文件； 例如：C_INCLUDE = C:\program files\iar systems\embedded workbench 8.n\8051\inc;C:\headers
QCCX51	指定命令行选项； 例如：QCCX51=-lA asm.lst

表 32：编译器环境变量

包括文件搜索过程

这是编译器#include 文件搜索过程的详细描述：

● 如果#include 文件的名称是尖括号或双引号中指定的绝对路径，则打开该文件。

● 如果编译器在尖括号中遇到#include 文件的名称，例如：

#include <stdio.h>

它在这些目录中搜索要包含的文件：

1 使用 -I 选项指定的目录，按照它们指定的顺序，请参见 -I，第 306 页。

2 使用 C_INCLUDE 环境变量指定的目录，如果有的话； 请参见环境变量，第 280 页。

3 自动建立的图书馆系统包括目录。 请参见 --clib，第 292 页、--dlib，第 300 页和 --dlib_config，第 300 页。

● 如果编译器在双引号中遇到 # include 文件的名称，例如:

#include "vars.h"
它搜索出现#INCLUDE 语句的源文件的目录，然后执行与尖括号文件名相同的顺序。
如果有嵌套的#INCLUDE 文件，则编译器开始搜索最后包含的文件的目录，向上迭代每个包含的文件，最后搜索源文件目录。例如：
src.c in directory dir\src
#include "src.h"
...
src.h in directory dir\include
#include "config.h"
...

当 dir\ exe 是当前目录时，使用此命令进行编译：
icc8051 ..\src\src.c -I..\include -I..\debugconfig
然后按照下面列出的顺序在以下目录中搜索文件
config.h，在本例中位于 dir\debugconfig 目录中：
dir\include　　当前文件是 src.h。
dir\src　　文件包括当前文件 (src.c)。
dir\include　　与第一个 -I 选项指定的一样。
dir\debugconfig 与第二个 -I 选项指定的一样。
对标准头文件（如 stdio.h）使用尖括号，对属于应用程序的文件使用双引号。
注意：\ 和 / 都可以用作目录分隔符。
有关详细信息，请参阅预处理器概述，第 387 页。

编译器输出
编译器可以产生以下输出：
● 可链接的目标文件
编译器生成的目标文件使用一种称为 UBROF 的专有格式，它代表通用二进制可重定位目标格式。 默认情况下，目标文件的文件扩展名为 r51。

● 可选列表文件

可以使用编译器选项-l，请参阅-l，第 306 页指定各种列表文件。默认情况下，这些文件的文件名扩展名为 lst。

● 可选的预处理器输出文件

使用 -- preprocess 选项时会生成预处理器输出文件；默认情况下，该文件的扩展名为 i。

● 诊断信息

诊断消息被定向到标准错误流并显示在屏幕上，并打印在可选的列表文件中。有关诊断消息的详细信息，请参阅诊断，第 283 页。

● 错误返回代码

这些代码向操作系统提供状态信息，可以在批处理文件中进行测试，请参阅错误返回代码，第 282 页。

● 尺寸信息

有关每个内存的函数和数据的生成字节量的信息将定向到标准输出流并显示在屏幕上。一些字节可能被报告为共享的。

共享对象是模块之间共享的函数或数据对象。如果其中任何一个发生在多个模块中，则仅保留一个副本。例如，在某些情况下，内联函数没有内联，这意味着它们被标记为共享，因为最终应用程序中只会包含每个函数的一个实例。

这种机制有时也用于编译器生成的代码或与特定函数或变量不直接关联的数据，并且在最终应用程序中只需要一个实例时。

错误返回代码

编译器将可以在批处理文件中测试的状态信息返回给操作系统。

支持这些命令行错误代码:

代码	描述
0	编译成功，但可能已出现警告。
1	生成了警告， 并使用了选项--WARNINGS_EFECT_EXIT_CODE。
2	发生错误。
3	发生致命错误，导致编译器中止。
4	发生内部错误，导致编译器中止。

表 33：错误返回代码

页码:282

诊断

本节介绍诊断消息的格式，并说明如何将诊断消息划分为不同的严重性级别。

消息格式

所有诊断消息都以完整、不言自明的消息形式发布。来自编译器的典型诊断消息的格式如下：

filename,linenumber level[tag]: message

包括这些元素：

filename	遇到问题的源文件的名称
linenumber	编译器检测到问题的行号
level	问题的严重性程度
tag	标识诊断消息的唯一标记
message	一个解释，可能有几行长

诊断消息显示在屏幕上，并打印在可选列表文件中。

使用选项 --diagnostics_tables 列出所有可能的编译器诊断消息。

严重程度

诊断消息分为不同的严重级别：

评论

当编译器发现可能导致生成代码中的错误行为的源代码构造时生成的诊断消息。 备注默认情况下不发布，但可以启用，请参见 --remarks，第 319 页。

警告

当编译器发现潜在的编程错误或遗漏时产生的诊断消息，但不妨碍编译的完成。 可以使用命令行选项禁用警告

--no_warnings，参见 --no_warnings，第 313 页。

页码:283

错误

当编译器发现明显违反 C 或 C++ 语言规则的构造时生成的诊断消息，因此无法生成代码。错误将产生非零退出代码。

致命错误

当编译器发现不仅阻止代码生成而且使源代码的进一步处理变得毫无意义的条件时生成的诊断消息。消息发出后，编译终止。致命错误将产生非零退出代码。

设置严重级别

可以抑制诊断消息或更改所有诊断消息的严重性级别，但致命错误和一些常规错误除外。

有关可用于设置严重性级别的编译器选项的信息，请参阅编译器选项一章。

有关可用于设置严重性级别的 Pragma 预编译指令的信息,请参阅 Pragma 预编译指令一章。

内部错误

内部错误是一条诊断消息，表示由于编译器中的故障而导致出现严重的意外故障。它是使用这种形式产生的：

内部错误：消息，其中消息是解释性消息。如果发生内部错误，应将其报告给您的软件分销商或 IAR Systems 技术支持。包括足够的信息来重现问题，通常：

- 产品名称
- 编译器的版本号，可以在编译器生成的列表文件的头部看到
- 您的许可证号
- 确切的内部错误消息文本
- 产生内部错误的应用程序源文件
- 发生内部错误时使用的选项列表。

编译器选项

- 选项语法
- 编译器选项摘要
- 编译器选项说明

选项语法

编译器选项是您可以指定以更改编译器默认行为的参数。您可以从命令行（在本节中更详细地描述）以及从 IDE 中指定选项。

有关 IDE 中可用的编译器选项以及如何设置它们的信息，请参阅在线帮助系统。

选项类型

命令行选项有两种类型的名称，短名称和长名称。

有些选项两者兼而有之。

- 短选项名称由一个字符组成，可以有参数。您可以使用单个破折号指定它，例如 -e
- 长选项名称由一个或多个下划线连接的单词组成，可以有参数。例如，您可以用双破折号指定它

--char_is_signed。

有关传递选项的不同方法的信息，请参阅传递选项，第 280 页。

指定参数的规则

有一些用于指定选项参数的通用语法规则。首先，描述取决于参数是可选的还是强制的，以及选项是短名称还是长名称的规则。然后，列出指定文件名和目录的规则。最后，列出其余规则。

可选参数的规则

对于具有短名称和可选参数的选项，任何参数都应在前面不带空格的情况下指定，例如：

-O or -Oh

对于具有长名称和可选参数的选项，任何参数都应使用前面的等号 (=) 指定，例如：
--misrac2004=*n*

强制参数的规则
对于具有短名称和必选参数的选项，可以使用或不使用前导空格来指定参数，例如：
-I..\src or -I ..\src\

对于具有长名称和强制参数的选项，参数可以用前一个等号(=)或前一个空间来指定，例如：
--diagnostics_tables=MyDiagnostics.lst
或
--diagnostics_tables MyDiagnostics.lst

带有可选参数和强制参数的选项规则
对于同时采用可选参数和强制参数的选项，指定参数的规则是：
● 对于短选项，指定可选参数，前面不带空格
● 对于长选项，可选参数以前面的等号 (=) 指定
● 对于短选项和长选项，必须指定参数，前面带有空格。
例如，带有可选参数后跟强制参数的短选项：
-lA MyList.lst

例如，带有可选参数后跟强制性参数的长选项:

--preprocess=n PreprocOutput.lst

将文件名或目录指定为参数的规则
这些规则适用于将文件名或目录作为参数的选项：
● 采用文件名作为参数的选项可以选择采用文件路径。 路径可以是相对的或绝对的。 例如，要为目录中的文件 List.lst 生成一个列表..\listings\:
icc8051 prog.c -l ..\listings\List.lst

● 对于将文件名作为输出目标的选项，可以将参数指定为不指定文件名的路径。 编译器将输出存储在该目录中，并根据选项存储在具有扩展名的文件中。 文件名将与编译的源文件的名称相同，除非使用选项 -o 指定了不同的名称，在这种情况下使用该名称。 例如：
icc8051 prog.c -l ..\listings\
生成的列表文件将具有默认名称 ..\listings\prog.lst
● 当前目录用句点(.) 指定。 例如：
icc8051 prog.c -l.
● 可以使用/ 代替\ 作为目录分隔符。
● 通过指定-，输入文件和输出文件可以分别重定向到标准输入和输出流。 例如：
icc8051 prog.c -l -

附加规则
这些规则也适用：
相反，您可以在参数前面加上两个破折号； 此示例将创建一个名为 -r 的列表文件：
icc8051 prog.c -l ---r

● 对于接受多个相同类型参数的选项，参数可以以逗号分隔列表（不带空格）的形式提供，例如：
--diag_warning=Be0001,Be0002
或者，可以为每个参数重复该选项，例如：
--diag_warning=Be0001
--diag_warning=Be0002

编译器选项汇总

下表总结了编译器命令行选项：

命令行选项	描述	描述(英文)
--c89	指定 C89 方言	Specifies the C89 dialect
--calling_convention	指定调用约定	Specifies the calling convention
--char_is_signed	将 char 视为已签名	Treats char as signed
--char_is_unsigned	将 char 视为无符号	Treats char as unsigned
--clib	使用 CLIB 库的系统包含文件	Uses the system include files for the CLIB library
--code_model	指定代码模型	Specifies the code model
--core	指定 CPU 内核	Specifies a CPU core
-D	定义预处理器符号	Defines preprocessor symbols
--data_model	指定数据模型	Specifies the data model
--debug	生成调试信息	Generates debug information
--dependencies	列出文件依赖项	Lists file dependencies
--diag_error	将这些视为错误	Treats these as errors
--diag_remark	将这些视为备注	Treats these as remarks

--diag_suppress	禁止这些诊断	Suppresses these diagnostics
--diag_warning	将这些视为警告	Treats these as warnings
--diagnostics_tables	列出所有诊断消息	Lists all diagnostic messages
--disable_register_banks	禁用编译器对寄存器组的使用	Disables the compiler's use of register banks
--discard_unused_publics	丢弃未使用的公共符号	Discards unused public symbols
--dlib	使用 DLIB 库的系统包含文件	Uses the system include files for the DLIB library
--dlib_config	使用 DLIB 库的系统包含文件并确定要使用的库配置	Uses the system include files for the DLIB library and determines which configuration of the library to use
--dptr	启用对多数据指针的支持	Enables support for multiple data pointer
-e	启用语言扩展	Enables language extensions
--ec++	指定嵌入式 C++	Specifies Embedded C++
--eec++	指定扩展嵌入式 C++	Specifies Extended Embedded C++
--enable_multibytes	启用对源文件中的多字节字符的支持	Enables support for multibyte characters in source files
--enable_restrict	启用标准 C 关键字限制	Enables the Standard C keyword restrict
--error_limit	指定编译停止前允许的错误数	Specifies the allowed number of errors before compilation stops
--extended_stack	指定使用扩展堆栈	Specifies the use of an extended stack
-f	扩展命令行	Extends the command line
--guard_calls	为函数静态变量初始化启用保护	Enables guards for function static variable initialization
--has_cobank	通知编译器设备在分页选择寄存器中有用于常量的 COBANK 位	Informs the compiler that the device has COBANK bits in the bank selection register for constants
--header_context	列出所有引用的源文件和头文件	Lists all referred source files and header files
-I	指定包含文件路径	Specifies include file path
-l	创建一个列表文件	Creates a list file
--library_module	创建一个库模块	Creates a library module
--macro_positions_in_diagnostics	获取诊断消息中宏内部的位置	Obtains positions inside macros in diagnostic messages
--mfc	启用多文件编译	Enables multi-file compilation

--misrac1998	启用特定于 MISRA-C:1998 的错误消息。 请参阅 IAR Embedded Workbench MISRA C:1998 参考指南。	Enables error messages specific to MISRA-C:1998. See the IAR Embedded Workbench MISRA C:1998 Reference Guide.
--misrac2004	启用特定于 MISRA-C:2004 的错误消息。 请参阅 IAR Embedded Workbench MISRA C:2004 参考指南。	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 参考指南或 IAR Embedded Workbench MISRA C:2004 参考指南。	IAR Embedded Workbench® MISRA C:1998 Reference Guide or the IAR Embedded Workbench MISRA C:2004 Reference Guide.
--module_name	设置对象模块名称	Sets the object module name
--no_call_frame_info	禁用调用帧信息的输出	Disables output of call frame information
--no_code_motion	禁用代码运动优化	Disables code motion optimization
--no_cross_call	禁用交叉调用优化	Disables cross-call optimization
--no_cse	禁用公共子表达式消除	Disables common subexpression elimination
--no_inline	禁用函数内联	Disables function inlining
--no_path_in_file_macros	从符号 __FILE__ 和 __BASE_FILE__ 的返回值中删除路径	Removes the path from the return value of the symbols __FILE__ and __BASE_FILE__
--no_size_constraints	在优化速度时放宽代码大小扩展的正常限制。	Relaxes the normal restrictions for code size expansion when optimizing for speed.
--no_static_destruction	在程序退出时禁用 C++ 静态变量的破坏	Disables destruction of C++ static variables at program exit
--no_system_include	禁用系统包含文件的自动搜索	Disables the automatic search for system include files
--no_tbaa	禁用基于类型的别名分析	Disables type-based alias analysis
--no_typedefs_in_diagnostics	禁用在诊断中使用 typedef 名称	Disables the use of typedef names in diagnostics
--no_ubrof_messages	从 UBROF 文件中排除消息	Excludes messages from UBROF files
--no_unroll	禁用循环展开	Disables loop unrolling
--no_warnings	禁用所有警告	Disables all warnings
--no_wrap_diagnostics	禁用诊断消息的包装	Disables wrapping of diagnostic messages
--nr_virtual_regs	设置工作区大小	Sets the work area size
-O	设置优化级别	Sets the optimization level
-o	设置对象文件名。 --output 的别	Sets the object filename. Alias for

	名。	--output.
--omit_types	排除类型信息	Excludes type information
--only_stdout	仅使用标准输出	Uses standard output only
--output	设置对象文件名	Sets the object filename
--pending_instantiations	设置给定 C++ 模板的最大实例化数	Sets the maximum number of instantiations of a given C++ template.
--place_constants	指定常量和字符串的位置	Specifies the location of constants and strings
--predef_macros	列出预定义的符号。	Lists the predefined symbols.
--preinclude	在读取源文件之前包含一个包含文件	Includes an include file before reading the source file
--preprocess	生成预处理器输出	Generates preprocessor output
--public_equ	定义一个全局命名的汇编器标签	Defines a global named assembler label
-r	生成调试信息。 --debug 的别名。	Generates debug information. Alias for --debug.
--relaxed_fp	放宽优化浮点表达式的规则	Relaxes the rules for optimizing floating-point expressions
--remarks	启用备注	Enables remarks
--require_prototypes	验证函数在定义之前是否已声明	Verifies that functions are declared before they are defined
--rom_mon_bp_padding	使用通用 ROM 监视器进行调试时，可以在所有 C 语句上设置断点	Enables setting breakpoints on all C statements when debugging using the generic ROM-monitor
--silent	设置静音操作	Sets silent operation
--strict	检查是否严格遵守标准 C/C++	Checks for strict compliance with Standard C/C++
--system_include_dir	指定系统包含文件的路径	Specifies the path for system include files
--use_c++_inline	在 C99 中使用 C++ 内联语义	Uses C++ inline semantics in C99
--version	将编译器输出发送到控制台，然后退出	Sends compiler output to the console and then exits.
--vla	启用 C99 VLA 支持	Enables C99 VLA support
--warn_about_c_style_casts	在 C++ 源代码中使用 C 样式强制转换时使编译器发出警告	Makes the compiler warn when C-style casts are used in C++ source code
--warnings_affect_exit_code	警告影响退出代码	Warnings affect exit code
--warnings_are_errors	警告被视为错误	Warnings are treated as errors

表 34：编译器选项汇总

编译器选项说明

以下部分提供了有关每个编译器选项的详细参考信息。
请注意，如果您使用选项页面额外选项来指定特定的命令行选项，IDE 不会立即检查一致性问题，例如选项冲突、选项重复或使用不相关选项。

--c89

语法：

--c89
描述：
使用此选项可启用 C89 C 方言而不是标准 C。
备注：
当启用 MISRA C 检查时，此选项是必需的。
另见：
C 语言概述，第 221 页。
Project>Options>C/C++ Compiler>Language 1>C dialect>C89

--calling_convention

语法：

--calling_convention=convention
参数：
公约是以下之一：
data_overlay|do idata_overlay|io idata_reentrant|ir pdata_reentrant|pr xdata_reentrant|xr ext_stack_reentrant|er
描述：
使用此选项指定模块的默认调用约定。应用程序中的所有运行时模块必须使用相同的调用约定。但是，请注意，可以通过使用关键字来覆盖单个函数。
另见：
调用约定，第 197 页。
Project>Options>General Options>Target>Calling convention

--char_is_signed

语法：

--char_is_signed
描述：
默认情况下，编译器将纯字符类型解释为无符号。使用此选项可使编译器将纯字符类型解释为有符号。例如，当您想要保持与另一个编译器的兼容性时，这可能很有用。
备注：
运行时库是在没有 --char_is_signed 选项的情况下编译的，并且不能与使用此选项编译的代码一起使用。
Project>Options>C/C++ Compiler>Language 2>Plain 'char' is

--char_is_unsigned

语法：

--char_is_unsigned
描述：
使用此选项可使编译器将纯字符类型解释为无符号。这是普通字符类型的默认解释。
Project>Options>C/C++ Compiler>Language 2>Plain 'char' is

--clib

语法：

--clib
描述：
使用此选项可使用 CLIB 库的系统头文件；编译器会自动定位文件并在编译时使用它们。
备注：

CLIB 库默认用于 CLIB 项目。要使用 DLIB 库，请改用 --dlib 或 --dlib_config

选项。

另见：
--dlib，第 300 页和 --dlib_config，第 300 页。
要设置相关选项，请选择：
Project>Options>General Options>Library Configuration

--code_model

语法：

--code_model={near|n|banked|b|banked_ext2|b2|far|f}
参数：
near|n 最多允许 64 KB 的 ROM；核心变体 Plain 的默认值。
banked|b 允许通过多达 16 个 64 KB 的存储库和一个根存储库提供多达 1 MB 的 ROM；支持存储的 24 位调用。
banked_ext2|b2 允许通过多达 16 个 1-MB 存储区实现多达 16 MB 的 ROM；支持存储的 24 位调用。核心变体 Extended2 的默认设置。
far|f 最多允许 16 MB 的 ROM 并支持真正的 24 位调用。核心变体 Extended1 的默认值。
描述：
使用此选项可选择为其生成代码的代码模式。如果不选择代码模式，编译器将使用默认代码模式。请注意，应用程序的所有模块必须使用相同的代码模式。
另见：
函数存储的代码模式和内存属性，第 95 页。
Project>Options>General Options>Target>Code model

--core

语法：

--core={plain|pl|extended1|e1|extended2|e2}
参数：
plain|pl 对应于具有 64 KB 地址区域的 ROM 的经典 8051 内核或具有扩展代码存储器的经典 8051/8052 内核，最多可增加 256 个附加 ROM 组。
extended1|e1 对应于具有高达 16 MB 外部连续数据和代码存储器的内核。

extended2|e2 对应于具有扩展寻址机制的内核，该机制可以将代码存储器扩展至多达 16 个 64 KB 的存储区。
描述：
使用此选项选择将为其生成代码的处理器内核。如果您不使用该选项指定内核，则编译器默认使用普通内核。请注意，您的应用程序的所有模块必须使用相同的内核。编译器支持不同的 8051 微控制器内核和基于这些内核的设备。编译器为不同内核生成的目标代码不是二进制兼容的。
另见：
了解内存架构，第 59 页。
Project>Options>General Options>Target>Core

-D

语法：

-D symbol[=value]
参数：
symbol 预处理器符号的名称
value 预处理器符号的值
描述：
使用此选项定义预处理器符号。如果未指定值，则使用 1。此选项可以在命令行上使用一次或多次。
选项 -D 与源文件顶部的 #define 语句具有相同的效果：
-Dsymbol
相当于：
#define symbol 1
得到等价的：
#define FOO
指定 = 符号，但后面没有，例如：
-DFOO=
Project>Options>C/C++ Compiler>Preprocessor>Defined symbols

--data_model

语法：

--data_model=
{tiny|t|small|s|large|l|far|f|far_generic|fg|generic|g}
参数：
tiny|t 默认内存属性 __data
默认指针属性 __idata
small|s 默认内存属性 __idata 默认指针属性 __idata 核心变体 Plain 的默认值
large|l 默认内存属性 __xdata 默认指针属性 __xdata 核心变体 Extended2 的默认值
far|f 默认内存属性 __far 默认指针属性 __far 核心变体 Extended1 的默认值
far_generic|fg 默认内存属性 __far22
默认指针属性 __generic
generic|g 默认内存属性 __xdata
默认指针属性 __generic
描述：
使用此选项可选择数据模式，这意味着数据对象的默认放置。如果不选择数据模式，编译器将使用默认数据模式。请注意，应用程序的所有模块都必须使用相同的数据模式。
另见：
数据模式，第 80 页。
Project>Options>General Options>Target>Data model

--debug, -r

语法：

--debug
-r
描述：
使用 --debug 或 -r 选项使编译器在 IAR C-SPY® 调试器和其他符号调试器所需的目标模块中包含信息。
备注：
包含调试信息将使目标文件比其他文件更大。
Project>Options>C/C++ Compiler>Output>Generate debug information

--dependencies

语法：

--dependencies[=[i|m|n][s]] {filename|directory|+}
参数：
i（默认）仅列出文件名
●makefile 风格的列表（多规则）
●makefile 风格的列表（一条规则）
s 抑制系统文件
+ 提供与 -o 相同的输出，但文件扩展名为 d
另见：
将文件名或目录指定为参数的规则，第 286 页。
描述：
使用此选项可以使编译器列出所有打开的源文件和头文件的名称，以便输入到具有默认文件扩展名 i 的文件中。
示例 如果使用 --dependencies 或 --dependencies=i，则每个打开的输入文件的名称，包括完整路径（如果可用）将在单独的行上输出。例如：
c:\iar\product\include\stdio.h d:\myproject\include\foo.h
如果使用 --dependencies=m，则输出为 makefile 样式。对于每个输入文件，都会生成包含 makefile 依赖规则的一行。每行由目标文件的名称、冒号、空格和输入文件的名称组成。例如： foo.r51: c:\iar\product\include\stdio.h foo.r51: d:\myproject\include\foo.h
将 --dependencies 与流行的 make 实用程序一起使用的示例，例如 gmake (GNU make)：
1 将编译文件的规则设置为：
%.r51 : %.c
$(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
也就是说，除了生成目标文件之外，该命令还生成 makefile 样式的依赖文件（在本示例中，使用扩展名 .d）。
2 在 makefile 中包含所有依赖文件，例如：
-include $(sources:.c=.d)
由于破折号 (-)，当 .d 文件尚不存在时，它第一次工作。此选项在 IDE 中不可用。

--diag_error

语法：

--diag_error=tag[,tag,...]
参数：
tag 诊断消息的编号，例如消息编号 Pe117

描述：
使用此选项可将某些诊断消息重新分类为错误。错误表示违反了 C 或 C++ 语言规则，其严重程度将不会生成目标代码。退出代码将非零。此选项可以在命令行上多次使用。
Project>Options>C/C++ Compiler>Diagnostics>Treat these as errors

--diag_remark

语法：

--diag_remark=tag[,tag,...]
参数：
tag 诊断消息的编号，例如消息编号 Pe177
描述：
使用此选项可将某些诊断消息重新分类为备注。备注是最不严重的诊断消息类型，表示可能导致生成代码出现奇怪行为的源代码构造。此选项可以在命令行上多次使用。
备注：
默认不显示备注；使用 --remarks 选项来显示它们。
Project>Options>C/C++ Compiler>Diagnostics>Treat these as remarks

--diag_suppress

语法：

--diag_suppress=tag[,tag,...]
参数：
tag 诊断消息的编号，例如消息编号 Pe117
描述：
使用此选项可抑制某些诊断消息。这些消息将不会显示。此选项可以在命令行上多次使用。
Project>Options>C/C++ Compiler>Diagnostics>Suppress these diagnostics

--diag_warning

语法：

--diag_warning=tag[,tag,...]
参数：
tag 诊断消息的编号，例如消息编号 Pe826
描述：
使用此选项可将某些诊断消息重新分类为警告。警告表示关注的错误或遗漏，但不会导致编译器在编译完成之前停止。此选项可以在命令行上多次使用。
Project>Options>C/C++ Compiler>Diagnostics>Treat these as warnings

--diagnostics_tables

语法：

--diagnostics_tables {filename|directory}
参数：
请参见将文件名或目录指定为参数的规则，第 286 页。
描述：
使用此选项可将所有可能的诊断消息列出到命名文件。这可能很方便，例如，如果您使用 pragma 指令来抑制或更改任何诊断消息的严重级别，但忘记记录原因。通常，此选项不能与其他选项一起提供。
To set this option, use Project>Options>C/C++ Compiler>Extra Options.

--disable_register_banks

语法：

--disable_register_banks
描述：
使用此选项可禁用编译器对寄存器组的使用。
另见：
禁用的寄存器分页，第 269 页。

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Disabled register banks

--discard_unused_publics

语法：

--discard_unused_publics
描述：
使用 --mfc 编译器选项编译时，使用此选项丢弃未使用的公共函数和变量。
备注：
不要仅在应用程序的某些部分上使用此选项，因为必要的符号可能会从生成的输出中删除。使用对象属性根来保留从编译单元外部使用的符号，例如中断处理程序。如果符号没有 __root 属性并且在库中定义，则将使用库定义。
另见：
--mfc，第 308 页和多文件编译单元，第 264 页。
Project>Options>C/C++ Compiler>Discard unused publics

--dlib

语法：

--dlib
描述：
使用此选项可使用 DLIB 库的系统头文件；编译器会自动定位文件并在编译时使用它们。
备注：
默认使用 DLIB 库：要使用 CLIB 库，请改用 --clib 选项。
另见：
--dlib_config，第 300 页，--no_system_include，第 311 页，--system_include_dir，第 321 页和 --clib，第 292 页。
要设置相关选项，请选择：
Project>Options>General Options>Library Configuration

--dlib_config

语法：

--dlib_config filename.h|config
参数：
文件名 DLIB 配置头文件，请参阅将文件名或目录指定为参数的规则，第 286 页。
config 将使用指定配置的默认配置文件。选择：
none，不使用任何配置
tiny，将使用 tiny 库配置
正常，将使用正常的库配置（默认）
full，将使用完整的库配置。
描述：
使用此选项指定要使用的库配置，方法是指定显式文件或指定库配置--在这种情况下，将使用该库配置的默认文件。确保您指定了与您正在使用的库相对应的配置。如果不指定此选项，将使用默认库配置文件。
所有预构建的运行时库都附带相应的配置文件。您可以在目录 8051\lib 中找到库对象文件和库配置文件。有关预构建运行时库的示例和信息，请参阅预构建运行时库，第 156 页。
如果您构建自己的自定义运行时库，还应创建相应的自定义库配置文件，该文件必须指定给编译器。有关详细信息，请参阅自定义和构建您自己的运行时库，第 153 页。
备注：
此选项仅适用于 IAR DLIB 运行时环境。要设置相关选项，请选择：
Project>Options>General Options>Library Configuration

--dptr

语法：

--dptr={[size][,number][,visibility][,select]}
参数：
size=16|24 以位为单位的指针大小。对于 Extended1 内核，默认值为 24，对于其他内核，默认值为 16。
number=1|2|3|4|5|6|7|8 数据指针（DPTR 寄存器）的数量。对于 Extended1 内核，默认值为 2，对于所有其他内核，默认值为 1。
可见性=分离|阴影

如果您使用 2 个或更多数据指针，DPTR0 寄存器可以隐藏（隐藏）其他寄存器，使它们对编译器不可用，或者它们都可以在单独的特殊功能寄存器中可见。默认可见性是单独的。
select=inc|xor(mask) 指定选择活动数据指针的方法。
XOR 使用 ORL 或 ANL 指令来设置数据指针选择寄存器中的活动指针。使用的位在位掩码中指定。例如，如果使用四个数据指针并且选择位是位 0 和位 2，则掩码应该是 0x05（二进制格式的 00000101）。普通内核的默认值 (0x01)。
INC 递增数据指针选择寄存器中的位以选择活动数据指针。请参见选择活动数据指针，第 65 页。Extended1 内核的默认值。
描述：
使用此选项启用对多个数据指针的支持；许多 8051 设备中的一项功能。您可以指定指针的数量、指针的大小、是否可见以及在它们之间切换的方法。
要使用多个 DPTR，您必须在链接器命令文件或 IAR Embedded Workbench IDE 中指定 DPTR 寄存器和数据指针选择寄存器 (?DPS) 的位置。
示例 要使用两个 16 位数据指针，请使用：--dptr=2,16
在这种情况下，默认值分离用于 DPTR 可见性，xor(0x01) 用于 DPTR 选择。
要使用四个 24 位指针，它们都在单独的寄存器中可见，要在使用 XOR 方法之间切换，请使用：
--dptr=24,4,separate,xor(0x05)
或
--dptr=24 --dptr=4 --dptr=separate --dptr=xor(0x05)
另见：
函数存储的代码模式和内存属性，第 95 页。
Project>Options>General Options>Target>Data Pointer

-e

语法：

-e
描述：
在编译器的命令行版本中，默认情况下禁用语言扩展。如果您在源代码中使用扩展关键字和匿名结构和联合等语言扩展，则必须使用此选项来启用它们。
备注：
-e 选项和 --strict 选项不能同时使用。
另见：
启用语言扩展，第 223 页。
Project>Options>C/C++ Compiler>Language 1>Standard with IAR extensions
Note:
默认情况下，在 IDE 中选择此选项。

--ec++

语法：

--ec++

描述：

在编译器中，默认语言为 C。如果使用 Embedded C++，则必须使用此选项将编译器使用的语言设置为 Embedded C++。

Project>Options>C/C++ Compiler>Language 1>C++

及

Project>Options>C/C++ Compiler>Language 1>C++ dialect>Embedded C++

--eec++

语法：

--eec++

描述：

在编译器中，默认语言是 C。如果您利用扩展嵌入式 C++ 功能（如名称空间或源代码中的标准模板库），则必须使用此选项将编译器使用的语言设置为扩展嵌入式 C++。

另见：

扩展嵌入式 C++，第 230 页。

Project>Options>C/C++ Compiler>Language 1>C++

及

Project>Options>C/C++ Compiler>Language 1>C++ dialect>Extended Embedded C++

--enable_multibytes

语法：

--enable_multibytes
描述：
默认情况下，C 或 C++ 源代码中不能使用多字节字符。使用此选项可使源代码中的多字节字符根据主机计算机对多字节支持的默认设置进行解释。
C 和 C++ 样式的注释、字符串文字和字符常量中允许使用多字节字符。它们原封不动地转移到生成的代码中。
Project>Options>C/C++ Compiler>Language 2>Enable multibyte support

--enable_restrict

语法：

--enable_restrict
描述：
启用标准 C 关键字限制。此选项可用于在优化期间提高分析精度。
To set this option, use Project>Options>C/C++ Compiler>Extra options

--error_limit

语法：

--error_limit=n
参数：
n 编译器停止编译之前的错误数。 n 必须是正整数； 0 表示没有限制。
描述：
使用 --error_limit 选项指定编译器停止编译之前允许的错误数。默认情况下，允许 100 个错误。
此选项在 IDE 中不可用。

--extended_stack

语法：

--extended_stack
描述：
如果您使用 8051 扩展设备，请使用此选项启用可用的扩展堆栈。如果使用扩展堆栈可重入调用约定，则默认设置此选项。对于所有其他调用约定，默认情况下未设置扩展堆栈选项。
备注：
扩展堆栈选项不能与 idata 或 xdata 堆栈一起使用，并且暗示也不能与 idata 可重入或 xdata 可重入调用约定一起使用。
另见：
函数存储的代码模式和内存属性，第 95 页。
Project>Options>General Options>Target>Do not use extended stack

-f

语法：

-f filename
参数：
请参见将文件名或目录指定为参数的规则，第 286 页。
描述：
使用此选项使编译器从命名文件读取命令行选项，默认文件扩展名为 xcl。
在命令文件中，您可以像在命令行本身一样格式化项目，除了您可以使用多行，因为换行符仅充当空格或制表符。
文件中允许使用 C 和 C++ 样式的注释。双引号的行为方式与 Microsoft Windows 命令行环境中的方式相同。
To set this option, use Project>Options>C/C++ Compiler>Extra Options.

--guard_calls

语法：

--guard_calls
描述：
使用此选项为函数静态变量初始化启用保护。此选项应在线程 C++ 环境中使用。
备注：
此选项需要线程 C++ 环境，IAR C/C++ 编译器不支持 8051。

此选项在 IDE 中不可用。

--has_cobank

语法：

--has_cobank
描述：
使用此选项通知编译器您正在使用的设备在存储区选择寄存器中有用于常量的 COBANK 位。特别是，如果它是 Silicon Laboratories C8051F120 器件，这一点很重要。
To set this option, use Project>Options>C/C++ Compiler>Extra Options.

--header_context

语法：

--header_context
描述：
有时，要找出问题的原因，有必要知道哪个头文件包含在哪个源代码行中。对于每条诊断消息，使用此选项不仅可以列出问题的源位置，还可以列出该点的整个包含堆栈。
此选项在 IDE 中不可用。

-I

语法：

-I path
参数：
path #include 文件的搜索路径
描述：
使用此选项指定#include 文件的搜索路径。此选项可以在命令行上多次使用。

另见：
包括文件搜索程序，第 280 页。
Project>Options>C/C++ Compiler>Preprocessor>Additional include directories

-l

语法：

-l[a|A|b|B|c|C|D][N][H] {filename|directory}
参数：
一个（默认）汇编器列表文件
带有 C 或 C++ 源代码作为注释的汇编程序列表文件
b 基本汇编器列表文件。此文件与使用 -la 生成的列表文件具有相同的内容，除了不包含额外的编译器生成的信息（运行时模式属性、调用帧信息、帧大小信息）*
JBasic 汇编器列表文件。此文件与使用 -lA 生成的列表文件具有相同的内容，除了不包含额外的编译器生成信息（运行时模式属性、调用帧信息、帧大小信息）*
c C 或 C++ 列表文件
K（默认）带有汇编源代码的 C 或 C++ 列表文件作为注释
LC 或 C++ 列表文件，带有作为注释的汇编源代码，但没有指令偏移和十六进制字节值
N 文件中没有诊断
H 在输出中包含来自头文件的源代码行。如果没有此选项，则仅包含主源文件中的源代码行
* 这使得列表文件作为汇编器的输入不太有用，但对于人类阅读更有用。
另见：
将文件名或目录指定为参数的规则，第 286 页。
描述：
使用此选项可生成文件的汇编程序或 C/C++ 列表。请注意，此选项可以在命令行上使用一次或多次。
要设置相关选项，请选择：
Project>Options>C/C++ Compiler>List

--library_module

语法：

--library_module
描述：
使用此选项使编译器生成库模块而不是程序模块。链接期间始终包含程序模块。只有在您的程序中引用了库模块时，才会包含它。
Project>Options>C/C++ Compiler>Output>Module type>Library Module

--macro_positions_in_diagnostics

语法：

--macro_positions_in_diagnostics
描述：
使用此选项可获取诊断消息中宏内部的位置参考。这对于检测宏中不正确的源代码结构很有用。
To set this option, use Project>Options>C/C++ Compiler>Extra Options.

--mfc

语法：

--mfc
描述：
使用此选项启用多文件编译。这意味着编译器将命令行中指定的一个或多个源文件作为一个单元进行编译，从而增强了过程间优化。
备注：
编译器将为每个输入源代码文件生成一个目标文件，其中第一个目标文件包含所有相关数据，其他的为空。如果您只想生成第一个文件，请使用 -o 编译器选项并指定某个输出文件。
示例 icc8051 myfile1.c myfile2.c myfile3.c --mfc
另见：
--discard_unused_publics，第 299 页，--output，-o，第 315 页和多文件编译单元，第 264 页。
Project>Options>C/C++ Compiler>Multi-file compilation

--module_name

语法：

--module_name=name
参数：
name 明确的对象模块名称
描述：
通常，目标模块的内部名称是源文件的名称，没有目录名称或扩展名。使用此选项显式指定对象模块名称。
当多个模块具有相同的文件名时，此选项很有用，因为生成的重复模块名通常会导致链接器错误；例如，当源文件是由预处理器生成的临时文件时。
Project>Options>C/C++ Compiler>Output>Object module name

--no_call_frame_info

语法：

--no_call_frame_info
描述：
通常，编译器总是在输出中生成调用帧信息，以使调试器即使在来自没有调试信息的模块的代码中也能显示调用堆栈。使用此选项可禁用调用帧信息的生成。
另见：
调用框架信息，第 211 页。
To set this option, use Project>Options>C/C++ Compiler>Extra Options.

--no_code_motion

语法：

--no_code_motion
描述：

使用此选项可禁用代码运动优化。
备注：
此选项对低于中等的优化级别无效。
另见：
代码运动，第 267 页。
Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Code motion

--no_cross_call

语法：

--no_cross_call
描述：
使用此选项禁用交叉调用优化。
备注：
此选项对低于 High 的优化级别无效。
另见：
交叉调用，第 268 页。
Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Cross call

--no_cse

语法：

--no_cse
描述：
使用此选项禁用公共子表达式消除。
备注：
此选项对低于中等的优化级别无效。
另见：
公共子表达式消除，第 267 页。
Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination

--no_inline

语法：

```
--no_inline
```

描述：

使用此选项禁用函数内联。

另见：
内联函数，第 103 页。

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Function inlining

--no_path_in_file_macros

语法：

```
--no_path_in_file_macros
```

描述：

使用此选项可从预定义的预处理器符号 __FILE__ 和 __BASE_FILE__ 的返回值中排除路径。

另见：
预定义预处理器符号的描述，第 388 页。

此选项在 IDE 中不可用。

--no_size_constraints

语法：

```
--no_size_constraints
```

描述：

在针对高速进行优化时，使用此选项可以放宽对代码大小扩展的正常限制。

备注：

除非与 -Ohs 一起使用，否则此选项无效。
另见：
速度与尺寸，第 265 页。
Project>Options>C/C++ Compiler>Optimizations>Enable transformations>No size constraints

--no_static_destruction

语法：

--no_static_destruction
描述：
通常，编译器会发出代码来销毁需要在程序退出时销毁的 C++ 静态变量。有时，不需要这种破坏。
使用此选项可抑制此类代码的发射。
另见：
系统终止，第 166 页。
To set this option, use Project>Options>C/C++ Compiler>Extra Options.

--no_system_include

语法：

--no_system_include
描述：
默认情况下，编译器会自动定位系统包含文件。使用此选项可禁用系统包含文件的自动搜索。在这种情况下，您可能需要使用 -I 编译器选项设置搜索路径。
另见：
--dlib，第 300 页，--dlib_config，第 300 页，和 --system_include_dir，第 321 页。
Project>Options>C/C++ Compiler>Preprocessor>Ignore standard include directories

--no_tbaa

语法：

--no_tbaa
描述：
使用此选项禁用基于类型的别名分析。
备注：
此选项对低于 High 的优化级别无效。
另见：
基于类型的别名分析，第 268 页。
Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Type-based alias analysis

--no_typedefs_in_diagnostics

语法：

--no_typedefs_in_diagnostics
描述：
使用此选项可禁用在诊断中使用 typedef 名称。通常，当在来自编译器的消息中提到类型时，最常见的是在某种诊断消息中，只要它们使结果文本更短，就会使用原始声明中使用的 typedef 名称。
示例 typedef int (*MyPtr)(char const *); MyPtr p = "我的文本字符串";将给出如下错误消息：
错误[P   e144]：“char *”类型的值不能用于初始化“MyPtr”类型的实体
如果使用 --no_typedefs_in_diagnostics 选项，错误消息将如下所示：
错误[P   e144]：“char *”类型的值不能用于初始化“int (*)(char const *)”类型的实体
To set this option, use Project>Options>C/C++ Compiler>Extra Options.

--no_ubrof_messages

语法：

--no_ubrof_messages
描述：
使用此选项可通过从 UBROF 文件中排除消息来最小化应用程序对象文件的大小。文件大小最多可减少 60%。请注意，当您使用此选项时，XLINK 诊断消息的用处将减少。
To set this option, use Project>Options>C/C++ Compiler>Extra Options.

--no_unroll

语法：

--no_unroll
描述：
使用此选项禁用循环展开。
备注：
此选项对低于 High 的优化级别无效。
另见：
循环展开，第 267 页。
Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling

--no_warnings

语法：

--no_warnings
描述：
默认情况下，编译器会发出警告消息。使用此选项可禁用所有警告消息。
此选项在 IDE 中不可用。

--no_wrap_diagnostics

语法：

--no_wrap_diagnostics
描述：
默认情况下，诊断消息中的长行被分成几行以使消息更易于阅读。使用此选项可禁用诊断消息的换行。
此选项在 IDE 中不可用。

--nr_virtual_regs

语法：

--nr_virtual_regs=n
参数：
n 工作区的大小（虚拟寄存器的数量）；一个介于 8 到 32 之间的值。
描述：
使用此选项指定虚拟寄存器的数量，这决定了工作区的大小。虚拟寄存器位于数据存储器中。
另见：
虚拟寄存器，第 92 页。
Project>Options>General Options>Target>Number of virtual registers

-O

语法：

-O[n|l|m|h|hs|hz]
参数：
n 无*（最佳调试支持）
l（默认）低*
m 中
h 高、平衡
hs 高，有利于速度
hz 高，有利于尺寸

*None 和 Low 之间最重要的区别在于，在 None 时，所有非静态变量都将在其整个范围内存在。
描述：
使用此选项可设置编译器在优化代码时使用的优　　化级别。如果未指定优化选项，则默认使用优化级别 Low。如果只使用-O，不带任何参数，则使用优化级别 High balance。
低级别的优化使得在调试器中跟踪程序流程相对容易，相反，高级别的优化使其相对困难。
另见：
控制编译器优化，第 263 页。
Project>Options>C/C++ Compiler>Optimizations

--omit_types

语法：

--omit_types
描述：
默认情况下，编译器在对象输出中包含有关变量和函数的类型信息。如果您不希望编译器在输出中包含此类型信息，请使用此选项，这在您构建不应包含类型信息的库时很有用。目标文件将只包含作为符号名称一部分的类型信息。这意味着链接器无法检查符号引用的类型正确性。
To set this option, use Project>Options>C/C++ Compiler>Extra Options.

--only_stdout

语法：

--only_stdout
描述：
使用此选项可使编译器也将标准输出流 (stdout) 用于通常定向到错误输出流 (stderr) 的消息。
此选项在 IDE 中不可用。

--output, -o

语法：

--output {filename|directory}
-o {filename|directory}
参数：
请参见将文件名或目录指定为参数的规则，第 286 页。
描述：
默认情况下，编译器生成的目标代码输出位于与源文件同名但扩展名为 r51 的文件中。使用此选项为目标代码输出显式指定不同的输出文件名。
此选项在 IDE 中不可用。

--pending_instantiations

语法：

--pending_instantiations number
参数：
number 指定限制的整数，其中 64 是默认值。如果 0 使用，则没有限制。
描述：
使用此选项指定给定 C++ 模板的最大实例化数，该模板允许在给定时间被实例化。这用于检测递归实例化。
Project>Options>C/C++ Compiler>Extra Options

--place_constants

语法：

--place_constants={data|data_rom|code}
参数：
数据（默认）将常量和字符串从代码存储器复制到数据存储器。具体数据存储器取决于默认数据模式。

data_rom 将常量和字符串放置在 xdata 或远存储器中，具体取决于数据模式，位于 ROM 所在的范围内。在大数据模式中，对象被放置在 xdata 内存中，而在远数据模式中，它们被放置在远内存中。在其余数据模式中，不允许使用 data_rom 修饰符。
代码 将常量和字符串放在代码存储器中。在这种情况下，无法按原样使用预构建的运行时库。
描述：
使用此选项指定常量和字符串的默认位置。可以使用关键字覆盖单个常量和字符串的默认位置。
如果您在代码内存中定位常量和字符串，您可能需要使用一些 8051 特定的 CLIB 库函数变体，这些函数变体允许访问代码内存中的字符串。
另见：
常量和字符串，第 82 页和 8051 特定的 CLIB 函数，第 404 页。
Project>Options>General Options>Target>Location for constants and strings
Project>Options>Linker>Output>Output file

--predef_macros

语法：

--predef_macros {filename|directory}
参数：
请参见将文件名或目录指定为参数的规则，第 286 页。
描述：
使用此选项列出预定义的符号。使用此选项时，请确保也使用与项目其余部分相同的选项。
如果指定了文件名，编译器会将输出存储在该文件中。如果指定了目录，则编译器将输出存储在该目录中，存储在具有 predef 文件扩展名的文件中。
请注意，此选项要求您在命令行上指定源文件。此选项在 IDE 中不可用。

--preinclude

语法：

--preinclude includefile
参数：

请参见将文件名或目录指定为参数的规则，第 286 页。
描述：
使用此选项使编译器在开始读取源文件之前读取指定的包含文件。如果您想更改整个应用程序的源代码中的某些内容，这将非常有用，例如，如果您想定义一个新符号。
Project>Options>C/C++ Compiler>Preprocessor>Preinclude file

--preprocess

语法：

--preprocess[=[c][n][l]] {filename|directory}
参数：
c 保留注释
n 仅预处理
l 生成#line 指令
另见：
将文件名或目录指定为参数的规则，第 286 页。
描述：
使用此选项将预处理输出生成到命名文件。
Project>Options>C/C++ Compiler>Preprocessor>Preprocessor output to file

--public_equ

语法：

--public_equ symbol[=value]
参数：
symbol 要定义的汇编程序符号的名称
value 定义的汇编符号的可选值
描述：
此选项等效于使用 EQU 指令以汇编语言定义标签并使用 PUBLIC 指令将其导出。
此选项可以在命令行上多次使用。
此选项在 IDE 中不可用。

--relaxed_fp

语法：

--relaxed_fp
描述：
使用此选项允许编译器放宽语言规则并执行更积极的浮点表达式优化。此选项可提高满足以下条件的浮点表达式的性能：
C 表达式由单精度和双精度值组成
D 双精度值可以转换为单精度而不损失精度
E 表达式的结果被转换为单精度。
请注意，以单精度而不是双精度执行计算可能会导致精度损失。
示例 float F(float a, float b)
{
返回 a + b * 3.0;
}
C 标准规定此示例中的 3.0 具有 double 类型，因此应以双精度计算整个表达式。然而，当使用 --relaxed_fp 选项，3.0 将转换为浮点数，整个表达式可以以浮点精度计算。
要设置相关选项，请选择：
Project>Options>C/C++ Compiler>Language 2>Floating-point semantics

--remarks

语法：

--remarks
描述：
最不严重的诊断消息称为备注。备注表示可能在生成的代码中导致奇怪行为的源代码构造。默认情况下，编译器不生成备注。使用此选项使编译器生成注释。
另见：
严重性级别，第 283 页。
Project>Options>C/C++ Compiler>Diagnostics>Enable remarks

--require_prototypes

语法：

--require_prototypes
描述：
使用此选项强制编译器验证所有函数是否具有正确的原型。使用此选项意味着包含以下任何内容的代码将生成错误：
●没有声明或带有 Kernighan & Ritchie C 声明的函数的函数调用
●没有先前原型声明的公共函数的函数定义
●通过类型不包含原型的函数指针的间接函数调用。
Project>Options>C/C++ Compiler>Language 1>Require prototypes

--rom_mon_bp_padding

语法：

--rom_monitor_bp_padding
描述：
使用通用 C-SPY ROM-monitor 调试器时，使用此选项可以在所有 C 语句上设置断点。
当 C-SPY ROM 监视器设置断点时，它用 3 字节指令 LCALL 监视器替换原始指令。对于原始指令的大小与三个字节不同的情况，编译器将插入额外的 NOP 指令（填充）以确保到该目标的所有跳转都正确对齐。
备注：
此机制仅支持在 C 语句级别设置的断点。对于汇编代码中的断点，您必须手动添加焊盘。
Project>Options>C/C++ Compiler>Code>Padding for ROM-monitor breakpoints

--silent

语法：

--silent
描述：
默认情况下，编译器会发布介绍性消息和最终统计报告。使用此选项可使编译器在不将这些消息发送到标准输出流（通常是屏幕）的情况下运行。

此选项不影响错误和警告消息的显示。此选项在 IDE 中不可用。

--strict

语法：

```
--strict
```

描述：
默认情况下，编译器接受标准 C 和 C++ 的宽松超集。使用此选项可确保您的应用程序的源代码符合严格的标准 C 和 C++。
备注：
-e 选项和 --strict 选项不能同时使用。
另见：
启用语言扩展，第 223 页。
Project>Options>C/C++ Compiler>Language 1>Language conformance>Strict

--system_include_dir

语法：

```
--system_include_dir path
```

参数：
path 系统包含文件的路径，请参见将文件名或目录指定为参数的规则，第 286 页。
描述：
默认情况下，编译器会自动定位系统包含文件。使用此选项显式指定系统包含文件的不同路径。如果您尚未在默认位置安装 IAR Embedded Workbench，这可能会很有用。
另见：
--dlib_config，第 300 页，和 --no_system_include，第 311 页。
此选项在 IDE 中不可用。

--use_c++_inline

语法：

--use_c++_inline
描述：
标准 C 对 inline 关键字使用的语义与 C++ 稍有不同。
如果您在使用 C 时需要 C++ 语义，请使用此选项。
另见：
内联函数，第 103 页
Project>Options>C/C++ Compiler>Language 1>C dialect>C99>C++ inline semantics

--version

语法：

--version
描述：
使用此选项使编译器将版本信息发送到控制台，然后退出。
此选项在 IDE 中不可用。

--vla

语法：

--vla
描述：
使用此选项可启用对 C99 可变长度数组的支持。这样的数组位于堆上。此选项需要标准 C，不能与 --c89 编译器选项一起使用。
备注：
--vla 不应与 longjmp 库函数一起使用，因为这会导致内存泄漏。
另见：
C 语言概述，第 221 页。
Project>Options>C/C++ Compiler>Language 1>C dialect>Allow VLA

--warn_about_c_style_casts

语法：

--warn_about_c_style_casts
描述：
使用此选项可在 C++ 源代码中使用 C 样式转换时使编译器发出警告。
此选项在 IDE 中不可用。

--warnings_affect_exit_code

语法：

--warnings_affect_exit_code
描述：
默认情况下，退出代码不受警告影响，因为只有错误会产生非零退出代码。 使用此选项，警告还将生成非零退出代码。
此选项在 IDE 中不可用。

--warnings_are_errors

语法：

--warnings_are_errors
描述：
使用此选项可使编译器将所有警告视为错误。 如果编译器遇到错误，则不会生成目标代码。 已更改为备注的警告不视为错误。
备注：
已被选项重新分类为警告的任何诊断消息
--diag_warning 或 #pragma diag_warning 指令在使用 --warnings_are_errors 时也将被视为错误。
另见：
--diag_warning，第 298 页。
Project>Options>C/C++ Compiler>Diagnostics>Treat all warnings as errors

（空白页）
页码:324

数据表示

- 对齐
- 基本数据类型--整数类型
- 基本数据类型--浮点类型
- 指针类型
- 结构类型
- 类型限定符
- C++ 中的数据类型

有关哪些数据类型和指针为您的应用程序提供最高效代码的信息，请参阅嵌入式应用程序的高效编码一章。

对齐

每个 C 数据对象都有一个对齐方式，用于控制对象在内存中的存储方式。例如，如果一个对象的对齐方式为 4，则它必须存储在可被 4 整除的地址上。

对齐概念的原因是某些处理器对如何访问内存有硬件限制。

假设处理器可以使用一条指令读取 4 个字节的内存，但前提是读取的内存位于可被 4 整除的地址上。那么，4 字节对象（例如长整数）将具有对齐 4。

另一个处理器可能一次只能读取 2 个字节；在那种环境中，4 字节长整数的对齐方式可能是 2。

所有数据类型的大小都必须是其对齐的倍数。否则，只能保证按照对齐要求放置数组的第一个元素。

这意味着编译器可能会在结构的末尾添加填充字节。

注意，通过#pragma data_alignment 指令，你可以增加对特定变量的对齐要求。

8051 微控制器上的对齐方式

8051 微控制器没有任何对齐限制。

基本数据类型-整数类型

编译器支持所有标准 C 基本数据类型和一些其他类型。

整数类型-概述

此表给出了每个整数数据类型的大小和范围：

数据类型	长度	范围	对齐
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	-231 to 231-1	1
unsigned long	32 bits	0 to 232-1	1
signed long long	32 bits	-231 to 231-1	1
unsigned long long	32 bits	0 to 232-1	1

表 35：整数类型

有符号的变量用两者的补体形式来表示。

布尔

默认使用 C-++语言支持 bool 数据类型。如果您启用了语言扩展，如果包含文件 stdbool.h，还可以在 C 源代码中使用 bool 类型。这也将使布尔值为假和真。

枚举类型

编译器将使用保留枚举常数所需的最小类型，它更喜欢有符号而不是无符号。

启用 IAR Systems 语言扩展时,在 C++ 中,枚举常量和类型也可以是 long、unsigned long、long long 或 unsigned long long 类型。
要使编译器使用比它自动使用的更大的类型,请定义一个具有足够大值的枚举常量。 例如:

```
/* Disables usage of the char type for enum */
enum Cards{Spade1, Spade2,
DontUseChar=257}
```

字符类型

char 类型在编译器中默认是无符号的,但 --char_is_signed 编译器选项允许您将其设为有符号。但是请注意,该库是使用 char 类型作为 unsigned 编译的。

WCHAR_T 类型

wchar_t 数据类型是一个整数类型,其值范围可以表示受支持的局部变量中指定的最大扩展字符集的所有成员的不同代码。
C++ 语言默认支持 wchar_t 数据类型。要在 C 源代码中也使用 wchar_t 类型,您必须包含运行时库中的文件 stddef.h。
注意:IAR CLIB 库仅对 wchar_t 提供基本支持。

位域

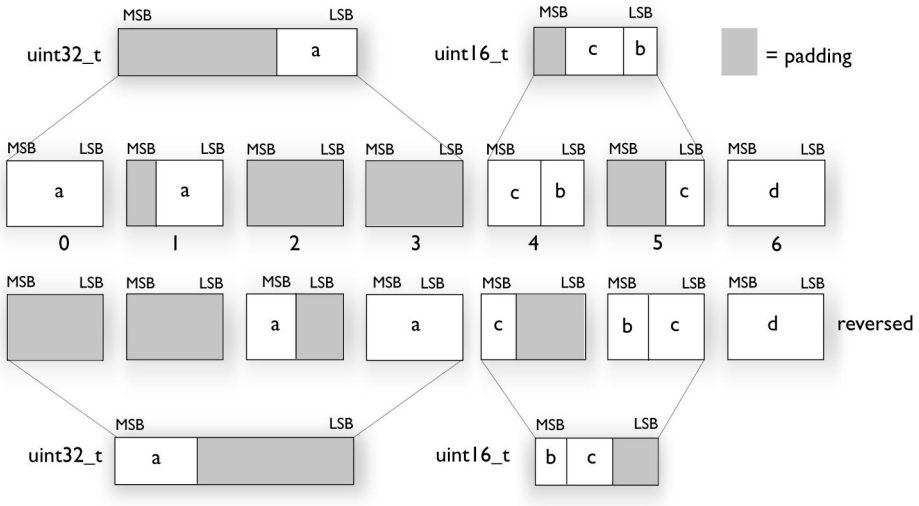
在标准 C 中,int、signed int 和 unsigned int 可用作整数位域的基本类型。在标准 C++ 中,以及在编译器中启用语言扩展的 C 中,任何整数或枚举类型都可以用作基本类型。它是由实现定义的纯整数类型(char、short、int 等)是否导致有符号或无符号位域。
在 8051 的 IAR C/C++ 编译器中,纯整数类型被视为有符号。
如果 int 可以表示位域的所有值,则表达式中的位域被视为 int。
否则,它们将被视为位域基本类型。请注意,如果在 bdata 内存中声明,包含 1 位字段的位字段将非常紧凑。这些字段也将非常有效地访问。
从最低有效位到最高有效位,每个位域都放置在其基本类型的容器中。如果最后一个容器是相同类型并且有足够的可用位,则将位域放置到此容器中,否则分配一个新容器。这种分配方案称为不相交类型位域分配。

如果您使用指令#pragma bitfields=reversed，则位域从每个容器中的最高有效位到最低有效位放置。 参见位域，第 363 页。
假设这个例子：

```
struct BitfieldExample
{
  uint32_t a : 12;
  uint 16_t b : 3;
  uint16_t c : 7;
  uint8_t d;
};
```

不相交类型位域分配策略中的示例 为了放置第一个位域 a,编译器在偏移量 0 处分配一个 32 位容器，并将 a 放入容器的最低有效 12 位。
为了放置第二个位域 b，在偏移量 4 处分配了一个新容器，因为位域的类型与前一个的类型不同。 b 被放入此容器的最低有效三位中。
第三个位域 c 与 b 具有相同的类型并适合同一个容器。
第四个成员 d 分配到偏移量 6 处的字节中。d 不能与 b 和 c 放在同一个容器中，因为它不是位域，它不是同一类型，并且不适合。
使用反向顺序时，每个位域都从其容器的最高有效位开始放置。

这是 bitfield_example 的布局：



基本数据类型--浮点类型

在 8051 的 IAR C/C++ 编译器中，浮点值以标准 IEEE 754 格式表示。不同浮点类型的大小为：

类型	大小	范围 (+/-)	指数	尾数
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

表 36: 浮点类型

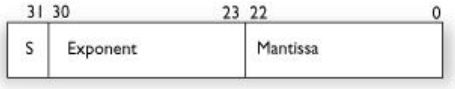
编译器不支持次正态数。所有应该产生次正态数的操作都会产生零。

浮点环境

不支持异常标志。feraiseexcept 函数不会引发任何异常。

32 位浮点格式

32 位浮点数作为整数的表示方法是:



指数为 8 位，尾数为 23 位。

该数字的值为：

(-1)S * 2(指数-127) * 1.尾数

数字的范围至少是：

±1.18E-38 至 ±3.39E+38

浮点运算符（+、-、* 和 /）的精度约为 7 位十进制数字。

特殊浮点数的表示

此列表描述了特殊浮点数的表示：

- 零由零尾数和指数表示。符号位表示正零或负零。
- 通过将指数设置为最大值并将尾数设置为零来表示无穷大。符号位表示正无穷或负无穷。
- 非数字 (NaN) 通过将指数设置为最大正值并将尾数设置为非零值来表示。符号位的值被忽略。

注意：IAR CLIB 库不完全支持浮点数的特殊情况，例如无穷大、NaN。将浮点数的这些特殊情况之一作为参数的库函数可能会出现意外行为。

指针类型

编译器有两种基本类型的指针：函数指针和数据指针。

函数指针

代码指针有两种大小：16 位或 24 位。这些函数指针可用：

指针	大小	地址范围	描述
__near_func	2 bytes	0 - 0xFFFF	使用 LCALL/LJMP 指令调用函数
__banked_func	2 bytes	0 - 0xFFFFFF	调用执行分页切换并跳转到分组函数的中继函数。用途 ?BRET 从函数返回。参见第 114 页的分页代码模式中的分页切换
__banked_func_ext2	3 bytes	0 - 0xFFFFFF	使用 MEX1 寄存器和内存扩展堆栈。参见第 11 页分页扩展 2 代码模式中的分页切换
__far_func	3 bytes	0 - 0xFFFFFF	使用支持 24 位目标地址的扩展 LCALL/LJMP 指令调用函数。（这些说明仅在某些设备中可用）

表 37：函数指针

数据指针

数据指针具有三种大小：8、16 或 24 位。8 位指针用于 data、bdata、idata 或 pdata 内存，16 位指针用于 xdata 或 16 位代码内存，24 位指针用于扩展内存和通用指针类型 . 这些数据指针可用：

指针	地址范围	指针大小	索引类型	描述
__idata	0 - 0xFF	1 byte	signed char	间接访问的数据存储器，使用 MOV A,@Ri 访问
__pdata	0 - 0xFF	1 byte	signed char	Parameter data, accessed using MOVX A,@Ri
__xdata	0 - 0xFFFF	2 bytes	signed short	Xdata memory, accessed usingMOVX A,@DPTR
__generic	0 - 0xFFFF	3 bytes	signed short	最高有效字节标识指针指向代码存储器还是数据存储器
__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
__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

表 38：数据指针

* far22 和 far 指针类型有一个小于指针大小的索引类型，这意味着指针运算将只在低 16 位上进行。
这就限制了指针所指向的对象的位置。也就是说，该对象只能放在 64K 字节的页面内。

通用指针

一个通用指针可以访问位于数据和代码内存中的对象。这些指针的大小为 3 个字节。最重要的字节揭示了对象位于哪种内存类型中，其余的位指定了该内存中的地址。

分段的数据指针比较

请注意，在数据指针上使用关系运算符（<, <=, <, <=）的结果只有在指针指向同一个对象时才会被定义。对于分段数据指针，在使用这些运算符时，只有指针的偏移部分被比较。比如说

```
void MyFunc(char __far * p, char __far * q)
{
  if (p == q) /* 对整个指针进行比较 */
  ...
  if (p < q) /* 只比较指针的低 16 位 */
  ...
}
```

强制转换

指针之间的强制转换具有以下特征：

- 将整数类型的值转换为较小类型的指针是通过截断执行的
- 将整数类型的值转换为更大类型的指针是通过零扩展执行的
- 通过截断将指针类型转换为更小的整数类型
- 将指针类型转换为更大的整数类型是通过零扩展执行的
- 将数据指针转换为函数指针，反之亦然是非法的
- 将函数指针转换为整数类型会产生未定义的结果
- 通过零扩展执行从较小指针到较大指针的转换
- 从较大指针到较小指针的转换是通过截断来执行的。

size_t

size_t 是 sizeof 运算符结果的无符号整数类型。在 8051 的 IAR C/C++ 编译器中，用于 size_t 的类型。

请注意，某些数据内存类型可能能够容纳比默认指针指向的内存更大或仅更小的对象。在这种情况下，sizeof 运算符的结果类型可以是更大或更小的无符号整数类型。

每个内存类型都有一个对应的 size_t typedef，以内存类型命名。换句话说，__pdata_size_t 用于 __pdata 内存。

ptrdiff_t

ptrdiff_t 是两个指针相减结果的有符号整数类型。在 8051 的 IAR C/C++ 编译器中，用于 ptrdiff_t 的类型是有符号整数 size_t 类型的变体。

请注意，减去默认指针以外的指针可能会导致更小或更大的整数类型。在每种情况下，此整数类型都是相应 size_t 类型的有符号整数变体。

注意：有时可能会创建一个太大的对象，以至于减去该对象中的两个指针的结果是负数。看这个例子：

intptr_t

intptr_t 是一个有符号整数类型，大到足以包含一个 void *。在 8051 的 IAR C/C++ 编译器中，用于 intptr_t 的类型是有符号长型。

uintptr_t

uintptr_t 等价于 intptr_t，但它是无符号的。

结构类型

结构的成员按照声明的顺序依次存储：第一个成员具有最低的内存地址。

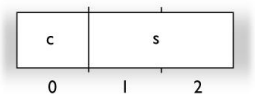
总体布局

结构的成员总是按照声明中指定的顺序分配。

每个成员都根据指定的对齐方式（偏移量）放置在结构中。

```
struct First
{
  char c;
  short s;
} s;
```

这张图显示了内存中的布局：



结构的对齐方式为 1 字节，大小为 3 字节。

类型限定符

根据 C 标准，volatile 和 const 是类型限定符。

声明对象易变

通过声明一个 volatile 的对象，编译器被告知该对象的值可以改变超出编译器的控制。 编译器还必须假定任何访问都可能产生副作用--因此必须保留对 volatile 对象的所有访问。 将对象声明为 volatile 有三个主要原因：

- 共享访问； 该对象在多任务环境中的多个任务之间共享
- 触发访问； 至于内存映射的 SFR，其中访问发生的事实会产生影响
- 修改访问； 其中对象的内容可以以编译器不知道的方式更改。

页码:334

对易失性对象的访问的定义

C 标准定义了一个抽象机器，它控制对易失性声明对象的访问行为。一般而言，根据抽象机器：

- 编译器将对声明为易失性对象的每个读写访问权限视为访问
- 访问的单位可以是整个对象，也可以是组合对象（如数组、结构、类或联合）中元素的访问单位。例如：

```
char volatile a;
a = 5;  /* 写入存取 */
a += 6; /* 先读后写入存取 */
```

- 对位域的访问被视为对基础类型的访问
- 向 volatile 对象添加 const 限定符将使对该对象的写访问变得不可能。但是，对象将按照 C 标准的规定放置在 RAM 中。

但是，这些规则不够详细，无法处理与硬件相关的要求。 8051 的 IAR C/C++ 编译器的特定规则如下所述。

访问规则

在 8051 的 IAR C/C++ 编译器中，对 volatile 声明对象的访问受以下规则的约束：

- 保留所有访问
- 所有访问都是完整的，即访问了整个对象
- 所有访问都按照抽象机中给定的顺序执行
- 所有访问都是原子的，即不能被中断。

对于任何数据存储器 (RAM) 中易失性声明对象的 8 位访问，以及位于可位寻址 sfr 存储器或 bdata 存储器中的易失性声明对象的 1 位访问，编译器都遵循这些规则。

对于未列出的所有对象类型的组合，只有声明所有访问都被保留的规则适用。

页码:335

声明对象的 volatile 和 const

如果你声明一个 volatile 对象为 const，它将被写保护，但它仍将按照 C 标准的规定存储在 RAM 内存中。

要想把对象存储在只读内存中，但仍然可以把它作为一个 const volatile 对象来访问，可以用这些内存属性之一来声明它。__code, __far_code, __far_rom, __far22_code, __far22_rom, __huge_code, __huge_rom, 或 __xdata_rom。

声明对象的常量

const 类型修饰符用于表示一个数据对象，直接或通过指针访问，是不可写的。一个指向 const 声明数据的指针可以指向常量和非常量对象。

尽可能使用常量声明的指针是很好的编程实践,因为这可以提高编译器优化生成代码的可能性，并减少由于错误地修改数据而导致应用失败的风险。

声明为常数的静态和全局对象，并位于使用内存属性__code、__far_code 和__huge_code 的内存中，在 ROM 中分配。

对于所有其他的内存属性，这些对象被分配在 RAM 中，并在启动时由运行时系统初始化。

在 C++中，需要运行时初始化的对象不能被放在 ROM 中。

C++中的数据类型

在 C++中，所有普通 C 语言的数据类型的表示方法与本章前面描述的相同。

然而，如果一个类型使用了 Embedded C++的任何特性，就不能对数据表示进行假设。这意味着，例如，不支持编写访问类成员的汇编代码。

页码:336

扩展关键字

- 扩展关键字的一般语法规则
- 扩展关键词总结
- 扩展关键词说明

有关不同内存区域的地址范围的信息，请参阅“段参考”一章。

扩展关键字的一般语法规则

编译器提供了一组可用于函数或数据对象的属性，以支持 8051 微控制器的特定功能。属性有两种类型--类型属性和对象属性：

- 类型属性影响数据对象或函数的外部功能
- 对象属性影响数据对象或函数的内部功能。

关键字的语法略有不同，具体取决于它是类型属性还是对象属性，以及它是应用于数据对象还是函数。

有关每个属性的详细信息，请参阅扩展关键字的描述，第 342 页。有关如何使用属性修改数据的信息，请参阅数据存储一章。

有关如何使用属性修改函数的信息，请参阅函数一章。

注意：扩展关键字仅在编译器中启用语言扩展时可用。

在 IDE 中，默认启用语言扩展。

使用 -e 编译器选项启用语言扩展。参见 -e，第 302 页。

类型属性

类型属性定义如何调用函数，或如何访问数据对象。这意味着如果您使用类型属性，则必须在定义函数或数据对象时以及声明时都指定它。

您可以在声明中显式放置类型属性，也可以使用编译指示指令#pragma type_attribute。

类型属性又可以分为内存类型属性和通用类型属性。内存类型属性在文档的其余部分中简称为内存属性。

内存属性

内存属性对应于微控制器中的某个逻辑或物理内存。

可用的函数内存属性：

__near_func, __far_func, __banked_func

可用数据存储器属性：

__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 和 __xdata_rom。

指针或 C++ 引用的数据对象、函数和目标始终具有内存属性。如果声明中或编译指示指令 #pragma type_attribute 没有明确指定属性，则编译器会隐式使用适当的默认属性。您可以为每一级指针间接指定一个内存属性。

通用类型属性

可用的函数类型属性（影响函数的调用方式）：

__data_overlay，__ext_stack_reentrant，__idata_overlay，__idata_reentrant 、 __interrupt 、 __monitor 、 __pdata_reentrant 、 __task 和 __xdata_reentrant

您可以为每个级别的指针间接指定所需的类型属性。

用于数据对象的类型属性的语法

类型属性使用与类型限定符 const 和 volatile 几乎相同的语法规则。例如：

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

i 和 j 都放在 pdata 内存中。
与常量和易失性不同，当在派生类型中的类型说明符之前使用类型属性时，类型属性将应用于对象或 typedef 本身，结构成员声明除外。

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

在所有情况下，如果未指定内存属性，则使用适当的默认内存类型，这取决于所使用的数据模式。
使用类型定义有时可以使代码更清晰：

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

d16_int 是 pdata 内存中整数的 typedef。 变量 q1 可以指向这样的整数。

您还可以使用#pragma type_attributes 指令为声明指定类型属性。 在 pragma 指令中指定的类型属性应用于正在声明的数据对象或 typedef。

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

变量 q2 被放置在 pdata 内存中。
关于使用内存属性的更多例子，请看更多例子，第 78 页。
在函数上使用类型属性的语法
在函数上使用类型属性的语法与在数据对象上使用类型属性的语法略有不同。对于函数来说，属性必须放在返回类型的前面，或者放在括号里，例如：

```
__interrupt void my_handler(void);
```

或

```
void (__interrupt my_handler)(void);
```

此 my_handler 声明与前一个声明等效：

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

若要声明函数指针，请使用以下语法：
int (__near_func * fp) (double);
在这个声明之后，函数指针 fp 指向 pdata 存储器。 指定存储的一种更简单的方法是使用类型定义:
typedef __near_func void FUNC_TYPE(int);
typedef FUNC_TYPE *FUNC_PTR_TYPE;
FUNC_TYPE func();
FUNC_PTR_TYPE funcptr;

注意，#实用的 type_attribute 可以与类型定义声明一起使用。
对象属性
这些对象属性是可用的：
●对象属性，可用于以下变量：
__no_alloc, __no_alloc16, __no_alloc_str, __no_alloc_str16,
__no_init, __ro_placement
● 可用于函数和变量的对象属性:
location, @, __root
● 可用于函数的对象属性：
__intrinsic, __noreturn, vector
对于特定的函数或数据对象，可以根据需要指定任意多的对象属性。 有关位置和@的更多信息，请参见第 259 页的控制数据和函数在存储器中的位置。有关 vector 的更多信息，请参见 vector，381 页。

对象属性的语法
object 属性必须放在类型的前面。例如，要将 myarray 放入启动时未初始化的内存中:
__no_init int myarray[10];
#pragma object_attribute 指令也可以使用。 这个声明等同于前一个：
#pragma object_attribute=__no_init
int myarray[10];
注意: object 属性不能与 typedef 关键字组合使用。

扩展关键字概述

下表总结了扩展关键字：

扩展的关键字	描述
__banked_func	控制函数的存储
__banked_func_ext2	控制函数的存储
__bdata	控制数据对象的存储
__bit	控制数据对象的存储
__code	控制数据对象的存储
__data	控制数据对象的存储
__data_overlay	控制自动数据对象的存储
__ext_stack_reentrant	控制自动数据对象的存储
__far	控制自动数据对象的存储
__far_code	控制常量数据对象的存储
__far_func	控制函数的存储
__far_rom	控制常量数据对象的存储
__far22	控制数据对象的存储
__far22_code	控制常量数据对象的存储
__far22_rom	控制常量数据对象的存储
__generic	指针类型属性
__huge	控制数据对象的存储
__huge_code	控制常量数据对象的存储
__huge_rom	控制常量数据对象的存储
__idata	控制数据对象的存储
__idata_overlay	控制自动数据对象的存储
__idata_reentrant	控制自动数据对象的存储
__ixdata	控制数据对象的存储
__interrupt	指定中断函数
__intrinsic	仅供编译器内部使用
__monitor	指定函数的原子执行
__near_func	控制函数的存储
__no_alloc,__no_alloc16	使执行文件中的常量可用
__no_alloc_str,__no_alloc_str16	使执行文件中的字符串文字可用
__no_init	将数据对象放置在非易失性存储器中
__noreturn	通知编译器该函数不会返回
__overlay_near_func	仅供编译器内部使用
__pdata	控制数据对象的存储
__pdata_reentrant	控制自动数据对象的存储
__root	确保即使未使用函数或变量也包含在目标代码中

页码:341

__ro_placement	将 const volatile 数据放入只读存储器
__sfr	控制数据对象的存储
__xdata	控制数据对象的存储
__xdata_reentrant	控制自动数据对象的存储
__xdata_rom	控制常量数据对象的存储

表 39: 扩展关键词汇总

扩展关键字的描述

本节给出了每个扩展关键字的详细信息。

__banked_func

语法：

请参阅函数中使用的类型属性的语法，第 339 页。

描述：

__banked_func 内存属性覆盖所选代码模式给定的函数的默认存储，并将各个函数放置在内存中，使用分页的 24 位调用调用它们。您还可以使用 __banked_func 属性来创建显式指向位于存储区内存中的对象的指针。

储存信息：

●内存空间：代码内存空间

●地址范围：0-0xFFFFFF

●最大大小：64 KB

●指针大小：2 字节

备注：

该关键字仅在使用 分页代码模式时可用，在这种情况下，函数默认为 __banked_func。有一些例外情况，请参阅无法存储的代码，第 113 页。无法存储覆盖和扩展堆栈函数。这意味着您不能将 __banked_func 关键字与 __data_overlay 或 __idata_overlay 以及 __ext_stack_reentrant 关键字结合使用。

示例：

```
__banked_func 无效 myfunction(void);
```

另见：

分页函数，第 107 页。

__banked_func_ext2

语法：

请参阅函数中使用的类型属性的语法，第 339 页。

描述：

__banked_func_ext2 内存属性覆盖所选代码模式给定的函数的默认存储，并将单个函数放置在内存中，使用分页的 24 位调用调用它们。您还可以使用 __banked_func_ext2 属性创建一个指针，明确指向位于存储区内存中的对象。
储存信息：
●内存空间：代码内存空间
●地址范围：0-0xFFFFFF
●最大大小：64 KB
●指针大小：3 字节
备注：
此关键字仅在使用 分页扩展 2 代码模式时可用，在这种情况下，所有函数默认为 __banked_func_ext2。此类函数需要 Xdata 可重入调用约定。
示例：
__banked_func_ext2 void myfunction(void);
另见：
分页函数，第 107 页。

__bdata

语法：
请参阅用于数据对象的类型属性的语法，第 338 页。
描述：
__bdata 内存属性覆盖所选数据模式给定的变量的默认存储，并将单个变量和常量放置在 bdata 内存中。
备注：
没有 __bdata 指针。 Bdata 内存由 __idata 指针引用。
储存信息：
●内存空间：内部数据内存空间
●地址范围：0x20 – 0x2F
●最大对象大小：16 字节
●指针大小：1 字节，__idata 指针
示例：
__bdata int x;
另见：
内存类型，第 68 页。

__bit

语法：
请参阅用于数据对象的类型属性的语法，第 338 页。
描述：

__bit 内存属性覆盖所选数据模式给定的变量的默认存储，并将各个变量和常量放在可位寻址的内存中。您不能创建指向位存储器的指针。

储存信息：

●内存空间：内部数据内存空间

●地址范围：0x20 - 0x2F

●最大对象大小：1 位

●指针大小：不适用

示例：

```
__no_init __bit bool x;
```

另见：

内存类型，第 68 页。

__code

语法：

请参阅用于数据对象的类型属性的语法，第 338 页。

描述：

__code memory 属性覆盖所选数据模式给定的变量的默认存储，并将单个常量和字符串放置在代码内存中。您还可以使用 __code 属性来创建显式指向位于代码内存中的对象的指针。

储存信息：

●内存空间：代码内存空间

●地址范围：0x0 - 0xFFFF

●最大对象大小：64 KB

●指针大小：2 字节

示例：

```
__code const int x;
```

另见：

内存类型，第 68 页。

__data

语法：

请参阅用于数据对象的类型属性的语法，第 338 页。

描述：

__data memory 属性覆盖由所选数据模式给出的变量的默认存储，并将单个变量和常量放置在数据存储器中。

备注：

没有 __data 指针。数据存储器由 __idata 指针引用。

储存信息：

●内存空间：内部数据内存空间

●地址范围：0x0 - 0x7F

●最大对象大小：64 KB
●指针大小：1 字节，__idata 指针
示例：
__data int x;
另见：
内存类型，第 68 页。
__data_overlay
语法：
请参阅函数中使用的类型属性的语法，第 339 页。
描述：
__data_overlay 关键字将参数和自动变量放置在数据覆盖内存区域中。
备注：
此关键字仅在使用 Tiny 或 Small 数据模式时可用。
示例：
__data_overlay 无效 myfunction(void);
另见：
自动变量和参数的存储，第 83 页。
__ext_stack_reentrant
语法：
请参阅函数中使用的类型属性的语法，第 339 页。
描述：
__ext_stack_reentrant 关键字将参数和自动变量放在扩展堆栈上。
备注：
仅当指定了 --extended_stack 选项时才能使用此关键字。
示例：
__ext_stack_reentrant void myfunction(void);
另见：
自动变量和参数的存储，第 83 页，扩展堆栈，第 133 页和 --extended_stack，第 304 页。
__far
语法：
请参阅用于数据对象的类型属性的语法，第 338 页。
描述：
__far 内存属性覆盖所选数据模式给定的变量的默认存储，并将单个变量和常量放置在远内存中。
您还可以使用 __far 属性来创建显式指向位于远内存中的对象的指针。

备注：
此内存属性仅在使用 Far 数据模式时可用。
储存信息：
●存储空间：外部数据存储空间
●地址范围：0-0xFFFFFF
●最大对象大小：64 KB。对象不能跨越 64 KB 边界。
●指针大小：3 字节。仅对两个低字节执行算术运算，除了总是在整个 24 位地址上执行的比较。
示例：
__far int x;
另见：
内存类型，第 68 页。

__far_code

语法：
请参阅用于数据对象的类型属性的语法，第 338 页。
描述：
__far_code 内存属性覆盖所选数据模式给定的变量的默认存储，并将单个常量和字符串放置在远代码内存中。
您还可以使用 __far_code 属性来创建显式指向位于远代码内存中的对象的指针。
备注：
此内存属性仅在使用 Far 代码模式时可用。
储存信息：
●内存空间：代码内存空间
●地址范围：0-0xFFFFFF
●最大对象大小：64 KB。对象不能跨越 64 KB 边界。
●指针大小：3 字节。仅对两个低字节执行算术运算，除了总是在整个 24 位地址上执行的比较。
示例：
__far_code const int x;
另见：
内存类型，第 68 页。

__far_func

语法：
请参阅函数中使用的类型属性的语法，第 339 页。
描述：
__far_func 内存属性覆盖所选代码模式给出的函数的默认存储，并将各个函数放置在远内存中 - 使用真正的 24 位调用调用函数的内存。您还可以使用 __far_func 属性创建显式指向位于远内存中的对象的指针。

备注：
此内存属性仅在使用 Far 代码模式时可用。
储存信息：
●内存空间：代码内存空间
●地址范围：0-0xFFFFFF
●最大大小：64 KB。函数不能跨越 64 KB 边界。
●指针大小：3 字节
示例：
__far_func void myfunction(void);
另见：
函数存储的代码模式和内存属性，第 95 页。
__far_rom
语法：
请参阅用于数据对象的类型属性的语法，第 338 页。
描述：
__far_rom 内存属性覆盖所选数据模式给出的变量的默认存储，并将单个常量和字符串放置在位于远内存范围的 ROM 中。
您还可以使用 __far_rom 属性来创建显式指向位于 far_rom 内存中的对象的指针。
备注：
此内存属性仅在使用 Far 数据模式时可用。
储存信息：
●存储空间：外部数据存储空间
●地址范围：0-0xFFFFFF
●最大对象大小：64 KB。对象不能跨越 64 KB 边界。
●指针大小：3 字节。仅对两个低字节执行算术运算，除了总是在整个 24 位地址上执行的比较。
示例：
__far_rom const int x;
另见：
内存类型，第 68 页。
__far22
语法：
请参阅用于数据对象的类型属性的语法，第 338 页。
描述：
__far22 内存属性覆盖所选数据模式给定的变量的默认存储，并将单个变量和常量放置在 far22 内存中。
您还可以使用 __far22 属性来创建显式指向位于 far22 内存中的对象的指针。

备注：

此内存属性仅在使用 Far Generic 数据模式时可用。

储存信息：

●存储空间：外部数据存储空间

●地址范围：0-0x3FFFFF

●最大对象大小：64 KB。对象不能跨越 64 KB 边界。

●指针大小：3 字节。仅对两个低字节执行算术运算，除了总是在整个 22 位地址上执行的比较。

示例：

```
__far22 int x;
```

另见：

内存类型，第 68 页。

__far22_code

语法：

请参阅用于数据对象的类型属性的语法，第 338 页。

描述：

__far22_code 内存属性覆盖所选数据模式给定的变量的默认存储，并将单个常量和字符串放置在 far22 代码内存中。

您还可以使用 __far22_code 属性来创建显式指向位于 far22 代码内存中的对象的指针。

备注：

此内存属性仅在使用 Far Generic 代码模式时可用。

储存信息：

●内存空间：代码内存空间

●地址范围：0-0x3FFFFF

●最大对象大小：64 KB。对象不能跨越 64 KB 边界。

●指针大小：3 字节。仅对两个低字节执行算术运算，除了总是在整个 22 位地址上执行的比较。

示例：

```
__far22_code const int x;
```

另见：

内存类型，第 68 页。

__far22_rom

语法：

请参阅用于数据对象的类型属性的语法，第 338 页。

描述：

__far22_rom 内存属性覆盖所选数据模式给定的变量的默认存储，并将单个常量和字符串放在位于 far22 内存范围的 ROM 中。

您还可以使用 __far22_rom 属性创建一个指针，明确指向位于 far22_rom 内存中的对象。
备注：
此内存属性仅在使用 Far Generic 数据模式时可用。
储存信息：
●存储空间：外部数据存储空间
●地址范围：0-0x3FFFFF
●最大对象大小：64 KB。对象不能跨越 64 KB 边界。
●指针大小：3 字节。仅对两个低字节执行算术运算，除了总是在整个 22 位地址上执行的比较。
示例：
__far22_rom const int x;
另见：
内存类型，第 68 页。
__generic
语法：
请参阅用于数据对象的类型属性的语法，第 338 页。
描述：
__generic 指针属性指定可以访问内部数据存储空间、外部数据存储空间或代码存储空间中的数据的通用指针。
如果使用此关键字声明变量，它将位于外部数据存储空间中。
备注：
使用 Far 数据模式时，此内存属性不可用。
示例：
int __generic * ptr;
另见：
通用指针，第 332 页。
__huge
语法：
请参阅用于数据对象的类型属性的语法，第 338 页。
描述：
__huge memory 属性覆盖所选数据模式给定的变量的默认存储，并将单个变量和常量放置在巨大的内存中。您还可以使用 __huge 属性来创建显式指向位于巨大内存中的对象的指针。
备注：
此内存属性仅在使用 Far 数据模式时可用。
储存信息：
●存储空间：外部数据存储空间
●地址范围：0-0xFFFFFF

●最大对象大小：16 MB
●指针大小：3 字节
示例：
__huge int x;
另见：
内存类型，第 68 页。
__huge_code
语法：
请参阅用于数据对象的类型属性的语法，第 338 页。
描述：
__huge_code 内存属性覆盖所选数据模式给定的变量的默认存储，并将单个常量和字符串放置在巨大的代码内存中。
您还可以使用 __huge_code 属性创建一个指针，明确指向位于巨大代码内存中的对象。
备注：
此内存属性仅在使用 Far 代码模式时可用。
储存信息：
●内存空间：代码内存空间
●地址范围：0-0xFFFFFF
●最大对象大小：16 MB
●指针大小：3 字节
示例：
__huge_code const int x;
另见：
内存类型，第 68 页。
__huge_rom
语法：
请参阅用于数据对象的类型属性的语法，第 338 页。
描述：
__huge_rom 内存属性覆盖由所选数据模式给出的变量的默认存储，并将单个常量和字符串放置在位于巨大内存范围的 ROM 中。
您还可以使用 __huge_rom 属性创建一个指针，明确指向位于 huge_rom 内存中的对象。
备注：
此内存属性仅在使用 Far 数据模式时可用。
储存信息：
●存储空间：外部数据存储空间
●地址范围：0-0xFFFFFF
●最大对象大小：16 MB

●指针大小：3 字节
示例：
__huge_rom const int x;
另见：
内存类型，第 68 页。
__idata
语法：
请参阅用于数据对象的类型属性的语法，第 338 页。
描述：
__idata 内存属性覆盖所选数据模式给定的变量的默认存储，并将单个变量和常量放置在 idata 内存中。
您还可以使用 __idata 属性来创建显式指向位于 idata 内存中的对象的指针。
储存信息：
●内存空间：内部数据内存空间
●地址范围：0x0 - 0xFF
●最大对象大小：256 字节
●指针大小：1 字节
示例：
__idata int x;
另见：
内存类型，第 68 页。
__idata_overlay
语法：
请参阅函数中使用的类型属性的语法，第 339 页。
描述：
__idata_overlay 关键字将参数和自动变量放置在 idata 覆盖内存区域中。
备注：
此关键字仅在使用 Tiny 或 Small 数据模式时可用。
示例：
__idata_overlay void myfunction(void);
另见：
自动变量和参数的存储，第 83 页。
__idata_reentrant
语法：
请参阅函数中使用的类型属性的语法，第 339 页。
描述：

__idata_reentrant 关键字将参数和自动变量放在 idata 堆栈上。

示例：

```
__idata_reentrant void myfunction(void);
```

另见：

自动变量和参数的存储，第 83 页，Idata 堆栈，第 133 页

__ixdata

语法：

请参阅用于数据对象的类型属性的语法，第 338 页。

描述：

__ixdata 内存属性覆盖所选数据模式给定的变量的默认存储，并将单个变量和常量放置在数据内存中。 __ixdata 内存属性需要支持片上外部数据 (xdata) 的设备。

备注：

没有 __ixdata 指针。数据存储器由 __xdata 指针引用。

储存信息：

●存储空间：外部数据存储空间

●地址范围：0x0 - 0xFFFF

●最大对象大小：64 KB

●指针大小：2 字节，__xdata 指针

示例：

```
__ixdata int x;
```

另见：

内存类型，第 68 页。

__interrupt

语法：

请参阅函数中使用的类型属性的语法，第 339 页。

描述：

__interrupt 关键字指定中断函数。要指定一个或多个中断向量，请使用#pragma vector 指令。中断向量的范围取决于所使用的设备。可以在没有向量的情况下定义中断函数，但是编译器不会在中断向量表中生成条目。

中断函数必须具有 void 返回类型并且不能有任何参数。

头文件 iodevice.h，其中 device 对应于所选设备，包含现有中断向量的预定义名称。

为确保中断处理程序尽可能快地执行，您应该使用 -Ohs 对其进行编译，或者如果使用另一个优化目标编译模块，则使用 #pragma optimize=speed。

示例：

```
__interrupt void my_interrupt_handler(void);
```

另见：

中断函数，第 97 页，向量，第 381 页，以及 INTVEC，第 430 页。

__intrinsic

描述：

__intrinsic 关键字仅供编译器内部使用。

__monitor

语法：

请参阅函数中使用的类型属性的语法，第 339 页。

描述：

__monitor 关键字导致在函数执行期间禁用中断。这允许执行原子操作，例如控制多个进程对资源的访问的信号量上的操作。使用 __monitor 关键字声明的函数在所有其他方面都等同于任何其他函数。

示例：

```
__monitor int get_lock(void);
```

另见：

监控函数，第 99 页。有关相关内部函数的信息，请分别参见 __disable_interrupt，第 383 页，__enable_interrupt，第 384 页，__get_interrupt_state，第 384 页和 __set_interrupt_state，第 385 页。

__near_func

语法：

请参阅函数中使用的类型属性的语法，第 339 页。

描述：

__near_func 内存属性覆盖所选代码模式给定的函数的默认存储，并将单个函数放置在近内存中。在 分页代码模式中，使用 __near_func 属性将函数显式放置在根区域中。

储存信息：

- 内存空间：代码内存空间
- 地址范围：0-0xFFFF
- 最大大小：64 KB
- 指针大小：2 字节

示例：

```
__near_func 无效 myfunction(void);
```

另见：

函数存储的代码模式和内存属性，第 95 页。

__no_alloc, __no_alloc16

语法：

请参阅对象属性的语法，第 340 页。

描述：

在常量上使用 __no_alloc 或 __no_alloc16 对象属性可以使该常量在可执行文件中可用，而不会占用链接应用程序中的任何空间。

您无法从应用程序访问此类常量的内容。您可以获取它的地址，它是常量段的整数偏移量。使用 __no_alloc 时偏移量的类型为 unsigned long，当使用 __no_alloc 时为 unsigned short

__no_alloc16 被使用。

示例：

```
__no_alloc const struct MyData my_data @ "XXX" = {    ...};
```

另见：

__no_alloc_str，__no_alloc_str16，第 355 页。

__no_alloc_str, __no_alloc_str16

语法：

__no_alloc_str(string_literal @segment) 和
__no_alloc_str16(string_literal @segment)

在哪里

字符串字面量

您希望在可执行文件中可用的字符串文字。

部分

放置字符串文字的段的名称。

描述：

使用 __no_alloc_str 或 __no_alloc_str16 运算符使字符串文字在可执行文件中可用，而不会占用链接应用程序中的任何空间。

表达式的值是段中字符串文字的偏移量。对于 __no_alloc_str，偏移量的类型是 unsigned long。对于 __no_alloc_str16，偏移量的类型是 unsigned short。

示例：

```
#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);
}
```

根据您的调试器和运行时支持，这可能会在主机上产生跟踪输出。请注意，C-SPY 中没有这样的运行时支持，除非您使用外部插件模块。

另见：

__no_alloc，__no_alloc16，第 355 页。

__no_init

语法：

请参阅对象属性的语法，第 340 页。

描述：

使用 __no_init 关键字将数据对象放置在非易失性内存中。这意味着变量的初始化（例如在系统启动时）被禁止。

示例：

```
__no_init int myarray[10];
```

另见：

未初始化的变量，第 274 页。

__noreturn

语法：

请参阅对象属性的语法，第 340 页。

描述：

__noreturn 关键字可用于函数上，以通知编译器该函数不会返回。如果在此类函数上使用 this 关键字，编译器可以更有效地优化。不返回的函数示例有 abort 和 exit。

备注：

在中等或高优化级别，__noreturn 关键字可能会在任何可以确定当前函数无法返回的点上导致错误的调用堆栈调试信息。

示例：

```
__noreturn 无效终止（无效）；
```

__overlay_near_func

描述：

__overlay_near_func 关键字仅供编译器内部使用。

__pdata

语法：

请参阅用于数据对象的类型属性的语法，第 338 页。

描述：

__pdata 内存属性覆盖所选数据模式给定的变量的默认存储，并将单个变量和常量放置在 pdata 内存中。

您还可以使用 __pdata 属性来创建显式指向位于 pdata 内存中的对象的指针。

储存信息：

●存储空间：外部数据存储空间

●地址范围：0x0 - 0xFF
●最大对象大小：256 字节
●指针大小：1 字节
示例：
__pdata int x;
另见：
内存类型，第 68 页。
__pdata_reentrant
语法：
请参阅函数中使用的类型属性的语法，第 339 页。
描述：
__pdata_reentrant 关键字将参数和自动变量放在 pdata 堆栈上。
示例：
__pdata_reentrant void myfunction(void);
另见：
自动变量和参数的存储，第 83 页，Pdata 堆栈，第 134 页
__root
语法：
请参阅对象属性的语法，第 340 页。
描述：
具有 __root 属性的函数或变量无论是否被应用程序的其余部分引用，都将被保留，前提是它的模块被包含在内。始终包含程序模块，仅在需要时才包含库模块。
示例：
__root int myarray[10];
另见：
有关模块、段和链接过程的更多信息，请参阅 IAR 链接器和库工具参考指南。
__ro_placement
语法：
请参阅对象属性的语法，第 340 页。
描述：
__ro_placement 属性指定数据对象应放置在只读内存中。在两种情况下，您可能希望使用此对象属性：
● 声明为 const volatile 的数据对象默认放置在读写内存中。使用 __ro_placement 对象属性将数据对象放置在只读内存中。

●在 C++中，声明为 const 且需要动态初始化的数据对象被放置在读写内存中，并在系统启动时进行初始化。如果使用 ro_placement 对象属性，如果数据对象需要动态初始化，编译器将给出错误消息。

您只能在 const 对象上使用 __ro_placement 对象属性。

在某些情况下（主要涉及简单的构造函数），编译器将能够将数据对象的 C++ 动态初始化优化为静态初始化。在这种情况下，不会为该对象发出错误消息。

示例：

```
__ro_placement const volatile int x = 10;
```

__sfr

语法：

请参阅用于数据对象的类型属性的语法，第 338 页。

描述：

__sfr 内存属性覆盖所选数据模式给定的变量的默认存储，并将单个变量和常量放置在 SFR 内存中。

您不能创建指向位于 SFR 内存中的对象的指针。

储存信息：

●内存空间：内部数据内存空间

●地址范围：0x80 - 0xFF

●最大对象大小：128 字节

●指针大小：不适用

示例：

```
__sfr int x;
```

另见：

内存类型，第 68 页。

__task

语法：

请参阅函数中使用的类型属性的语法，第 339 页。

描述：

该关键字允许函数放宽保存寄存器的规则。通常，该关键字用于 RTOS 中任务的启动函数。

默认情况下，函数在进入时将使用的保留寄存器的内容保存在堆栈中，并在退出时恢复它们。声明为 __task 的函数不会保存所有寄存器，因此需要较少的堆栈空间。

因为声明为 __task 的函数会损坏调用函数所需的寄存器，所以您应该只对不返回或从汇编代码调用此类函数的函数使用 __task。

函数 main 可以声明为 __task，除非它是从应用程序显式调用的。在具有多个任务的实时应用程序中，可以将每个任务的根函数声明为 __task。

示例：

```
__task void my_handler(void);
```

__xdata

语法：

请参阅用于数据对象的类型属性的语法，第 338 页。

描述：

__xdata 内存属性覆盖所选数据模式给定的变量的默认存储，并将单个变量和常量放置在 xdata 内存中。

您还可以使用 __xdata 属性来创建显式指向位于 xdata 内存中的对象的指针。

储存信息：

●存储空间：外部数据存储空间

●地址范围：0x0 - 0xFFFF

●最大对象大小：64 KB

●指针大小：2 字节

示例：

```
__xdata int x;
```

另见：

内存类型，第 68 页。

__xdata_reentrant

语法：

请参阅函数中使用的类型属性的语法，第 339 页。

描述：

__xdata_reentrant 关键字将参数和自动变量放在 xdata 堆栈上。

示例：

```
__xdata_reentrant void myfunction(void);
```

另见：

自动变量和参数的存储，第 83 页，Xdata 堆栈，第 134 页

__xdata_rom

语法：

请参阅用于数据对象的类型属性的语法，第 338 页。

描述：

__xdata_rom 内存属性覆盖所选数据模式给定的变量的默认存储，并将单个常量和字符串放在位于 xdata 内存范围的 ROM 中。

您还可以使用 __xdata_rom 属性创建一个指针，明确指向位于 xdata_rom 内存中的对象。

储存信息：

●存储空间：外部数据存储空间

●地址范围：0-0xFFFF

●最大对象大小：64 KB
●指针大小：2 字节
示例：
__xdata_rom const int x;
另见：
内存类型，第 68 页。

（空白页）
页码:360

Pragma 指令

- pragma 指令摘要
- pragma 指令的说明

Pragma 指令摘要

#pragma 指令由标准 C 定义，是一种以受控方式使用供应商特定扩展以确保源代码仍然可移植的机制。
pragma 指令控制编译器的行为，例如它如何为变量和函数分配内存、是否允许扩展关键字以及是否输出警告消息。
编译指令总是在编译器中启用。
下表列出了编译器的编译指令，可以与 #pragma 预处理指令或 _Pragma() 预处理运算符一起使用：

Pragma 指令	描述
basic_template_matching	使模板函数完全感知内存属性。
bitfields	控制位域成员的顺序。
constseg	将常量变量放在命名段中。
cstat_disable	请参阅 C-STAT® 静态分析指南。
cstat_enable	请参阅 C-STAT® 静态分析指南。
cstat_restore	请参阅 C-STAT® 静态分析指南。
cstat_suppress	请参阅 C-STAT® 静态分析指南。
data_alignment	为变量提供更高（更严格）的对齐方式。
dataseg	将变量放在命名段中。
default_function_attributes	为函数的声明和定义设置默认类型和对象属性。
default_variable_attributes	为变量的声明和定义设置默认类型和对象属性。
diag_default	更改诊断消息的严重性级别。
diag_error	更改诊断消息的严重性级别。
diag_remark	更改诊断消息的严重性级别。
diag_suppress	抑制诊断消息。
diag_warning	更改诊断消息的严重性级别。
error	解析时发出错误信号。

include_alias	指定包含文件的别名。
inline	控制函数的内联。
language	控制 IAR Systems 语言扩展。
location	指定变量的绝对地址，或将一组函数或变量放在命名段中。
message	打印一条消息。
object_attribute	将对象属性添加到变量或函数的声明或定义中。
optimize	指定优化的类型和级别。
__printf_args	验证是否使用正确的参数调用了具有 printf 样式格式字符串的函数。
public_equ	定义一个公共汇编标签并给它一个值。
register_bank	为所需的中断功能设置寄存器组 确保链接输出中包含另一个符号所需的符号。
rtmodel	向模块添加运行时模型属性。
__scanf_args	验证是否使用正确的参数调用了具有 scanf 样式格式字符串的函数。
section	该指令是#pragma 段的别名。
segment	声明内部函数要使用的段名称。
STDC CX_LIMITED_RANGE	指定编译器是否可以使用普通的复杂数学公式。
STDC FENV_ACCESS	指定您的源代码是否访问浮点环境。
STDC FP_CONTRACT	指定是否允许编译器收缩浮点表达式。
vector	指定中断或陷阱函数的向量。
weak	使定义成为弱定义，或为函数或变量创建弱别名。
type_attribute	将类型属性添加到声明或定义中。

表 40：Pragma 指令汇总

注意：出于可移植性原因，另请参见公认的编译指示指令 (6.10.6)，第 449 页。

Pragma 指令的描述

本节提供有关每个 pragma 指令的详细信息。

basic_template_matching

语法：
#pragma basic_template_matching
描述：
在模板函数声明之前使用这个 pragma 指令可以使函数完全感知内存属性，在极少数情况下这很有用。
然后，该模板函数将匹配模板而不进行修改，请参见模板和数据存储器属性，第 239 页。
例子：

```
#pragma basic_template_matching template<typename T> void fun(T *);
void MyF()
{
fun((int    pdata *) 0); // T = int  pdata
}
```

bitfields

语法：
#pragma bitfields={reversed|default}
参数：
reversed 位域成员从最高有效位到最低有效位放置。
default 位域成员从最低有效位到最高有效位放置。
描述：
使用此 pragma 指令来控制位域成员的顺序。
例子：

```
#pragma bitfields=reversed
/* 使用反转位域的结构。 */ struct S
{
unsigned char error : 1; unsigned char size : 4; unsigned short code  :
10;
};
#pragma bitfields=default /* 恢复到默认设置 */
```

另见：
位域，第 327 页。

constseg

语法：

```
#pragma constseg=[__memoryattribute ]{SEGMENT_NAME|default}
```

参数：

__memoryattribute 一个可选的内存属性，表示段将被放置在什么内存中；如果未指定，则使用默认内存。

SEGMENT_NAME 用户定义的段名； 不能是预定义供编译器和链接器使用的段名。

default 使用常量的默认段。

描述：

使用此 pragma 指令将常量变量放置在命名段中。 段名不能是预定义供编译器和链接器使用的段名。 该设置保持活动状态，直到您使用 #pragma constseg=default 指令再次将其关闭。

使用#pragma constseg 指令放置在命名段中的常量必须位于 ROM 内存中。当常量位于 code 或 data_rom 中时就是这种情况。 否则，必须使用适当的内存属性明确指定段应该驻留的内存。

备注：

可以使用#pragma dataseg 指令将位于数据存储器中的未初始化常量段放置在命名段中。

例子：

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

data_alignment

语法：

#pragma data_alignment=expression

参数：

表达式 必须是 2 的幂（1、2、4 等）的常数。

描述：

使用此 pragma 指令为紧随其后的变量提供比其他情况下更高（更严格）的起始地址对齐方式。 该指令可用于具有静态和自动存储持续时间的变量。

当您对具有自动存储持续时间的变量使用此指令时，每个函数的允许对齐有一个上限，由使用的调用约定确定。

备注：

通常，变量的大小是其对齐方式的倍数。 data_alignment 指令仅影响变量起始地址的对齐，而不影响其大小，因此可用于创建大小不是对齐倍数的情况。

dataseg

语法：

#pragma dataseg=[__memoryattribute]{SEGMENT_NAME|default}

参数：

__memoryattribute 一个可选的内存属性，表示段将被放置在什么内存中；如果未指定，则使用默认内存。

SEGMENT_NAME 用户定义的段名； 不能是预定义供编译器和链接器使用的段名。

default 使用默认段。

描述：

使用此 pragma 指令将变量放置在命名段中。 段名不能是预定义供编译器和链接器使用的段名。 该变量在启动时不会被初始化，因此可以没有初始化器，这意味着它必须声明为 __no_init。 在您使用 #pragma dataseg=default 指令再次将其关闭之前，该设置将保持活动状态。

例子：

```
#pragma dataseg=__pdata MY_SEGMENT
__no_init char myBuffer[1000];
#pragma dataseg=default
```

default_function_attributes

语法：

#pragma default_function_attributes=[attribute...]

其中属性可以是：

type_attribute object_attribute

@ segment_name

参数：

type_attribute 请参见类型属性，第 337 页。

object_attribute 请参见对象属性，第 340 页。

@segment_name 请参见段中的数据和函数放置，第 262 页。

描述：

使用此 pragma 指令为函数声明和定义设置默认段放置、类型属性和对象属性。 默认设置仅用于未以其他方式指定类型或对象属性或位置的声明和定义。 指定不带属性的 default_function_attributes pragma 指令会恢复未将此类默认值应用于函数声明和定义的初始状态。

例子：

```
/* 将以下函数放在段 MYSEG" */
#pragma default_function_attributes = @ "MYSEG" int fun1(int x)
{ return x + 1; }
int fun2(int x) { return x - 1;
/* 停止将函数放入 MYSEG */
#pragma default_function_attributes =
```

与以下效果相同：

```
int fun1(int x) @ "MYSEG" { return x + 1; } int fun2(int x) @ "MYSEG"
{ return x - 1; }
```

另见：

location, 第 372 页

object_attribute, 第 373 页

type_attribute, 第 380 页

default_variable_attributes

语法：

#pragma default_variable_attributes=[attribute...]

其中属性可以是：

type_attribute object_attribute

@ segment_name

参数：

type_attribute 请参见类型属性，第 337 页。

object_attributes 请参见对象属性，第 340 页。

@segment_name 请参见段中的数据和函数放置，第 262 页。
描述：
使用此 pragma 指令为具有静态存储持续时间的变量的声明和定义设置默认段放置、类型属性和对象属性。

默认设置仅用于未以其他方式指定类型或对象属性或位置的声明和定义。

指定不带属性的 default_variable_attributes pragma 指令将恢复初始状态，即没有将此类默认值应用于具有静态存储持续时间的变量。
例子：
```
/* 将以下变量放在段 MYSEG" */
#pragma default_variable_attributes = @ "MYSEG" int var1 = 42;
int var2 = 17;
/* 停止将变量放入 MYSEG */
#pragma default_variable_attributes =
```
与以下效果相同：
```
int var1 @ "MYSEG" = 42; int var2 @ "MYSEG" = 17;
```
另见：
地点，第 372 页； 对象属性，第 373 页； type_attribute，第 380 页

diag_default

语法：
```
#pragma diag_default=tag[,tag,...]
```
参数：
tag 诊断消息的编号，例如消息编号 Pe177。
描述：
使用此 pragma 指令将严重性级别更改回默认值，或更改为命令行上通过任何选项 --diag_error、--diag_remark、--diag_suppress 或 --diag_warnings 定义的严重性级别，用于诊断消息 用标签指定。 此级别一直有效，直到被另一个诊断级别的 pragma 指令更改。
另见：
诊断，第 283 页。

diag_error

语法：
```
#pragma diag_error=tag[,tag,...]
```

参数：
tag 诊断消息的编号，例如消息编号 Pe177。
描述：
使用此 pragma 指令将指定诊断的严重性级别更改为错误。 此级别一直有效，直到被另一个诊断级别的 pragma 指令更改。
另见：
诊断，第 283 页。

diag_remark

语法：
#pragma diag_remark=tag[,tag,...]
参数：
tag 诊断消息的编号，例如消息编号 Pe177。
描述：
使用此 pragma 指令更改严重性级别以备注指定的诊断消息。 此级别一直有效，直到被另一个诊断级别的 pragma 指令更改。
另见：
诊断，第 283 页。

diag_suppress

语法：
#pragma diag_suppress=tag[,tag,...]
参数：
tag 诊断消息的编号，例如消息编号 Pe117。
描述：
使用此 pragma 指令来抑制指定的诊断消息。 此级别一直有效，直到被另一个诊断级别的 pragma 指令更改。
另见：
诊断，第 283 页。

diag_warning
语法：
#pragma diag_warning=tag[,tag,...]
参数：
tag 诊断消息的编号，例如消息编号 Pe826。
描述：
使用此 pragma 指令将指定诊断消息的严重性级别更改为警告。 此级别一直有效，直到被另一个诊断级别的 pragma 指令更改。
另见：
诊断，第 283 页。

error
语法：
#pragma error message
参数：
message 表示错误消息的字符串。
描述：
使用此 pragma 指令可在解析时产生错误消息。 此机制与预处理器指令#error 不同，因为#pragma 错误指令可以使用指令的_Pragma 形式包含在预处理器宏中，并且仅在使用宏时才会导致错误。
例子：
```
#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
语法：
#pragma include_alias ("orig_header" , "subst_header")
#pragma include_alias (<orig_header> , <subst_header>)
参数：

orig_header 要为其创建别名的头文件的名称。
subst_header 原始头文件的别名。
描述：
使用此 pragma 指令为头文件提供别名。 这对于用另一个头文件替换一个头文件以及指定相对文件的绝对路径很有用。
此 pragma 指令必须出现在相应的 #include 指令之前，并且 subst_header 必须与其相应的 #include 指令完全匹配。
例子：

```
#pragma include_alias (<stdio.h> , <C:\MyHeaders\stdio.h>)
#include <stdio.h>
```

此示例将使用根据指定路径定位的对应文件替换相关文件 stdio.h。

另见：
包括文件搜索程序，第 280 页。

inline

语法：

```
#pragma inline[=forced|=never]
```

参数：
无参数 与 inline 关键字的作用相同。
强制禁用编译器的启发式和强制内联。
never 禁用编译器的启发式并确保函数不会被内联。
描述：
使用#pragma inline 来告知编译器在指令之后立即定义的函数应该根据 C++ 内联语义进行内联。
指定 #pragma inline=forced 将始终内联定义的函数。 如果编译器由于某种原因（例如由于递归）未能内联函数，则会发出警告消息。
内联通常仅在高优化级别上执行。 指定 #pragma inline=forced 将在中等优化级别启用函数的内联。
另见：
内联函数，第 103 页。

language

语法：

```
#pragma language={extended|default|save|restore}
```

参数：

扩展 从第一次使用 pragma 指令起启用 IAR Systems 语言扩展，并继续。

默认值 从第一次使用 pragma 指令起，将 IAR Systems 语言扩展的设置恢复为编译器选项指定的任何设置。

save|restore 分别保存和恢复围绕一段源代码设置的 IAR Systems 语言扩展设置。

每次使用 save 之后都必须在同一文件中进行匹配的还原，而无需任何干预 #include 指令。

描述：

使用此 pragma 指令来控制语言扩展的使用。

例子：

在需要在启用 IAR Systems 扩展的情况下编译的文件的顶部：

```
#pragma language=extended
/* 文件的其余部分。 */
```

围绕需要在启用 IAR Systems 扩展的情况下编译的源代码的特定部分，但不能假定序列之前的状态与使用的编译器选项指定的状态相同：

```
#pragma language=save
#pragma language=extended
/* 部分源代码。 */
#pragma language=restore
```

另见：

-e, 第 302 页 及 --strict, 第 320 页。

location

语法：

```
#pragma location={address|NAME}
```

参数：

address 需要绝对位置的全局或静态变量的绝对地址。

NAME 用户定义的段名； 不能是预定义供编译器和链接器使用的段名。

描述：

使用此 pragma 指令来指定其声明遵循 pragma 指令的全局或静态变量的位置（绝对地址）。 该变量必须声明为 __no_init 或 const。

或者，该指令可以采用一个字符串，该字符串指定一个用于放置变量或函数的段，其声明遵循 pragma 指令。

不要将通常位于不同段中的变量（例如，声明为 __no_init 的变量和声明

为 const 的变量）放在同一个命名段中。

例子：

```
#pragma location=0xFF20
  no_init volatile char PORT1; /* PORT1 位于地址 0xFF20 */
#pragma segment="FLASH"
#pragma location="FLASH"
  no_init char PORT2; /* PORT2 位于 FLASH 段 */
/* 更好的方法是使用相应的机制 */
#define FLASH _Pragma("location=\"FLASH\"")
/* ... */
FLASH    no_init int i; /* i 放在 FLASH 段 */
```

另见：

控制内存中的数据和函数放置，第 259 页和放置用户定义的段，第 132 页。

message

语法：

```
#pragma message(message)
```

参数：

message 要定向到标准输出流的消息。

描述：

使用此 pragma 指令使编译器在编译文件时将消息打印到标准输出流。

例子：

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

object_attribute

语法：

```
#pragma object_attribute=object_attribute[ object_attribute...]
```

参数：

有关可与此 pragma 指令一起使用的对象属性的信息，请参阅第 340 页的对象属性。

描述：

使用此 pragma 指令将一个或多个 IAR 特定对象属性添加到变量或函数的声明或定义中。 对象属性影响实际的变量或函数，而不是它的类型。 定义变量或函数时，将使用所有声明（包括定义）中的对象属性的并集。

例子：

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

相当于：

```
__no_init char bar;
```

另见：

扩展关键字的一般语法规则，第 337 页。

optimize

语法：

```
#pragma optimize=[goal][level][no_optimization...]
```

参数：

goal

请选择：

size，优化尺寸

balanced，优化速度和尺寸速度之间的平衡，优化速度。

no_size_constraints，优化速度，但放宽代码大小扩展的正常限制。

level 指定优化级别； 在无、低、中或高之间进行选择。

no_optimization

禁用一项或多项优化；请选择：

no_code_motion，禁用代码运动 no_crosscall，禁用过程间交叉调用

no_cse，禁用公共子表达式消除 no_inline，禁用函数内联

no_tbaa，禁用基于类型的别名分析

no_unroll，禁用循环展开

描述：

使用这个 pragma 指令来降低优化级别，或者关闭一些特定的优化。此 pragma 指令仅影响紧跟在该指令之后的函数。

参数 size、balanced、speed 和 no_size_constraints 仅对高优化级别有影响，并且只能使用其中之一，因为不可能同时优化 speed 和 size。

也不能使用嵌入在这个 pragma 指令中的预处理器宏。预处理器不会扩展任何此类宏。

备注：

如果您使用#pragma optimize 指令指定的优化级别高于您使用编译器选项指定的优化级别，则忽略 pragma 指令。

例子：

```
#pragma optimize=speed int SmallAndUsedOften()
{
/* 在这里做点什么。（其他任务） */
}
#pragma optimize=size int BigAndSeldomUsed()
{
/* 在这里做点什么。（其他任务） */
}
```

另见：

微调启用的转换，第 266 页。

__printf_args

语法：

```
#pragma __printf_args
```

描述：

在具有 printf 样式格式字符串的函数上使用此 pragma 指令。 对于对该函数的任何调用，编译器都会验证每个转换说明符（例如 %d）的参数在语法上是否正确。

例子：

```
#pragma   printf_args
int printf(char const *,...);
void PrintNumbers(unsigned short x)
{
printf("%d", x); /* 编译器检查 x 是否为整数 */
}
```

public_equ

语法：

```
#pragma public_equ="symbol",value
```

参数：

symbol 要定义的汇编器符号的名称（字符串）。

value 定义的汇编符号的值（整数常量表达式）。

描述：

使用这个 pragma 指令来定义一个公共汇编器标签并给它一个值。
例子：
```
#pragma public_equ="MY_SYMBOL",0x123456
```
另见：
--public_equ，第 318 页。

register_bank

语法：
```
#pragma register_bank=(0|1|2|3)
```
参数：
0|1|2|3 要使用的寄存器组的编号。
描述：
使用此 pragma 指令指定要由在 pragma 指令之后声明的中断函数使用的寄存器组。

指定寄存器组后，中断功能切换到指定的寄存器组。因此，寄存器 R0-R7 不必单独保存在堆栈中。 结果是更小更快的中断序言和结语。

已使用的寄存器组所占用的内存不能用于其他数据。
备注：

可以相互中断的中断不能使用相同的寄存器组，因为这会导致寄存器被无意破坏。如果未指定寄存器组，则中断函数将使用默认组。

例子：
```
#pragma register_bank=2
__interrupt void my_handler(void);
```

required

语法：
```
#pragma required=symbol
```
参数：
symbol 任何静态链接的函数或变量。
描述：

使用此 pragma 指令可确保链接输出中包含第二个符号所需的符号。 该指令必须紧挨在第二个符号之前。如果对符号的要求在应用程序中不可见，则使用该指令，例如，如果变量仅通过其所在的段间接引用。

例子：

```
const char copyright[] = "Copyright by me";
#pragma required=copyright int main()
{
/* 在这里做点什么。（其他任务） */
}
```

即使应用程序没有使用版权字符串，它仍然会被链接器包含并在输出中可用。

另见：

内联汇编器，第 193 页

rtmodel

语法：

```
#pragma rtmodel="key","value"
```

参数：

key 指定运行时模型属性的文本字符串。

value 一个文本字符串，它指定运行时模型属性的值。 使用特殊值 * 相当于根本不定义属性。

描述：

使用此 pragma 指令向模块添加运行时模型属性，链接器可以使用该属性检查模块之间的一致性。

这个 pragma 指令对于强制模块之间的一致性很有用。 链接在一起并定义相同运行时属性键的所有模块必须具有相同的对应键值，或特殊值 *。

但是，明确声明模块可以处理任何运行时模型可能很有用。一个模块可以有多个运行时模型定义。

备注：

预定义的编译器运行时模型属性以双下划线开头。 为避免混淆，此样式不得在用户定义的属性中使用。

例子：

```
#pragma rtmodel="I2C","ENABLED"
```

如果包含此定义的模块与未定义相应运行时模型属性的模块链接，则链接器将生成错误。

另见：

检查模块一致性，第 141 页。

__scanf_args

语法：

#pragma __scanf_args

描述：

在具有 scanf 样式格式字符串的函数上使用此 pragma 指令。 对于对该函数的任何调用，编译器都会验证每个转换说明符（例如 %d）的参数在语法上是否正确。

例子：

```
#pragma  scanf_args
int scanf(char const *,...);
int GetNumber()
{
int nr;
scanf("%d", &nr);  /* 编译器检查参数是一个指向整数的指针 */
return nr;
}
```

segment

语法：

#pragma segment="NAME" [__memoryattribute] [align]

alias

#pragma section="NAME" [__memoryattribute] [align]

参数：

NAME 段的名称。

__memoryattribute 一个可选的内存属性，标识段将被放置的内存； 如果未指定，则使用默认内存。

align 指定段的对齐方式。 该值必须是 2 次方的常量整数表达式。

描述：

使用此 pragma 指令定义可由段运算符 __segment_begin、__segment_end 和 __segment_size 使用的段名称。

特定段的所有段声明必须具有相同的内存类型属性和对齐方式。

align 和 memoryattribute 参数仅在与段运算符 __segment_begin、__segment_end 和 __segment_size 一起使用时才相关。
如果您考虑对单个变量使用 align 以实现更高的对齐，则必须改为使用 #pragma data_alignment 指令。
如果使用可选的内存属性，则段运算符 __segment_begin 和 __segment_end 的返回类型为：void __memoryattribute *.
备注：
要将变量或函数放置在特定段中，请使用 #pragma location 指令或 @ 运算符。
例子：

```
#pragma segment="MYPDATA" __pdata 4
```

另见：
专用段运算符，第 225 页和章节链接概述和链接您的应用程序。

STDC CX_LIMITED_RANGE

语法：

```
#pragma STDC CX_LIMITED_RANGE {ON|OFF|DEFAULT}
```

参数：
ON 可以使用普通复数数学公式。
OFF 不能使用普通的复杂数学公式。
DEFAULT 设置默认行为，即关闭。
描述：
使用此 pragma 指令指定编译器可以对 *（乘法）、/（除法）和 abs 使用普通的复杂数学公式。
备注：
标准 C 需要该指令。该指令被识别但在编译器中无效。

STDC FENV_ACCESS

语法：

```
#pragma STDC FENV_ACCESS {ON|OFF|DEFAULT}
```

参数：
ON 源代码访问浮点环境。 请注意，编译器不支持此参数。
OFF 源代码不访问浮点环境。

DEFAULT 设置默认行为，即关闭。
描述：

使用此 pragma 指令指定您的源代码是否访问浮点环境。

备注：
标准 C 要求该指令。

STDC FP_CONTRACT

语法：
#pragma STDC FP_CONTRACT {ON|OFF|DEFAULT}
参数：
ON 允许编译器收缩浮点表达式。

OFF 不允许编译器收缩浮点表达式。 请注意，编译器不支持此参数。

DEFAULT 设置默认行为，即 ON。
描述：
使用此 pragma 指令指定是否允许编译器收缩浮点表达式。 标准 C 要求该指令。
例子：
#pragma STDC FP_CONTRACT=ON

type_attribute

语法：
#pragma type_attribute=type_attr[type_attr...]
参数：
有关可与此 pragma 指令一起使用的类型属性的信息，请参阅 类型属性，第 337 页。
描述：
使用这个 pragma 指令来指定 IAR 特定的类型属性，它们不是标准 C 的一部分。但是请注意，给定的类型属性可能不适用于所有类型的对象。 该指令影响标识符、下一个变量或紧跟在 pragma 指令之后的下一个函数的声明。
例子：
在此示例中，定义了具有内存属性 __pdata 的 int 对象：
#pragma type_attribute=__pdata int x;

这个使用扩展关键字的声明是等价的：
__pdata int x;
另见：
扩展关键字一章。

vector
语法：
#pragma vector=vector1[, vector2, vector3, ...]
参数：
vectorN 中断函数的向量编号。
描述：
使用此 pragma 指令来指定函数的向量，该函数的声明遵循 pragma 指令。请注意，可以为每个函数定义多个向量。
例子：
#pragma vector=0x14
__interrupt void my_handler(void);

weak
语法：
#pragma weak symbol1[=symbol2]
参数：
symbol1 具有外部链接的函数或变量。
symbol2 定义的函数或变量。
描述：
此 pragma 指令可以通过以下两种方式之一使用：

● 使具有外部链接的函数或变量的定义成为弱定义。

●为另一个函数或变量创建一个弱别名。 您可以为同一个函数或变量创建多个别名。
例子：
要使 foo 的定义成为弱定义，请编写：
#pragma weak foo
要使 NMI_Handler 成为 Default_Handler 的弱别名，请编写：

```
#pragma weak NMI_Handler=Default_Handler
```

如果 NMI_Handler 没有在程序的其他地方定义，所有对 NMI_Handler 的引用都将引用 Default_Handler。

（空白页）

内联函数

- 内联功能总结
- 内联功能说明

内联函数总结

内联函数提供对低级处理器操作的直接访问，并且在时间要求严格的例程中非常有用。 内部函数编译 内联代码，或者作为单个指令或作为短指令序列。
要在应用程序中使用内部函数，请包含头文件 intrinsics.h。
请注意，内部函数名称以双下划线开头，例如：
__disable_interrupt

本表概述了其内联函数:

内联函数	说明
__disable_interrupt	禁用中断
__enable_interrupt	启用中断
__get_interrupt_state	返回中断状态
__no_operation	插入一条 NOP 指令
__parity	表示参数的奇偶性
__set_interrupt_state	恢复中断状态
__tbac	原子读、修改、写指令

表 41: 内联函数汇总

内联函数的描述

本节提供了有关每个内联函数的参考信息。

__disable_interrupt

语法 void __disable_interrupt(void);
说明 通过清除中断启用 (IE) 寄存器中的第 7 位来禁用中断

__enable_interrupt
语法 void __enable_interrupt(void);
说明 通过设置中断启用 (IE) 寄存器中的第 7 位来启用中断。

__get_interrupt_state
语法 __istate_t __get_interrupt_state(void);
说明 返回全局中断状态。 返回值可以作为参数
__set_interrupt_state 内部函数，它将恢复中断状态。

例子
```
#include "intrinsics.h"
void CriticalFn()
{
__istate_t s = __get_interrupt_state();
__disable_interrupt();
/* Do something here. */
__set_interrupt_state(s);
}
```
与使用此代码序列相比，使用此代码序列的优势 __disable_interrupt 和 __enable_interrupt 就是本例中的代码 在调用 __get_interrupt_state 之前不会启用任何禁用的中断。

__no_operation
语法 void __no_operation(void);
说明 插入 NOP 指令。

__parity
语法 char __parity(char);
说明 表示 char 参数的奇偶性； 也就是说，参数是否包含设置为 1 的偶数或奇数位。如果数字是偶数，则返回 0，如果数字是奇数，则返回 1。

__set_interrupt_state
语法 void __set_interrupt_state(__istate_t);
说明 将中断状态恢复为
__get_interrupt_state 函数先前返回的值。
有关 __istate_t 类型的信息，请参见 __get_interrupt_state，第 384 页。

__tbac
语法 bool __tbac(bool bitvar);
说明 使用此内联函数创建信号量或类似的互斥函数。
它采用单个位变量 bitvar 并使用 JBC 汇编器指令来执行原子读取、修改和写入指令（测试位和清除）。 该函数返回 bitvar 的原始值（0 或 1）并将 bitvar 重置为 0。
注意：要在 C 源代码中使用 bool 类型，请参见 Bool，第 326 页。

（空白页）

预处理

- 预处理概述
- 预定义预处理符号说明
- 各种预处理扩展的描述

预处理概述

用于 8051 的 IAR C/C++ 编译器的预处理遵循标准 C。编译器还为您提供这些与预处理相关的功能：

- 预定义的预处理符号

这些符号允许您检查编译时环境，例如编译的时间和日期。有关详细信息，请参阅预定义预处理符号的说明，第 388 页。

- 使用编译器选项定义的用户定义的预处理符号

除了使用 #define 指令定义您自己的预处理符号外，您还可以使用选项 -D，请参见 -D，第 294 页。

- 预处理扩展

有几个预处理扩展，例如许多 pragma 指令；有关详细信息，请参阅 Pragma 指令一章。有关相应的 _Pragma 运算符和与预处理相关的其他扩展的信息,请参阅第 393 页的杂项预处理扩展的描述。

- 预处理输出

使用选项 --preprocess 将预处理输出定向到命名文件,请参见 --preprocess,第 317 页。
要指定包含文件的路径，请使用正斜杠：

```
#include "mydirectory/myfile"
```

在源代码中，使用正斜杠:

```
file = fopen("mydirectory/myfile","rt");
```

注意，也可以使用反斜杠。在这种情况下，在 include 文件路径中使用一个，在源代码字符串中使用两个 在源代码字符串中使用。

预定义预处理程序符号的描述

本节列出并描述了预处理程序符号。
注意：要列出预定义的预处理程序符号，请使用编译器选项
--predef_macros. 见-predef_macros，第 317 页。

预定义预处理器符号的描述
本节列出并描述了预处理器符号。
注意：要列出预定义的预处理器符号，请使用编译器选项 --predef_macros。参见
--predef_macros，第 317 页。

__BASE_FILE__
描述：

一个字符串，它标识正在编译的基本源文件（即，不是头文件）的名称。

另见：
__FILE__，第 390 页和 --no_path_in_file_macros，第 310 页。

__BUILD_NUMBER__
描述：
一个唯一的整数，用于标识当前使用的编译器的内部版本号。

__CALLING_CONVENTION__
描述：
一个整数，标识正在使用的调用约定。该符号反映了 --calling_convention 选项，定义为 0 表示数据覆盖，1 表示 idata 覆盖，2 表示 idata 可重入，3 表示 pdata 可重入，4 表示 xdata 可重入，5 表示扩展堆栈可重入。在测试 __CALLING_CONVENTION__ 符号时可以使用这些符号名称：__CC_DO__、__CC_IO__、__CC_IR__、__CC_PR__、__CC_XR__ 或 __CC_ER__。

__CODE_MODEL__
描述：

一个整数，用于标识正在使用的代码模型。该值反映了 --code_model 选项的设置，定义为 1 表示 Near，2 表示 分页，3 表示 Far，4 表示 分页扩展 2 代码模型。在测试 __CODE_MODEL__ 符号时可以使用这些符号名称：__CM_NEAR__、__CM_BANKED__、__CM_FAR__ 或 __CM_BANKED_EXT2__。

__CONSTANT_LOCATION__

描述：

一个整数，用于标识常量和字符串的默认位置。该符号反映了 --place_constants 选项的设置，定义为 0 表示数据，1 表示数据 ROM，2 表示代码。

__CORE__

描述：

一个整数，标识正在使用的芯片内核。该值反映了 --core 选项的设置，定义为 1 表示普通内核，2 表示 Extended1，3 表示 Extended2 内核。测试 __CORE__ 符号时可以使用这些符号名称：__CORE_PLAIN__、__CORE_EXTENDED1__ 或 __CORE_EXTENDED2__..

__COUNTER__

描述：

每次扩展时扩展为新整数的宏，从零 (0) 开始并向上计数。

__cplusplus

描述：

编译器在任何 C++ 模式下运行时定义的整数，否则未定义。定义后，其值为 199711L。此符号可与#ifdef 一起使用，以检测编译器是否接受 C++ 代码。在创建要由 C 和 C++ 代码共享的头文件时，它特别有用。标准 C 要求使用此符号。

__DATA_MODEL__

描述：

一个整数，用于标识正在使用的数据模型。该值反映了 --data_model 选项的设置，定义为 0 代表 Tiny，1 代表 Small，2 代表 Large，3 代表 Generic，4 代表 Far，5 代表 Far Generic 数据模型。测试 __DATA_MODEL__ 符号时可以使用这些符号名称：__DM_TINY__、__DM_SMALL__、__DM_LARGE__、__DM_GENERIC__、__DM_FAR__ 或 __DM_FAR_GENERIC__。

__DATE__

描述：
一个标识编译日期的字符串，以“Mmm dd yyyy”的形式返回，例如“Oct 30 2014” 这个符号是标准 C 要求的。

__embedded_cplusplus
描述：

当编译器在任何 C++ 模式下运行时定义为 1 的整数，否则符号未定义。此符号可与 #ifdef 一起使用，以检测编译器是否接受 C++ 代码。在创建要由 C 和 C++ 代码共享的头文件时，它特别有用。标准 C 要求使用此符号。

__EXTENDED_DPTR__
描述：

使用 24 位数据指针时设置为 1 的整数。否则，当使用 16 位数据指针时，符号未定义。

__EXTENDED_STACK__
描述：

使用扩展堆栈时设置为 1 的整数。否则，当不使用扩展堆栈时，符号未定义。

__FILE__
描述：
一个字符串，它标识正在编译的文件的名称，它可以是基本源文件和任何包含的头文件。标准 C 要求使用此符号。
另见：
__BASE_FILE__，第 388 页和 --no_path_in_file_macros，第 310 页。

__func__
描述：

一个预定义的字符串标识符，它使用使用符号的函数的名称进行初始化。这对于断言和其他跟踪实用程序很有用。该符号要求启用语言扩展。标准 C 要求使用此符号。

另见：
-e，第 302 页和 __PRETTY_FUNCTION__，第 391 页。

__FUNCTION__
描述：

一个预定义的字符串标识符，它使用使用符号的函数的名称进行初始化。这对于断言和其他跟踪实用程序很有用。该符号要求启用语言扩展。

另见：
-e，第 302 页和 __PRETTY_FUNCTION__，第 391 页。

__IAR_SYSTEMS_ICC__
描述：

一个标识 IAR 编译器平台的整数。当前值为 8。请注意，该数字在产品的未来版本中可能会更高。可以使用 #ifdef 测试此符号，以检测代码是否由 IAR Systems 的编译器编译。

__ICC8051__
描述：

使用 IAR C/C++ Compiler for 8051 编译代码时设置为 1 的整数。

__INC_DPSEL_SELECT__
描述：
当使用 INC 方法选择活动数据指针时设置为 1 的整数。否则，当使用 XOR 方法时，符号未定义。

__LINE__
描述：

一个整数，标识正在编译的文件的当前源代码行号，它可以是基本源文件和任何包含的头文件。标准 C 需要此符号。

__NUMBER_OF_DPTRS__
描述：

一个整数，标识正在使用的数据指针的数量； 1 到 8 之间的值。

__PRETTY_FUNCTION__
描述：

一个预定义的字符串标识符，用函数名称初始化，包括参数类型和返回类型，在其中使用符号的函数，例如“void func(char)”。此符号对于断言和其他跟踪实用程序很有用。该符号要求启用语言扩展。

另见：

-e，第 302 页和__func__，第 390 页。

__STDC__
描述：

设置为 1 的整数，表示编译器遵循标准 C。可以使用 #ifdef 测试此符号，以检测正在使用的编译器是否遵循标准 C。* 标准 C 需要此符号。

__STDC_VERSION__
描述：

一个整数，用于标识正在使用的 C 标准的版本。符号扩展为 199901L，除非使用 --c89 编译器选项,在这种情况下符号扩展为 199409L。此符号不适用于 EC++ 模式。标准 C 要求使用此符号。

__SUBVERSION__
描述：

一个整数，用于标识编译器版本号的 subversion 号，例如 1.2.3.4 中的 3。

__TIME__
描述：

以“hh:mm:ss”形式标识编译时间的字符串。标准 C 要求使用此符号。

__TIMESTAMP__
描述：

一个字符串常量,标识当前源文件最后一次修改的日期和时间。字符串的格式与 asctime 标准函数使用的格式相同（换句话说，“Tue Sep 16 13:03:52 2014”）。

__VER__
描述：

一个整数，用于标识正在使用的 IAR 编译器的版本号。数字的值是这样计算的：（100＊主要版本号 + 次要版本号）。例如，对于编译器版本 3.34，3 是主要版本号，34 是次要版本号。因此，__VER__ 的值为 334。

__XOR_DPSEL_SELECT__
描述：

当使用 XOR 方法选择活动数据指针时设置为 1 的整数。否则，当使用 INC 方法时，符号未定义。

其他预处理器扩展的描述 本节提供有关预处理器扩展的参考信息，除了预定义的符号、pragma 指令和标准 C 指令之外，这些可用的预处理器扩展。

NDBUG

描述：

该预处理器符号决定您在应用程序中编写的任何断言宏是否应包含在构建的应用程序中。

如果未定义此符号，则评估所有断言宏。如果定义了符号，则从编译中排除所有断言宏。换句话说，如果符号是：

- 已定义，不包含断言代码
- 未定义，将包含断言代码

这意味着，如果您编写任何断言代码并构建应用程序，您应该定义此符号以从最终应用程序中排除断言代码。

请注意，assert 宏是在 assert.h 标准包含文件中定义的。

在 IDE 中，如果您在 Release 构建配置中构建应用程序，则会自动定义 NDEBUG 符号。

另见：_ReportAssert，第 174 页。

＃警告信息

语法 #warning message 其中 message 可以是任何字符串。

说明 使用此预处理器指令生成消息。通常，这对于断言和其他跟踪实用程序很有用，类似于使用标准 C #error 指令的方式。使用 --strict 编译器选项时无法识别此指令。

（空白页）
页码:394

C/C++ 标准库函数

- C/C++标准库概述
- DLIB 运行环境--实现细节
- CLIB 运行环境--实现细节
- 8051 特定的 CLIB 函数

有关库函数的详细参考信息，请参见在线帮助系统。

C/C++ 标准库概述

编译器带有 C/C++ 标准库的两种不同实现：

IAR DLIB 运行时环境是 C/C++ 标准库的完整实现，符合标准 C 和 C++。该库还支持 IEEE 754 格式的浮点数，并且可以配置为包括对区域设置、文件描述符、多字节字符等的不同级别的支持。

IAR CLIB Runtime Environment 是 C 标准库的轻量级实现，它不完全符合标准 C。它也不完全支持 IEEE 754 格式的浮点数，也不支持 C++。

有关自定义的更多信息，请分别参见 DLIB 运行时环境和 CLIB 运行时环境一章。有关库函数的详细信息，请参阅产品随附的在线文档。还有 DLIB 的关键字参考信息库函数。要获取函数的参考信息，请在编辑器窗口中选择函数名称并按 F1。有关库函数的更多信息，请参阅本指南中标准 C 的实现定义的行为一章。

头文件

您的应用程序通过头文件获得对库定义的访问权，它使用#include 指令将其合并。这些定义被分成几个不同的头文件，每个头文件都覆盖一个特定的函数区域，让您只包括那些需要的。

在引用其定义之前，必须包含适当的头文件。不这样做可能会导致调用在执行期间失败，或者在编译时或链接时生成错误或警告消息。

库对象文件

大多数库定义无需修改即可使用，即直接从产品随附的库对象文件中使用。有关如何选择运行时库的信息，请参阅基本项目配置，第 56 页。链接器将仅包含应用程序直接或间接需要的那些例程。

有关如何使用自己的版本覆盖库模块的信息，另请参阅第 152 页的覆盖库模块。

更准确的库功能 cos、sin、tan 和 pow 的默认实现被设计为又快又小。

作为替代方案，有些版本旨在提供更高的准确性。对于函数的浮点变量，它们被命名为 __iar_xxx_accuratef，对于函数的长双精度变量，它们被命名为 __iar_xxx_accuratel，其中 xxx 是 cos、sin 等。

要使用这些更准确的版本，请使用 -e 链接器选项。

重入

可以在主应用程序和任意数量的中断中同时调用的函数是可重入的。使用静态分配数据的库函数是 DLIB 运行时环境的大多数部分是可重入的，但以下函数和部分不是可重入的，因为它们需要静态数据：

- 堆函数--malloc、free、realloc、calloc 等以及 C++ 运算符 new 和 delete
- 语言环境函数--localeconv、setlocale
- 多字节函数—mblen、mbrlen、mbrtowc、mbsrtowc、mbtowc、wcrtomb、wcsrtomb、wctomb
- 兰德函数--rand、srand
- 时间函数--asctime、localtime、gmtime、mktime
- 杂项函数 atexit、perror、strerror、strtok

页码:396

● 以某种方式使用文件或堆的函数。这包括 scanf、sscanf、getchar、getwchar、putchar 和 putwchar。此外,如果您使用选项 --enable_multibyte 和 --dlib_config=Full,printf 和 sprintf 函数（或任何变体）也可以使用堆。

对于 CLIB 库，qsort 函数和以某种方式使用文件的函数是不可重入的。这包括 printf、scanf、getchar 和 putchar。但是，函数 sprintf 和 sscanf 是可重入的。

可以设置 errno 的函数是不可重入的,因为这些函数之一产生的 errno 值可能会在读取之前被随后使用该函数破坏。这适用于数学和字符串转换函数等。

对此的补救措施是：

● 不要在中断服务程序中使用不可重入函数

● 通过互斥体或安全区域等保护对不可重入函数的调用。

LONGJMP 函数

longjmp 实际上是跳转到先前定义的 setjmp。在堆栈展开期间驻留在堆栈上的任何可变长度数组或 C++ 对象都不会被销毁。这可能导致资源泄漏或不正确的应用程序行为。

DLIB 运行时环境--实现细节

DLIB 运行时环境提供了适用于嵌入式系统的大部分重要的 C 和 C++ 标准库定义。这些是以下类型：

● 遵守标准 C 的独立实现。该库支持大多数托管功能，但您必须实现它的一些基本功能。有关更多信息，请参阅本指南中标准 C 的实现定义的行为一章。

● 标准 C 库定义，用于用户程序。

● C++ 库定义，用于用户程序。

● CSTARTUP，包含启动代码的模块，参见本指南中的 DLIB 运行环境章节。

● 运行时支持库；例如低级浮点例程。

● 内在函数，允许底层使用 8051 特性。有关详细信息，请参阅内部函数一章。

此外，DLIB 运行时环境包括一些添加的 C 功能，请参阅添加的 C 功能，第 401 页。

页码:397

C 头文件 本节列出了特定于 DLIB 运行时环境的 C 头文件。 头文件可能还包含特定于目标的定义； 这些都记录在使用 C 的章节中。 下表列出了 C 头文件：

头文件	用法
assert.h	在函数执行时强制断言
complex.h	计算常见的复杂数学函数
ctype.h	对字符进行分类
errno.h	测试库函数报告的错误代码
fenv.h	浮点异常标志
float.h	测试浮点类型属性
inttypes.h	为 stdint.h 中定义的所有类型定义格式化程序
iso646.h	使用修正 1—iso646.h 标准头文件
limits.h	测试整数类型属性
locale.h	适应不同的文化习俗
math.h	计算常用数学函数
setjmp.h	执行非本地 goto 语句
signal.h	控制各种异常情况
stdarg.h	访问不同数量的参数
stdbool.h	在 C 中添加对 bool 数据类型的支持。
stddef.h	定义几个有用的类型和宏
stdint.h	提供整数特征
stdio.h	执行输入和输出
stdlib.h	执行各种操作
string.h	操作几种字符串
tgmath.h	类型通用数学函数
time.h	在各种时间和日期格式之间转换
uchar.h	Unicode 功能（标准 C 的 IAR 扩展）
wchar.h	支持宽字符
wctype.h	对宽字符进行分类

表 42： 传统标准 C 头文-DLIB

页码:398

C++头文件

本节列出了 C++头文件：

● C++库头文件

构成嵌入式 C++库的头文件。

● C++标准模板库 （STL） 头文件

构成扩展嵌入式 C++库的 STL 的头文件。

● C++ C 头文件

C++头文件，这些头文件提供 C 库中的资源。

C++库头文件

下表列出了可在嵌入式 C++中使用的头文件：

头文件	用法
complex	定义一个支持复杂算术的类
fstream	定义几个操作外部文件的 I/O 流类
iomanip	声明几个带参数的 I/O 流操纵器
ios	定义作为许多 I/O 流类的基础的类
iosfwd	在必须定义之前声明几个 I/O 流类
iostream	声明操作标准流的 I/O 流对象
istream	定义执行提取的类
new	声明几个分配和释放存储的函数
ostream	定义执行插入的类
sstream	定义几个操作字符串容器的 I/O 流类
streambuf	定义缓冲 I/O 流操作的类
string	定义一个实现字符串容器的类
strstream	定义几个操作内存中字符序列的 I/O 流类

表 43：C++头文件

C++ 标准模板库 (STL) 头文件 下表列出了可在扩展嵌入式 C++ 中使用的标准模板库 (STL) 头文件：

头文件	说明
algorithm	定义了对序列的几种常见操作
deque	一个双端队列序列容器
functional	定义了几个函数对象
hash_map	基于哈希算法的地图关联容器
hash_set	基于散列算法的集合关联容器
iterator	定义通用迭代器，以及对迭代器的操作
list	一个双向链表序列容器
map	地图关联容器
memory	定义管理内存的设施
numeric	对序列执行广义数值运算
queue	队列序列容器
set	一组关联容器
slist	单链表序列容器
stack	堆栈序列容器
utility	定义了几个实用组件
vector	一个向量序列容器

表 44：标准模板库头文件

在 C++ 中使用标准 C 库

C++ 库与标准 C 库中的一些头文件一起工作，有时会进行一些小改动。 头文件有两种形式--新的和传统的--例如，cassert 和 assert.h。

下表显示了新的头文件：

头文件	用法
cassert	在函数执行时强制断言
cctype	对字符进行分类
cerrno	测试库函数报告的错误代码
cfloat	测试浮点类型属性
cinttypes	为 stdint.h 中定义的所有类型定义格式化程序
climits	测试整数类型属性
clocale	适应不同的文化习俗
cmath	计算常用数学函数
csetjmp	执行非本地 goto 语句

页码:400

csignal	控制各种异常情况
cstdarg	访问不同数量的参数
cstdbool	在 C 中添加对 booldata 类型的支持。
cstddef	定义几个有用的类型和宏
cstdint	提供整数特征
cstdio	执行输入和输出
cstdlib	执行各种操作
cstring	操作几种字符串
ctime	在各种时间和日期格式之间转换
cwchar	支持宽字符
cwctype	对宽字符进行分类

表 45：新的标准 C 头文件 DLIB

作为内在函数的库函数

某些 C 库函数在某些情况下将作为内部函数处理，并将生成内联代码而不是普通函数调用，例如 memcpy、memset 和 strcat。

添加了 C 函数

DLIB 运行时环境包括一些附加的 C 功能。

以下包含文件提供了这些函数：

- fenv.h
- stdio.h
- stdlib.h
- string.h

fenv.h

在 fenv.h 中，浮点数的陷阱处理支持使用函数 fegettrapenable 和 fegettrapdisable 定义。

stdio.h

这些函数提供额外的 I/O 功能：

fdopen	根据低级文件描述符打开文件。
fileno	从文件描述符 (FILE*) 中获取低级文件描述符。
__gets	对应于 fgetson 标准输入。
getw Gets	来自标准输入的 wchar_tcharacter。
putw Puts	一个 wchar_tcharacter 到标准输出。
__ungetchar	对应于 ungetcon 标准输出。
__write_array	对应于 fwriteon 标准输出。

string.h

这些是 string.h	中定义的附加函数：
strdup	在堆上复制一个字符串。
strcasecmp	比较字符串不区分大小写。
strncasecmp	比较字符串不区分大小写和有界。
strnlen	有界字符串长度。

库内部使用的符号

库使用以下符号，这意味着它们在库源文件等中可见：

__assignment_by_bitwise_copy_allowed

此符号确定类对象的属性。

__constrange()

确定内部函数的参数的允许范围，并且该参数的类型必须是 const:

__construction_by_bitwise_copy_allowed

这个符号决定了类对象的属性。

__has_constructor, __has_destructor

这些符号确定类对象的属性，它们的功能类似于 sizeof 运算符。

当类、基类或成员（递归地）分别具有用户定义的构造函数或析构函数时，这些符号为真。

__memory_of
确定类内存。类内存确定类对象可以驻留在哪个内存中。此符号只能作为类内存出现在类定义中。

注：符号为保留符号，只能由库使用。

使用编译器选项 --predef_macros 来确定任何预定义符号的值。

CLIB 运行时环境--实现细节

CLIB 运行时环境提供了适用于嵌入式系统的大部分重要的 C 标准库定义。 这些是以下类型：

- 可用于用户程序的标准 C 库定义。 这些都记录在本章中。
- 系统启动代码； 请参阅本指南中的 CLIB 运行时环境一章。
- 运行时支持库； 例如低级浮点例程。
- 内在函数，允许底层使用 8051 特性。 见章节

内在函数了解更多信息。

库定义摘要

下表列出了特定于 CLIB 运行时环境的 C 头文件：

头文件	说明
assert.h	断言
ctype.h*	字符处理
errno.h	错误返回值
float.h	浮点类型的限制和大小
iccbutl.h	低级例程
limits.h	整数类型的限制和大小
math.h	数学
setjmp.h	非局部跳跃
stdarg.h	可变参数
stdbool.h	在 C 中添加对 booldata 类型的支持
stddef.h	常用定义包括 size_t、NULL、ptrdiff_t 和 offsetof
stdio.h	输入输出
stdlib.h	通用功能
string.h	字符串处理

表 46：CLIB 运行时环境头文件

*在头文件 ctype.h 中声明的函数是 xxxx、输出器和命令。多次计算它们的参数。
这并不符合 ISO/ANSI 的标准。

页码:403

8051 特定的 CLIB 函数

本节列出了在 pgmspace.h 中声明的特定于 8051 的 CLIB 库函数，它们允许访问代码内存中的字符串。

指定读取和写格式化程序

通过编辑链接器配置文件，可以覆盖 printf_Pandscanf_P 的默认格式化程序。请注意，不可能使用 IDE 来覆盖特定于 8051 的库例程的默认格式化程序。

要覆盖默认的 printf_Pformatter，请在链接器命令文件中键入以下任意行：

-e_small_write_P=_formatted_write_P

-e_medium_write_P=_formatted_write_P

若要重写默认 scanf_Pformatter，请在链接器命令文件中键入以下行：

-e_medium_read_P=_formatted_read_P

注意:在下面的描述中，PGM_VOID_P 是一个符号，根据数据模式扩展为 void const __code * 或 void const __far_code *；而 PGM_P 是一个符号，根据数据模式扩展为 char const __code *或 char const __far_code *。

段参考

- 细分市场摘要
- 段的描述

有关段放置的更多信息，请参阅链接应用程序一章。

概述

编译器将代码和数据放入 IAR XLINK 链接器引用的命名段中。编写汇编语言模块时需要有关这些段的详细信息，在解释编译器的汇编语言输出时也很有用。

下表列出了编译器中可用的段：

段	描述
BANKED_CODE	保存声明为 __banked_func 的代码。
BANKED_CODE_EXT2_AC	当使用 Banked extended2 代码模型时，保存定位的常量数据。
BANKED_CODE_EXT2_AN	使用 Banked extended2 代码模型时，保存已定位的未初始化数据。
BANKED_CODE_EXT2_C	使用 Banked extended2 代码模型时，保存常量数据。
BANKED_CODE_EXT2_N	在使用 Banked extended2 代码模型时，保存 __no_initstatic 和全局变量。
BANKED_CODE_INTERRUPTS_EXT2	为扩展 2 内核编译时持有 __interruptfunctions。
BANKED_EXT2	为扩展 2 内核编译时保持跳板功能。
BANK_RELAYS	为 Banked 代码模型编译时，保留用于库切换的中继能。
BDATA_AN	保存 __bdatalocated 未初始化的数据。
BDATA_I	保存 __bdatastatic 和全局初始化变量。
BDATA_ID	在 BDATA_I 中保存 __bdatastatic 和全局变量的初始值。
BDATA_N	保存 __no_init __bdatastatic 和全局变量

页码:405

BDATA_Z	保存零初始化的 __bdatastatic 和全局变量。
BIT_N	保存 __no_init __bitstatic 和全局变量。
BREG	保存编译器的虚拟位寄存器。
CHECKSUM	保存链接器生成的校验和。
CODE_AC	保存 __codelocated 常量数据。
CODE_C	保存 __codeconstant 数据。
CODE_N	保存 __no_init __codestatic 和全局变量。
CSTART	保存启动代码。
DATA_AN	保存 __datalocated 未初始化的数据。
DATA_I	保存 __datastatic 和全局初始化变量。
DATA_ID	在 DATA_I 中保存 __datastatic 和全局变量的初始值。
DATA_N	保存 __no_init __datastatic 和全局变量。
DATA_Z	保存零初始化的 __datastatic 和全局变量。
DIFUNCT	保存指向代码的指针，通常是 C++ 构造函数，应该在调用 main 之前由系统启动代码执行。
DOVERLAY	保存静态数据覆盖区域。
EXT_STACK	存放Maxim (Dallas Semiconductor) 390/400扩展数据堆栈。
FAR_AN	保存 __farlocated 未初始化的数据。
FAR_CODE	保存声明为 __far_func 的代码。
FAR_CODE_AC	保存 __far_codelocated 常量数据。
FAR_CODE_C	保存 __far_codeconstant 数据。
FAR_CODE_N	保存 __no_init __far_codestatic 和全局变量。
FAR_HEAP	在远内存中保存用于动态分配数据的堆。
FAR_I	保存 __farstatic 和全局初始化变量。
FAR_ID	保存 FAR_I 中 __farstatic 和全局变量的初始值。
FAR_N	保存 __no_init __farstatic 和全局变量。
FAR_ROM_AC	保存 __far_romlocated 常量数据。
FAR_ROM_C	保存 __far_romconstant 数据。
FAR_Z	保存零初始化的 __farstatic 和全局变量。
FAR22_AN	保存 __far22located 未初始化的数据。
FAR22_CODE	保存声明为 __far22_code 的代码。
FAR22_CODE_AC	保存 __far22_codelocated 常量数据。

FAR22_CODE_C	保存 __far22_codeconstant 数据。
FAR22_CODE_N	保存 __no_init __far22_codestatic 和全局变量。
FAR22_HEAP	在 far22 内存中保存用于动态分配数据的堆。
FAR22_I	保存 __far22static 和全局初始化变量。
FAR22_ID	保存 FAR22_I 中 __far22static 和全局变量的初始值。
FAR22_N	保存 __no_init __far22static 和全局变量。
FAR22_ROM_AC	保存 __far22_romlocated 常量数据。
FAR22_ROM_C	保存 __far22_romconstant 数据。
FAR22_Z	保存零初始化的 __far22static 和全局变量。
HUGE_AN	保存 __hugelocated 未初始化的数据。
HUGE_CODE_AC	保存 __huge_codelocated 常量数据。
HUGE_CODE_C	保存 __huge_codeconstant 数据。
HUGE_CODE_N	保存 __no_init __huge_codestatic 和全局变量。
HUGE_HEAP	在大内存中保存用于动态分配数据的堆。
HUGE_I	保存 __hugestatic 和全局初始化变量。
HUGE_ID	在 HUGE_I 中保存 __hugestatic 和全局变量的初始值。
HUGE_N	保存 __no_init __hugestatic 和全局变量。
HUGE_ROM_AC	保存 __huge_romlocated 常量数据。
HUGE_ROM_C	保存 __huge_romconstant 数据。
HUGE_Z	保存零初始化的 __hugestatic 和全局变量。
IDATA_AN	保存 __idatalocated 未初始化的数据。
IDATA_I	保存 __idatastatic 和全局初始化变量。
IDATA_ID	在 IDATA_I 中保存 __idatastatic 和全局变量的初始值。
IDATA_N	保存 __no_init __idatastatic 和全局变量。
IDATA_Z	保存零初始化的 __idatastatic 和全局变量。
INTVEC	包含复位和中断向量。
INTVEC_EXT2	包含内核扩展 2 时的复位和中断向量。
IOVERLAY	保存静态 idata 覆盖区域。
ISTACK	保存内部数据堆栈。
IXDATA_AN	保存 __ixdatalocated 未初始化的数据。
IXDATA_I	保存 __ixdatastatic 和全局初始化变量。

IXDATA_ID	在 IXDATA_I 中保存 __ixdatastatic 和全局变量的初始值。
IXDATA_N	保存 __no_init __ixdatastatic 和全局变量。
IXDATA_Z	保存零初始化的 __ixdatastatic 和全局变量。
NEAR_CODE	保存声明为 __near_func 的代码。
PDATA_AN	保存 __pdatalocated 未初始化的数据。
PDATA_I	保存 __pdatastatic 和全局初始化变量。
PDATA_ID	在 PDATA_I 中保存 __pdatastatic 和全局变量的初始值。
PDATA_N	保存 __no_init __pdatastatic 和全局变量。
PDATA_Z	保存零初始化的 __pdatastatic 和全局变量。
PSP	保存指向 pdata 堆栈的堆栈指针。
PSTACK	保存 pdata 堆栈。
RCODE	保存声明为 __near_func 的代码。
SFR_AN	保存 __sfrlocated 未初始化的数据。
VREG	包含编译器的虚拟寄存器区域。
XDATA_AN	保存 __xdatalocated 未初始化的数据。
XDATA_HEAP	持有用于动态分配数据的堆。
XDATA_I	保存 __xdatastatic 和全局初始化变量。
XDATA_ID	保存 XDATA_I 中 __xdatastatic 和全局变量的初始值。
XDATA_N	保存 __no_init __xdatastatic 和全局变量。
XDATA_ROM_AC	保存 __xdata_romlocated 常量数据。
XDATA_ROM_C	保存 __xdata_romconstant 数据。
XDATA_Z	保存零初始化的 __xdatastatic 和全局变量。
XSP	保存指向 xdata 堆栈的堆栈指针。
XSTACK	保存 xdata 堆栈。

表 47：段摘要

段的描述

本节提供有关每个段的参考信息。

这些段由段放置链接器指令 -Z 和 -P 放置在内存中，分别用于顺序放置和打包放置。 有些段不能使用打包放置，因为它们的内容必须是连续的。

有关这些指令的信息，请分别参见使用 -Z 命令进行顺序放置，第 129 页和使用 -P 命令进行打包放置，第 130 页。
对于每个段，都指定了段内存类型，这表明段应该放置在哪种类型的内存中； 请参见段内存类型，第 120 页。
有关如何在链接器配置文件中定义段的信息，请参阅链接您的应用程序，第 127 页。
有关此处提到的扩展关键字的更多信息，请参阅扩展关键字一章。

BANKED_CODE

描述：
保存声明为 __banked_func 的程序代码，这是 Banked 代码模型中的默认值。
段存储器类型：
CODE
存储器地址：
该段可以放置在代码存储空间中的任何位置。
访问类型：
只读

BANKED_CODE_EXT2_AC

描述：
当使用 Banked extended2 代码模型时，保存定位的常量数据。如果指定了编译器选项 --place_constants=code，则该段还保存默认声明的初始化定位的 const 对象。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

BANKED_CODE_EXT2_AN

描述：
当使用 Banked extended2 代码模型时，保存 __no_init 定位的数据。除非已指定选项 --place_constants=code 或 --place_constants=data_rom，否则该段还保存已定位的非初始化对象，声明为 __data const。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

BANKED_CODE_EXT2_C

描述：

使用 Banked extended2 代码模型时，保存常量数据。这可以包括常量变量、字符串和聚合文字等。如果指定了编译器选项 --place_constants=code，则该段还保存默认声明的非定位常量数据和字符串。
段存储器类型：
CODE
存储器地址：
0F0000-0FFFFF
访问类型：
读写
另见：
--output, -o, 第 315 页.

BANKED_CODE_EXT2_N

描述：
使用 Banked extended2 代码模型时，保存静态和全局 __no_init 变量。
如果指定了编译器选项 --place_constants=code,则该段还保存默认声明的非定位常量数据和字符串。
段存储器类型：
CODE
存储器地址：
0F0000-0FFFFF
访问类型：
读写
另见：
--output, -o, 第 315 页.

BANKED_CODE_INTERRUPTS_EXT2

描述：
为扩展 2 内核编译时持有 __interrupt 函数。
段存储器类型：
CODE
存储器地址：
该段可以放置在代码存储空间中的任何位置,但必须与段 INTVEC_EXT2 位于同一存储区中。
访问类型：
读写

BANKED_EXT2

描述：

保存跳板函数，即在为扩展 2 内核编译时需要复制到每个库的函数。

段存储器类型：
CODE
存储器地址：
0xFFF0-0xFFFF in each 64-Kbyte block.
访问类型：
只读

BANK_RELAYS

描述：

在为扩展 2 内核和 Banked 扩展 2 代码模型编译时，保存用于库切换的中继函数。

段存储器类型：
CODE
存储器地址：
该段可以放置在 0x0-0xFFFF 内的任何位置，但必须位于根库中。
访问类型：
只读

BDATA_AN

描述：

保存 __no_init __bdata 定位的数据。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

BDATA_I

描述：

保存 __bdata 静态和全局初始化变量,通过在应用程序启动时从段 BDATA_ID 复制初始化。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
DATA
存储器地址：
0x20-0x2F
访问类型：
读写

BDATA_ID

描述：
在 BDATA_I 段中保存 __bdata 静态和全局变量的初始值。这些值在应用程序启动时从 BDATA_ID 复制到 BDATA_I。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。
段存储器类型：
CODE
存储器地址：
该段可以放置在代码存储空间中的任何位置。
访问类型：
只读

BDATA_N

描述：
保存静态和全局 __no_init __bdata 变量。
段存储器类型：
DATA
存储器地址：
0x20-0x2F
访问类型：
只读

BDATA_Z

描述：

保存零初始化的 __bdata 静态和全局变量。该段的内容由系统启动代码声明。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
DATA
存储器地址：
0x20-0x2F
访问类型：
读写

BIT_N

描述：
保存静态和全局 __no_init __bit 变量。
段存储器类型：
BIT
存储器地址：
0x00-0x7F
访问类型：
只读

BREG

描述：
保存编译器的虚拟位寄存器。
段存储器类型：
BIT
存储器地址：
0x00-0x7F
访问类型：
读写

CHECKSUM

描述：
保存链接器生成的校验和字节。该段还包含 __checksum 符号。请注意，此段的大小受链接器选项 -J 的影响。
段存储器类型：
CODE
存储器地址：
该段可以放置在 ROM 存储器中的任何位置。
访问类型：
只读

CODE_AC

描述：

保存 __code 定位的常量数据。如果指定了编译器选项 --place_constants=code，则该段还保存 const 对象。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

CODE_C

描述：

保存 __code 常量数据。这可以包括常量变量、字符串和聚合文字等。如果指定了编译器选项 --place_constants=code，则该段还保存常量数据和字符串。

段存储器类型：
CODE
存储器地址：
0-0xFFFF
访问类型：
只读
另见：
--output, -o, 第 315 页.

CODE_N

描述：
保存静态和全局 __no_init __code 变量。如果指定了编译器选项
--place_constants=code，则该段还保存常量数据和字符串。
段存储器类型：
CODE
存储器地址：
0-0xFFFF
访问类型：
只读
另见：
--output, -o, 第 315 页.

CSTART

描述：
保存启动代码。

该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
CODE
存储器地址：
该段必须放置在复位后微控制器开始执行的地址，对于 8051 微控制器，该地址位于地址 0x0。
访问类型：
只读

DATA_AN

描述：
保存 __no_init __data 定位的数据。除非已指定选项 --place_constants=code 或 --place_constants=data_rom，否则该段还包含声明为 __data const 的已定位非初始化对象，并且在 Tiny 数据模型中，默认声明已定位非初始化常量对象。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

DATA_I

描述：

保存 __data 静态和全局初始化变量，通过在应用程序启动时从段 DATA_ID 复制初始化。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
DATA
存储器地址：
0x0-0x7F
访问类型：
读写

DATA_ID

描述：

在 DATA_I 段中保存 __data 静态和全局变量的初始值。这些值在应用程序启动时从 DATA_ID 复制到 DATA_I。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
CODE
存储器地址：
该段可以放置在内存中的任何位置。
访问类型：
只读

DATA_N

描述：

保存静态和全局 __no_init __data 变量。
段存储器类型：
DATA
存储器地址：
0x0-0x7F
访问类型：
读写

DATA_Z

描述：

保存零初始化的 __data 静态和全局变量。该段的内容由系统启动代码声明。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
DATA
存储器地址：
0x0-0x7F
访问类型：
读写

DIFUNCT

描述：
保存 C++ 使用的动态初始化向量。
段存储器类型：
CONST
存储器地址：
在小数据模型中，该段必须放在内存的前 64 KB。在其他数据模型中，该段可以放置在内存中的任何位置。
访问类型：
只读

DOVERLAY

描述：
保存使用数据覆盖调用约定调用的函数的静态覆盖区域。
段存储器类型：
DATA
存储器地址：
0x0-0x7F

访问类型：
读写

EXT_STACK

描述：
保存扩展数据堆栈。
段存储器类型：
XDATA
存储器地址：
该段必须放置在外部存储空间中。
访问类型：
读写
另见：
有关如何在链接器配置文件中定义此段及其长度的信息，请参见设置堆栈内存，第 132 页。

FAR_AN

描述：
保存 __no_init __far 数据。除非已指定选项 --place_constants=code 或 --place_constants=data_rom，否则该段还保存声明为 __far const 的已定位非初始化对象，并且在 Far 数据模型中，默认声明位于未初始化的常量对象。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

FAR_CODE

描述：
保存声明为 __far_func 的应用程序代码。
段存储器类型：
CODE
存储器地址：
该段必须放在代码存储空间中。
访问类型：
只读

FAR_CODE_AC

描述：

保存 __far_code 定位的常量数据。在 Far 数据模型中，如果指定了编译器选项 --place_constants=code，则该段还保存默认声明的初始化定位的 const 对象。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

FAR_CODE_C

描述：
保存 __far_code 常量数据。这可以包括常量变量、字符串和聚合文字等。在 Far 数据模型中，如果指定了编译器选项 --place_constants=code，则该段还保存默认声明的非初始化常量数据。
段存储器类型：
CODE
存储器地址：
该段可以放置在代码存储空间中的任何位置。
访问类型：
只读
另见：
--output, -o, 第 315 页.

FAR_CODE_N

描述：

保存静态和全局 __no_init __far_code 变量。在 Far 数据模型中，如果指定了编译器选项 --place_constants=code，则该段还保存默认声明的非初始化常量数据。

段存储器类型：
CODE
存储器地址：
该段可以放置在代码存储空间中的任何位置。
访问类型：
只读
另见：
--output, -o, 第 315 页.

FAR_HEAP

描述：
在远内存中保存用于动态分配的数据的堆，即由 far_malloc 和 far_free 分配的数据，在 C++ 中为 new 和 delete。
段存储器类型：
XDATA

存储器地址：
该段可以放置在外部数据存储器中的任何位置。
访问类型：
读写
另见：
有关如何在链接器配置文件中定义此段及其长度的信息以及有关在远内存中对堆使用 new 和 delete 运算符的信息，请参阅设置堆内存，第 135 页和新建和删除运算符，第 236 页。

FAR_I

描述：

保存 __far 静态和全局初始化变量，通过在应用程序启动时从段 FAR_ID 复制来初始化。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
XDATA
存储器地址：
该段必须放置在外部数据存储空间中。
访问类型：
只读

FAR_ID

描述：

保存 FAR_I 段中的 __far 静态变量和全局变量的初始值。这些值在应用程序启动时从 FAR_ID 复制到 FAR_I。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
CODE
存储器地址：
该段可以放置在代码存储空间中的任何位置。
访问类型：
只读

FAR_N

描述：

保存静态和全局 __no_init __far 变量。除非已指定选项 --place_constants=code 或 --place_constants=data_rom，否则该段还包含声明为 __far const 的非初始化对象，

以及在 Far 数据模型中，默认声明的非初始化常量对象。

段存储器类型：
CONST
存储器地址：
该段必须放置在外部数据存储空间中。
访问类型：
只读

FAR_ROM_AC

描述：
保存 __far_rom 定位的常量数据。在 Far 数据模型中，如果指定了编译器选项 --place_constants=data_rom，则该段还保存默认声明的初始化定位的 const 对象。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

FAR_ROM_C

描述：
保存 __far_rom 常量数据。这可以包括常量变量、字符串和聚合文字等。在 Far 数据模型中，如果指定了编译器选项 --place_constants=data_rom，则该段还包含默认声明的非定位常量数据和字符串。
段存储器类型：
CONST
存储器地址：
该段可以放置在外部数据存储空间中的任何位置。
访问类型：
只读
另见：
--output, -o, 第 315 页.

FAR_Z

描述：
保存零初始化的 __far 静态和全局变量。该段的内容由系统启动代码声明。除非已指定选项 --place_constants=code 或 --place_constants=data_rom，否则该段还包含未初始化或零初始化的 __far 常量。在 Far 数据模型中，除非已指定选项 --place_constants=code 或 --place_constants=data_rom，否则该段还包含默认声明的零初始化常量对象。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
XDATA
存储器地址：
该段必须放置在外部数据存储空间中。
访问类型：
读写
另见：
--output, -o, 第 315 页.

FAR22_AN

描述：

保存 __no_init __far22 定位的数据。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

FAR22_CODE

描述：
保存声明为 __far22_code 的应用程序代码。
段存储器类型：
CODE
存储器地址：
0x0-0x3FFFFF
访问类型：
只读

FAR22_CODE_AC

描述：

保存 __far22_code 定位的常量数据。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

FAR22_CODE_C

描述：
保存 __far22_code 常量数据。这可以包括常量变量、字符串和聚合文字等。
段存储器类型：
CODE
存储器地址：

0x0-0x3FFFFF
访问类型：
只读
另见：
--output, -o, 第 315 页.

FAR22_CODE_N

描述：
保存静态和全局 __no_init __far22_code 变量。
段存储器类型：
CODE
存储器地址：
0x0-0x3FFFFF
访问类型：
只读
另见：

FAR22_HEAP

描述：
在 far22 内存中保存用于动态分配数据的堆,即由 far22_malloc 和 far22_free 分配的数据，在 C++ 中为 new 和 delete。
段存储器类型：
XDATA
存储器地址：
0x0-0x3FFFFF
访问类型：
读写
另见：
有关如何在链接器配置文件中定义此段及其长度的信息以及有关在远内存中对堆使用 new 和 delete 运算符的信息，请参阅设置堆内存，第 135 页和新建和删除运算符，第 236 页。

FAR22_I

描述：

保存 __far22 静态和全局初始化变量,通过在应用程序启动时从段 FAR22_ID 复制初始化。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：

XDATA
存储器地址：
0x0-0x3FFFFF
访问类型：
只读

FAR22_ID

描述：
保存 FAR22_I 段中的 __far22 静态和全局变量的初始值。这些值在应用程序启动时从 FAR22_ID 复制到 FAR22_I。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。
段存储器类型：
CODE
存储器地址：
该段可以放置在内存中的任何位置。
访问类型：
只读

FAR22_N

描述：
保存静态和全局 __no_init __far22 变量。
段存储器类型：
CONST
存储器地址：
0x0-0x3FFFFF
访问类型：
只读

FAR22_ROM_AC

描述：

保存 __far22_rom 定位的常量数据。定位意味着使用 @ 运算符或 #pragmalocation 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

FAR22_ROM_C

描述：

保存 __far22_rom 常量数据。这可以包括常量变量、字符串和聚合文字等。
段存储器类型：
CONST
存储器地址：
0x0-0x3FFFFF
访问类型：
只读
另见：
--output, -o, 第 315 页.

FAR22_Z

描述：

保存零初始化的 __far22 静态和全局变量。该段的内容由系统启动代码声明。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
XDATA
存储器地址：
0x0-0x3FFFFF
访问类型：
读写
另见：
--output, -o, 第 315 页.

HUGE_AN

描述：
保存 __no_init __huge 定位数据。除非已指定选项 --place_constants=code 或 --place_constants=data_rom ，否则还保留声明为 __huge const 的已定位非初始化对象。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

HUGE_CODE_AC

描述：

保存 __huge_code 定位的常量数据。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

HUGE_CODE_C

描述：
保存 __huge_code 常量数据。这可以包括常量变量、字符串和聚合文字等。
段存储器类型：
CODE
存储器地址：
该段可以放置在代码存储空间中的任何位置。
访问类型：
只读

HUGE_CODE_N

描述：
保存静态和全局 __no_init __huge_code 变量。
段存储器类型：
CODE
存储器地址：
该段可以放置在代码存储空间中的任何位置。
访问类型：
只读

HUGE_HEAP

描述：
在大内存中保存用于动态分配数据的堆，即由 huge_malloc 和 huge_free 分配的数据，在 C++ 中为 new 和 delete。
段存储器类型：
XDATA
存储器地址：
该段可以放置在外部数据存储空间中的任何位置。
访问类型：
读写
另见：
有关如何在链接器配置文件中定义此段及其长度的信息，以及有关在大内存中对堆使用 new 和 delete 运算符的信息，请参阅设置堆内存，第 135 页和新建和删除 运算符，第 236 页。

HUGE_I

描述：

保存 __huge 静态和全局初始化变量，通过在应用程序启动时从段 HUGE_ID 复制初始化。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
XDATA
存储器地址：
该段可以放置在外部数据存储空间中的任何位置。
访问类型：
读写

HUGE_ID

描述：

在 HUGE_I 段中保存 __huge 静态和全局变量的初始值。这些值在应用程序启动时从 HUGE_ID 复制到 HUGE_I。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
CODE
存储器地址：
该段可以放置在代码存储空间中的任何位置。
访问类型：
只读

HUGE_N

描述：
保存静态和全局 __no_init __huge 变量。除非已指定选项 --place_constants=code 或 --place_constants=data_rom，否则该段还包含声明为 __huge const 的未初始化对象。
段存储器类型：
XDATA
存储器地址：
该段可以放置在外部数据存储空间中的任何位置。
访问类型：
只读

HUGE_ROM_AC

描述：

保存 __huge_rom 定位的常量数据。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

HUGE_ROM_C

描述：
保存 __huge_rom 常量数据。这可以包括常量变量、字符串和聚合文字等。
段存储器类型：
CONST
存储器地址：
该段可以放置在外部数据存储空间中的任何位置。
访问类型：
只读

HUGE_Z

描述：
保存零初始化的 __huge 静态和全局变量。该段的内容由系统启动代码声明。除非已指定选项 --place_constants=code 或 --place_constants=data_rom，否则该段还包含未初始化或零初始化的 __huge 常量。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。
段存储器类型：
XDATA
存储器地址：
该段可以放置在外部数据存储空间中的任何位置。
访问类型：
读写

IDATA_AN

描述：
保存 __no_init __idata 定位的数据。除非已指定选项 --place_constants=code 或 --place_constants=data_rom，否则该段还保存声明为 __idata const 的已定位非初始化对象，并且在小数据模型，默认声明位于未初始化的常量对象。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

IDATA_I

描述：
保存 __idata 静态和全局初始化变量，通过在应用程序启动时从段 IDATA_ID 复制初始化。除非已指定选项 --place_constants=code 或 --place_constants=data_rom，否则该段还包含声明为 __idata const 的初始化对象，以及在小数据模型中默认声明的初始化常量对象。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。
段存储器类型：
DATA
存储器地址：
0-0xFF
访问类型：
只读

IDATA_ID

描述：
在 IDATA_I 段中保存 __idata 静态和全局变量的初始值。这些值在应用程序启动时从 IDATA_ID 复制到 IDATA_I。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。
段存储器类型：
CODE
存储器地址：
该段可以放置在代码存储空间中的任何位置。
访问类型：
只读

IDATA_N

描述：
保存静态和全局 __no_init __idata 变量。
段存储器类型：
DATA
存储器地址：
0-0xFF
访问类型：
读写

IDATA_Z

描述：

保存零初始化的 __idata 静态和全局变量。该段的内容由系统启动代码声明。在小数据模型中，除非已指定选项 --place_constants=code 或 --place_constants=data_rom，否则默认声明的零初始化常量对象也成立。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：

DATA

存储器地址：

0x0-0xFF

访问类型：

读写

INTVEC

描述：

保存使用 __interrupt 扩展关键字和 #pragma vector 指令生成的中断向量表。

段存储器类型：

CODE

存储器地址：

该段可以放置在内存中的任何位置。

访问类型：

只读

INTVEC_EXT2

描述：

当为 extended2 内核编译时，该段保存通过使用__interrupt extended 关键字和 #pragma vector 指令生成的中断向量表。

段存储器类型：

CODE

存储器地址：

该段可以放置在代码存储空间中的任何位置，但必须与段
BANKED_CODE_INTERRUPTS_EXT2 位于同一组中。

访问类型：

只读

IOVERLAY

描述：

保存使用 idata 覆盖调用约定调用的函数的静态覆盖区域。

段存储器类型：
DATA
存储器地址：
0x0-0xFF
访问类型：
读写

ISTACK

描述：
保存内部数据堆栈。
段存储器类型：
DATA
存储器地址：
0x0-0xFF
访问类型：
读写
另见：
有关如何在链接器配置文件中定义此段及其长度的信息，请参见设置堆栈内存，第 132 页。

IXDATA_AN

描述：

保存 __no_init __ixdata 定位的数据。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

IXDATA_I

描述：

保存 __ixdata 静态和全局初始化变量，通过在应用程序启动时从段 IXDATA_ID 复制来初始化。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
XDATA
存储器地址：
0-0xFFFF
访问类型：
只读

IXDATA_ID

描述：
在 IXDATA_I 段中保存 __ixdata 静态和全局变量的初始值。这些值在应用程序启动时从 IXDATA_ID 复制到 IXDATA_I。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。
段存储器类型：
CODE
存储器地址：
该段可以放置在代码存储空间中的任何位置。
访问类型：
只读

IXDATA_N

描述：
保存静态和全局 __no_init __ixdata 变量。
段存储器类型：
XDATA
存储器地址：
0-0xFFFF
访问类型：
读写

IXDATA_Z

描述：
保存零初始化的 __ixdata 静态和全局变量。该段的内容由系统启动代码声明。在小数据模型中，除非已指定选项 --place_constants=code 或 --place_constants=data_rom，否则默认声明的零初始化常量对象也成立。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。
段存储器类型：
XDATA
存储器地址：
0x0-0xFFFF
访问类型：
只读

NEAR_CODE

描述：
保存声明为 __near_func 的程序代码。
段存储器类型：
CODE
存储器地址：
0-0xFFFF
访问类型：
只读

PDATA_AN

描述：

保存 __no_init __pdata 定位的数据。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

PDATA_I

描述：

保存 __pdata 静态和全局初始化变量,通过在应用程序启动时从段 PDATA_ID 复制初始化。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
XDATA
存储器地址：
0-0xnnFF (xdata memory)；0-0xnnnnFF (far memory) 该段必须放在一个 256 字节的 xdata 页面或远内存中。因此，nn 可以是从 00 到 FF（xdata）的任何值，而 nnnn 可以是从 0000 到 FFFF（远）的任何值。
访问类型：
读写

PDATA_ID

描述：
在 PDATA_I 段中保存 __pdata 静态和全局变量的初始值。这些值在应用程序启动时从 PDATA_ID 复制到 PDATA_I。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。
段存储器类型：

CODE
存储器地址：
该段可以放置在代码存储空间中的任何位置。
访问类型：
只读

PDATA_N

描述：
保存静态和全局 __no_init __pdata 变量。
段存储器类型：
XDATA
存储器地址：
0-0xnnFF (xdata memory)；0-0xnnnnFF (far memory) 该段必须放在一个 256 字节的 xdata 页面或远内存中。因此，nn 可以是从 00 到 FF（xdata）的任何值，而 nnnn 可以是从 0000 到 FFFF（远）的任何值。
访问类型：
读写

PDATA_Z

描述：

保存零初始化的 __pdata 静态和全局变量。该段的内容由系统启动代码声明。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
XDATA
存储器地址：
0-0xnnFF (xdata memory)；0-0xnnnnFF (far memory) 该段必须放在一个 256 字节的 xdata 页面或远内存中。因此，nn 可以是从 00 到 FF（xdata）的任何值，而 nnnn 可以是从 0000 到 FFFF（远）的任何值。
访问类型：
读写

PSP

描述：
保存指向 pdata 堆栈的堆栈指针。
段存储器类型：
DATA
存储器地址：

0-0x7F
访问类型：
读写
另见：
有关设置堆栈指针的信息，请参阅系统启动，第 164 页。

PSTACK

描述：
保存参数数据堆栈。
段存储器类型：
XDATA
存储器地址：
0-0xnnFF (xdata memory)；0-0xnnnnFF (far memory) 该段必须放在一个 256 字节的 xdata 页面或远内存中。因此，nn 可以是从 00 到 FF（xdata）的任何值，而 nnnn 可以是从 0000 到 FFFF（远）的任何值。
访问类型：
读写
另见：
有关如何在链接器配置文件中定义此段及其长度的信息，请参见设置堆栈内存，第 132 页。

RCODE

描述：
保存汇编程序编写的运行时库代码。
段存储器类型：
CODE
存储器地址：
该段可以放置在代码存储空间中的任何位置。
访问类型：
只读

SFR_AN

描述：

保存 __no_init __sfr 定位的数据。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

VREG

描述：
保存编译器的虚拟寄存器区域。
段存储器类型：
DATA
存储器地址：
0-0x7FF
访问类型：
读写

XDATA_AN

描述：
保存 __no_init __xdata 定位的数据。除非已指定选项 --place_constants=code 或 --place_constants=data_rom，否则该段还包含声明为 __xdata const 的已定位非初始化对象，以及在大数据模型中，默认声明的已定位非初始化常量对象。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

XDATA_HEAP

描述：
保存用于在 xdata 内存中动态分配的数据的堆，即由 xdata_malloc 和 xdata_free 分配的数据，以及在 C++ 中的 new 和 delete。
段存储器类型：
XDATA
存储器地址：
该段可以放置在外部数据存储空间中的任何位置。
访问类型：
读写
另见：
有关如何在链接器配置文件中定义此段及其长度的信息，以及有关在 xdata 内存中对堆使用 new 和 delete 运算符的信息，请参阅设置堆内存，第 135 页和新建和删除 运算符，第 236 页。

XDATA_I

描述：

保存 __xdata 静态和全局初始化变量，通过在应用程序启动时从段 XDATA_ID 复制初始化。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
XDATA
存储器地址：
0-0xFFFF
访问类型：
读写

XDATA_ID

描述：
在 XDATA_I 段中保存 __xdata 静态和全局变量的初始值。这些值在应用程序启动时从 XDATA_ID 复制到 XDATA_I。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：
CODE
存储器地址：
该段可以放置在代码存储空间中的任何位置。
访问类型：
只读

XDATA_N

描述：
保存静态和全局 __no_init __xdata 变量。除非已指定选项 --place_constants=code 或 --place_constants=data_rom，否则该段还包含声明为 __xdata const 的非初始化对象，以及在大数据模型中，默认声明的非初始化常量对象。

段存储器类型：
XDATA
存储器地址：
0-0xFFFF
访问类型：
读写

XDATA_ROM_AC

描述：

保存 __xdata_rom 定位的常量数据。在大数据模型中，如果指定了编译器选项 --place_constants=data_rom，则该段还保存默认声明的初始化定位常量对象。参见 --place_constants，第 316 页。定位意味着使用 @ 运算符或 #pragma location 指令放置在绝对位置。因为位置是已知的，所以不需要在链接器命令文件中指定该段。

XDATA_ROM_C

描述：

保存 __xdata_rom 常量数据。这可以包括常量变量、字符串和聚合文字等。

段存储器类型：

CONST

存储器地址：

0-0xFFFF

访问类型：

只读

XDATA_Z

描述：

保存零初始化的 __huge 静态和全局变量。该段的内容由系统启动代码声明。在大数据模型中，除非指定了选项 --place_constants=code 或 --place_constants=data_rom，否则段还包含默认声明的零初始化常量对象。该段不能通过使用 -P 指令进行打包放置来放置在内存中，因为内容必须是连续的。相反，当您在链接器配置文件中定义此段时，必须使用 -Z 指令。

段存储器类型：

XDATA

存储器地址：
0-0xFFFF
访问类型：
读写
另见：
--output, -o, 第 315 页.

XSP

描述：
保存指向 xdata 堆栈的堆栈指针。
段存储器类型：
DATA
存储器地址：
0-0x7F
访问类型：
读写
另见：
有关设置堆栈指针的信息，请参阅系统启动，第 164 页。

XSTACK

描述：
保存 xdata 堆栈。
段存储器类型：
XDATA
存储器地址：
0-0FFFF
访问类型：
读写

另见：
有关如何在链接器配置文件中定义此段及其长度的信息，请参见设置堆栈内存，第 132 页。

页码:439

（空白页）
页码:440

标准 C 的实现定义的行为

● 实现定义行为的描述

如果您使用 C89 而不是标准 C，请参阅 C89 的实现定义行为，第 457 页。有关标准 C 和 C89 之间差异的简短概述，请参阅 C 语言概述，第 221 页。

本章内容适用于 DLIB 运行时环境。因为 CLIB 运行时环境不遵循标准 C，所以没有记录其实现定义的行为。另见 CLIB 运行时环境，第 181 页。

实现定义行为的描述

本节遵循与 C 标准相同的顺序。每一项都包括对解释实现定义的行为的 ISO 章节（在括号中）的引用。

注意：IAR Systems 实现遵循标准 C 的独立实现。这意味着可以在实现中排除标准库的部分内容。

J.3.1 翻译

诊断（3.10、5.1.1.3）

诊断以以下形式生成：filename,linenumber level[tag]: message 其中 filename 是遇到错误的源文件的名称，linenumber 是编译器检测到错误的行号，level 是级别消息的严重性（备注、警告、错误或致命错误），tag 是标识消息的唯一标记，message 是解释性消息，可能有几行。

空白字符 (5.1.1.2)
在翻译阶段三，保留每个非空的空白字符序列。

J.3.2 环境
字符集 (5.1.1.2)
源字符集与物理源文件多字节字符集相同。
默认情况下，使用标准 ASCII 字符集。但是，如果您使用 --enable_multibytes 编译器选项，则会使用主机字符集。

main (5.1.2.1)
程序启动时调用的函数称为 main。 main 没有声明原型，main 支持的唯一定义是：
诠释主要（无效）
要更改 DLIB 运行时环境的此行为，请参见系统初始化，第 167 页。

程序终止的影响 (5.1.2.1)
终止应用程序将执行返回到启动代码（就在调用 main 之后）。

定义 main 的替代方法 (5.1.2.2.1)
没有其他方法可以定义 main 函数。

main (5.1.2.2.1) 的 argv 参数
不支持 argv 参数。

流作为交互式设备 (5.1.2.3)
流 stdin、stdout 和 stderr 被视为交互式设备。
信号、它们的语义和默认处理 (7.14)
在 DLIB 运行时环境中，支持的信号集与标准 C 中的相同。引发的信号将什么也不做，除非信号函数被定制以适合应用程序。

计算异常的信号值 (7.14.1.1)
在 DLIB 运行时环境中，没有对应于计算异常的实现定义的值。

系统启动时的信号 (7.14.1.1)
在 DLIB 运行时环境中，没有在系统启动时执行的实现定义的信号。

环境名称 (7.20.4.5)
在 DLIB 运行时环境中，没有由 getenv 函数使用的实现定义的环境名称。

系统功能（7.20.4.6）
不支持系统功能。

J.3.3 标识符
标识符中的多字节字符 (6.4.2)
标识符中可能不会出现其他多字节字符。

标识符中的重要字符 (5.2.4.1, 6.1.2)
无论有无外部链接的标识符中的有效首字母数均保证不少于 200 个。

J.3.4 字符
一个字节中的位数 (3.6)
一个字节包含 8 位。

执行字符集成员值 (5.2.1)
执行字符集成员的值是 ASCII 字符集的值，可以通过主机字符集中额外字符的值来扩充。

字母转义序列 (5.2.2)
标准字母转义序列具有值 \a－7、\b－8、\f－12、\n－10、\r－13、\t－9 和 \v－11。

页码:443

基本执行字符集之外的字符（6.2.5）
存储在字符中的基本执行字符集之外的字符不会被转换。

普通炭（6.2.5、6.3.1.1）
普通字符被视为无符号字符。

源代码和执行字符集（6.4.4.4、5.1.1.2）
源字符集是可以出现在源文件中的合法字符集。默认情况下，源字符集是标准的 ASCII 字符集。但是，如果您使用命令行选项--enable_multibytes，那么源字符集将是主机的默认字符集。
执行字符集是可以出现在执行环境中出现的合法字符集。默认情况下，执行字符集是标准的 ASCII 字符集。
但是，如果您使用命令行选项--enable_multibytes，则执行字符集将是主机的默认字符集。DLIB 运行时环境需要一个多字节字符扫描器来支持一个多字节执行字符集。参见区域设置，第 178 页。

具有多个字符的整数字符常量（6.4.4.4）
包含多个字符的整数字符常量将被视为整数常量。该值将通过将最左边的字符作为最重要的字符，而最右边的字符作为最不重要的字符，以一个整数常量来计算。如果该值不能用整数常数表示，则将发出一条诊断消息。

具有多个字符的宽字符常数（6.4.4.4）
包含多个多字节字符的宽字符常量会生成诊断消息。

用于宽字符常数的区域设置（6.4.4.4）
默认情况下，将使用 C 区域设置。如果使用--enable_multibytes 编译器选项，则使用默认的主机区域设置。

用于宽字符串文字的区域设置（6.4.5）
默认情况下，将使用 C 区域设置。如果使用--enable_multibytes 编译器选项，则使用默认的主机区域设置。

源字符作为执行字符（6.4.5）
所有源字符都可以表示为执行字符。

J.3.5 整数
扩展整数类型（6.2.5）
没有扩展的整数类型。

整数值的范围（6.2.6.2）
整数值的表示是用这两个值的补体形式表示的。最重要的部分有符号；1 表示负，0 表示正，和零。
有关不同整数类型的范围的信息，请参阅基本数据类型-整数类型，第 326 页。

扩展整数类型的等级（6.3.1.1）
没有扩展的整数类型。

转换为有符号整数类型时的信号（6.3.1.3）
当一个整数被转换为一个有符号的整数类型时，不会发出任何信号。
签名位操作（6.5）
对有符号整数的位操作与对无符号整数的位操作相同；换句话说，符号位将被视为任何其他位，除了运算符>>将表现为算术右移。

J.3.6 浮点
浮点运算的精度（5.2.4.2.2）
浮点运算的准确性是未知的。

舍入行为（5.2.4.2.2）
FLT_ROUNDS 没有非标准值。

评价方法（5.2.4.2.2）
FLT_EVAL_METHOD 没有非标准值

将整数值转换为浮点值 (6.3.1.4)
当整数值转换为不能精确表示源值的浮点值时，使用四舍五入模式(FLT_ROUNDS 定义为 1)。

将浮点值转换为浮点值 (6.3.1.5)
当浮点值转换为不能精确表示源值的浮点值时，使用四舍五入模式(FLT_ROUNDS 定义为 1)。

表示浮点常量的值 (6.4.4.2)
使用四舍五入模式（FLT_ROUNDS 定义为 1）。

浮点值的收缩 (6.5)
浮点值是收缩的。但是，精度没有损失，并且由于不支持信令，这无关紧要。
FENV_ACCESS (7.6.1) 的默认状态
pragma 指令 FENV_ACCESS 的默认状态是 关。

额外的浮点机制 (7.6, 7.12)
没有额外的浮点异常、舍入模式、环境和分类。

FP_CONTRACT (7.12.2) 的默认状态
pragma 指令 FP_CONTRACT 的默认状态是 OFF。

J.3.7 数组和指针
从/到指针的转换 (6.3.2.3)
有关转换数据指针和函数指针的信息，请参阅转换，第 333 页。

ptrdiff_t (6.5.6)
有关 ptrdiff_t 的信息，请参阅 ptrdiff_t，第 333 页。

J.3.8 提示

认可寄存器关键字 (6.7.1)
用户对寄存器变量的请求不被尊重。

内联函数 (6.7.4)
用户对内联函数的请求增加了函数实际被内联到另一个函数中的机会,但并不意味着一定如此。参见内联函数，第 103 页。

J.3.9 结构、联合体、枚举和 位域

普通 位域的符号（6.7.2, 6.7.2.1）
关于如何处理 ″普通 ″int 位域的信息，请参见 Bitfields, page 327。

位域的可能类型 (6.7.2.1)
在编译器的扩展模式下，所有的整数类型都可以作为位域使用，见-e，第 302 页。

跨越存储单元边界的位域 (6.7.2.1)
一个位域总是被放置在一个且仅有一个存储单元中,这意味着该位域不能跨越存储单元的边界。

位域在一个单元中的分配顺序（6.7.2.1）
关于位字段如何在一个存储单元内分配的信息，请看位字段，第 327 页。
非位字段结构成员的对齐 (6.7.2.1)
结构的非位域成员的对齐方式与成员类型相同，参见对齐方式，第 325 页。

用于表示枚举类型的整数类型（6.7.2.2）
为特定枚举类型选择的整数类型取决于为该枚举类型定义的枚举常量。选择的整数类型是最小的。

页码:447

J.3.10 限定词

访问易失性对象 (6.7.3)

对具有 volatile 限定类型的对象的任何引用都是一种访问，请参见声明对象 volatile，第 334 页。

J.3.11 预处理指令

标头名称的映射 (6.4.7)

标头名称中的序列逐字映射到源文件名。反斜杠 '\' 不被视为转义序列。参见预处理器概述，第 387 页。

常量表达式中的字符常量 (6.10.1)

控制条件包含的常量表达式中的字符常量与执行字符集中相同字符常量的值相匹配。

单字符常量的值 (6.10.1)

如果将纯字符 (char) 视为有符号字符，则单字符常量只能具有负值，请参见 --char_is_signed，第 292 页。

包括括号内的文件名 (6.10.2)

有关尖括号 <> 中用于文件规范的搜索算法的信息,请参阅第 280 页的包含文件搜索过程。

包括引用的文件名 (6.10.2)

有关用于用引号括起来的文件规范的搜索算法的信息，请参阅第 280 页的包含文件搜索过程。

#include 指令中的预处理标记 (6.10.2)

#include 指令中的预处理标记与#include 指令之外的组合方式相同。

#include 指令的嵌套限制 (6.10.2)

#include 处理没有明确的嵌套限制。

通用字符名称 (6.10.3.2)

不支持通用字符名称 (UCN)

公认的编译指示指令 (6.10.6)

除了 Pragma 指令一章中描述的 pragma 指令之外，以下指令被识别并具有不确定的效果。如果在 Pragma 指令章节和此处都列出了 pragma 指令,则 Pragma 指令章节中提供的信息将覆盖此处的信息。

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

默认 __DATE__ 和 __TIME__ (6.10.8)
__TIME__ 和 __DATE__ 的定义始终可用。

J.3.12 库函数

额外的库设施 (5.1.2.1)
支持大多数标准库设施。其中一些--需要操作系统的--需要在应用程序中进行低级实现。有关详细信息，请参阅 DLIB 运行时环境，第 145 页。

断言功能打印的诊断 (7.2.1.1)
assert() 函数打印：
filename:linenr 表达式--当参数计算为零时断言失败。

浮点状态标志的表示 (7.6.2.2)
有关浮点状态标志的信息，请参阅 fenv.h，第 401 页。
Feraiseexcept 引发浮点异常 (7.6.2.3)
有关引发浮点异常的 feraiseexcept 函数的信息，请参阅浮点环境，第 329 页。

传递给 setlocale 函数的字符串 (7.11.1.1)
有关传递给 setlocale 函数的字符串的信息，请参见区域设置，第 178 页。

为 float_t 和 double_t 定义的类型 (7.12)
FLT_EVAL_METHOD 宏的值只能为 0。

域错误 (7.12.1)
没有任何函数会产生除标准要求之外的其他域错误。

域错误的返回值 (7.12.1)
数学函数为域错误返回浮点 NaN（不是数字）。

下溢错误 (7.12.1)
数学函数将 errno 设置为宏 ERANGE（errno.h 中的宏）并为下溢错误返回零。

fmod 返回值 (7.12.10.1)
fmod 函数在第二个参数为零时返回浮点 NaN。

remquo 的大小 (7.12.10.3)
幅度与模 INT_MAX 一致。

信号()（7.14.1.1）
不支持库的信号部分。
注意:信号的默认实现不执行任何操作。使用模板源代码来实现特定于应用程序的信号处理。
分别参见第 174 页的信号和第 172 页的提高。

NULL 宏 (7.17)
NULL 宏定义为 0。

终止换行符 (7.19.2)
stdout 流函数将换行符或文件结尾 (EOF) 识别为行的终止字符。

换行符前的空格字符 (7.19.2)
在换行符之前立即写入流的空格字符被保留。

附加到写入二进制流的数据的空字符 (7.19.2)
写入二进制流的数据不会附加空字符。

附加模式下的文件位置 (7.19.3)
当以附加模式打开文件时，文件位置最初位于文件的开头。

文件截断 (7.19.3)
文本流上的写操作是否会导致相关文件在该点之后被截断,取决于低级文件例程的特定于应用程序的实现。请参见输入和输出 (I/O) 简介，第 146 页。

文件缓冲 (7.19.3)
打开的文件可以是块缓冲的、行缓冲的或无缓冲的。

零长度文件 (7.19.3)
是否存在零长度文件取决于低级文件例程的特定于应用程序的实现。

合法文件名 (7.19.3)
文件名的合法性取决于低级文件例程的特定于应用程序的实现。

文件可以打开的次数 (7.19.3)
是否可以多次打开文件取决于低级文件例程的特定于应用程序的实现。

文件中的多字节字符 (7.19.3)
文件中多字节字符的编码取决于低级文件例程的特定于应用程序的实现。

删除()（7.19.4.1）
删除操作对打开文件的影响取决于低级文件例程的特定于应用程序的实现。请参见输入和输出 (I/O) 简介，第 146 页。

重命名()（7.19.4.2）
将文件重命名为现有文件名的效果取决于低级文件例程的特定于应用程序的实现。请参见输入和输出 (I/O) 简介，第 146 页。

删除打开的临时文件 (7.19.4.3)
是否删除打开的临时文件取决于低级文件例程的特定于应用程序的实现。

模式更改 (7.19.5.4)
freopen 关闭命名流，然后以新模式重新打开它。流 stdin、stdout 和 stderr 可以在任何新模式下重新打开。

用于打印无穷大或 NaN 的样式（7.19.6.1、7.24.2.1）
用于为浮点常量打印无穷大或 NaN 的样式分别是 inf 和 nan（INF 和 NAN 用于 F 转换说明符）。 n-char-sequence 不用于 nan。

%p in printf() (7.19.6.1, 7.24.2.1)
printf() 的 %p 转换说明符 print 指针的参数被视为具有 void * 类型。该值将打印为十六进制数字，类似于使用 %x 转换说明符。

scanf 中的读取范围（7.19.6.2、7.24.2.1）
-（破折号）字符始终被视为范围符号。

%p 在 scanf (7.19.6.2, 7.24.2.2)
scanf() 的 %p 转换说明符扫描指针读取一个十六进制数并将其转换为 void * 类型的值。

文件位置错误（7.19.9.1、7.19.9.3、7.19.9.4）
对于文件位置错误，函数 fgetpos、ftell 和 fsetpos 将 EFPOS 存储在 errno 中。

nan (7.20.1.3, 7.24.4.1.1) 之后的 n 字符序列
读取并忽略 NaN 之后的 n 字符序列。

下溢时的 errno 值 (7.20.1.3, 7.24.4.1.1)
如果遇到下溢，则将 errno 设置为 ERANGE。

零大小的堆对象 (7.20.3)
对零大小堆对象的请求将返回有效指针而不是空指针。

中止和退出的行为 (7.20.4.1, 7.20.4.4)
调用 abort() 或 _Exit() 不会刷新流缓冲区，不会关闭打开的流，也不会删除临时文件。

终止状态（7.20.4.1、7.20.4.3、7.20.4.4）
终止状态将作为参数传播到 __exit()。 exit() 和 _Exit() 使用输入参数，而 abort 使用 EXIT_FAILURE。

系统函数返回值（7.20.4.6）
不支持系统功能。

时区 (7.23.1)
本地时区和夏令时必须由应用程序定义。有关详细信息，请参阅第 175 页的 __time32。

时间范围和精度 (7.23)
时间接口支持从 1900 到 2035 的年份，并为 time_t 使用 32 位整数。应用程序必须提供时间和时钟功能的实际实现。分别参见第 175 页的 __time32 和第 170 页的时钟。

时钟()（7.23.2.1）
应用程序必须提供时钟功能的实现。参见时钟，第 170 页。

%Z 替换字符串 (7.23.3.5, 7.24.5.1)
默认情况下，“:”被用作 %Z 的替代品。您的应用程序应该实现时区处理。参见 __time32，第 175 页。

数学函数舍入模式 (F.9)
math.h 中的函数遵循 FLT-ROUNDS 中的舍入方向模式。

J.3.13 架构
分配给某些宏的值和表达式（5.2.4.2、7.18.2、7.18.3）
一个字节中总是有 8 位。
MB_LEN_MAX 最多为 6 个字节，具体取决于所使用的库配置。
有关所有基本类型的大小、范围等信息，请参阅第 325 页的数据表示。
stdint.h 中定义的精确宽度、最小宽度和最快最小宽度整数类型的限制宏与 char、short、int、long 和 long long 具有相同的范围。
浮点常数 FLT_ROUNDS 的值为 1（最接近），浮点常数 FLT_EVAL_METHOD 的值为 0（按原样处理）。

字节的数量、顺序和编码 (6.2.6.1)
请参见数据表示，第 325 页。

sizeof 运算符的结果值 (6.5.3.4)
请参见数据表示，第 325 页。

J.4 语言环境

源和执行字符集的成员 (5.2.1)
默认情况下，编译器接受主机默认字符集中的所有单字节字符。如果使用编译器选项 --enable_multibytes，则主机多字节字符也可以在注释和字符串文字中接受。

附加字符集的含义（5.2.1.2）
扩展源字符集中的任何多字节字符都被逐字翻译成扩展执行字符集。由您的应用程序在库配置的支持下正确处理字符。

用于编码多字节字符的转换状态 (5.2.1.2)
使用编译器选项 --enable_multibytes 可以将主机的默认多字节字符用作扩展源字符。

连续打印字符的方向 (5.2.2)
该应用程序定义了显示设备的特性。

小数点字符 (7.1.1)
默认的小数点字符是“.”。您可以通过定义库配置符号 _LOCALE_DECIMAL_POINT 来重新定义它。

打印字符（7.4、7.25.2）
打印字符集由所选语言环境决定。

控制字符（7.4、7.25.2）
控制字符集由所选语言环境决定。

测试的字符（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)
测试的字符集由选择的语言环境决定

页码:455

原生环境（7.1.1.1）
本机环境与“C”语言环境相同。

数字转换函数的主题序列（7.20.1，7.24.4.1）
没有其他主题序列可以被数值转换函数接受。

执行字符集的排序规则（7.21.4.3、7.24.4.4.2）
执行字符集的排序规则由所选语言环境决定。

strerror（7.21.6.2）返回的消息
strerror 函数根据参数返回的消息是：

参数	消息
EZERO	没有错误
EDOM	域错误
ERANGE	范围错误
EFPOS	文件定位错误
EILSEQ	多字节编码错误
<0 || >99	未知错误
所有其他错误	nnn

表 48：strerror() 返回的消息—DLIB 运行环境

C89 的实现定义的行为

● 实现定义行为的描述

如果您使用的是标准 C 而不是 C89，请参阅标准 C 的实现定义行为，第 441 页。有关标准 C 和 C89 之间差异的简短概述，请参阅 C 语言概述，第 221 页。

实现定义行为的描述

描述遵循与 ISO 附录相同的顺序。涵盖的每个项目都包括对解释实现定义的行为的 ISO 章节（在括号中）的引用。

编译

诊断 (5.1.1.3)

诊断以以下形式生成：

filename,linenumber level[tag]: message 其中 filename 是遇到错误的源文件的名称，linenumber 是编译器检测到错误的行号，level 是消息的严重程度（备注，警告、错误或致命错误），标签是标识消息的唯一标签，消息是解释性消息，可能有几行。

环境

main 的参数(5.1.2.2.2.1)

程序启动时调用的函数称为 main。没有为 main 声明原型，并且 main 支持的唯一定义是：

诠释主要（无效）

要更改 DLIB 运行时环境的此行为，请参阅系统初始化，第 167 页。要更改 CLIB 运行时环境的此行为，请参阅自定义系统初始化，第 188 页。

交互式设备 (5.1.2.3)
流 stdin 和 stdout 被视为交互式设备。
身份标识
没有外部链接的重要字符 (6.1.2)
没有外部链接的标识符中的有效初始字符数为 200。

具有外部联系的重要特征 (6.1.2)
具有外部链接的标识符中的有效初始字符数为 200。

案例区别很重要 (6.1.2)
具有外部链接的标识符被视为区分大小写。

字符集
源和执行字符集 (5.2.1)
源字符集是可以出现在源文件中的合法字符集。默认源字符集是标准的 ASCII 字符集。但是,如果您使用命令行选项 --enable_multibytes,源字符集将是主机计算机的默认字符集。
执行字符集是可以出现在执行环境中的合法字符集。默认执行字符集是标准 ASCII 字符集。但是,如果您使用命令行选项 --enable_multibytes,则执行字符集将是主机的默认字符集。
DLIB 运行时环境需要一个多字节字符扫描器来支持多字节执行字符集。 CLIB 运行时环境不支持多字节字符。
请参见区域设置，第 178 页。

执行字符集中每个字符的位数 (5.2.4.2.1)
字符中的位数由清单常量 CHAR_BIT 表示。
标准包含文件 limits.h 将 CHAR_BIT 定义为 8。

字符映射 (6.1.3.4)
源字符集成员(在字符和字符串文字中)到执行字符集成员的映射是以一对一的方式进行的。换句话说，除了 ISO 标准中列出的转义序列外，字符集中的每个成员都使用相同的表示值。

未表示的字符常量 (6.1.3.4)
包含基本执行字符集或宽字符常量的扩展字符集中未表示的字符或转义序列的整数字符常量的值会生成诊断消息，并将被截断以适应执行字符集。

具有多个字符的字符常量 (6.1.3.4)
包含多个字符的整数字符常量将被视为整数常量。
该值将通过将最左边的字符视为最重要的字符,将最右边的字符视为最不重要的字符来计算，在一个整数常量中。如果值不能以整数常量表示，则会发出诊断消息。
包含多个多字节字符的宽字符常量会生成诊断消息。

转换多字节字符 (6.1.3.4)
唯一支持的语言环境--即 IAR C/C++ 编译器提供的唯一语言环境--是“C”语言环境。如果您使用命令行选项 --enable_multibytes,如果您向库中添加支持多字节的语言环境或多字节字符扫描器,DLIB 运行时环境将支持多字节字符。 CLIB 运行时环境不支持多字节字符。请参见区域设置，第 178 页。

“普通”字符的范围（6.2.1.1）
'plain'（平纹字体的） char 与 unsigned char 具有相同的范围。

整数

整数值范围 (6.1.2.5)
整数值的表示形式为二进制补码形式。最高有效位持有符号； 1 表示负数，0 表示正数和零。
有关不同整数类型范围的信息，请参阅基本数据类型--整数类型，第 326 页。

整数降级 (6.2.1.2)
通过截断将整数转换为更短的有符号整数。如果在将无符号整数转换为等长有符号整数时无法表示该值，则位模式保持不变。换句话说，足够大的值将被转换为负值。
有符号位运算 (6.3)
对有符号整数的按位运算与对无符号整数的按位运算的工作方式相同；换句话说，符号位将被视为任何其他位，除了运算符 >> 它将表现为算术右移。

整数除法的余数符号 (6.3.5)
整数除法余数的符号与被除数的符号相同。

负值有符号右移 (6.3.7)
负值有符号整数类型的右移结果保留符号位。
例如，将 0xFF00 下移一级产生 0xFF80。

浮点
浮点值的表示 (6.1.2.5)
各种浮点数的表示和集合遵循 IEEE 854 - 1987。典型的浮点数由符号位 (s)、偏置指数 (e) 和尾数 (m) 组成。
有关不同浮点类型的范围和大小的信息，请参见基本数据类型 - 浮点类型，第 329 页：float 和 double。

将整数值转换为浮点值 (6.2.1.3)
当整数转换为不能精确表示该值的浮点值时，该值将四舍五入（向上或向下）到最接近的合适值。
降级浮点值 (6.2.1.4)
当浮点值转换为不能精确表示该值的较窄类型的浮点值时，该值将四舍五入（向上或向下）到最接近的合适值。

数组和指针

size_t (6.3.3.4, 7.1.1)

有关 size_t 的信息，请参见第 333 页的 size_t。

从/到指针的转换 (6.3.4)

有关数据指针和函数指针转换的信息，请参见转换，第 333 页。

ptrdiff_t (6.3.6, 7.1.1)

有关 ptrdiff_t 的信息，请参见 ptrdiff_t，第 333 页。

寄存器

遵守 register 关键字 (6.5.1)

不接受用户对寄存器变量的请求。

结构、联合、枚举和位域

对联合的不当访问 (6.3.2.3)

如果联合通过成员获取其值，然后使用不同类型的成员访问，则结果仅取决于第一个成员的内部存储。

结构成员的填充和对齐 (6.5.2.1)

有关数据对象对齐要求的信息，请参见第 326 页的基本数据类型 — 整数类型部分。

“普通”位域的符号 (6.5.2.1)

'plain' int 位域被视为带符号的 int 位域。所有整数类型都允许作为位域。

单元内位域的分配顺序 (6.5.2.1)

位域在从最低有效位到最高有效位的整数内分配。

位域能否跨越存储单元边界 (6.5.2.1)

位域不能跨越所选位域整数类型的存储单元边界。

选择整数类型来表示枚举类型 (6.5.2.2)
为特定枚举类型选择的整数类型取决于为枚举类型定义的枚举常量。选择的整数类型是可能的最小的。
限定符

访问易失性对象 (6.5.3)
对具有 volatile 限定类型的对象的任何引用都是一种访问。

声明者
声明符的最大数量 (6.5.4)
声明符的数量没有限制。该数量仅受可用内存的限制。

声明
case 语句的最大数量 (6.6.4.2)
switch 语句中的 case 语句（case 值）的数量不受限制。该数量仅受可用内存的限制。

预处理指令
字符常量和条件包含 (6.8.1)
预处理器指令中使用的字符集与执行字符集相同。如果将“普通”字符视为有符号字符，则预处理器会识别负字符值。

包括括号内的文件名 (6.8.2)
对于用尖括号括起来的文件规范，预处理器不会搜索父文件的目录。父文件是包含#include的文件指示。相反，它首先在编译器命令行指定的目录中搜索文件。

包括引用的文件名 (6.8.2)
对于用引号括起来的文件规范，预处理器目录搜索从父文件的目录开始，然后通过任何祖父文件的目录进行。

因此，搜索相对于包含当前正在处理的源文件的目录开始。如果没有祖父文件并且未找到该文件，则继续搜索，就像文件名包含在尖括号中一样。

字符序列 (6.8.2)

预处理器指令使用源字符集，但转义序列除外。 因此，要指定包含文件的路径，只需使用一个反斜杠：

```
#include "mydirectory\myfile"
```

在源代码中，需要两个反斜杠：

```
file = fopen("mydirectory\\myfile","rt");
```

公认的编译指示指令 (6.8.6)

除了 Pragma 指令一章中描述的 pragma 指令之外，以下指令被识别并具有不确定的效果。如果在 Pragma 指令章节和此处都列出了 pragma 指令,则 Pragma 指令章节中提供的信息将覆盖此处的信息。

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

页码:463

默认 __DATE__ 和 __TIME__ (6.8.8)

__TIME__ 和 __DATE__ 的定义始终可用。

IAR DLIB 运行时环境的库函数

请注意，此列表中的某些项目仅适用于库配置支持文件描述符时。 有关运行时库配置的更多信息，请参阅 DLIB 运行时环境一章。

NULL 宏 (7.1.6)

NULL 宏定义为 0。

断言功能打印的诊断 (7.2)

assert() 函数打印：

filename:linenr 表达式--当参数计算为零时断言失败。

域错误 (7.5.1)

NaN（非数字）将由域错误的数学函数返回。浮点值的下溢将 errno 设置为 ERANGE (7.5.1) 数学函数在下溢范围错误时将整数表达式 errno 设置为 ERANGE（errno.h 中的宏）。

fmod() 功能 (7.5.6.4)

如果 fmod() 的第二个参数为零，则函数返回 NaN； errno 设置为 EDOM。

信号()（7.7.1.1）
不支持库的信号部分。
注意：信号的默认实现不执行任何操作。使用模板源代码来实现特定于应用程序的信号处理。
分别参见第 174 页的信号和第 172 页的提高。

终止换行符 (7.9.2)
stdout 流函数将换行符或文件结尾 (EOF) 识别为行的终止字符。

空行 (7.9.2)
在换行符之前立即写入标准输出流的空格字符被保留。无法通过 stdin 流读取通过 stdout 流写入的行。

附加到写入二进制流的数据的空字符 (7.9.2)
写入二进制流的数据不会附加空字符。

文件 (7.9.3)
附加模式流的文件位置指示符最初是位于文件的开头还是结尾，取决于低级文件例程的特定于应用程序的实现。
文本流上的写操作是否会导致相关文件在该点之后被截断，取决于低级文件例程的特定于应用程序的实现。请参见输入和输出 (I/O) 简介，第 146 页。
文件缓冲的特点是实现支持无缓冲、行缓冲或完全缓冲的文件。
零长度文件是否实际存在取决于低级文件例程的特定于应用程序的实现。
组成有效文件名的规则取决于低级文件例程的特定于应用程序的实现。
同一文件是否可以同时打开多次取决于低级文件例程的特定于应用程序的实现。

删除()（7.9.4.1）
删除操作对打开文件的影响取决于低级文件例程的特定于应用程序的实现。请参见输入和输出 (I/O) 简介，第 146 页。

重命名()（7.9.4.2）
将文件重命名为现有文件名的效果取决于低级文件例程的特定于应用程序的实现。请参见输入和输出 (I/O) 简介，第 146 页。

%p in printf() (7.9.6.1)
printf() 的 %p 转换说明符 print 指针的参数被视为具有 void* 类型。该值将打印为十六进制数字，类似于使用 %x 转换说明符。

%p 在 scanf() (7.9.6.2)
scanf() 的 %p 转换说明符扫描指针读取一个十六进制数并将其转换为 void * 类型的值。scanf() (7.9.6.2) 中的读取范围-（破折号）字符始终被视为范围符号。

文件位置错误（7.9.9.1、7.9.9.4）
对于文件位置错误，函数 fgetpos 和 ftell 将 EFPOS 存储在 errno 中。

perror() 生成的消息 (7.9.10.4)
生成的消息是：
用户提供的前缀：错误消息

分配零字节内存 (7.10.3)
calloc()、malloc() 和 realloc() 函数接受零作为参数。
将分配内存，返回指向该内存的有效指针，并且稍后可以通过 realloc 修改内存块。

abort() 的行为 (7.10.4.1)
abort() 函数不刷新流缓冲区，也不处理文件，因为这是不受支持的功能。

exit() 的行为 (7.10.4.3)

传递给 exit 函数的参数将是 main 函数返回给 cstartup 的返回值。

环境 (7.10.4.4)

可用的环境名称集和更改环境列表的方法在 getenv，第 170 页中进行了描述。

system() （7.10.4.5）

命令处理器的工作方式取决于您如何实现系统功能。 参见系统，第 175 页。

strerror() 返回的消息 (7.11.6.2)

strerror() 根据参数返回的消息是：

参数	消息
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

表 49: strerror ()- DLIB 执行期函式库返回的消息

时区 (7.12.1)

本地时区和夏令时实现在 __time32，第 175 页中进行了描述。

clock() (7.12.2.1)

系统时钟从哪里开始计数取决于您如何实现时钟功能。 参见时钟，第 170 页。

库函数的 CLIB 运行时环境

NULL 宏 (7.1.6)

NULL 宏定义为 (void *) 0.

assert 函数打印的诊断报告(7.2)
assert()函数打印出：
断言失败：表达式、文件名、行号-当参数计算结果为零时。

领域错误 (7.5.1)
HUGE_VAL，双浮点类型中最大的可表示值，将由数学函数在域错误中返回。

浮点值的下溢将 errno 设置为 ERANGE(7.5.1)
数学函数在下溢范围错误时将整数表达式 errno 设置为 ERANGE（errno.h 中的一个宏）。

fmod()功能 (7.5.6.4)
如果 fmod()的第二个参数是 0，该函数返回 0（它不改变整数表达式 errno）。

signal() (7.7.1.1)
不支持库中的信号部分。

终止换行符 (7.9.2)
stdout 流函数将换行符或文件结束符（EOF）作为一行的结束符。

空白行 (7.9.2)
在换行符之前写到 stdout 流的空格字符被保留下来。没有办法通过 stdin 流读取通过 stdout 流写入的那一行。

空字符附加写入二进制流的数据(7.9.2)
没有实现二进制流。

文件 (7.9.3)
除了 stdin 和 stdout，没有其他流。这意味着没有实现文件系统。

页码:468

remove() （7.9.4.1）
除了 stdin 和 stdout 之外，没有其他流。这意味着没有实现文件系统。

rename()（7.9.4.2）
除了 stdin 和 stdout 之外，没有其他流。这意味着没有实现文件系统。

%printf()中的 p（7.9.6.1）
printf()的%p 转换说明符 print pointer 的参数被视为具有“char*”类型。该值将打印为十六进制数，类似于使用%x 转换说明符。

%scanf()中的 p（7.9.6.2）
scanf()的%p 转换说明符 scan pointer 读取十六进制数并将其转换为类型为“void*”的值。

scanf()中的读数范围（7.9.6.2）
（破折号）字符始终显式地视为-字符。

文件位置错误（7.9.9.1、7.9.9.4）
除了 stdin 和 stdout 之外，没有其他流。这意味着没有实现文件系统。

perror()生成的消息（7.9.10.4）
不支持 perror()。

分配零字节内存（7.10.3）
calloc()、malloc()和 realloc()函数接受零作为参数。
将分配内存，并返回指向该内存的有效指针，随后可通过 realloc 修改内存块。

abort()的行为（7.10.4.1）
abort()函数不刷新流缓冲区，也不处理文件，因为这是一个不受支持的功能。

exit()的行为（7.10.4.3）
exit()函数不返回

环境 (7.10.4.4)
不支持环境。

system() (7.10.4.5)
不支持 system() 函数。

strerror() 返回的消息 (7.11.6.2)
strerror() 根据参数返回的消息是：

参数	消息
EZERO	无错误
EDOM	域错误
ERANGE	范围错误
<0 || >99	未知错误
其他错误	No.xx

表 50：strerror()-CLIB 运行时环境的返回消息

time zon (7.12.1)
不支持时区功能。

clock() (7.12.2.1)
不支持 clock()函数

索引(中英对照) index

附录：

翻译说明：

IAR C/C++编译器是业界较好的支持 8051 单片机 C 及 C++的编译器（注：国内使用较多的是 KEIL C51,但其不支持 C++。（STC32 系列的 C251 单片机目前主流仅有 KEIL C251 编译器，且不支持 C++，KEIL MDK 支持 ARM 的 C++）。为方便大家使用 IAR，译者试对 IAR 公司的《IAR C/C++ Compiler User Guide for the 8051 Microcontroller Architecture》（第七版：2017 年 3月，元件编号：C8051-7）进行翻译。（英文原版下载地址：https://wwwfiles.iar.com/8051/webic/doc/EW8051_CompilerGuide.pdf ，若地址改变，请至其官网搜索下载）

翻译中使用了多个翻译网站（主要为谷歌及百度、必应等，特此感谢）的服务，机器翻译结果互相对照并人工处理，力求做到准确完整。同时译文的排版、页码也尽量与原版一一对应，方便读者查阅对照。具体内容信息请以英文原版为准。原文档版权属于其合法持有人使用。译者从 2022.03 开始试翻译处理，目前翻译版本为 V0.99A1，因水平有限，欢迎指正，电子邮箱为：stcisp @ 163.com。

STC 网站可下载 STC 单片机的 C/C++参考代码。另可在搜索引擎搜索“IAR C++ 8051 例程”。

autopccopy
2022.03-2023.08.09

以下内容摘自 IAR 官网说明：
https://www.iar.com/cn/product/architectures/iar-embedded-workbench-for-8051/

IAR C/C++ 语言和标准

IAR C/C++ 编译器提供 C 和 C++ 编程语言的不同方言，以及针对嵌入式编程的不同扩展（请注意，并非所有语言标准都支持所有目标实现）。编译器可以被指示禁用扩展，以严格遵守标准。

支持多种行业标准的调试和映像格式，与大多数流行的调试器和模拟器兼容。其中包括 ELF/DWARF（如适用）。

符合 ISO/ANSI C/C++ 标准

IAR C/C++ 编译器遵守以下 C 语言编程标准的独立实现：

INCITS/ISO/IEC 9899:2018，即 C18（仅限最新版本）

编译器支持所有 C++17 功能。C++ 库支持 C++14，没有补充 C++17（仅限最新版本）

ISO/IEC 14882:2015，即 C++14

INCITS/ISO/IEC 9899:2012，即 C11

ANSI X3.159-1989，即 C89

不同编译器的 ISO/ANSI C/C++ 兼容性水平存在差异。如需了解完整信息，请参考您所选产品中的 IAR C/C++ 编译器用户文档。

IEEE 754 标准

IAR Embedded Workbench 支持 IEEE 754 标准的浮点运算。

MISRA C

MISRA C 是由 MISRA（汽车工业软件可靠性协会）制定的 C 编程语言的软件开发标准。它的目的是促进嵌入式系统中的代码安全、可移植性和可靠性，特别是那些用 ISO C 语言编程的系统。

MISRA C 标准的第一版是《汽车软件中 C 语言的使用指南》，诞生于 1998 年。2004 年该协会发布了第二版指南，其中进行了许多重大修改，包括对规则进行了全面的重新编号。此外，MISRA C:2012 的扩展支持和 MISRA C++:2008 用于识别 C++ 标准中的不安全代码结构也被添加到标准中。

IAR Embedded Workbench 包含一个插件产品 C-STAT，因此您可以检查应用是否符合 MISRA C:2004、MISRA C++:2008 和 MISRA C:2012 定义的规则。

测试验证

我们使用以下商业测试套件来测试我们的工具是否符合标准：

适于 ISO/IEC C 标准一致性的 Plum Hall 验证测试套件
适于测试 C++ 标准一致性的 Perennial EC++ 验证套件
Dinkum C++ Proofer 测试我们的库是否符合 C 和 C++ 标准，并根据 C++ 标准测试我们的 STL 实现

除商业套件外，我们还使用一些内部测试套件来测试新功能、执行回归测试、纠正错误等。