



DSP集成开发环境

CCS

开发指南

尹 勇 欧光军 关荣锋 编著



北京航空航天大学出版社



责任编辑：孔祥燮

封面设计：伟路 DESIGN

TI 公司 DSP 器件系列丛书

DSP 集成开发环境 CCS 开发指南

尹 勇 欧光军 关荣锋 编著

TMS320LF240x DSP C 语言开发应用

刘和平

等编著

DSP 基础理论与应用技术

李哲英

等编著

DSP 基础与应用系统设计

王念旭

编著

TMS320C54x DSP 结构、原理及应用

戴明桢 周建江

编

TMS320C54x DSP 应用程序设计与开发

刘益成

编著

TMS320C54x DSP 应用系统设计

郑 红 吴 冠

编著

TMS320LF240x DSP 结构、原理及应用

刘和平

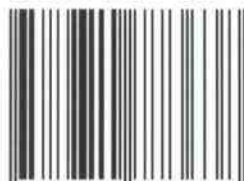
等编著

TMS320F206 DSP 结构、原理及应用

李 刚

等编著

ISBN 7-81077-363-1



9 787810 773638 >

ISBN 7-81077-363-1

定价：28.00元

DSP 集成开发环境 CCS 使用指南

尹 勇 欧光军 关荣锋 编著

北京航空航天大学出版社

内容简介

本书详细介绍了如何使用 TI 公司的 DSP 软件开发集成环境 CCS2 开发和调试 C 语言及 DSP 汇编语言程序, 尽量达到帮助读者熟练使用 CCS2 软件的目的。本书不是介绍 DSP 软件和硬件方面的知识, 而是采用实例方式循序渐进地介绍如何操作 CCS2 软件。读者只要按照实例步骤实践, 就能在最短的时间内熟练使用 CCS2。

本书可供从事 DSP 应用系统开发的技术人员、工程师以及高等院校工科电子类专业师生学习参考。

图书在版编目(CIP)数据

DSP 集成开发环境 CCS 使用指南/尹勇等编著. —北京:
北京航空航天大学出版社, 2003. 11
ISBN 7-81077-363-1

I. D… II. 尹… III. 软件工具, CCS 2.0—指南
IV. TP311.56-62

中国版本图书馆 CIP 数据核字(2003)第 094310 号

DSP 集成开发环境 CCS 使用指南

尹 勇 欧光军 关荣锋 编著

责任编辑 孔祥燮

*

北京航空航天大学出版社出版发行

北京市海淀区学院路 37 号(邮编 100083) 发行部电话:(010)82317024 传真:(010)82328026

<http://www.buaapress.com.cn> E-mail: bhp@263.net

北京市松源印刷有限公司印装 各地书店经销

*

开本: 787×1092 1/16 印张: 18.25 字数: 467 千字

2003 年 11 月第 1 版 2003 年 11 月第 1 次印刷 印数: 5 000 册

ISBN 7-81077-363-1 定价: 28.00 元

前 言

数字信号处理器 DSP(Digital Signal Processor)是针对数字信号处理需要而设计的一种可编程的单片机,是现代电子技术、计算机技术和信号处理技术相结合的产物。可编程 DSP 芯片的开发与应用是当前电子领域的热点,CCS 的推出是 DSP 软件开发的一次革命性突破。CCS2(Code Composer Studio Version 2.0)代码调试器是一种针对标准 TMS320 调试接口的 DSP 芯片集成开发环境 IDE(Integrated Development Environment),由 TI 公司在 1999 年推出。CCS 目前有 CCS1.1、CCS1.2、CCS2.0 和 CCS2.2 等几个版本,有 CCS2000(针对 C2XX)、CCS5000(针对 C54XX、C55XX)、CCS6000(针对 C6X)等几个不同的型号。

本书讨论了基于 TMS320C54X 系列芯片的 CCS2 集成开发环境的使用,尽量达到帮助读者熟练使用 CCS2.0 软件的目的。读者应了解 TMS320C54X 的硬件结构以及芯片的各种资源,熟悉 TMS320C54X 的指令系统。同时,还要求读者会使用 DSP 汇编语言和标准 C/C++ 语言进行程序设计。

本书不涉及 DSP 的硬件和软件开发设计以及信号处理的理论问题,在有关的程序例子中假定读者对所涉及的 DSP 硬件和软件开发设计以及信号处理的理论有基本的了解。

全书正文共 9 章。

第 1 章介绍 CCS2 的安装与配置;

第 2 章简单介绍 CCS2;

第 3 章深入介绍 CCS 集成开发环境;

第 4 章介绍 CCS 的 C 语言调试实例;

第 5 章介绍 CCS 的汇编语言调试实例;

第 6~9 章介绍 CCS2 高级使用——使用文件 I/O、DSP/BIOS 原理与应用、RTDX 原理与应用和使用 GEL。为了便于查阅,在附录中列出了 TMS320C54X 的汇编助记符指令集和汇编伪指令。

本书的第 1 章、第 2 章、第 3 章、第 5 章由华中科技大学尹勇博士执笔,第 4 章由华中科技大学欧光军博士执笔,第 6 章由华中科技大学关荣锋博士执笔,第 7 章由华中科技大学李红杰博士执笔,第 8 章、第 9 章和附录由华中科技大学张超勇、朱传军、李林凌博士和将闻悦同学执笔。全书由尹勇统编。本书在编写过程中受到中南财经政法大学项铭硕士的鼎力帮助,特表感谢。书的出版同时得到北京航空航天大学大力支持和鼓励,在此深表敬意。

由于作者水平有限,书中出现的错误和不妥之处,恳请广大读者批评指正!

作 者

2003 年 6 月于喻园

《嵌入式实时操作系统—— $\mu\text{C}/\text{OS}-\text{II}$, 第2版》

《MicroC/OS-II, The Real-Time Kernel, second edition》

[美] Jean J. Labrosse 著 邵贝贝 等译

2003年5月第1版 定价:79.00(含光盘)

内容简介

$\mu\text{C}/\text{OS}-\text{II}$ 是著名的、源码公开的实时内核,是专为嵌入式应用设计的,可用于各类8位、16位和32位单片机或DSP。从 $\mu\text{C}/\text{OS}$ 算起,该内核已有10余年应用史,在诸多领域得到了广泛应用。本书是“MicroC/OS-II The Real-Time Kernel”一书的第2版本,在第1版本(V2.0)基础上做了重大改进与升级。通过对 $\mu\text{C}/\text{OS}-\text{II}$ 源代码的分析与描述,讲述了多任务实时的基本概念、竞争与调度算法、任务间同步与通信、存储与定时的管理以及如何处理优先级反转问题;介绍如何将 $\mu\text{C}/\text{OS}-\text{II}$ 移植到不同CPU上,如何调试移植代码。在所附光盘中,给出已通过FAA安全认证的 $\mu\text{C}/\text{OS}-\text{II}$ V2.52的全部源码以及可在PC机上运行的移植范例。本书可用作高等院校嵌入式实时系统课程教材或工程师培训教材,也可供嵌入式应用开发人员研究与使用。



《SoC设计与测试》

System-on-a-Chip: Design and Test

[美] Rochit Rajsuman 著 于敦山 盛世敏 田泽译

2003年8月第1版 定价:35.0元

内容简介

本书概括地描述了与SoC设计和测试有关的基本问题,介绍了最新的设计与测试的方法和流程。全书分2大部分内容。第一部分是SoC的设计。在介绍了SoC的产生背景以及相关概念后,讲述了数字逻辑核(包括软核、固核及硬核)的设计方法和嵌入式存储器与模拟电路设计的有关问题,对IP核和SoC的设计验证和测试平台开发也做了详细说明。第二部分讨论了SoC的测试。分别讨论了嵌入式数字逻辑核,嵌入式存储器,以及嵌入式模拟、混合信号IP核的测试方法。另外还单设一章讨论了Iddq的可测性设计和测试向量的生成。最后讨论了生产测试的问题。本书读者对象为SoC和ASIC的设计工程师,IP核的设计与测试工程师,以及相关专业的高等院校师生。对SoC设计感兴趣的其他相关专业人员也可以阅读本书以了解SoC设计与测试的概貌。



图书邮购地址:北京航空航天大学出版社邮购组(邮编:100083) 另加2元挂号费

电话:010-82316936 传真:010-82317031 Email: bhpress@263.net

单片机与嵌入式系统图书详细介绍请查阅出版社网站: <http://www.buaapress.com.cn>

投稿单片机与嵌入式系统图书请联系:

北航出版社 马广云 电话:010-82317022 传真:010-82317022 Email: press@public3.bta.net.cn

《单片机与嵌入式系统应用》月刊

- 北京航空航天大学出版社承办
- 何立民教授主编
- 中央级科技期刊
- 嵌入式系统专业期刊
- 特色: 专业期刊、专家办刊、着眼世界、面向国内、应用为主、读者第一
- 已开辟有 8 个栏目: 业界论坛、专题论述、技术纵横、新器件新技术、应用天地、经验交流、学习园地、编读往来
- 杂志社网址: <http://www.microcontroller.com.cn>,
<http://www.dpj.com.cn>

诚挚欢迎业界人士向本刊投稿, 欢迎广大读者订阅本刊。



杂志社联系方式

通信或汇款地址: 北京市海淀区学院路 37 号《单片机与嵌入式系统应用》杂志社

邮编: 100083

电话: (010)82317029, 82313656

传真: 010-82317043

E-mail: mcu@publica.bj.cninfo.net mcupress@263.net.cn (投稿专用)

广告业务联系人: 王金萍 魏洪亮

开户行: 北京市商业银行学院路支行

户名: 《单片机与嵌入式系统应用》杂志社

账号: 010903391001201110299-36

杂志订阅方式

国内统一刊号: CN 11-4530/V

国际标准刊号: ISSN 1009-623X

每期定价: 8 元, 全年定价: 96 元, 每月 1 日出版。

第一种订阅方式: 通过邮局订阅, 邮发代号: 2-765。

第二种订阅方式: 直接向《单片机与嵌入式系统应用》杂志社订阅, 另加 15% 邮资。

向北航出版社直接邮购图书地址: 北京航空航天大学出版社邮购组(100083) 另加 2 元挂号费。

邮购 E-mail: bhpress@263.net 电话: 010-82316936, 传真: 010-82317031。

单片机与嵌入式系统图书详细介绍请查阅出版社网站: <http://www.buaapress.com.cn>

进入出版社网站主页后, 点击“单片机与嵌入式系统图书专版”

投稿单片机与嵌入式系统图书请联系: 北航出版社 马广云 电话: 010-82317022 传真: 010-82317022

E-mail: pressb@public3.bta.net.cn

目 录

第 1 章 CCS2 的安装与配置	1
1.1 DSP 芯片的开发工具介绍	1
1.2 CCS2 的主要特性	1
1.3 CCS2 的安装	3
1.3.1 系统需求	3
1.3.2 安装 CCS2	3
1.4 CCS2 的系统配置	4
第 2 章 初识 CCS2	8
2.1 CCS2 的组成	8
2.2 CCS2 代码生成工具	8
2.3 CCS2 集成开发环境功能介绍	10
2.3.1 强大的源代码编辑器	10
2.3.2 方便的应用程序生成特性	11
2.3.3 方便的应用程序调试特性	11
2.4 DSP/BIOS 插件	12
2.4.1 DSP/BIOS 设置	13
2.4.2 DSP/BIOS 的 API 模块	14
2.5 硬件仿真和实时数据交换	15
2.5.1 硬件仿真	15
2.5.2 实时数据交换	15
第 3 章 深入 CCS 集成开发环境	18
3.1 CCS 集成开发环境的特性	18
3.2 菜单栏	19
3.2.1 File 菜单	19
3.2.2 Edit 菜单	21
3.2.3 View 菜单	23
3.2.4 Project 菜单	25
3.2.5 Debug 菜单	26
3.2.6 Profiler 菜单	27
3.2.7 Option 菜单	28
3.2.8 GEL 菜单	31
3.2.9 Tool 菜单	31
3.2.10 DSP/BIOS 菜单	34

3.3	工具栏.....	34
3.3.1	标准工具栏.....	34
3.3.2	工程工具栏.....	35
3.3.3	调试工具栏.....	35
3.3.4	编辑工具栏.....	36
3.3.5	剖析工具栏.....	36
3.4	工程管理.....	37
3.4.1	建立、打开和关闭工程	37
3.4.2	向工程中添加或删除文件.....	38
3.4.3	查看文件关联性.....	39
3.4.4	工程文件剖析.....	40
3.5	源文件管理.....	41
3.5.1	创建新的源文件.....	41
3.5.2	打开文件.....	41
3.5.3	保存文件.....	41
3.6	文件编辑.....	42
3.6.1	设置编辑属性.....	42
3.6.2	拷贝、剪切和粘贴文本	43
3.6.3	编辑整列.....	43
3.6.4	跳到指定行.....	43
3.6.5	查找和替换文本.....	45
3.6.6	利用书签.....	46
第 4 章	CCS 的 C 语言调试实例	48
4.1	创建一个新工程.....	48
4.2	向工程中添加文件.....	49
4.3	代码浏览.....	50
4.4	编译、链接和运行程序	52
4.5	改变程序设置并查找语法错误.....	54
4.6	使用断点和观察窗口.....	55
4.7	观察结构体的值.....	56
4.8	测试代码执行统计.....	57
第 5 章	CCS 的汇编语言调试实例	59
5.1	载入可执行程序.....	61
5.2	使用反汇编工具.....	61
5.3	利用断点调试程序.....	62
5.3.1	断点的设置和取消.....	62
5.3.2	程序的执行.....	63

5.3.3 流水线冲突的解决	64
5.3.4 查看 CPU 寄存器的值	66
5.3.5 查看内存数据	72
5.3.6 查看变量的值	75
5.4 剖析点的调试	76
第 6 章 CCS2 高级使用——使用文件 I/O	80
6.1 探测点与文件 I/O	80
6.2 利用探测点观察寄存器的值	80
6.2.1 探测点的设置与删除	80
6.2.2 观察寄存器的值	81
6.3 利用文件 I/O	82
6.3.1 I/O 文件格式	82
6.3.2 打开工程文件	83
6.3.3 阅读源代码	84
6.3.4 设置 PC 数据文件与探测点关联	86
6.3.5 设置图形显示窗口	89
6.3.6 程序的动画执行	90
6.3.7 增益的调节	92
6.3.8 查看变量属性和值	93
6.3.9 从文件读入数据到内存	95
第 7 章 CCS2 高级使用——DSP/BIOS 原理与应用	98
7.1 DSP/BIOS 介绍	98
7.2 DSP/BIOS 组件	99
7.2.1 实时库与 API 函数	100
7.2.2 DSP/BIOS 配置工具	100
7.2.3 DSP/BIOS 插件	102
7.3 DSP/BIOS 命名规则	103
7.3.1 头文件命名	103
7.3.2 对象命名	103
7.3.3 函数命名	103
7.3.4 数据类型名	104
7.3.5 存储器段命名	104
7.4 DSP/BIOS 程序生成过程	105
7.4.1 使用配置工具	105
7.4.2 创建 DSP/BIOS 程序所使用的文件	108
7.4.3 编译和链接 DSP/BIOS 应用程序	109
7.4.4 DSP/BIOS 应用程序执行顺序	112

7.5 DSP/BIOS 仪表	124
7.5.1 实时分析	124
7.5.2 软件仪表与硬件仪表	125
7.5.3 仪表性能	125
7.5.4 仪表 API	126
7.6 创建一个 DSP/BIOS 程序	126
7.6.1 打开存在的工程	126
7.6.2 剖析 stdio.h 的执行时间	128
7.6.3 使用 HELLO2 文件	129
7.6.4 创建一个配置文件	130
7.6.5 添加 DSP/BIOS 文件到工程	131
7.6.6 用 CCS2 测试	132
7.6.7 剖析 DSP/BIOS 代码执行时间	133
7.7 调试 DSP/BIOS 程序	134
7.7.1 打开并检查工程	134
7.7.2 观察源代码	135
7.7.3 修改配置文件	137
7.7.4 使用执行图观察线程执行	139
7.7.5 更改和观察 Load 函数的执行	141
7.7.6 分析线程统计	142
7.7.7 添加显式 STS 仪表	143
7.7.8 观察显式仪表	145
第 8 章 CCS2 高级使用——RTDX 的原理与应用	148
8.1 RTDX 介绍	148
8.2 使用 RTDX 插件	149
8.2.1 RTDX 配置控制窗口	149
8.2.2 RTDX 通道控制窗口	150
8.2.3 RTDX 诊断控制窗口	151
8.3 配置 RTDX	152
8.3.1 修改 DSP 目标缓冲区大小	152
8.3.2 修改主机缓冲区大小	152
8.3.3 RTDX 工作模式	152
8.3.4 RTDX 目标中断屏蔽	153
8.4 实时通信程序的设计方法	153
8.4.1 编写目标 DSP 应用程序	154
8.4.2 编写 RTDX OLE 自动化客户程序	155
8.4.3 在 CCS 中使能 RTDX	156
8.4.4 运行 OLE 自动化客户程序	156

8.5	RTDX 实例一	156
8.5.1	从 DSP 目标系统传输一个整数到 PC 主机	156
8.5.2	从 PC 主机传输一个整数到 DSP 目标系统	159
8.6	RTDX 实例二	161
8.6.1	从 DSP 目标系统传输一个整数到 PC 主机	161
8.6.2	从 PC 主机传输一个整数到 DSP 目标系统	161
第 9 章	CCS2 高级使用——使用 GEL	166
9.1	GEL 语言简介	166
9.2	GEL 语法	166
9.3	GEL 函数	169
9.3.1	GEL 函数定义	169
9.3.2	GEL 函数参数	169
9.3.3	调用 GEL 函数	171
9.3.4	加载/卸载 GEL 函数	171
9.4	用关键词将 GEL 函数添加到菜单中	171
9.5	输出窗口函数	174
9.6	在 CCS2 启动时自动执行 GEL 函数	175
9.7	嵌入式 GEL 函数	176
9.8	GEL 使用实例	183
9.8.1	一个简单的 GEL 函数	183
9.8.2	定义局部变量	183
9.8.3	GEL 函数的自动执行	184
9.8.4	用 GEL 控制 DSP 变量	186
附 录	指令详解	188
参考文献		279

第 1 章 CCS2 的安装与配置

1.1 DSP 芯片的开发工具介绍

DSP 芯片的开发需要一套完整的软、硬件开发工具。DSP 芯片的开发工具可以分为代码生成工具和代码调试工具两类。

代码生成工具的作用是将 C 语言、汇编语言或两者的混合语言编写的 DSP 源代码程序编译、汇编并链接成可执行的 DSP 代码。C 编译器、汇编器和链接器是 DSP 代码生成工具所必须的。

- C 编译器(C compiler)将 C 语言源代码程序自动地编译成 C54X 的汇编语言源代码程序。
- 汇编器(assembler)将汇编语言源代码文件汇编成机器语言 COFF 目标文件,在源文件中包含了汇编指令、宏命令及指令等。
- 链接器(linker)把汇编生成的可重定位的 COFF 目标模块组合成一个可执行的 COFF 目标模块。它能调整并解决外部符号参考。链接器的输入是可重定位的 COFF 目标文件和目标库文件,它也可以接受来自文档管理器中的目标文件以及链接以前运行时所产生的输出模块。

代码调试工具的作用是对 DSP 程序及目标系统进行调试,使之能够达到设计目标。TMS320 系列 DSP 芯片的系统集成和调试工具主要有:C 语言源代码调试器、汇编语言源代码调试器、初学者工具 DSK(Designer's Starter Kit)、软件仿真器(simulator)、评估模块 EVM(Evaluation Module)、仿真器 XDS(eXtended Developing System)和软件开发系统 SWDS(Software Developing System)等。

1.2 CCS2 的主要特性

CCS2(Code Composer Studio Version 2.0)代码调试器是一种针对标准 TMS320 调试接口的集成开发环境 IDE(Integrated Development Environment),由 TI 公司在 1999 年推出。CCS 目前有 CCS1.1、CCS1.2、CCS2.0 和 CCS2.2 等几个版本,有 CCS2000(针对 C2XX)、CCS5000(针对 C54XX、C55XX)和 CCS6000(针对 C6X)等几个不同的型号。各个不同的版本和型号之间的差别并不大。在 TI 公司的官方网站(<http://www.ti.com>)上,可以下载免费使用期限为 30 天的试用版本。

CCS2 包含源代码编辑工具、代码调试工具、可执行代码生成工具和实时分析工具,并支持设计和开发的整个流程,如图 1.1 所示。

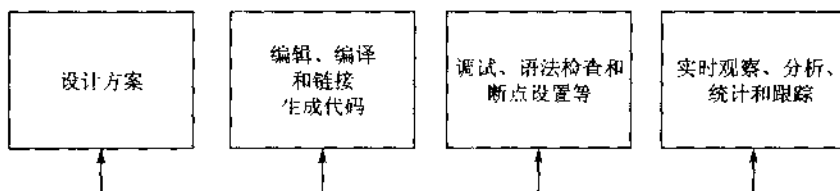


图 1.1 CCS2 的开发流程

CCS2 除了包含 HLL 调试器的主要特性外,还包括下列特性:

- 完全集成的开发环境 CCS2 将 TI 公司的汇编器、编译器、链接器和调试器等都集成到它的开发环境中,用户可以从菜单栏中选择 TI 公司的各种工具,并且可以直接观察到流水线输出到窗口的编译结果。同时,出错信息加亮显示,只要双击出错信息便可以打开源文件,光标停留在出错的地方。在 Windows 环境中,用户可以很方便地同时编辑、调试和编译源程序。代码编译器可以跟踪一个项目中的所有文件及其相关内容。用户可以选择编译单个的文件,或者将所有文件包括在一个项目中,或者逐步建立项目。在编辑器、编译器、链接器和调试器选项中有方便的对话框。
- 高度集成的源代码编辑器 能动态提示 C 语言和 DSP 汇编语言源代码,能很容易地阅读和理解源代码,及时发现和定位语法错误。CCS2 是一个完全集成的包含 TI 编译器的开发环境。CCS2 目标管理系统、内建编辑器和所有的调试分析能力都集成在 Windows 环境中。
- 支持编辑和调试的后台编辑 用户在编译和调试程序时,不必退出系统而回到 DOS 系统中,CCS2 会自动将这些工具交互式地装载到它的环境中。在代码调试窗口中,只要双击错误,就可以直接显示源代码的出错处。
- 对 C 语言源文件和 DSP 汇编语言文件的目标管理 编辑器能跟踪所有文件及其相关内容。这样,编译器只对最近一次编译中改变过的文件进行编译,节省了编译时间。CCS2 在 Windows 98 和 Windows 2000 中支持多线程处理,并行管理调试器(PDM),允许将命令传播给所有的或所选择的处理器。
- 文件探针在算法中通过文件提取或加入信号和数据 CCS2 允许用户从 PC 机中的文件直接读取或写入信号流,而不是实时地读取输入信号。这样,用户可以使用已有的文件来仿真算法。
- 可以在后台执行 DOS 程序 可以执行后台 DOS 命令,并将其输出通过流水线的方式输出,允许用户将应用集成到 CCS2。
- 图形分析功能 具有强大的图形分析功能。
- 方便的代数分解窗口 可以选择查看 C 语言格式的代数表达式,以便容易读懂操作码。
- 有在任何算法点观察信号的图形窗口探针 图形显示窗口使用户能够观察时域或者频域内的信号。对于频域图,FFT 在 PC 机上执行,以便观察所感兴趣的部分而不需要改变它的 DSP 代码。图形显示也可以同探针相连接,当前显示窗口被更新时,探针被确定;当代码执行到这一点时,可以迅速地观察到信号。
- 有状态观察窗口 CCS2 的可视窗口允许用户直接进入 C 表达式及相关变量,结构、数

组和指针等都能很简单地增加和减少,以便进入复杂结构。

1.3 CCS2 的安装

1.3.1 系统需求

下面给出了 CCS2 对 PC 机的最低配置要求。

软件需求: Microsoft Windows 95、Microsoft Windows 98、Microsoft Windows 2000 或 Windows NT4.0。

硬件需求:至少 32 MB 的 RAM、100 MB 的剩余磁盘空间、Pentium100 以上的处理器和 SVGA 显示器(分辨率 800×600 以上)。

1.3.2 安装 CCS2

将 CCS2 安装光盘插入 CD-ROM 驱动器中,此时启动光盘自动运行程序,提示用户是否要安装 CCS2。也可以运行光盘根目录下的 setup.exe,按照安装提示,一步步完成安装(系统的默认安装目录是 c:\ti)。同时,建议安装 Acrobat,这是为方便用户阅读 CCS2 自带的帮助文档。安装成功后,安装程序将自动在桌面上创建 CCS2studio 和 Setup CCS2studio 2 个快捷图标。

成功地安装完 CCS2 后,将在安装文件夹中产生下列目录和文件:

- C5400\bios T1 代码产生工具目录;
- C5400\examples 源代码例子目录;
- C5400\rtidx RTDX(实时数据交换)文件目录;
- C5400\tutorial CCS2 使用帮助目录;
- CC\bin CCS2 环境配置目录;
- CC\gel CCS2 中使用的 GEL(通用扩展语言)目录;
- Bin 应用程序目录;
- Docs CCS2 的相关文档和手册;
- Myprojects CCS2 存放用户设计工作目录。

在 CCS2 中,常用的文件类型如下:

- .mak CCS2 使用的工程文件;
- .c C 语言程序源代码文件;
- .asm DSP 汇编语言程序源代码文件;
- .h C 语言程序和 DSP/BIOS API 中的头文件;
- .lib 库文件;
- .cmd 链接命令文件;
- .obj 汇编或编译后产生的目标文件;
- .out 完成汇编、链接和编译所产生的可执行程序文件;
- .wks 保存环境设置的工作文件;
- .cdb 调用 DSP/BIOS API 所必须的配置数据库文件。

如果使用了 DSP/BIOS, 当保存配置文件的同时, 还将产生下列类型的文件:

- *cfg.cmd 由配置工具所产生的链接命令文件, 在链接生成可执行文件时使用。此文件定义了 DSP/BIOS 应用程序的链接选项、目标名和 DSP 程序的通用数据块, 如 .txt、.bss 和 .data 等。注意这样的文件名在扩展名前面有 cfg 三个字母。
- *cfg.h54 配置工具所产生的头文件。此头文件由 *cfg.s54 所包含。
- *cfg.s54 配置工具所产生的汇编源代码。
- *cfg.o54 配置工具产生的源代码文件所生成的目标文件。

CCS2 安装完成后, 系统将自动设置以下环境变量:

- C54X_A_DIR 汇编器查找 DSP/BIOS 和 RTDX 及代码生成工具所需要的库文件和头文件的搜索路径。
- C54X_C_DIR 编译器和链接器查找 DSP/BIOS 和 RTDX 及代码生成工具所需要的库文件和头文件的搜索路径。
- PATH CCS2 安装完成后, 在安装目录中将 \cgtools\bin 和 \bin 添加到路径中。比如, CCS2 安装到 D:\programs, 则把 D:\programs\c5400\cgtools\bin 和 D:\programs\c5400\bin 添加到路径中。

如果使用的是 Windows 95 操作系统, 则应该增加 DOS 运行环境的内存容量, 以支持运行 CCS2 所需要的环境变量。把下面一行程序添加到系统文件 config.sys 中, 重新启动计算机。

```
shell=c:\windows\command.com /e,4096 /p
```

1.4 CCS2 的系统配置

在第一次运行 CCS2 软件之前, 需要运行 CCS2 软件设置程序。CCS2 软件设置程序是建立 CCS2 集成开发环境与 DSP 目标系统或者 Simulator 之间的通信接口。CCS2 软件集成了 TI 公司的 Simulator 和 Emulator 的驱动程序, 用户可以直接使用 TI 的仿真器进行开发和调试。如果使用的仿真器不是 TI 公司的, 则需要安装相应的仿真器的驱动程序。

每个标准的配置或用户自定义的配置都放在一个特殊格式的文件中。当前系统的配置记录在系统的注册表中, 这个配置可以被 CCS2 的配置工具修改更新。当 CCS2 的配置工具建立了一个新的配置时, 这个新的配置就记录到注册表中, 取代了原先的配置。如果在工程中使用几种仿真工具时, 就需要给每个仿真工具做相应的配置, 并且在使用时需要在这几个配置中来回切换。

双击桌面上的 Setup CCS2studio 图标, 或者在开始菜单中运行 Setup Code Composer Studio 程序, 可以激活 CCS2 的配置工具, 配置界面如图 1.2 所示。

配置界面分为三部分:

图 1.2 左边一栏中的 My System 表示系统配置; C55X Simulator/C54X Simulator 表示当前的仿真器目标板; CPU 表示相应的处理器。这个界面中列举了已经安装的可供选择的配置。改变当前的工作配置时, 需要保存改变的配置才能生效。中间一栏表示已经安装的仿真器类型。图中是已经安装的 CCS2 自带的 TI 公司的 C54X 和 C55X 系列仿真器。右边一栏是配置命令/信息选项, 单击 Import a Configuration File, 便出现仿真器导入配置对话框。对话

框中显示的是已经安装了驱动程序的可以导入的仿真器类型,如图 1.3 所示。如果不改变当前的配置,单击 Close 按钮。

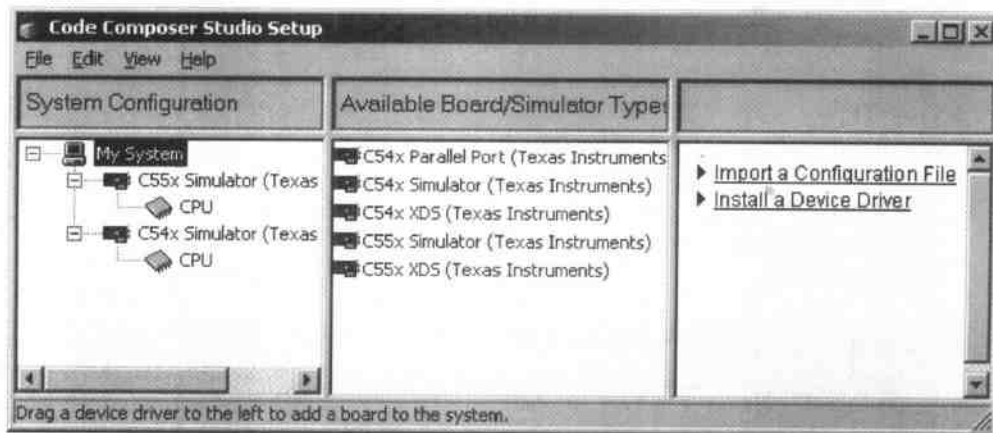


图 1.2 CCS2 配置界面

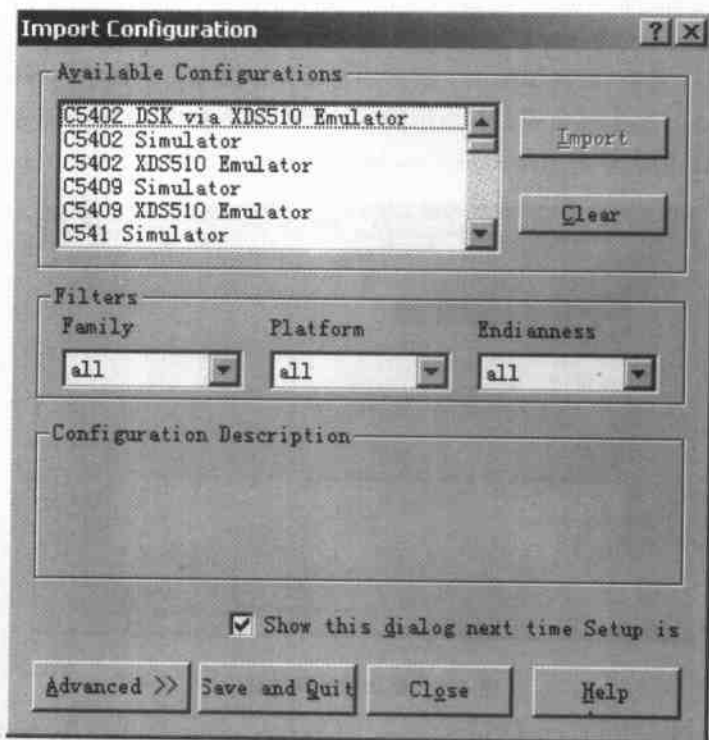


图 1.3 导入配置对话框

单击 Install a Device Driver 选项,便出现驱动程序配置对话框,如图 1.4 所示。

下面举例说明 CCS2 的配置步骤,假设已经将驱动程序安装在 D:\programs\CCS2 目录下。

(1) 双击桌面上的 Setup CCS2studio 图标,运行 CCS2 的配置程序,出现如图 1.2 所示的

配置界面。

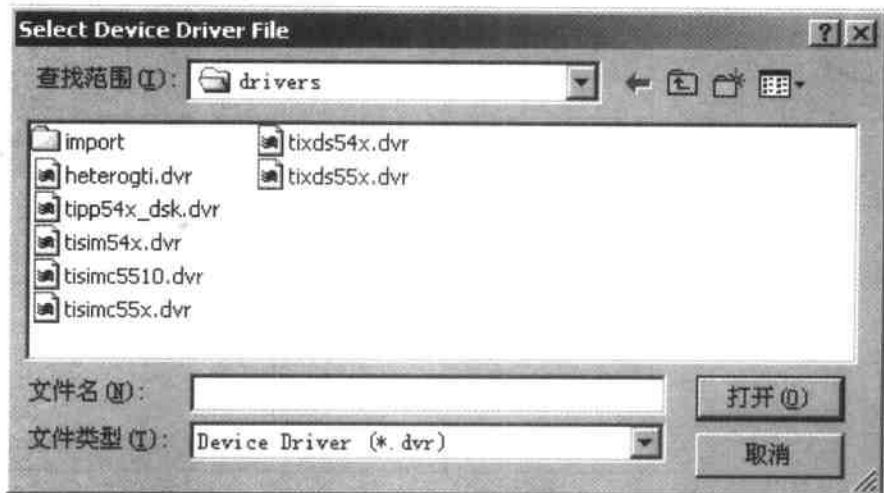


图 1.4 驱动程序配置对话框

(2) 选择 Edit→Install Driver 项,或者在如图 1.2 所示的配置界面的右边一栏单击 Install a Device Driver 选项,出现如图 1.4 所示的驱动程序配置对话框。选择需要的驱动程序,这里选 heterogti.dvr,会出现如图 1.5 所示的驱动程序属性窗口,在此可以为驱动程序重新命名。该驱动程序默认的名字是 Heterogeneous Multi-Target (Texas Instruments),单击 OK。

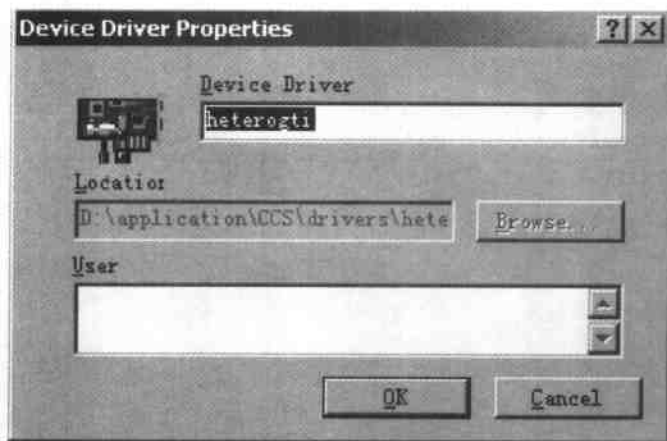


图 1.5 驱动程序属性窗口

(3) 执行菜单命令 Edit→Add to System, Heterogeneous Multi-Target (Texas Instruments)便出现在 CCS2 配置界面的中间一栏,如图 1.6 所示(与图 1.2 对照,中间一栏多了 Heterogeneous Multi-Target (Texas Instruments)一项)。

如果是配置 Simulator,在这一步就可以使用 CCS2 的 Simulator 功能。如果要配置 Emulator,则还需要进行步骤(4)~(6)的配置。

(4) 双击 Heterogeneous Multi-Target (Texas Instruments),出现如图 1.7 所示的属性设置对话框,单击 Board Properties 标签,设置 I/O 端口,这里为 0x240。

(5) 单击图 1.7 中的 Processor Configuration 标签,单击 Add Single 按钮将 CPU-1 加入配置,单击 Finish 按钮结束。

(6) 执行菜单命令 File→Save,保存设置,使配置生效。

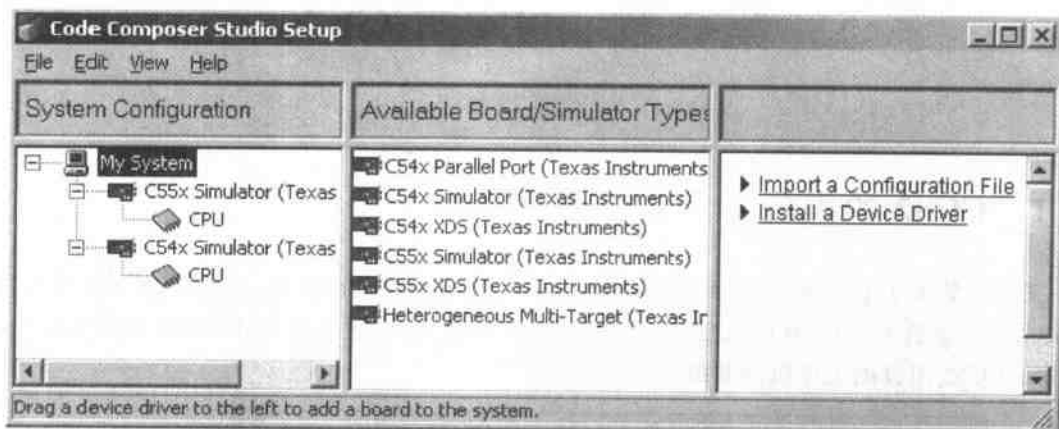


图 1.6 配置后的界面

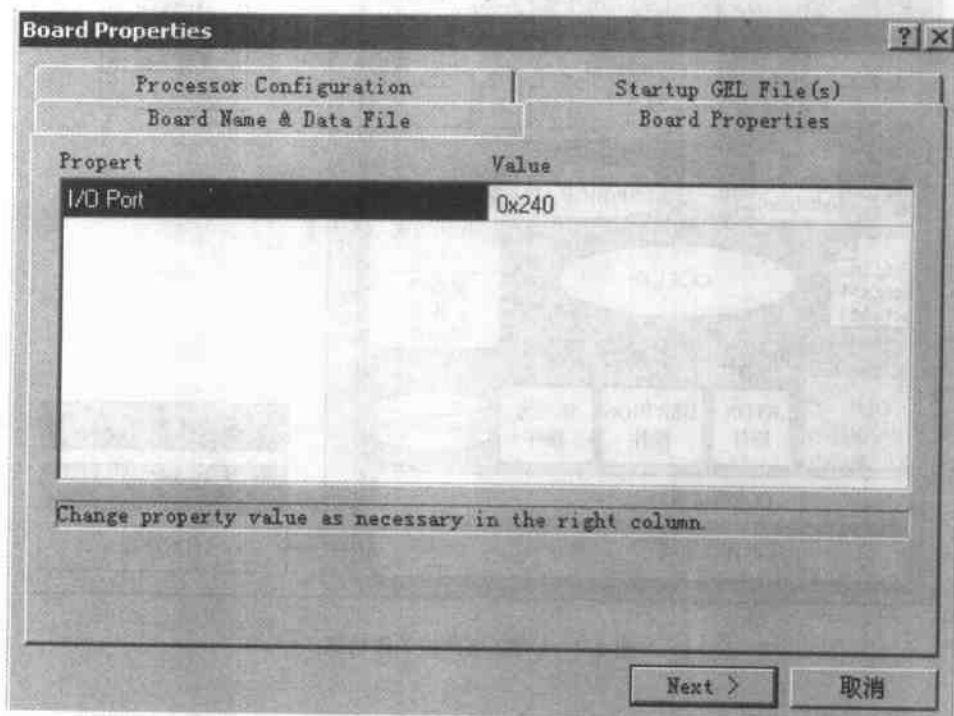


图 1.7 属性设置对话框

第 2 章 初识 CCS2

2.1 CCS2 的组成

CCS2 集成了下列组件：TMS320 系列 DSP 可执行代码生成工具、源代码集成开发环境、DSP/BIOS 插件和 API、RTDX 插件、主机接口和 API。所有这些组件在 CCS2 开发包中协调工作。CCS2 组件的工作机理如图 2.1 所示。

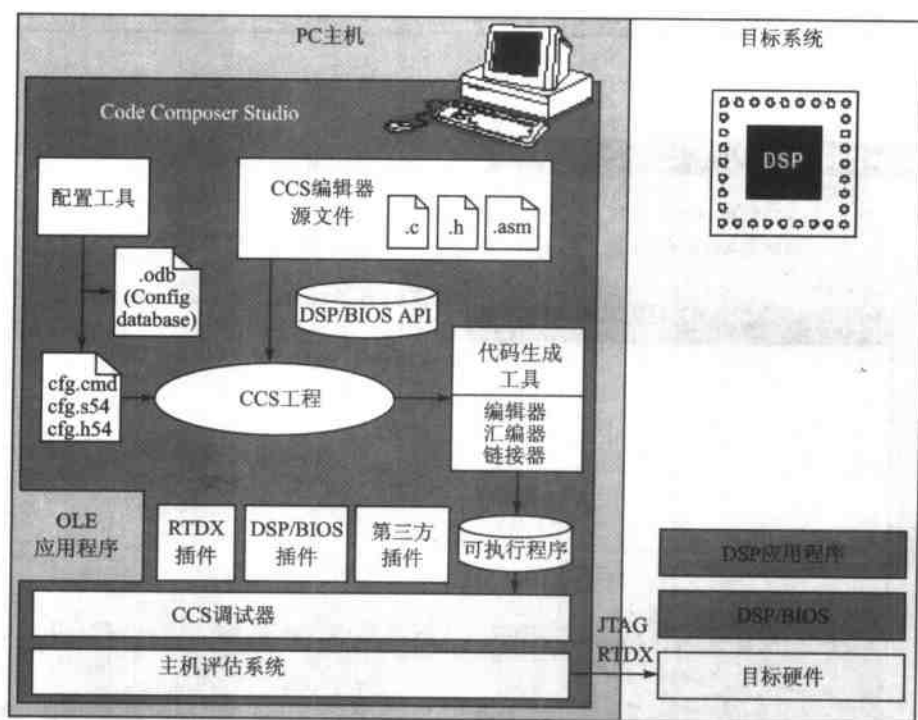


图 2.1 CCS2 组件的工作机理

2.2 CCS2 代码生成工具

代码生成工具是 CCS2 集成开发环境的基础，它的作用是将 C 语言、汇编语言或两者的混合语言编写的 DSP 源代码程序编译、汇编并链接成可执行的 DSP 代码。代码生成工具的工作流程如图 2.2 所示，其中 C 语言程序的最常用的开发路径用阴影显示，其他部分是加强开发过程中的附加功能。

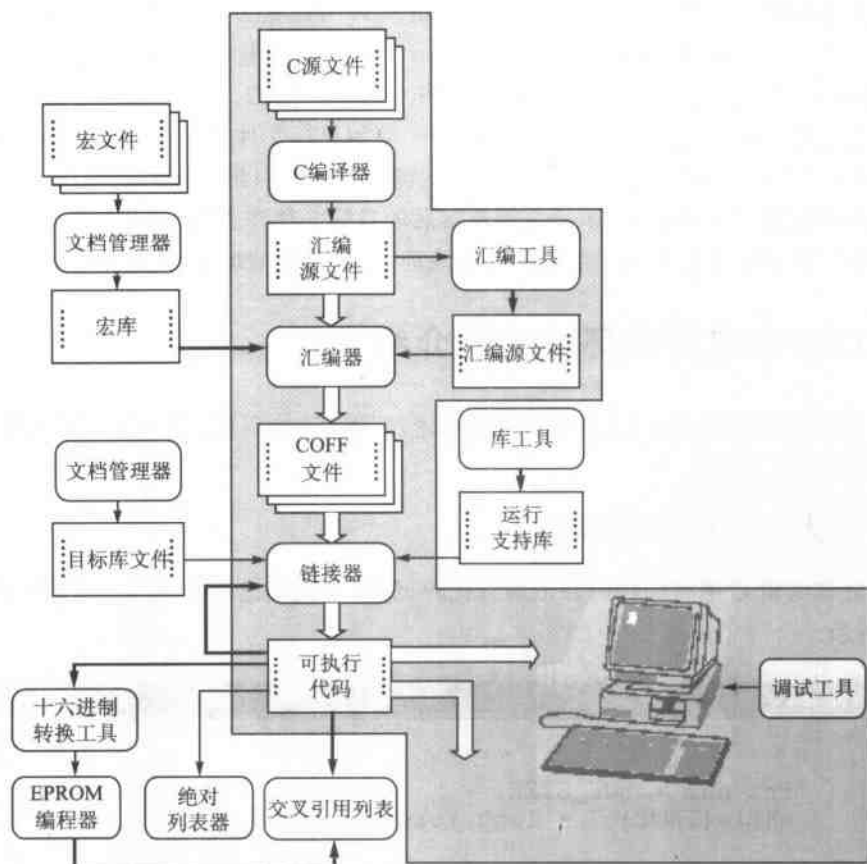


图 2.2 代码生成工具的工作流程

代码生成工具各部分的功能介绍如下：

- C 编译器(C Compiler) 将 C 语言源代码程序自动的编译成 C54X 的汇编语言源代码程序。
- 汇编器(Assembler) 将汇编语言源代码文件汇编成机器语言 COFF 目标文件,在源文件中包含了汇编指令、宏命令及指令等。
- 链接器(Linker) 把汇编生成的可重定位的 COFF 目标模块组合成一个可执行的 COFF 目标模块。它能调整并解决外部符号参考。链接器的输入是可重定位的 COFF 目标文件和目标库文件,它也可以接受来自文档管理器中的目标文件以及链接以前运行时所产生的输出模块。
- 文档管理器(Archiver) 也叫文档库,它将一组文件(源代码文件、中间文件或目标文件等)集中成一个文档库来管理,使用户可以很方便地查找、替换、提取和删除文件。
- 运行支持公用程序(Runtime - support Utility) 建立用户自己的用 C 语言编写的运行支持库。在链接时,使用 rts. src 提供的源代码文件或 rts. lib 中的目标代码所提供的标准支持运行库函数。
- 运行支持库(Runtime - support Library) 包含 ANSI 标准运行支持库函数、编译器公用程序函数、C 语言输入\输出函数和数学函数等。

- 十六进制转换程序(Hex Conversion Utility) 可以很方便地将 COFF 目标文件格式转换成 TI-tagged、ASCII-hex 和 Motorola-s 等目标格式,还可以方便地将文件下载到 EPROM 编程器中,并对用户的 EPROM 进行编程。
- 交叉引用列表(Cross-reference Lister) 使用目标文件产生一个交叉引用清单,分别列出符号、符号的定义以及它们在链接的源文件中的引用情况。
- 绝对列表器(Absolute Lister) 将链接后的目标文件作为输入,生成扩展名为 .abs 的文件。对 .abs 文件汇编产生包含绝对地址(不是相对地址)的文件清单。

2.3 CCS2 集成开发环境功能介绍

CCS2 集成开发环境(IDE)支持从编辑、编译、汇编、链接到调试 DSP 应用程序的整个开发过程。它包括以下特征。

2.3.1 强大的源代码编辑器

CCS2 允许编辑 C 语言源代码和汇编源代码,能在 C 代码之后显示与之对应的汇编指令,如图 2.3 所示。

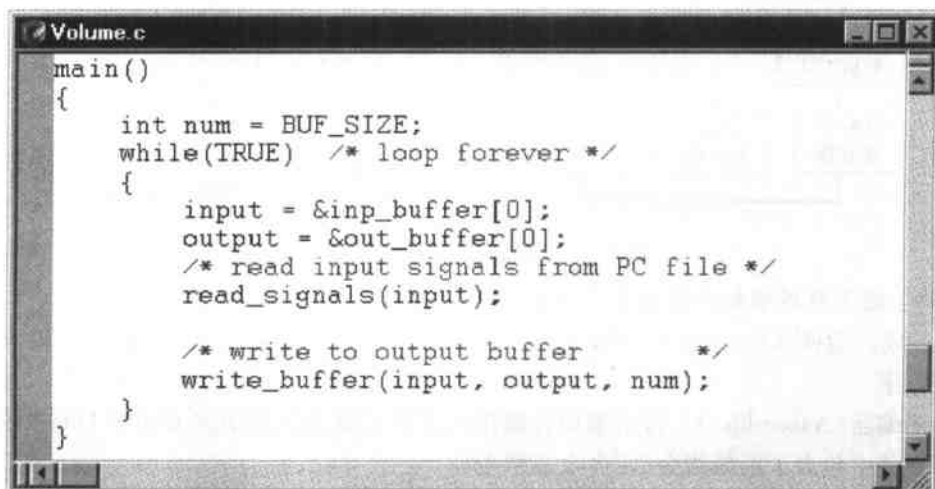


图 2.3 源代码编辑窗口

集成编辑环境提供以下功能:

- 关键词、注释和字符串的高亮度显示;
- 自动缩进和自定义 Tab 键的跳格数;
- 圆括号和花括号对 C 语言代码块做标记,并可方便地查到与之配对的右括号;
- 多次 Undo 和 Redo;
- 随时获得相关的在线帮助;
- 快速查找,在一个或多个文件中查找或替换;
- 自定义键盘命令。

2.3.2 方便的应用程序生成特性

CCS2 使用工程(project)来管理整个应用程序设计的所有文档,如图 2.4 所示。工程中可能包含 C 语言源代码、汇编源代码、库文件、链接命令文件、头文件和目标文件等。CCS2 使用工程文件,与传统的开发工具完全不一样,编译、汇编、链接和调试不再是各个独立的子程序,开发设计人员可以不需要熟悉每个程序的繁琐的命令和相关参数,CCS2 能自动查找工程所需要的相关文件并加入到工程中。当然,如果不想使用 CCS2 的 Build 功能,也可以使用传统的 makefile。

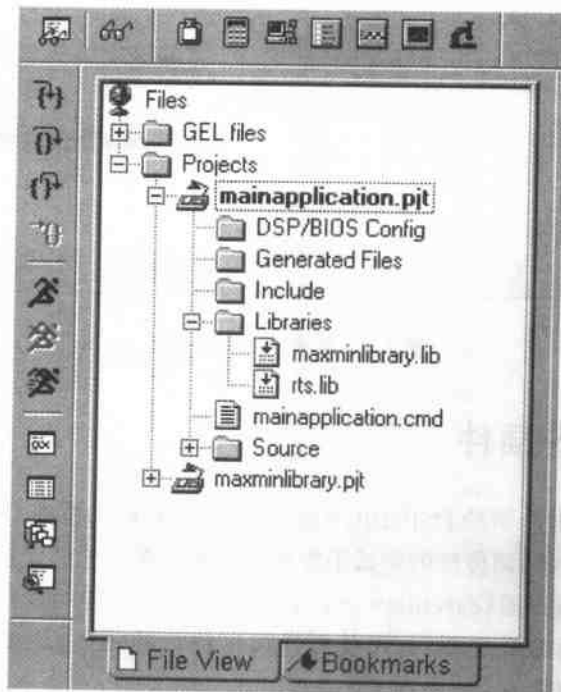


图 2.4 工程管理界面

2.3.3 方便的应用程序调试特性

应用程序调试界面如图 2.5 所示。

CCS2 的调试工具具有以下功能:

- 设置一个或多个断点;
- 在断点处自动更新;
- 使用 Watch 窗口查看变量;
- 查看、编辑存储器和寄存器的值;
- 使用 Probe Point 工具在主机与目标系统间传输数据;
- 观察目标系统中执行的反汇编代码和 C 语言指令;
- 对目标系统中的信号绘图显示;
- 使用 Profile Point 查看执行统计信息;

- 提供 GEL 语言, 允许用户向 CCS2 菜单添加功能。

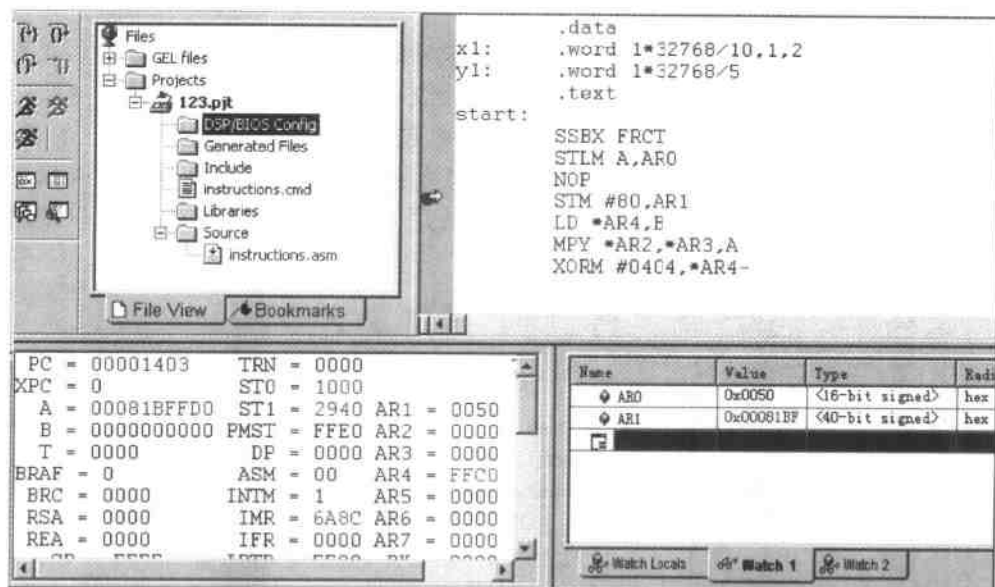


图 2.5 应用程序调试界面

2.4 DSP/BIOS 插件

CCS2 提供支持实时分析的 DSP/BIOS 插件, 能实时跟踪和监视 DSP 的应用程序, 同时对实时性能的影响达到最小, 而传统的调试手段对诊断实时系统中的复杂问题无能为力。

如图 2.6 所示的执行图(execution graph)显示了不同线程的执行顺序。在 CCS2 中, 线程和 Visual C++ 里面的线程不同, 它是指任何可执行的任务, 比如硬件中断服务子程序(ISR)、周期函数和宏替换等。

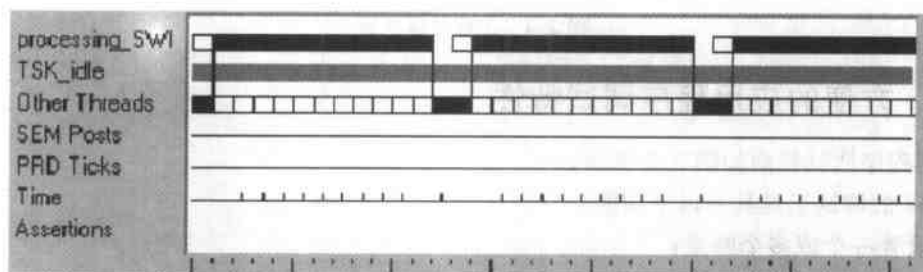


图 2.6 DSP 应用程序执行图

DSP/BIOS 提供下列实时分析特性:

- 程序跟踪 显示写入目标日志的事件, 并在程序执行过程中反映动态控制流程。
- 性能监控 跟踪统计目标资源的使用情况, 如处理器的负载和线程时序等。
- 文件流 将目标系统上的 I/O 对象与主机上的文件关联在一起, 可以利用 PC 机上的文件来输入信号。

2.4.1 DSP/BIOS 设置

在 CCS2 环境中可以创建配置文件,用来定义 DSP/BIOS API 使用的对象。该文件还可以简化存储器映射和硬件中断向量(ISR)的映射。因此,即使不调用 DSP/BIOS API,也可以使用这个配置文件。DSP/BIOS 配置文件有以下两个作用:

- 设置全局运行参数;
- 作为一个可视化编辑器,创建和设置目标程序 DSP/BIOS API 所调用的运行对象属性,这些对象包括硬件中断、软件中断、文件流、I/O 通道和事件日志等。

当打开一个配置文件时,会出现如图 2.7 所示的窗口。

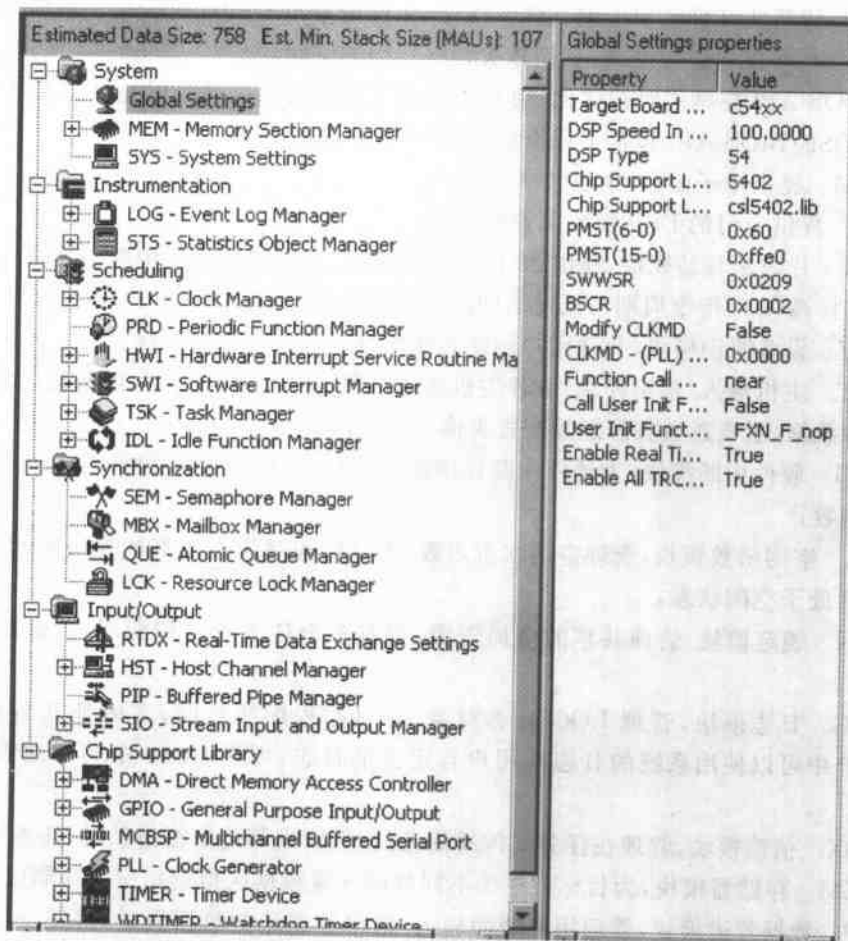


图 2.7 配置文件窗口

存储配置文件将同时产生链接命令文件(*.cfg.cmd)、头文件(*.cfg.h54)和汇编源文件(*.cfg.s54),并将其自动加到工程文件中。

由于配置文件简化了程序的中断矢量和存储器段寻址,因此它既可以在调用 DSP/BIOS API 的程序中使用,也可以在不调用 DSP/BIOS API 的程序中使用。

通用的 API 函数调用是在程序运行时创建对象的,这将导致在目标 DSP 中占用额外的代

码空间。而 DSP/BIOS 对象与通用的 API 函数不同,是静态配置的,并且在执行程序中占用固定的程序代码空间。这种静态配置特性可以在程序执行之前通过验证对象属性的合法性来提前检查错误。

2.4.2 DSP/BIOS 的 API 模块

在调试过程中,传统调试手段的调试是处于执行程序之外的,而 DSP/BIOS 应用程序与传统的调试手段不同,它要求与特定的 DSP BIOS API 模块链接起来。

在配置文件中定义了 DSP/BIOS 对象后,一个程序可以使用一个或多个 DSP/BIOS 模块。在定义了这些对象为外部函数后,在源文件中即可调用 DSP/BIOS 函数。每个模块都有自己独立的 C 语言头文件或宏汇编文件。这样,在使用某个 DSP/BIOS 模块的时候就不会将其他的 DSP/BIOS 模块包含进来,从而减少程序代码的空间。

DSP/BIOS API 在调用时被优化(用 C 语言或汇编语言),这样可以在目标 DSP 上使用最小的资源。DSP/BIOS API 包含下列模块(按字母顺序排列):

- ATM 提供原子函数,生成共享数据;
- C54 提供专门的 DSP 函数来管理中断;
- CLK 片内定时器模块,提供 32 位的实时时钟,其最高速率的中断与片内定时器寄存器一样高(1 个指令周期),低速的中断却可以长达几 ms 甚至更长;
- DEV 设备驱动模块,允许用户创建并使用自己的设备驱动程序;
- HST 主机输入/输出模块,管理主机通道对象,使应用程序能在目标系统和主机之间传输数据,主机通道被静态的配置为输入或输出;
- HWI 硬件中断模块,为硬件中断程序提供支持,在配置文件中可指定发生时要运行的函数;
- IDL 空闲函数模块,管理空闲状态函数,当目标程序没有更高优先级的程序执行时,DSP 处于空闲状态;
- LCK 锁定模块,管理共享的全局资源,当有多个任务竞争资源时,仲裁资源的使用权;
- LOG 日志模块,管理 LOG 日志对象,在目标程序执行时,实时捕获各种事件,在 DSP 中可以使用系统的日志或用户自定义的日志,可在 CCS2 中实时地观察这些日志;
- MBX 信箱模块,管理在任务间传递信息的对象,当等待信箱信息时,任务被阻塞;
- MEM 存储器模块,为目标程序的不同代码区或数据区指定相应的存储区间;
- PIP 数据管道模块,管理用来缓冲输入/输出数据流的数据管道,提供一个一致的软件数据结构,用来驱动 DSP 器件与其他实时外设器件的输入/输出;
- PRD 周期函数模块,管理触发程序中函数周期执行的周期对象,这些对象的执行速率可以由 CLK 模块的时钟速率控制,也可以通过周期性调用 PRD 来控制;
- QUE 队列模块,管理数据队列结构;
- RTDX 实时数据交换模块,使数据能在主机和目标系统之间实时交换,同时在主机端能进行显示和分析;
- SEM 信号量模块,管理任务之间同步和互斥操作的信号量;

- SIO 流模块,管理能提供高效实时与设备无关的输入/输出对象;
- STS 统计模块,管理当程序运行时存储关键统计数据的统计累加器,可在 CCS2 环境中实时观察这些统计数字;
- SWI 软件中断模块,管理软件中断,当一个目标程序使用 API 调用发送一个 SWI 对象时,SWI 模块调度相关函数的执行,软件中断共有 15 个优先级,但所有优先级都比硬件中断优先级低;
- SYS 系统服务模块,提供执行基本系统服务的通用函数,比如中止程序执行和打印格式文本等;
- TRC 跟踪模块,管理一组跟踪控制位,通过事件日志和统计累加器实时捕获程序信息,由于不存在 TRC 对象,因此在配置文件中没有列出 TRC 模块;
- TSK 任务模块,管理比软件中断优先级低的任务线程。

2.5 硬件仿真和实时数据交换

2.5.1 硬件仿真

TI 公司的 DSP 芯片具有片上仿真功能,而 CCS2 能控制程序运行并实时监控程序活动。主机与目标 DSP 系统的通信是通过一个 JTAG 接口来实现的,这种连接方式对 DSP 目标系统的实时性能没有太大的影响。一般的仿真器提供了与主机通信的 JTAG 接口,评估板则提供板上的 JTAG 仿真接口。

片上硬件仿真提供以下功能:

- 运行、停止或复位 DSP 芯片;
- 将代码和数据加载到 DSP 芯片;
- 检查 DSP 芯片中的寄存器和存储器;
- 检查硬件指令或者数据相关的断点;
- 提供各种计算功能,包括精确到指令周期的剖析功能(profiling);
- 提供主机和目标 DSP 系统间的实时数据交换。

2.5.2 实时数据交换

CCS2 的实时数据交换(RTDX)功能使主机与 DSP 目标系统之间能进行双向实时通信,如图 2.8 所示。

RTDX 是 CCS2 的一大特点,由目标板和主机两部分组成。其工作机理是:在目标 DSP 系统上运行一个小的 RTDX 软件库;而用户的应用程序在主机中运行,它调用 RTDX 软件库的 API 函数,从而能够在目标 DSP 系统和主机之间接受和发送数据。RTDX 软件库使用 DSP 芯片内部的仿真硬件模块,通过增强的 JTAG 接口与主机通信,数据的传输是实时的,不影响目标 DSP 系统的程序运行。

RTDX 提供的实时和连续的可视环境,使开发者能看到 DSP 应用程序运行的真实过程。它允许开发者在不停止目标应用程序运行的情况下,在主机和 DSP 目标系统之间实时传输数据,同时还可以在主机上利用对象链接和嵌入(OLE)技术观察和分析数据。这样,可以提供

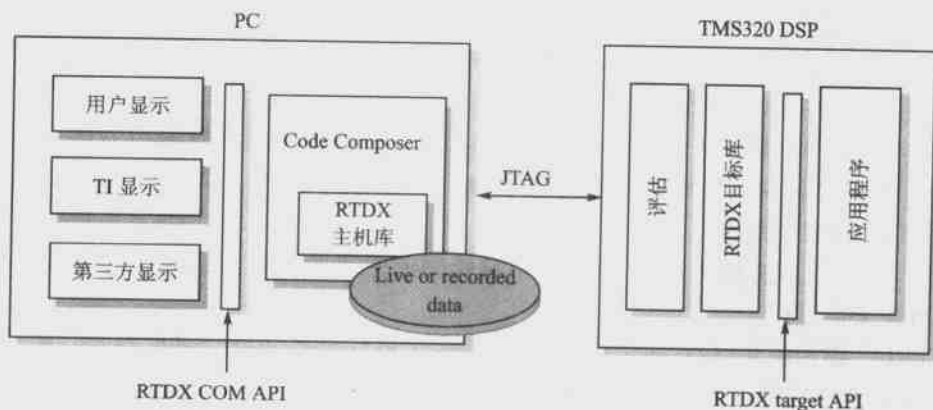


图 2.8 主机和 DSP 目标系统的通信

给开发者一个真实的系统运行过程,缩短开发时间。

主机平台提供一个 RTDX 库配合 CCS2 的工作。主机上的显示和分析工具可以通过一个 COM API 与 RTDX 通信,从 DSP 目标系统获得数据或将数据发送到 DSP 目标系统。开发者可以使用标准的软件显示包显示,如 National Instruments 的 LabView, Quinn - Curtis 的实时图形工具或者 Microsoft Excel 等,也可以使用 VB 或 VC 编写的显示程序。

RTDX 还可以记忆实时数据并回放,以供以后进行非实时分析。如图 2.9 所示为采用 LabView 工具进行的非实时分析的实例。

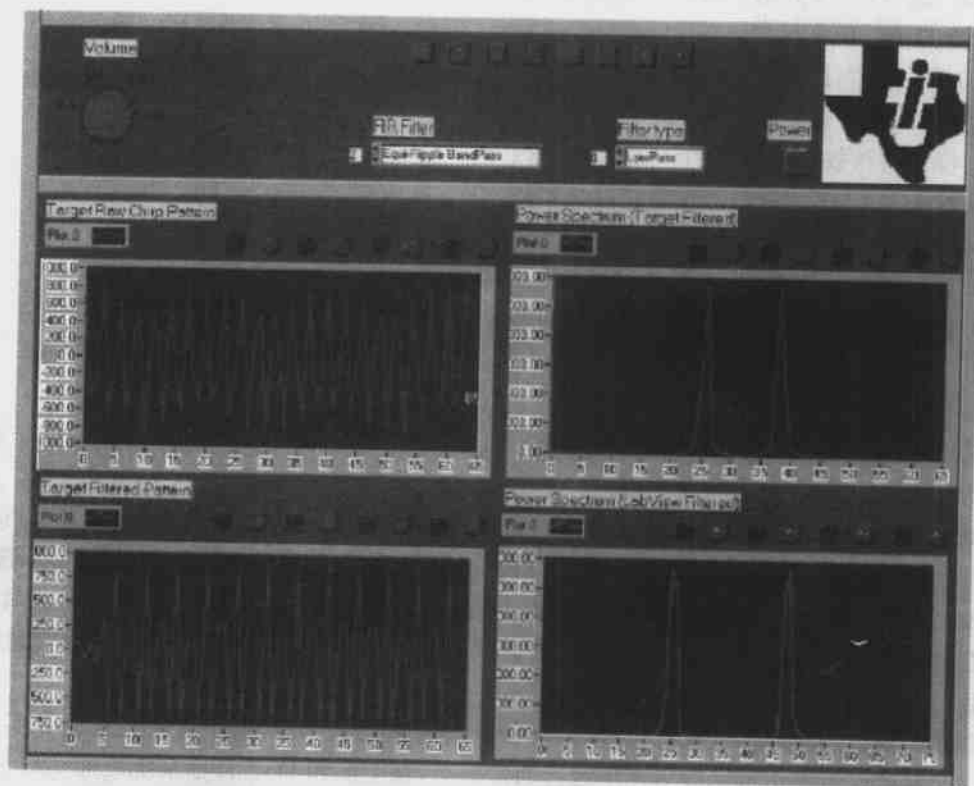


图 2.9 LabView 工具的非实时分析实例

假设在 DSP 目标系统板上存在一个原始信号和一个经 FIR 滤波器滤波后的信号,这两个信号均通过 RTDX 送到主机中。在主机方,LabView 通过 RTDX COM API 获得信号数据,并将这两个信号显示出来,如图 2.9 左半部分所示。为了验证 FIR 滤波器的功能,使用 LabView 软件产生一个功率谱波形,如图 2.9 右半部分所示。滤波信号功率谱显示在右上角,DSP 目标系统上的原始信号经过一个 LabView 软件滤波器进行软件仿真后的功率谱显示在右下角。比较这两个功率谱即可判断 FIR 滤波器算法的正确性。

RTDX 非常适合在控制系统、伺服系统和语音处理中应用。比如,无线电应用开发者可以捕获声码器的输出,检查语音处理算法是否正确;对硬盘驱动器设计者而言,可以在无须向伺服电机加上不正确的信号而损坏硬盘的情况下,测试硬盘产品;发动机控制设计者可以在控制器正常运行时分析温度及其他环境变化带来的影响等。在这些应用中,设计者可以选择可视化工具,显示对调试和分析有用的信息。

未来的 DSP 芯片将增加 RTDX 的带宽,提供对大型程序的可视化控制和管理。

第3章 深入 CCS 集成开发环境

3.1 CCS 集成开发环境的特性

CCS 集成开发环境中包含 Simulator(软件仿真器)和 Emulator(硬件仿真器)两部分。它们使用的是同一个集成开发环境,区别在于,Simulator 可以在不安装 DSP 硬件仿真器的情况下使用户的应用程序在主机上仿真运行,而 Emulator 则必须安装硬件仿真器。因此,对应用系统进行硬件调试或者软硬件联调之前,在没有 DSP 目标板的情况下,设计者可使用 Simulator 模拟 DSP 环境运行用户的代码程序。Simulator 使用主机(一般是 PC 机)上的内存空间模拟 DSP 芯片上的 RAM 和 ROM 等,这对初学者而言,能非常方便地学习 DSP 的指令和编程而不用考虑硬件的设计。

如果主机中只安装了 Simulator 的驱动程序,则启动 CCS 时便可立即运行 Simulator,出现如图 3.1 所示的软件仿真界面;如果主机中同时安装了 Simulator 和 Emulator 的驱动程序,当启动 CCS 时,则先启动如图 3.2 所示的并口调试管理器,此时,需要从菜单中选择 Open→C54XX Simulator 项,以启动 Simulator 的运行(这里假设安装的是 CCS5000),才能出现如图 3.1 所示的界面。

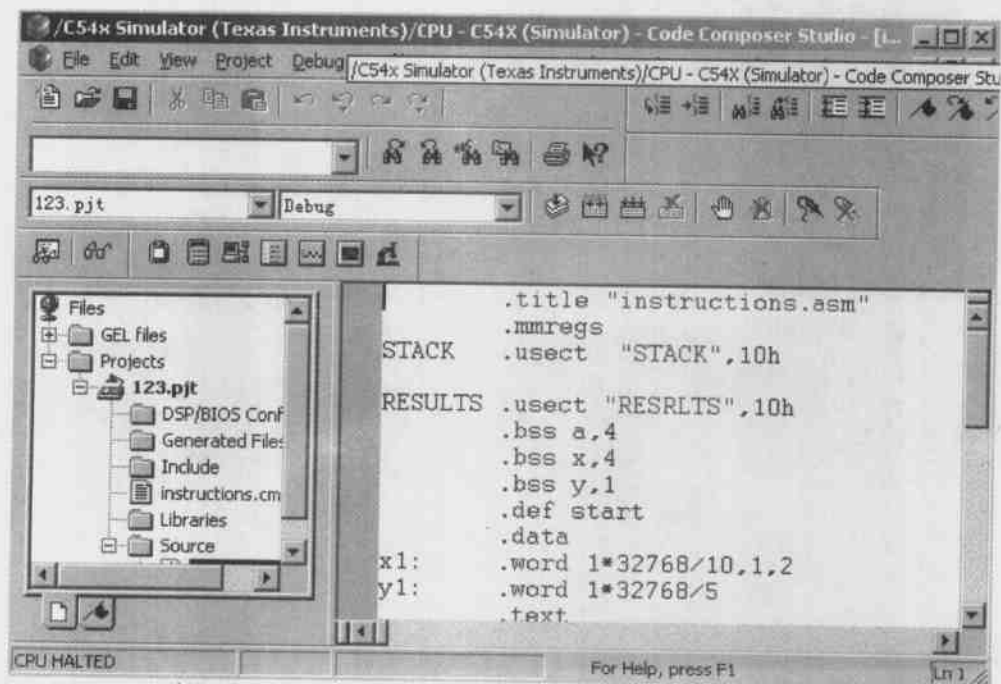


图 3.1 软件仿真界面

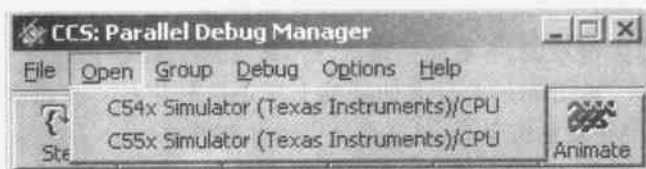


图 3.2 并口调试管理器

在 CCS 集成开发环境中,除 Edit 窗口外,其他所有的窗口和工具栏都是可移动的(docking),可以将这些窗口和工具栏拖到屏幕的任何一个位置。同时,所有的窗口都支持相关菜单(context menu),在窗口内右击,可弹出与内容相关的菜单选项。

3.2 菜单栏

CCS 集成开发环境中共有 12 项菜单,下面只对其中重要的菜单功能进行介绍。

3.2.1 File 菜单

File 菜单提供了与文件操作相关的命令,其中比较重要的操作命令如下:

- (1) New→Source File 建立一个新的源文件,包括扩展名为 *.c、*.asm、*.h、*.cmd、*.gel、*.map 和 *.inc 等文件。
- (2) New→DSP/BIOS Configuration 建立一个新的 DSP/BIOS 配置文件。
- (3) New→Visual Linker Recipe 打开一个 Visual Linker Recipe 向导。
- (4) New→ActiveX Document 在 CCS 中打开一个 ActiveX 类型的文档(如 Microsoft word 或 Bitmap 等),当单击该命令后,将打开如图 3.3 所示的对话框。

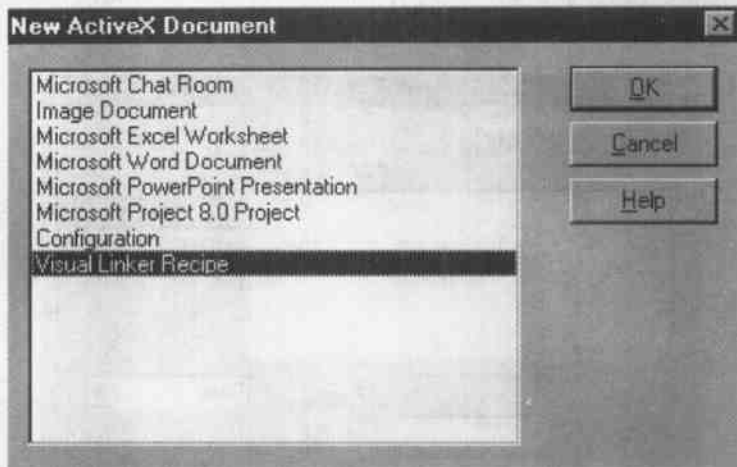


图 3.3 ActiveX 类型文档对话框

(5) Load Program 将 DSP 可执行的目标代码文件 COFF(.out)载入仿真器中(Simulator 或 Emulator)。

(6) Load Symbol 将符号信息载入 DSP 目标系统。这个菜单命令只在 Emulator 硬件

仿真器中使用,当调试器不能或没必要加载目标代码 COFF 文件时(如目标代码存放在 ROM 中),应用该命令只清除符号表,不更改存储器内容和设置程序入口。

(7) Add Symbol 在原来的符号表的基础上添加符号信息。该命令和 Load Symbol 不同,它不清除原来已经存在的符号表,只是在其中添加新的符号。

(8) Reload Program 重新加载 DSP 可执行的目标代码 COFF 文件。如果程序未做更改,则该命令只加载可执行程序代码而不加载符号表。

(9) Load GEL 加载通用扩展语言文件到 CCS 中。在调用 GEL 函数之前,应将包含该函数的文件加入 CCS 中,以将 GEL 函数先调入内存。当加载的文件被修改后,应先卸掉该文件,再重新加载以使修改生效。加载 GEL 函数时将检查文件的语法错误,但不检查变量。



图 3.4 存储器设置对话框

(10) Data→Load 将主机文件中的数据加载到 DSP 目标系统板,可以指定存放的数据长度和地址。数据文件的格式可以是 COFF 格式,也可以是 CCS 所支持的数据格式,缺省文件格式是 .dat 的文件。当打开一个文件时,会出现存储器设置的对话框,如图 3.4 所示。对话框中是指加载主机文件到数据段的从 0x0B00 处开始的长度为 0x0FFF 的存储器中。

(11) Data→Save 将 DSP 目标系统板上存储器中的数据加载到主机上的文件中,该命令和 Data→

Load 是一个相反的过程。

(12) File I/O 允许 CCS 在主机文件和 DSP 目标系统板之间传送数据,一方面可以从 PC 机文件中取出算法文件或样本用于模拟,另一方面也可以将 DSP 目标系统处理后的数据保存在主机文件中。File I/O 功能主要与 Probe Point 配合使用。Probe Point 将告诉调试器何时从主机文件中输入/输出数据。File I/O 功能不支持实时数据交换。实时数据交换应使用 RTDX。当单击该命令时,会出现如图 3.5 所示的对话框。

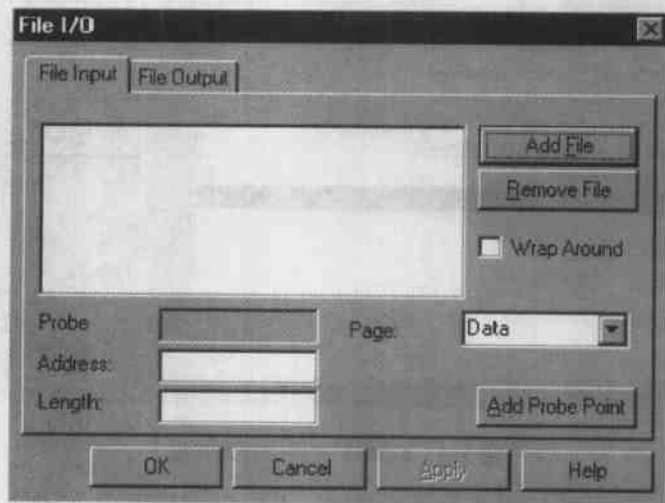


图 3.5 File I/O 对话框

(13) Launch Setup 可以运行 CCS 系统的仿真配置程序,也可以直接在系统的开始菜单中运行。

File 菜单中的其他命令,如 Open、Close、Save、Save As 和 Save All 等,意义比较明显,这里不介绍。

3.2.2 Edit 菜单

Edit 菜单提供的是与编辑相关的命令,除了 Undo、Redo、Cut、Copy 和 Paste 等常用的文件编辑命令外,还有如下一些比较重要的命令:

(1) Find in Files 能够在多个文本文件中查找特定的字符串或表达式。

(2) Go To 能够快速定位并跳转到源文件中的某一指定的行或书签处。

(3) Memory→Edit 编辑存储器的某一存储单元。单击该命令,将打开如图 3.6 所示的对话框。对话框中是指修改地址在数据段 0x0400 处的存储器的值。

(4) Memory→Copy 能将某一存储块(利用起始地址和长度)的数据复制到另一存储块。单击该命令,将出现如图 3.7 所示的对话框。对话框中是指将从 0x0080 开始的长度为 4 的数据段存储器的值拷贝到数据段 0x00FF 处。

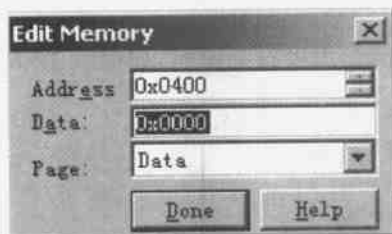


图 3.6 存储器编辑对话框

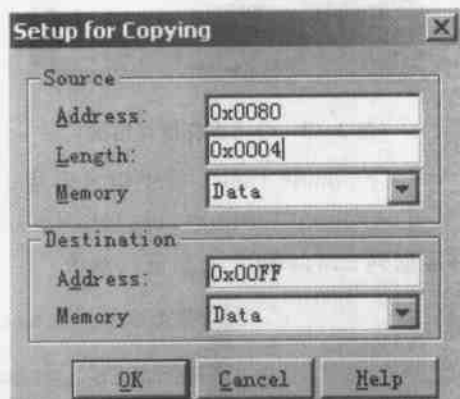


图 3.7 存储块复制对话框

(5) Memory→Fill 将一段存储块全部填入一固定的值。单击此命令,将打开如图 3.8 所示的对话框。对话框中是指将从 0x0080 开始的长度为 5 的数据段存储器中的值全部修改为 0x000A。

(6) Memory→Patch Asm 能在不修改源代码文件的情况下,修改 DSP 可执行代码 COFF 文件的内容。单击此命令,将出现如图 3.9 所示的对话框。对话框中是指将程序段存储器中地址为 0x001234 的指令修改为 LD *AR3, A。

(7) Register 能编辑指定寄存器中的值,包括 CPU 的寄存器和外围寄存器中的值。由于 Simulator 只能进行软件仿真,不支持外围寄存器,故不能在 Simulator 中编辑外设寄存器的内容,但能在 Emulator 中对外设寄存器进行管理。单击此命令,将出现如图 3.10 所示的对话框。对话框中是指将 CPU 寄存器 AR2 中的值修改为 0x0012。

(8) Variable 能修改某一变量的值。对定点的 DSP 芯片而言,如果 DSP 目标系统的内

存有多页,则可以使用@prog、@data 和@io 来分别指定存储器中的程序段、数据段和 I/O。单击此命令,将出现如图 3.11 所示的对话框。对话框中是指将变量 x1 的值修改为 0.1。

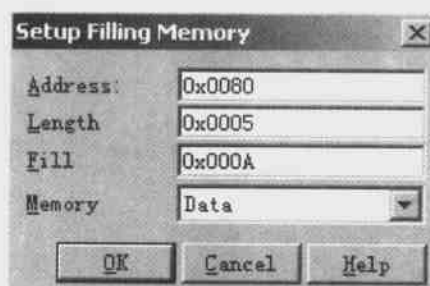


图 3.8 存储块填充对话框

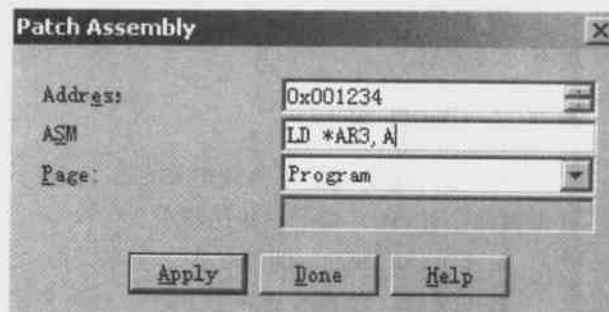


图 3.9 修改可执行代码对话框



图 3.10 编辑寄存器对话框

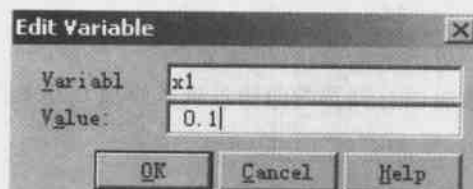


图 3.11 修改变量对话框

(9) Command Line 提供键入表达式或执行 GEL 函数的快捷方法。

(10) Column Editing 可以选择某一矩形区域内的文本进行列编辑,如图 3.12 所示,能够对数据列 2、3、4 和 5 一起进行修改。

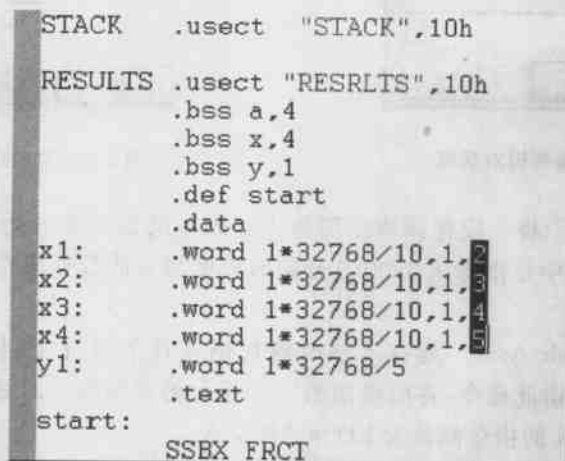


图 3.12 列编辑界面

(11) Bookmarks 可以在源文件中定义一个或多个书签,便于快捷定位。

3.2.3 View 菜单

在 View 菜单中,可以选择是否显示各种工具栏、各种窗口和各种对话框等。其中比较重要的命令介绍如下:

(1) Disassembly 当将 DSP 可执行程序 COFF 文件载入目标系统后,CCS 将自动打开一个反汇编窗口,反汇编窗口根据存储器的内容显示反汇编指令和符号信息,如图 3.13 所示。

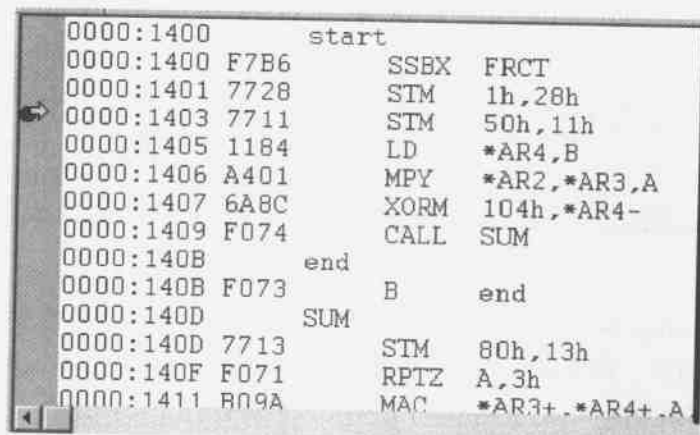


图 3.13 反汇编窗口

(2) Memory 能够显示指定的存储器中的内容。单击此命令,将显示如图 3.14 所示的对话框,在其中的 Address 框里面输入“0x1403”,则 Memory 窗口将显示出地址 0x1403 处的内容为“0x7711”,如图 3.15 所示。可以对照图 3.13 中的一行:

0000:1403 7711 STM 50h,11h

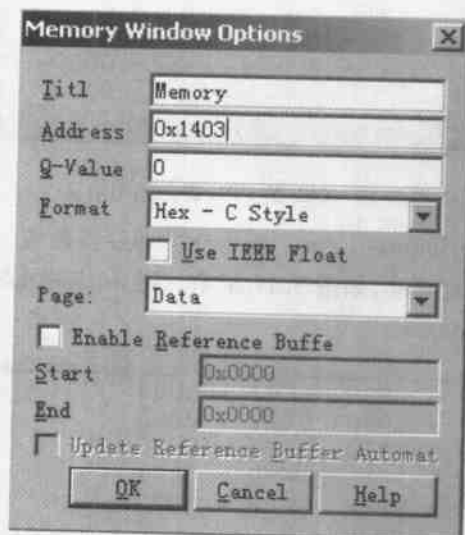


图 3.14 显示指定的存储器内容

0x1403:	0x7711	0x0050	0x1184	0xA401	0x6A8C	0x0104	0xF074
0x140A:	0x140D	0xF073	0x140B	0x7713	0x0080	0xF071	0x0003
0x1411:	0xB09A	0x8008	0xFC00	0xFC00	0xFC00	0x0000	0x0000
0x1418:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x141F:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x1426:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x142D:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x1434:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x143B:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x1442:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x1449:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x1450:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x1457:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x145E:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x1465:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x146C:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000

图 3.15 地址 0x1403 处的内容

(3) CPU Registers→CPU Registers 显示 DSP 的 CPU 寄存器中的值。单击此命令,将显示 CPU 寄存器窗口。该窗口可以任意移动(docking),如图 3.16 所示。图 3.16 中 DSP 的 PC 寄存器的值是 0x00001403。可以对照图 3.13 中的一行:

```
0000:1403 7711      STM    50h,11h
```

这里,DSP 程序运行到 0000:1403,在断点处停止,故当前 PC 的值是 0x00001403。

PC = 00001403	TRN = 0000	INIM = 1
XPC = 0	STO = 1000	IMR = 6A8C
A = 001848C2AC	ST1 = 2940	AR1 = 0050
B = 0000000000	PMST = FFE0	AR2 = 0012
T = 0000	DP = 0000	AR3 = 0000
BRAF = 0	ASM = 00	AR4 = F820
BRC = 0000	TC = 1	AR5 = 0000
RSA = 0000	C = 0	AR6 = 0000
REA = 0000		AR7 = 0000
		SP = 0000
		ARO = 40E4
		OVA = 0
		OVB = 0
		OVM = 0

图 3.16 DSP 的 CPU 寄存器中的值

(4) CPU Registers→Peripheral Regs 显示外设寄存器的值。单击此命令,将显示外设寄存器窗口。该窗口可以任意移动,如图 3.17 所示。图 3.17 中 DSP 的外设寄存器 SWWSR (等待周期寄存器)的值是 0x0001。

```
DRRO = 0000 DRR1 = 0000
DXRO = 0000 DXR1 = 0000
SPC0 = 0800 SPC1 = 0800
TIM = FFFF SWWSR = 0001
PRD = FFFF BSCR = 0002
TCR = 0020
```

图 3.17 外设的寄存器的值

对照图 3.13 中的一行：

0000:1401 7728 STM 1h,28h

这条指令就是：

STM #0001h, SWWSR

它把数据 0001h 送给 SWWSR 寄存器。

(5) Watch Window 检查和编辑 C 语言表达式或变量的值。可以用不同的格式显示数值,可以显示数组、结构或指针等包含多个元素的变量。单击该命令,将显示如图 3.18 所示的观察窗口。窗口中变量 x1 的值,就是 x1 所在的地址 0x00000089。

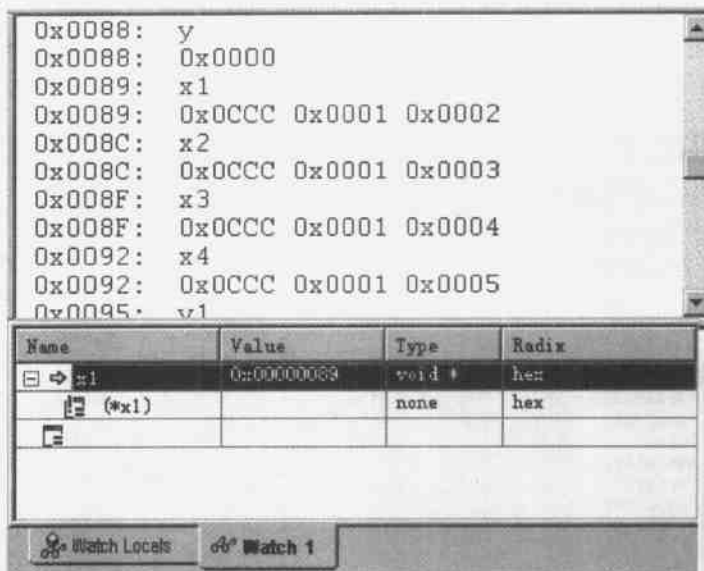


图 3.18 变量观察窗口

(6) Call Stack 检查所调试程序的函数调用情况。该命令只在调试 C 语言程序时有效,而且程序中必须有一个堆栈段和一个主函数;否则将显示“C source is not available”。

(7) Expression List 所有的 GEL 函数和表达式都采用表达式求值程序来估算。求值程序可对多个表达式求值,在求值过程中可选择表达式并单击 Abort 按钮取消求值。该命令在 GEL 函数执行到死循环或执行时间太长时非常有用。

(8) Mixed Source/Asm CCS 能同时显示 C 语言代码及与之关联的反汇编代码(反汇编代码位于 C 语言代码下方)。

3.2.4 Project 菜单

CCS 使用工程(Project)来管理整个设计过程,它不允许直接对 DSP 汇编源代码或 C 语言源代码文件 Build 生成 DSP 可执行代码。只有在建立工程文件的基础上,在菜单或工具栏上运行 Build 命令才会生成可执行代码。工程文件被存盘为 .mak 文件。在 Project 菜单下,除了 New/Open/Close 等常见命令外,其中比较重要的命令介绍如下:

(1) Add Files to Project 将文件加载到工程中。CCS 根据文件的扩展名将文件加到相应的子目录中。工程文件支持 C 语言源文件(*.c or *.c*)、汇编语言源文件(.a* or *.s*)、

库文件(.o* or .lib)、头文件(.h)和链接命令(.cmd)文件。其中,C 语言源文件和汇编语言源文件可以被编译和链接,库文件和链接命令文件只能被链接,CCS 会自动将头文件添加到工程中。

(2) Compile File 编译 C 语言或汇编语言源代码文件。

(3) Build 编译和链接 C 语言或汇编语言源代码文件,对于没有修改的源文件,CCS 将不重新编译。

(4) Rebuild All 对工程中所有文件重新编译,并链接生成 DSP 可执行的 COFF 格式的文件。

(5) Build Options 用来设定编译器、汇编器和链接器的参数。单击此命令,将显示如图 3.19 所示的对话框,可以在其中设置编译器、汇编器和链接器的各种参数。编译器、汇编器和链接器的参数设置非常复杂,一般使用 CCS 默认的即可。

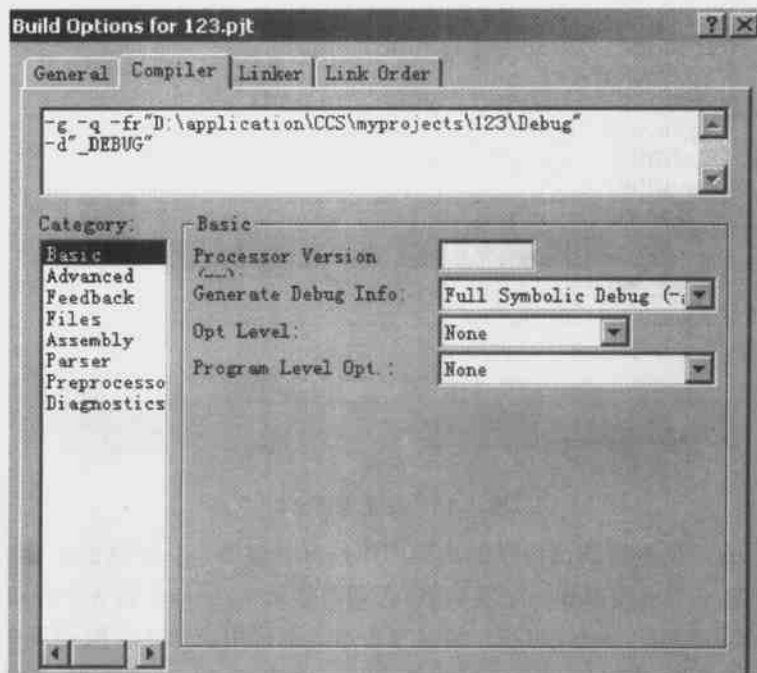


图 3.19 编译器设置对话框

3.2.5 Debug 菜单

Debug 菜单中包含的是比较常用的调试命令,其中比较重要的命令介绍如下:

(1) Breakpoints 设置/取消断点命令。当程序执行到断点时,将停止运行,这时可以检查程序的运行状态,查看并修改变量、存储器和寄存器的值等,也可以查看堆栈。设置程序的断点时应注意下面两点:

- ① 不要将断点设置在任何延迟分支或调用的指令的地方;
- ② 不要将断点设置在重复的块指令倒数第 1、2 行的指令的地方。

(2) Probe Points 探测点设置。允许更新观察窗口并在算法的指定处(也就是设置探测

点的地方)将主机文件的数据读到 DSP 目标系统的存储器,或将 DSP 目标系统的存储器的数据写入到主机上的文件中,此时应设置为 File 的 I/O 属性。

对每一个建立的窗口,默认情况是在每个断点处更新窗口显示。当使用探测点更新窗口时,目标程序将临时停止执行;当窗口更新后,程序将继续执行。因此,Probe Points 不能满足实时数据交换(RTDX)需要。

(3) Step Into 单步执行。如果运行到调用函数处,将跳入到函数中单步执行。

(4) Step Over 单步执行。与 Step Into 不同的是,为了保护处理器的流水线操作,该指令后的若干条延迟指令或调用指令将同时被执行。如果运行到函数调用处,将直接执行完整个函数的功能而不能跳入到函数内部单步执行,除非在函数内部设置了断点。

(5) Step Out 跳出函数或子程序执行。当使用 Step Into 或 Step Over 单步执行指令时,如果程序运行到一个子程序中,执行该命令将使程序执行完函数或子程序后,回到调用该函数或子程序的地方。在 C 语言源程序模式下,根据标准运行堆栈来推断返回地址;否则,根据堆栈顶的值来求得调用函数的返回地址。因此,如果汇编程序使用堆栈来存储其他信息,该指令有可能运行不正常。

(6) Run 从当前程序计数器(PC)执行程序,碰到断点时暂停。

(7) Halt 中止程序执行。

(8) Animate 动画运行程序。当碰到断点时程序暂时停止运行,在更新未与任何 Probe Point 相关联的窗口后程序继续执行。该命令的作用是在每个断点处显示处理器的状态,可以在 Option 菜单下选择 Animate Speed 来控制其速度。

(9) Run Free 从当前程序计数器(PC)处开始执行程序,将忽略所有的断点(包括 Breakpoints 和 Probe Points)。该命令在 Simulator 下无效,在使用 Emulator 进行仿真调试时,该命令将断开与目标系统板的连接,因此可以移走 JTAG 电缆。在运行 Run Free 指令时将 DSP 目标系统复位。

(10) Run to Cursor 程序执行到光标处,光标所在行必须为有效的代码行。

(11) Multiple Operation 设置单步执行的次数。

(12) Reset CPU 复位 DSP 目标系统,初始化所有的寄存器,中止程序的执行。

(13) Restart 将程序计数器(PC)的值恢复到程序的入口,但该命令不开始程序的执行。

(14) Go Main 在程序的 Main 符号处设置一个临时断点,该命令仅在调试 C 语言源代码时起作用。

3.2.6 Profiler 菜单

剖析点(profile point)是 CCS 的一个重要的功能,它可以在调试程序时,统计某一块程序执行所需要的 CPU 时钟周期数、程序分支数、子程序被调用数和中断发生次数等统计信息。Profile Point 和 Profile Clock 作为统计代码执行的两种机制,常常一起配合使用。Profiler 菜单中的主要命令介绍如下:

(1) Enable Clock 为了获得指令的周期及其他事件的统计数据,必须使能剖析时钟(profile clock)。当剖析时钟被禁止时,将只能计算到达每个剖析点的次数,而不能计算统计数据。

指令周期的计算方式与 DSP 的驱动程序有关,对使用 JTAG 扫描路径进行通信的驱动程序,指令周期通过处理器的片内分析功能进行计算,其他的驱动程序则可以使用其他类型的定时器。Simulator 使用模拟的 DSP 片内分析接口来统计剖析数据。当时钟使能时,CCS 调试器将占用必要的资源实现指令周期的计数。

剖析时钟作为一个变量(CLK)通过 Clock 窗口被访问。CLK 变量可在 Watch 窗口观察,并可以在 Edit Variable 对话框中修改其值。CLK 还可以在用户定义的 GEL 函数中使用。

Instruction Cycle Time 域用于执行一条指令的时间,其作用是在显示统计数据时将指令周期数转化成时间或频率。

(2) Clock Setup 时钟设置。单击该命令将出现如图 3.20 所示的 Clock Setup 对话框。

在 Count 域内选择剖析的事件。对某些驱动程序而言,CPU Cycle 可能是惟一的选项。对于使用片内分析功能的驱动程序而言,可以剖析其他事件,比如中断次数、子程序或中断返回次数、分支数或子程序调用次数等。

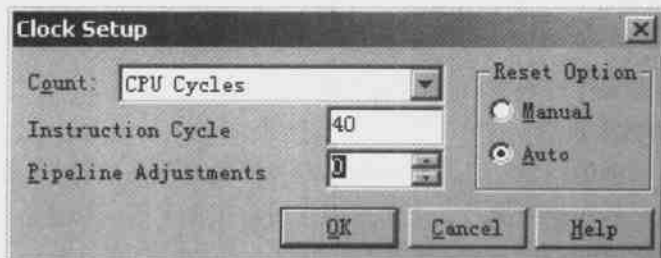


图 3.20 Clock Setup 对话框

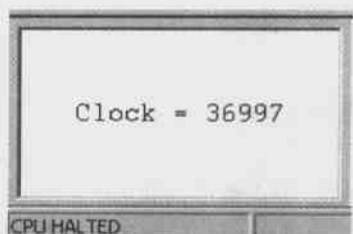


图 3.21 CLK 变量的值

使用 Reset Option 参数可以决定如何计数。如选择 Manual 选项,则 CLK 变量将不断累计指令周期数;如选择 Auto 选项,则在每次 DSP 运行前自动将 CLK 设置为 0。因此,CLK 变量显示的是上一次运行以来的指令周期数。

(3) View Clock 打开 Clock 窗口,以显示 CLK 变量的值,如图 3.21 所示。双击 Clock 窗口的内容可直接复位 CLK 变量(使 Clock=0)。

3.2.7 Option 菜单

Option 菜单用于设置字体、颜色和键盘等。其中比较重要的 Option 菜单命令介绍如下:

(1) Font 设置字体。单击该命令后出现如图 3.22 所示的字体设置对话框,在该对话框中可以设置字体、大小及显示样式等。

(2) Disassembly Style 设置反汇编窗口的显示模式。单击该命令,将出现如图 3.23 所示的设置对话框。在该对话框中,可以设置反汇编的显示为助记符或者代数符号,直接寻址与间接寻址显示为十进制、二进制或十六进制等。按照图 3.23 的设置,在反汇编窗口中的显示如图 3.24 所示。

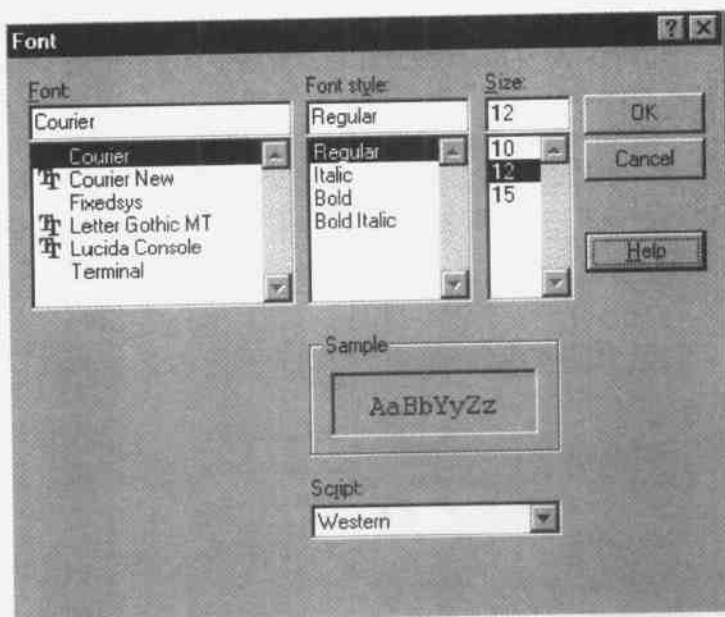


图 3.22 字体设置对话框

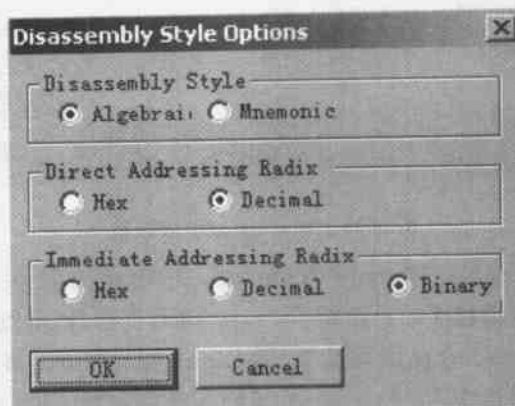


图 3.23 反汇编窗口显示模式的设置

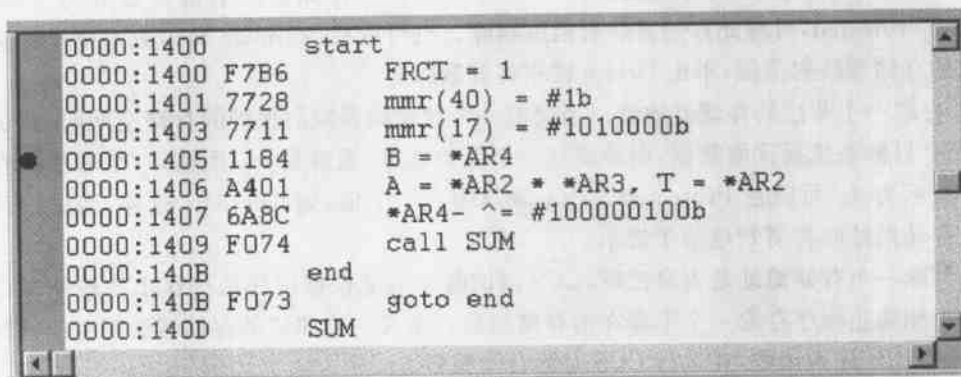


图 3.24 反汇编窗口中的显示

(3) Memory Map 定义存储器映射。存储器映射指明了 CCS 调试器能访问的哪段寄存器,不能访问的哪段寄存器。通常情况下,存储器映射与命令文件(*.cmd)定义的相一致。第一次运行 CCS 的时候,存储器映射是禁用的状态(没选中 Enable Memory Mapping),也就是说,CCS 调试器可以存取 DSP 目标板上所有的可寻址的 RAM 存储器。当使能存储器映射后,CCS 调试器将根据存储器映射的设置,检查其可以访问的存储器。如果要存取的是未定义或保护区的数据,则调试器将显示默认值而不是存取 DSP 目标系统的数据。

① 添加存储器映射设置。单击 Memory Map 命令,将弹出 Memory Map 对话框,如图 3.25 所示。

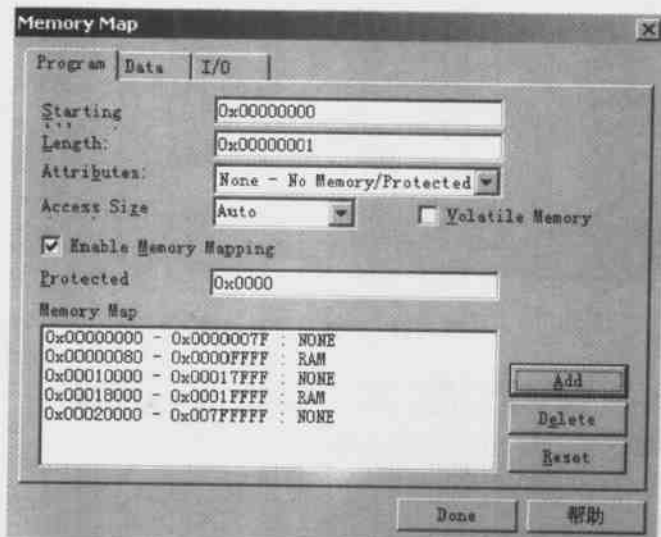


图 3.25 Memory Map 对话框

在对话框中选中 Enable Memory Mapping,使能存储器映射。选择修改的页面(Program, Data 或 IO),如果程序中只使用了一个存储器页面,则可以跳过这一步。

按照命令文件(*.cmd)的存储器定义,在 Starting address 域里面键入起始地址,在 Length 域里面键入存储器长度,在 Attributes 里面选择存储器的读/写属性,再单击 Add 按钮就可以添加一个新的存储器映射范围。

② 删除一个存储器映射范围。将一个已经存在的存储器映射属性设置为 None_No Memory/Protected,可将此存储器映射范围删除。也可以在 Memory Map 列表框里面选中需要删除的存储器映射范围,单击 Delete 按钮将其删除。

③ 存取一个非法的存储器地址。当读取一个被存储器映射保护的存储空间时,调试器不是从 DSP 目标系统板读取数据,而是读取一个保护数据,通常是 0。因此,一个非法的存储地址通常显示为 0。可以在 Protected Value 域里输入一个值,如 0x1234。这样,当试图读取一个非法存储地址时将清楚地给予提示。

在判断一个存储地址是否合法时,CCS 调试器并不是根据硬件结构作出比较结果,因此,调试器不能防止程序存取一个不存在的存储地址。定义一个非法的存储器映射范围的最好办法是使用 GEL 嵌入函数,在运行 CCS 时能自动执行。

3.2.8 GEL 菜单

CCS 软件 C5000 本身提供了 C54X 和 C55X 的 GEL 函数,它们在 C5000.gel 文件中定义。GEL 菜单中包括 C54X_CPU_Reset 和 C54X_Init 命令。

(1) C54X_CPU_Reset 命令复位目标 DSP 系统、复位存储器映射(处于禁止状态)以及初始化寄存器。

(2) C54X_Init 命令也对目标 DSP 系统复位,与 C54X_CPU_Reset 命令不同的是,该命令使能存储器映射,同时复位外设和初始化寄存器。

3.2.9 Tool 菜单

Tool 菜单提供了常用的工具集,比较重要的命令介绍如下:

(1) Data Converter Support 能快速配置与 DSP 芯片相连接的数据转换器。

(2) C54X McBSP 能观察和编辑 McBSP 的内容。C54X DSP 有多个高速、全双工的多信道缓冲器串行口(McBSP),从而使该 DSP 芯片能直接与系统中其他器件相连。McBSP 是建立在标准串行口之上的。

(3) C54XX Emulator Analysis 能设置、监视事件和硬件中断点的发生。C54X 芯片有一个片内分析模块,使用该模块,可以计算特定的硬件功能发生的次数或设置相应的硬件断点。单击该命令,将弹出如图 3.26 所示的观察窗口。在窗口中右击,可以设置监视的事件或硬件中断点,如图 3.27 所示。

Break	Address	Ac...	Dval	Hval	Status
DATA	0xffff	R	0xff...	0x0000	
PROG1	0xffff				- prog_win_start
PROG2	0xffff				- prog_win_end

图 3.26 事件和硬件中断点的观察窗口

(4) C54XX DMA 能观察和编辑 DMA 寄存器的内容。

(5) C54XX Simulator Analysis 能设置和监视事件的发生,并为加载调试器使用的特定的伪寄存器提供了一个透明的手段,其使用方法和 C54XX Emulator Analysis 完全一致。

(6) Command Window 能在命令窗口中键入可以执行的命令,键入的命令遵循 TI 调试器的命令语法格式。单击此命令,将弹出如图 3.28 所示的命令输入对话框。在命令输入栏中键入 run 命令,就可以直接运行程序。

(7) Port Connect 该命令将主机文件与存储器(端口)地址相连,从而可以从文件中读出数据或将数据从寄存器写入主机文件中。单击此命令,将弹出如图 3.29 所示的窗口。单击 Connect 按钮,弹出如图 3.30 所示的设置存储器地址(包括程序、数据或 I/O 地址)的对话框。在对话框中设置好地址后,单击 OK 按钮,又弹出如图 3.31 所示的端口文件输入对话框。该

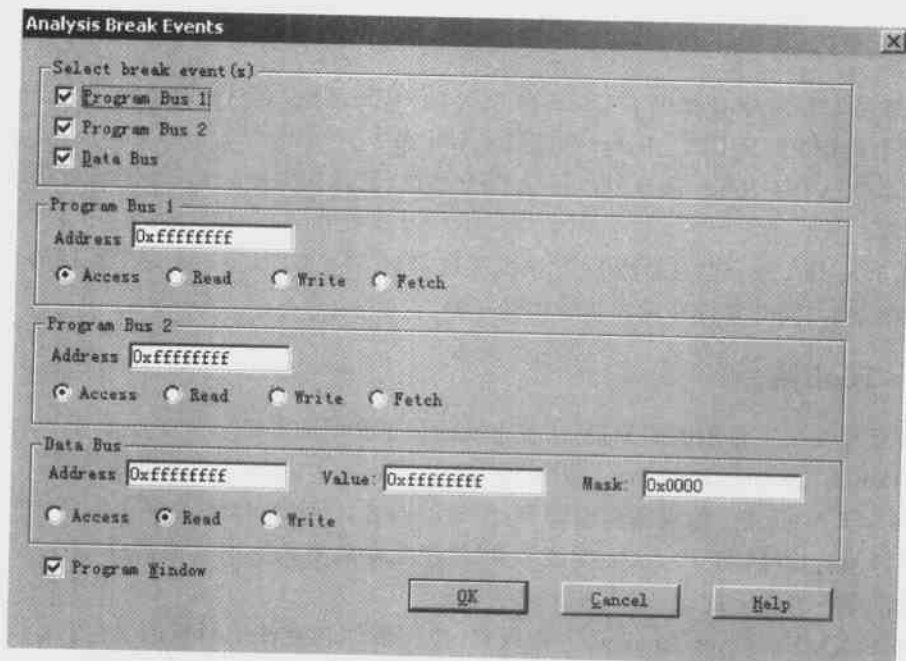


图 3.27 事件或硬件中断点的设置

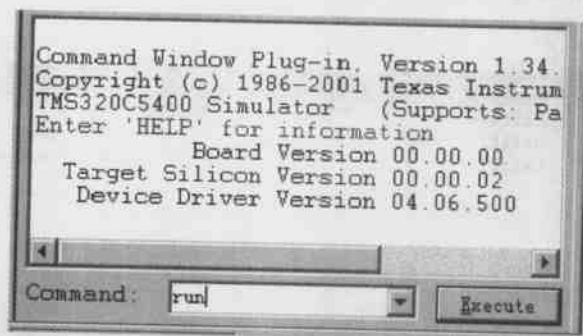


图 3.28 命令输入对话框

对话框中已经设置了一个主机中 C 盘下的文件 error.txt, 把数据写到从存储器地址 0x0080 开始的长度为 100 的数据区中。

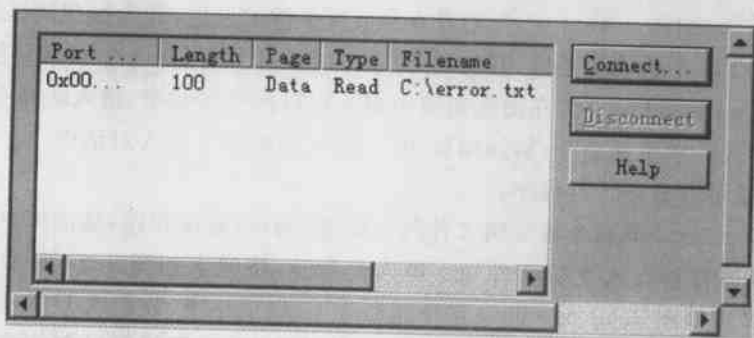


图 3.29 Port Connect 窗口

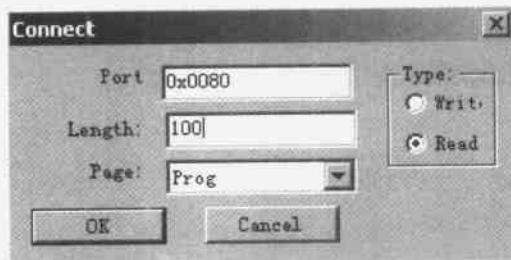


图 3.30 存储器地址设置对话框



图 3.31 端口文件输入对话框

(8) Pin Connect 用于指定外部中断发生的时间间隔,因此可以用 Simulator 来仿真和模拟外部中断信号。该命令设置的步骤如下:

- ① 创建一个数据文件来指定中断间隔时间,该文件是用 CPU 的时钟参数表示的,例如:

```
10 (+5 +20) rpt EOS
```

表示中断在第 10 个时钟周期被仿真,然后分别在第 15 个时钟周期(10+5)、第 35 个时钟周期(15+20)、第 40 个时钟周期(35+5)、第 60 个时钟周期(40+20)、第 65 个时钟周期(60+5)和第 85 个时钟周期(65+20)被仿真……按照此模式一直到仿真结束(EOS: end of simulation)。

- ② 从 Tool 菜单下选择 Pin Connect 命令,将弹出如图 3.32 所示的窗口,该窗口中所有外部引脚都没有连接(not connected)。

- ③ 单击 Connect 按钮,将弹出如图 3.33 所示的外部中断引脚数据文件对话框,选择创建好的数据文件,将其连接到所需要的外部中断引脚。

- ④ 加载并运行程序。

关于此命令的详细使用,将在第 8 章介绍。

(9) Linker Configuration 使用 Visual Linker 链接程序。

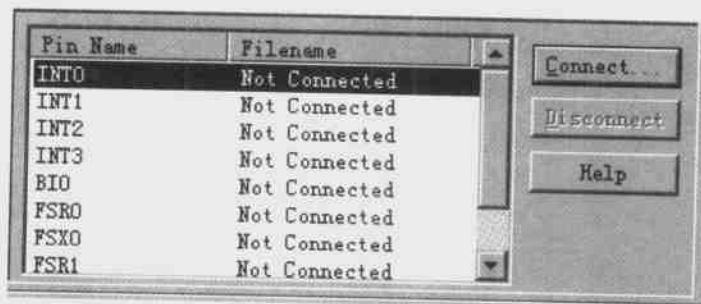


图 3.32 Pin Connect 窗口

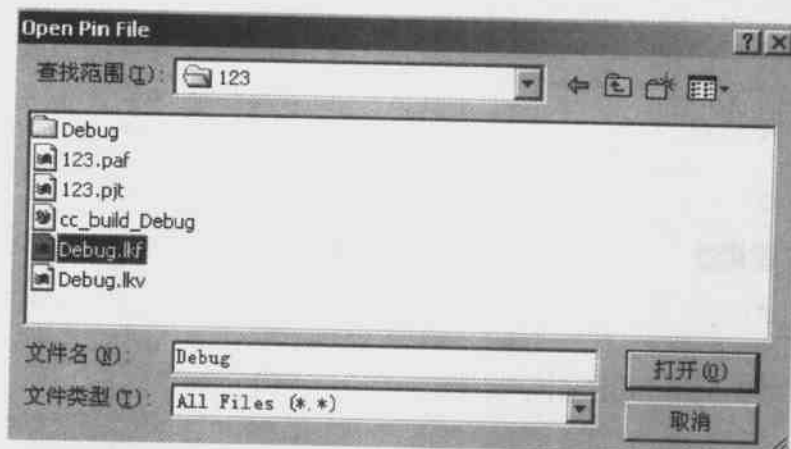


图 3.33 外部中断引脚数据文件对话框

3.2.10 DSP/BIOS 菜单

关于 DSP/BIOS 的功能,将在第 7 章详细介绍。

3.3 工具栏

CCS 集成开发环境提供了 5 种工具栏,以便执行各种菜单上相应的命令。

3.3.1 标准工具栏

标准工具栏(Standard Toolbar),如图 3.34 所示。

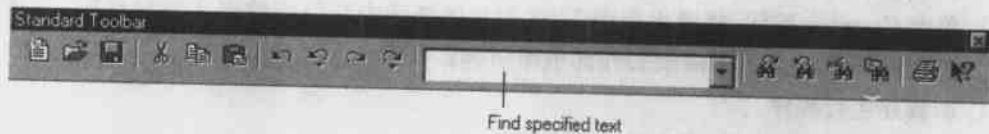


















图 3.34 标准工具栏

-  新建一个文档;
-  打开一个已经存在的文档;
-  保存一个文档,如果文档没有命名,则打开 Save As 对话框;
-  剪切;
-  复制;
-  粘贴;
-  取消上一次编辑操作;
-  显示可取消编辑操作的历史;
-  重复上一次取消的操作;
-  显示重复操作的历史;
-  查找下一个;
-  查找上一个;
-  查找指定的文本;
-  在多个文件中查找;
-  打印;
-  获得帮助。

3.3.2 工程工具栏

工程工具栏(Project Toolbar),如图 3.35 所示。

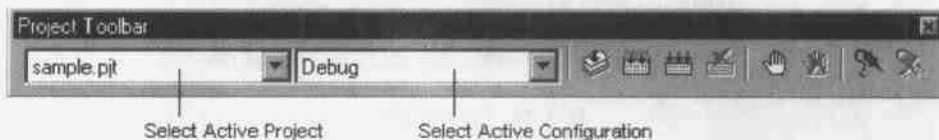










图 3.35 工程工具栏

-  编译当前文件;
-  重新编译只修改过的文件,再链接生成可执行程序;
-  编译整个工程并链接生成可执行文件;
-  停止当前 Build 操作;
-  在当前位置设置/取消断点;
-  取消所有断点;
-  在当前位置设置/取消剖析点;
-  取消所有剖析点。

3.3.3 调试工具栏

调试工具栏(Debug Toolbar),如图 3.36 所示。



图 3.36 调试工具栏

- 单步执行,与 Debug 菜单中的 Step Into 命令一致;
- 单步执行,与 Debug 菜单中的 Step Over 命令一致;
- 跳出函数或子程序执行,与 Debug 菜单中的 Step Out 命令一致;
- 运行到光标处;
- 运行程序;
- 中止程序运行;
- 动画运行,与 Debug 菜单中的 Animate 命令一致;
- 观察或编辑 CPU 寄存器的值;
- 查看指定地址的存储器的值;
- 查看堆栈的值;
- 查看反汇编代码。

3.3.4 编辑工具栏

编辑工具栏(Edit Toolbar),如图 3.37 所示。



图 3.37 编辑工具栏

- 将光标放在括号前面,单击此图标,则标记此括号内的所有文本;
- 查找下一个括号对,并标记其中的文本;
- 将光标放在括号前面,单击此图标,则光标跳至与之配对的括号;
- 将光标跳至下一个括号处;
- 将选择的文本左移动一个 Tab 宽度;
- 将选择的文本右移动一个 Tab 宽度;
- 设置一个书签(供查找定位);
- 查找下一个书签;
- 查找上一个书签;
- 打开书签对话框;
- 打开一个外部编辑器(如记事本)。

3.3.5 剖析工具栏

剖析工具栏(Profile Toolbar),如图 3.38 所示。



图 3.38 剖析工具栏

- 打开剖析会话;
- 关闭剖析会话;
- 剖析所有函数;
- 创建剖析区域;
- 创建剖析起始点;
- 创建剖析结束点;
- 使能剖析区域;
- 禁止剖析区域。

3.4 工程管理

3.4.1 建立、打开和关闭工程

建立新的工程文件,先运行 CCS,进入 C54XX 集成开发环境。在 Project 菜单中选择 New... 项,将弹出如图 3.39 所示的工程对话框,在 Project 栏中输入工程名字,其他栏目可以按自己习惯设置。工程文件的扩展名是 *.pj1。

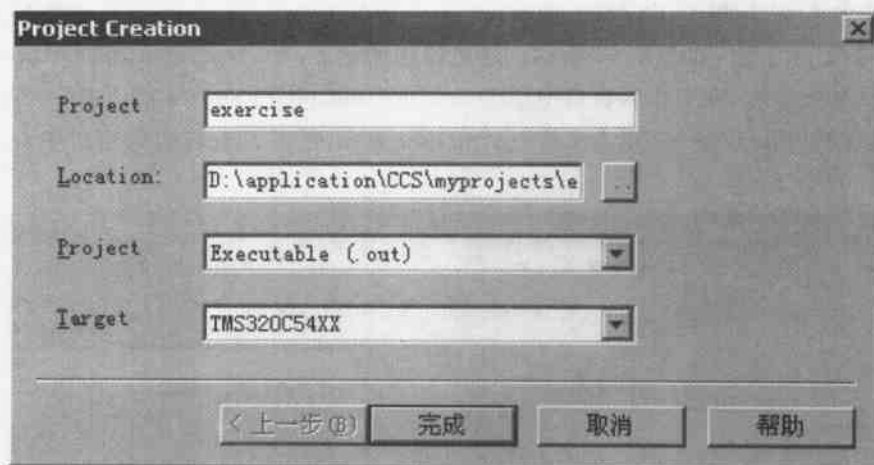


图 3.39 建立新的工程对话框

打开已经存在的工程,在 Project 菜单中选择 Open... 项,将弹出如图 3.40 所示的打开工程对话框。双击需要打开的文件(图中是 instructions.pj1)便可。

在 Project 菜单中选择 Close...,便可关闭当前工程。

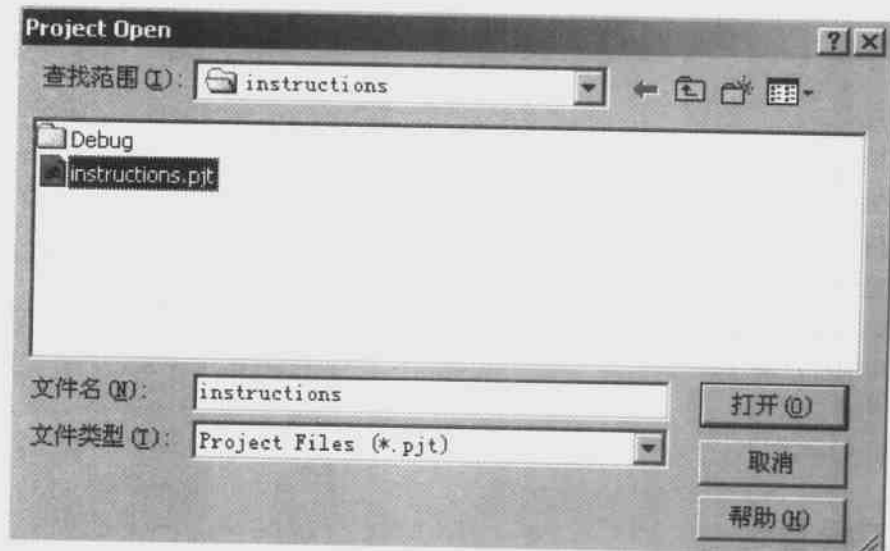


图 3.40 打开工程对话框

3.4.2 向工程中添加或删除文件

从如图 3.41 所示的 Project 菜单中选择 Add Files to Project... 命令, 或者在图 3.41 中右击 instructions.pjt, 则弹出 Add Files... 命令。根据需要, 添加各种类型的文件 (如 *.c、*.asm、*.cmd、*.h 和 *.lib 等)。文件添加后, 根据文件的类型, CCS 自动将其存放在工程文件的不同目录下面, 如图 3.42 所示。这里添加的是 instruction.asm 和 instruction.cmd 两个文件, 添加完毕后, 在工作区窗口中单击 instructions.pjt 和 Source 前面的“+”号, 可以展开显示的文件树 (File Tree)。双击文件 instruction.asm, 便可以在右边的窗口中看到汇编语言源代码。

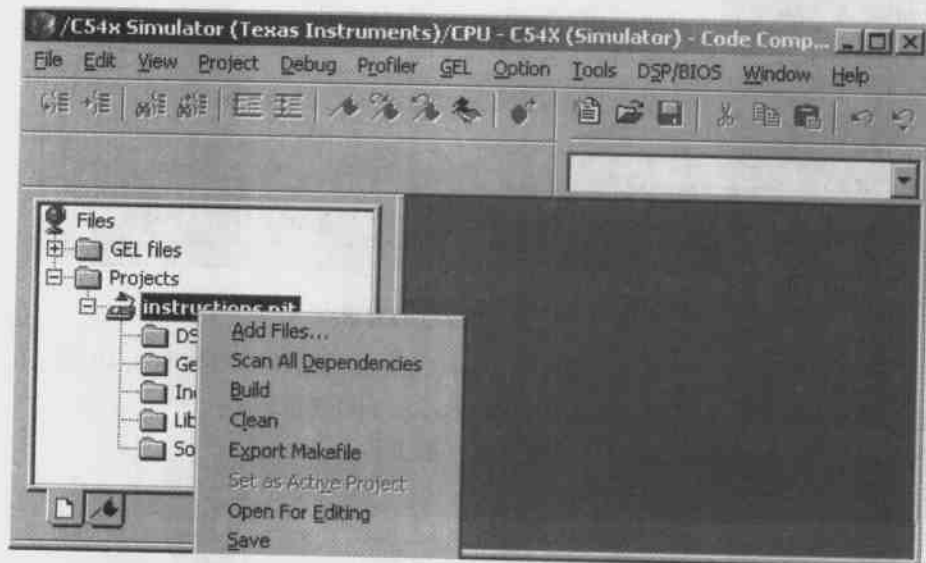


图 3.41 向工程中添加文件窗口

对于 C 语言源代码文件,如果源程序中有 `#include .h` 这样的语句,在添加了该源文件后,工程中将自动包含这些头文件(.h)在 Include 目录下。

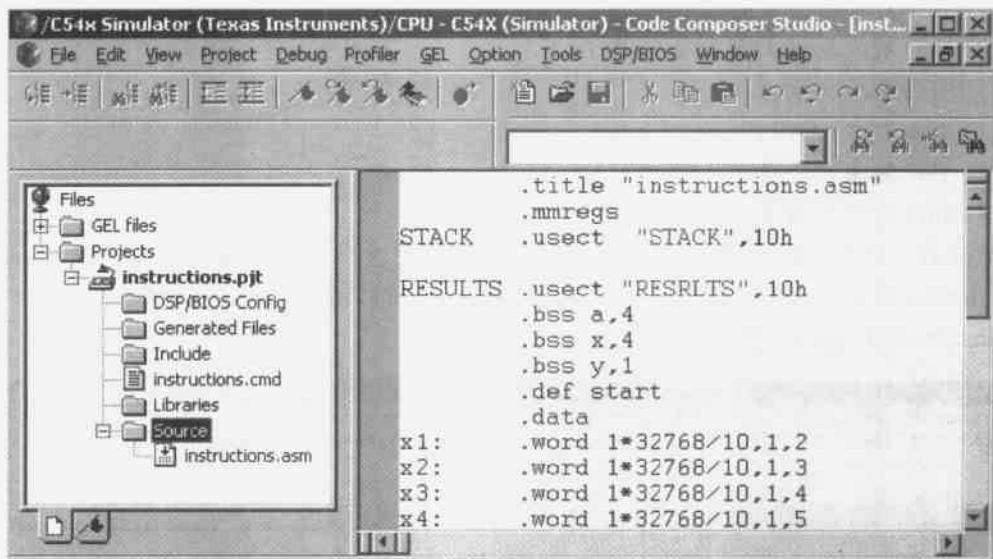


图 3.42 工程文件的不同目录

要从工程中删除文件,直接在图 3.42 中左边的窗口右击需要删除的文件名,选择 `Remove from Project` 命令便可。删除文件 `instruction.asm` 的操作如图 3.43 所示。

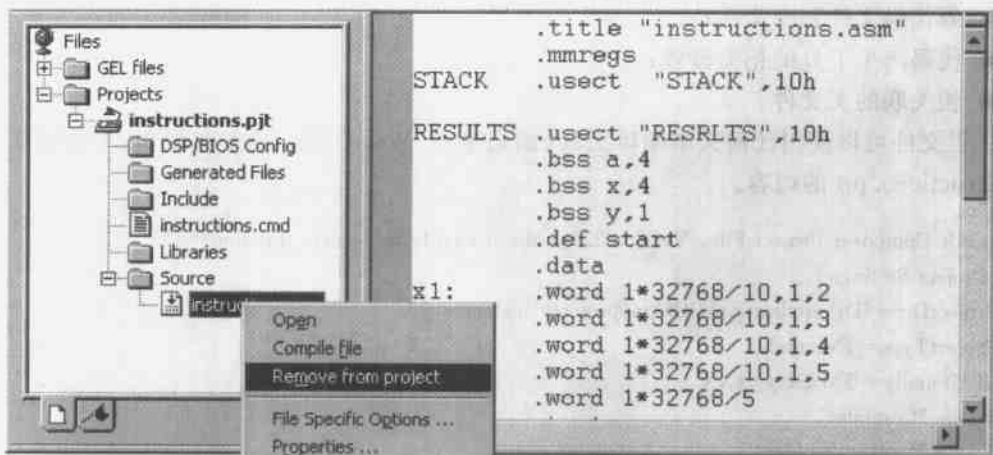


图 3.43 从工程中删除文件

3.4.3 查看文件关联性

CCS 的工程文件维持着一个关联文件的列表,以决定在编译工程时需要哪些文件。CCS 通过下面的路径查找相关联的文件,如: `#include .`, `include` 和 `copy` 等,并把这些文件名加入到工程的列表中。查找的当前路径就是源代码(.c 或 .asm)文件的存放路径,然后按照下面的顺序搜索相关路径。

对于 C 语言源文件而言：

- (1) 当前路径；
- (2) 在编译环境中设置的路径；
- (3) 由环境变量 C_DIR 设置的路径。

对于汇编源文件而言：

- (1) 当前路径；
- (2) 在汇编编译环境中设置的路径；
- (3) 由环境变量 A_DIR 设置的路径。

要显示工程文件的关联性，可以在 Project 菜单中选择 Show Dependence 或 Show All Dependences 命令，将弹出如图 3.44 所示的关联显示窗口，该窗口中显示的是图 3.42 的工程文件 instructions.pjt 的关联性。

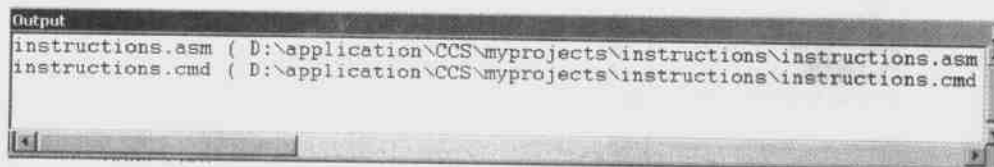


图 3.44 工程文件显示窗口

3.4.4 工程文件剖析

工程文件保存了应用程序的所有信息，其中包括：

- 源代码文件和库文件；
- 代码产生工具的相关设置；
- 相关联的头文件。

工程文件可以使用任何文本编辑工具（如记事本、Word）进行观察和编辑，下面是工程文件 instructions.pjt 的内容。

```
;Code Composer Project File, Version 2.0 (do not modify or remove this line)
```

```
[Project Settings]
```

```
ProjectDir="D:\application\CCS\myprojects\instructions\"
```

```
ProjectType=Executable
```

```
CPUFamily=TMS320C54XX
```

```
Tool="Compiler"
```

```
Tool="DspBiosBuilder"
```

```
Tool="Linker"
```

```
Config="Debug"
```

```
Config="Release"
```

```
[Source Files]
```

```
Source="instructions.asm"
```

```
Source="instructions.cmd"
```

```
["Compiler" Settings: "Debug"]
```

```
Options=-g-q-fr"D:\application\CCS\myprojects\instructions\Debug"-d"_DEBUG"
```

```
[ "Compiler" Settings: "Release"
```

```
Options=-g-o2-fr"D:\application\CCS\myproject\instructions\Release"
```

```
[ "DspBiosBuilder" Settings: "Debug"
```

```
Options=-v54
```

```
[ "DspBiosBuilder" Settings: "Release"
```

```
Options=-v54
```

```
[ "Linker" Settings: "Debug"
```

```
Options=-q-c-o".\Debug\instructions.out"-x
```


```
[ "Linker" Settings: "Release"
```

```
Options=-q-c-o".\Release\instructions.out"-x
```

3.5 源文件管理

3.5.1 创建新的源文件

创建新的源文件时,不影响已经存在的源文件。可以按照以下步骤创建新的源文件:


(1) 从 File 菜单中选择 New 命令,将打开一个新的源代码编辑窗口;也可在标准工具栏 (Standard Toolbar)单击图标.

(2) 在新的源代码编辑窗口输入代码。注意在编辑窗口的最顶部的标题栏上,有一个“*”号出现在文件名的右上角,当文件保存后“*”号消失,表示文件没被更新。

(3) 在 File 菜单中选择 Save 或 Save As 命令,保存文件。

3.5.2 打开文件

可以在编辑窗口中打开已经存在的任何 ASCII 类型的文件。

(1) 在 File 菜单中选择 Open 命令,将出现打开文件对话框;也可以在标准工具栏 (Standard toolbar)单击图标.

(2) 在打开文件对话框中双击需要打开的文件,或者选择需要打开的文件,并单击 Open 按钮。

3.5.3 保存文件

保存文件按如下步骤进行:

(1) 单击编辑窗口,使要保存的文件处于激活状态(即成为当前编辑的文件);在 File 菜单中选择 Save 命令,如果是第一次保存,文件的名字是 Untitled,并出现如图 3.45 所示的 Save as 对话框,要求输入保存文件的名字。

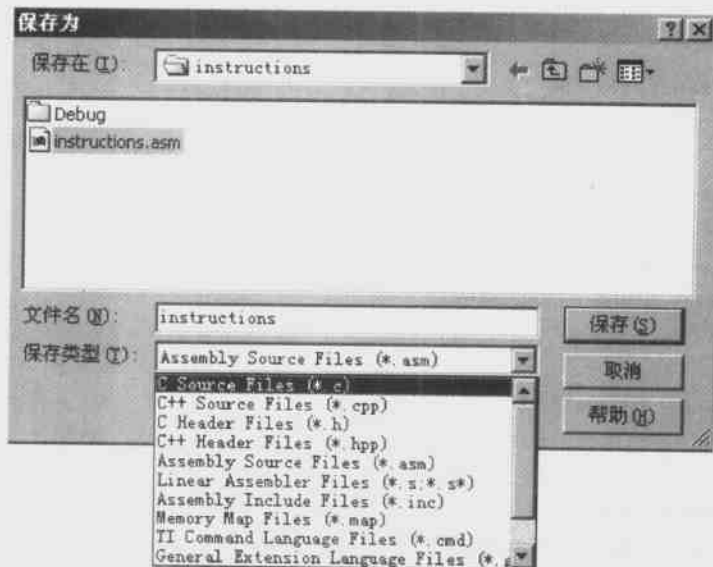



图 3.45 Save as 对话框

(2) 如果要更改已经保存的文件的文件名,则单击 Save as 命令,输入“另存为”的名字;也可以在标准工具栏上单击图标.

(3) 在保存类型栏中,如果源文件是汇编文件,则需要保存为 *.asm;如果是 C 语言文件,则需要保存为 *.c;如果是 C 语言的头文件,则需要保存为 *.h……。在缺省状态下,保存为 C 语言源文件,如图 3.45 所示。

(4) 单击“保存”按钮。

3.6 文件编辑

3.6.1 设置编辑属性

设置文件编辑属性是一个非常有用的功能,设置的方法如下:

在 Option 菜单中,选择 Customize... 命令,出现 Customize 对话框,单击 Editor Properties 页面,出现如图 3.46 所示的设置界面。

在图 3.46 中,可以设置以下功能:

- Tab 键的空格数,缺省空格数是 4;
- 在程序调试过程中,是否允许浏览源文件;
- 是否以只读方式打开源文件;
- 在编译程序前,是否自动保存文件;
- 是否允许汇编语言源文件和 C 语言源文件混合编程;
- 是否支持外部编辑器,如(记事本、Word 等);
- 设置在 File 菜单中显示的最近打开文件的个数。

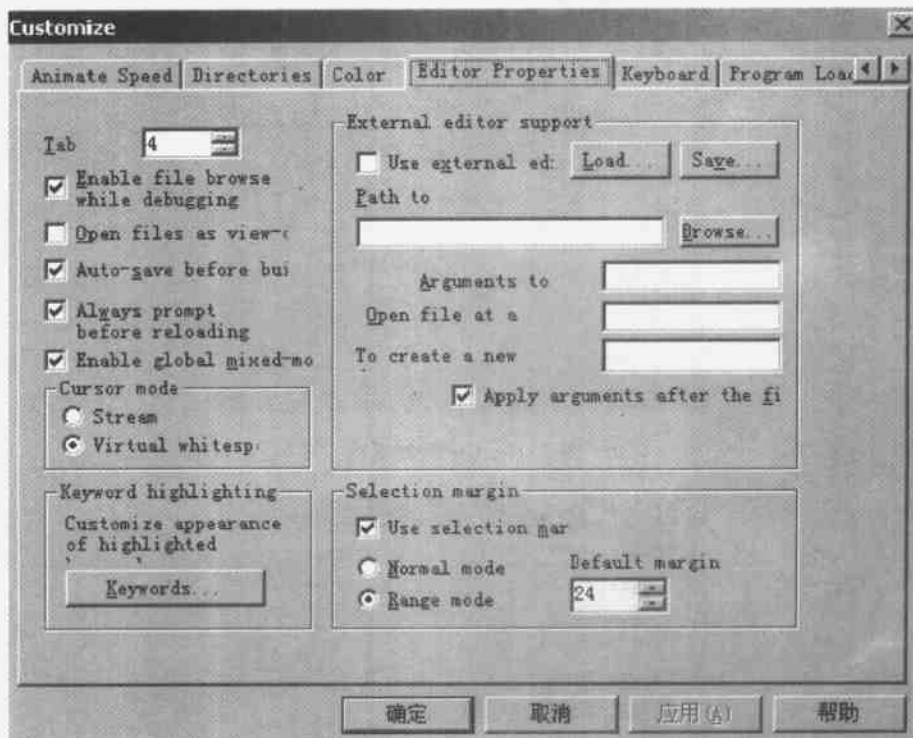





图 3.46 文件编辑属性设置界面

3.6.2 拷贝、剪切和粘贴文本

在源文件中,拷贝和剪切文本前先要选定需要操作的内容,再在 Edit 菜单中选择 Copy 或 Cut 命令;也可在标准工具栏上单击  或  图标进行拷贝或剪切。

然后将光标定位在需要粘贴的位置右击,选 Paste 命令;也可在 Edit 菜单中选择 Paste 命令或在标准工具栏上单击  图标进行粘贴。

3.6.3 编辑整列

可以对源文件中的几列进行复制、剪切或粘贴操作,而不需要涉及整行。

单击编辑窗口,使需要编辑的文件处于当前状态,在 Edit 菜单中选择 Column Editing 命令或按 Ctrl + Shift + F8 键,进入列编辑状态,然后可对选择的文字做编辑操作(复制、剪切或删除等)。

3.6.4 跳到指定行

可以利用 Go To 命令快速地在编辑窗口中找到指定的文本行或书签。其步骤如下:

- (1) 在 Edit 菜单中选择 Go To 命令,将出现 Go To 对话框,如图 3.47 所示。
- (2) 选中 Line 项,在 Enter 栏中填入需要查找的行号,图中是第 10 行,单击 OK,则光标将自动跳到第 10 行去。
- (3) 选中 Bookmark 项,则出现文件中已经设置的所有书签,如图 3.48 所示。

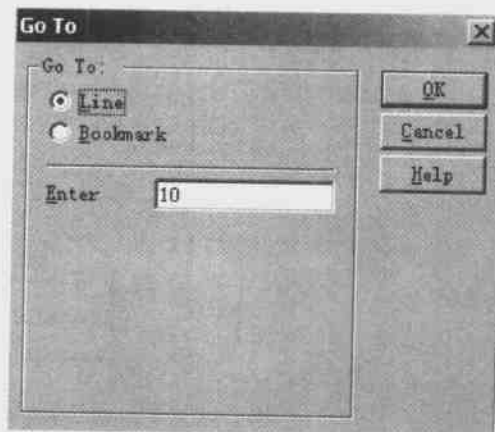


图 3.47 Go To 对话框

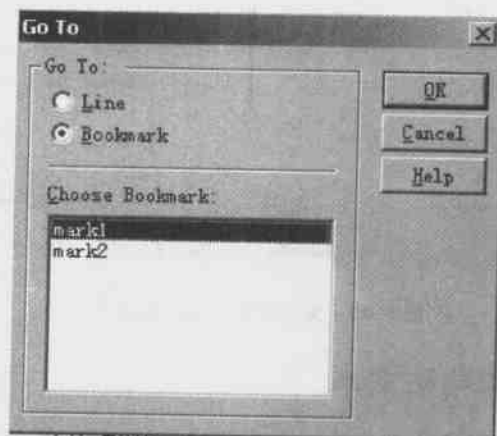


图 3.48 已经设置的书签

(4) 单击需要查找的书签,图 3.48 中是 mark1,单击 OK,则光标将自动跳到 mark1 处。如图 3.49 所示。RESULTS 前面的小旗,表示书签 mark1。

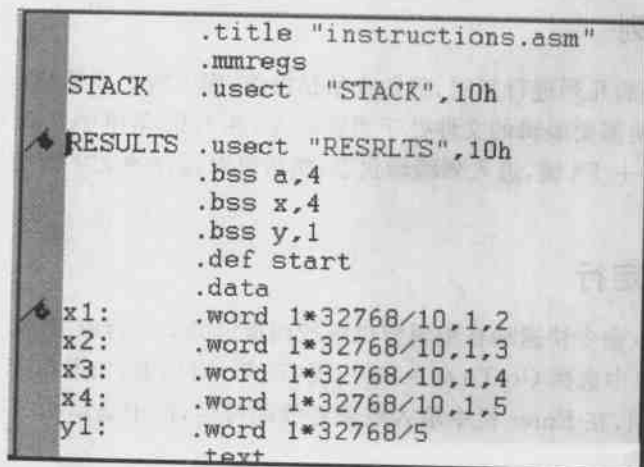


图 3.49 跳到书签 mark1 处




3.6.5 查找和替换文本

可以在当前活动的文本文件中查找字符串,也可以在整个工程的所有文本文件中查找字符串。

(1) 在标准工具栏中的查找文字域中,输入需要查找的文字,这里输入的是字符串“word”,如图 3.50 所示。



图 3.50 标准工具栏中查找字符串

(2) 单击 、 或  图标,则光标可直接跳到需要查找的字符串的位置,如图 3.51 所示。再单击该图标,则光标跳到下一个“word”字符串的位置。

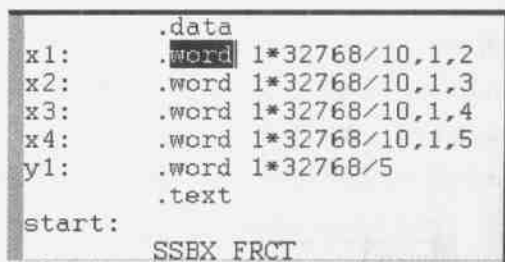


图 3.51 光标跳到需要查找的字符串的位置

(3) 也可以在 Edit 菜单中选择 Find/Replace 命令,则弹出如图 3.52 所示的对话框。分别在 Find 域中输入需要查找的字符串,在 Replace 域中输入代替查找的字符串,单击相应的 Find Next、Replace 或 Replace All 按钮即可。在这里是把字符串“word”替换成“tyte”。

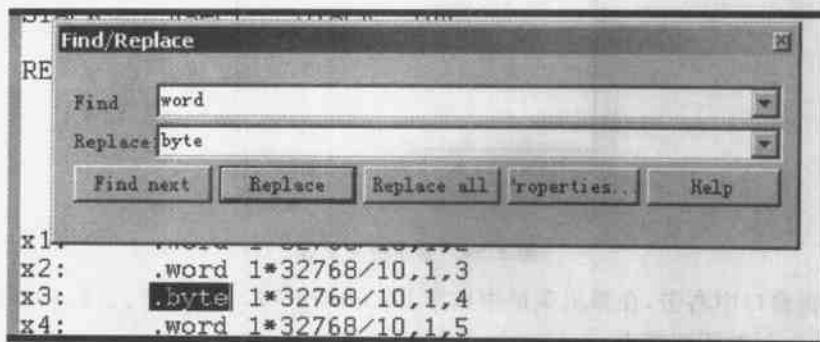



图 3.52 Find/Replace 对话框

(4) 也可在整个工程文件中查找字符串,在 Edit 菜单中选择 Find in Files 命令,或者在标准工具栏中单击  图标,则弹出如图 3.53 所示的对话框。这里的设置是:在工程文件 myproject 中的扩展名 *.c 或 *.h 的文件中,查找字符串“word”。

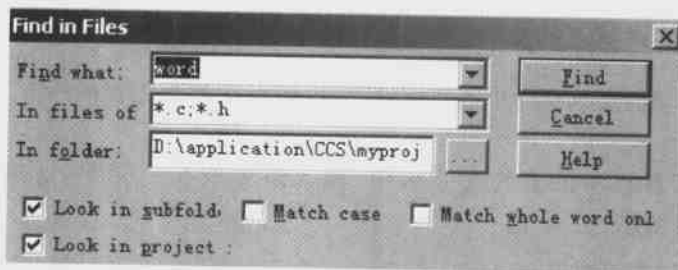



图 3.53 工程文件中查找字符串对话框

3.6.6 利用书签

在编辑窗口中,可以利用书签快速地查找和定位文本。书签可以设置在源代码文件的任何位置。书签的操作包括:

(1) Set a bookmark(设置书签) 在需要设置书签的地方右击,选择 Bookmark→Set a Bookmark 命令,或者在 Edit Toolbar(编辑工具栏)中单击  图标,就可以设置一个书签。

(2) View the list of bookmarks(观察书签列表) 有以下 4 种方法可以观察书签列表:

① 在工程窗口中,单击 Bookmarks 书签,则显示所有的书签名字,如图 3.54 所示。

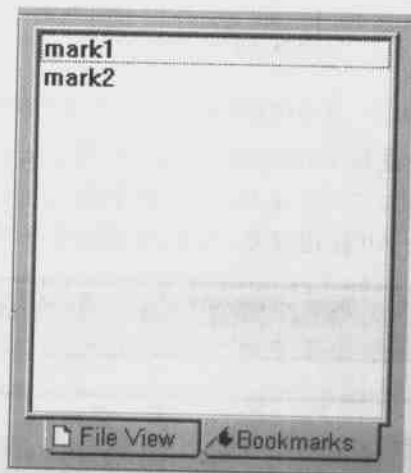




图 3.54 显示所有书签

② 在编辑窗口中右击,在弹出菜单中选择 Bookmark→Bookmark... 项,将弹出如图 3.55 所示的 Bookmark 对话框列表。

③ 在 Edit 菜单中,选择 Bookmark 命令,将弹出如图 3.55 所示的 Bookmark 对话框列表。

④ 在 Edit Toolbar(编辑工具栏)中单击  图标,将弹出如图 3.55 所示的 Bookmark 对话框列表。

(3) Move to the next or previous bookmark(跳到下一个或前一个书签位置) 在 Edit Toolbar 中单击  图标,光标就移到下一个书签位置;单击  图标,光标就移到前一个书签

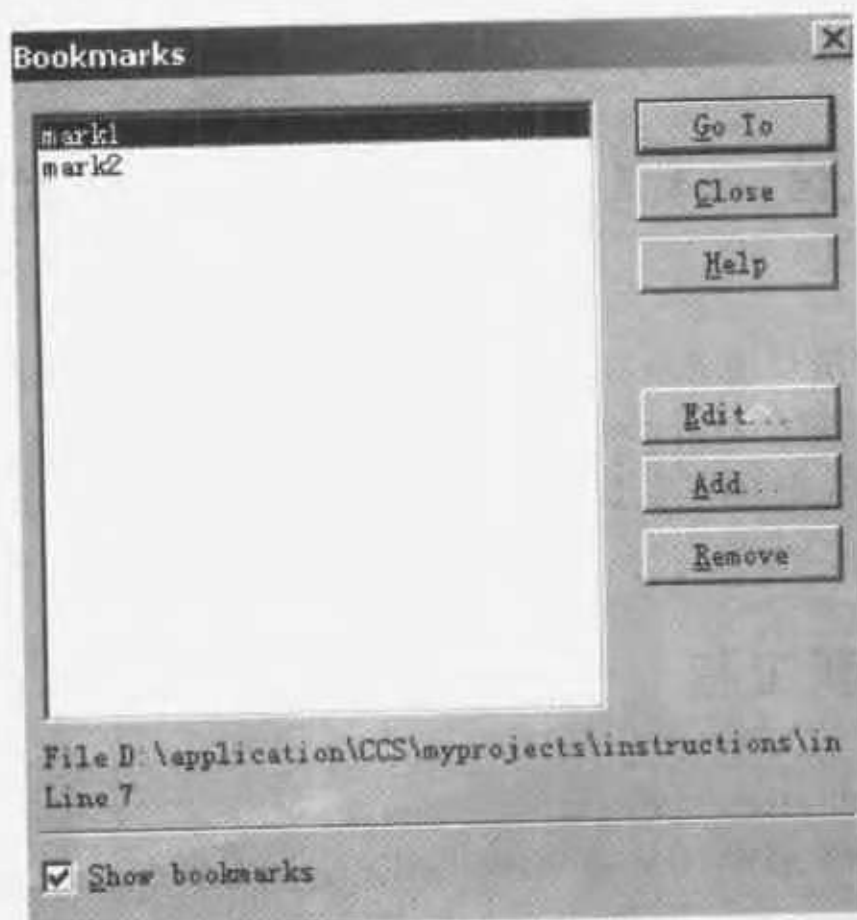



图 3.55 Bookmark 对话框列表

位置。

(4) Remove a Bookmark from a Source File(清除书签) 有以下 3 种方法可以清除书签标记:

① 在编辑窗口中,把光标移动到需要删除的书签的位置右击,选择 Bookmarks→Remove Bookmark 命令。

② 在 Edit 菜单中,选择 Bookmark 命令,将弹出如图 3.53 所示的 Bookmark 对话框列表,选中需要删除的书签,单击 Remove 按钮。

③ Edit Toolbar 中单击  图标,将弹出如图 3.53 所示的 Bookmark 对话框列表,选中需要删除的书签,单击 Remove 按钮。

第4章 CCS的C语言调试实例

本章以一个简单的用C语言编写的DSP程序 hello.c 为例,介绍用 CCS 开发 DSP 应用程序的步骤。假设 CCS 安装在 D:\application 目录,本章中用到的 C 语言编写的 DSP 程序,是 CCS 自带的源代码程序,在 CCS 装好后的 D:\application\ccs\tutorial\sim54XX\hello1 下面。

4.1 创建一个新工程

- (1) 运行 Code Composer Studio,启动 Simulator。
- (2) 在 Project 菜单中选择 New 命令,弹出如图 4.1 所示的 Project Creation 对话框。假



图 4.1 Project Creation 对话框

设 CCS 安装在 D:\application 目录下,在 Project 栏中输入需要创建的工程名字:hello,其扩展名缺省为 *.pjt。新创建的工程,将保存在 D:\application\ccs\myprojects 下。单击 Location 栏后面的  按钮,可以改变 hello 工程的保存路径。

(3) 在 Project 栏中,选择 Executable,在 Target 栏中选择 TMS320C54XX。这里假设 DSP 芯片 CPU 是 54XX 系列的。

(4) 单击“完成”按钮,将在工程窗口中的 Projects 下面创建 hello 工程,如图 4.2 所示。

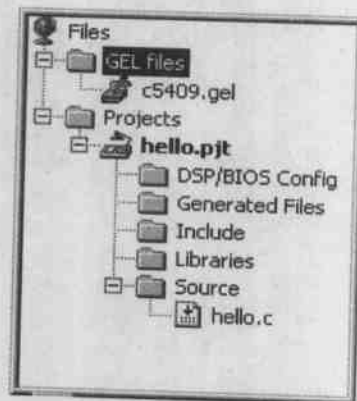


图 4.2 新建的 hello 工程

4.2 向工程中添加文件

向工程中添加文件的步骤如下:

(1) 在 Project 菜单中单击 Add Files to Project... 命令, 或者在工程窗口中的 hello. pj1 处右击, 再单击 Add Files... 命令, 打开 Add Files to Project 对话框, 如图 4.3 所示。在 D:\application\ccs\myprojects 目录下找到 hello. c 文件, 将其加入到工程中。



图 4.3 添加源文件

(2) 用同样的方法, 把文件 D:\application\ccs\tutorial\sim54xx\hello1\vectors. asm 也添加到工程 hello. pj1 中。vectors. asm 中包含的是将 RESET 中断并指向 C 程序入口 c_int00 的汇编指令。用户可以在 vectors. asm 中自定义中断矢量, 如果需要调试的程序比较复杂, 也可以使用 DSP\BIOS 定义的所有中断矢量。

(3) 将 hello. cmd 加入到工程 hello. pj1 中。该文件是本工程的存储器地址分配配置。

(4) 将 rts. lib 加入到工程 hello. pj1 中。该文件是 C 语言开发的 DSP 应用程序的运行支持库文件。

(5) 在工程视图窗口中单击所有的“+”号, 可以看到整个 hello 工程的结构, 如图 4.4 所示。

在上面的操作中, 没有将 C 语言源代码中的头文件 #include“hello. h”加入到工程中, 这是因为 CCS 在编译时将自动查找所需要的头文件, 在编译链接完成后, 可以在工程视图窗口中观察到工程所需要的所有头文件。

如果要从工程中删除一个文件, 可以在工程视图窗口中选择该文件, 然后按 Delete 键即可, 或者在所删除的文件上右击, 选择 Remove 命令即可。

当编译链接一个应用程序时, CCS 会自动依次从以下路径中查找工程所需要的文件:

(1) 源文件所在的目录。

(2) 在 Project 菜单的 Build Option 命令下, 在 Compiler 和 Assembler 选项中设置的路径。

(3) 在环境变量 C54X_C_DIR(C 编译器)和 C54X_A_DIR(汇编器)中声明的路径,C54X_C_DIR 所指向的路径中包含有 rts.lib 文件的目录。

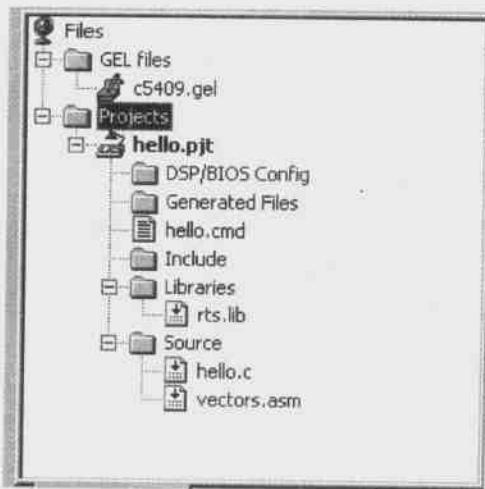


图 4.4 展开后的工程文件

4.3 代码浏览

在工程窗口中,双击工程 hello.pjt 中的 hello.c,在右边的源代码编辑窗口中可以观察到如下的源代码:

```

/*
 * Copyright 2001 by Texas Instruments Incorporated.
 * All rights reserved. Property of Texas Instruments Incorporated.
 * Restricted rights to use, duplicate or disclose this code are
 * granted through contract.
 * U. S. Patent Nos. 5 283 900 5 392 448
 */
/* "@(#) DSP/BIOS 4.51.0 05-23-01 (barracuda-il0)" */
/*****
/*
/* HELLO.C
/*
/* Basic C standard I/O from main.
/*
/*
/*
/*****

#include <stdio.h>
#include "hello.h"

```

```

#define BUFSIZE 30

struct PARMS str {
    int          i;
    char         scanStr[BUFSIZE];
    char         fileStr[BUFSIZE];
    size_t       readSize;
    FILE         * fptr;
};

/*
 * ===== main =====
 */
void main()
{
    #ifdef FILEIO
    int          i;
    char         scanStr[BUFSIZE];
    char         fileStr[BUFSIZE];
    size_t       readSize;
    FILE         * fptr;
    #endif

    /* write a string to stdout */
    puts("hello world! \n");

    #ifdef FILEIO
    /* clear char arrays */
    for (i = 0; i < BUFSIZE; i++) {
        scanStr[i] = 0; /* deliberate syntax error */
        fileStr[i] = 0;
    }

    /* read a string from stdin */
    scanf("%s", scanStr);

    /* open a file on the host and write char array */
    fptr = fopen("file.txt", "w");
    fprintf(fptr, "%s", scanStr);
    fclose(fptr);

    /* open a file on the host and read char array */
    fptr = fopen("file.txt", "r");
    fseek(fptr, 0L, SEEK_SET);

```

```

    readSize = fread(fileStr, sizeof(char), BUFSIZE, fptr);
    printf("Read a %d byte char array: %s\n", readSize, fileStr);
    fclose(fptr);
#endif
}


```

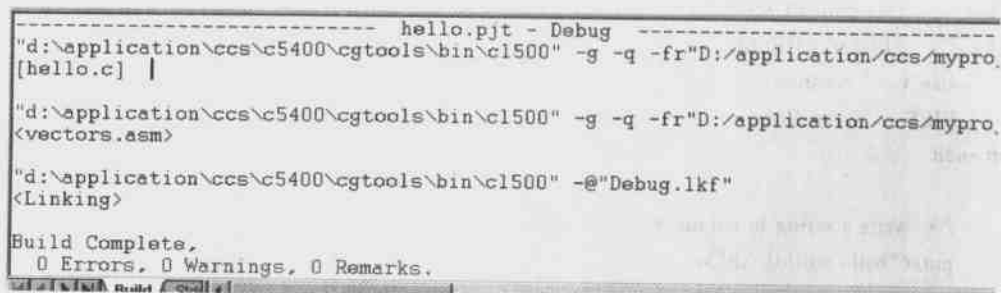
在 Option 菜单中选择 Font 命令, 可以改变字体大小。

该程序的目的是, 如果 FILEIO 未定义, 此程序只能在屏幕上显示“hello world!”字符串; 如果定义了 FILEIO, 则程序将提示输入一个字符串并将其存入文件, 然后从文件中读出该字符串并显示在屏幕上。

4.4 编译、链接和运行程序

编译、链接和运行程序的步骤如下:

(1) 执行菜单命令 Project→Rebuild All 或者在 Project 工具栏上双击  图标, 对工程进行编译、汇编和链接, 在 Output 窗口中将显示相关信息, 如图 4.5 所示。如果汇编失败, 则返回工程中继续修改程序, 直到汇编通过为止。



```

----- hello.pjt - Debug -----
"d:\application\ccs\c5400\cgtools\bin\cl500" -g -q -fr"D:\application\ccs\mypro.
[hello.c] |
"d:\application\ccs\c5400\cgtools\bin\cl500" -g -q -fr"D:\application\ccs\mypro.
<vectors.asm>
"d:\application\ccs\c5400\cgtools\bin\cl500" -@"Debug.lkf"
<Linking>
Build Complete.
0 Errors, 0 Warnings, 0 Remarks.

```


图 4.5 CCS 汇编信息输出窗口

(2) 目标文件 hello.out 生成成功后, 就可在 CCS 集成环境中调试程序。假设 CCS 安装在 D:\application 目录下, 则所生成的目标程序位于 D:\application\ccs\myprojects\hello\debug 目录下。

(3) 执行菜单命令 File→Load Program, 选择 hello.out 并打开, 将生成的可执行程序加载到 DSP 中, CCS 将自动打开一个“反汇编”窗口, 显示加载程序的反汇编指令。

正常情况下, 目标程序可以成功载入。如果因为 DSP 的 CPU 型号不对, 或者配置文件不对, 或者内存溢出等, 则无法装载目标程序。这时, CCS 会提示装载失败的原因, 如图 4.6 所示。

(4) 在“反汇编”窗口中, 单击任何一个指令, 或将鼠标放在一条指令上, 按 F1 键, 则 CCS 将搜索该指令的相关帮助信息, 这是获取一条指令帮助的最好方法。比如, 将光标放在 c_int00 行下的 ADDM 指令下, 按 F1 键, 将弹出如图 4.7 所示的帮助信息。

(5) 执行菜单命令 Debug→Run, 或者在 Debug 工具栏上单击  按钮, 可以在 Output 窗口中观察到“hello world!”信息, 如图 4.8 所示。

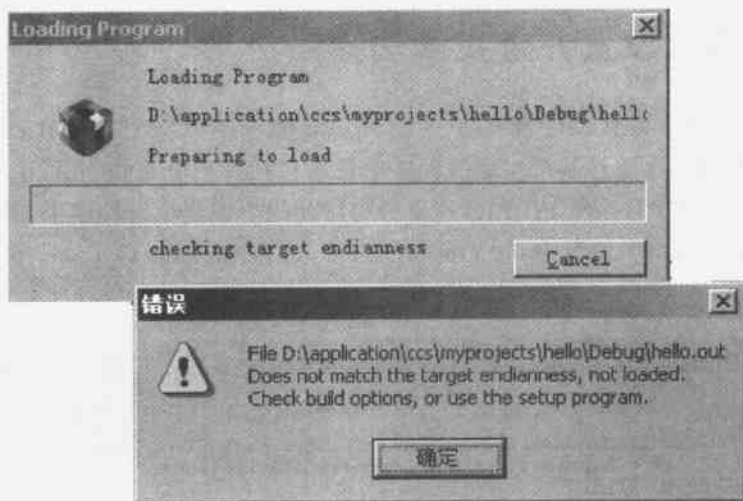


图 4.6 目标程序装载失败提示

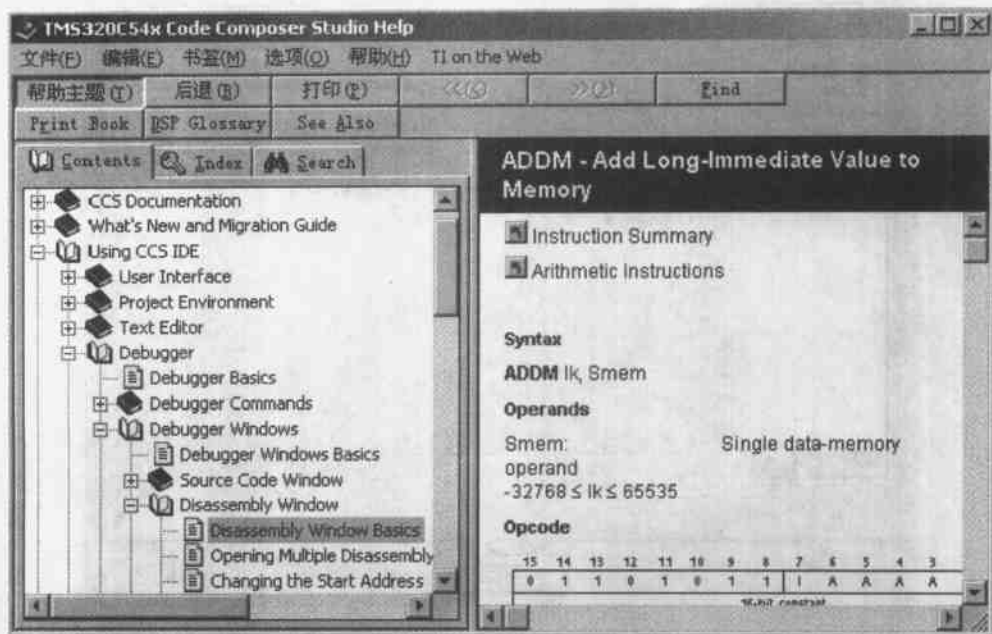


图 4.7 ADDM 指令的帮助信息

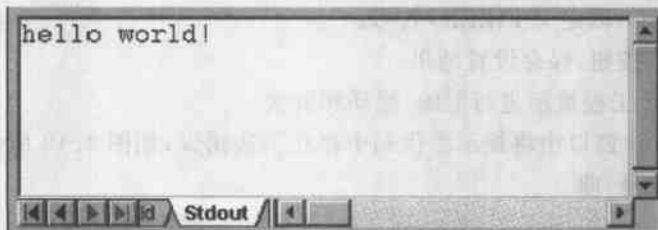


图 4.8 Output 窗口显示的程序运行结果

4.5 改变程序设置并查找语法错误

在 4.3 节的程序中, FILEIO 没有被定义, 所以在源程序中“#ifdef FILEIO”到“#endif”的两段代码将被忽略, 在编译链接生成的程序中不包括这部分可执行代码。可以通过更改程序选项来定义 FILEIO, 并将 FILEIO 这部分程序代码生成到执行程序中, 其步骤如下:

(1) 在 Project 菜单中执行 Build Options 命令, 将弹出 Build Options for hello.pjt 对话框, 如图 4.9 所示。

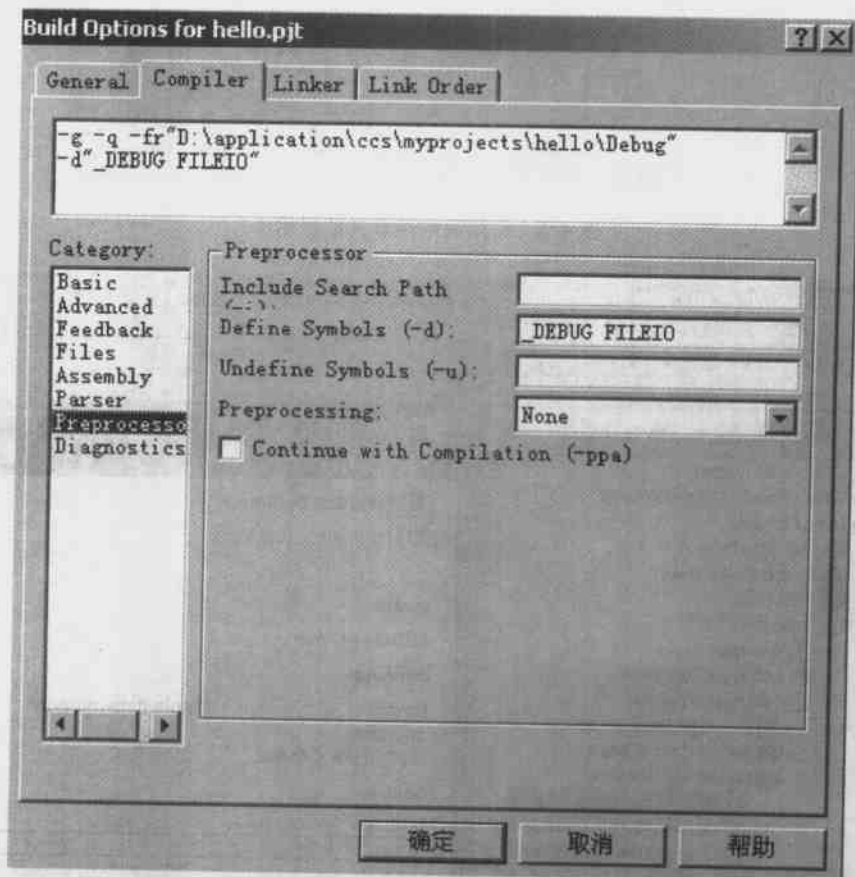


图 4.9 Build Options for hello.pjt 对话框

(2) 单击 Compiler 标签, 在 Category 栏中选择 Preprocessor, 然后在右侧的 Define Symbols 栏中键入 FILEIO, 以定义 FILEIO 符号。

(3) 单击“确定”按钮, 保存设置结果。

(4) 对 hello.pjt 工程重新进行汇编、编译和链接。

(5) 这时, Output 窗口中将提示源代码中存在语法错误, 如图 4.10 所示。错误出现在第 53 行, 丢失了一个分号, 即

"hello.c", line 53: error: expected a ";"

(6) 在第53行后加上分号存盘,再将修改后的文件重新汇编、编译和链接,生成DSP应用程序。

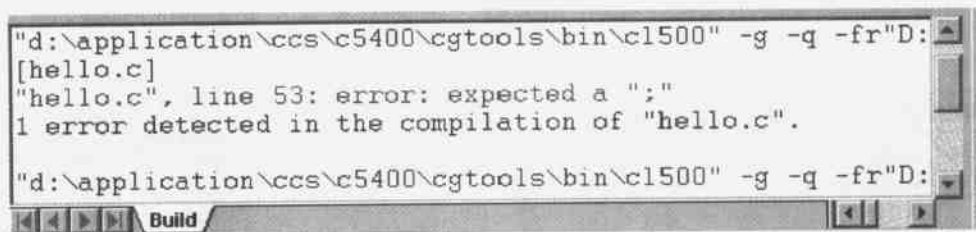




图 4.10 语法错误提示

4.6 使用断点和观察窗口

使用断点和观察窗口的操作步骤如下:

(1) 成功生成目标文件 hello.out 后,先让系统复位,在 File 菜单中执行 Load program... 命令,加载可执行程序。

(2) 在工程窗口中双击 hello.c 文件,打开源文件编辑窗口,将光标放在行语句“fprintf(fp, “%s”, scanStr)”上,按 F9 键,或者在工具栏上单击  按钮,设置断点,此时,该行语句前面将出现一个红色的圆点。

(3) 执行菜单命令 Debug→Run,或者在 Debug 工具栏上单击  按钮,将出现如图 4.11 所示的输入一个字符串对话框,并将输入的字符串存入文件,然后从文件中读出该字符串,并输出到屏幕上,如图 4.12 所示。

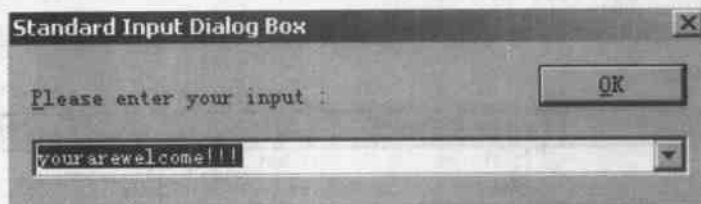


图 4.11 输入字符串对话框

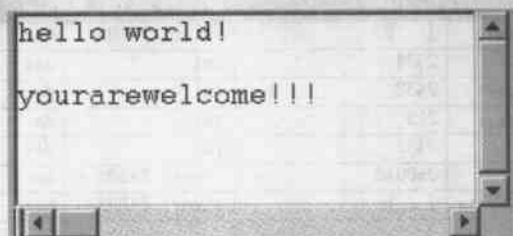


图 4.12 运行结果

(4) 在 View 菜单中执行 Watch Window 命令,将出现观察窗口。在程序运行过程中,可以从 Watch 窗口中观察变量的值。如图 4.13 所示,Watch 窗口中显示了变量 scanStr 的值为“youarewelcom!!!”。单击变量 scanStr 前面的“+”号,可以展开整个字符串,看到每个字符的值。

Name	Value	Type	Radix
scanStr	0x0FDD "youarewelcom!!!"	char[31]	hex
[0]	'y'	char	char
[1]	'o'	char	char
[2]	'u'	char	char
[3]	'r'	char	char
[4]	'a'	char	char
[5]	'r'	char	char

图 4.13 Watch 窗口中的变量 scanStr 的值

(5) 可按 F10(Step Over)、F8(Step Into)、S+F7(Step Out)或 C+F10(Run to Cursor)等键进行其他方式的调试。

4.7 观察结构体的值

根据 4.6 节的方法,在 Watch 窗口中加入 str 变量,可以看到在 str 的左边有一个“+”号,表明 str 是一个结构体变量。双击“+”号后可看到结构体中包含的元素,如图 4.14 所示,在这里还可以编辑各个变量的值。由于在文件 hello.h 中定义 str 为 PARMS 结构体,而 Link 为一个指向结构体的指针,其地址每次显示的值可能会不同。双击 Link 左边的“+”号,可发现它是一个无穷嵌套。

Name	Value	Type	Radix
str	{...}	struct PARMS	hex
Beta	2934	int	dec
EchoPower	9432	int	dec
ErrorPower	213	int	dec
Ratio	9432	int	dec
Link	0x00A5	struct PARMS *	hex
(*str.Link)	{...}	struct PARMS	hex
Beta	2934	int	dec
EchoPower	9432	int	dec
ErrorPower	213	int	dec
Ratio	9432	int	dec
Link	0x00A5	struct PARMS *	hex
((*str.Link).Link)	{...}	struct PARMS	hex

图 4.14 结构体变量 str 的值

4.8 测试代码执行统计

利用 CCS 的剖析(profile)特性,可以显示程序的有关执行统计。

(1) 开始一个新的剖析会话。在 Profiler 菜单中,执行 Start New Session 命令,将弹出如图 4.15 所示的对话框。

(2) 在图 4.15 中,输入一个会话名字。在缺省情况下,会话名字为“MySession”。单击 OK 按钮,会出现如图 4.16 所示的 Profile View 窗口。由于还没有载入 DSP 应用程序,所以 Profile View 窗口出现的是“No program loaded”。

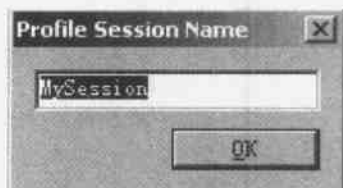


图 4.15 剖析对话框

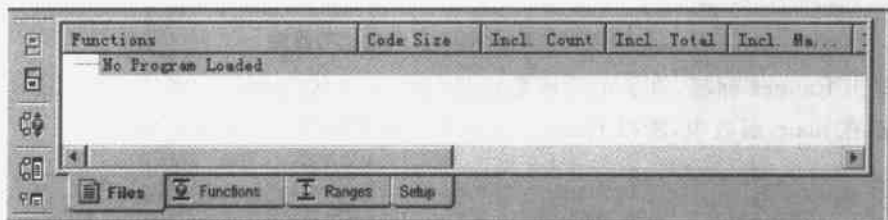


图 4.16 Profile View 窗口

在 File 菜单中,执行 Load Program 命令,载入 hello.out 程序。当载入 hello.out 程序后,剖析会话将自动地执行两个任务:使能剖析时钟和打开剖析会话,如图 4.17 所示。

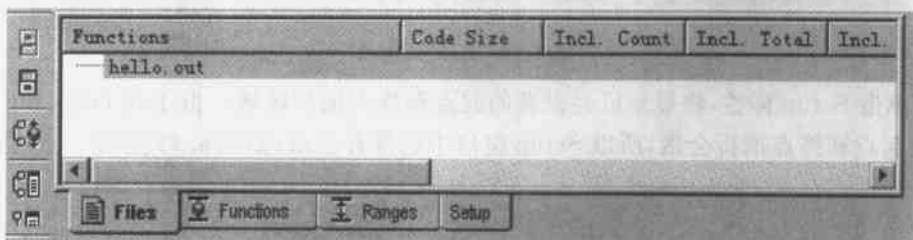


图 4.17 载入 hello.out 程序后的 Profile View 窗口

(3) 在 Files 窗口中显示了剖析会话中创建的所有文件,单击“+”号,可以看到文件下面的函数,如图 4.18 所示。

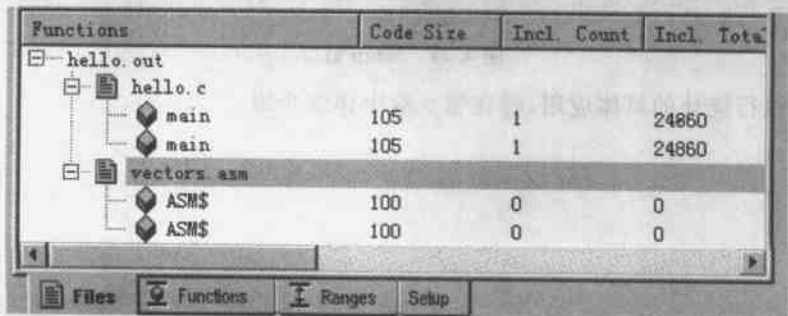


图 4.18 剖析会话中的文件

(4) 单击 Functions 标签, 在 Functions 窗口中显示了剖析的函数。在 hello.out 程序中只有 main 函数, 而该程序把嵌入的汇编代码也作为一个函数进行剖析, 如图 4.19 所示。

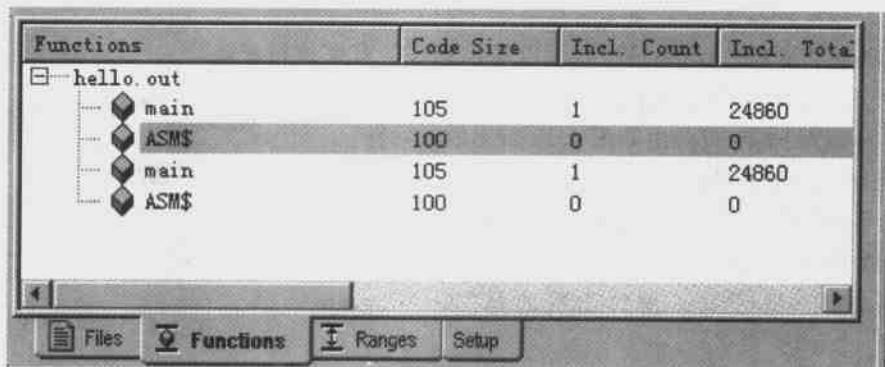


图 4.19 Functions 窗口剖析的函数

(5) 单击 Ranges 标签, 将显示除函数以外的所有区域的剖析。由于在 hello.out 程序中所有代码都在 main 函数中, 所以 Ranges 窗口中没有任何显示, 如图 4.20 所示。

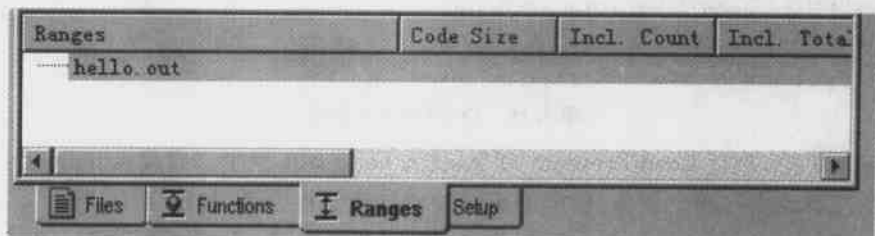


图 4.20 Ranges 窗口

(6) 单击 Setup 标签, 将显示已经设置的起点和终点剖析区域。由于在 hello.out 程序中没有设置起点和终点剖析会话, 所以 Setup 窗口中也没有显示, 如图 4.21 所示。

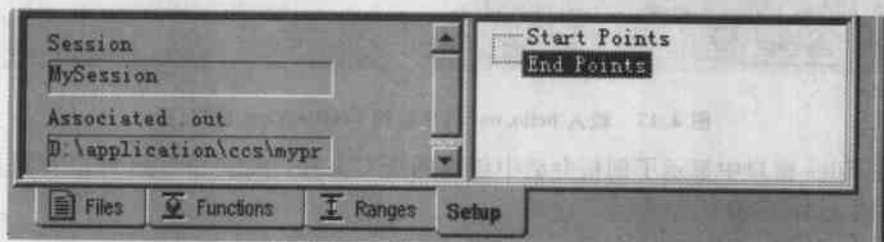


图 4.21 Setup 窗口

关于代码执行统计的具体应用, 将在第 5 章中详细介绍。

第 5 章 CCS 的汇编语言调试实例

本章将用 CCS 的 Simulator 调试一个 DSP 汇编程序。该 DSP 应用程序是计算 $y =$

$\sum_{i=1}^4 a_i x_i$, 其中的数据为

$$\begin{array}{llll} a1 = 1 & a2 = 2 & a3 = 3 & a4 = 4 \\ x1 = 8 & x2 = 6 & x3 = 4 & x4 = 2 \end{array}$$

程序的源代码如下:

```
;/ ***** /
;/ * * /
;/ * EXERCISE. ASM * /
;/ * * /
;/ * ----- BY YIYING IN HUST * /
;/ * * /
;/ * * /
;/ *****

        . title "exercise. asm"
        . mmregs
STACK   . usect "STACK", 10h
        . bss x, 4
        . bss a, 4
        . bss y, 1
        . data
table:   . word 1
        . word 2
        . word 3
        . word 4
        . word 8
        . word 6
        . word 4
        . word 2
        . text

start:   SSBX FRCT
        STM #0, SWWSR
        STM #STACK+10h, SP
        STM #a, AR1
        RPT #7
        MVPD table, *AR1+
```

```

        CALL SUM
end:    B end
SUM:
        STM #a,AR3
        STM #x,AR4
        RPTZ A,#3
        MAC *AR3+,*AR4+,A
        STL A,@y
        RET
        .end

```

假设 CCS 安装在 D:\application 目录下,程序使用的链接命令文件是 lnk.cmd,这里只作了部分修改,并且将其放在 D:\application\ccs\c5400\cgtools\lib 下。如果对存储器没有什么特殊的要求,CCS 的命令文件一般可以通用。lnk.cmd 的源代码如下:

```

/*****
-stack 0x800 /* size of .stack section */
-heap 0x800 /* size of .system section */

MEMORY {
    PAGE 0: /* program memory */

        /* IF THIS IS */
        /* == = == = == = */
        /* OVLY=1 some of internal RAM and/or external */
        /* RAM depending on which C54X this is. */
        /* OVLY=0 external RAM */
    PROG_RAM (RWX) : origin = 0x1400, length = 0x6C00

    PAGE 1: /* data memory, addresses 0-7Fh are reserved */

        /* some (or all) of internal DARAM */
        DATA_RAM (RW): origin = 0x0080, length = 0x30
        DATA_ROM (RW): origin = 0x0100, length = 0x30
        STACK_RAM (RW): origin = 0x0200, length = 0x30

    PAGE 2: /* I/O memory */
        /* no devices declared */
} /* MEMORY */


SECTIONS {
    .text >>> PROG_RAM PAGE 0 /* code */
    .data >>> DATA_ROM PAGE 1 /* initialized data */
    .bss >>> DATA_RAM PAGE 1 /* global & static variables */
    .const >>> DATA_RAM PAGE 1 /* constant data */
    .system >>> DATA_RAM PAGE 1 /* heap */

```



```
.stack >> STACK_RAM PAGE 1 /* stack */
} /* SECTIONS */
```

5.1 载入可执行程序

执行菜单命令 Project→Rebuild All 或者在 Project 工具栏上双击  图标,对工程进行编译、汇编和链接,在 Output 窗口中将显示相关信息。如果汇编失败,返回工程中继续修改程序,直到汇编通过为止。

将 DSP 可执行的目标代码文件 exercise.out (COFF 格式)载入 Simulator 仿真器中(或 Emulator 仿真器中)。注意,Load Symbol 命令是将符号信息载入 DSP 目标系统。这个菜单命令只在 Emulator 硬件仿真器中使用,当调试器不能或没必要加载目标代码 COFF 文件时(如目标代码存放在 ROM 中),应用该命令只清除符号表,而不更改存储器内容和设置程序入口。Add Symbol 命令是在原来的符号表的基础上添加符号信息。该命令和 Load Symbol 不同,它不清除原来已经存在的符号表,只是在其中添加新的符号。另外,也可以用 Reload Program 命令重新加载 DSP 可执行的目标代码 COFF 文件,如果程序未做更改,则该命令只加载可执行程序代码而不加载符号表。

5.2 使用反汇编工具

当 DSP 可执行程序 exercise.out 载入目标系统后,CCS 将自动打开一个反汇编窗口,反汇编窗口根据存储器的内容显示反汇编指令和符号信息。对于每一个汇编指令,反汇编窗口都显示了反汇编指令、反汇编指令的地址和相应的机器代码。要产生相应的反汇编列表,则汇编器从 DSP 目标系统读取程序的机器代码(二进制数据)并反汇编,从 PC 指针开始的地方加上符号信息,有黄色小箭头显示的一行就是当前 PC 指向的代码行,如图 5.1 所示。

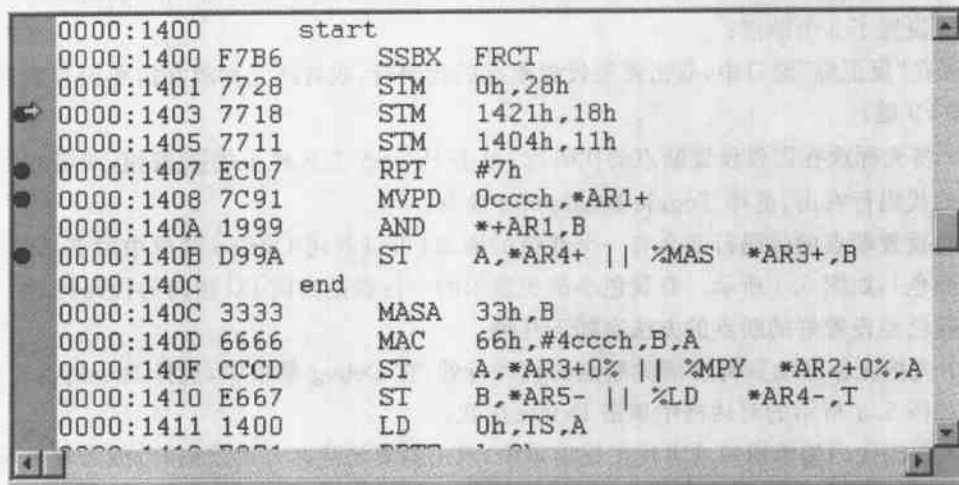


图 5.1 程序 exercise.out 的反汇编窗口

可以在反汇编窗口中改变程序运行的起始地址,方法如下:

(1) 在反汇编窗口中右击,在弹出的菜单中选择 Start Address 命令,将弹出如图 5.2 所示的起始地址设置对话框。



图 5.2 起始地址设置对话框

(2) 在对话框中输入需要执行的起始地址即可。在默认状态下,对话框中的地址是上一次设置的起始地址。

5.3 利用断点调试程序

5.3.1 断点的设置和取消

断点会停止当前程序的执行,当程序停止执行时,程序员可以检查程序的运行状态,查看或改变变量的值,检查堆栈和子程序的调用情况等。


在设置程序的断点时应注意以下两点:

- (1) 不要将断点设置在任何延迟分支或调用的指令处;
- (2) 不要将断点设置在重复的块指令倒数的第 1、2 行的指令处。

可以在源代码编辑窗口或者反汇编窗口中设置断点,断点设置后,保存在项目的工作区中。设置断点的方法有如下几种:

(1) 在 Debug 菜单中,选择 Breakpoint 命令,将会弹出如图 5.3 所示的设置断点对话框,其中已经设置了 3 个断点;

(2) 在“反汇编”窗口中,双击需要设置断点的代码行,或者将光标放在需要设置断点的代码行,按 F9 键;


(3) 将光标放在需要设置断点的代码行,单击 Project 工具栏上的  按钮,或者在需要设置断点的代码行右击,选择 Toggle Breakpoint 命令。

此时设置断点的代码行前会有一个红色的圆点(可以利用 Option 菜单中的 Font 命令来改变其颜色),如图 5.4 所示。有黄色小箭头显示的一行就是当前 PC 指向的代码行。

清除已经设置好的断点的方法有如下几种:

(1) 先把光标移动到需要清除断点的代码行处,在 Debug 菜单中,选择 Breakpoint 命令,在弹出如图 5.3 所示的对话框中单击 Delete 按钮;

(2) 在源代码编辑窗口或者反汇编窗口中,双击需要清除断点的代码行,或者将光标放在需要清除断点的代码行,按 F9 键;

(3) 将光标放在需要清除断点的代码行,单击 Project 工具栏上的  按钮,或者在需要清除断点的代码行右击,选择 Toggle Breakpoint 命令;

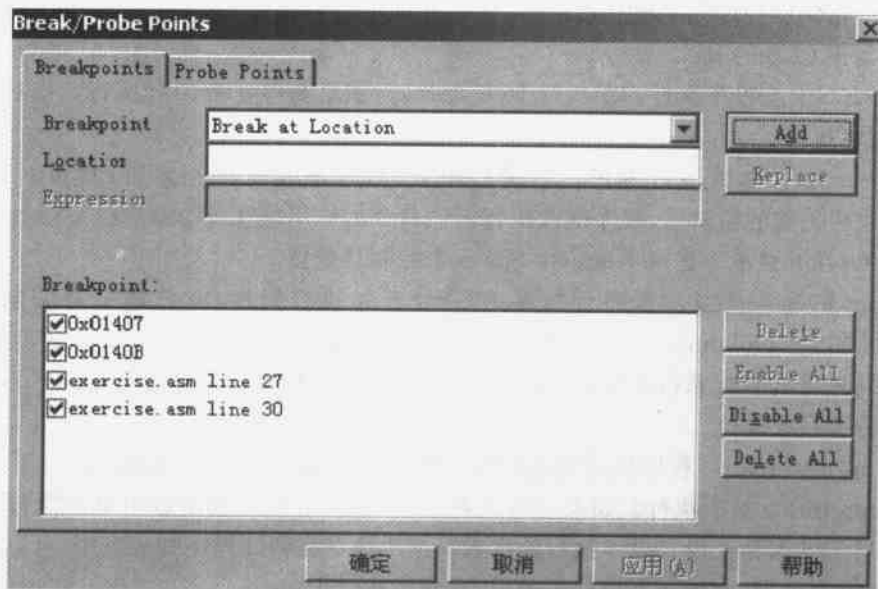



图 5.3 设置断点对话框

(4) 在 Debug 菜单中, 选择 Breakpoint 命令, 在弹出的如图 5.3 所示的对话框中单击 Delete All 按钮, 删除所有断点, 或者单击 Project 工具栏上的  按钮, 取消所有断点。

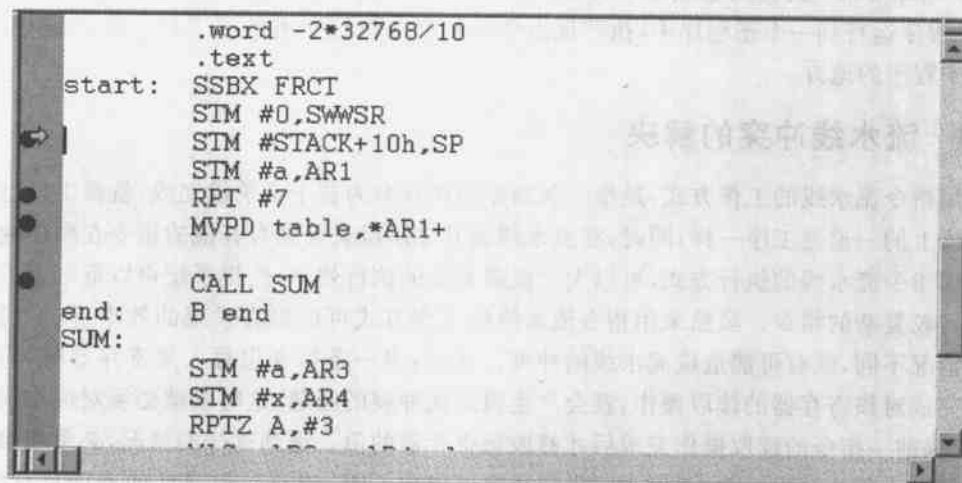


图 5.4 设置好的断点

5.3.2 程序的执行

设置断点是调试程序的必要手段, 这对了解程序的运行情况是非常有帮助的。运行程序有以下几种方式:

(1) Run 从当前程序计数器(PC)执行程序, 碰到断点时暂停, 这时候可以利用单步执行程序等方式调试程序。执行 Run 命令时, 如果没有断点, 程序会直接执行到最后一条指令。

一般情况下,在保证程序完全没有问题时,把可执行代码加载到 DSP 目标 CPU 后,就可以直接运行,相当于 CCS 的 RUN 命令。

(2) Halt 中止程序执行。在执行 Run 命令后,如果要停止程序的执行,可以选择 Halt 命令停止程序执行。

(3) Animate 动画运行程序。当碰到断点时,程序暂时停止运行;在更新未与任何 Probe Point 相关联的窗口后,程序继续执行。该命令的作用是在每个断点处显示处理器的状态,可以在 Option 菜单下选择 Animate Speed 来控制其速度。

(4) Run Free 从当前程序计数器(PC)处开始执行程序,将忽略所有的断点(包括 Breakpoints 和 Probe Points)。该命令在 Simulator 下无效,在使用 Emulator 进行仿真调试时,该命令将断开与目标系统板的连接,因此可以移走 JTAG 电缆。在运行 Run Free 指令时,将对 DSP 目标系统复位。

(5) Run to Cursor 程序执行到光标处。光标所在行必须为有效的代码行。

(6) Step Into 单步执行。如果运行到调用函数处,将跳入到函数中单步执行。可以用 Multiple Operation 命令设置单步执行的次数。这个命令在调试程序时最有用,可以边单步运行程序边查看寄存器、存储器和堆栈等的值,直到正确为止。

(7) Step Over 单步执行。与 Step Into 不同的是,为了保护处理器的流水线操作,该命令后的若干条延迟指令或调用指令将同时被执行。如果运行到函数调用处,将直接执行完整个函数的功能而不能跳入到函数内部单步执行,除非在函数内部设置了断点。

(8) Step Out 跳出函数或子程序执行。当使用 Step Into 或 Step Over 单步执行指令时,如果程序运行到一个子程序中,执行该命令将使程序执行完函数或子程序后,回到调用该函数或子程序的地方。

5.3.3 流水线冲突的解决

采用指令流水线的工作方式,是指一条指令的执行分为若干个阶段完成,就像工厂里的生产流水线上的一道道工序一样;同时,在流水线的其他阶段又分别有其他的指令在顺序地执行着。采用指令流水线的执行方式,可以大大提高系统的执行效率,使得系统可以低延迟或无延迟地执行较复杂的指令。虽然采用指令流水线的工作方式可以提高系统的效率,但不同的指令执行情况不同,就有可能造成流水线的冲突。比如:当一条指令想写入某寄存器时,前一指令还未完成对该寄存器的读取操作,就会产生流水线冲突的问题,这时候就必须对流水线进行保护,确保前一指令的读取操作完成后才修改该寄存器的值。遇到这样的情况,必须在第二条指令之前加入等待延迟。在 C54X 中,有些冲突可以由 CPU 通过延迟寻址的方法自动缓解,但有些冲突是不能自动解决的,必须由程序员重新安排指令顺序或者插入空指令(NOP)加以解决。

一般来说,如果 C54X 源程序是采用 C 语言编写的,经过编译生成的代码是没有流水线冲突的;如果是采用汇编语言编写的,凡是 CALU 操作,或者在初始化期间就对 MMR 进行设置,也不会发生流水线冲突。因此,大多数 C54X 程序是不需要对流水线冲突问题特别关注的,只有某些 MMR 的写操作才需要注意。

在 CCS2 中,对汇编语言源程序进行编译时,如果对 MMR 写操作发生流水线时序上的冲突,将会自动发出警告,但需要对 CCS 的编译环境进行设置,其设置方法如下:

在 Project 菜单中选择 Build Options... 命令,将弹出如图 5.5 所示的设置对话框。单击 Compiler 页,在 Category 域里选择 Diagnostics 选项,选中 Warn on Pipeline Conflicts 选项即可。

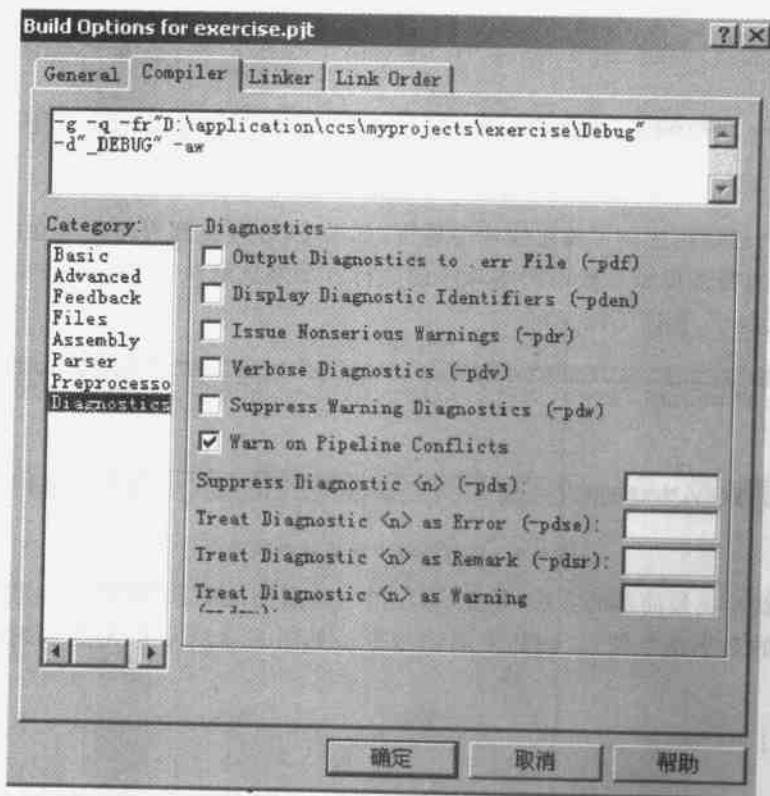


图 5.5 流水线时序冲突警告设置对话框

在例子 exercise.asm 中,添加如下两行代码:

```
STL A,AR1
LD *AR1+,T
```

这两条指令对程序计算 $y = \sum_{i=1}^4 a_i x_i$ 没有用处,放在程序中只是为了演示如何处理流水线冲突问题。

这两条指令的流水线执行如图 5.6 所示,其中 W 表示写到 AR1, R 表示指令需要读取 AR1 中的值。指令 STL 是在流水线的执行阶段进行写 AR1 操作的,而指令 LD 是在寻址阶段生成地址,LD 指令需要根据 AR1 进行间接寻址读数时,STM 指令还没有为 AR1 准备好数据。这样,如果不采取措施,程序必然会产生流水线冲突。

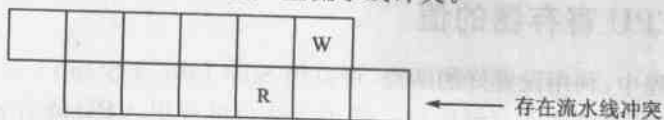


图 5.6 流水线执行时序图

在 CCS 的编译环境设置了流水线冲突警告选项后,重新编译程序,Output 窗口将给出这两条指令流水线冲突提示,如图 5.7 所示。

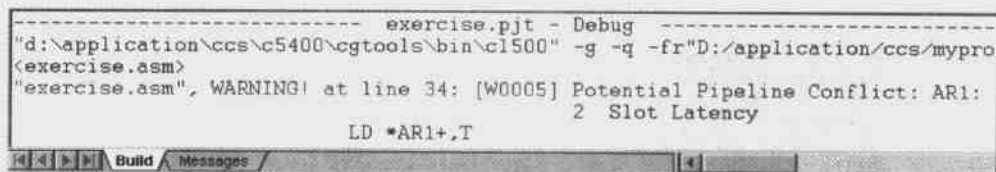


图 5.7 流水线冲突提示

CCS 的运行环境还提供了流水线冲突警告,如果 CCS 检查到程序中存在时序上的冲突,程序 CCS 编译的警告提示一般不影响程序的执行。如果这时执行 exercise.out,将出现如图 5.8 所示的错误提示,程序不能运行。

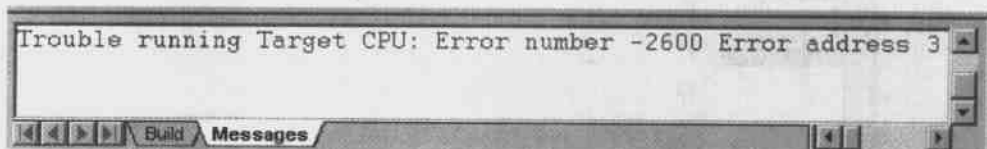


图 5.8 运行环境中流水线冲突提示

解决流水线冲突最简单的方法,是在 STL 指令后面加入两条 NOP 指令或者加入任何一条与程序无关的 2 字指令即可,例如下面的指令。如图 5.9 所示是无流水线冲突的执行时序图。

```
STL A,AR1
NOP
NOP
LD *AR1+,T
```

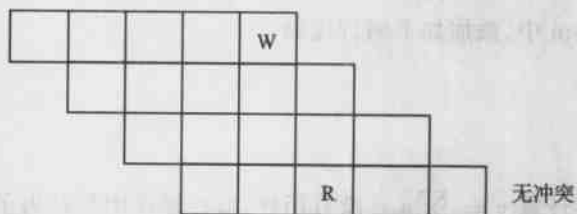


图 5.9 无流水线冲突的执行时序图

加两条 NOP 指令后,再重新编译,程序就可以正常执行了。至于为什么要加两条 NOP 指令而不是加入一条或者三条,这是由流水线指令的等待周期决定的,详细内容可参考其他有关书籍。

5.3.4 查看 CPU 寄存器的值

在程序执行过程中,利用设置好的断点,可以用 Step Into 或 Step Over 命令单步执行、观察和修改 CPU 寄存器及外围寄存器的值。在单步执行过程中,CPU 寄存器和外围寄存器的值是随时随指令执行进行更新的。

运行程序前,需要执行 Reset 命令将 DSP 目标 CPU 复位。复位时,DSP 处理器从 FF80h 处开始执行程序,同时下列寄存器和状态位被置成初始值:

PC=FF80 PMST=FFE0 DRAM=0 ASM=0 SXM=1

ARP=0 CMPT=0 INTM=1 FRCT=0 OVM=0

OVA0 XF=1 DP=0 C=1 OVLY=0 TC=1 C16=0

HM=0 INTM=1 CPL=0 AVIS=0 BRAF=0

如图 5.10 所示,其他的寄存器值保持以前的值不变,或者是不确定状态。

PC = 0000FF80	INTM = 1	
SP = 0514	IMR = 0123	
A = 000000007B	IFR = 0000	OVA = 0
B = 0000003230	IPTR = FF80	OVV = 0
T = 000C		OVM = 0
TRN = 0123		SXM = 1
ST0 = 1800	ARO = 0000	C16 = 0
ST1 = 2900	AR1 = 140D	FRCT = 0
PMST = FFE0	AR2 = 0000	CMPT = 0
DP = 0000	AR3 = 1B55	CPL = 0
ASM = 00	AR4 = 7E51	XF = 1
BRAF = 0	AR5 = FFEE	HM = 0
BRC = 0123	AR6 = 0000	MP/MC = 1
RSA = 0012	AR7 = 0000	OVLY = 1
REA = 0123	BK = 0123	AVIS = 0
TC = 1	ARP = ARO	DROM = 0
C = 1		

图 5.10 CPU 复位时寄存器和状态位的初始值

开始运行程序,在 Debug 菜单中执行 Run 命令,程序在第一个断点处停止,如图 5.11 所示。这时,PC 的值、FRCT 的值和 C 的值都发生了变化,如图 5.12 所示。PC 的值装入程序的入口地址,由于在配置文件 lnk.cmd 中设置了“PROG_RAM (RWX) : origin = 0x1400”,也就是程序的入口地址是 0x1400。第一条指令 SSBX FRCT 将状态位 FRCT 置位为 1,第二条指令 STM #0,SWWSR 将状态位 C 复位为 0。

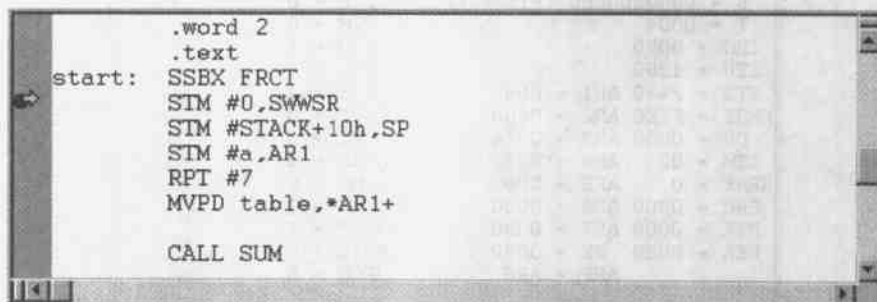


图 5.11 程序的执行状况

按 F8 键,单步运行程序,这时程序运行到下一条指令:

STM #STACK+10h,SP

这时,PC 的值加 1,SP 的值变成 0x1421,这是因为 lnk.cmd 中设置的 STACK 的起始地

PC = 00001401	
XPC = 0	
A = 0000001000	
B = 0000000000	
T = 0000	
TRN = 0000	SP = 0000
ST0 = 1000	ARO = 0000
ST1 = 2940 AR1 = 0000	OVA = 0
PMST = FFE0 AR2 = 0000	OVb = 0
DP = 0000 AR3 = 0000	OVM = 0
ASM = 00 AR4 = 0000	SXM = 1
BRAF = 0 AR5 = 0000	C16 = 0
BRC = 0000 AR6 = 0000	FRCT = 1
RSA = 0000 AR7 = 0000	CMPT = 0
REA = 0000 BK = 0000	CPL = 0
INIM = 1 ARP = ARO	XF = 1
IMR = 0000	HM = 0
IFR = 0000	MP/MC = 1
IPTR = FF80	OVLY = 1
TC = 1	AVIS = 0
C = 0	DROM = 0

图 5.12 PC、FRCT 和 C 的值发生的变化

址为 0x0080。

DATA_RAM (RW): origin = 0x0080, length = 0x100

而在程序 exercise.asm 中,宏汇编指令:

```
STACK .usect "STACK",10h
```

给堆栈留出了 10 B 的空间,SP 的值指向 0x1410,执行完指令“STM #STACK+10h,SP”后,SP 的值便再加上 10h,故 SP=0x0099,如图 5.13 所示。

PC = 00001403	INIM = 1	SP = 0099	TC = 1
XPC = 0	IMR = 0000	ARO = 0000	C = 0
A = 0000001000	IFR = 0000	OVA = 0	
B = 0000000000	IPTR = FF80	OVb = 0	
T = 0004		OVM = 0	
TRN = 0000		SXM = 1	
ST0 = 1000		C16 = 0	
ST1 = 2940 AR1 = 0080		FRCT = 1	
PMST = FFE0 AR2 = 0000		CMPT = 0	
DP = 0000 AR3 = 0084		CPL = 0	
ASM = 00 AR4 = 0088		XF = 1	
BRAF = 0 AR5 = 0000		HM = 0	
BRC = 0000 AR6 = 0000		MP/MC = 1	
RSA = 0000 AR7 = 0000		OVLY = 1	
REA = 0000 BK = 0000		AVIS = 0	
ARP = ARO		DROM = 0	

图 5.13 SP 的值

再按 F8 键,单步执行到下一条指令:

```
STM #a, AR1
```


该指令将 a 的地址,也就是 0x080 送给 AR1,同时 PC 的值也变成 0x00001405,如图 5.14 所示。

PC = 00001405	INIM = 1	SP = 0099	TC = 1
XPC = 0	IMR = 0000	ARO = 0000	C = 0
A = 0000001000	IFR = 0000	OVA = 0	
B = 0000000000	IPTR = FF80	OVB = 0	
T = 0004		OVM = 0	
TRN = 0000		SXM = 1	
ST0 = 1000		C16 = 0	
ST1 = 2940	AR1 = 0080	FRCT = 1	
PMST = FFE0	AR2 = 0000	CMPT = 0	
DP = 0000	AR3 = 0000	CPL = 0	
ASM = 00	AR4 = 0000	XF = 1	
BRAF = 0	AR5 = 0000	HM = 0	
BRC = 0000	AR6 = 0000	MP/MC = 1	
RSA = 0000	AR7 = 0000	OVLY = 1	
REA = 0000	BK = 0000	AVIS = 0	
	ARP = ARO	DROM = 0	

图 5.14 a 的地址送给 AR1

继续按 F8 键,单步执行下一条指令:

RPT #7

该指令表示将下一条指令重复执行 8 次。执行这条指令后,CPU 寄存器的值没发生变化。继续单步执行,下一条指令被重复执行 8 次,也就是把程序存储器 table 地址单元所存放的 8 个数据送到 AR1 寄存器所指向的数据存储单元中暂时存放起来,以供后面的计算使用。同时,每重复执行一次数据传输指令,AR1 的值加 1;重复执行 8 次,AR1 的值变成 0x1408,而 PC 也执行到 0x0000140A。应该注意的是,由于调用了子程序 SUM,用了堆栈来保存一些临时变量,所以 SP 指针向上移动了,变成 0x0098,如图 5.15 所示。

PC = 0000140A	INIM = 1	SP = 0098	TC = 1
XPC = 0	IMR = 0000	ARO = 0000	C = 0
A = 0000001000	IFR = 0000	OVA = 0	
B = 0000000000	IPTR = FF80	OVB = 0	
T = 0004		OVM = 0	
TRN = 0000		SXM = 1	
ST0 = 1000		C16 = 0	
ST1 = 2940	AR1 = 0088	FRCT = 1	
PMST = FFE0	AR2 = 0000	CMPT = 0	
DP = 0000	AR3 = 0000	CPL = 0	
ASM = 00	AR4 = 0000	XF = 1	
BRAF = 0	AR5 = 0000	HM = 0	
BRC = 0000	AR6 = 0000	MP/MC = 1	
RSA = 0000	AR7 = 0000	OVLY = 1	
REA = 0000	BK = 0000	AVIS = 0	
	ARP = ARO	DROM = 0	

图 5.15 AR1 的值

继续单步执行,如果按 F8 键,执行 Step Into 命令,那么程序将进入到子程序 SUM 中,可

以进一步追踪执行情况。如果按 F10 键, 执行 Step Out 命令, 程序直接执行完整个 SUM 子程序后, 跳到语句:

```
end;      B end
```

再按 F8 键, 进入子程序内部, 执行语句:

```
STM #a, AR3
```

将变量 a 的地址送给 AR3, 使 AR3 的值发生了变化, 如图 5.16 所示。

PC = 0000140E	INIM = 1	SP = 0098	TC = 1
XPC = 0	IMR = 0000	ARO = 0000	C = 0
A = 0000001000	IFR = 0000	OVA = 0	
B = 0000000000	IPTR = FF80	OVb = 0	
T = 0004		OVm = 0	
TRN = 0000		SXM = 1	
ST0 = 1000		C16 = 0	
ST1 = 2940	AR1 = 0088	FRCT = 1	
PMST = FFE0	AR2 = 0000	CMPT = 0	
DP = 0000	AR3 = 0080	CPL = 0	
ASM = 00	AR4 = 0000	XF = 1	
BRAF = 0	AR5 = 0000	HM = 0	
BRC = 0000	AR6 = 0000	MP/MC = 1	
RSA = 0000	AR7 = 0000	OVLY = 1	
REA = 0000	BK = 0000	AVIS = 0	
	ARP = ARO	DROM = 0	

图 5.16 AR3 的值

继续单步执行, 下一条语句是:

```
STM #x, AR4
```

将变量 x 的地址送给 AR4, AR4 的值发生了变化, 如图 5.17 所示。

PC = 00001410	INIM = 1	SP = 0098	TC = 1
XPC = 0	IMR = 0000	ARO = 0000	C = 0
A = 0000001000	IFR = 0000	OVA = 0	
B = 0000000000	IPTR = FF80	OVb = 0	
T = 0004		OVm = 0	
TRN = 0000		SXM = 1	
ST0 = 1000		C16 = 0	
ST1 = 2940	AR1 = 0088	FRCT = 1	
PMST = FFE0	AR2 = 0000	CMPT = 0	
DP = 0000	AR3 = 0080	CPL = 0	
ASM = 00	AR4 = 0084	XF = 1	
BRAF = 0	AR5 = 0000	HM = 0	
BRC = 0000	AR6 = 0000	MP/MC = 1	
RSA = 0000	AR7 = 0000	OVLY = 1	
REA = 0000	BK = 0000	AVIS = 0	
	ARP = ARO	DROM = 0	

图 5.17 AR4 的值

继续单步执行, 下一条语句是:

```
RPTZ A, #3
```

该语句表示将下一条指令连续执行4次,同时将寄存器A的值清0。

继续单步执行,下一条语句是:

MAC *AR3+,*AR4+,A

将AR3和AR4所指向的地址的值相乘、相加,并且把结果存入A中,把AR3指向的地址中的数据(1)送给T寄存器,则T=1,同时AR3和AR4的值加1。由于是单步执行,这里只执行了一次乘法和加法运算,并且A的值还保持为0,可以看到PC的值也相应地发生了变化,如图5.18所示。

PC = 00001414	INTM = 1	SP = 0098	TC = 1
XPC = 0	IMR = 0000	ARO = 0000	C = 0
A = 0000000000	IFR = 0000	OVA = 0	
B = 0000000000	IPTR = FF80	OVB = 0	
T = 0001		OVM = 0	
TRN = 0000		SXM = 1	
ST0 = 1000		C16 = 0	
ST1 = 2940	AR1 = 0088	FRCT = 1	
PMST = FFE0	AR2 = 0000	CMPT = 0	
DP = 0000	AR3 = 0082	CPL = 0	
ASM = 00	AR4 = 0086	XF = 1	
BRAF = 0	AR5 = 0000	HM = 0	
BRC = 0000	AR6 = 0000	MP/MC = 1	
RSA = 0000	AR7 = 0000	OVLY = 1	
REA = 0000	BK = 0000	AVIS = 0	
	ARP = ARO	DROM = 0	

图 5.18 第一次乘加运算后 AR3、AR4 的值

继续单步执行,完成4次乘法和加法运算,运算结果为40,也就是十六进制的0x0000000028放入A中,AR3和AR4的值也加4,并且把AR3寄存器所指向的地址所存放的值,也就是4,送给T寄存器,同时SP返回,如图5.19所示。

PC = 00001415	INTM = 1	SP = 0099	
XPC = 0	IMR = 0000	ARO = 0000	
A = 0000000028	IFR = 0000	OVA = 0	
B = 0000000000	IPTR = FF80	OVB = 0	
T = 0004		OVM = 0	
TRN = 0000		SXM = 1	
ST0 = 1000		C16 = 0	
ST1 = 2900	AR1 = 0088	FRCT = 0	
PMST = FFE0	AR2 = 0084	CMPT = 0	
DP = 0000	AR3 = 0088	CPL = 0	
ASM = 00	AR4 = 7714	XF = 1	
BRAF = 0	AR5 = 0000	HM = 0	
BRC = 0000	AR6 = 0000	MP/MC = 1	
RSA = 0000	AR7 = 0000	OVLY = 1	
REA = 0000	BK = 0000	AVIS = 0	
TC = 1	ARP = ARO	DROM = 0	
C = 0			

图 5.19 运算结果

5.3.5 查看内存数据

查看内存数据是程序调试中常用的方法。在 CCS 中可以用许多方式来查看存储器数据单元的内容。

查看数据时,选择 View 菜单中的 Memory 命令,会弹出如图 5.20 所示的对话框。

- Address 表示所要查看数据的开始单元,输入相关的数据,可以快速地跳到需要查看的存储器单元。输入的数据必须以 0x 开头。
- Q-Value 表示所要查看的小数点的位置,其值可以在 0~31 之间。如果输入 0,则存储器中的数据将以整数的方式显示;如果输入 15,数据将以纯小数的方式显示。
- Format 表示所查看的数据的格式,CCS 提供多个选择,包括: C-style hex、Hex、Signed - integer、Unsigned - integer、Binary、Character、Packed - character、Floating - point 和 Exponential - float。

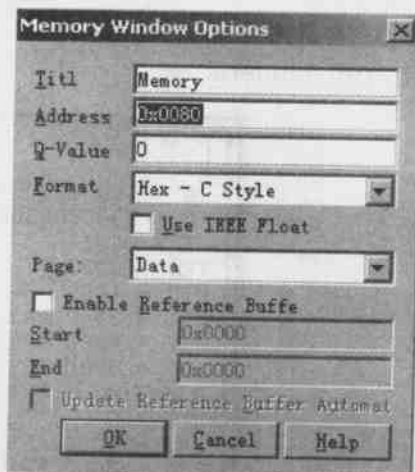


图 5.20 Memory 对话框

经常用的有十六进制数、十六进制有符号数、十六进制无符号数、二进制数、32 位无符号数和 32 位有符号数等。对于 32 位数据的显示,CCS 将连续的两个 16 位的字合并成一个 32 位的数据,高位在前,低位在后,同时小数点的格式可以选择大于 16 的数值。在没有用到小数的数据格式中,例如十六进制数据、二进制数据或者字符数据及小数点必须选择为 0;否则将以小数显示数据,将整型数据也扩展成小数。

如图 5.21 所示为对一段存储器单元的各种不同数据格式的显示结果,从上到下依次显示的格式是:十六进制无符号数据、32 位无符号数据和二进制数据显示。

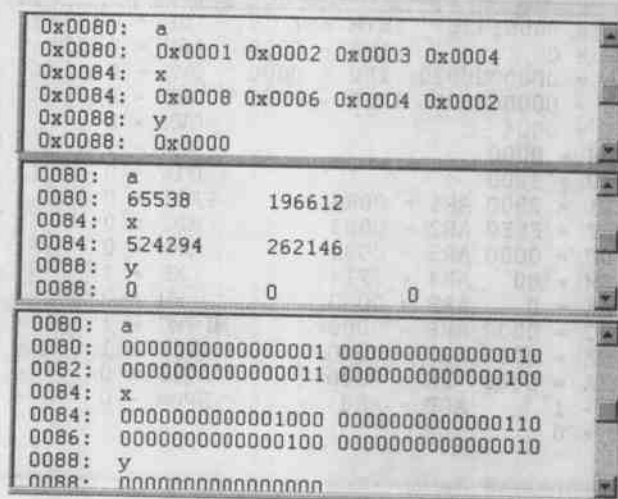


图 5.21 不同数据格式的存储器单元显示结果

下面以工程 exercise. pj1 为例来进行调试,在调试程序时,以十六进制无符号格式显示存储器单元的数据。

由于命令链接文件 lnk.cmd 将初始化的数据安排到 0x0100 开始的存储段:

```
DATA_ROM (RWX): origin = 0x0100, length = 0x30
.data >> DATA_ROM PAGE 1
```

而在程序 exercise.asm 中,设置已经初始化的数据位于变量 table 地址开始的地方:

```
.data
* table: .word 1
        .word 2
        .word 3
        .word 4
        .word 8
        .word 6
        .word 4
        .word 2
```

所以变量 table 的地址位于 0x0100。

选择 File 菜单中的 Load Program 命令,将 DSP 可执行的目标代码文件 exercise.out (COFF 格式)载入 Simulator 仿真器中(或 Emulator);选择 View 菜单中的 Memory 命令,在 Address 栏里填入 0x0100;在 Q-Value 栏里填入 0;在 Format 栏里填入 Hex - C Style,如图 5.22 所示。

单击 OK 按钮,可以看到已经初始化的数据:

```
a1 = 1  a2 = 2  a3 = 3  a4 = 4
x1 = 8  x2 = 6  x3 = 4  x4 = 2
```

数据在存储器中的位置如图 5.23 所示。

按 F8 键,单步执行指令:

```
MVPD table, *AR1+
```

如图 5.24 所示。

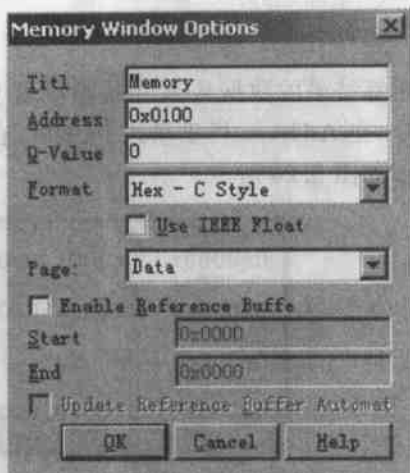


图 5.22 Memory 对话框设置

0x0100:	table				
0x0100:	0x0001	0x0002	0x0003	0x0004	
0x0104:	0x0008	0x0006	0x0004	0x0002	
0x0108:	edata				
0x0108:	0x0000	0x0000	0x0000	0x0000	
0x010C:	0x0000	0x0000	0x0000	0x0000	

图 5.23 初始化的数据在存储器中的位置

该指令将放在程序存储器 table 单元的已经初始化的数据移动到寄存器 AR1 所指向的数据存储器位置,也就是变量 a 所在的位置,如图 5.25 所示。

单步执行程序,4 次乘法和加法运算完成,运算结果为 40,也就是十六进制的

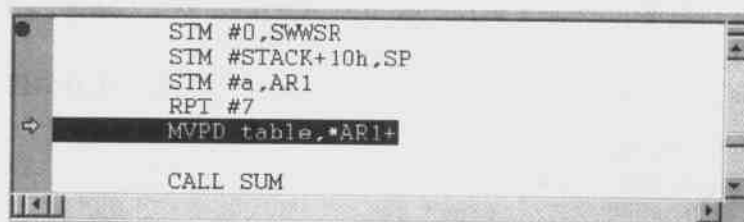


图 5.24 单步执行

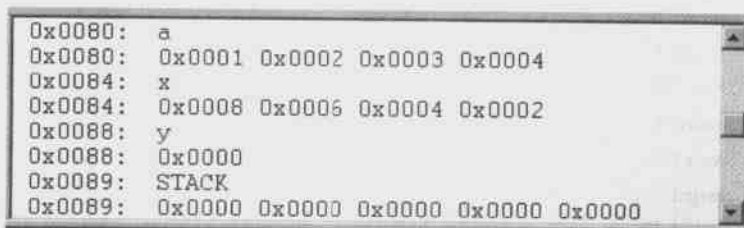


图 5.25 移动数据到数据存储器

0x0000000028, 结果放入 A 中, AR3 和 AR4 的值也加 4。再执行指令:

STL A, @y

把运算结果的数据从 A 中传给变量 y, 再来观察变量 y 的值。选择 View 菜单中的 Memory 命令, 在 Address 栏里填入 0x0088, 在 Q-Value 栏里填入 0, 在 Format 栏里填入 C-style hex, 如图 5.26 所示。

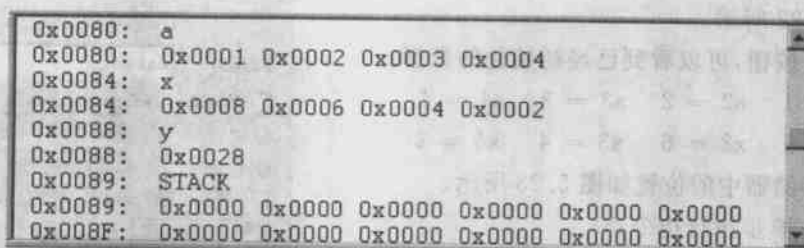


图 5.26 变量 y 的值

同时, 可以看 CPU 寄存器在存储器中的映像。CPU 寄存器在存储器中的映像位置是 0~1E。这时, CPU 各寄存器的值如图 5.27 所示。

其中, 状态寄存器 ST0=1000, ST1=2940, A=0000000028, T=0004, AR0=0000, AR1=0088, AR2=0000, AR3=0084, AR4=0088 及 SP=0099 等。

选择 View 菜单中的 Memory 命令, 在 Address 栏里填入 0x0000, 在 Q-Value 栏里填入 0, 在 Format 栏里填入 C-style hex。这时, CPU 各寄存器的值如图 5.28 所示。

其中, 地址 0x0006 存放的是寄存器 ST0 的值, 为 0x1000; 地址 0x0007 存放的是寄存器 ST1 的值, 为 0x2940; 地址 0x0008 存放的是寄存器 AL 的值, 为 0x0028, 也就是寄存器 A 的低位; 地址 0x0009 存放的是寄存器 AH 的值, 为 0x0000, 也就是寄存器 A 的高位; 地址 0x000A 存放的是寄存器 AG 的值, 为 0x0000, 也就是寄存器 A 的保护位……

PC = 0000140C	INTM = 1	SP = 0099
XPC = 0	IMR = 0000	ARO = 0000
A = 0000000028	IFR = 0000	OVA = 0
B = 0000000000	IPTR = FF80	OVB = 0
T = 0004		OVM = 0
TRN = 0000		SXM = 1
ST0 = 1000		C16 = 0
ST1 = 2940	AR1 = 0088	FRCT = 1
PMST = FFEO	AR2 = 0000	CMPT = 0
DP = 0000	AR3 = 0084	CPL = 0
ASM = 00	AR4 = 0088	XF = 1
BRAF = 0	AR5 = 0000	HM = 0
BRC = 0000	AR6 = 0000	MP/MC = 1
RSA = 0000	AR7 = 0000	OVLY = 1
REA = 0000	BK = 0000	AVIS = 0
TC = 1	ARP = ARO	DROM = 0
C = 0		

图 5.27 CPU 各寄存器的值

0x0000:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x0006:	0x1000	0x2940	0x0028	0x0000	0x0000	0x0000
0x000C:	0x0000	0x0000	0x0004	0x0000	0x0000	0x0088
0x0012:	0x0000	0x0084	0x0088	0x0000	0x0000	0x0000
0x0018:	0x0099	0x0000	0x0000	0x0000	0x0000	0xFFEO
0x001E:	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x0024:	0xEBD3	0xFFFF	0x0000	0x0000	0x0000	0x0002
0x002A:	0x0000	0x0000	0x0000	0x0001	0x0001	0x0001

图 5.28 CPU 寄存器在存储器中的映像

5.3.6 查看变量的值

CCS 的变量观察窗口可以用来检查和编辑变量的值,可以用不同的格式显示数值,可以显示数组、结构或指针等包含多个元素的变量。在程序 EXERCISE.ASM 中,定义了 x、y、a、table、STACK、start、end 和 SUM 几个地址标号,利用变量观察窗口可以观察其中的值。

选择 View 菜单中的 Watch Window 命令,将弹出变量观察窗口,在 Name 栏里输入 a、x、y、table、STACK、start、end 和 SUM 几个地址标号,将分别显示其内容,如图 5.29 所示。

Name	Value	Type	Radix
<input checked="" type="checkbox"/> a	0x00000080	void	hex
<input checked="" type="checkbox"/> x	0x00000084	void	hex
<input checked="" type="checkbox"/> y	0x00000088	void	hex
<input checked="" type="checkbox"/> table	0x00000100	void	hex
<input checked="" type="checkbox"/> STACK	0x00000089	void	hex
<input checked="" type="checkbox"/> start	0x00001400	void	hex
<input checked="" type="checkbox"/> end	0x0089	int *	hex
<input checked="" type="checkbox"/> SUM	0x0000140E	void	hex
<input type="checkbox"/>			

☒ Watch Locals ☒ Watch 1

图 5.29 变量观察窗口

可以看到, a、x、y、table、STACK、start、end 和 SUM 的地址标号的值分别为 0x00000080、0x00000084、0x00000088、0x00000100、0x00000089、0x00001400、0x0089 和 0x0000140E。

5.4 剖析点的调试

在 Profiler 菜单中, 执行 Start New Session 命令, 将弹出如图 5.30 所示的对话框。首先, 在 Session Name 栏里输入会话名称: exercise。

由于 DSP 程序 exercise.asm 中没有函数, 所以就利用 Ranges、Start Point 与 End Point 来统计程序块执行的信息。

在 exercise.asm 程序中, 选择从指令“STM #a, R3”到“MAC *AR3+, *AR4+, A”的代码行, 然后右击; 选择 Profile Range→in exercise Range 命令, 如图 5.31 所示。

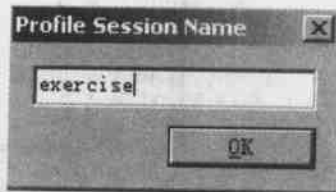


图 5.30 Profile 会话名称

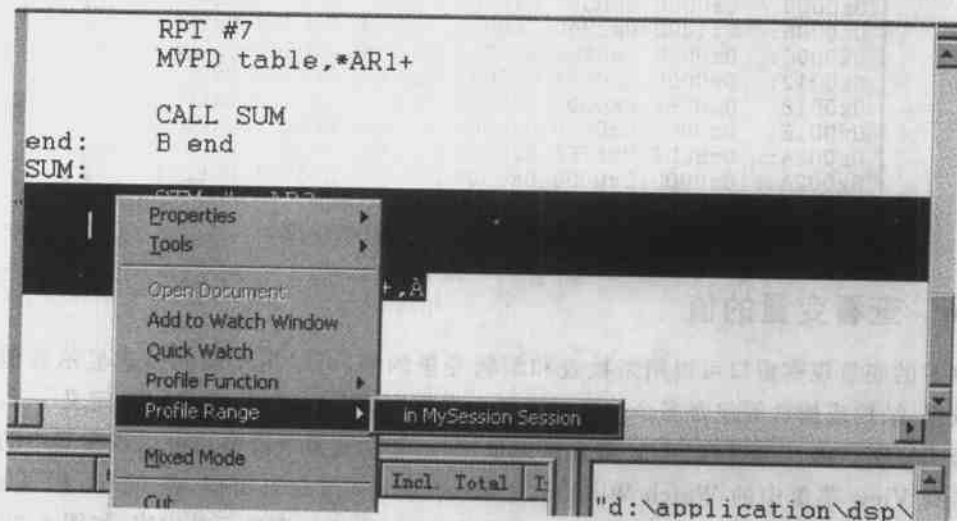


图 5.31 选择代码行设置剖析区域

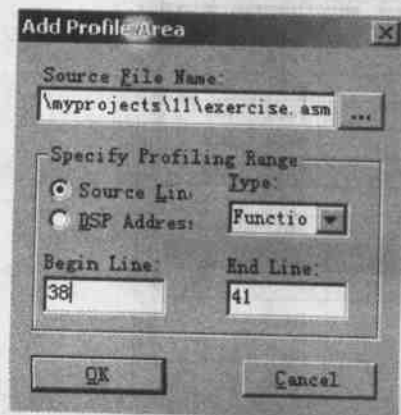



图 5.32 剖析区域设置对话框

这样, 从代码行“STM #a, R3”到“MAC *AR3+, *AR4+, A”的范围就被设置成剖析区域。也可以单击剖析工具栏上的  图标, 将弹出如图 5.32 所示的剖析区域设置对话框。

在 Begin Line 栏里输入 38, 在 End Line 栏里输入 41, 就相当于如图 5.31 所示的选择代码行设置。单击 OK 按钮, 则完成剖析区域设置, 如图 5.33 所示。

在全速运行或单步执行程序时, Profile Clock 可计算处理器指令周期或其他事件触发次数。Profile Clock 的设置、使能和查看的设置菜单 Profiler 下面。在菜单 Profiler 中选择 Clock Setup 命令时, 将弹出如图 5.34

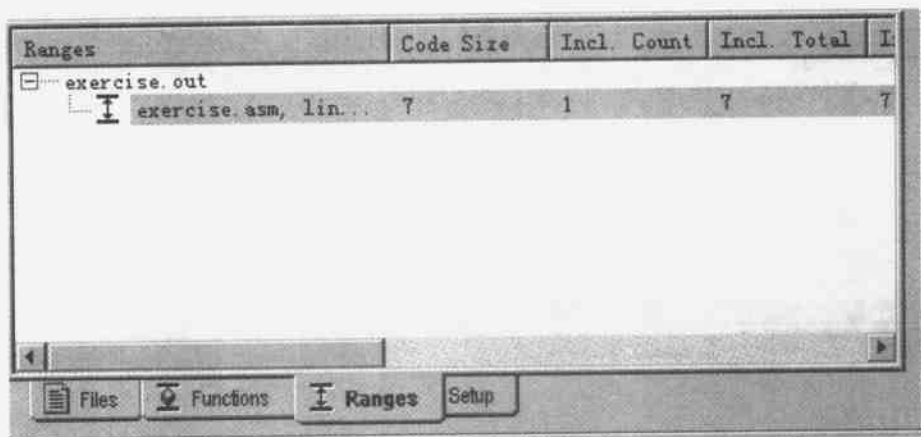


图 5.33 设置好的剖析区域


所示的时钟设置对话框。



图 5.34 时钟设置对话框

在 Count 下拉选择项目中,可以选择 CPU Cycle 指令周期、流水线时钟、堆栈调用次数、指令分支执行次数和中断次数等,以便让 Clock 记录不同的项目周期数。在 exercise.pjt 项目中,不改变时钟设置。

在菜单 Profiler 中选择 View Clock 命令,弹出 Clock 统计窗口,可以查看时钟统计数,如图 5.35 所示。双击该窗口时,时钟数被清 0。

将光标放在代码行“STL A, @y”,单击 Project 工具栏上的  按钮,在此代码行上设置断点。运行程序时,在 Debug 菜单中执行 Run 命令,程序在断点“STL A, @y”处停止,这时,剖析会话窗口发生了变化。由于剖析会话窗口的指标很多,一个图容纳不下,用如图 5.36(a)和 5.36(b)所示的两个图来表示。

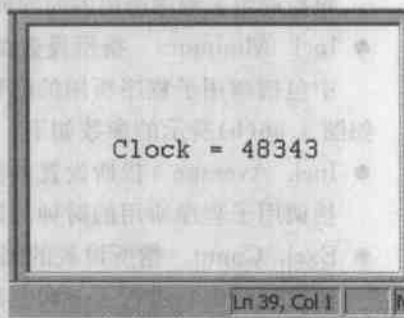


图 5.35 Clock 统计窗口

如图 5.36(a)所示的参数如下:

- Code Size 指最小二进制指令数(与源代码行数不一样)。可以看出,剖析区域有 7 条最小二进制指令数。
- Incl. Count 指所设置的剖析区域在程序运行时进行统计的次数,剖析区域在程序运行过程中只进行了一次统计。

Ranges	Code Size	Incl. Count	Incl. Total	Incl. Maximum	Incl. Minimum
exercise.out					
exercise.asm, line 38-41	7	1	7	7	7

(a)

Incl. Average	Excl. Count	Excl. Total	Excl. Maximum	Excl. Minimum	Excl. Average
7	1	6	6	6	6

(b)

图 5.36 程序数据统计

- Incl. Total 指所设置的剖析区域在程序运行时所花费的总的时钟周期数,其中包括调用子程序所用的时钟周期。剖析区域花费了 7 个时钟周期。
- Incl. Maximum 指所设置的剖析区域在程序运行时所花费的最大的时钟周期数,其中包括调用子程序所用的时钟周期。剖析区域在程序运行时最多要花费 7 个时钟周期。
- Incl. Minimum 指所设置的剖析区域在程序运行时所花费的最小的时钟周期数,其中包括调用子程序所用的时钟周期。剖析区域最少要花费 7 个时钟周期。

如图 5.36(b)所示的参数如下:

- Incl. Average 指所设置的剖析区域在程序运行时所花费的平均时钟周期数,其中包括调用子程序所用的时钟周期。可以看出,剖析区域平均花费了 7 个时钟周期。
- Excl. Count 指所设置的剖析区域在程序运行时进行统计的次数。剖析区域在程序运行过程中只进行了一次统计。
- Excl. Total 指所设置的剖析区域在程序运行时所花费的总的时钟周期数,其中不包括调用子程序所用的时钟周期。剖析区域花费了 6 个时钟周期。
- Excl. Maximum 指所设置的剖析区域在程序运行时所花费的最大的时钟周期数,其中不包括调用子程序所用的时钟周期。剖析区域在程序运行时最多要花费 6 个时钟周期。
- Excl. Minimum 指所设置的剖析区域在程序运行时所花费的最少的时钟周期数,其中不包括调用子程序所用的时钟周期。剖析区域在程序运行时最少要花费 6 个时钟周期。

周期。

- Excl. Average 指所设置的剖析区域在程序运行时所花费的平均时钟周期数,其中不包括调用子程序所用的时钟周期。可以看出,剖析区域平均花费了6个时钟周期。

需要指出的是,Profile Clock 所统计的数据不是很精确的。如果分别在第38行、第39行、第40行和第41行各设置成一个剖析区域,然后把代码行38~41设置成一个剖析区域,并且在代码行“STL A, @y”设置断点。在 Debug 菜单中执行 Run 命令开始运行程序,可以比较两次的统计数据,如图5.37(a)和图5.37(b)所示。

Ranges	Code Size	Incl. Count	Incl. Total	Incl. Maximum	Incl. Minimum
exercise.out					
exercise.asm, line 38	2	2	5	1	1
exercise.asm, line 39	2	2	3	1	1
exercise.asm, line 40	2	2	1	1	1
exercise.asm, line 41	1	1	1	1	1
exercise.asm, line 38-41	7	1	7	7	7

(a)

Incl. Average	Excl. Count	Excl. Total	Excl. Maximum	Excl. Minimum	Excl. Average
2	1	2	0	0	2
1	1	2	0	0	2
0	1	0	0	0	0
1	1	0	0	0	0
7	1	6	6	6	6

(b)

图 5.37 程序数据统计比较

关于探测点(probe points)的使用,将在第6章详细介绍。

第 6 章 CCS2 高级使用——使用文件 I/O

6.1 探测点与文件 I/O

CCS2 可以让用户从 PC 机文件中输入/输出数据到真实的或者仿真的 DSP 目标系统,还可以使用已知的数据去调试源代码。文件 I/O 使用探测点(probe point)能够停止当前程序的执行。当程序停止执行时,程序员可以检查程序运行状态,查看或改变变量的值,检查堆栈和子程序的调用情况,并与文件 I/O、CPU 寄存器或者剖析点连接起来。与设置断点一样,在设置程序的探测点时应注意以下两点:


- (1) 不要将探测点设置在任何延迟分支或调用的指令的地方;
- (2) 不要将探测点设置在重复的块指令倒数第 1、2 行指令的地方。

在没有设置探测点时,程序运行到断点,将停止在断点处,各寄存器更新的值是程序执行到断点时当前寄存器的值,对于其他窗口,比如内存和图形窗口也是一样。当设置了探测点后,程序运行到断点时,各窗口的更新值将不再是程序执行到该断点处的值,而是保持程序执行到探测点的值。由于 CCS2 新增加的文件 I/O 功能,利用探测点调试,就可以查看外部文件数据流与 DSP 算法程序代码交换的情况。

文件 I/O 不支持实时(real-time)数据交换。如果要进行实时数据交换,则应当使用 CCS2 的 RTDX 功能。

6.2 利用探测点观察寄存器的值

6.2.1 探测点的设置与删除

将光标移动到需要设置探测点的代码行,在 Debug 工具栏上单击  图标,则可以在当前行设置探测点。探测点设置成功后,当前位置会出现一个蓝色的菱形标记,如图 6.1 所示。

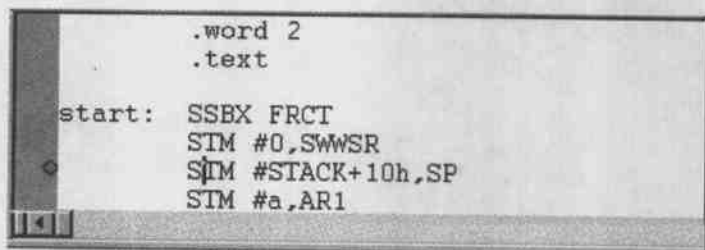



图 6.1 设置探测点

如果要删除已经设置好的探测点,可以把光标移动到需要删除的代码行,再一次在 Debug 工具栏上单击  图标,设置的探测点将被删除。

也可以在 Debug 菜单上单击 Probe Points 命令,会弹出如图 6.2 所示的探测点设置窗口。可以在探测点设置窗口中设置、删除和连接探测点。

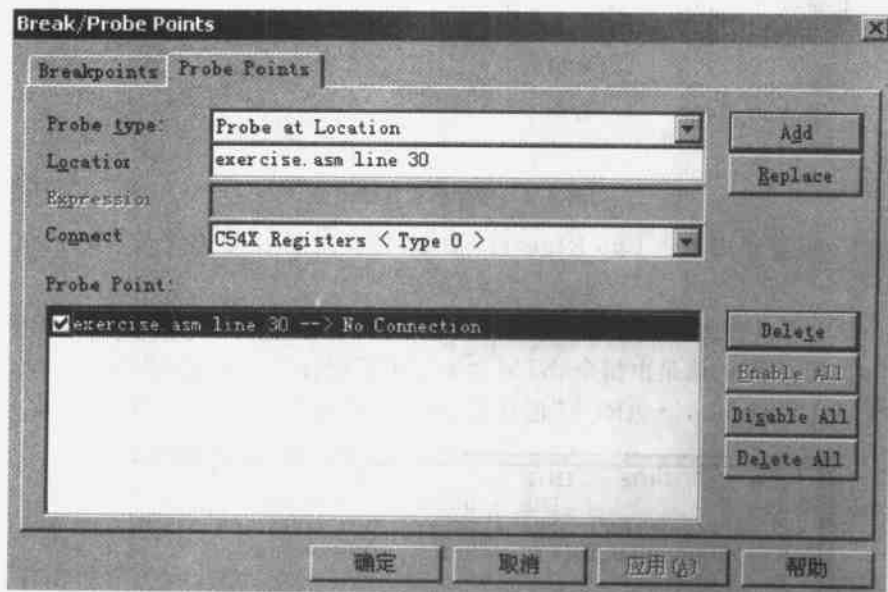


图 6.2 探测点设置窗口

6.2.2 观察寄存器的值

以第 5 章的 exercise.pjt 为例,下面介绍如何用探测点观察 CPU 寄存器的值。

(1) 在代码行“STM #a, AR1”处设置探测点,在代码行“CALL SUM”处设置断点,如图 6.3 所示。

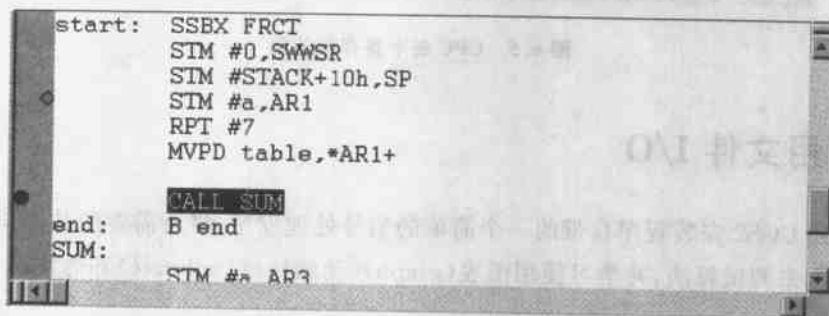


图 6.3 设置探测点和断点

(2) 在 Debug 菜单上单击 Probe Points 命令,在弹出如图 6.2 所示的窗口中建立连接。从 Connect 栏中选择 C54X Registers <Type0> 选项,然后单击“确定”按钮,连接建立完成后,即可在设置好探测点和断点的情况下观察 CPU 寄存器的值。

在 Debug 菜单中执行 Run 命令, 可以看到, 在程序行“MVPD table, *AR1+”前面有个黄色的小箭头, 表示程序停留在该行的断点处, 如图 6.4 所示。

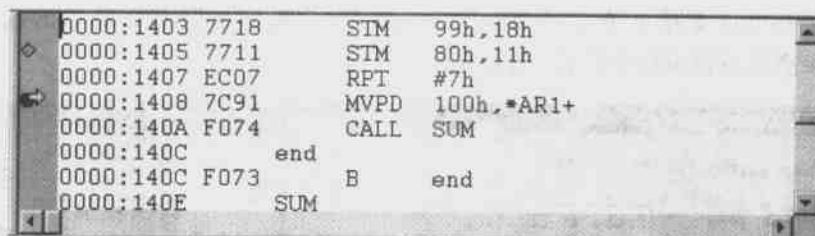


图 6.4 程序停留在断点处

(3) 在 View 菜单中单击 Cpu Registers 命令, 可以观察 CPU 各个寄存器的值, 如图 6.5 所示。

从图 6.5 中可以看到, 虽然程序运行到地址 0000:1408 处, 但 PC 的值却是探测点的地址 0000:1405; 而且 AR1 的值是由指令“STM #a, AR1”把标号 a 的地址赋予的, 即 0x0081, 而不是由指令“MVPD table, *AR1+”把符号 table 的地址 0x0100 赋予的。

PC = 00001408	INTM = 1	SP = 0099	IC = 1
XPC = 0	IMR = 0000	ARO = 0000	C = 0
A = 0000001000	IFR = 0000	OVA = 0	
B = 0000000000	IPTR = FF80	OVB = 0	
T = 0004		OVM = 0	
TRN = 0000		SXM = 1	
ST0 = 1000		C16 = 0	
ST1 = 2940	AR1 = 0081	FRCT = 1	
PMST = FFE0	AR2 = 0000	CMPT = 0	
DP = 0000	AR3 = 0084	CPL = 0	
ASM = 00	AR4 = 0088	XF = 1	
BRAF = 0	AR5 = 0000	HM = 0	
BRC = 0000	AR6 = 0000	MP/MC = 1	
RSA = 0000	AR7 = 0000	OVLY = 1	
REA = 0000	BK = 0000	AVIS = 0	
	ARP = ARO	DROM = 0	

图 6.5 CPU 各个寄存器的值

6.3 利用文件 I/O

本节利用 CCS2 安装程序自带的一个简单的信号处理程序, 将所需要的数据从 PC 机中的一个文件读出来测试算法, 并学习使用图表(graph)、动画执行(animate)和通用扩展语言文件(GEL)。

6.3.1 I/O 文件格式

CCS2 的 DSP 目标系统从 PC 机文件中只能读取和写入下面两种文件格式:

- (1) 公共目标文件格式(COFF), 是二进制文件格式;
- (2) CCS2 数据文件格式, 是一种文本文件。

公共目标文件格式 COFF 有 3 种形式:COFF0、COFF1 和 COFF2。每种形式的 COFF 文件的标题格式不相同,但其数据格式是一样的。C54X 汇编器和 C 编译器建立的是 COFF2 文件格式。当然,C54X 能够读写所有形式的 COFF 文件,在缺省设置下,链接器生成的是 COFF2 文件格式。COFF 文件使模块化编程和管理变得更加方便。

CCS2 数据文件格式包含一行头信息,然后每一行都是一个数据,数据格式可以是十六进制数、十进制整数、十进制长整或者浮点数中的任何一种。

CCS2 数据文件格式的头信息使用下面的语法:

MagicNumber Format StartingAddress PageNum Length

- MagicNumber 固定为 1651。
- Format 是一个 1~4 的数字,用来指示数据文件中的数据存储格式,数据存储格式是十六进制数、十进制整数、十进制长整或者浮点数中的一种。
- StartingAddress 数据存放的起始地址。
- PageNum 页码。
- Length 数据的长度。表示有多少个十六进制数、十进制整数、十进制长整或者浮点数。

CCS2 数据文件格式的头信息一般定义的是缺省的信息,即 1651 1 0 0 0。当从 PC 机文件载入 DSP 目标系统的内存时,可以在弹出的文件 I/O 对话框中输入起始地址和数据长度等,输入的信息将自动地替换 PC 机数据文件的头信息。

下面是将要从 PC 机载入的数据文件 sine.dat 中的数据格式:

```
1651 1 0 0 0
0x00000000
0x0000000f
0x0000001e
0x0000002d
0x0000003a
0x00000046
0x00000050
0x00000059
0x0000005f
0x00000062
0x00000063
0x00000062
0x0000005f
0x00000059
0x00000050
0x00000046
;
```

可以看出,数据文件 sine.dat 中的头信息是缺省的,数据格式是十六进制数。

6.3.2 打开工程文件

假设 CCS2 安装在 D:\application 目录下,本章中所用的 DSP 工程文件 volumel.pjt 是

CCS2 自带的源代码程序,在 CCS2 装好后的 D:\application\ccs\tutorial\sim54xx\下面。打开工程文件的步骤如下:

- (1) 在 D:\application\ccs\myprojects 下建立名为 Volumel 的文件夹。
- (2) 将 D:\application\ccs\tutorial\sim54xx\volumel 下面的所有文件复制到 D:\application\ccs\myprojects\Volumel 中。
- (3) 运行 Code Composer Studio,启动 Simulator。
- (4) 在 Project 菜单中选择 Open 命令,打开工程文件 volumel.pjt。这时将出现如图 6.6 所示的找不到库文件 D:\application\c5400\cgtools\lib\rts.lib 的提示,原因在于工程文件 volumel.pjt 已经被移动到 D:\application\ccs\myprojects\ Volumel 中了。单击 Browse 按钮,在 D:\application\c5400\cgtools\lib\rts.lib 下选取 rts.lib 即可。

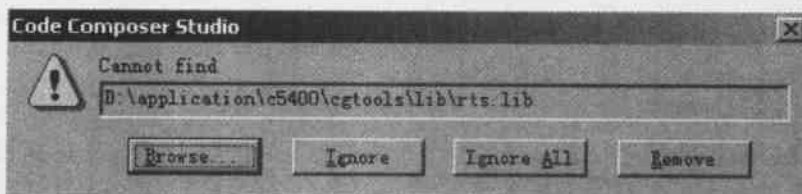


图 6.6 找不到库文件提示框

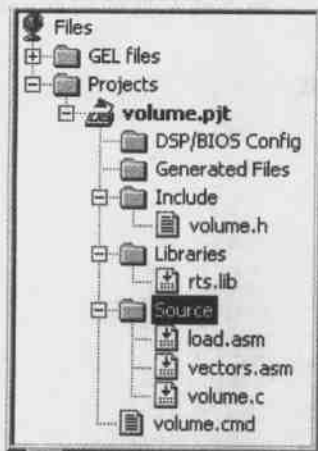


图 6.7 展开后的工程视图

(5) 在工程视图中展开所有文件。工程中的文件包括:

- Volume.c 工程中的 C 语言主程序,包含 main() 函数;
- Volume.h C 语言程序的头文件;
- Load.asm 一个简单的汇编程序,被 Volume.c 调用;
- Vectors.asm DSP 中断向量表复位入口点;
- Volume.cmd 命令链接文件;
- Rts.lib 运行支持库,提供目标板上 DSP 运行支持。

volumel.pjt 工程视图展开后如图 6.7 所示。

6.3.3 阅读源代码

双击 v 如图 6.7 所示的工程视图中的 Volume.c 文件,解析源代码。

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

```
/* Global declarations */
```

```
int inp_buffer[BUFSIZE];
```

```
int out_buffer[BUFSIZE];
```

```
int gain = MINGAIN;
```

```
/* processing data buffers */
```

```
/* volume control variable */
```



```

unsigned int processingLoad    BASELOAD;    * processing routine load value *

struct PARMS str =
{
    2934,
    9432,
    213,
    9432,
    &str
};

/* Functions */
extern void load(unsigned int loadValue);

static int processing(int * input, int * output);
static void dataIO(void);

/*
 * ===== main =====
 */
void main()
{
    int * input = &inp_buffer[0];
    int * output = &out_buffer[0];

    puts("volume example started\n");
    /* loop forever */
    while(TRUE)
    {
        /*
         *   Read input data using a probe - point connected to a host file.
         *   Write output data to a graph connected through a probe - point.
         */
        dataIO();
        #ifdef FILEIO
        puts("begin processing")           /* deliberate syntax error */
        #endif
        /* apply gain */
        processing(input, output);
    }
}

/*
 * ===== processing =====
 */
* FUNCTION: apply signal processing transform to input signal,

```

```

*
* PARAMETERS: address of input and output buffers.
*
* RETURN VALUE: TRUE.
* /
static int processing(int *input, int *output)
{
    int size = BUFSIZE;

    while(size--){
        *output++ += *input++ * gain;
    }
    /* additional processing load */
    load(processingLoad);
    return(TRUE);
}

/* ===== dataIO ===== */
*
* FUNCTION: read input signal and write processed output signal.
*
* PARAMETERS: none.
*
* RETURN VALUE: none.
* /
static void dataIO()
{
    /* do data I/O */
    return;
}

```

从上面的源代码可以看出,主函数 main()显示出一条提示信息后就进入一个无限的循环,不断地调用函数 dataIO()和函数 processing(input, output)。在 processing()函数中将输入 buffer 中的数与增益 gain 相乘,并将结果送给输出 buffer,同时调用汇编 load()函数,根据传给它的参数 processingLoad 的值计算指令周期的时间。函数 dataIO()不做任何工作,它并没有使用 C 代码执行 I/O 操作,而是通过 CCS2 中的探测点,从 PC 机的数据文件 sine.dat 中读取数据到 inp_buffer 中,作为 processing(input, output)函数的参数输入。



6.3.4 设置 PC 数据文件与探测点关联

探测点是开发 DSP 算法的一个非常有用的工具,使用探测点的方法有:

- (1) 将 PC 机文件中的数据传送到 DSP 目标板上的内存中,以供测试算法;
- (2) 将 DSP 目标板上的算法运行后输出的数据传送到 PC 机文件中,以供以后分析;
- (3) 更新窗口内容,实时反应数据的变化。

下面使用探测点将 PC 机上的数据文件 sine.dat 送往 DSP 目标系统作为测试数据,同时

使用断点以便在到达探测点时自动更新所打开的输入/输出数据图表窗口。其步骤如下：

- (1) 单击工程工具栏上的  图标, 或者在 Project 菜单上单击 Rebuild All 命令, 重新生成可执行程序 volume.out。
- (2) 在 File 菜单上执行命令 Load Program, 选择 volume.out 并打开。
- (3) 在工程视图中双击文件 volume.c, 在右边的编辑窗口中显示源代码。
- (4) 将光标移动到 main() 函数中的 dataIO() 代码行, 在工程工具栏上单击  图标, 添加探测点。
- (5) 在 File 菜单上执行命令 File I/O, 将弹出如图 6.8 所示的 File I/O 对话框。

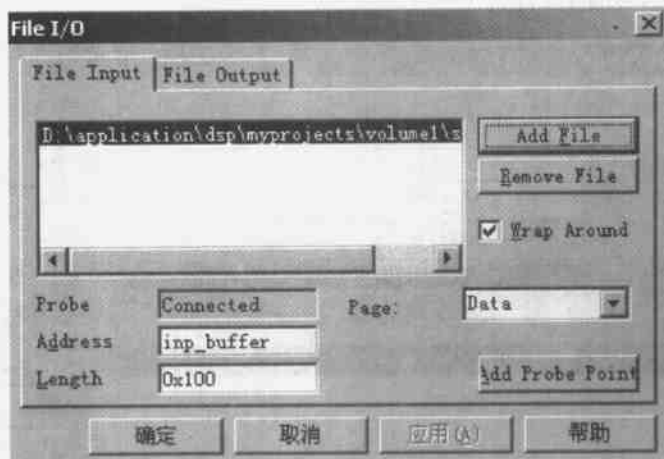


图 6.8 File I/O 对话框

- (6) 单击 File Input 选项, 单击 Add File 按钮。
- (7) 选择 sine.dat 文件, 该文件与 volume.c 等文件在同一目录下。
- (8) 将数据文件 sine.dat 添加到 File I/O 列表中, 将出现一个控制窗口, 如图 6.9 所示。可以在运行程序时使用这个控制窗口来控制数据文件传输的开始、停止、前进或后退等操作。

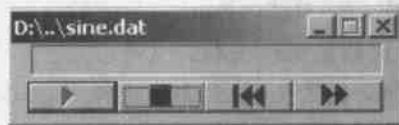






图 6.9 数据文件控制窗口

-  开始按钮: 开始文件传输, 也可以在暂停后继续文件传输。
-  停止按钮: 停止所有的文件 I/O 传输。
-  复位按钮: 重新传输文件, 如果从 PC 机数据文件读数据到 DSP 目标系统中, 那么下一个数据将从数据文件开始读出; 如果将数据从 DSP 目标系统中输出到 PC 机文件中, 那么新的数据将写到 PC 机数据文件的开始处。
-  快进按钮: 当程序运行到探测点时, 这个按钮能再次在 PC 机文件和 DSP 目标系统间传输数据。

- (9) 在 File I/O 对话框中的 Address 栏中输入 inp_buffer, 在 Length 栏中输入 0x100, 选

中 Wrap Around 域。各选项的意义如下：

- Address 从文件中读取的数据将要存放的存储器的地址。
- Inp_buffer 在 volume.c 中定义的整型数组,其长度为 BUFSIZE。
- Length 每次达到探测点时从数据文件读取样点数。由于在 volume.h 中定义了 BUFSIZE=0x64,也就是十进制的 100,表示每次取 100 个样值存放在输入缓冲中。
- Wrap Around 表示读取数据时需要循环,每次读到文件结尾将自动地从文件头开始重新读取数据。这样将从数据文件中读取一个连续的数据流。

(10) 在如图 6.8 所示的 File I/O 对话框中,单击 Add Probe Points 按钮,将弹出如图 6.10 所示的探测点设置对话框。

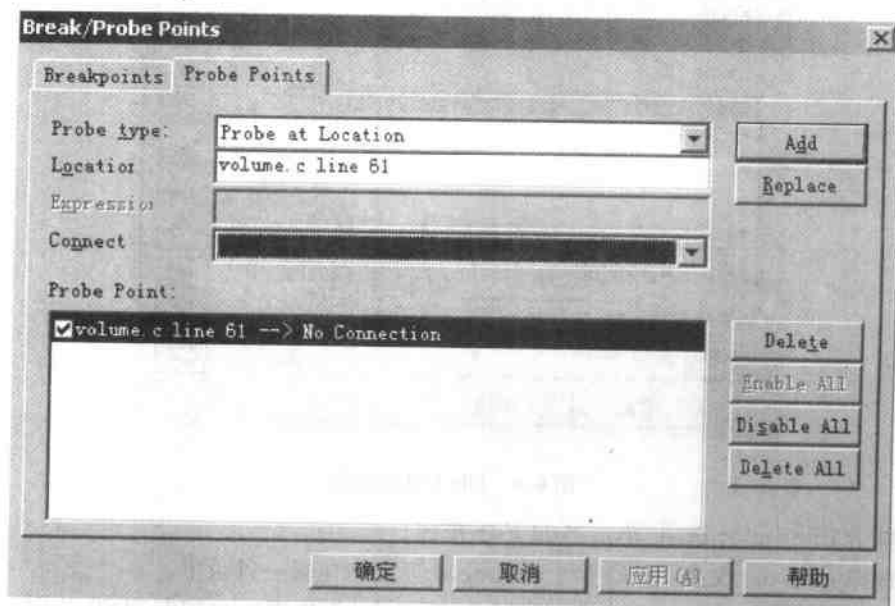


图 6.10 探测点设置对话框

(11) 在 Probe Point 列表中,有一行“volume.c line 61—>No Connection”,表明第 61 行设置了探测点,但还没有与 PC 机数据文件关联。

(12) 在 Connect 栏,单击向下的箭头,从列表中选择 sine.dat 文件。

(13) 单击 Replace 按钮,Probe Point 列表显示探测点已经与 sine.dat 文件关联,如图 6.11 所示。

完成上述步骤后,就准备好了从 PC 机数据文件 sine.dat 读取数据到 volume.c 中的工作。如果此时运行程序,将观察不到任何运行结果。可以设置变量观察窗口来观察变量 inp_buffer 和 out_buffer 的值,但由于程序是循环运行的,输入/输出的数据太多,而且显示的是非常枯燥的数值信息,因此不太直观。

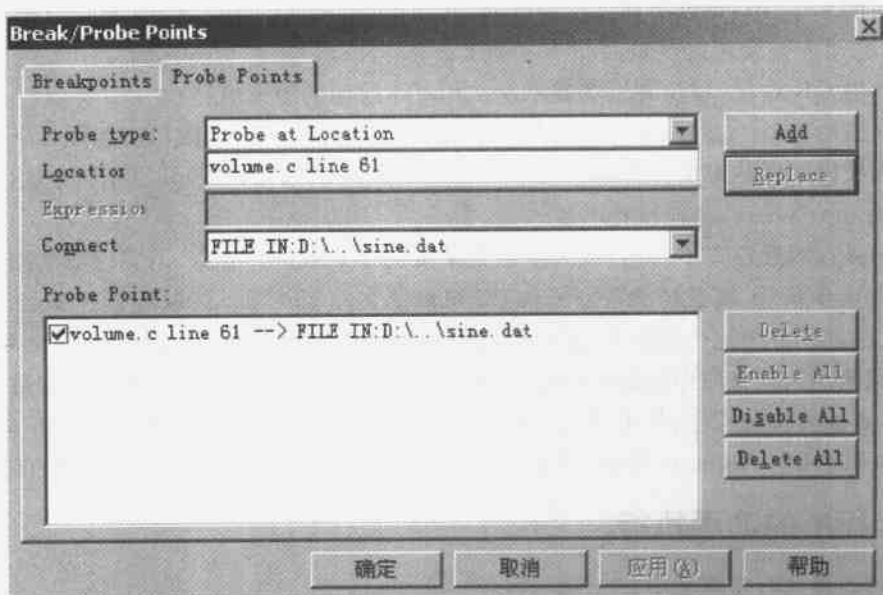


图 6.11 探测点与 sine.dat 文件关联

6.3.5 设置图形显示窗口

CCS2 提供了很多方法将程序运行所产生的数据用图形窗口显示,包括时域/频域波形显示、星座图、眼镜图和图像等。下面设置如何使用时域/频域波形图显示功能来观察一个时域波形。

(1) 在 View 菜单中执行 Graph→Time/Frequency 命令,将弹出 Graph Property 对话框,如图 6.12 所示。

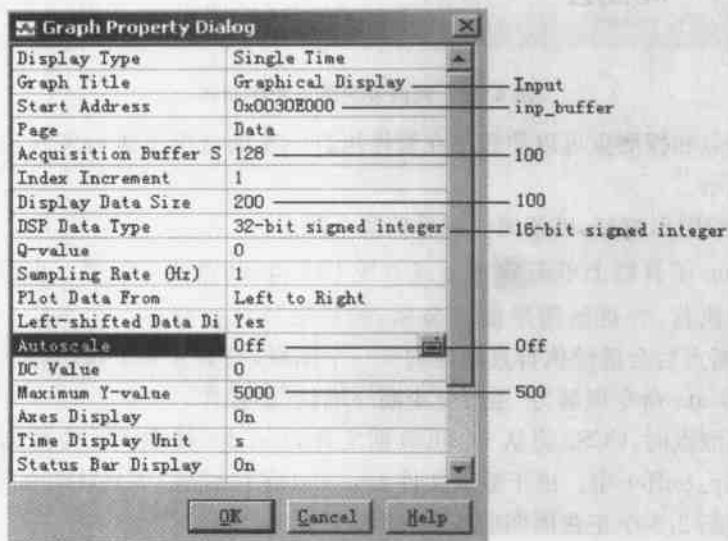


图 6.12 Graph Property 对话框

(2) 在图 6.12 中,更改图形显示的标题、起始地址、采集缓冲区大小、显示数据大小、DSP 数据类型、自动伸缩属性以及最大的 Y 值等。

(3) 单击 OK 按钮,将出现一个显示 `inp_buffer` 波形的图形窗口。

(4) 在图形窗口中右击,从弹出的菜单中选择 `Clear Display`,清除已有的图形。

(5) 再次以同样的方法打开一个 `Graph Property` 对话框,将 `Graph Title` 改为 `Output`,开始地址改为 `out_buffer`,其他设置不变。


(6) 同样在图形窗口中右击,从弹出的菜单中选择 `Clear Display`,清除已有的图形。

完成上述步骤后,就已经准备好了从 PC 机数据文件 `sine.dat` 读取数据到 `volume.c` 中的工作,并设置好了程序运行时输入/输出数据显示的图形窗口。但是,在 `volume.c` 中第 61 行所设置的探测点只能临时中断程序的运行,将 PC 机上的数据文件 `sine.dat` 中的数据传给目标系统,然后继续执行程序,却不能更新图形窗口的显示。在下面要介绍的 6.3.6 节中将设置一个断点,同时使用 `Animate` 命令,使程序运行到断点后能更新图形窗口,并继续自动执行。

6.3.6 程序的动画执行

程序的动画执行步骤如下:

(1) 在 `volume.c` 窗口中,将光标放在代码行“`dataIO()`”上。

(2) 在工程工具栏中单击  图标,在代码行“`dataIO()`”上设置断点,这时可以看到,该行同时有红色小圆点和蓝色小菱形,表示在该行上同时设置了一个断点和一个探测点,如图 6.13 所示。

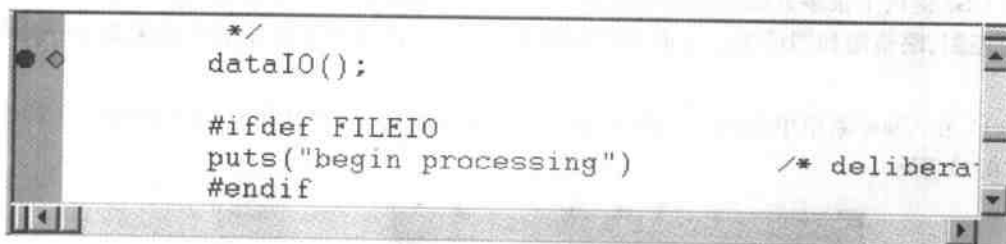



图 6.13 同时设置断点和探测点

同时设置断点和探测点可以使程序在暂停执行一次的情况下进行两种操作:传输数据和更新图形窗口显示。

(3) 重新组织图形窗口,以便能同时看到两个图形。

(4) 在 Debug 工具栏上单击  图标或者按 F12 键,动画执行程序。当程序运行时,碰到断点后临时中断执行,并更新图形窗口显示,然后继续执行程序。与 `RUN` 命令不同的是, `Animate` 在碰到断点后会继续执行直到碰到下一个断点,只有人为干预时,程序才真正停止执行。可以将 `Animate` 命令理解为“运行→中断→继续”的操作。

每次碰到探测点时,CCS2 将从 PC 机数据文件 `sine.dat` 读取 100 个数据样值,并将数据样值写入缓冲 `inp_buffer` 中。由于数据文件 `sine.dat` 保存的是 40 个采样值的正弦波形数据,因此每个图形包括 2.5 个正弦周期的波形,如图 6.14 所示。

从图 6.14 可以看出,输入的波形和输出的波形是反相的,这是因为输入的 `inp_buffer` 中包含的数据是从数据文件 `sine.dat` 读来的,而输出的 `out_buffer` 中包含的数据是处理函数处

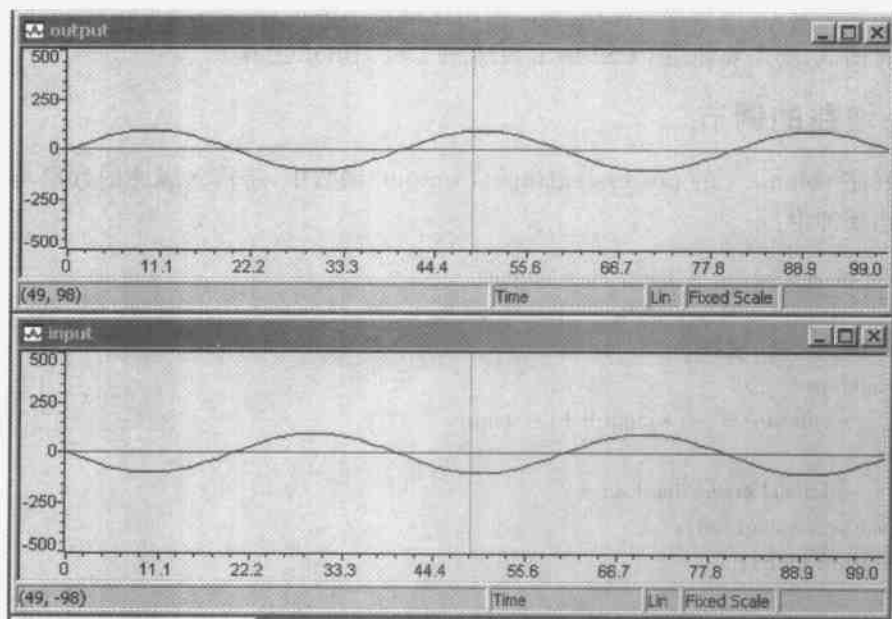


图 6.14 输入/输出波形的图形显示

理的最后一组数据。

使用如图 6.9 所示的文件 I/O 控制窗口,可以控制数据文件输入的开始、停止、前进和后退等操作。在停止状态下,输入 `inp_buffer` 与输出 `out_buffer` 的波形会完全一致,如图 6.15 所示。

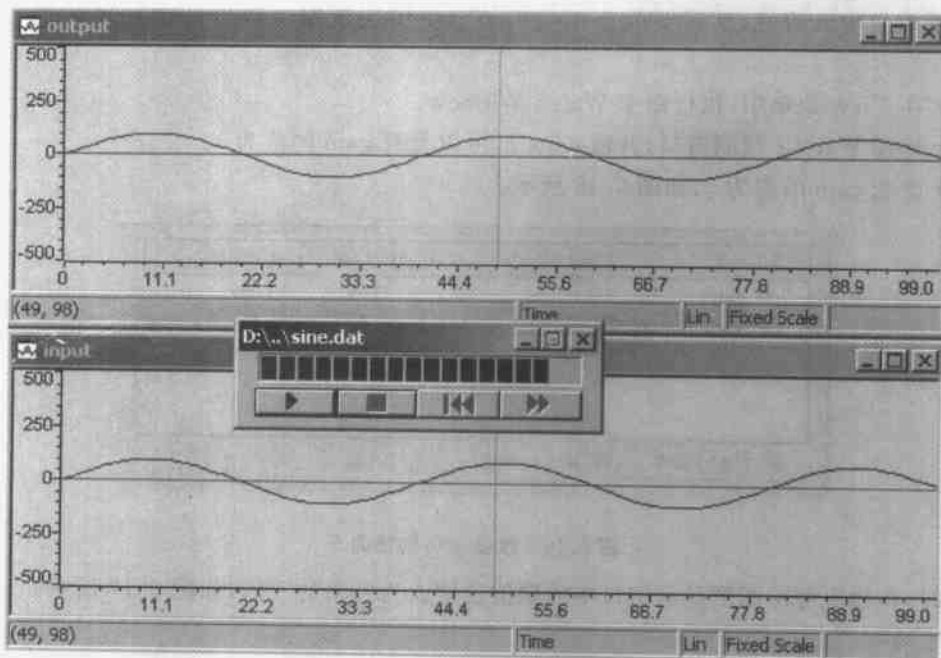


图 6.15 暂停时的输入/输出波形的图形显示

需要注意的是,与剖析点的使用一样,使用探测点不能进行程序的实时调试,如果要对程序进行实时调试,则需要使用 CCS2 的 RTDX 和 DSP/BIOS 工具。

6.3.7 增益的调节

在源程序 volume.c 的 processing(input, output) 函数中,将输入缓冲的数据与增益相乘后送到输出缓冲中:

```
static int processing(int *input, int *output)
{
    int size = BUFSIZE;
    while(size--){
        *output++ = *input++ * gain;
    }
    /* additional processing load */
    load(processingLoad);
    return(TRUE);
}
```

增益又在 volume.c 中被初始化为

```
int gain = MINGAIN
```

而 MINGAIN 在 Volume.h 中被定义为 1

```
#define MINGAIN 1
```

为了改变输出的值,就需要改变增益的值。使用观察窗口可以改变增益的值,其步骤如下:

- (1) 在 View 菜单中,执行命令 Watch Window。
- (2) 选择 Watch1 观测窗口,并输入 gain,可以看到 gain 的值为 1。
- (3) 改变 gain 的值为 5,如图 6.16 所示。

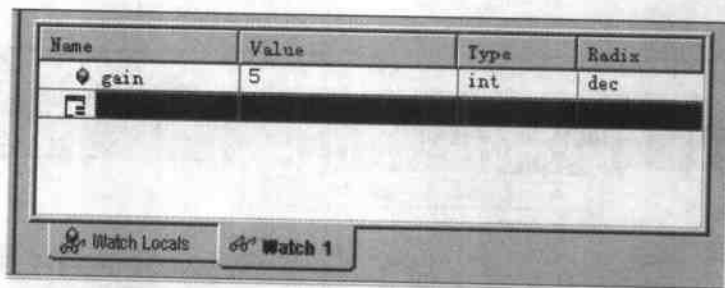


图 6.16 改变 gain 的值为 5

这时,在输出缓冲图形显示窗口中的幅值已增大为原来的 5 倍,如图 6.17 所示。

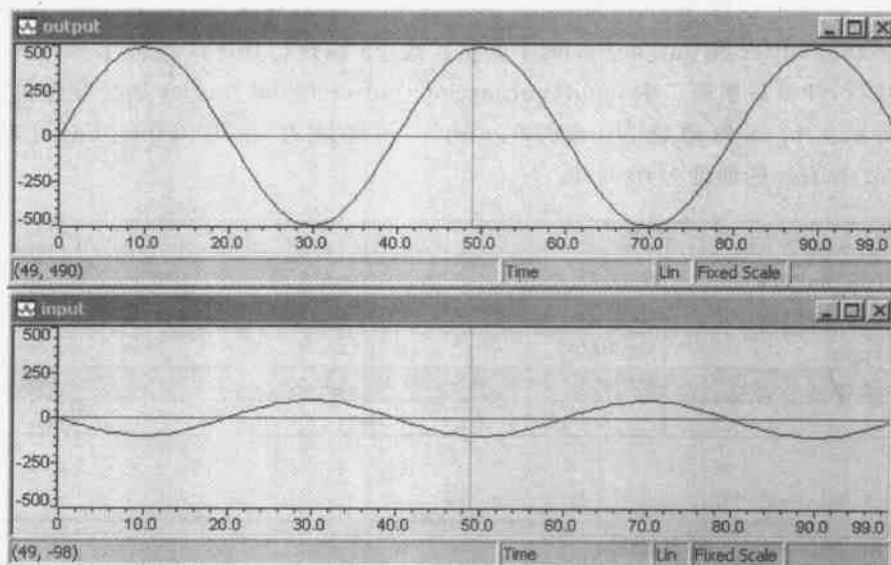


图 6.17 gain=5 时输入/输出图形显示

6.3.8 查看变量属性和值

使用观察窗口(watch window)可以观察变量的属性和变量的值。在 GEL 菜单中执行 CPU_RESET 命令,复位 CPU 目标系统,重新装入 volume.out 可执行程序。在源代码程序 volume.c 中设置 3 个断点,设置好断点的源代码如图 6.18 所示。

```

void main()
{
    int *input = &inp_buffer[0];
    int *output = &out_buffer[0];
    puts("volume example started\n");
    while(TRUE)
    {
        dataIO();
        #ifdef FILEIO
        puts("begin processing")
        #endif
        processing(input, output);
    }
}

static int processing(int *input, int *output)
{
    int size = BUFSIZE;
    while(size--){
        *output++ = *input++ * gain;
    }
    load(processingLoad);
    return(TRUE);
}






static void dataIO()
{
    /* do data I/O */

    return;
}

```

图 6.18 设置好断点的源代码

在 View 菜单上执行 Watch Window 命令, 打开变量观察窗口, 在 Watch1 里面输入 input、output、inp_buffer 和 out_buffer 四个变量。按 F5 键执行程序。当程序执行到第一个断点时暂停执行, 变量观察窗口中 input、output、inp_buffer 和 out_buffer 四个变量的值显示如图 6.19 所示。input 的地址为 0x00A5, output 的地址为 0x0109, inp_buffer 的地址为 0x00A5, out_buffer 的地址为 0x0109。

Name	Value	Type	Radix
 inp_buffer	0x00A5	int[100]	hex
 out_buffer	0x0109	int[100]	hex
 input	0x00A5	int *	hex
 output	0x0109	int *	hex
			




 Watch Locals
 Watch 1

图 6.19 变量观察窗口四个变量的值

继续按 F5 键执行程序, 程序碰到第二个断点暂停执行, 这时变量观察窗口中 input 的值发生了变化, 而变量 output、inp_buffer 和 out_buffer 三个变量的值不变。继续按 F5 键执行程序, 可以看到这时变量观察窗口中变量 input 和 output 变成了 identifier not found。

由于第一个断点设置在 main() 函数的结尾处, 第二个断点设置在 process() 函数结尾处, 第三个断点设置在 dataIO() 函数处, 可以看出 input、output、inp_buffer 和 out_buffer 四个变量在 main() 函数和 process() 函数中有效, 而变量 input 和 output 在 dataIO() 函数中无效, 变量 inp_buffer 和 out_buffer 在 dataIO() 函数中有效。这是因为变量 inp_buffer 和 out_buffer 是全局变量, 而变量 input 和 output 在 main() 函数和 process() 函数中定义成局部变量, 而在 dataIO() 函数中没有定义。

由于变量 inp_buffer 和 out_buffer 被定义成数组, 单击变量 inp_buffer 和 out_buffer 前面的“+”号展开, 可显示其 100 个数组元素。由于没有和数据文件 sine.dat 关联, 也没有将文件 sine.dat 中的数据传入缓存中, 所以变量 inp_buffer 和 out_buffer 的数组元素中的值全部为 0, 如图 6.20 所示。

按照前面介绍的方法, 在代码行“dataIO()”上设置一个断点和探测点, 把程序 volume.out 与数据文件 sine.dat 关联。在 Debug 工具栏上单击  图标或者按 F12 键, 动画执行程序。可以看出变量 inp_buffer 和 out_buffer 的数组元素中的值已经是数据文件 sine.dat 传来的数据, 如图 6.21 所示。输入的数据和输出的数据是反相的, 与如图 6.14 所示的波形显示完全相符。

Name	Value	Type	Radix
inp_buffer	0x00A5	int[100]	hex
[0]	0	int	dec
[1]	0	int	dec
[2]	0	int	dec
[3]	0	int	dec
[4]	0	int	dec
.....	-		
out_buffer	0x0109	int[100]	hex
[0]	0	int	dec
[1]	0	int	dec
[2]	0	int	dec
[3]	0	int	dec
[4]	0	int	dec
.....	-		

Watch Locals Watch 1

图 6.20 数组元素中的值为 0

Name	Value	Type	Radix
inp_buffer	0x00A5	int[100]	hex
[0]	0	int	dec
[1]	-15	int	dec
[2]	-30	int	dec
[3]	-45	int	dec
[4]	-58	int	dec
.....	-		
out_buffer	0x0109	int[100]	hex
[0]	0	int	dec
[1]	15	int	dec
[2]	30	int	dec
[3]	45	int	dec
[4]	58	int	dec
.....	-		

Watch Locals Watch 1

图 6.21 与文件 sine.dat 关联后数组元素中的值

6.3.9 从文件读入数据到内存

可以直接将 PC 机数据文件载入 DSP 目标系统内存中,其步骤如下:

(1) 在 File 菜单中执行 Data→Load 命令,将打开一个如图 6.22 所示的 Load Data 对话框,在其中选择文件 sine.dat。单击“打开”按钮。

(2) 这时将弹出一个如图 6.23 所示的 Load File Into Memory 对话框。由于 inp_buffer 的地址为 0x00A5,输入数据的长度为 100,所以在图 6.23 中的 Address 栏中输入 0x00A5,在 Length 栏中输入 0x64,而 Page 栏中保持 Data 不变。



图 6.22 Load Data 对话框



图 6.23 Load File Into Memory 对话框

数据文件 sine.dat 中的数据。


(3) 单击 OK 按钮, 这时 PC 机数据文件 sine.dat 中的数据已经载入 DSP 目标系统内存地址 0x00A5 开始的 100 个字节。这时, 可以观察内存中 inp_buffer 的数据。

(4) 在 View 菜单中执行 Memory 命令, 在弹出的如图 6.23 所示的内存观察设置窗口中的 Address 栏中输入 0x00A5, 单击 OK 按钮, 可以看到如图 6.24 所示的内存中 inp_buffer 的数据, 也就是数



图 6.24 内存中 inp_buffer 的数据

按照前面介绍的方法设置图形窗口显示的起始地址、采集缓冲区大小、显示数据大小、DSP 数据类型、自动伸缩属性以及最大的 Y 值等。因为不需要将 volume.out 与 PC 机数据文件 sine.dat 通过探测点关联, 所以不用设置探测点。

在 Debug 工具栏上单击  图标或者按 F12 键, 动画执行程序。重新组织图形窗口, 以便能同时看到输入/输出两个图形, 如图 6.25 所示。

按同样的方法在 File 菜单中执行 Data→Save 命令, 可以将程序运行的结果保存到文件中供分析使用, 具体步骤不再介绍。

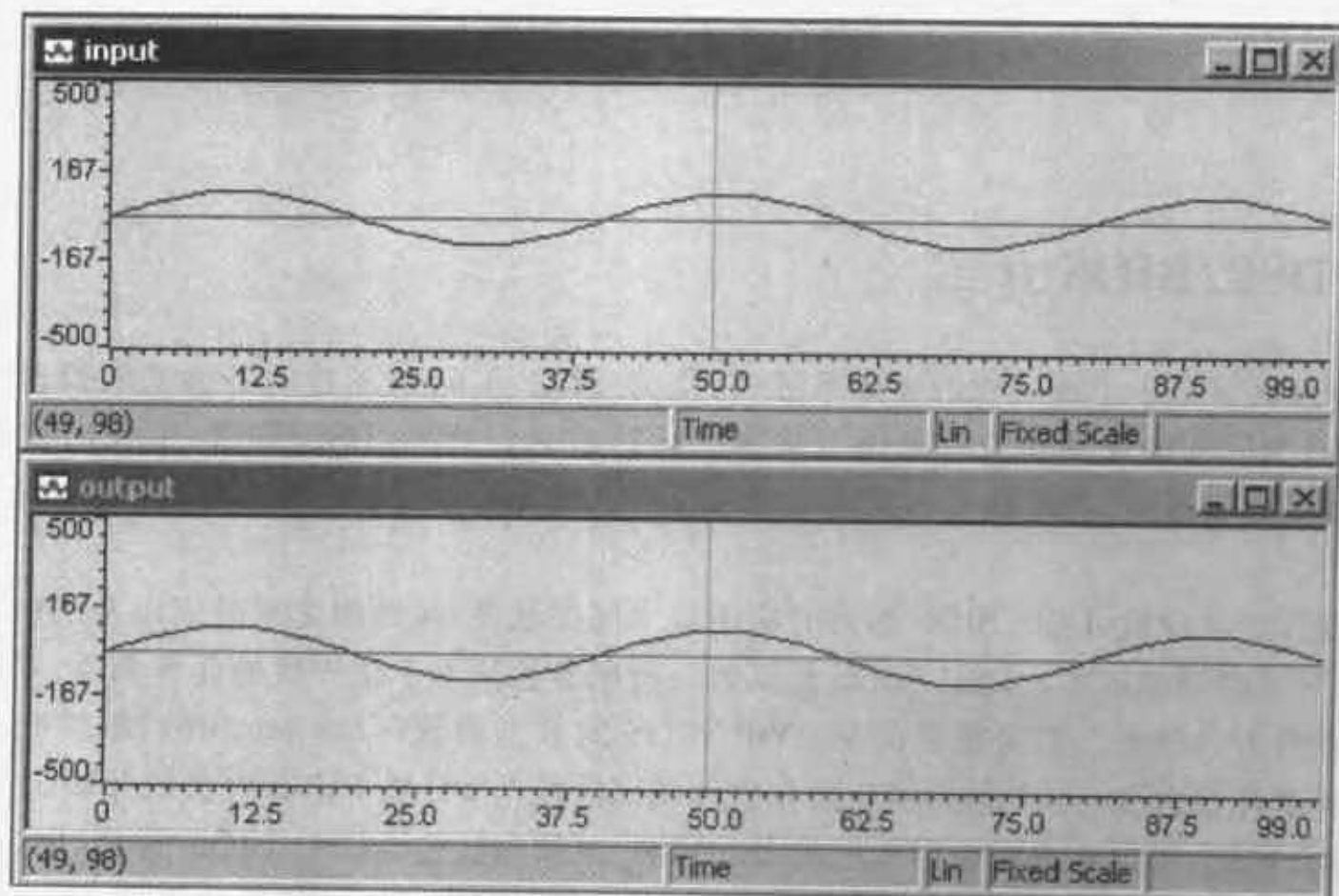


图 6.25 输入/输出波形的图形显示

第 7 章 CCS2 高级使用——DSP/BIOS

原理与应用

7.1 DSP/BIOS 介绍

DSP/BIOS 是一个准实时的操作系统,它作为 TI 公司 DSP 芯片的各种实时操作系统的底层软件,为嵌入式开发和应用提供了基本的运行服务。此外,DSP/BIOS 能实时捕获 DSP 目标系统的各种信息,并将这些信息传送给 PC 机上的 BIOScope 分析工具,对应用程序进行实时分析。

在软件开发阶段,DSP/BIOS 为实时应用提供底层软件,从而简化实时应用系统的软件设计,节约开发时间和成本。DSP/BIOS 提供的运行服务包括基于优先级的任务调度、中断处理和准实时的 I/O 服务。更为重要的是,DSP/BIOS 的数据捕获(data capture)、事件统计(statistics)和事件日志(event logging)功能在软件调试阶段与 PC 机 CCS2 内部的分析工具 BIOScope 配合使用,可以完成对应用程序的实时探测、跟踪和监控。DSP/BIOS 同 RTDX 技术和 CCS2 的可视化工具相结合,不但可以直接实时显示原始数据(二维信号波形或者三维图像)外,还能对原始数据进行处理,对数据进行实时的 FFT 频谱分析,对星座图和眼睛图进行处理等。

DSP/BIOS 具有以下特性:

- 提供系统级的内核服务,如内存管理、通信机制和中断处理等,可以让 DSP 应用程序开发者不必把精力放在硬件方面,而能集中精力开发 DSP 软件。
- 应用 DSP/BIOS 能动态地创建对象和删除对象,也可以在配置工具中静态地创建和删除对象。
- DSP/BIOS 的线程模型支持各种类型的线程,包括硬件中断、软件中断、多任务、周期函数以及 IDLE 函数等。通过选择不同类型的线程可以控制线程的优化等级和 blocking 特性。
- 能使用信号量(semaphores)、信箱(mailboxes)以及资源锁等手段,支持线程间的通信和同步。
- 能支持管道和流两种 I/O 模型。管道用于 DSP 目标系统与 PC 机之间的通信,它支持用一个线程往管道写数据,同时用另外一个线程从管道读数据这种简单的方式。流能在更复杂的 I/O 中使用,而且支持设备驱动程序。
- 能提供低级的系统原语(primitives)。在处理错误、创建通用数据结构和管理存储器时非常有用。
- 支持实时分析功能。能实时地捕获和显示数据,用以在开发软件阶段诊断和查找系统级的缺陷。DSP/BIOS 能提供多种机制实时地获取、传输和显示数据。

- DSP/BIOS 占用极少的系统资源。在 TMS320C54X DSP 平台上, DSP/BIOS 最小只需要 24 字的程序存储空间和 475 字的数据存储空间。此外, DSP/BIOS 内核的大小仅依赖于 DSP 应用程序的需要, 只有直接或间接被应用程序所使用的模块才能被链接到可执行程序。DSP/BIOS 配置工具根据应用程序需要提供了大量可替代的执行模块, 进一步缩小了 DSP/BIOS 的代码大小, 并缩短了执行时间。
- DSP/BIOS 支持 TI 公司的 TMS320C5000 和 TMS320C6000 DSP 平台。运行 DSP/BIOS 的应用程序能在评价模块板 EVM (Evaluation Module)、初学者工具 DSK (DSP Starter Kit)、第三方目标板、定制目标板、软件模拟器(simulators)、软件开发系统 SWDS(Soft Developing System)和仿真器 XDS(Extended Developing System)中使用。
- 通过校验属性, DSP/BIOS 配置工具能在应用程序编译链接前探测和校正错误。同时 DSP/BIOS 对象的运行日志和统计数据能在运行时记录发生的事件。
- DSP/BIOS 提供了通用的标准 API, 允许 DSP 软件开发者所开发的算法方便地与其他应用程序集成。

DSP/BIOS 的设计方法如下:

(1) 所有在配置工具中创建的 DSP/BIOS 对象被链接到可执行程序代码中, 因而可以减少代码并优化程序的内部数据结构。

(2) PC 主机上可以使用“虚拟仪表”(比如 logs 和 traces)来显示 DSP 目标系统板上的数据。

(3) DSP/BIOS 的 API 函数被模块化, 所以 API 函数中只有程序真正用到的模块才能被链接到可执行程序代码中。

(4) DSP/BIOS 的函数库基本上都是用汇编语言实现的, 因此执行效率非常高。

(5) DSP 目标系统板与 DSP/BIOS 插件之间的通信, 都是在后台 idle loop 时期完成的, 因此 DSP/BIOS 插件并不影响 DSP 应用程序的执行。如果目标系统板 DSP 应用程序太忙, 以至于无法正常执行后台任务, DSP/BIOS 插件能暂停与目标板的通信。

利用 DSP/BIOS 插件, 开发人员能在 CCS2 中测试 DSP 目标系统板上的应用程序, 监视 CPU 的负荷(load)、时序(timing)、日志(logs)以及线程执行(thread execution)等。应用 DSP/BIOS 插件的应用程序的开发思路是: 在 PC 主机上, 开发人员采用 C 语言或者 DSP 汇编语言编写使用 DSP/BIOS API 的 DSP 应用程序, 然后采用 DSP/BIOS 配置工具定义程序中使用的对象。DSP/BIOS 所起的作用就像一种硬件的逻辑分析仪器, 只是它使用软件而不是用硬件来监视 DSP 应用程序的运行。

7.2 DSP/BIOS 组件

DSP 生成与调试环境下的 DSP/BIOS 组件框图如图 7.1 所示。

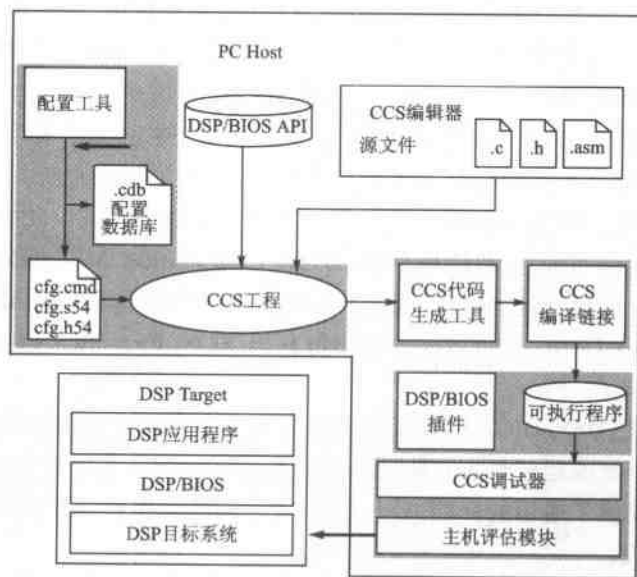


图 7.1 DSP/BIOS 组件框图

7.2.1 实时库与 API 函数

DSP/BIOS 短小的固件实时库为 DSP 目标系统板上的嵌入式程序提供了基本的运行服务,其包括多线程调度、I/O 管理以及数据的实时捕获等。

DSP/BIOS API 被分成很多不同的模块,2.4.2 节已经介绍了 DSP/BIOS 的各种模块。应用程序通过调用 DSP/BIOS API 函数来完成各种诊断调试功能。对于 C 程序,API 函数需要在头文件中声明。对于 DSP 汇编程序,API 函数需要在宏中声明。为了减少开发周期和代码量,DSP/BIOS 实时库采用汇编语言编写,因此,使用汇编语言编写 DSP/BIOS 应用程序(不是 DSP 应用程序)比使用 C 语言编写更为方便。

7.2.2 DSP/BIOS 配置工具

DSP/BIOS 配置工具为 DSP 软件开发人员创建和配置了在 DSP 应用程序中所使用的 DSP/BIOS 对象,包括软件中断、I/O 流和事件日志等,也可以利用 DSP/BIOS 配置工具配置内存、线程优先权和中断句柄等。DSP/BIOS API 在运行时使用的是利用 DSP/BIOS 配置工具所创建的对象和所设置的属性。

DSP/BIOS 配置工具在 .cdb 文件中保存所创建的对象和各种设置,同时还创建一些其他的文件,所有这些文件会自动添加在 CCS2 的工程视图中。

打开 DSP/BIOS 配置窗口有如下两种方法:

(1) 如果要在 CCS2 中使用 DSP/BIOS 配置工具,则在 File 菜单中执行 New→DSP/BIOS Configuration 命令。

(2) 如果要作为一个单独的程序运行 DSP/BIOS 配置工具,则直接在“开始”菜单中,选择 Programs→Texas Instruments→Code Composer Studio Configuration Tool 项,便可以打开配置窗口。配置窗口与 Windows 资源管理器具有很相似的界面,如图 7.2 所示。

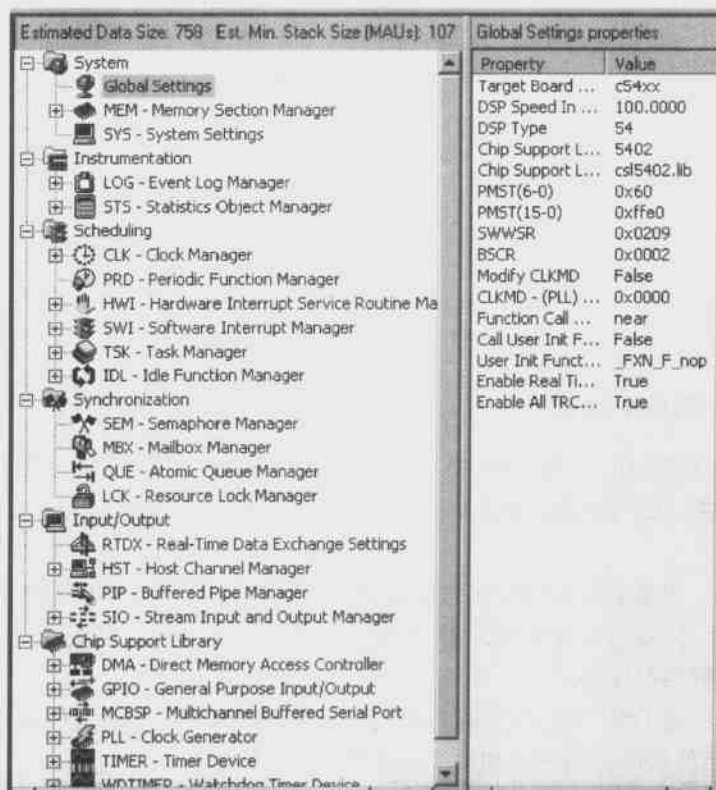


图 7.2 DSP/BIOS 配置窗口

DSP/BIOS 配置文件有如下两个作用：

- (1) 设置全局运行参数；
- (2) 作为一个可视化编辑器，创建目标程序 DSP/BIOS API 所调用的运行对象和设置其属性。

下面按字母顺序，列举 DSP/BIOS 配置窗口中的各种 DSP/BIOS 模块在配置窗口树型目录中的位置：

DSP/BIOS 模块	树型目录中的位置
CLK	Scheduling
DGN	Input/Output, SIO Drivers
DHL	Input/Output, SIO Drivers
DPI	Input/Output, SIO Drivers
GBL	System
HST	Input/Output
HWI	Scheduling
IDL	Scheduling
LCK	Synchronization
LOG	Instrumentation
MBX	Synchronization
MEM	System
PIP	Input/Output

PRD	Scheduling
QUE	Synchronization
RTDX	Input/Output
SEM	Synchronization
SIO	Input/Output
STS	Instrumentation
SWI	Scheduling
SYS	System
TSK	Scheduling
UDEV	Input/Output, SIO Drivers

7.2.3 DSP/BIOS 插件

CCS2 能利用 DSP/BIOS 插件对 DSP 应用程序进行实时分析,并可以实时监视 DSP 应用程序的运行,同时对 DSP 应用程序的实时性的性能影响很小。DSP/BIOS 插件提供了以下实时性分析特性:

- 程序跟踪 能显示写入目标日志的事件并在程序的执行过程中反映动态控制流程。
- 性能监控 能动态跟踪和统计 DSP 目标系统板上的资源使用情况,如 DSP 处理器的负载和线程的时序等。
- 文件流 能将 DSP 目标系统板上的 I/O 对象与 PC 主机上的文件关联起来。

DSP/BIOS 插件实时分析窗口如图 7.3 所示。

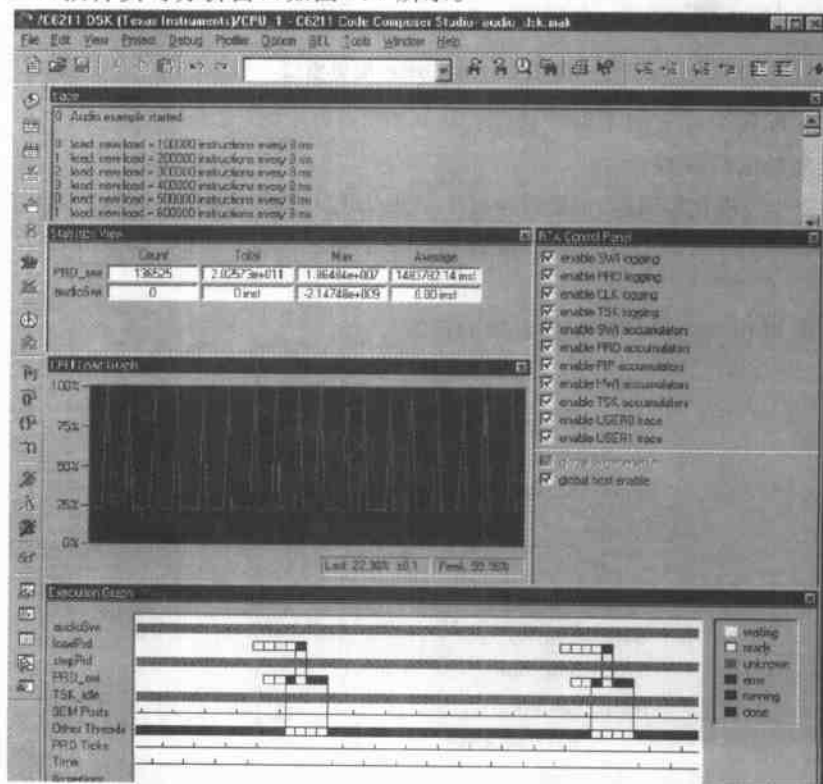


图 7.3 DSP/BIOS 插件实时分析窗口

7.3 DSP/BIOS 命名规则

每个 DSP/BIOS 模块都有惟一的一个由 3~4 个字母组成的名字作为前缀,以表明该模块的操作功能、头文件和对象等。所有以大写字母开头的、用下划线连接起来的形如 XXX_ 的标识符,都是 DSP/BIOS 的保留字。

7.3.1 头文件命名

每个 DSP/BIOS 模块都有两个头文件,其中声明该模块接口的所有常量、结构类型和函数等。

- XXX.h C 语言程序的 DSP/BIOS API 函数的头文件。任何 C 语言函数使用的模块,都必须在 C 源代码中包含 std.h 和这个头文件。
- XXX.h54 汇编语言程序的 DSP/BIOS API 函数头文件。任何 DSP 汇编语言编写的源代码使用的模块都必须包含这个头文件。这个文件中包含了与芯片相关的宏定义,还包含了一个名为 module.hri 的头文件,并在其中定义了所有支持芯片共享的数据结构。

在 DSP 应用程序中还应当包含与该模块相关的头文件。在 C 程序中,头文件 std.h 必须放在其他所有头文件的前面,因为在头文件 std.h 中包含了标准类型和常量的定义。除了头文件 std.h 外,其余的头文件包含的位置没有先后之分,比如:

```
#include <std.h>
#include <tsk.h>
#include <sem.h>
#include <prd.h>
#include <swi.h>
```

7.3.2 对象命名

在 DSP/BIOS 配置文件中,包含的系统对象通常使用由 3~4 个字母组成的前缀,比如包含 LOG 系统对象在配置中使用的默认名为 LOG_system。

在配置工具中所创建的对象也可以按照设计者的命名习惯来命名,可以使用模块名作为对象的前缀。比如,一个对数据进行编码的 TSK 对象,可以被命名为 encoderTSK。

7.3.3 函数命名

DSP/BIOS API 操作名的命名格式为 MOD_action,其中 MOD 为包含此操作的模块名,action 为操作所完成的功能名。比如 SWI_post 函数是在 SWI 模块中定义的函数,该函数的功能是发送一个软件中断。

DSP/BIOS API 中还包括一些嵌入的函数,由内置对象运行,例如:

- CLK_F_isr 由 HWI 对象 HWI_TINT 运行,提供低分辨率的时钟。
- PRD_F_TICK 由 CLK 对象 PRD_clock 运行,提供系统时钟。
- PRD_F_swi 由具有最高优先级的 SWI 对象 PRD_swi 运行,运行 PRD 函数。

- `_KNL_run` 由具有最低优先级的 SWI 对象 `KNL_swi` 运行,运行任务调度器(如果任务调度器被使能)。这是一个名为 `KNL_run` 的 C 函数,前缀的下划线表明该函数被汇编代码调用。
- `IDL_loop` 由具有最低优先级的 TSK 对象 `TSK_idle` 运行,运行 IDL 函数。
- `IDL_F_busy` 由 IDL 对象 `IDL_cpuload` 运行,计算当前 CPU 负荷。
- `RTA_F_dispatch` 由 IDL 对象 `RTA_dispatcher` 运行,获取实时分析数据。
- `LNK_F_datapump` 由 IDL 对象 `LNK_dataPump` 运行,进行实时分析并将 HST 通道数据传送给主机。
- `HWI_unused` 它不是一个实际的函数名,在配置工具中用来标明一个未使用的 HWI 对象。

在用户程序代码中不能调用任何以 `MOD_F_` 开头的嵌入函数,这些函数只能作为函数参数在配置工具中调用。

7.3.4 数据类型名

DSP/BIOS API 函数并不明确地使用基本的 C 数据类型(如 `int`、`char` 等)。为了将程序移植到其他支持 DSP/BIOS API 函数的处理器中使用,DSP/BIOS 定义了自己的标准数据类型。下面是在 `std.h` 中定义的数据类型:

- `Arg` 能存放指针和整型参数的数据类型;
- `Bool` 布尔类型;
- `Char` 字符类型;
- `Int` 有符号整数类型;
- `LgInt` 有符号长整数类型;
- `LgUns` 无符号大整数类型;
- `Ptr` 指针类型;
- `String` 字符串类型;
- `Uns` 无符号整数类型;
- `Void` 空类型。

其余在 `std.h` 中定义的数据类型,不在 DSP/BIOS API 中使用。此外,常量 `NULL(0)` 用于表示一个空的指针值,常量 `TRUE(1)` 和 `FALSE(0)` 用做布尔值。

DSP/BIOS API 使用的对象通常命名为 `MOD_Obj`,其中 `MOD` 是模块名代号。当在应用程序中使用这些对象时,应当使用 `extern` 语句声明,例如:

```
extern LOG_Obj trace;
```

7.3.5 存储器段命名

DSP/BIOS 使用如下的存储器段名:

- `IDATA` 内部数据存储器段;
- `EDATA` 外部数据存储器段的主块(primary block);
- `EDATA1` 外部数据存储器段的从块(secondary block),与 `EDATA` 不相连;

- IPROG 内部程序存储器段；
- EPROG 外部程序存储器段的主块；
- EPROG1 外部程序存储器段的从块,不与 EPROG 相连；
- USERREGS 页面0用户存储器段(28个字)；
- BIOSREGS 页面0保留寄存器段(4个字)；
- VECT 中断向量段。

使用配置工具,可改变除 IPRAM 段和 IDRAM 段外的存储器段的起始地址、大小及名字。配置工具定义标准存储器段及其地址分配如下:

- IDATA 应用程序堆栈段和 BIOS 堆栈段；
- EDATA 应用程序参数存储器段、常量存储器段和 BIOS 数据存储器段；
- IPROG BIOS 程序存储器段；
- EPROG BIOS 启动代码存储器段。

可以在配置工具中用 MEM 管理器改变默认的地址分析。

7.4 DSP/BIOS 程序生成过程

DSP/BIOS 支持可重复的程序开发周期,开发者可以先创建应用程序的基本框架,在 DSP 算法载入目标系统前进行测试。开发者也可以很方便地改变程序组件的类型和优先级,以实现各种功能。

一个 DSP/BIOS 程序的开发步骤如下:

- (1) 使用 C 语言或汇编语言编写程序框架。
- (2) 使用配置工具创建程序使用的对象。
- (3) 保存配置文件,保存时产生的文件将在编译和链接程序时使用。
- (4) 使用工程文件编译和链接程序。
- (5) 使用 simulator 或初始硬件(initial hardware)和 DSP/BIOS 插件测试程序。可以对日志(log)、跟踪(trace)、统计对象(statistics object)、时序(timing)及软件中断等进行监视。
- (6) 重复(2)~(5)步直到程序运行正确为止。此时可在基本程序结构的基础上添加功能,并对框架作修改。
- (7) 当硬件产品设计好后,修改配置文件以支持硬件,并在硬件上测试程序。

7.4.1 使用配置工具

配置工具是一个与 Windows 资源管理器具有相似界面的可视化编辑器,可在 DSP/BIOS 运行时初始化其数据结构和设置其参数。当保存配置文件时,配置工具产生与设置相匹配的汇编文件、头文件以及链接命令文件。当生成 DSP 应用程序时,这些文件被接入程序中。

1. 新建配置

在 CCS2 中,在菜单 File 中执行命令 New→DSP/BIOS Config,或者直接在 Windows 的“开始”菜单下运行配置工具,选择合适的模板(template)并单击 OK 按钮。

2. 定制一个模板

通过创建一个配置文件并将其存储在 include 目录下,可以添加一个定制的模板,这样就

可以根据硬件定义配置设置,以便将其作为模板使用,而不必每次都重新配置。比如,要为 C5000 定点 DSP 程序创建一个 DSP/BIOS 程序,可以使用 C5000 环境提供的设置,也可以添加一个定制的模板,加入定点的运行库支持,以便下次可以直接使用该配置。

定制模板的步骤如下:

(1) 在 Windows 环境下执行 Start→Programs→Code Composer Studio' C5000→Configuration Tool。

(2) 在 File 菜单下选择 New,弹出 New 窗口,选择 sd54.cdb。这时将弹出如图 7.4 所示的树状形窗口。

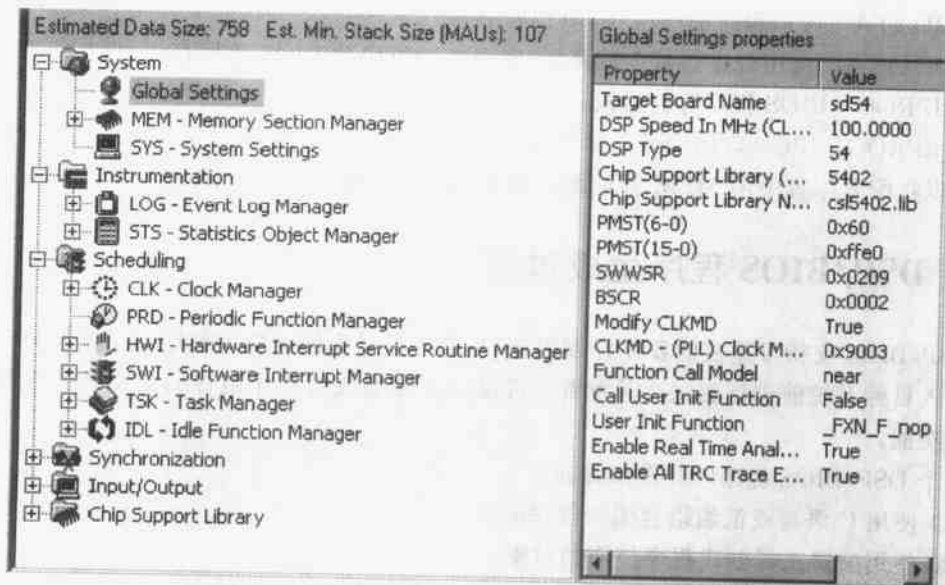


图 7.4 sd54.cdb 配置文件


(3) 在 Global Settings 上右击并选择 Property,将弹出如图 7.5 所示的属性设置窗口。

(4) 在 Target Board Name 栏改名为 evm54。

(5) 在 File 菜单下选择 Save As,将文件存在 D:/application/ccs/c5400/bios/include 目录下,取名为 evm54.cdb,在保存类型栏选择 Seed files(*.cdb),保存文件。

(6) 在 File 菜单下选择 Exit,退出配置工具。

3. 设置模块的全局属性

在如图 7.4 所示的窗口中,左边窗口显示的是模块,右边窗口显示的是该模块的属性。当双击一个模块时,右边的窗口将显示此模块的属性,如果看到的是一系列优先级而不是属性列表,则在该模块上右击并选择 Property。如果右边窗口为灰色,则表明此模块没有全局属性。在模块名上右击并选择 Properties,打开属性窗口,可以更改该模块的属性。如果需要某个模块的帮助信息,单击图标 ,然后再单击该模块。

4. 创建对象

可使用配置工具来创建对象并设置每个对象的属性,也可以通过调用 XXX_create()函数来动态创建或删除对象。对典型的 DSP 应用而言,大多数对象都是通过配置工具创建的,因为这些对象在程序运行的全过程都要使用。一些默认的对象自动地在配置模板中定义。

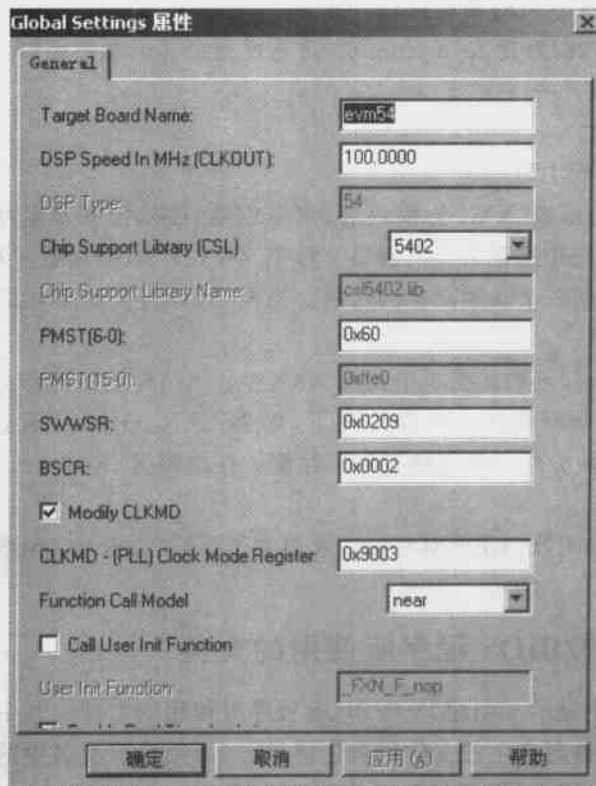


图 7.5 sd54.cdb 配置文件属性设置

使用配置工具创建对象具有以下好处：

- 能与 DSP/BIOS 插件更好地结合起来。系统日志显示了在配置工具下创建的对象名。此外，还可以只显示用配置工具创建的对象统计数据。
- 可以减小代码量。对一个典型的模块而言，XXX_create() 和 XXX_delete() 函数包含了实现此模块的 50% 的代码量。使用配置工具创建对象则可以极大地减少应用程序的代码量。
- 改善运行性能。除了节省代码空间外，还可以避免动态创建及缩短程序执行时进行系统设置所花费的时间。

使用配置工具创建对象存在以下局限性：

- 可能会创建根本就不需要的对象。如果一些对象在运行时不常用到，则应动态创建对象。
- 不能使用 XXX_delete() 函数来删除在配置工具下创建的对象。当使用 XXX_delete() 函数来删除在配置工具下创建的对象时，要调用 SYS_error() 函数。

使用配置工具创建对象的方法如下：

- (1) 在模块名上右击，并在弹出的菜单中选择 Insert MOD (MOD 为模块名，如模块名为 CLK，则选择 Insert CLK)，为此模块添加一个新对象。
- (2) 在新建对象上右击，并在弹出的菜单中选择 Rename，重新为对象命名。
- (3) 在新建对象上右击，并在弹出的菜单中选择 Properties。

注意:当指定一个被不同对象调用的 C 函数时,应在 C 函数名前加一个下划线(如调用一个名为 myfunc 的 C 函数,则键入 _myfunc)。这是因为配置工具产生汇编源代码,而从汇编语言中调用 C 函数将需要一个下划线。

(4) 更改属性设置。

5. 动态创建和删除对象

使用 XXX_creat() 函数(XXX 为模块名)可以创建许多(但并不是所有的)DSP/BIOS 对象。但是,有些对象只能在配置工具下创建。每个 XXX_creat() 函数分配用于存放对象内部状态信息的内存,并返回一个新建对象的句柄。当调用 XXX 模块提供的其他函数时,该句柄包含了新建对象的信息。

大多数 XXX_creat() 函数接受一个指向 XXX_Attrs 结构的指针作为其最后一个参数,该指针为新创建的对象分配属性。通常情况下,如果 XXX_creat() 函数的最后一个参数是 NULL,新创建的对象被分配一系列缺省的属性值。在结构 XXX_Attrs 中包含了对象的缺省属性值。

使用 XXX_creat() 函数创建的对象可以通过调用 XXX_delete() 函数来释放分配的内存,供以后使用。

7.4.2 创建 DSP/BIOS 程序所使用的文件

在如图 7.6 所示中显示了创建 DSP/BIOS 程序时使用的文件。其中:用户创建的文件用白色背景表示;保存文件时产生的文件用灰色背景表示;文件扩展名中的 54 是芯片代码的缩写,代表 C5000 系列器件。下面对这些文件进行说明。

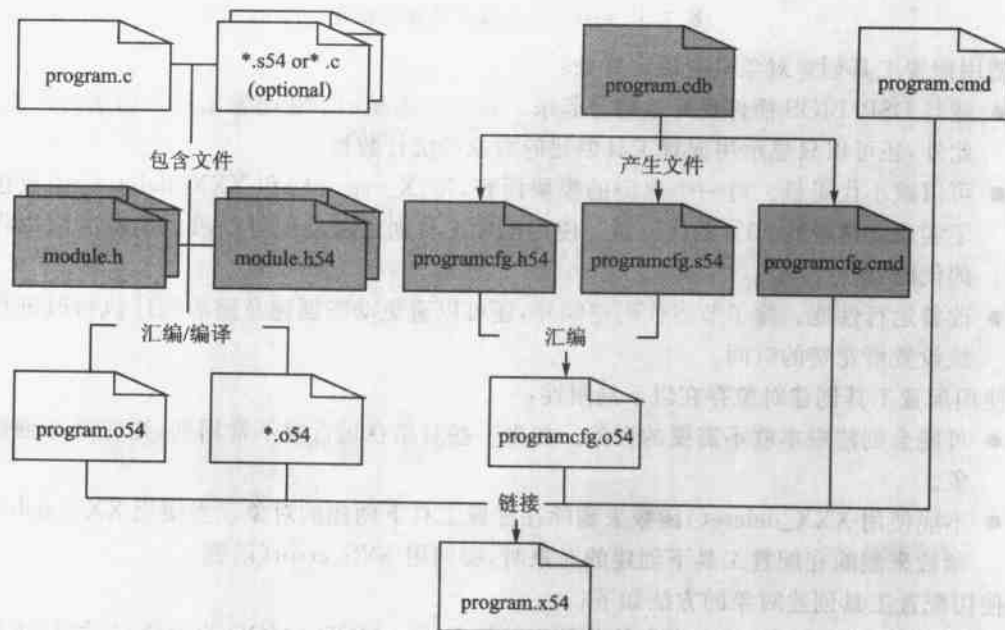


图 7.6 创建 DSP/BIOS 程序时使用的文件

- `program.c` 包含有 `main` 函数的 C 程序。除此 C 文件以外,还可以有其他 C 文件和头文件。
- `*.s54` 可选的汇编源文件。一个汇编源文件可以包含一个名为 `_main` 的函数,此时可以不使用 `program.c` 文件。
- `module.h` C 程序的 DSP/BIOS API 头文件。在 C 源文件中应包括 `std.h` 及其他模块使用的头文件。
- `module.h54` 汇编程序的 DSP/BIOS API 头文件。
- `program.o54` 源文件经编译或汇编产生的目标文件。
- `program.cdb` 配置文件,用于保存配置设置。此文件由配置工具产生,可由配置工具和 DSP/BIOS 插件使用。
- `programcfg.h54` 由配置工具产生的头文件,此头文件由 `programcfg.s54` 所包含。
- `programcfg.s54` 配置工具产生的汇编源代码。
- `programcfg.cmd` 由配置工具产生的链接命令文件,在链接生成可执行文件时使用。此文件定义了 DSP/BIOS 应用程序的链接选项、目标名及 DSP 程序的通用数据块(如, `text`、`bss` 和 `data` 等)。
- `programcfg.o54` 由配置工具产生的源文件生成的目标文件。
- `program.cmd` 可选的链接命令文件,其中包括了配置工具中未定义的其他段。
- `program.o54` 经完全编译、汇编和链接后产生的 DSP 应用程序,可在 CCS2 中加载并运行此程序。

在以上文件中,DSP/BIOS 插件使用以下两个文件。

- `program.cdb` DSP/BIOS 插件使用这个配置文件来获得对象名及其他程序信息。
- `program.out` DSP/BIOS 插件使用这个可执行文件来获得符号地址及其他程序信息。

7.4.3 编译和链接 DSP/BIOS 应用程序

在生成 DSP/BIOS 程序时,可以使用 CCS2 的工程,也可以使用自己的 `makefile`(在 CCS2 中包含了 `gmake.exe` 可执行程序),但一般情况下,不使用 `makefile`。

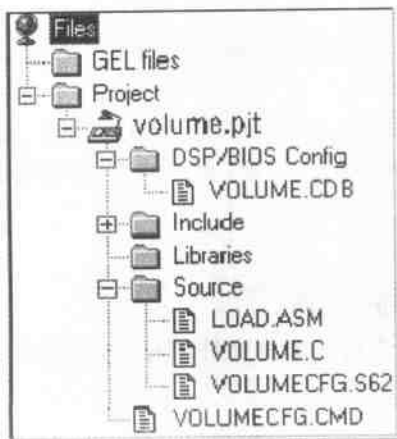


图 7.7 DSP/BIOS 工程

1. 用 CCS2 编译和链接

当使用 CCS2 工程来生成一个 DSP/BIOS 应用程序时,必须将 `program.cdb`(配置文件)和 `programcfg.cmd`(链接命令文件)添加到工程中。CCS2 会自动将 `programcfg.s54`(配置源文件)添加到工程中。

在一个 DSP/BIOS 应用程序中,`programcfg.cmd` 就是工程的链接命令文件。此文件中已包含了搜索所需库文件(如 `bios.a54`、`rtdx.lib` 和 `rts5401.lib` 等)的链接指令,因此不需要人工将这些文件添加到工程中。CCS2 还会自动扫描工程文件相关树,将必要的 DSP/BIOS RTDX 的头文件添加到工程 `include` 目录中,如图 7.7 所示。

大多数情况下,由配置工具产生的链接命令文件 `programcfg.cmd` 已足够描述所有的存储段的分配情况。所有的 DSP/BIOS 存储段和对象都由此命令文件来管理。此外,常用的段(如. `text`、. `bss` 和. `data` 等)已在 `programcfg.cmd` 中包含,其位置和大小都可通过配置工具的 MEM 管理器控制,如图 7.8 所示。

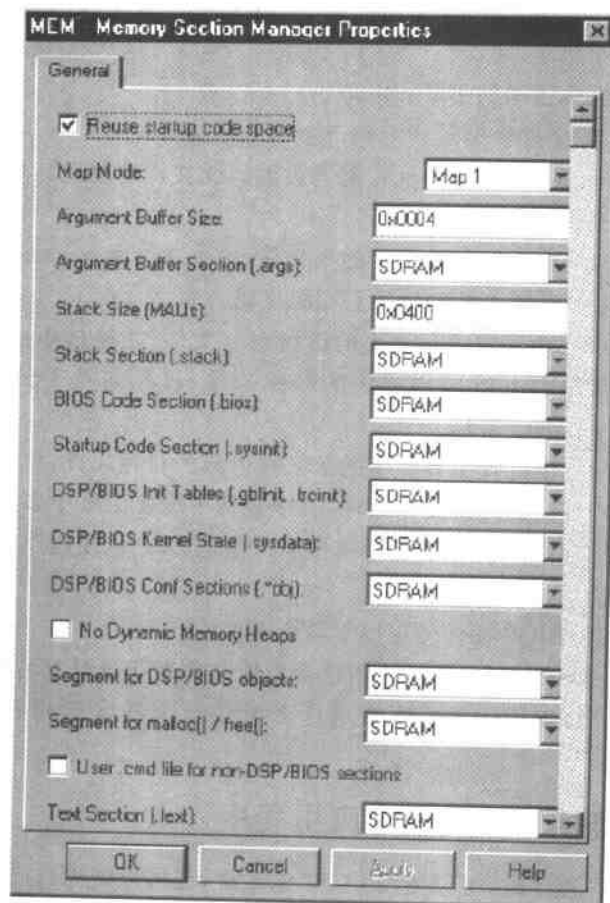


图 7.8 MEM 管理器

有些情况下,应用程序可能要求描述 `programcfg.cmd` 没有涉及到的与应用有关的段,这就需要使用其他的链接命令文件(`app.cmd`)。

注意:在 CCS2 中,一个工程只能有一个链接命令文件。当 `programcfg.cmd` 和 `app.cmd` 都需要时,则工程应使用 `app.cmd` 作为工程的链接命令文件。同时,为了在链接时包括 `programcfg.cmd`,应在 `app.cmd` 的第一行加入以下语句:

```
-l programcfg.cmd
```

这样, `programcfg.cmd` 就是链接器使用的第一个链接命令文件。

如果使用 `makefile` 来生成 DSP 应用程序,则可允许多个链接命令文件的存在,而且它们必须出现在 `programcfg.cmd` 文件之后。

2. 用 `makefile` 文件编译和链接

可以用 `makefile` 文件代替 CCS2 工程来编译和链接应用程序。一个典型的用来编译和链

接 DSP/BIOS 应用程序的 makefile 文件如下：

```
# Makefile for creation of program named by the PROG variable
#
# The following naming conventions are used by this makefile;
# <prog>.asm - C54 assembly language source file
# <prog>.obj - C54 object file (compiled/assembled source)
# <prog>.out - C54 executable (fully linked program)
# <prog>.cfg.s54 - configuration assembly source file
# generated by Configuration Tool
# <prog>.cfg.h54 - configuration assembly header file
# generated by Configuration Tool
# <prog>.cfg.cmd - configuration linker command file
# generated by Configuration Tool
#
include $(TI_DIR)/c5400/bios/include/c54rules.mak
#
# Compiler, assembler, and linker options.
#
# -g enable symbolic debugging
CC54OPTS = -g
AS54OPTS =
# -q quiet run
LD54OPTS = -q
# Every BIOS program must be linked with,
# $(PROG).cfg.o54 - object resulting from assembling
# $(PROG).cfg.s54
# $(PROG).cfg.cmd - linker command file generated by
# the Configuration Tool. If additional
# linker command files exist,
# $(PROG).cfg.cmd must appear first.
#
PROG = volume
OBS = $(PROG).obj load.obj
LIBS =
CMDS = $(PROG).cmd
#
# Targets;
#
all: $(PROG).out
$(PROG).out: $(OBS) $(CMDS)
$(PROG).cfg.obj: $(PROG).cfg.h54
$(PROG).obj:
$(PROG).cfg.s54 $(PROG).cfg.h54 $(PROG).cfg.cmd:
@ echo Error: $$ must be manually regenerated;
@ echo Open and save $(PROG).cdb within the BIOS Configuration Tool.
```

```
@ check $@  
.clean clean;:  
@ echo removing generated configuration files ...  
@ remove --f $(PROG)cfg.s54 $(PROG)cfg.h54 $(PROG)cfg.cmd  
@ echo removing object files and binaries ...  
@ remove -f *.obj *.out *.lst *.map
```

与 CCS2 不同的是,makefile 文件能够使用多个链接命令文件,如果应用程序需要额外的链接命令文件,开发者可以很容易地在上面 makefile 文件的 CMDS 变量中添加链接命令文件。但是,这些额外的命令链接文件必须在配置工具所产生的 programcfg.cmd 命令链接文件之后出现。

7.4.4 DSP/BIOS 应用程序执行顺序

当一个 DSP/BIOS 应用程序开始执行时,执行顺序由启动文件 boot.cs54 中的指令所决定。在 bios.s54 库中提供了一个已经编译好的启动文件,开发者不需要改变启动文件 boot.cs54 中的任何内容。

DSP/BIOS 应用程序的执行顺序如下:

(1) 初始化 DSP。DSP/BIOS 应用程序从 C 语言环境入口点 c_int00 处开始执行。当复位后,复位矢量把程序入口点设置到 c_int00 处,同时堆栈指针 SP 指向堆栈的末端,状态寄存器 ST0 和 ST1 也被初始化。当堆栈指针 SP 设置好后,将调用初始化例程来初始化, cinit 记录中的变量。

(2) 调用 BIOS_init 初始化 DSP/BIOS 模块。BIOS_init 由配置工具所产生,位于 programcfg.s54 文件中,它负责基本模块的初始化,同时包含了每个 DSP/BIOS 模块的 MOD_init 宏,具体过程如下:

① HWL_init 清除 IFR 寄存器。

② HST_init 初始化主机 I/O 通道接口。HST_init 的具体操作依赖于主机与 DSP 目标系统连接的特定应用。

③ 如果在配置工具中的空闲函数 (Idle function) 管理器中选择了空闲循环指令自动计数框,IDL_init 将在这里开始计数空闲循环指令执行次数。空闲循环指令计数框用来显示所计算的 CPU 载入图中的 CPU 负荷。

(3) 调用程序的主函数。当所有的 DSP/BIOS 模块初始化完成后,主函数被调用。主函数可以用汇编语言或者 C 语言编写。由于 C 编译器在主函数前自动地加上一个下划线的前缀,因此主函数的形式可以是 C 语言的 main() 或者汇编语言的 _main()。启动文件中的函数将传递参数 argc、argv 和 envp 给主函数,这三个参数分别代表 C 语言命令行计数、命令行参数数组大小和环境变量数值。在此阶段,由于没有任何硬件或软件中断被打开,开发者可以在初始化过程中调用自己的硬件初始化程序。

(4) 调用 BIOS_start,开始执行 DSP/BIOS。与 BIOS_init 一样,BIOS_start 也由配置工具产生,位于 programcfg.s54 文件中。BIOS_start 在主函数返回后立即被调用,它主要负责使能 DSP/BIOS 模块,同时为所有 DSP/BIOS 模块调入 MOD_startup 宏。具体过程如下:

① CLK_startup 宏 设置 PRD 寄存器,使能在 CLK 管理器中选择的计数器在 IMR 寄

寄存器中的位,然后启动计数器。该宏只有在配置工具中使能了 CLK 管理器时才起作用。

- ② PIP_startup 为所有创建的管道对象调用 notifyWriter 函数。
- ③ SWI_startup 使能所有软件中断。
- ④ TSK_startup 使能任务调度。
- ⑤ HWI_startup 通过清除 ST1 寄存器中的 INTM 位使能硬件中断。

(5) 进入空闲循环周期。通过调用 IDL_loop 函数,启动程序进入 DSP/BIOS 空闲循环周期,并一直保持循环状态。此时硬件中断或软件中断可以执行,以终止空闲状态。由于在空闲循环周期中管理主机与 DSP 目标系统的通信,主机与 DSP 目标系统间的数据传输就在空闲循环周期中执行。

启动文件的代码如下所示:

```
; ===== boot.s54 =====
.include bios.h54
.c_mode
.mmregs
CONST_COPY .set 0
*****
* This module contains the following definitions : *
* __stack - Stack memory area *
* _c_int00 - Boot function *
* _var_init - Function which processes initialization tables *
*****
.ref cinit, pinit
.global _c_int00
.global _main, __STACK_SIZE
; alternate label for c_int00 -- referenced by HWI_RESET
; to pull in BIOS boot code instead of rts.lib
.global BIOS_reset
*****
* Declare the stack. Size is determined by the linker option - stack. The *
DSP/BIOS Startup Sequence
Program Generation 2 - 15
* default value is 1K words. *
*****
__stack: .usect ".stack",0
args .sect ".args"
*****
* FUNCTION DEF : _c_int00 *
* 1) Set up stack *
* 2) Set up proper status *
* 3) If "cinit" is not - 1, init global variables *
* 4) call users' program *
*****
.sect ".sysinit"
```

```

BIOS_reset:
_c_int00:
xc 1, unc
ssbx intm ; set interrupt mask bit
stm #0, imr ; disable all interrupts

*****
* INIT STACK POINTER. REMEMBER STACK GROWS FROM HIGH TO LOW ADDRESSES. *
*****
STM #__stack, SP ; set to beginning of stack memory
ADDM #(__STACK_SIZE--1), *(SP) ; add size to get to top
ANDM #0fffh, *(SP) ; make sure it is an even address
SSBX SXM ; turn on SXM for LD #cinit, A

*****
* SET UP REQUIRED VALUES IN STATUS REGISTER *
*****
SSBX CPL ; turn on compiler mode bit
RSBX OVM ; clear overflow mode bit

*****
* SETTING THESE STATUS BITS TO RESET VALUES. IF YOU RUN _c_int00 FROM *
* RESET, YOU CAN REMOVE THIS CODE *
*****
LD #0, ARP
RSBX C16
RSBX CMPT
RSBX FRCT

*****
* SETUP PMST - GBL_PMST is defined by DSP/BIOS Configuration Tool
* The PMST must be initialized before the .pinit section is processed since
* the RTDX initialization triggers an interrupt (IVTP is in PMST). If
* the Interrupt Vector Table Pointer is not correct, the processor will
* execute an undefined interrupt vector.
*****
.ref GBL_PMST
stm #GBL_PMST, pmst

*****
* IF cinit IS NOT -1, PROCESS INITIALIZATION TABLES *
* TABLES ARE IN PROGRAM MEMORY IN THE FOLLOWING FORMAT: *
DSP/BIOS Startup Sequence
* . word <length of init data in words> *
* . word <address of variable to initialize> *
* . word <init data> *
* . word ... *
* The cinit table is terminated with a zero length *
*****
.if __far_mode

```

```

LDX #cinit,16,A
OR #cinit,A,A
.else
LD #cinit,A ; Get pointer to init tables
.endif
ADD #1,A,B
BC DONE_CINIT,BEQ ; if (cinit == -1) no init tables
*****
* PROCESS INITIALIZATION TABLES. TABLES ARE IN PROGRAM MEMORY IN THE *
* FOLLOWING FORMAT; *
* . word <length of init data in words> *
* . word <address of variable to initialize> *
* . word <init data> *
* . word ... *
* The init table is terminated with a zero length *
*****
BD START_CINIT ; start processing
RSBX SXM ; do address arithmetic unsignedly
nop
LOOP_CINIT:
READA *(AR2) ; AR2 = address
ADD #1,A ; A += 1
RPT *(AR1) ; repeat length+1 times
READA *AR2+ ; copy from table to memory
ADD *(AR1),A ; A += length (READA doesn't change A)
ADD #1,A ; A += 1
START_CINIT:
READA *(AR1) ; AR1 = length
ADD #1,A ; A += 1
BANZ LOOP_CINIT,*AR1- ; if (length-- != 0) continue
DONE_CINIT:
*****
* IF pinit IS NOT -1, PROCESS INITIALIZATION TABLES *
* TABLES ARE IN PROGRAM MEMORY IN THE FOLLOWING FORMAT; *
* .if __far_mode *
* . long <32-bit address of initialization routine to call> *
* .else *
* . word <16-bit address of initialization routine to call> *
* .endif *
* The pinit table is terminated with a NULL pointer *
DSP/BIOS Startup Sequence
Program Generation 2-17
*****
SSBX SXM
FRAME -4

```

可以根据硬件定义配置设置,以便将其作为模板使用,而不必每次都重新配置。比如,要为 C5000 定点 DSP 程序创建一个 DSP/BIOS 程序,可以使用 C5000 环境提供的设置,也可以添加一个定制的模板,加入定点的运行库支持,以便下次可以直接使用该配置。

定制模板的步骤如下:

(1) 在 Windows 环境下执行 Start→Programs→Code Composer Studio' C5000→Configuration Tool。

(2) 在 File 菜单下选择 New,弹出 New 窗口,选择 sd54.cdb。这时将弹出如图 7.4 所示的树状形窗口。

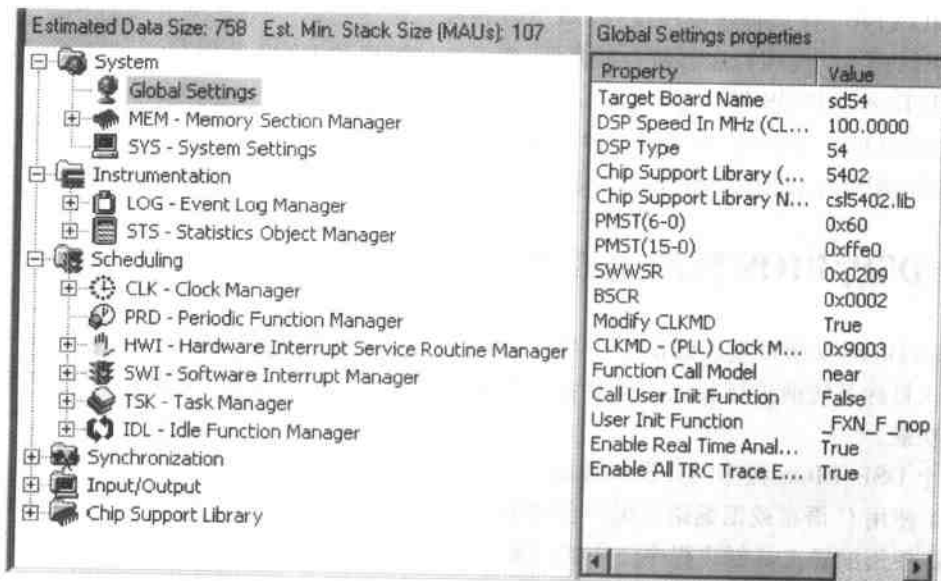


图 7.4 sd54.cdb 配置文件


(3) 在 Global Settings 上右击并选择 Property,将弹出如图 7.5 所示的属性设置窗口。

(4) 在 Target Board Name 栏改名为 evm54。

(5) 在 File 菜单下选择 Save As,将文件存在 D:/application/ccs/c5400/bios/include 目录下,取名为 evm54.cdb,在保存类型栏选择 Seed files(*.cdb),保存文件。

(6) 在 File 菜单下选择 Exit,退出配置工具。

3. 设置模块的全局属性

在如图 7.4 所示的窗口中,左边窗口显示的是模块,右边窗口显示的是该模块的属性。当双击一个模块时,右边的窗口将显示此模块的属性,如果看到的是一系列优先级而不是属性列表,则在该模块上右击并选择 Property。如果右边窗口为灰色,则表明此模块没有全局属性。在模块名上右击并选择 Properties,打开属性窗口,可以更改该模块的属性。如果需要某个模块的帮助信息,单击图标,然后再单击该模块。

4. 创建对象

可使用配置工具来创建对象并设置每个对象的属性,也可以通过调用 XXX_create()函数来动态创建或删除对象。对典型的 DSP 应用而言,大多数对象都是通过配置工具创建的,因为这些对象在程序运行的全过程都要使用。一些默认的对象自动地在配置模板中定义。


```

CALL _const_init
.endif
.endif

*****
* CALL BIOS_init
DSP/BIOS Startup Sequence
*****
call BIOS_init ; initialize the BIOS
; cpl = 1 when return from BIOS_init
*****
* Set up C environment before calling main(argc, argv, envp). *
*****
ld #args,b
stlm b,ar1
nop ; these 2 nops are necessary for
nop ; the latency of "stlm b, ar1"
ld *ar1+,a ; a = argc
frame - 2
ld *ar1 ; ,b
stl b,*sp(0) ; sp(0) = argv
ld *ar1,b
stl b,*sp(1) ; sp(1) = envp
*****
* CALL main()
*****
,if __far_mode ; Use far calls for C548 in far mode
FCALL _main ; far call to the user's entry point
,else
CALL _main
.endif
frame 2
*****
* CALL BIOS_start()
*****
call BIOS_start ; 'start' the BIOS
*****
* DROP INTO IDL LOOP
*****
rsbx cpl ; cpl = 0 is precondition of IDL_loop
IDL_loop ; fall into BIOS "idle task", never
; return
,if CONST_COPY
*****
* FUNCTION DEF : __const_init *
* COPY ,CONST SECTION FROM PROGRAM TO DATA MEMORY *

```

注意:当指定一个被不同对象调用的 C 函数时,应在 C 函数名前加一个下划线(如调用一个名为 myfunc 的 C 函数,则键入 _myfunc)。这是因为配置工具产生汇编源代码,而从汇编语言中调用 C 函数将需要一个下划线。

(4) 更改属性设置。

5. 动态创建和删除对象

使用 XXX_creat() 函数(XXX 为模块名)可以创建许多(但并不是所有的)DSP/BIOS 对象。但是,有些对象只能在配置工具下创建。每个 XXX_creat() 函数分配用于存放对象内部状态信息的内存,并返回一个新建对象的句柄。当调用 XXX 模块提供的其他函数时,该句柄包含了新建对象的信息。

大多数 XXX_creat() 函数接受一个指向 XXX_Attrs 结构的指针作为其最后一个参数,该指针为新创建的对象分配属性。通常情况下,如果 XXX_creat() 函数的最后一个参数是 NULL,新创建的对象被分配一系列缺省的属性值。在结构 XXX_Attrs 中包含了对象的缺省属性值。

使用 XXX_creat() 函数创建的对象可以通过调用 XXX_delete() 函数来释放分配的内存,供以后使用。

7.4.2 创建 DSP/BIOS 程序所使用的文件

在如图 7.6 所示中显示了创建 DSP/BIOS 程序时使用的文件。其中:用户创建的文件用白色背景表示;保存文件时产生的文件用灰色背景表示;文件扩展名中的 54 是芯片代码的缩写,代表 C5000 系列器件。下面对这些文件进行说明。

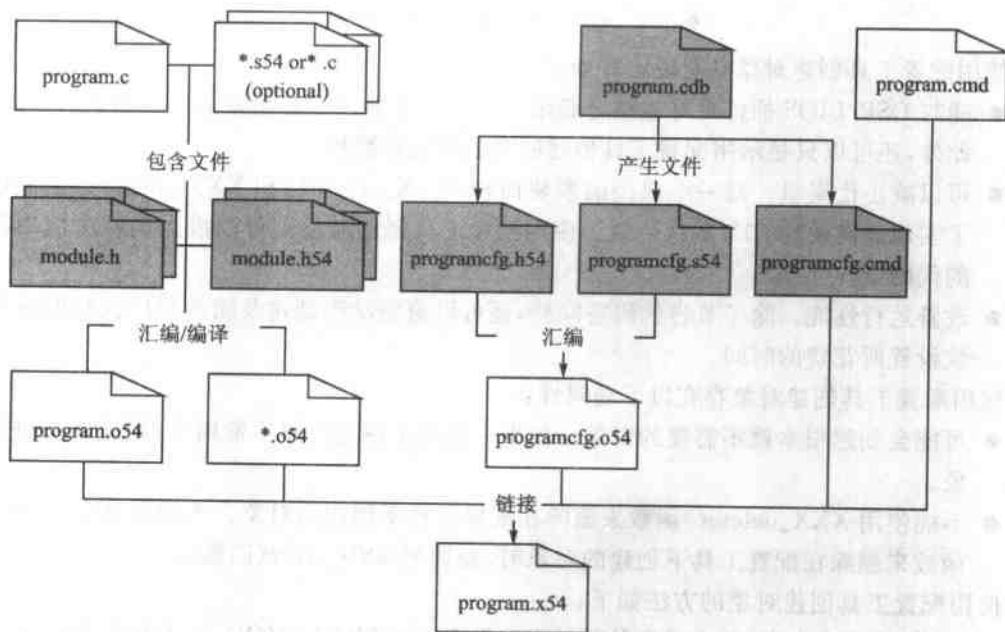


图 7.6 创建 DSP/BIOS 程序时使用的文件

```

.include bios, h54
.c_mode
.mmregs
CONST_COPY, set 0
*****
* This module contains the following definitions ; *
* __stack - Stack memory area *
* _c_int00 - Boot function *
* _var_init - Function which processes initialization tables *
*****

.ref cinit, pinit
.global _c_int00
DSP/BIOS Startup Sequence
.global _main, __STACK_SIZE
; alternate label for c_int00 -- referenced by HWL_RESET
; to pull in BIOS boot code instead of rts, lib
.global BIOS_reset
*****
* Declare the stack. Size is determined by the linker option - stack. The *
* default value is 1K words. *
*****
__stack: .usect " , stack", 0
args .sect " , args"
*****
* FUNCTION DEF ; _c_int00 *
* 1) Set up stack *
* 2) Set up proper status *
* 3) If "cinit" is not -1, init global variables *
* 4) call users' program *
*****
.sect " , sysinit"
BIOS_reset:
_c_int00:
xc 1, unc
ssbx intrm; set interrupt mask bit
stm #0, imr; disable all interrupts
*****
* INIT STACK POINTER. REMEMBER STACK GROWS FROM HIGH TO LOW ADDRESSES. *
*****
STM #__stack, SP; set to beginning of stack memory
ADDM #(__STACK_SIZE - 1), * (SP) ; add size to get to top
ANDM #0fffh, * (SP); make sure it is an even address
SSBX SXM; turn on SXM for LD # cinit, A
*****
* SET UP REQUIRED VALUES IN STATUS REGISTER *

```

```

*****
SSBX CPL; turn on compiler mode bit
RSBX OVM; clear overflow mode bit
*****
* SETTING THESE STATUS BITS TO RESET VALUES. IF YOU RUN _c_init0 FROM *
* RESET, YOU CAN REMOVE THIS CODE *
*****
LD #0,ARP
RSBX C16
RSBX CMPT
RSBX FRCT
*****
DSP/BIOS Startup Sequence
* SETUP PMST - GBL_PMST is defined by DSP/BIOS Configuration Tool
* The PMST must be initialized before the .pinit section is processed since
* the RTDX initialization triggers an interrupt (IVTP is in PMST). If
* the Interrupt Vector Table Pointer is not correct, the processor will
* execute an undefined interrupt vector.
*****
.ref GBL_PMST
stm #GBL_PMST, pmst
*****
* IF cinit IS NOT - 1, PROCESS INITIALIZATION TABLES *
* TABLES ARE IN PROGRAM MEMORY IN THE FOLLOWING FORMAT: *
* .word <length of init data in words> *
* .word <address of variable to initialize> *
* .word <init data> *
* .word ... *
* The cinit table is terminated with a zero length *
*****
.if __far_mode
LDX #cinit,16,A
OR #cinit,A,A
.else
LD #cinit,A ; Get pointer to init tables
.endif
ADD #1,A,B
BC DONE_CINIT,BEQ; if (cinit == - 1) no init tables
*****
* PROCESS INITIALIZATION TABLES. TABLES ARE IN PROGRAM MEMORY IN THE *
* FOLLOWING FORMAT: *
* .word <length of init data in words> *
* .word <address of variable to initialize> *
* .word <init data> *
* .word ... *

```

```

* The init table is terminated with a zero length *
*****
RSBX SXM; do address arithmetic unsignedly
.if __far_mode
.else
NOP ; pipe delay
LD #cinit,A; don't want this sign extended anymore
.endif
B START_CINIT; start processing
LOOP_CINIT;
READA *(AR2); AR2 = address
DSP/BIOS Startup Sequence
ADD #1,A; A += 1
RPT *(AR1); repeat length+1 times
READA *AR2+; copy from table to memory
ADD *(AR1),A; A += length (READA doesn't change A)
ADD #1,A; A -= 1
START_CINIT;
READA *(AR1); AR1 = length
ADD #1,A; A += 1
BANZ LOOP_CINIT,*AR1-; if (length-- != 0) continue
DONE_CINIT;
*****
* IF pinit IS NOT -1, PROCESS INITIALIZATION TABLES *
* TABLES ARE IN PROGRAM MEMORY IN THE FOLLOWING FORMAT: *
* .if __far_mode *
* .long <32-bit address of initialization routine to call> *
* .else *
* .word <16-bit address of initialization routine to call> *
* .endif *
* The pinit table is terminated with a NULL pointer *
*****
SSBX SXM
FRAME - 4
.if __far_mode
LDX #pinit,16,A
OR #pinit,A,A; A = &pinit table
.else
LD #pinit,A; A = &pinit table
.endif
ADD #1,A,B; B = A + 1
BC DONE_PINIT,BEQ; if (pinit == -1) no pinit tables
.if __far_mode
.else
RSBX SXM; do address arithmetic unsignedly

```

```

NOP ; pipe delay
LD #pinit,A; don't want this sign extended anymore
.endif
BD START_PINIT
DST A, @2; save 32 - bit PGM pointer on stack
NOP
LOOP_PINIT;
. if __far_mode
FCALA B; call function
DSP/BIOS Startup Sequence
Program Generation 2 - 23
. else
CALA B; call function
.endif
DLD @2, A; put 32 - bit PINIT pointer in A
START_PINIT;
READA @0; "push" (MSB) address of function
. if __far_mode
ADD #1, A
READA @1; "push" LSB address of function
.endif
. if __far_mode
ADD #1, A
DST A, @2
DLD @0, B
BC LOOP_PINIT,BNEQ
. else
LD @0, B; "pop" address of function
BCD LOOP_PINIT,BNEQ ; if not NULL, loop.
ADDM #1,@3; move PINIT pointer (in stack)
.endif
DONE_PINIT;
RSBX SXM
FRAME 4
*****
* COPY .CONST SECTION
*****
. if CONST_COPY
. if __far_mode ; Use far calls for C548 in far mode
FCALL _const_init ; move .const section to
DATA mem
. else
CALL _const_init
.endif
.endif

```

```

*****
* CALL BIOS_init
*****
call BIOS_init ; initialize the BIOS
; cpl = 1 when return from BIOS_init
*****
* Set up C environment before calling main(argc, argv, envp). *
*****
ld #args,b
stlm b,ar1
nop ; these 2 nops are necessary for
nop ; the latency of "stlm b, ar1"
ld *ar1+,a; a = argc
DSP/BIOS Startup Sequence
frame -2
ld *ar1+,b
stl b,*sp(0); sp(0) = argv
ld *ar1,b
stl b,*sp(1); sp(1) = envp
*****
* CALL main()
*****
.if __far_mode ; Use far calls for C548 in far mode
FCALL _main ; far call to the user's entry point
.else
CALL _main
.endif
frame 2
*****
* CALL BIOS_start()
*****
call BIOS_start ; start the BIOS
*****
* DROP INTO IDL LOOP
*****
rsbx cpl; cpl = 0 is precondition of IDL_loop
IDL_loop ; fall into BIOS "idle task", never
; return
.if CONST_COPY
*****
* FUNCTION DEF ; __const_init *
* COPY .CONST SECTION FROM PROGRAM TO DATA MEMORY *
* The function depends on the following variables *
* defined in the linker command file *
* __c_load ; global var containing start *

```

```

* of .const in program memory *
* __const_run ; global var containing run *
* address in data memory *
* __const_length ; global var length of .const *
* section *
*****
.global __const_length, __c_load
.global __const_run
__const_init;
.sect ".c_mark" ; establish LOAD adress of
.label __c_load ; .const section
.sect ".sysinit"
DSP/BIOS Startup Sequence
Program Generation *****
* C54X VERSION *
*****
LD #__const_length, A
BC __end_const, AEQ
STM #__const_run, AR2 ; Load RUN address of .const
RPT #__const_length - 1
MVPD #__c_load, *AR2+ ; Copy .const from program to data
*****
* AT END OF .CONST SECTION RETURN TO CALLER *
*****
__end_const;
.if __far_mode
FRET
.else
RET
.endif ; __far_mode
.endif ; CONST_COPY
.end

```

7.5 DSP/BIOS 仪表

DSP/BIOS 提供了显式的或隐式的实时程序分析手段。这种分析机制对 DSP 应用程序的实时运行影响很小。

7.5.1 实时分析

实时分析就是在 DSP 应用系统实时运行期间对应用系统数据进行分析,以便判别系统是否满足设计要求,是否还有继续开发的余地。传统的调试方法是让程序顺序执行,如果发生错误,停止程序运行,检查程序状态,然后插入断点,重新运行程序以获得程序运行期间的信息。这种调试方法在非实时顺序软件的开发中是有效的,但在实时系统中是无效的,这是因为实时

系统是以程序的连续操作、不确定运行和严格的时序要求为特征的。

DSP/BIOS 仪表类 API 函数和 DSP/BIOS 插件是对传统调试工作的补充,可以用来监视程序的实时运行。

7.5.2 软件仪表与硬件仪表

软件仪表是由在目标板上的与仪表有关的代码组成,这些代码在程序运行时执行,设计者所关心的数据存储的目标系统的存储器中。因此仪表代码使用了目标系统的存储器和 DSP 的运算能力。

软件仪表的优势在于其非常灵活且不需要额外的硬件,但是,由于仪表代码是目标应用程序的一部分,因此对系统的性能有一定影响。在不使用硬件仪表的情况下,设计者应考虑的是既要记录足够的信息,而又不影响系统正常工作。有限的软件仪表功能已不能提供足够的实时运行信息,过多的软件仪表功能又可能使被测量的系统实时性能下降到不可承受的程度。在这种情况下,DSP/BIOS 提供了一系列机制,使设计者能在搜集信息和干扰系统之间寻求平衡。此外,所有 DSP/BIOS 仪表操作都有一个固定的较短的执行时间。由于时间开支是固定的,因此仪表的影响可以预先知道。

7.5.3 仪表性能

当所有的隐式仪表使能时,CPU 负荷将增加不到 1%,这是由于采用了以下措施来减小仪表对应用系统性能的影响。

- 目标板与主机之间的通信是在后台线程(IDL)中执行的。后台线程具有最低优先级,因此仪表数据的通信不会影响应用系统的实时性能。
- 在主机端,可控制主机查询目标板的速率。如果要消除对目标板的所有不必要干扰,还可停止所有主机与目标板的交互。
- 除非跟踪(tracing)被使能,否则目标板不会存储执行图或隐式统计信息。可通过使用 TRC 模块或保留的跟踪掩码(TRC_USER0AK TRC_USER1)来使能或禁止显式仪表。
- 日志(LOG)或统计数据均在主机方处理。STS 对象、CPU 负荷的平均值及显示执行图(execution graph)所需的计算也在主机方处理。
- LOG、STS 及 TRC 模块操作速度很快,其执行时间是一个常量:
 - LOG_printf 与 LOG_event 执行时间:大约 30 个指令周期;
 - STS_add 执行时间:大约 30 个指令周期;
 - STS_delta 执行时间:大约 40 个指令周期;
 - TRC_enable 与 TRC_disable 执行时间:大约 4 个指令周期。
- 每个 STS 对象在数据存储器中只占用 8 个字的空间,也就是说,主机只需传送 8 个字就可以从一个统计对象中上载数据。
- 统计操作在主机方使用一个 64 位的变量,在目标板使用一个 32 位的变量。当主机向目标板查询实时统计数据时,首先复位目标板的变量,这样,就不需要在目标板方提供一个保存历史数据的空间。运行时间越长,统计数据越大,占用的空间也越大。
- 可以指定 LOG 对象的缓冲大小。缓冲大小直接影响了程序的数据区大小和上载

LOG 数据所需的时间。

- 考虑到性能要求,在默认情况下隐式硬件中断监视功能被禁止,此时对硬件性能不会有任何影响。当此功能被使能时,对于监视中断中的每个中断,更新统计对象数据大约占用 20~30 个指令周期。

7.5.4 仪表 API

高效的仪表既需要搜集数据操作,又需要响应程序事件控制数据的搜集操作。DSP/BIOS 提供以下 4 个 API 模块用于数据搜集。

- LOG(Event Log Manager) 日志对象实时捕获事件信息。系统事件通过系统日志捕获,在配置工具下也可以创建其他日志,在程序中还可以向任何日志添加消息。
- STS(Statistics Object Manager) 统计对象实时捕获任何变量的计数值、最大值及总和值。有关 SWI(软件中断)、PRD(周期函数)、HWI(硬件中断)及 PIP(管道)对象的统计数据将被自动捕获。同时,在程序中还可以创建统计对象来捕获其他统计数据。
- HST(Host Channel Manager) 主机通道对象允许程序将原始数据送往主机用于分析使用。
- TRC(Trace Manager) 用于控制由目标程序实时捕获或通过 DSP/BIOS 插件捕获的事件及统计数据。

LOG 和 STS 为捕获高频发生事件的实时序列或快速变化的统计数据值提供了一种有效的手段。由于事件发生和数据变化的速率可能很高,而带宽是有限的,所以不可能将所有的序列传送给主机,或者即使能将这些序列传送给主机也可能干扰程序的实时运行。因此 DSP/BIOS 还提供了一个 TRC API 模块,用于控制其他模块提供的数据搜集机制。

控制数据搜集是一件非常重要的工作,因为它可以限制仪表对实时程序的影响,确保 LOG 和 STS 对象包含必要的信息,在运行时开始或停止录制事件和数据值。

DSP/BIOS API 专门为实时 DSP 程序优化,可以为嵌入式程序提供基本的运行服务。与标准 C 库函数(如 puts 函数)不同,它可在不中断目标板硬件的情况下对 DSP 系统进行实时分析。同时,DSP/BIOS API 代码占用更少空间,运行速度比标准 C 输入/输出更快。一个 DSP 程序可以根据需要使用一个或多个 DSP/BIOS 模块。

7.6 创建一个 DSP/BIOS 程序

本节通过一个简单的例子来介绍如何使用 DSP/BIOS 创建、生成、调试和测试程序。该例子就是本书最常用的“hello world”程序。在前面的章节中,程序使用标准的 C 输入函数,在本节中,改变工程则使用 DSP/BIOS 功能。利用 CCS2 的剖析特性可以比较标准输入函数和利用 DSP/BIOS 函数执行的性能。值得注意的是,开发 DSP/BIOS 应用程序时仅有 Simulator(软件仿真器)是不行的,还需要使用 Emulator(硬件仿真器)和 DSP/BIOS 插件(安装时装入)。

7.6.1 打开存在的工程

打开存在的工程的步骤如下:

- (1) 在 D:\application\ccs\myprojects 下创建一个新的文件夹 hellobios。
- (2) 将文件夹 D:\application\ccs\tutorial\sim54xx\hello1 中所有文件拷贝到新建的 hellobios 文件夹中。
- (3) 如果 CCS2 还没运行,启动 CCS2。
- (4) 打开 D:\application\ccs\myprojects\hellobios 下的 hello. pjt 工程。
- (5) CCS2 会弹出一个如图 7.9 所示的对话框,提示没找到库文件,这是因为工程被移动了。单击 Browse 按钮,在 C:\application\ccs\c5400\cgtools\lib 下找到 rts. lib 库文件。

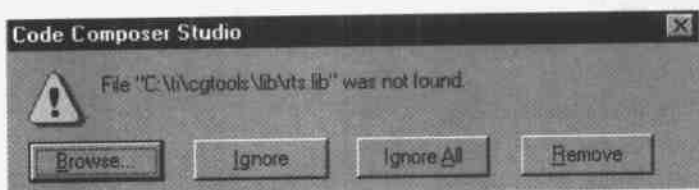


图 7.9 提示没找到库文件

- (6) 单击 hello. pjt、Libraries 和 Source 旁边的“+”号,展开工程视图。
- (7) 双击 hello. c 程序,将其打开。如果显示了汇编指令,在 View 菜单中选择 View mixed Source/ASM 命令隐藏汇编指令,并检查 hello. c 源代码。

```
/* HELLO.C */
#include <stdio.h>
#include "hello.h"
#define BUFSIZE 30
struct PARMS str =
{
    2934,
    9432,
    213,
    9432,
    &str
};
/* ===== main ===== */
void main()
{
    #ifdef FILEIO
        int    i;
        char    scanStr[BUFSIZE];
        char    fileStr[BUFSIZE];
        size_t   readSize;
        FILE    * fptr;
    #endif

    /* write a string to stdout */
    puts("hello world! \n");
    #ifdef FILEIO
```

```

/* clear char arrays */
for (i = 0; i < BUFSIZE; i++) {
    scanStr[i] = 0 /* deliberate syntax error */
    fileStr[i] = 0;
}

/* read a string from stdin */
scanf("%s", scanStr);


/* open a file on the host and write char array */
fptr = fopen("file.txt", "w");
fprintf(fptr, "%s", scanStr);
fclose(fptr);

/* open a file on the host and read char array */
fptr = fopen("file.txt", "r");
fseek(fptr, 0L, SEEK_SET);
readSize = fread(fileStr, sizeof(char), BUFSIZE, fptr);
printf("Read a %d byte char array: %s \n", readSize, fileStr);
fclose(fptr);

#endif
}

```

7.6.2 剖析 stdio.h 的执行时间

可以利用 CCS2 的剖析特性来测试调用 puts 函数的指令执行周期数。在 Project 菜单中选择 Rebuild All 命令或者在 Project 工具栏单击  图标,对工程进行编译和链接,其步骤如下:

- (1) 在 File 菜单中选择 Load Program 命令,载入编译好的 hello.out 文件。由于调用的 puts 函数向标准控制台输出,Stdout 窗口便显示在 CCS2 窗口中。
- (2) 在 Profiler 菜单中选择 Start New Session 命令,同时选择 disable use of RTDX 栏。
- (3) 在如图 7.10 所示的 MySession 剖析窗口中选择 Ranges 选项,在工程视图中双击 hello.c 文件,选择调用 puts 函数的代码行。
- (4) 将选择的代码行拖动到 MySession 剖析窗口中,如图 7.10 所示。MySession 剖析窗口中具体的代码行号也许不同,视代码行在程序中的具体位置而定。

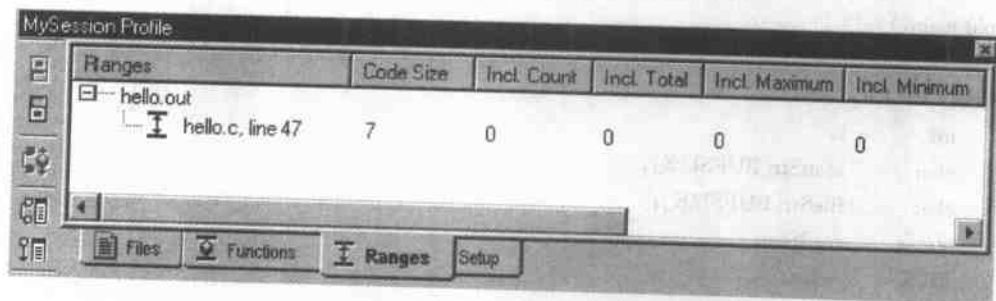



图 7.10 MySession 剖析窗口

(5) 执行菜单命令 Debug→Run, 或者在 Debug 工具栏上单击  按钮运行程序, 并在 MySession 剖析窗口中, 观察 Incl. Total 栏的数字。由于这个函数只运行一次, Total、Maximum 和 Minimum 栏的数值是一样的。Incl. Total 栏显示了与该行代码相关的汇编语言代码从开始执行到执行完成所花的指令周期数, 如图 7.11 所示。这个数值可以和后面的 DSP/BIOS LOG_printf 函数的执行周期相比较。指令的实际数值可能会随着 DSP 平台的不同而不同。

(6) 在 MySession 剖析窗口中, 单击  (turn session off) 按钮释放剖析资源, 然后关闭剖析窗口。

(7) 在 Profiler 菜单中, 取消 Enable Clock 选项。

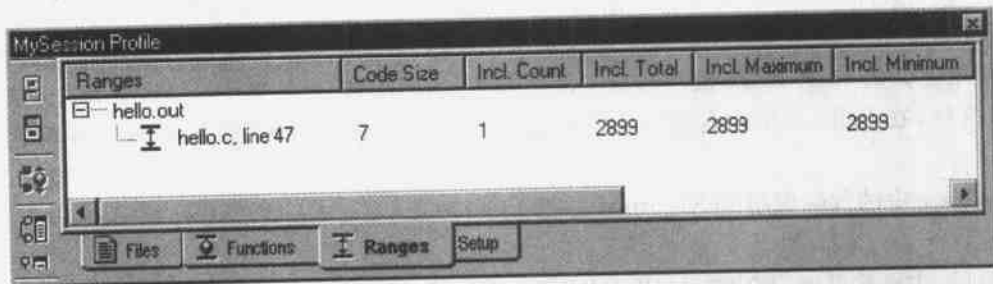


图 7.11 puts 函数执行的指令周期数

7.6.3 使用 HELLO2 文件

7.6.1 节中的 hello.c 程序使用的是标准 C 语言, puts 函数将消息“hello world”输出到标准控制台 stdout, hello2 程序使用了 DSP/BIOS 函数输出这个消息。对于使用 DSP/BIOS 程序而言, 让 DSP 目标系统在 PC 主机上输出这个消息的最有效的方式是使用 DSP/BIOS API 提供的 LOG 模块。DSP/BIOS API 模块在实时 DSP 运行中被优化, 与 puts 等传统的 C 语言库函数不同, DSP/BIOS 在不停止 DSP 目标硬件系统的情况下进行实时分析。此外, API 函数比标准的 C 语言 I/O 函数占用的空间和执行的时间少得多。

在 CCS2 中关闭 hello.c 窗口, 将 D:\application\ccs\tutorial\sim54xx\hello2 下的文件复制到 hellobios 文件夹中, 选择覆盖已经存在的文件。在工程视图中双击 hello.c 文件, 观察源代码:

```

/*****
 *   H E L L O . C
 *
 *   Basic LOG event operation from main.
 *****/
#include <std.h>
#include <log.h>
#include "hellocfg.h"
/* ===== main ===== */
Void main()
{
    LOG_printf(&trace, "hello world!");
}

```

```

/* fall into DSP/BIOS idle loop */
return;
}

```

这里要注意到改变的代码。

在 hello.c 文件中,包含了 std.h 和 log.h 头文件。使用 DSP/BIOS API 函数的程序必须包含 std.h 头文件和程序使用的 API 模块所需要的头文件。log.h 头文件定义了 LOG_Obj 结构并声明 LOG 模块中的 API 函数的操作。log.h 头文件必须放在任何 DSP/BIOS 模块的头文件前,其他的 DSP/BIOS 头文件的顺序没有要求。

hello.c 文件代码包含了 hellocfg.h 头文件,该头文件将在创建和保存了配置文件后产生。hellocfg.h 头文件包含在配置文件中定义的 DSP/BIOS 对象的额外声明,同时包含在配置文件中定义的 DSP/BIOS 模块的头文件。虽然在 hellocfg.h 头文件中包含了 std.h 文件和 log.h 文件,在 hello.c 中也包含了 std.h 文件和 log.h 文件,但同时包含头文件两次并不冲突。

hello.c 中的代码调用 LOG_printf 并把 LOG 对象(&trace)的地址和“hello world”信息传给它。

最后,主函数返回,程序进入 DSP/BIOS 空闲周期循环,DSP/BIOS 等待软件中断或者硬件中断等线程的产生。本例中没有这些线程。

值得注意的是,尽量避免同时使用 LOG 模块函数和标准的 C 语言输入/输出函数传输数据。标准的 C 语言输入/输出函数在 CCS2 中使用大量的内部断点在 PC 主机和 DSP 目标系统间传输数据,而断点在某些平台和一定的环境下会影响程序的实时性能。DSP/BIOS 使用 RTDX 实时地传输数据,不影响程序的实时执行。

7.6.4 创建一个配置文件

使用 DSP/BIOS API 时,程序必须有一个配置文件用来定义程序所需要的 DSP/BIOS 对象。创建配置文件的步骤如下:

- (1) 选择菜单命令 File→New→DSP/BIOS Configuration。
- (2) 选择与 DSP 仿真器相对应的模板并单击 OK 按钮,将出现如图 7.2 所示的窗口。窗口左半部分为 DSP/BIOS 模块及对象名,右半部分为模块和对象的属性。
- (3) 右击 LOG - Event Log Manager,在弹出的菜单中选择 Insert Log,此时创建一个被称为 LOG0 的 LOG 对象。
- (4) 右击 LOG0 对象,在弹出的菜单中选择 Rename,将对象更名为 trace,如图 7.12 所示。

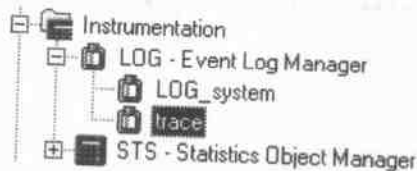


图 7.12 新创建的 LOG 对象

(5) 将配置文件存为 hello.cdb, 存盘到 D:\application\ccs\myprojects\hellobio 中, 此时将产生以下文件:

- hello.cdb 配置文件, 保存配置设置;
- hellocfg.cmd 链接命令文件;
- hellocfg.s54 汇编语言源代码;
- hellocfg.h54 myhellocfg.s54 包含的头文件;
- hellocfg.h 包含了 DSP/BIOS 模块的头文件, 声明在配置文件中所创建的对象的外部变量;
- hellocfg.c 芯片支持库 Chip Support Library (CSL) 的结构和设置的 C 源代码。

另外, 在工程视图目录中也产生了 hello.ccf 文件, 这是一个临时文件, 当配置文件打开时才存在, 对其不要打开、删除或者进行更改。

7.6.5 添加 DSP/BIOS 文件到工程

添加 DSP/BIOS 文件到工程的步骤如下:

(1) 执行菜单命令 Project→Add Files to Project, 将 hello.cdb 加入, 此时在工程视图中将添加一个名为 DSP/BIOS 的目录, hello.cdb 被列在该目录下。此外, CCS2 自动将 hellocfg.s54 和 hellocfg.c.c 等文件拷贝到 Generated Files 文件夹下。

(2) 链接的输出文件名必须与 .cdb 文件名一样, 在 Project→Options 的 Linker 栏中查看输出文件名是否为 hello.out, 如不是则将其更改为 myhello.out。

(3) 执行菜单命令 Project→Add Files to Project, 将 hellocfg.cmd 加入到 CCS2 中。由于一个工程中只能有一个链接命令文件, 所以, 此时会产生警告信息, 如图 7.13 所示。

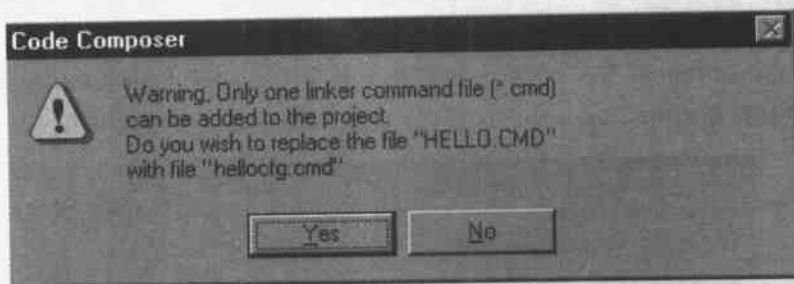


图 7.13 多个链接命令文件警告信息

(4) 单击 Yes 按钮, 用 myhellocfg.cmd 替换原来的 hello.cmd 命令文件。

(5) 在 Project 视图中双击 Vectors.asm, 在弹出的菜单中选择 Remove from project, 将 Vectors.asm 从工程中移去, 这是因为硬件中断矢量已在 DSP/BIOS 配置文件中自动定义。

(6) 将 rts.lib 文件从工程中移去, 因为此运行支持库也已在 myhellocfg.cmd 中指定, 在链接时将自动加入。

(7) 保存 hello.c。

(8) 执行菜单命令 Project→Options, 直接将 Compiler 栏的命令行参数“-d FILEIO”删除, 也可以在 Category→Symbolos→Define Symbolos 框中将 FILEIO 删除。

(9) Rebuild All 重新生成 DSP 程序。

改变前和改变后的工程视图如图 7.14 所示。

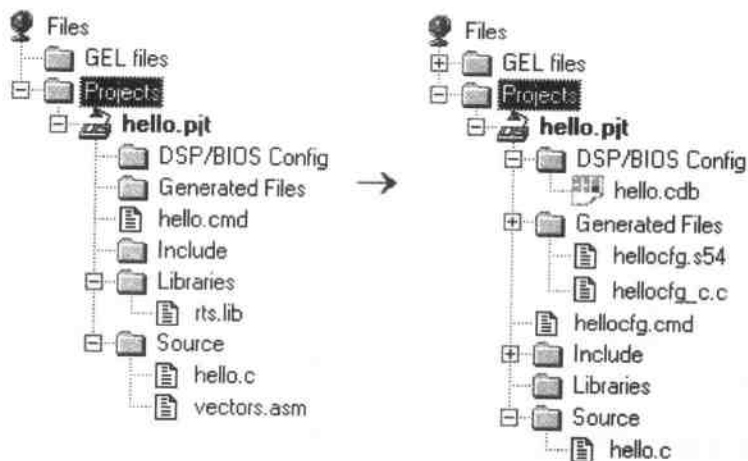


图 7.14 改变前和改变后的工程视图

7.6.6 用 CCS2 测试

用 CCS2 对程序进行测试的步骤如下：

- (1) 选择 File→Load Program 项,加载 hello.out 文件。
- (2) 选择 Debug→Go main 项,编辑窗口将显示 hello.c 文件内容,而且 main 函数的第一行被高亮显示,表明程序执行到此行后暂停。
- (3) 选择 Tools→DSP/BIOS→Message Log 项,此时将在 CCS2 窗口下方出现 Message Log 区域。
- (4) 在 Message Log 区右击,在弹出的菜单中选择 Property Page,在 Message Log 属性窗口的 Log to fi 栏输入 hello.txt 作为文件名,如图 7.15 所示。单击 OK 按钮。

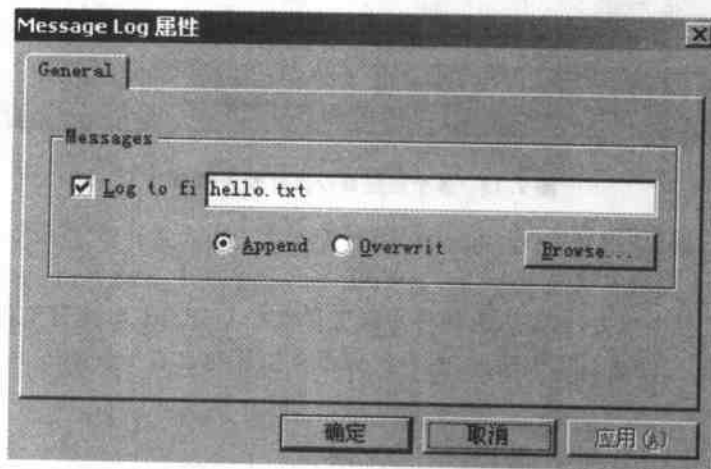


图 7.15 Message Log 属性窗口

- (5) 选择 trace 作为要观察的 LOG 名。默认的刷新速率为每次 1 s,要改变刷新速率,选

择 Tools→DSP/BIOS→RTA Control Panel 项,在新出现的窗口区域内右击并选择 Property Page,可选择一个新的刷新速率。

(6) 按 F5 键运行程序。当主函数返回时,程序进入 DSP/BIOS 空闲循环周期,等待事件的发生。

(7) 在 Message Log 区右击并选择 Close,为使用剖切功能(Profiler)作准备。

(8) 在 hello.txt 文件中包含了“hello world”信息。hello.txt 文件将产生在 D:\application\ccs\myprojects\hellobios\Debug 文件夹中。如果在关闭 Message Log 窗口前或者关掉了 Log to file 设置而打开 hello.txt 文件,将会发现该文件已被锁定。

(9) 选择 Tools→RTDX 项,任意打开一个 RTDX 控制窗口,从下接列表中选择 RTDX Disable,在 RTDX 区右击并选择 Hide。

DSP/BIOS 插件使用 RTDX 作为主机/目标板通信工具,在某些 DSP 目标板上不能同时使用 Profiler 和 RTDX。此时,使用剖切功能前关闭所有使用 RTDX 的 Tools,如 Message Log 和其他 DSP/BIOS 插件。为确保 RTDX 被禁用,尤其是在使用 DSP/BIOS 插件后,选择 Tools→RTDX 项,打开 RTDX 插件,从下拉列表中选择 RTDX Disable,然后在 RTDX 区右击并选择 Hide。

7.6.7 剖析 DSP/BIOS 代码执行时间

本节使用 CCS2 的剖析(profiler)获得 LOG_printf 的执行时间,其步骤如下:

(1) 执行菜单命令 File→Reload Program,重新加载 hello.out 文件。

(2) 执行菜单命令 Profiler→Enable Clock,使能时钟。

(3) 在 Profiler 菜单中选择 Start New Session 命令,同时选中 disable use of RTDX 栏。

(4) 在 MySession 剖析窗口中选择 Ranges 选项,在工程视图中双击 hello.c 文件,选择调用 LOG_printf 函数的代码行。

(5) 将选择的代码行拖动到 MySession 剖析窗口中,如图 7.16 所示。MySession 剖析窗口中具体的代码行号也许不同,视代码行在程序中的具体位置而定。

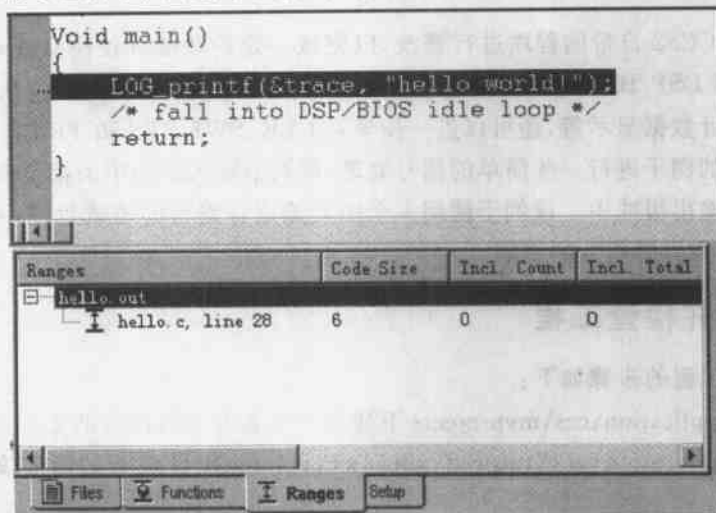



图 7.16 MySession 剖析窗口

(6) 执行菜单命令 Debug→Run, 或者在 Debug 工具栏上单击  按钮运行程序。在 MySession 剖析窗口中, 观察 Incl. Total 栏的数字。由于这个函数只运行一次, Total、Maximum 和 Minimum 栏的数值是一样的。Incl. Total 栏显示了与该行代码相关的汇编语言代码从开始执行到执行完成所花的指令周期数, 如图 7.17 所示。

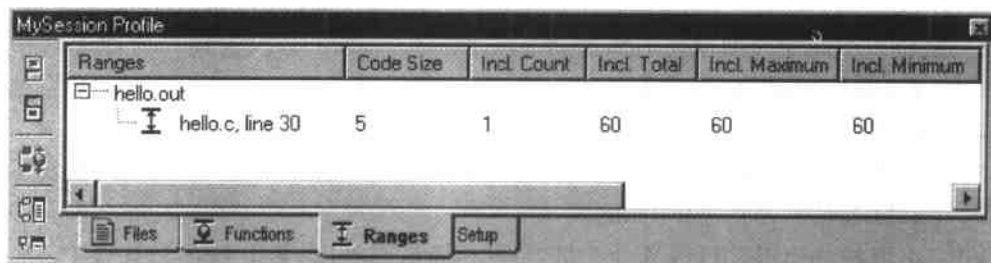


图 7.17 执行 LOG_printf 的指令周期数

可以看到 Incl. Total 栏的指令周期数约为 60, 即为执行 LOG_printf 的时间, 而在 7.6.2 节中调用 C 中的 puts 函数所需要的指令周期为 2899。调用 LOG_printf 函数比调用 C 中的 puts 函数更为有效, 这是因为字符串格式是在主机上而不是像 puts 函数那样在目标 DSP 上处理。使用 LOG_printf 函数监视系统状态对程序的实时运行影响比使用 puts 函数小得多。

(7) 按 Shift+F5 键停止程序运行。

(8) 执行以下操作以释放被 Profile 任务占用的资源:

- ① 选择 Profiler→Enable Clock 项, 禁止时钟;
- ② 关闭 MySession 剖析窗口(在窗口中右击并在弹出菜单中选 Hide);
- ③ 关闭所有源文件和配置窗口;
- ④ 选择 Project→Close 项, 关闭工程。

7.7 调试 DSP/BIOS 程序

本节对一个 CCS2 自带的程序进行修改, 以完成一定的功能并使用多线程运行。在调试过程中, 可以查看 DSP/BIOS 程序的性能, 学习 DSP/BIOS 更多的特性, 包括执行图、实时分析控制面板和统计数据显示等, 还可以进一步学习 CLK、SWI、STS 和 TRC 等模块的使用。

对本节使用的例子进行一些简单的信号处理, 能调节输入缓冲中的幅度值的增益, 并将处理后的结果放到输出缓冲中。该例子使用一个用汇编语言编写的装载程序, 利用传入的 processingLoad 参数来计算指令的周期。

7.7.1 打开并检查工程

打开并检查工程的步骤如下:

- (1) 在 D:\application\ccs\myprojects 下建立一个名为 volume2 的文件夹。
- (2) 将 D:\application\ccs\tutorial\sim54xx\volume2t 目录下的所有文件复制到此目录中。
- (3) 从桌面或“开始”菜单程序组中运行 CCS2。

(4) 执行菜单命令 Project→Open, 打开 Volume. pj1 工程文件。

(5) 在工程视图图中将工程展开, 如图 7.18 所示。本工程中包括以下文件:

- VOLUME.CDB 工程的配置文件;
- VOLUME.C 主程序的源文件;
- VOLUME.H VOLUME.C 使用的头文件, 定义各种常量和结构;
- LOAD.ASM 一个简单的汇编例程, 在 C 函数中调用, 带一个参数;
- VOLUMECFG.CMD 保存配置文件时产生的链接命令文件;
- VOLUMECFG.S54 保存配置文件时产生的汇编源文件;
- VOLUMECFG.H54 保存配置文件时产生的头文件;
- VOLUMECFG.H 当保存配置文件时会产生 VOLUMECFG.H 头文件, 其包含 DSP/BIOS 模块的头文件, 同时声明在配置文件中创建的对象的外部变量;
- VOLUMECFG_C.C 芯片支持库 CSL(Chip Support Library) 的结构和设置的 C 源代码文件。

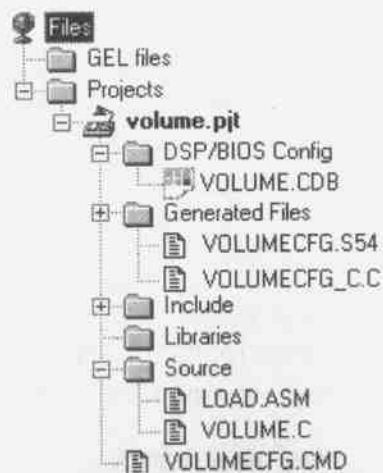


图 7.18 工程中包括的文件

7.7.2 观察源代码

前面介绍的程序中, 主函数处在一个无限循环当中, 而在实际应用程序中通常不会采用这种设计方法, 而是使用一个周期性的外部中断来完成数据的输入/输出任务。本例展示了一个应用程序如何使用 DSP/BIOS 去处理实时功能并管理通过外部事件触发所产生的数据的传输。其中一个简单的仿真周期性的外部中断的方法是使用片内定时器的定时中断。

双击工程视图图中的 VOLUME.C, 可以观察源代码内容。

```

/*      volume.c      */
#include <std.h>
#include <log.h>
#include <swi.h>
#include "volumecfg.h"
#include "volume.h"
/* Global declarations */
Int inp_buffer[BUFSIZE]; /* processing data buffers */
Int out_buffer[BUFSIZE];
Int gain = MINGAIN; /* volume control variable */
Uns processingLoad = BASELOAD; /* processing load value */
/* Functions */
extern Void load(Uns loadValue);
Int processing(Int *input, Int *output);

```

```

Void dataIO(Void):
/* ===== main ===== */
Void main()
{
    LOG_printf(&trace, "volume example started\n");
    /* fall into DSP/BIOS idle loop */
    return;
}

/* ===== processing ===== */
* FUNCTION:      Called from processing SWI to apply signal
* processing transform to input signal.
* PARAMETERS:    Address of input and output buffers.
* RETURN VALUE:  TRUE
Int processing(Int *input, Int *output)
{
    Int size = BUFSIZE;
    while(size--){
        *output++ = *input++ * gain;
    }
    /* additional processing load */
    load(processingLoad);
    return(TRUE);
}

/* ===== dataIO ===== */
* FUNCTION:      Called from timer ISR to fake a periodic
*                  hardware interrupt that reads in the input
*                  signal and outputs the processed signal.
* PARAMETERS:    none
* RETURN VALUE:  none
Void dataIO()
{
    /* do data I/O */
    /* post processing_SWI software interrupt */
    SWI_dec(&processing_SWI);
}

```

本程序有如下的一些特点:

- 改变了用大写的形式定义数据类型。DSP/BIOS 提供易移植到其他处理器的数据类型,很多 DSP/BIOS 使用的数据类型是相应的 C 类型的大写形式。
- 使用 #include 引用 3 个 DSP/BIOS 头文件 std. h、log. h 和 swi. h,其中 std. h 必须放在 log. h 和 swi. h 两个头文件之前。
- 代码中包含了 volumecfg. h 头文件,该头文件在更改或者保存配置文件时产生。本配置文件中创建的对象完全声明为 external。
- 主函数在调用 LOG_printf 后直接返回,不再调用 dataIO 和 processing 函数。主函数同时使程序进入 DSP/BIOS 空转循环(idle loop)中,由 DSP/BIOS 调度器管理线程的执行。

- Processing 函数由一个软件中断(processing_SWI)调用,其优先级比任何硬件中断都要低。DSP/BIOS 提供多种类型的线程,其中包括硬件中断、软件中断、任务和空闲线程等。不使用软件中断(SWI),硬件中断服务子程序也可直接执行信号处理函数,只是信号处理时间较长,甚至可能在下一个中断到来之前还未完成处理过程,因此有可能阻止中断的发生。
- DataIO 函数调用 SWI_dec,当软件中断到来时,它将计数器减 1。当计数值为 0 时,软件中断调度函数执行,同时复位计数器。DataIO 函数仿真基于硬件的数据 I/O。一个典型的程序会开设一个缓冲区收集数据,当数据足够时进行处理。本例中,dataIO 函数执行 10 次后会调用 processing 函数进行处理,SWI_dec 控制计数器完成计数功能。

7.7.3 修改配置文件

在本例中,配置文件已经被创建,下面检查在默认配置中已加入的对象,其步骤如下:

- (1) 在工程视图中双击 DSP/BIOS 文件夹下的 VOLUME.CDB,打开配置文件。
- (2) 在配置工具窗口中单击 CLK、LOG 和 SWI 管理器旁的“+”号,展开各个模块。配置文件除包含默认的对象外,还创建了这 3 个模块的对象。
- (3) 单击名为 trace 的 LOG 对象,在右边的窗口可看到此日志的属性,如图 7.19 所示。VOLUME.C 调用 LOG_printf,将“volume example started”信息写入此日志。

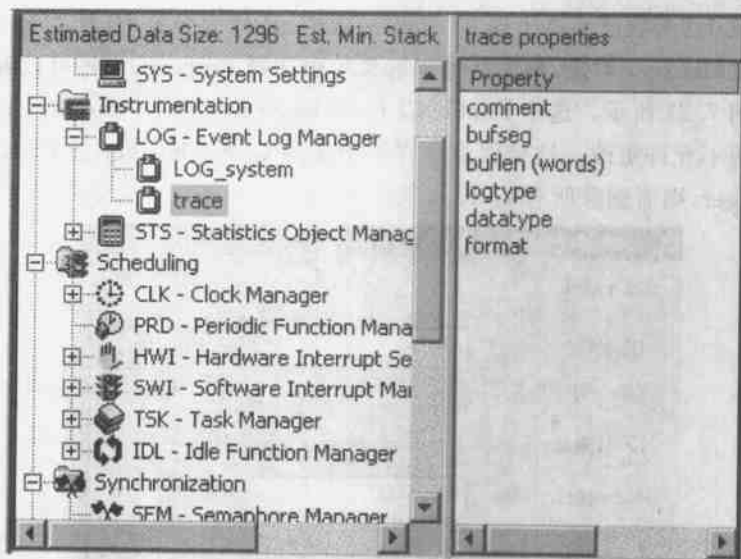


图 7.19 配置工具窗口

(4) 右击称为 LOG_system 的对象,从弹出的菜单中选择 Properties,将出现此对象的属性对话框,如图 7.20 所示。在程序运行时,此 LOG 保存了系统跟踪不同 DSP/BIOS 模块的事件。C 函数有一个下划线前缀,这是因为保存配置时产生了汇编语言文件。加下划线就可在汇编代码中调用 C 函数。此规则只适用于自编的 C 函数,对于调用的 DSP/BIOS API 或配置产生的对象不用加下划线前缀。因为每个对象会自动产生两个名字,其中一个有下划线,另一个没有下划线。

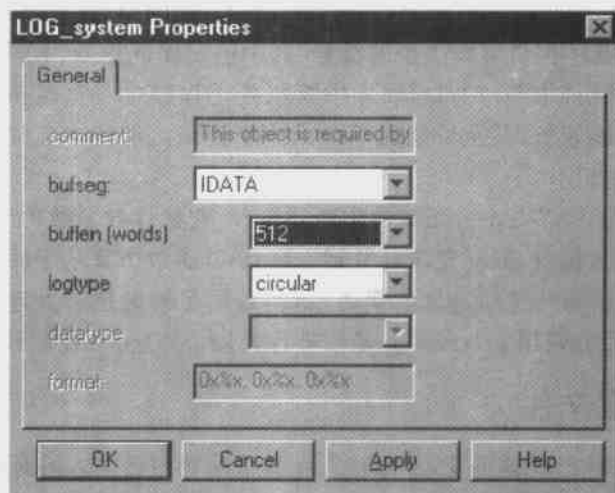


图 7.20 LOG_system 对象的属性

(5) 将 buflen 一栏内改为 data IO_CLK 并单击 OK 按钮。

(6) 双击被称为 dataIO_CLK 的 CLK 对象, 当此 CLK 对象激活时调用的函数为 -dataIO, 此函数即为 VOLUME.C 中的 dataIO 函数。

(7) 既然 dataIO 函数已不再出现在 main 函数中, 是什么导致 CLK 对象运行此函数呢? 右击 CLK_Clock Manager 对象, 然后从弹出的菜单中选择 Properties, 将可以看到时钟管理器属性对话框, 如图 7.21 所示。选中 Enable CLK Manager 时, 定时中断驱动 CLK Manager, 此栏显示灰色, 表明不允许更改。这是因为已有一个使用 CLK Manager 的 CLK 对象。如果要禁止 CLK Manager, 则需删除所有 CLK 对象。

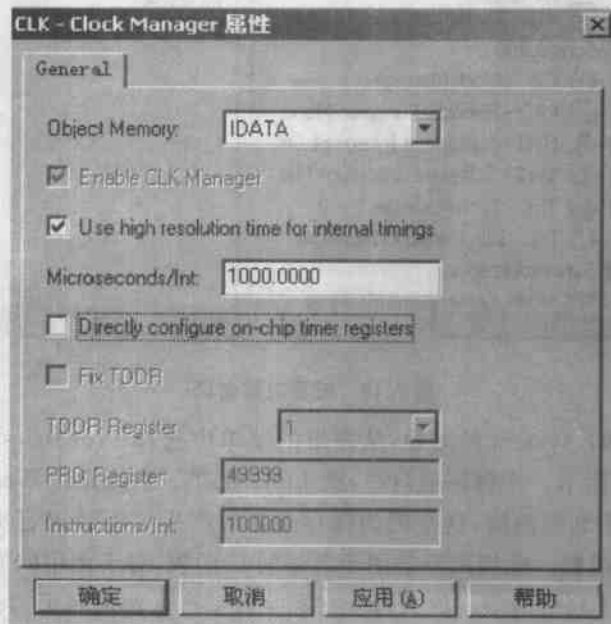


图 7.21 时钟管理器属性对话框

(8) 展开 HWI 对象, 右击 HWI_TINT 硬件中断对象, 在弹出的菜单中选择 HWI_TINT 对象的属性。HWI 的中断源为 DSP 定时器, 当片内定时器中断时, 运行一个名为 CLK_F_isr 的函数。CLK 对象函数在 CLK_F_isr 硬件中断函数中运行, 因此它比任何软件中断的优先级都要高, 能不受干扰地一直运行到结束。CLK_F_isr 会保存寄存器内容, 因此 CLK 函数不需要保存和恢复环境, 而通常的硬件中断服务程序则应进行保存和恢复环境。

(9) 在 processing_SWI 软件中断对象上右击, 在弹出的菜单中选择 Properties, 出现如图 7.22 所示的窗口。

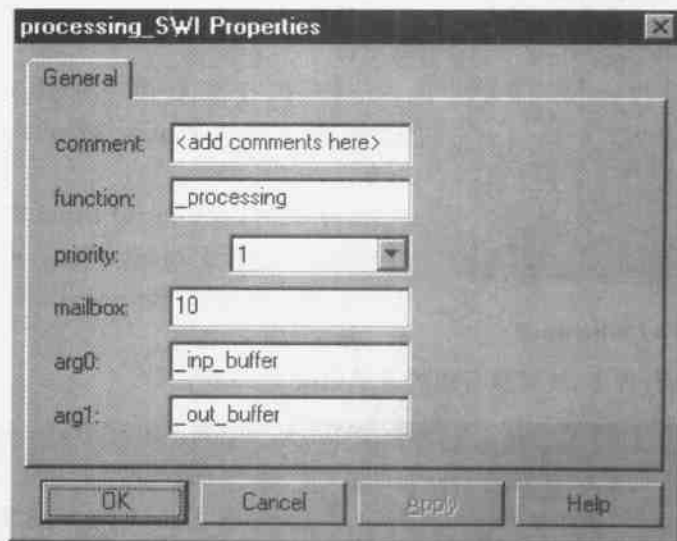


图 7.22 processing_SWI 软件中断对象的属性

- function 栏 当激活软件中断时, 将运行 _processing 函数。
- mailbox 栏 mailbox 的值能控制软件中断运行的时间。API 调用能影响 mailbox 的值, 根据其结果可发送软件中断。
- arg0 和 arg1 栏 _inp_buffer 和 _out buffer 将作为参数传给 _processing 函数。

(10) processing 函数不出现在 main 函数中, 但是, SWI 对象却运行此函数, 原因在于: 在 VOLUME.C 中, dataIO 函数调用 SWI_dec, 它对 mailbox 的值执行减 1 操作并在 mailbox 为 0 时发送软件中断。因此, 当 dataIO_CLK 对象运行 dataIO 函数 10 次时, 这个 SWI 对象运行 processing 函数。

(11) 执行 File 菜单下的 Close 命令, 单击 Yes 按钮保存 VOLUME.CDB。保存这个文件的同时将产生 VOLUME.CFB、CMD、VOLUME.CFB.S54 和 VOLUME.CFB.H54 文件。

(12) 执行菜单命令 Project→Build, 生成 DSP 程序。

7.7.4 使用执行图观察线程执行

在 Processing 函数中使用 Probe Point 观察输入/输出数据图形可以测试一个算法。为了测试 DSP/BIOS 的实时性, 假设信号处理算法已经经过测试, 以下步骤是测试线程是否满足实时操作需要。

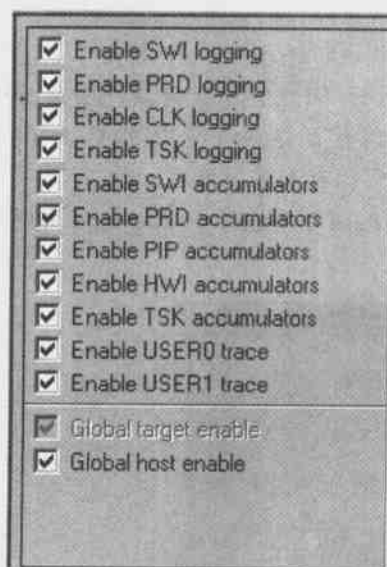


图 7.23 RTA Control Panel

(1) 执行菜单命令 File → Load Program, 在 volume2 目录下选择刚生成的 DSP 程序 VOLUME.OUT。

(2) 执行菜单命令 Debug → Go main, 程序执行到 main 函数的第一行。

(3) 执行菜单命令 DSP/BIOS → RTA Control Panel, 可在 CCS2 窗口下端观察到一系列仪表类型。在缺省状态下, 所有的仪表都是使能的, 如图 7.23 所示。

(4) 执行菜单命令 DSP/BIOS → Execution Graph, 执行图(execution graph)出现在 CCS2 窗口下方。

(5) 在 RTA Control Panel 窗口内右击, 并在弹出的菜单中选择 Property Page, 确保 Message Log/Execution Graph 的 Refresh Rate 为 1, 如图 7.24 所示。单击“确定”按钮。

(6) 按 F5 键运行程序, 线程执行图的显示如图 7.25 所示。

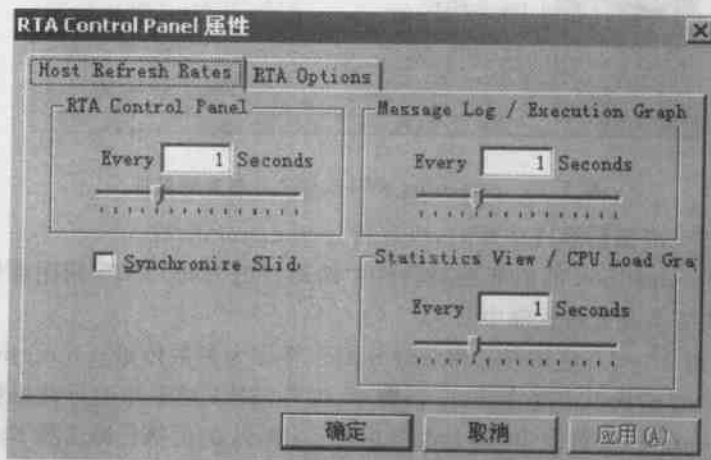


图 7.24 RTA Control Panel 的属性设置

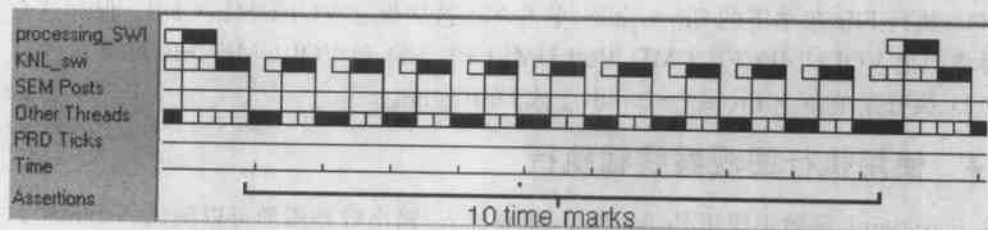


图 7.25 线程执行图

(7) Time 行的标记显示每次 Clock Manager 运行 CLK 函数的时间。可以看出,每次 Processing_SWI 对象运行的间隔为 10 个标记,表明 dataIO_CLK 对象每运行其函数 10 次则 Processing_SWI 对象运行其函数 1 次,这与 mailbox 的设置值(10)相符。

7.7.5 更改和观察 Load 函数的执行

通过线程执行图可以看出,程序能满足实时要求。然而一个典型的信号处理函数必须执行比乘一个数再送往输出缓冲这样的算法更复杂、更耗时的工作。可以模拟这种复杂的线程,只要把 load 函数执行周期数增大即可。

下面的 Load 值是针对运行在 100 MIPS 上的 TMS320C549DSP 芯片。如果 TMS320C54X 不是运行在 100 MIPS 上,就需要将 Load 值先乘以速率,再除以 100。如果不知道芯片运行速率的 MIPS 值,打开 VOLUME.CDB,再查看 Global Settings property 的 DSP MIPS(CLKOUT)。

(1) 执行菜单命令 DSP/BIOS→CPU Load Graph, 将出现一个空的 CPU Load Graph 窗口。

(2) 在 RTA 控制面板上右击,从弹出的菜单中选择 Property Page。

(3) 将 Statistics View/CPU Load Graph 的 Refresh Rate 改为 0.5 s 后单击 OK 按钮,可以看到 CPU Load 变得很低。Statistics View 和 CPU Load 功能在目标板和主机间传送的数据非常小,因此设置并周期性更新这些窗口对程序执行的影响也很小。Message Log 和 Execution Graph 功能传送配置文件中相关 LOG 对象的 buflen 长度的字数,传输的数据量较大,所以应降低这两个窗口的更新频率。

(4) 执行菜单命令 File→Load GEL,选择 VOLUME.GEL 并单击 OK 按钮。

(5) 执行菜单命令 GEL→Application Control→Load,出现 Load 调节窗口。

(6) 将 Load 改为 50,单击 Execute 按钮,CPU 负荷将增至约 9%,如图 7.26 所示。

(7) 在线程执行图上右击,在弹出的菜单中选择 Clear 项。可发现此时程序仍满足实时要求,因为每个 processing_SWI 执行间隔有 10 个标记符号。

(8) 在 Load 调节窗口将 Load 的值改为 200,单击 Execute 按钮。

(9) 在线程执行图上右击,在弹出的菜单中选择 Clear 项。此时每个 processing_SWI 执行间隔只有一个或两个标记符号,如图 7.27 所示。图 7.27 显示程序运行正常。运行 CLK 对象函数的硬件中断能够中断软件中断的处理过程,而软件中断也已在下次中断到来前完成其处理。

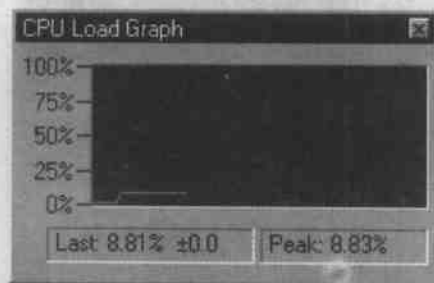


图 7.26 CPU 的负荷

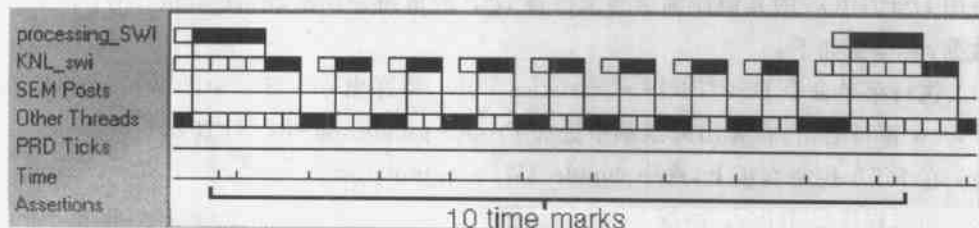


图 7.27 Load 值为 200 时的线程执行图

(10) 在 Load 调节窗口将 Load 改为 900, 单击 Execute 按钮。此时 CPU Load 增加大约 95%, 如图 7.28 所示。因此, 执行图和 CPU 负荷窗口的更新更缓慢了。

(11) 在线程执行图内右击, 从弹出的菜单中选择 Clear。因为 processing_SWI 在 10 个标记发生前已完成其操作, 因此程序仍能满足实时要求, 如图 7.29 所示。

(12) 在 Load 调节窗口将 Load 改为 1 200, 单击 Execute 按钮。CPU 负荷和线程执行图将不再频繁更新, 甚至根本就不更新, 这是因为执行图的相关数据在 idle 线程内送给主机, 其优先级是所有线程中最低的, 而高优先级线程用完了所有的处理时间。因此主机没有足够时间更新窗口, 此时程序就不能满足实时要求。

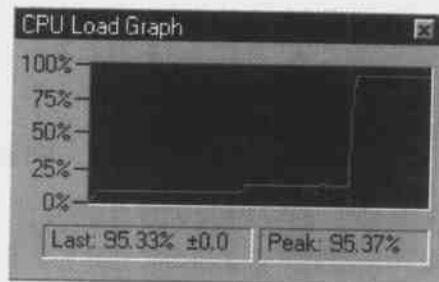


图 7.28 CPU Load 增加大到 95 %

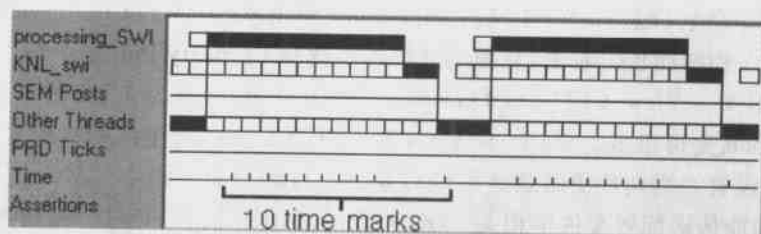


图 7.29 Load 值为 900 时的线程执行图

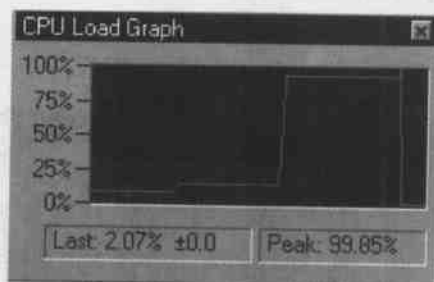


图 7.30 CPU Load 为 10 %

(13) 按 Shift+F5 键停止程序运行并更新线程执行图。此时在 Assertions 行有方块出现, 这表明 CCS2 已检测到应用程序不能满足实时要求。

(14) 在 Load 调节窗口将 Load 改为 10, 单击 Execute 按钮。此时 CPU 负荷和执行图重新被更新, 如图 7.30 所示。

使用 GEL 控制功能(本例为 Load 调节窗口)会临时中断 DSP 程序运行。如果正在分析一个实时系统, 而且又不想影响系统性能, 则可用 RTDX 改变

Load 值。有关 GEL 控制功能的使用将在第 8 章介绍。

7.7.6 分析线程统计

使用 DSP/BIOS 的其他控制功能来检查 DSP 的负荷并观察 processing_SWI 对象的处理统计数据, 其步骤如下:

(1) 执行菜单命令 DSP/BIOS→Statistics View, 将弹出一个 Statistics View 窗口, 在 Statistics View 窗口右击, 在弹出的菜单中选择 Property Page, 如图 7.31 所示。

(2) 在 RTA 控制面板上, 选中 enable SWI accumulators。

(3) 如果程序已中止运行, 按 F5 键运行程序。

(4) 观察 Statistics View 中的最大值, SWI 的统计数据是用指令周期来表示的。实际的

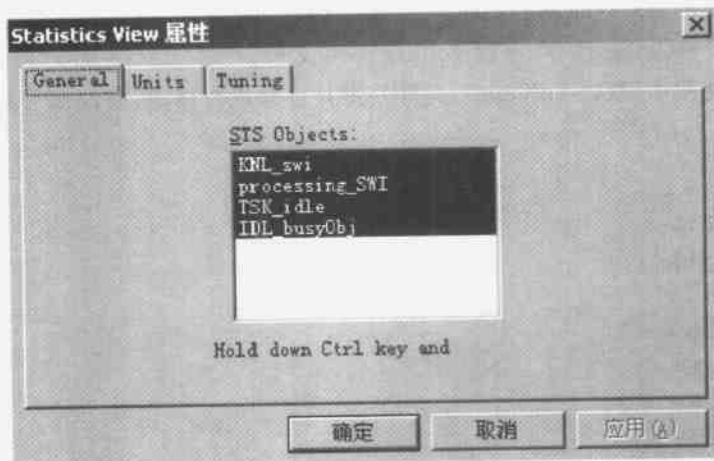


图 7.31 Statistics View 窗口

统计数值根据 DSP 目标平台的不同而有差异。然而当 LOAD 的值不变时, SWI 的最大值也保持不变, 如图 7.32 所示。

STS	Count	Total	Max	Average
KNL_swi	43	42792.00 inst	4420.00 inst	995.16 inst
processing_SWI	4	17268.00 inst	4318.00 inst	4317.00 inst
TSK_idle	0	0.00 inst	-2147483648.00 inst	0.00 inst
IDL_busyObj	194	-27870	-129	-143.66

保持不变

图 7.32 Statistics View 中值

(5) 在 Load 调节窗口增大 Load 值, 单击 Execute 按钮。可以看到执行 processing_SWI 函数的指令数的最大值增加了。

(6) 可以尝试将 Load 改为其他值, 观察统计数据的变化。如果减小了 Load 值, 则在 Statistics View 窗口右击, 并在弹出的菜单中选择 Clear 项。这样将会使所有观察对象复位到可能的最小值, 从而观察现在的 Max 指令周期值。

(7) 按 Shift+F5 键中止程序运行, 关闭所有打开的 DSP/BIOS 和 GEL 控制对象。

7.7.7 添加显式 STS 仪表

在 7.7.6 节中, 使用 Statistics View 来观察软件中断执行所用的指令数。如果使用一个配置文件, DSP/BIOS 将自动支持显示统计数据, 这称为隐式仪表; 也可以调用 API 函数获取其他的统计数据, 这称为显式仪表。添加显式 STS 仪表的步骤如下:

(1) 在工程视图中双击 VOLUME.CDB, 打开配置文件, 单击 Instrumentation 旁边的“+”号, 展开模块列表;

(2) 在 STS Manager 上右击并选择 Insert STS 项;

- (3) 将新建的 STS0 对象名更改为 processingLoad_STS, 保持其缺省的属性不变;
- (4) 执行 File→Close. 命令, 保存 volume. cdb 文件;
- (5) 在工程视图中, 双击 volume. c 文件并将其打开进行修改. 修改部分用黑体字标出, 如下所示;

```

/*      volume. c      */
#include <std. h>
#include <log. h>
#include <swi. h>
#include <clk. h>
#include <sts. h>
#include <trc. h>
#include "volumecfg. h"
#include "volume. h"
/* Global declarations */
Int inp_buffer[BUFSIZE];          /* processing data buffers */
Int out_buffer[BUFSIZE];
Int gain = MINGAIN;               /* volume control variable */
Uns processingLoad = BASELOAD; /* processing load value */
/* Functions */
extern Void load(Uns loadValue);
Int processing(Int *input, Int *output);
Void dataIO(Void);
/* ===== main ===== */
Void main()
{
    LOG_printf(&trace, "volume example started\n");
    /* fall into DSP/BIOS idle loop */
    return;
}
/* ===== processing ===== */
* FUNCTION;      Called from processing_SW1 to apply signal
*                processing transform to input signal.
* PARAMETERS;    Address of input and output buffers.
* RETURN VALUE;  TRUE.    */
Int processing(Int *input, Int *output)
{
    Int size = BUFSIZE;

    while(size--){
        *output++ = *input++ * gain;
    }

    /* enable instrumentation only if TRC_USER0 is set */
    if (TRC_query(TRC_USER0) == 0) {
        STS_set(&processingLoad_STS, CLK_gethtime());
    }
}

```

```

    }
    /* additional processing load */
    load(processingLoad);
    if (TRC_query(TRC_USER0) == 0) {
        STS_delta(&processingLoad_STS, CLK_gettime());
    }
    return(TRUE);
}

/* ===== dataIO ===== */
* FUNCTION:      Called from timer ISR to fake a periodic
*                hardware interrupt that reads in the input
*                signal and outputs the processed signal.
* PARAMETERS:    none
* RETURN VALUE:  none    */
Void dataIO()
{
    /* do data I/O */
    /* post processing_SWI software interrupt */
    SWI_dec(&processing_SWI);
}

```

(6) 执行 File→Close 命令,存盘退出;

(7) 在 Project 工具栏双击 Reuild All 图标,生成 DSP 程序。

7.7.8 观察显式仪表

本节的 Load 值也是针对运行在 100 MIPS 上的 TMS320C549DSP 芯片。如果 TMS320C54X 不是运行在 100 MIPS 上,就需要将 Load 值先乘以速率,再除以 100。如果不知道芯片运行速率的 MIPS 值,则打开 VOLUME.CDB,再查看 Global Settings property 的 DSP MIPS(CLKOUT)。

观察显式仪表的步骤如下:

(1) 执行菜单命令 File→Load Program,选择刚生成的 DSP 程序 VOLUME.OUT,单击 Open 按钮加载程序。

(2) 执行菜单命令 DSP/BIOS→RTA Control Panel,选中 enable SWI acumulators, enable USER0 trace 和 global host enable. Enable USER0 trace 将导致调用 TRC_query (TRC_USER0)后返回为 0。

(3) 执行菜单命令 DSP/BIOS→Statistics View,在 Statistics View 窗口右击,在弹出的菜单中选择 Property Page,将出现属性设置窗口。选中 processing_SWI 和 processingLoad_STS 对象,同时按 Ctrl 键,删除其他所有对象,如图 7.33 所示。

(4) 在 Statistics View 属性设置窗口上面单击 UnitsB 标签,设置 processingLoad_STS 对象的单位为 inc,设置 processing_SWI 对象的单位为 inst,如图 7.34 所示。

(5) 单击 OK 按钮,将看到 processing_SWI 和 processingLoad_STS 两个对象的统计域,将 Statistics View 窗口变为一个独立的窗口便能观察到所有的统计域。



图 7.33 只选择 processing_SWI 和 processingLoad_STIS 对象

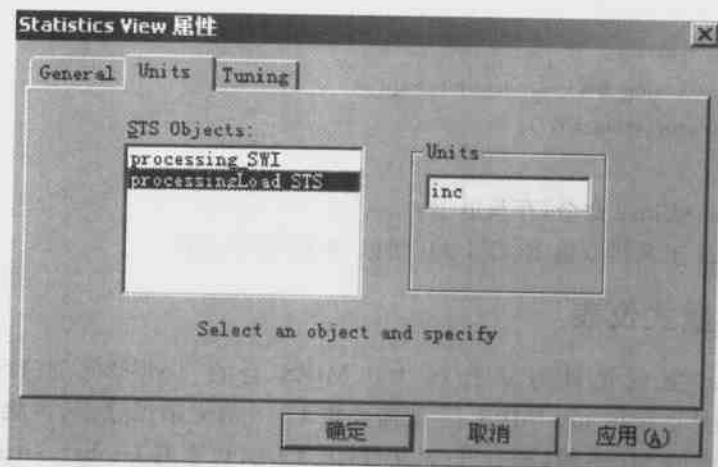


图 7.34 设置 processingLoad_STIS、processing_SWI 对象的单位

(6) 按 F5 键运行程序, 观察 Statistics View 窗口, 如图 7.35 所示。

STS	Count	Total	Max	Average
processing_SWI	3	12026.0...	4010.00 inst	4008.67 inst
processingLoad_STIS	3	1656 inc	552 inc	552 inc

图 7.35 processing_SWI 和 processingLoad_STIS 的统计值

(7) 将 processingLoad_STIS 的最大值乘以 2。SWI 统计是用指令数来计量的。由于程序中使用 CLK_getthtime 函数为 processingLoad 定标, 因此 processingLoad_STIS 统计以片内定时器的计数刻度为单位, 计数器的速率 = $\text{CLKOUT}/(\text{TDDR}+1)$ 。

其中 CLKOUT 为 DSP 的时钟速率 (单位: MIPS), 可在配置工具的 Global Settings

Property对话框中查到其值。TDDR 为定时器的 divide-down 寄存器值,可在配置工具的 CLK Manager Property 对话框中查到其值。因此,将 processingLoad_STS 单位转换为指令数,只需将其值乘以(TDDR+1)即可。例如,如果 DSP 运行速率为 100 MIPS, TDDR=1,则片内定时计数器以 50 MHz 的频率计数。在本例中,需将 processingLoad_STS 的值乘以 2 得到指令周期数。

(8) 用 processing_SWI 的最大值减去 processingLoad_STS 的最大值,其结果应大约为 2 792 个指令周期(实际结果可能有差异,特别是如果使用的是非通用的开发板)。这些指令为 processing 函数中除去 STS_set 和 STS_delta 之间的那部分指令,如下所示:

```

Int processing(Int *input, Int *output)
{
    Int size = BUFSIZE;
    while(size-- > 0)
    {
        *output++ += *input++; // *gain;
    }

    /* enable instrumentation only if TRC_USER0 is set */
    if(TRC_query(TRC_USER0) != 0)
    {
        processing_SWI 范围 {
            STS_set(&processingLoad_STS, CLK_gettime());
        }

        /* additional processing load */
        load(processingLoad);

        if(TRC_query(TRC_USER0) != 0)
        {
            STS_delta(&processingLoad_STS, CLK_gettime()); } processingLoad_STS 范围
        }

        return(TRUE);
    }
}

```

如果 TDDR=1, Load=10, 则 processingLoad_STS 最大值约为 5 085, 而 processing_SWI 的最大值约为 12 962。为了计算 processing 函数中除去 STS_set 和 STS_delta 之间的那部分指令周期数,可使用下式得出结果:

$$12\,962 - 5\,085 \times 2 = 2\,792$$

(9) 执行菜单命令 GEL→Application Control ▶Load(如果重新启动过 CCS2 则需重新加载 GEL 文件)。

(10) 改变 Load 值,单击 Execute 按钮,可以看到当两个最大值都变化时,其差值(processingLoad_STS 的最大值需乘以(TDDR+1))并没有改变。

(11) 在 RTA 控制面板内取消选中的 enable USER0 trace。

(12) 在 Statistics View 区右击并在弹出的菜单中选择 Clear 项,可以看到 processingLoad_STS 相关值并不改变,这是因为取消选中的 enable USER0 trace 后使程序中的表达式“TRC_query(TRC_USER1) == 0”为 FALSE。这样,程序将不调用 STS_set 和 STS_delta 两个函数。

(13) 按 Shift+F5 键停止程序运行,关闭所有 GEL 对话框、DSP/BIOS 插件及所有源文件。

第 8 章 CCS2 高级使用——RTDX 的原理与应用

8.1 RTDX 介绍

RTDX(Real-Time Data Exchange, 实时数据交换)提供一个实时和连续的可视化环境,使开发者能看到 DSP 应用程序工作的真实过程。RTDX 允许系统开发者在不停止运行目标应用程序的情况下,在计算机与 DSP 目标系统之间传输数据,同时还可在 PC 主机上利用对象链接嵌入(OLE)技术分析和观察数据。这样,可以提供给开发者一个真实的系统工作过程,从而缩短开发时间。

RTDX 可以在 DSP/BIOS 中使用,也可以脱离 DSP/BIOS 使用。目前,CCS Simulator 不支持 RTDX,RTDX 必须在 Emulator 下使用,因此要求有硬件仿真器和 DSP 目标板。

RTDX 的工作原理如图 8.1 所示,它由 PC 主机(Host)和目标(Target)DSP 两部分组成,CCS 则控制了 PC 主机与 DSP 之间的数据流动。在目标 DSP 上运行有一个小的 RTDX 库(RTDX target library),RTDX 库使用一个基于扫描的仿真器,通过增强型 JTAG 接口在 PC 主机和 DSP 目标系统之间传递数据。DSP 应用程序则通过调用 RTDX 库的 API 函数来完成 PC 主机与 DSP 之间的通信。

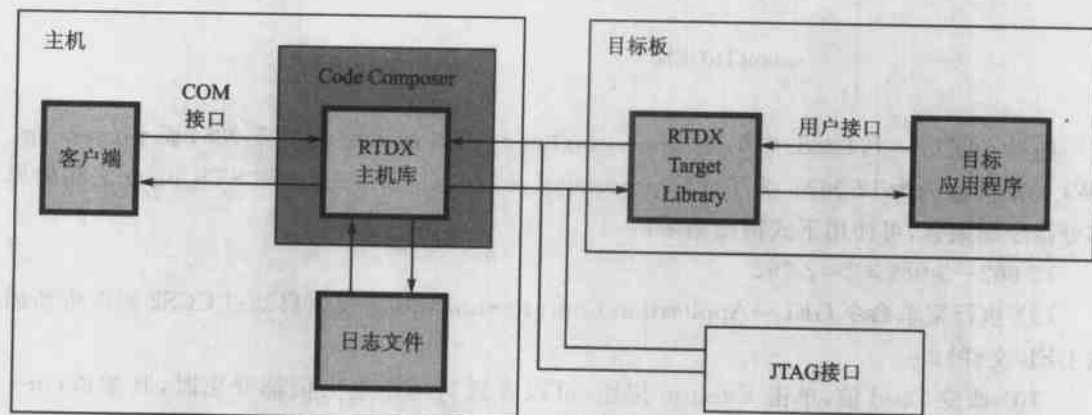


图 8.1 RTDX 的工作原理

PC 主机方运行 CCS 软件。CCS 软件同样带有一个 RTDX 库(RTDX host library),再过一个 OLE 接口将实时数据在 PC 主机上显示和分析。RTDX 数据交换过程如下:

(1) 目标板传输到 PC 主机数据流。为接收目标板上的数据,必须声明一个输出通道,通过用户接口定义的例程将数据写至输出通道。数据将立即被缓冲到一个在 RTDX 目标库中定义的缓冲器中,之后,缓冲器的数据通过一个 JTAG 接口送往 PC 主机。

RTDX PC 主机库从 JTAG 接口接收数据,根据指定的 RTDX PC 主机传输模式,数据可

存放在存储缓冲器中,也可存在 RTDX 日志(log)文件中。

接收的数据可通过 PC 主机应用程序存取。PC 主机应用程序是一个 OLE(对象链接嵌入)自动化客户程序(automation client),如 Visual Basic 应用程序、Visual C++ 应用程序、Lab view 及 Microsoft Excel 等。典型情况下,RTDX OLE 自动化客户程序是一个显示程序,通过一种有意义的方式将数据显示出来。

(2) PC 主机传输到目标板数据流。对一个从 PC 主机接收数据的目标板而言,必须声明一个输入通道,使用在用户接口中定义的程序向其请求数据。请求的数据被缓冲到 RTDX 目标缓冲器中,再通过 JTAG 接口送往目标板。OLE 自动化客房程序可通过 OLE 接口将数据送往目标板。所有送往目标板的数据被写至 RTDX PC 主机库的存储缓冲器中。当 RTDX PC 库接收到一个来自目标 DSP 程序的读请求时,PC 主机缓冲器中的数据就通过 JTAG 接口送往目标板,同时将数据的地址实时写至目标板。操作完成时,PC 主机将通知 RTDX 目标库。

(3) RTDX 目标库用户接口。RTDX 目标库用户接口提供最为安全的方法,在目标应用程序与 RTDX 目标库之间交换数据。用户接口中定义的数据类型和功能如下:

- 使能目标应用程序将数据送到 RTDX PC 主机库。
- 使能目标应用程序向 RTDX PC 主机请求数据。
- 在目标板上提供数据缓冲。数据在送往 PC 主机之前,先复制一份存储在目标板缓冲器中。这样可以保证数据的完全性,减小实时干扰。
- 提供安全的中断,并可以在中断处理程序中调用用户接口定义的例程。
- 提供有效的通信机制,以保证在同一时间内只能有一组数据通过 JTAG 接口在 PC 主机与 DSP 之间进行交换。

(4) RTDX 主机 OLE 接口。OLE 接口提供了 OLE 自动化客户程序与 RTDX 主机之间通信的能力,其功能如下:

- 使 OLE 自动化客户程序能存取在 RTDX 日志文件或缓存在 RTDX 主机库中的数据;
- 使 OLE 自动化客户程序能通过 RTDX 主机库将数据发送到目标板。

8.2 使用 RTDX 插件

8.2.1 RTDX 配置控制窗口

RTDX 主配置控制窗口可以实现下列功能:

- 观察当前 RTDX 配置;
- 使能/禁止 RTDX;
- 利用 RTDX 配置控制属性窗口设置 RTDX 启动配置。

可以通过以下步骤在 CCS 中打开 RTDX 主配置控制窗口:在 CCS 中选择菜单命令 Tools → RTDX → Configuration Control,弹出如图 8.2 所示的当前 RTDX 配置窗口,可以观察当前 RTDX 的配置。

需要使能/禁止 CCS 从 DSP 目标系统处理或传输数据,只须在如图 8.2 所示的 Enable RTDX 栏前面打上或者去掉“√”即可。

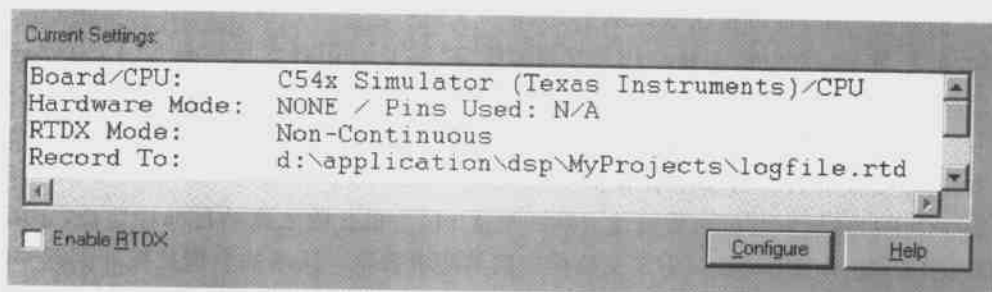


图 8.2 当前 RTDX 配置窗口

要进行 RTDX 的启动设置和端口设置,其步骤如下:

- (1) 禁止 RTDX 功能。
- (2) 在如图 8.2 所示的配置窗口中单击 Configure 按钮,弹出如图 8.3 所示的 RTDX 配置属性设置窗口。

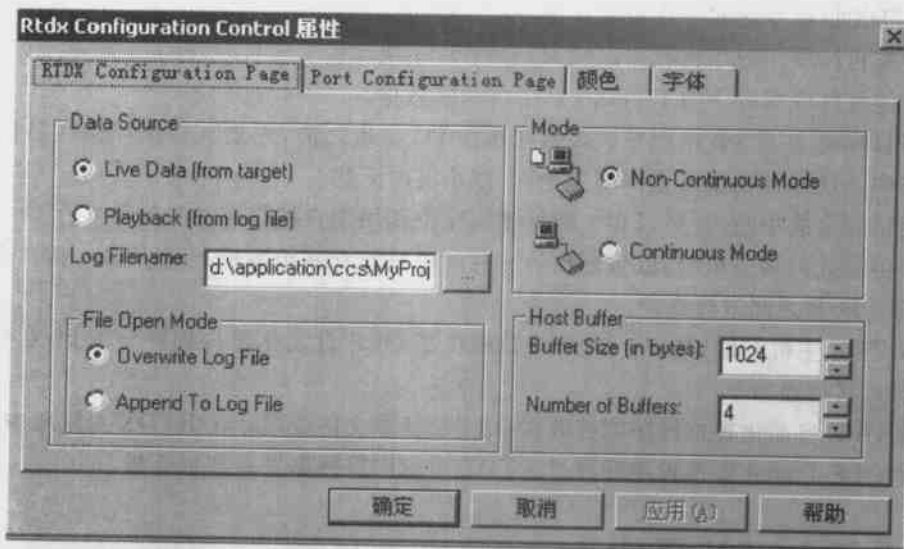


图 8.3 RTDX 配置属性设置窗口

(3) 在 'RTDX Configuration Page' 标签下,允许用户对数据源进行选择,对操作模式、PC 主机缓冲个数及缓冲大小进行设置。

(4) 在 Port Configuration Page 标签下,允许用户对数据交换端口进行设置,以及查看当前端口的属性。

8.2.2 RTDX 通道控制窗口

RTDX 通道控制窗口可以实现下列功能:

- 添加/删除目标系统声明的通道;
- 使能/禁止一个通道。

在 CCS 中,选择菜单命令 Tools→RTDX→Channel Viewer Control,RTDX 通道控制窗口便会出现在 CCS 中。右击 RTDX 通道控制窗口,在弹出的菜单中选择 Delete Channel 或

Add Channel 项,便可添加/删除一个通道;选择 Enable/Disable Channel 命令,便可使能/禁止一个通道,如图 8.4 所示。

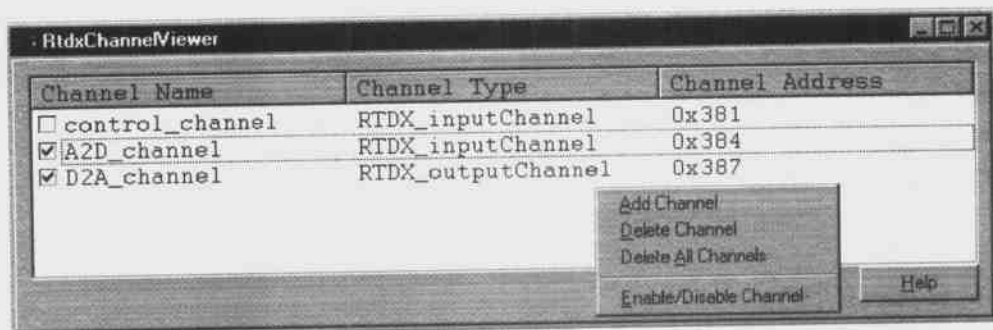


图 8.4 RTDX 通道控制窗口

8.2.3 RTDX 诊断控制窗口

诊断控制窗口可以实现下列功能:

- CCS 内部数据传输仿真测试;
- 测试 DSP 目标系统到 PC 主机的数据传输;
- 测试 PC 主机到 DSP 目标系统的数据传输。

在 CCS 中,选择菜单命令 Tools→RTDX→Diagnostics Control,将弹出如图 8.5 所示的诊断控制窗口。

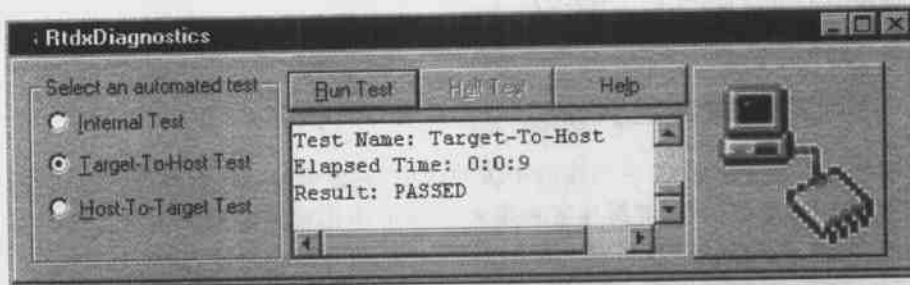


图 8.5 诊断控制窗口

在此窗口中可进行如下三项测试:

- CCS 内部数据传输仿真测试。该项测试可以在不需要硬件仿真器与 DSP 目标系统连接的情况下,仿真 PC 主机从 DSP 目标系统接受数据。
- DSP 目标系统到 PC 主机的数据传输测试。该项测试可以验证 DSP 目标系统传输数据到 PC 主机的能力,以及 PC 主机接受数据的能力,可以使用 D:\application\ccs\examples\dsk5402\rtdx\下面的 t2h.out 文件进行测试。
- PC 主机到 DSP 目标系统的数据传输测试。该项测试可以验证 PC 主机传输数据到 DSP 目标系统的能力,以及 DSP 目标系统接受数据的能力,可以使用 D:\application\ccs\examples\dsk5402\rtdx\下面的 h2t.out 文件进行测试。

可按以下步骤进行测试:

- (1) 在 CCS 中选择菜单命令 Tools → RTDX → Configuration Control。
- (2) 在出现的如图 8.2 所示的当前 RTDX 配置窗口中,使能 RTDX 功能。
- (3) 在 CCS 中选择菜单命令 Tools → RTDX → Diagnostics Control。
- (4) 在出现的如图 8.5 所示的诊断控制窗口中,选择下面的三种测试之一进行测试:
 - Internal Test;
 - Target - To - Host Test;
 - Host - To - Target Test。
- (5) 单击 Run Test 按钮,当测试结束时,测试结果将会出现在状态窗口中。

8.3 配置 RTDX

8.3.1 修改 DSP 目标缓冲区大小

RTDX 目标缓冲区用于临时存放等待传送到主机的数据,不同的 DSP 目标板有不同大小的目标缓冲区。当传输的数据量小时,应设置较小的目标缓冲区;否则应设置较大的目标缓冲区。

可按以下步骤更改 RTDX 目标缓冲区大小:

- (1) 将 D:\application\ccs\c5400\rtdx\lib\rtdx_buf.c 复制到工程所在目录下。
- (2) 将此文件添加入工程中。工程中应已包含 rtdx.lib 运行支持库文件,如没有,则将 rtdx.lib 文件(存放于 D:\application\ccs\5400\rtdx\lib 文件夹中)添加入工程中。
- (3) 编辑 rtdx_buf.c,修改 BUFRSZ。
- (4) 编译链接生成 DSP 程序。

8.3.2 修改主机缓冲区大小

主机缓冲区存在于 RTDX 主机库的主缓冲区中,其容量以 KB 为单位。主缓冲区保存的是来自 DSP 应用程序的信息,此缓冲区必须大于 DSP 应用程序的最大信息长度。对 32 位的目标结构而言,缓冲区应比信息长度大 8 B;对 16 位的目标结构来说,则应大 4 B。

当 CCS 出现以下提示信息时,必须改变主机缓冲区大小。

A data message was received which cannot fit into the buffer allocated on the host. To avoid data loss, reconfigure the buffer and run the application again.

改变主机缓冲区大小可按以下步骤进行:

- (1) 在 CCS 中,执行菜单命令 Tools → RTDX,将弹出如图 8.2 所示的当前 RTDX 配置窗口。
- (2) 在 RTDX 状态栏的下拉列表中,选择 RTDX Disable 项。单击 Configure 按钮,将弹出如图 8.3 所示的 RTDX 配置属性设置窗口。单击 General Settings 栏,增大或减小 Main Buffer 值,单击 OK 按钮确认即可。

8.3.3 RTDX 工作模式


RTDX 主机库提供了两种从目标应用程序中接收数据的模式。

1. 非连续方式

非连续方式是将数据写到主机的日志文件中。这种方式适于应用在传输有限长度数据的场合,此时应指定日志文件的路径及名称。日志文件用于在脱离硬件的情况下回放数据。

需要注意的是:日志文件只能存放在本地硬盘上,而不能通过网络存放在另外的机器上,因为这样可能满足不了实时数据传输的要求。

保存日志文件的步骤如下:

- (1) 设置 RTDX 操作模式为 Non-Continuous 模式;
- (2) 选择 Live Data 作为数据源;
- (3) 在日志文件栏,单击  按钮,将会弹出一个保存文件的对话框;
- (4) 在保存日志文件栏中,输入保存的文件名,注意文件的扩展名为 .rtd,单击“保存”按钮。

2. 连续方式

连续方式的数据不是写往主机日志文件,而是缓存在 RTDX 主机库中。这种方式适合应用于需要连续获取并显示来自 DSP 应用程序数据的场合。在这种情况下,没有必要将数据保存在一个文件中。

在使用连续方式接收数据时,OLE 自动化客户程序必须连续从目标板的每个输出通道中读取数据;否则任何数据流动停止,从而导致数据速率的降低,甚至可能导致通道无法获取数据。此外,OLE 自动化客户程序在启动时应打开目标板的所有输出通道,以避免丢失某个通道的数据。

8.3.4 RTDX 目标中断屏蔽

大多数 RTDX 函数都可在多个线程中同时调用,即使在中断处理程序内部也是如此。然而必须保护这些函数更改过全局数据结构,以保证在同一时刻只有一个函数能更改它。

符号 _RTDX_interrupt_mask 用于在链接时鉴别 RTDX 的客户程序,并保护 RTDX 临界区。

_RTDX_interrupt_mask 指明了须在 RTDX 临界中临时屏蔽的中断,同时也禁止了 RTDX 的其他客户程序,防止调用其他 RTDX 函数。只有那些调用 RTDX 函数的中断需要被屏蔽。

对 TMS320C54X 芯片而言,屏蔽中断是通过修改 IMR(中断屏蔽寄存器)来实现的。在退出临界区时,IMR 将恢复其原来的值。符号 _RTDX_interrupt_mask 必须在链接命令文件中指定。需屏蔽的中断在相应的位置用 0 表示。例如:

```
_RTDX_interrupt_mask=0x0008;
```

此命令将在进入临界区时屏蔽定时器中断。

需要注意的是,RTDX_poll 函数不能在多个线程中同时调用,因此,调用 RTDX_poll 函数时,必须等待先前的调用完成后才能进入。

8.4 实时通信程序的设计方法

RTDX 允许捕获 DSP 应用程序产生的实时数据,且不影响目标系统的实时性能。捕获的

数据被送往 RTDX 主机库, 由其在主机方传输数据。数据可以通过主机方的一个 OLE 自动化客户程序存取。

自动化客户程序(如 Microsoft Excel 或 Labview)必须由设计者提供, 设计者还可编写自己的 Visual Basic 或 Visual C++ 程序作为自动化客户程序。典型情况下, 将 DSP 程序处理数据从目标板送往主机包括以下几个步骤:

- (1) 准备目标 DSP 应用程序, 以捕获实时数据;
- (2) 准备 OLE 自动化客户程序, 以处理数据;
- (3) 启动 CCS, 将应用程序加载到 DSP 目标系统中;
- (4) 在 CCS 中使能 RTDX, 以从目标应用程序中接收数据, 并将其送给 OLE 自动化客户程序;
- (5) 运行目标应用程序, 以捕获实时数据, 并将其送到 RTDX 主机库中;
- (6) 运行 OLE 自动化客户程序处理数据。

8.4.1 编写目标 DSP 应用程序

编写目标 DSP 应用程序的步骤如下:

(1) 定义合适的全局对象 RTDX 通道。通道(channel)用于在目标板与主机之间传送数据。为了将目标应用程序的数据发往主机, 必须声明输出通道。为了将数据从主机接收到目标应用程序中, 则应声明输入通道。在 CCS 中不支持双向通道, 数据在通道中只能单向传送。通道的声明应使用 RTDX_CreateOutputChannel 宏和 RTDX_CreateInputChannel 宏。

(2) 设置中断矢量。RTDX_Poll 函数用于完成主机与目标板间的通信, 因此必须正确设置 RTDX_poll, 否则可能导致 CCS 挂起。对 TMS320C54X 应用程序而言, 中断矢量 ATRAP_V(偏移地址 03CH 必须调用 ATRAP_H, 保留此中断矢量用来实时监控中断)。CCS 提供了一个关于该矢量使用的例程 D:\application\ccs\examples\dsk5402\rt dx\shared 目录下的 intvccs.asm 文件。

(3) 在需要的地方调用 RTDX 函数。使用 RTDX_write 函数, 通过输出通道将数据送往主机; 使用 RTDX_read 或 RTDX_readNB 函数, 通过输入通道向主机请求数据。

(4) 定义符号 RTDX_interrupt_mask。

(5) 使用开发工具生成 DSP 应用程序。

(6) 当编译 DSP 程序时, 必须指定 RTDX 的 include 目录; 当链接时, 必须指定 RTDX 目标库和标准 C 运行库。

在 CCS 环境下可按以下步骤进行:

(1) 在工程视图中, 右击工程名, 在弹出的菜单中选择 Add Files。

(2) 在 Add Files to Project 对话框中, 将位于 D:\application\ccs\c5400\rt dx\lib 目录中的 RTDX 目标库 rtdx.lib 和位于 D:\application\ccs\c5400\cgtools\lib 目录中的标准 C 运行库 rts.lib 加入。如果需要使用扩展寻址功能, 则需加入的两个库文件为 rtdx_ext.lib 和 rts_ext.lib。

(3) 执行菜单命令 Project→Options, 在弹出的 Build Options 窗口中单击 Compiler 栏, 在 Include Search Path 域填入 RTDX include 目录名 D:\application\ccs\c5400\rt dx。如果须使用扩展寻址功能, 还应在参数行手工添加“_548_mf”参数。

(4) 在 Project 工具栏上单击 Build 图标,生成 DSP 程序。

8.4.2 编写 RTDX OLE 自动化客户程序

编写 RTDX OLE 自动化客户程序需要做好以下工作:

- 为每个需要的通道创建一个 RTDX 对象实例;
- 为以上指定的对象打开一个通道;
- 调用任意其他所需的函数。

在 D:\application\ccs\examples\hostapps\rtdx\excel 目录下提供一个名为 rtdx.xls 的 Microsoft Excel 工程文件,此工程中包含了几用 Visual Basic 编写的模块,包括:h_Read(将数据从目标板送往主机)、h_Write(将数据从主机送往目标板)及 h_Event(将事件数据从目标板送往主机)等模块。

编写 RTDX OLE 自动化客户程序的方法如下:

1. 使用 Visual Basic 编写 RTDX OLE 自动化客户程序

使用 Visual Basic 编写 RTDX OLE 客户程序的步骤如下:

(1) 定义一个 Object 类型的变量。

```
Dim rtdx As Object
```

(2) 创建一个 RTDX OLE 对象实例。

```
Set rtdx=CreatObject(RTDX)
```

(3) 释放 RTDX OLE 对象。

```
Set rtdx=Nothing
```

2. 使用 Visual C++ 编写 RTDX OLE 自动化客户程序

使用 Visual C++ 编写 RTDX OLE 客户程序的步骤如下:

(1) 导入 RTDX 服务器类型库。

```
#import<rtdxint.dll>  
using namespace RTDXINTLib;
```

如果存在 rtdxint.tlb,也可写成以下形式:

```
#import<rtdxint.tlb>  
using namespace RTDXINTLib;
```

(2) 声明一个 IrtdxExpPtr 类型的变量。

```
IrtdxExpPtr rtdx;
```

(3) 初始化 OLE。

```
::CoInitialize(NULL);
```

(4) 创建 RTDX OLE 对象实例。

```
HRESULT hr=rtdx.CreateInstance(L"RTDX");
```

(5) 释放 OLE 对象。

```
rtdx.Release();
```

(6) 反初始化 OLE。

```
;;CoUninitialize();
```

8.4.3 在 CCS 中使能 RTDX

在 CCS 中使能 RTDX 的步骤如下：

(1) 在 CCS 中执行菜单命令 Tools→RTDX。

(2) 在 RTDX 状态栏的下拉列表中选择 RTDX Disabled。

(3) 单击 Configure 按钮,打开 RTDX Properties 对话框。

(4) 选择 General Settings 栏。

(5) 选择所需的 RTDX 数据传输方式(非连续方式或连续方式)。当选择非连续方式时,选择合适的 RTDX 日志文件,日志文件的默认扩展名为 .rtd。

(6) 设置 Main buffer 的大小,默认值为 1 024 B。

(7) 单击 OK 按钮,退出配置窗口。

(8) 从 RTDX 状态栏的下拉菜单中选择 RTDX Enable。

8.4.4 运行 OLE 自动化客户程序

OLE 自动化客户程序可完成下列功能：

- 通过 RTDX 主机库读取并处理 RTDX 数据；
- 读取(回放)日志文件中的数据。

当目标 DSP 应用程序重新编译生成后,必须在使用 OLE 自动化客户程序之前重新运行 DSP 程序产生的新的日志文件。

回放日志文件中的数据可按以下步骤进行：

(1) 运行 CCS,在 File 菜单下选择 Load Program 或 Load Symbols 项,选取用来产生日志文件的目标应用程序文件(*.out)；

(2) 在 CCS 的 Tools 菜单下选择 RTDX,出现 RTDX 窗口；

(3) 在 RTDX 状态栏的下拉列表中选择 RTDXPlayback；

(4) 在 Select Log File 对话框中,选取需要的日志文件,单击 Open 按钮打开；

(5) 运行 OLE 自动化客户程序。

当正在运行一个向目标板写数据的 OLE 自动化客户程序时,不要使用日志文件,因为目标应用程序没有运行时,其结果是不可预料的。

8.5 RTDX 实例一

8.5.1 从 DSP 目标系统传输一个整数到 PC 主机

要从 DSP 目标系统传输一个整数到 PC 主机,需要做好以下工作：

- 在 CCS 中打开工程,观察源代码；

- 在源代码中插入特定的具有实时数据传输的 RTDX 语句;
- 进行数据处理,测试应用程序。

要完成传输工作可按以下步骤进行:

(1) 在 CCS 的 Project 菜单中,打开 D:\application\ccs\tutorial\sim54xx\sect_1\less_1 下的工程文件 S111.pjt。

(2) 在工程视图中双击 S111.c,编辑源代码,在其中加入下面的语句:

- 加入 RTDX 头文件 rtdx.h, RTDX 目标接口的原型在此头文件中被声明。

```
#include <rtdx.h>
```

- 定义一个全局的 PC 主机数据输出通道。RTDX 通道是一种必须在 DSP 目标存储器中定义成全局数据对象的数据结构,通道的名字可以任取。

```
RTDX_CreateOutputChannel( ochan );
```

- 初始化 DSP 目标系统。使用宏 TARGET_INITIALIZE()或者自己编写的代码初始化 DSP 目标系统。在初始化目标系统时,需要注意的是,初始化的顺序是与 DSP 处理器相关的。

```
TARGET_INITIALIZE();
```

- 使能输出通道写数据。RTDX 通道在初始化时缺省是禁止的,在数据传输时必须事先使能 RTDX 通道。

```
RTDX_enableOutput( &ochan );
```

- 传送数据到 PC 主机。

```
status = RTDX_write( &ochan, &data, sizeof(data) );
```

- 禁止输出通道传输数据。需要禁止 CCS 从 DSP 目标系统处理或传输数据,则需要调用 RTDX_disableOutput 函数。

```
RTDX_disableOutput( &ochan );
```

(3) 经过编辑的 S111.c 文件如下所示,其中修改部分用黑色重点显示。

```

*****
* S111.c - (Section 1, Lesson 1) SENDING AN INTEGER TO THE HOST
* This is the RTDX Target Code for Section 1, Lesson 1
* This example sends integer value 5 to the host
***** /
Include the rtdx header file, <rtdx.h>,
* which defines the RTDX target API.
*****
#include <rtdx.h>
*****
This example includes <target.h>, which defines TARGET_INITIALIZE.
***** /
#include "target.h" /* defines TARGET_INITIALIZE() */
#include <stdio.h> /* C_1/O */

```

```

/ * Define the value to send to the host *
#define VALUE_TO_SEND 5
/ *****

Declare a global output channel;
* the parameter name in this example must be "ochan."
***** /

RTDX_CreateOutputChannel( ochan );

void main()
{
/ *****

* Declare variables for value to send to host and to check status.
***** /
int data = VALUE_TO_SEND;
int status;

/ *****

Initialize target using the macro
* TARGET_INITIALIZE or insert your own code here.
*****

TARGET_INITIALIZE();

/ *****

Enable the output channel.
*****

RTDX_enableOutput( &ochan );

/ *****

Send data to the host.
/ *****

status = RTDX_write( &ochan, &data, sizeof(data) );

/ *****

* Check return status of RTDX_write().
***** /
if ( status == 0 ) {
puts( "ERROR: RTDX_write failed! \n" );
exit( -1 );
}

while ( RTDX_writing != NULL ) {
# if RTDX_POLLING_IMPLEMENTATION
RTDX_Poll();
# endif
}

/ *****

Disable the output channel.
***** /

RTDX_disableOutput( &ochan );

puts( "Program Complete! \n" );
}

```

- (4) 保存源代码文件,编译链接并生成 S11.1.out 可执行文件。
- (5) 在 File 菜单中载入可执行文件 S11.1.out。
- (6) 选择菜单命令 Tools → RTDX → Configuration Control,在出现的如图 8.2 所示的配置窗口中的 Enable RTDX 栏前面打上“√”。
- (7) 在 Debug 菜单中运行 Run 命令,执行 DSP 应用程序。
- (8) 打开一个 MS-DOS 窗口,把工作目录切换到 D:\application\ccs\tutorial\sim54xx\sect_1\less_1 下;
- (9) 在 DOS 下运行 S11.1.exe,激活客户端程序,这时数字“5”会出现在 MS-DOS 窗口中,可以检验 PC 主机从 DSP 目标系统读取了整数“5”。

8.5.2 从 PC 主机传输一个整数到 DSP 目标系统

要从 PC 主机传输一个整数到 DSP 目标系统,需要做好以下工作:

- 在 CCS 中打开工程,观察源代码;
- 在源代码中插入特定的具有实时数据传输的 RTDX 语句;
- 进行数据处理,测试应用程序。

要完成传输工作可按以下步骤进行:

- (1) 在 CCS 的 Project 菜单中,打开 D:\application\ccs\tutorial\sim54xx\sect_1\less_3 下的工程文件 S11.3.pjt。

- (2) 在工程视图中双击 S11.3.c,编辑源代码,在其中加入下面的语句:

- 定义一个全局的 PC 主机数据输入通道。RTDX 通道是一种必须在 DSP 目标存储器中定义成全局数据对象的数据结构,通道的名字可以任取。

```
RTDX_CreateInputChannel( ichan );
```

- 使能输入通道写数据。RTDX 通道在初始化时缺省是禁止的,在数据传输时必须事先使能 RTDX 通道。

```
RTDX_enableInput( &ichan );
```

- 请求并等待函数 RTDX_read 被调用以接受数据。

```
status = RTDX_read( &ichan, &data, sizeof(data) );
```

- 禁止输入通道传输数据。需要禁止 CCS 从 DSP 目标系统处理或传输数据,则需要调用 RTDX_disableInput 函数。

```
RTDX_disableInput( &ichan );
```

- (3) 经过编辑的 S11.3.c 文件如下所示,其中修改部分用黑色重点显示。

```
*****
* S11.3.c — (Section 1, Lesson 3) RECEIVING AN INTEGER FROM
* THE HOST
* This is the RTDX Target Code for Section 1, Lesson 3
*
* This example receives integer value 5 from the host
```

```

***** /
#include <rtdx.h> /* defines RTDX target API calls */
#include "target.h" /* defines TARGET_INITIALIZE() */
#include <stdio.h> /* C_I/O */
/*****
Create a global input channel:
    * the parameter name in this example must be "ichan."
***** /
RTDX_CreateInputChannel( ichan );
void main()
{
int data;
int status;
TARGET_INITIALIZE();
/*****
Enable the input channel.
***** /
RTDX_enableInput( &ichan );
/*****
Request and wait to receive an integer.
***** /
status = RTDX_read( &ichan, &data, sizeof(data) );
if ( status != sizeof(data) ) {
printf( "ERROR: RTDX_read failed! \n" );
exit( -1 );
} else
printf( "Value %d was received from the host\n", data );
/*****
Disable the input channel.
***** /
RTDX_disableInput( &ichan );
printf( "Program Complete! \n" );
}

```

(4) 保存源代码文件,编译链接并生成 S1L3.out 可执行文件。

(5) 在 File 菜单中载入可执行文件 S1L3.out。

(6) 选择菜单命令 Tools→RTDX→Configuration Control,在出现的如图 8.2 所示的配置窗口中的 Enable RTDX 栏前面打上“√”。

(7) 在 Debug 菜单中运行 Run 命令,执行 DSP 应用程序。

(8) 打开一个 MS-DOS 窗口,把工作目录切换到 D:\application\ccs\tutorial\sim54xx\sect_1\less_3 下。

(9) 在 DOS 下运行 S1L1.exe,激活客户端程序,这时在 CCS 的标准输出窗口将输出“Value 5 was received from the host,”,可以检验 DSP 目标系统从 PC 主机读取了整数“5”。

8.6 RTDX 实例二

8.6.1 从 DSP 目标系统传输一个整数到 PC 主机

要从 DSP 目标系统传输一个整数到 PC 主机,需要做好以下工作:

- 在 Visual Basic 或 Excel 中打开应用程序,观察源代码;
- 在源代码中插入特定的具有实时数据传输的 RTDX 语句;
- 进行数据处理,测试应用程序。

要完成该传输可按以下步骤进行:

(1) 在 Visual Basic 或者 Excel 的工程中,打开 D:\application\ccs\tutorial\sim54xx\sect_2\less_2 下的文件 S2L2.bas。

(2) 编辑源代码,在其中加入下面的语句:

- 定义和加入 RTDX 返回时的常量。

```
Const Success = &H0 'Method call is valid
Const Failure = &H80004005 'Method call failed
Const ENoDataAvailable = &H8003001E 'No data was available,
    'However, more data may be
    'available in the future.
Const EEndOfLogFile = &H80030002 'No data was available,
    'The end of the log file has
    'been reached.
```

- 声明一个对象的变量。

```
Dim rtdx As Object
```

- 创建一个 RTDX COM 对象的实例。

```
Set rtdx = CreateObject("RTDX")
```

- 打开一个读数据的通道。

```
status = rtdx.Open("ochan", "R")
```

- 从通道读入数据。

```
status = rtdx.ReadI2(data)
```

- 关闭读数据的通道。

```
status = rtdx.Close()
```

- 释放 RTDX COM 对象的句柄。

```
Set rtdx = Nothing
```

(3) 经过编辑的 S2L2.bas 文件如下所示,其中修改部分用黑色重点显示。

```

' *****
' * S2L2.has — Section 2, Lesson 2) RECEIVING AN INTEGER FROM THE
' * TARGET
' * This is the RTDX Host Client for Section 2, Lesson 2
' *
' * This example receives integer value 5 from the target
' *****
' *****
' Include the RTDX return code constants.
' *****

Const Success = &H0 ' Method call is valid
Const Failure = &H80004005 ' Method call failed
Const ENoDataAvailable = &H8003001E ' No data was available.
' However, more data may be
' available in the future.
Const EEndOfFile = &H80030002 ' No data was available.
' The end of the log file has
' been reached.
Sub main()
' *****
' Declare a variable of type Object. This will be used to
' access the methods of the interface.
' *****

Dim rtdx As Object
Dim data As Long
Dim status As Long
(On Error GoTo Error_Handler
' *****
' Create an instance of the RTDX COM Object.
' *****

Set rtdx = CreateObject("RTDX")
' *****
' Open a channel for reading;
' the parameter name in this example must be "ochan."
' *****

status = rtdx.Open("ochan", "R")
If status <> Success Then
Debug.Print "Opening of channel ochan failed"
Goto Error_Handler
End If
Do
' *****
' Read a 16—bit integer from the target.
' *****

status = rtdx.ReadI2(data)

```

```

Select Case (status)
Case Success
If status = Success Then
Debug.Print "Value " & _
data & _
" was received from the target"
End If
Case ENoDataAvailable
Debug.Print "No data is currently available"
Case EEndOfLogFile
Debug.Print "End of log file has been detected"
Case Failure
Debug.Print "ReadI2 returned failure"
Exit Do
Case Else
Debug.Print "Unknown return code"
Exit Do
End Select
Loop Until status = EEndOfLogFile
' *****
' Close the channel.
' *****
status = rtdx.Close()
' *****
' Release the reference to the RTDX COM object.
' *****
Set rtdx = Nothing
Exit Sub
Error_Handler:
Debug.Print "Error in COM method call"
Set rtdx = Nothing
End Sub

```

(4) 在 CCS 的 Project 菜单中, 打开 D:\application\ccs\tutorial\sim54xx\sect_2\less_2 下的工程文件 S2L2.pjt。

(5) 编译链接并生成 S2L2.out 可执行文件。

(6) 在 File 菜单中载入可执行文件 S2L2.out。

(7) 选择菜单命令 Tools→RTDX→Configuration Control, 在出现的如图 8.2 所示的配置窗口中的 Enable RTDX 栏前面打上“√”。

(8) 在 Debug 菜单中运行 Run 命令, 执行 DSP 应用程序。

(9) 在 Visual Basic 或者 Excel 中运行 S2L2.bas 程序, 这时在 Visual Basic 的标准输出窗口将输出“Value 5 was received from the target.”, 可以检验 PC 主机从 DSP 目标系统读取了整数“5”。

8.6.2 从 PC 主机传输一个整数到 DSP 目标系统

客户端应用程序通过间接的方式,可以传输数据到 DSP 目标系统。从客户端应用程序传输到 DSP 目标系统的数据,先缓存在 RTDX 主机库中,直到从 DSP 目标系统的一个数据请求出现。当 RTDX 主机库中的数据满足 DSP 目标系统的数据请求时,将数据在不干扰 DSP 目标系统应用程序的情况下写入目标。

缓冲的状态由 `bufferstate` 变量返回,如果 `bufferstate` 返回一个正数,则表明 RTDX 主机库已经缓冲了 DSP 目标系统还没有数据请求;如果 `bufferstate` 返回一个负数,则表明 RTDX 主机库没有满足 DSP 目标系统的数据请求(已经存在一个数据请求)。

要从 PC 主机传输一个整数到 DSP 目标系统,需要做好以下工作:

- 在 Visual Basic 或 Excel 中打开应用程序,观察源代码;
- 在源代码中插入特定的具有实时数据传输的 RTDX 语句;
- 进行数据处理,测试应用程序。

要完成该传输可按以下步骤进行:

(1) 在 Visual Basic 或者 Excel 的工程中打开 D:\application\ccs\tutorial\sim54xx\sect_2\less_4 下的文件 S2L4. bas。

(2) 编辑源代码,在其中加入下面的语句:

```
status = rtdx. WriteI2(data, bufferstate)
```

用以从 PC 主机传输一个 16 位的整数到 DSP 目标系统。

(3) 经过编辑的源代码如下:

```
' *****
' * S2L4. bas -- (Section 2, Lesson 4) SENDING AN INTEGER TO THE TARGET
' * This is the RTDX Host Client for Section 2, Lesson 4
' * This example sends integer value 5 to the target
' *****

Const Success = &H0 ' Method call is valid
Const Failure = &H80004005 ' Method call failed
Const ENoDataAvailable = &H8003001E ' No data was available.
' However, more data may be
' available in the future.
Const EEndOfLogFile = &H80030002 ' No data was available.
' The end of the log file has
' been reached.
Const VALUE_TO_SEND = 5
Sub main()
Dim rtdx As Object
Dim data As Long
Dim bufferstate As Long
Dim status As Long
On Error GoTo Error_Handler
' Create an instance of the RTDX COM object
```



```

Set rtdx = CreateObject("RTDX")
' Open channel ichan for writing
status = rtdx.Open("ichan", "W")
If status <> Success Then
Debug.Print "Opening of channel ichan failed"
GoTo Error_Handler
End If
data = VALUE_TO_SEND
' *****
' Send a 16-bit integer to the target.
' *****
status = rtdx.WriteI2(data, bufferstate)
If status = Success Then
Debug.Print "Value " & data & " was sent to the target"
Else
Debug.Print "WriteI2 failed"
End If
' Close the channel
status = rtdx.Close()
' Release the reference to the RTDX COM object
Set rtdx = Nothing
Exit Sub
Error_Handler:
Debug.Print "Error in COM method call"
Set rtdx = Nothing
End Sub

```

(4) 在 CCS 的 Project 菜单中,打开 D:\application\ccs\tutorial\sim54xx\sect_2\less_4 下的工程文件 S2L4. pj1。

(5) 编译链接并生成 S2L4. out 可执行文件。

(6) 在 File 菜单中载入可执行文件 S2L4. out。

(7) 选择菜单命令 Tools→RTDX→Configuration Control,在出现的如图 8.2 所示的配置窗口中的 Enable RTDX 栏前面打上“√”。

(8) 在 Debug 菜单中运行 Run 命令,执行 DSP 应用程序。

(9) 在 Visual Basic 或者 Excel 中运行 S2L4. bas 程序,这时在 CCS 的标准输出窗口将输出“Value 5 was received from the host,”,可以检验 DSP 目标系统从 PC 主机读取了整数“5”。

第 9 章 CCS2 高级使用——使用 GEL

9.1 GEL 语言简介

GEL(General Extension Language)语言是一种类似于 C 语言的功能强大的语言。在实际使用中,按照 GEL 语法创建 GEL 函数并将其加载到 CCS2 中,以扩展 CCS2 的用途。使用 GEL 语言可以访问实际的或者仿真的目标 DSP 存储器,还可以在 GEL 菜单增加新的功能选项,特别是在用于定制用户工作区和自动测试时非常有效。GEL 函数可在任何能键入表达式的地方调用,甚至还可以将 GEL 函数添加到 Watch 窗口。这样,在每个断点处都能自动执行该函数。

9.2 GEL 语法

GEL 语言是 C 语言的一个子集,但它不能声明主机变量,所有的变量必须在 DSP 程序中定义,存在于仿真的或实际的目标 DSP 程序中。惟一不在目标 DSP 程序中定义的标识符是 GEL 函数及其参数。当对一个变量估值时,CCS2 调试器从目标板获取必要的信息,在此之前应将带有符号信息的 COFF 文件加载至目标 DSP 中。

GEL 支持以下几种类型的语句:

- return 语句;
- if - else 语句;
- while 语句;
- for 语句;
- break 语句;
- 函数的局部变量;
- GEL 注释;
- 预处理语句。

1. return 语句

GEL 语言支持标准 C 语言形式的 return 语句。其形式如下:

Return 表达式;

GEL 函数无须返回一个值,当 return 后面没有跟表达式时,它返回的不是一个有效值,而是将控制权返回给调用函数;当函数结尾没有 return 语句时也是如此。调用函数可以忽略被调用函数的返回值。

与 C 语言不一样的是,GEL 函数不指定返回类型,返回类型可根据运行情况自动确定。

2. if - else 语句

GEL 语言支持标准 C 语言的 if - else 语句。其形式如下:

```

If(expression)
    Statement1;
else
    Sstatement2;

```

当表达式为真时,执行语句 1;否则执行语句 2。每个语句可以是单条语句,也可以是用花括号括起来的多条语句。例如:

```

if(a = 25)
    b = 30;          /* if - else 语句可以只有 if 而没有 else */
if(b = 20)
{
    a = 30;          /* 执行两条语句,用花括号括起来 */
    c = 30;
}
else
{
    d = 20;
}

```

3. while 语句

GEL 语言的 while 语句与标准 C 语言的 while 语句相似,但 GEL 语言不支持嵌入的 continue 和 break 语句。其形式如下:

```

while(expression)
statement;

```

当表达式为真时,执行下面的语句,并重新判断表达式是否为真。由于 GEL 语言不支持 continue 语句,因此当且仅当表达式为假时才能跳出循环。当 while 后须跟几条语句时,需要用花括号括起来,例如:

```

While(a! = Count)
{
    dataspace[a] = 0;
    a ++;
}

```

4. for 语句

GEL 语言支持与标准 C 语言类似的 for 语句。其形式如下:

```

for ( expression1; expression2; expression3 )
    statement1;

```

该语句和各个参数的含义与标准 C 语言完全一致。当 for 后须循环执行几条语句时,要用花括号括起来,例如:

```

for (i = 0; i < 9; i++)
{
    data1[i] = 0;
}

```

```
data2[j] = 0;  
}
```

5. break 语句

与标准 C 语言一样, GEL 语言支持在 for 和 while 等循环语句内部使用 break 语句。break 语句的形式如下:

```
break;
```

break 语句中止循环语句的执行, 并从 for 语句下面的语句继续执行。break 语句使 for 语句中止后, 将从 d=20 开始继续执行, 如下所示:

```
for (j=0; j<i; j++)  
{  
    if (j == k)  
        break;  
}  
d = 20;
```

6. 函数的局部变量

在函数的内部, 可以定义和使用局部变量。当使用局部变量时, CCS2 调试器从 COFF 文件中获取变量的相关信息, 因此在使用局部变量前, COFF 文件一定要事先加载。

在下面的例子中, 局部变量 i 在 GEL 函数内部定义, 在使用变量 i 前, 变量 num 一定要事先加载, 例如:

```
MyFunc(num)  
{  
    int i;  
    for (i=0; i<=num; i++)  
        data[i] = 0;  
}
```

7. GEL 注释

GEL 语言支持标准 C 语言的注释, “/*”为开始标记, “*/”为结束标记, 中间的一行或多行均为注释行, 以“//”开头的注释行, 注释持续到行末, 例如:

```
/* 该注释跨越两行  
*/  
// 该注释持续到本行末
```

8. 预处理语句

与标准 C 语言一样, GEL 语言支持标准 #define 预处理语句。#define 是 GEL 支持的一个预处理关键字。使用 #define 可以定义一个宏。下面的控制语句, 让 CCS2 预处理器用后面的符号序列(token - sequence)替代前面的符号(identifier), token - sequence 符号前面和后面的空格将被忽略。

```
#define identifier token - sequence
```

9.3 GEL 函数

9.3.1 GEL 函数定义

GEL 函数定义的形式如下：

```
函数名([参数 1[,参数 2...[,参数 6_]])  
{  
    语句  
}
```

GEL 函数在扩展名为 .gel 的文本文件中定义。一个 GEL 文件可以包含很多 GEL 函数定义。GEL 函数不标明任何返回值类型,也不需要头文件来定义函数中使用的参数类型。这些类型信息可自动从数据值中获得。

与标准 C 语言一样,GEL 函数定义不能嵌入在其他的 GEL 函数定义中。定义一个求平方的函数形式如下：

```
square(a)  
{  
    return a * a;  
}
```

如果在 CCS2 观察窗口中观察计算一个数的平方,形式如下：

```
square(1.2) = 1.44  
square(5) = 25
```

使用 GEL 函数时,可以在每个参数的后面跟一个字符串来说明参数的用法。下面的 Init() 函数用于在对话框中创建对话框函数：

```
dialog Init(filename "File to be Loaded",CPUname "CPU Name",  
            initValue "Initialization Value")  
{  
    GEL_Load(filename,CPUname);  
    a=initValue;  
}
```

对话框将这个函数加入到菜单栏。这个函数共有 3 个参数,其后的 3 个字符串分别用来说明这 3 个参数的用法。CPUname 是一个可选参数,用于设置多处理器。注意：“a=initValue”语句中,a 并没有在参数表中定义,因此它必须在 DSP 程序中定义;否则在调用此函数时将发生错误。以下是一种合法的调用方式：

```
Init("c:\\mydir\\myfile.out", "cpu_a", 0)
```

9.3.2 GEL 函数参数

通过在 GEL 函数定义中定义参数,可将值传递给 GEL 函数。与 C 函数参数不一样的

是, GEL 函数不需要定义参数类型, 只需定义参数名即可, 参数类型则自动由传递的值来判断。GEL 参数可以是以下几种类型之一:

- 一个实际或仿真的 DSP 程序符号值(如果传递的是 DSP 程序符号);
- 一个数字常量(如果传递的是表达式或常量值);
- 一个字符常量(如果传递的是字符串)。

下面是一个 GEL 函数定义的例子:

```
Initialize(a,filename,b)
{
    targVar = b;
    a = 0;           /* 必须将一个 DSP 符号传递给参数 a */
    GEL_Load(filename); /* 必须将一个字符串传递给参数 filename */
    return b * b;
}
```

对此 GEL 函数的正确调用方法是:

```
Initialize(targetSymbol, "c:\\myfile.out", 23 * 5 + 1.22);
```

当函数执行时, DSP 符号 targetSymbol 传递给参数 a, 字符串常量 "c: \\myfile.out" 传递给参数 filename, 而常量值 116.22 则传递给参数 b。

如果传递给参数 a 的不是一个 DSP 符号, 运行中当执行到第 2 条语句 $a=0$ 时会出现错误。例如, 如果将一个常量值 20 传递给 a, 则第 2 条语句将变为 $20=0$, 这显然是一条非法的赋值语句。

即使将一个合法的 DSP 符号传递给参数 a, 也必须确保在执行 GEL 函数时已将符号信息加载到 CCS2 调试器中。如果符号没有定义, 在执行到第二个声明 $a=0$ 时会出现运行错误。同样, 包含声明 targVar 的代码必须在执行这个函数前加载; 否则在执行到第一个声明 $targVar = b$ 时会出现执行错误。如果 targetSymbol 已定义, 则上面的函数调用将会赋 0 值给 targetSymbol。

GEL 参数可以是数值或字符串, 如 1.3、1415、0x100、“c: \\filename”等。对一个数值参数, 可以将一个合法的 C 表达式赋给它。表达式的值会先计算好, 再传递给 GEL 参数。如果最终结果包含小数点或指数符号(如 1.2 或 1.34e4), 则参数被认为是实数类型; 否则被认为是整数类型。

可以用以下两种形式调用上述的 initialize GEL 函数:

```
Initialize(targetSymbol, "c:\\mydir\\myfile.out", 10);
Initialize(targetSymbol, "c:\\filename.out", 1.2);
```

第 1 条调用语句中, 参数 b 被认为是整数值。第 2 条调用语句则将输入视为实数类型。如果目标变量 targVar 是整数类型, 则在赋给 targVar 时参数 b 被截尾。

当使用参数符号定义一个 GEL 函数时, 传递的变元是可选的。这是因为对数值参数而言, 其初始值为 0; 否则被初始化为一个空的字符串。如果不传递参数, 则函数会将默认值赋给参数。这就意味着还可以使用以下两种方式调用上述函数:

```
Initialize();
```

```
Initialize(targetSymbol, "c:\\myfile.out");
```

第 1 条调用语句将 0 赋给 targVar。然而,当试图执行 GEL_Load(filename)语句时会出现错误。由于没有变元传递给 filename,因此语句等效为 GEL_Load(""),从而导致一个非法文件名的错误。在执行 a=0 语句时也同样会出现错误。第 2 条调用语句则不会出现错误。它向 GEL 函数传递了两个参数,第 3 个参数被自动赋为 0。

GEL 定义的语法结构非常松,可以简单地使用默认值,这使得调用 GEL 函数具有很大的灵活性。GEL 函数可以传递很多参数,例如,GEL_TextOut()函数可带 6 个参数,调用时可使用所有的参数,也可仅使用一个参数,如下所示:

```
GEL_TextOut("Hello World!");
```

9.3.3 调用 GEL 函数

GEL 函数可以在任何能键入 C 表达式的地方调用,既可以在任何可键入 C 表达式的对话框中调用,也可在其他 GEL 函数中调用。但递归的 GEL 函数是不被支持的。当一个 GEL 函数在执行时,不能再运行它的另一个实例。传递给 GEL 函数的参数是可选的,如果传入的参数被忽略,则假设该参数取其默认值。

9.3.4 加载/卸载 GEL 函数

当包含 GEL 函数的文件已定义好时,必须将此文件加载到 CCS2 中,以便能调用这些 GEL 函数。这样,GEL 函数驻留在 CCS2 存储器中,可以在任何时候执行。GEL 函数将一直驻留在内存中,直到将其卸载为止。当一个加载的 GEL 文件被更改后,必须首先将其卸载,再重新加载,以使更改生效。

GEL 加载器会在加载 GEL 文件时检查其语法错误,但它不检查变量是否已定义,因此,可以在加载包含符号信息的 COFF 文件之前加载 GEL 函数。同样,在加载 GEL 函数时,还允许引用尚未定义或尚未加载的 GEL 函数。不过,在 GEL 函数执行之前必须定义好 GEL 函数中引用的符号。如果 CCS2 在加载 GEL 文件时发现语法错误,CCS2 将停止加载并显示相应的错误信息。此时应先修正这些错误,再重新加载文件。

1. 加载 GEL 文件

加载 GEL 文件共有两种方法:一种方法是执行菜单命令 File→Load Gel,选中所需的 GEL 文件并打开;另一种方法是在工程视图窗口中的 GEL Files 目录上右击,从弹出的菜单中选择 Load GEL,再选中所需的 GEL 文件并打开。

2. 卸载 GEL 文件

双击工程视图窗口中的 GEL Files 目录,可以观察到所有的 GEL 文件,在需要卸载的 GEL 文件上右击,从弹出的菜单中选择 Remove from the menu 命令即可。

9.4 用关键词将 GEL 函数添加到菜单中

可以将常用的 GEL 函数添加到 CCS2 的 GEL 菜单下,此时须使用 menuitem 关键词在 GEL 菜单下创建一个新的下拉菜单列表(一级菜单),再使用关键词 hotmenu、dialog 和 slider

在该菜单项中添加新的菜单项(二级菜单)。当在 GEL 菜单下选择用户定义的菜单项时,将出现一个对话框或滑动条。

1. hotmenu

使用 hotmenu 将 GEL 函数添加到 GEL 菜单中,当选择此函数时会立即执行。其语法如下:

```
hotmenu    func Name()
{
    statements
}
```

可用以下示例说明:

```
menuItem "My Functions";          /* 在 CCS2 的 GEL 菜单下添加一条 My Functions 菜单项 */
hotmenu InitTarget()              /* 在 My Functions 下添加一条 InitTarge 二级菜单项 */
{
    * waitState=0x11;
}
hotmenu LoadMyProg()             /* 在 My Functions 下添加一条 LoadMyProg 二级菜单项 */
{
    GEL_Load("c:\mydir\myfile.out");
}
```

此例将在 GEL 菜单下添加到如图 9.1 所示的菜单项。如果在菜单下选择 InteTarge 命令,则该命令会立即被执行。hotmenu 用在无须传递任何参数的 GEL 函数中,如果在调用 GEL 函数时须传递参数,应使用 dialog。



图 9.1 用 hotmenu 将 GEL 函数添加到菜单中

2. dialog

使用 dialog 将 GEL 函数添加到 GEL 菜单中,创建一个需要输入参数的对话框窗口。当从 GEL 菜单中选择此函数时,将弹出一个对话框窗口,提示输入参数。dialog GEL 函数的语法为

```
dialog funcName(参数 1"参数 1 说明",参数 2 "参数 2 说明",...)
{
    语句...
}
```

其中:参数是在函数内使用的变量名,参数说明则是在对话框中相应输入域的提示信息。通过对话框窗口最多能向 GEL 函数传递 6 个参数。下面的例子说明如何使用 dialog 向 GEL 菜单添加两个菜单选项。

```
MenuItem "My Functions";
```



```

Dialog InitTarget(startAddress "Starting Address",EndAddress "End Address")
{
    statements
}
dialog LoadMyProg()
{
    statement
}

```

此例将同样在 GEL 菜单下添加如图 9.2 所示的菜单项。当在菜单中执行 InitTarget 命令时,将出现 Function:InitTarget 对话框,提示输入起始地址和结束地址,如图 9.3 所示。当在输入域中键入数字并单击 Execute 按钮时,将使用这些键入的参数调用 GEL 函数。



图 9.2 用 dialog 将 GEL 函数添加到菜单中

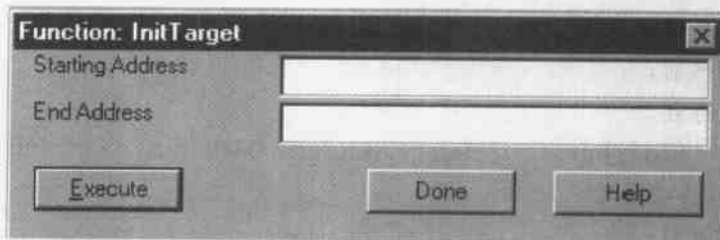


图 9.3 Function:InitTarget 对话框

3. slider

除上述关键词外,还可以使用 slider 将 GEL 函数添加到 GEL 菜单中。当从 GEL 菜单中选择此函数时,将出现一个滑动条,控制传递给 GEL 函数的参数值。每次移动滑动条上的滑杆时,将调用 GEL 函数,其参数值由滑杆的位置决定。Slider GEL 函数菜单上有 5 个参数,但使用时只能向 slider GEL 函数传递 1 个参数。Slider GEL 函数的格式如下:

```

Slider param_definition(min Val,max Val,increment pageIncrement,paramName)
{
    statements
}

```

各参数的说明如下:

- Param_definition 在 slider 对象旁显示的参数提示。
- min Val 当滑动条在最低位置时,传递给函数的整数常量值。
- max Val 当滑动条在最高位置时,传递给函数的整数常量值。
- Increment 每次移动一个滑动位置时,增加的整数常量值。
- PageIncrement 每次移动一页(page)滑动时,增加的整数常量值。
- ParamName 在函数内部使用的参数定义。

下面使用 slider 添加一个音量控制滑动条。

```
menuitem "My Functions";
slider VolumeControl(0,10,1,1,volume)
{
    /* initialize the target variable with the parameter passed by the slider object. */
    targVarVolume=volume;
}
```

此例将向 GEL 菜单添加一个 My Functions 子菜单,执行 VolumeControl 命令时将弹出一个滑动条,当拖动滑杆时,修改的是 volume 的值,同时 DSP 程序中的 targVarVolume 变量值会跟着变化。

4. 缺省的 GEL 函数

C5000 平台的 CCS2 提供了缺省的 GEL 函数,下面所有的 C54X 和 C55X 系列 DSP 处理器 GEL 函数,都是在缺省的初始化文件 c5000.gel 中定义的。

(1) C54X 系列 DSP 的 GEL 函数。

① C54X_CPU_Reset,其功能有:

- 复位目标系统;
- 禁止内存映射;
- 初始化寄存器。

② C541_Init、C542_Init、C543_Init、C545_Init、C546_Init、C548_Init 和 C549_Init 复位 C54X 的 CPU,其功能有:

- 使能内存映射;
- 为特定的 DSP 处理器配置内存。

③ C5402_Init、C5409_Init、C5410_Init、C5416_Init、C5420_Init、C5421_Init 和 C5402_DSK_Init,其功能有:

- 复位 C54X 的 CPU;
- 复位外围设备;
- 使能内存映射;
- 为特定的 DSP 处理器配置内存。

(2) C55X 系列 DSP 的 GEL 函数:C55XReset,其功能是复位目标 DSP 系统。

9.5 输出窗口函数

CCS2 提供了几个嵌入 GEL 函数,可以将结果显示到输出窗口中。这些函数能实现以下功能:

- 创建无限个数的输出窗口;
- 创建固定大小或带滚动条的窗口;
- 将输出输送到任何窗口;
- 打印多种颜色的输出;
- 更改高亮文本;

- 将带有格式的字符串打印到仿真目标板或实际目标板上。

相关的函数有：

- GEL_Open Windows 打开一个输出窗口；
- GEL_Close Windows 关闭一个输出窗口；
- GEL_TargetTextOut 格式化输出字符串；
- GEL_TextOut 向输出窗口输出字符串。

9.6 在 CCS2 启动时自动执行 GEL 函数

CCS2 允许用户按其需要使用 GEL 函数配置开发环境。要使用 GEL 函数,通常的方法是先执行菜单命令 File→Load Gel,再执行 GEL 函数。如果每次设置环境都采用这种方法就很繁琐。要设置启动环境,方便的方法是在 CCS2 启动后自动执行 GEL 函数,可以在 CCS2 启动时将 GEL 文件名传给 CCS2,让 CCS2 扫描并加载指定的 GEL 文件。

设置环境仅自动加载 GEL 文件是不够的,还要求能自动执行 GEL 函数,方法是将某个 GEL 函数命名为 StartUp()。这样,当 GEL 加载到 CCS2 时,它将自动搜索一个名为 StartUp()的 GEL 函数并自动执行。

在桌面的 CCS2 快捷方式上右击,从弹出菜单中选择 Properties(属性),将出现如图 9.4 所示的窗口。

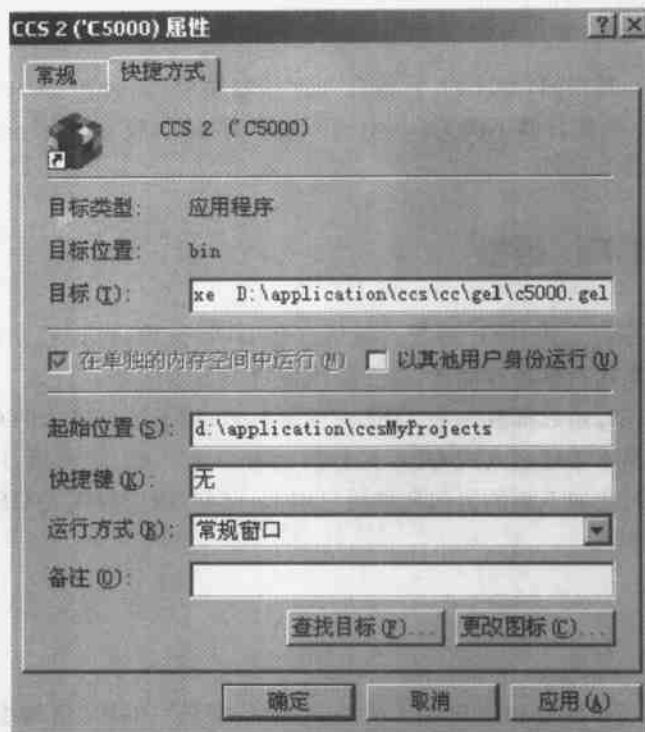


图 9.4 修改 CCS2 快捷方式自动加载 GEL 文件

从图 9.4 中可以看出,此快捷方式将运行 CCS2 可执行文件 D:\application\ccs\cc\bin\cc_app.exe,同时将自动加载 D:\application\ccs\cc\gel\c5000.gel 文件。实际上每次运行 CCS2 时,即使尚未打开工程文件,工程视图中的 GEL files 目录下也会有一个名为 c5000.gel 的文件。在该文件中有一个名为 StartUp() 的 GEL 函数,该函数只有一条语句“GEL_Reset()”,它在启动 CCS2 时自动执行。

将图 9.4 目标栏中运行 CCS2 可执行程序所带的参数更改为自己的 GEL 文件(如 D:\application\ccs\myprojects\myfile.gel),同时在 myfile.gel 文件中定义一个名为 StartUp() 的 GEL 函数,就可以在启动 CCS2 时自动加载此 GEL 文件,并自动执行 StartUp() GEL 函数中的相应语句。下面是一个在启动时加载的典型 GEL 文件:

```
StartUp()                                /* 此函数中所有语句将在启动 CCS2 时自动执行 */
{
    GEL_MapOn();                          /* 打开存储器映射 */
    GEL_MapAdd(0,0,0Xf000,1,1);
    GEL_MapAdd(0,1,0Xf000,1,1);
}
menuItem "Newmenu"
hotmenu LoadMyFile()                    /* 加载自己的 COFF 文件,并跳至 main()行 */
{
    GEL_Load "D:\\application\\dsp\\myprojects\\myfile.out";
    GEL_Go(main);
}
```

在上面的代码中,每次启动 CCS2 时将打开存储器映射,同时在 CCS2 的 GEL 下拉菜单下加入 Newmenu 项,当执行其下的 LoadMyFile 命令时,将加载 myfile.out 并打开主程序,跳至程序的 main()行。

9.7 嵌入式 GEL 函数

CCS2 提供了很多嵌入式 GEL 函数。使用这些 GEL 函数,可以控制仿真或实际目标板的状态,访问仿真或实际目标板存储器,并在输出窗口中显示结果。

所有的嵌入式 GEL 函数都有一个“GEL_”前缀标志,用于区分嵌入的 GEL 函数与用户定义的 GEL 函数。如果不希望嵌入式 GEL 函数前面带一个“GEL_”前缀,可以为嵌入式 GEL 函数定义自己的名字,比如下面的语句允许用户用 Load() 调用嵌入式 GEL_Load() 函数:

```
Load(a)
{
    GEL_Load(a);
}
```

所有的嵌入式 GEL 函数和用户定义的 GEL 函数都是由 GEL 语句组成的,可以直接在 GEL 工具栏中调用。GEL 工具栏由一个表达式域(左侧的文本输入框)和一个 Execute 按钮构成。要调用一条 GEL 语句或用户定义的函数,只须在表达式域键入须调用的函数,单击 Execute 按钮即可。表达式域还保存了最新调用过的 GEL 函数的历史记录,可以直接选择其

其中之一执行。

CCS2 的嵌入 GEL 函数有如下几类：

1. Debug(调试)类

(1) GEL_Animate();开始动画执行 DSP 程序。

(2) GEL_Go(address);执行 DSP 程序到一个由参数指定的地址。地址参数(可选项)指的是程序存储器地址。如果不指定参数,由 GEL_Go 函数与 GEL_Run 函数完成相同的功能。

如果程序始终执行不到指定的地址,则此 GEL 函数永远不能结束运行。此时应执行菜单命令 View→Expression,查看所有被估值的表达式,然后选择 GEL_Go(),并单击 Abort 按钮,中止 GEL 函数的运行。

合法的调用方式如下:

```
GEL_Go();  
GEL_Go(main);
```

(3) GEL_Run("Condition");开始执行 DSP 程序。带引号的参数(可选项)是一个表达式,如果运行 GEL 函数时带有此条件参数,则当条件为真时程序才运行。程序在运行到每个断点处都会判断条件是否满足,当不满足时,CCS2 调试器将在该地址停止。

合法的调用方式如下:

```
GEL_Run();  
GEL_Run("A! = B");
```

(4) GEL_Halt();停止正在运行的 DSP 程序。

(5) GEL_RunF();Debug 菜单下的 Run free 命令。此函数在运行 DSP 程序之前应先禁止所有的断点,同时还断开与目标系统的连接。此命令在对目标系统进行硬件复位或需要断开 JTAG 或 MPSD 电缆连接时有用。CCS2 调试器将在需要存取目标系统(如读存储器)或用户中止程序运行时,重新使能断点,并建立与目标系统的连接。

(6) GEL_Restart();复位 DSP 程序到其入口地址。此函数应在 DSP 程序(包括符号信息)已被加载至目标板后使用。

2. Breakpoint(断点)类

(1) GEL_BreakPtAdd(address, "Condition");在指定地址设置一个软件断点。参数 Condition 是可选的。如果指定了一个参数,则该断点变成一个条件断点,仅当条件为真时,该断点才有效。参数 address 是必需的,用于指定断点地址。此地址可以是绝对地址、C 表达式、C 函数名或者汇编代码标号名。

合法的调用方式如下:

```
GEL_BreakPtAdd(0x2000);  
GEL_BreakPtAdd(TargetLabel+100);  
GEL_BreakPtAdd(0x2000, "a<b");
```

(2) GEL_BreakPtDel(address);清除指定地址的软件断点。如果在指定地址没有软件断点,则该函数不做任何实质性工作。地址参数是必需的,可以是绝对地址、C 表达式、C 函数名或者汇编代码标号名。

合法的调用方式如下:

```
GEL_BreakPtDel(0x2000);  
GEL_BreakPtDel(TargetLabel + 100);
```

3. 窗口类

(1) GEL_OpenWindow("windowName", windowType, maxLines): 创建一个输出窗口。该窗口名由参数 windowName 指定, 其他的 GEL 函数可以根据窗口名来访问该输出窗口。输出窗口的个数可以不受限制。

该函数的 3 个参数均是可选的。如果不指定窗口名, 则将使用 Macro Output 作为窗口名。WindowType=1 时, 创建的是不带滚动条的窗口; 否则(等于 0 或不指定 windowType 时)创建的是带滚动条的窗口。当创建不带滚动条的窗口时, 参数 maxLines 用于指定该窗口能存放的最大行数; 如创建的是带滚动条的窗口, 则此参数将被忽略。

合法的调用方式如下:

```
GEL_OpenWindow();  
GEL_OpenWindow("Macro Output", 120);
```

(2) GEL_CloseWindow("windowName"): 关闭一个窗口, 该窗口名由参数 windowName (必需) 指定, 其必须与 GEL_OpenWindow 函数所指定的窗口名完全一致。

合法的调用方式如下:

```
GEL_CloseWindow("My Window");  
GEL_CloseWindow("Macro OutPut");
```

(3) GEL_TextOut("text", "windowName", textColor, lineNumber, appendToEnd, param1, ..., param4): 将一个固定的字符串打印到指定的输出窗口。

参数说明如下:

- text 必需的参数, 用于指定输出的格式文本(包括格式说明符, 如 %d 和 %f 等)。格式说明字符的个数必须与参数(param1, ..., param4)的个数相等。
- windowName 可选参数, 指定输出的窗口。如果窗口尚未打开, 将使用默认参数的 GEL_OpenWindows() GEL 函数创建窗口。
- textColor 可选参数, 指定打印文本的颜色。0——黑色, 1——蓝色, 2——红色。
- lineNumber 指定打印从哪行开始。如果窗口是一个不带滚动条的窗口时的标志符号, 当标志为 1 时, 文本被追加到现有的行上; 否则将原有行删除, 用新的一行代替。
- appendToEnd 可选参数, 当打印到不带滚动条窗口的标志符号。当标志为 1 时, 文本被追加到现有的行上; 否则将原有行删除, 用新的一行代替。

param1, ..., param4: 可选参数, 与 text 参数中的格式说明符匹配使用。GEL 语言中的格式说明符是 C 语言格式说明符的一个子集, 只支持以下几种:

```
%d    带符号的十进制整数;  
%u    无符号的十进制整数;  
%x    十六进制格式;  
%f    带符号的双精度值;
```

%d 带符号的双精度值,用科学计数法表示;

%s 字符串。

合法的调用方式如下:

```
GEL_TextOut("All Tests Passed.\n")
GEL_TextOut("Failed Memory Test.\n", "Diagnostic Results", 2);
GEL_TextOut("Tests Executed: %d, Test Passed: %d", targetExe, targetPass);
```

(4) GEL_TargetTextOut(startAddress, page, maxLength, format, "windowName", textColor, lineNumber, appendToEnd, changeHighlighting): 将一个格式字符串打印到输出窗口。这个字符串必须是目标板已存在的,且以二进制 0 作为结束标志。

参数说明如下:

- startAddress 包括格式字符的块起始地址。这个参数是必需的。
- page 可选参数,指示存储器类型。0——程序存储器,1——数据存储器,2——I/O。对 Simulator 而言,不支持 I/O(Page 为 2 时)。该参数的默认值为 0。
- maxLength 可选参数,当块大于 400 B 时用来指示块的最大长度。目标板的格式文本应以二进制 0 结束,如果没有碰到这个标志,则只打印前 400(或 maxLength)B。
- format 可选参数,指示打印的文本是打包的还是未经打包压缩的。0——未经打包的 ASCII 字符;1——打包的 ASCII 字符,使用 big endian 格式;2——打包的 ASCII 字符,使用 little endian 格式。
- windowName 可选参数,指定输出的窗口。如果窗口尚未打开,将使用默认参数的 GEL_Open Windows()GEL 函数创建窗口。
- textColor 可选参数,指定打印文本的颜色。0——黑色,1——蓝色,2——红色。
- lineNumber 指定打印从哪行开始。如果是一个不带滚动条的窗口,则该参数是可选的。
- appendToEnd 可选参数,当打印到不带滚动条窗口的标志符号。当标志为 1 时,文本被追加到现有的行上;否则将原有行删除,用新的一行代替。
- changeHighlight 可选参数,当打印到不带滚动条窗口时的标志符号。当此标志使能时,新的文本将与原来的文本比较,如果两者有不同,则新的文本被高亮显示。

合法的调用方式如下:

```
GEL_TargetTextOut(0x800);
GEL_TargetTextOut(0x1000, 0, 400, 1, "My Window", 1);
```

(5) GEL_Exit(): 关闭当前活动的控制窗口。对只有一个处理器的系统而言,它将直接关闭 CCS2。

4. 程序加载类

(1) GEL_Load("filename", "cpuName"): 将目标文件(COFF 文件)及相应的符号表加载至存储器中。参数 fileName 指定要加载的 COFF 文件,如果这个文件不在当前目录中,则需给出其完整路径。注意:在写路径时,反斜线符号“\”应写两次,即“\\”。参数 cpuName 给出 CPU 名,在设置多处理器时应与配置的 CPU 名一致。如果只有一个处理器,则该参数可以不填。

合法的调用方式如下:

```
GEL_Load("c:\\workdir\\test.out", "cpu_b");
```

(2) GEL_SymbolLoad("filename", "cpuName"): 加载指定目标文件的符号信息。这个函数在 CCS2 调试器不能或不需要将目标代码加载至存储器(如程序代码存放在 ROM 中)时有用,不更改程序入口点。参数 fileName 指定的是包含符号信息的 COFF 目标文件,cpuName 是可选参数,指定符号信息将要加载到的 CPU 名称,在多处理器环境中此参数有用。

合法的调用方法如下:

```
GEL_SymbolLoad("d:\\mydir\\myfile.out", "cpu_b");
```

5. 存储器类

(1) GEL_MapAdd(address, page, length, readable, writable): 设置存储器映射中某块目标存储器的读/写属性。如果存储器范围与一个已有的存储器范围重合,则在存储器映射中使用新设置的属性。

参数说明如下:

- address 必需的参数,指定存储器范围的开始地址。参数可以是绝对地址、C 表达式、C 函数名或汇编语言标号。
- page 必需的参数,指定存储器类型。0——程序存储器,1——数据存储器,2——I/O。
- length 必需的参数,定义存储器范围的长度。此参数可以是任何 C 表达式。
- readable 必需的参数,定义存储器范围是否可读。0——不可读,1——可读。
- writable 必需的参数,定义存储器范围是否可写。0——不可写,1——可写。

合法的调用方式如下:

```
GEL_MapAdd(0x1000, 0, 0x300, 1, 1);
```

(2) GEL_MapDelete(address, page): 从存储器映射中删除一块存储范围。删除后,CCS2 调试器将不从目标板的该块存储范围读或写数据。如果显示的存储地址为不可读的,则 CCS2 调试器将不显示目标板的相应值,而是显示其默认值。

两个参数 address 和 page 与 GEL_MapAdd 函数的相应参数含义相同。

合法的调用方式如下:

```
GEL_MapDelete(0x1000, 0);
```

(3) GEL_MapOn(): 使能存储器映射。CCS2 调试器不从具有不可读属性的存储块中读数据,也不向具有不可写属性的存储块中写数据。当存储器映射刚使能时,整个存储器范围都被假设为不可写/不可读属性,此时可使用 GEL_MapAdd 函数添加存储块,并允许调试器存取合法的存储块。当目标系统加电时,存储器映射是被禁止的。

(4) GEL_MapOff(): 禁止存储器映射。值得注意的是禁止存储器映射可能导致目标系统的总线错误,因为 CCS2 调试器可能存取一个不存在的存储器。

(5) GEL_MemoryReset(): 复位存储器映射。该函数使所有存储器属性为不可读/不可写。

(6) GEL_MemoryFill(startAddress, page, length, pattern): 使用指定的值填充某块存储器。

参数说明如下:

- startAddress 必需的参数,指定存储器块的起始地址。

- `page` 必需的参数,指定存储器类型。0- 程序存储器,1- 数据存储器,2- I/O。对 Simulator 而言,不支持 I/O(`page` 为 2 时)。该参数的默认值为 0。
- `length` 必需的参数,定义要填充的字数
- `pattern` 必需的参数,指定填充值(整个存储器块都充此值)。

合法的调用方式如下:

```
GEL_MemoryFill(0x1000,0,0x100,0xa5a5);
```

(7) `GEL_MemoryLoad(startAddress, page, length, "filename")`:从指定文件中加载一块存储器。数据块由 `startAddress`, `Page` 和 `Length` 参数指定。如果加载的文件后缀为 `.out`,则使用 COFF 格式;否则 CCS2 调试器根据文件中的头信息判断文件格式。

该函数的前 3 个参数含义与 `GEL_MemoryFill` 函数一样。`fileName`(用双引号标注)指定存放目标数据的文件名。

合法的调用方式如下:

```
GEL_MemoryLoad(0x1000,0,0x100, "c:\\workdir\\temp.dat");
```

(8) `GEL_Memorysave(startAddress, page, length, "filename")`:将某个目标板存储器块保存到指定的文件中。数据块由 `startAddress`, `Page` 和 `Length` 参数指定。如果加载的文件后缀名为 `.out`,则使用 COFF 格式;否则使用 C 语言风格的 Hex 格式。

该函数的参数含义与 `GEL_Memory` 函数完全相同。

合法的调用方式如下:

```
GEL_MemorySave(0x1000,0,0x100, "c:\\workdir\\temp.dat");
```

6. 插入汇编代码类

`GEL_PatchAssembly(address, page, "patchString")`:将汇编代码插入到某个指定地址。前两个参数在前面的 GEL 函数中已有说明,第三个参数 `patchSstring` 指定要插入的汇编指令字符串。

合法的调用方式如下:

```
GEL_PatchAssembly(0x1000,1,"LAR AR4, #01h");
```

7. 工程类

(1) `GEL_ProjectBuild()`:创建当前的工程,生成 DSP 程序。

(2) `GEL_ProjectLoad("fileName")`:加载一个指定的工程文件。

(3) `GEL_ProjectRebuildAll()`:对当前工程进行编译、汇编和链接,生成 DSP 程序。

8. 系统类

`GEL_System("dosCommand", param1, ..., param4)`:在 CCS2 集成开发环境(IDE)中执行一个 DOS 命令。执行 DOS 命令的输出,输送到 IDE 的一个输出窗口中。可执行的 DOS 命令限于那些只产生文本输出(不是图形方式)及在执行后不需要用户再输入信息的命令。

有的 DOS 命令可以带参数执行,`GEL_System()`函数中的 `param1, ..., param4` 可指定执行 DOS 命令的参数,参数还可以是 DSP 目标板上产生的数据。

如果需要带参数,则需使用格式说明符,有关格式说明符的说明参见 `GEL_TextOut()` 函数。格式说明符与参数一一对应,有多少个参数就应有多少个格式说明符。如果参数比格式

说明符多,则多余的参数被忽略。

GEL_System()函数可用来扩展 CCS2 的功能。例如,可以使用该函数在后台执行一个编译任务,并将结果输出到 CCS2 的输出窗口中。

调用方式如下:

```
GEL_System("dir");           /* 显示目录命令 */
GEL_System("dir *.dat");     /* 显示 *.dat 文件 */
GEL_System("dir%s", "*.dat"); /* 与上一条语句功能相同 */
GEL_System("myfunc %f %d %s", targVar, 3, "-01"); /* 假设 targVar 是在目标 DSP 程序中定义的变量,其值为 3.14,则最终的 DOS 命令为:myfunc 3.143-01 */
```

9. 观察窗口类

(1) GEL_WatchAdd("expression", "label"):将表达式添加到 Watch 窗口中。参数 label 是可选的,用来指示观察的条目。参数 expression 是必需的,指定需添加到 Watch 窗口的表达式。该表达式可能包含格式符号,用来改变 Watch 窗口中变量的显示格式。

Watch 窗口默认的显示格式与显示变量的类型有关。为改变显示格式,在变量后跟一个逗号和 1 个格式字母即可。如 myVar, x=0x1234 表示用十六进制显示变量 myVar。格式字母的说明如下:

- d——十进制数;
- e——指数浮点数;
- f——十进制浮点数;
- x——十六进制数;
- 0——八进制数;
- u——无符号整数;
- c——ASCII 字符(字节);
- p——打包的 ASCII 字符,使用 big endian 格式。

调用方式如下:

```
GEL_WatchAdd ("*(int*)0x1000,x", "Task Number");
GEL_WatchAdd("i");
```

(2) GEL_WatchDel("expression"):将一个由参数 expression 指定的表达式从 Watch 窗口串移去。该表达式应与 Watch 窗口的表达式完全一样。

合法的调用方式如下:

```
GEL_WatchDel("*(int*)0x1000,x");
```

(3) GEL_WatchReset():清除 Watch 窗口中的所有表达式。

10. 扩展存储器类

(1) GEL_XMDef(Map, RegAddr, Type, Start, Mask):定义 TTMS320C548/C549 等芯片的扩展存储器地址范围。

参数说明如下:

- Map 扩展存储器映射的存储空间类型。0——程序空间,1——数据空间。

- RegAddr 映射寄存器地址(0x1E)。
- Type 映射寄存器的存储器类型。0——程序空间,1——数据空间。
- Start 存储器映射范围的开始值(如果 OVLY=1,则使用 0x8000)。
- Mask 代表存储寄存器大小的标志位。

合法的调用方式如下:

```
GEL_Xmdef(0,0x1E,1,0x8000,0x7EF);
```

(2) GEL_XMOn():使能 TMS320C548/C549 等芯片的扩展存储器映射。

9.8 GEL 使用实例

本节将介绍如何利用 GEL 函数进行编译、链接和运行 DSP 应用程序,同时将介绍如何编写在函数内部定义所有变量的 GEL 函数。

9.8.1 一个简单的 GEL 函数

(1) 在 File 菜单中,执行 New→Source File 命令。

(2) 在源代码编辑窗口中,输入如下代码:

```
menuitem "GEL Welcome Tool";  
hotmenu Welcome_To_GEL_Function()  
{  
    GEL_TextOut("GEL is a solid tool.\n");  
}
```

(3) 在 File 菜单中,执行 Save As 命令。

(4) 浏览文件夹 D:\application\ccs\tutorial\sim54xx\gelsolid\,在保存对话框中,选择 GEL 类型的文件,把文件存为 test.gel。

(5) 在 File 菜单中,执行 Load GEL 命令,将刚才新建立的 test.gel 文件载入。

(6) 在 GEL 菜单中,选择 GEL Welcome Tool→Welcome To GEL Function 命令,下面的文字将出现在 output 窗口,如图 9.5 所示。

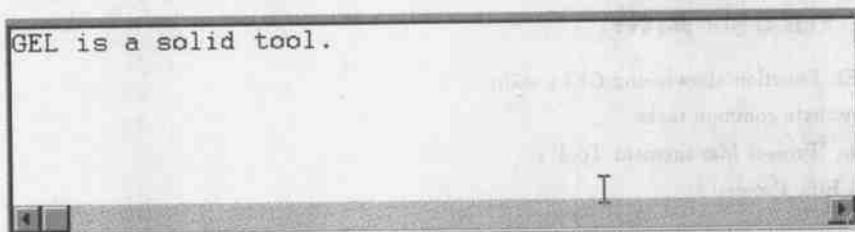


图 9.5 output 窗口的输出

9.8.2 定义局部变量

在 GEL 函数中定义局部变量和标准 C 语言方式一样,必须在程序的数据段声明,然后在程序段中使用。下面的例子利用定义的局部变量在 output 窗口中输出 10 次信息。

(1) 在 File 菜单中, 执行 New→Source File 命令。

(2) 在源代码编辑窗口中, 输入如下代码:

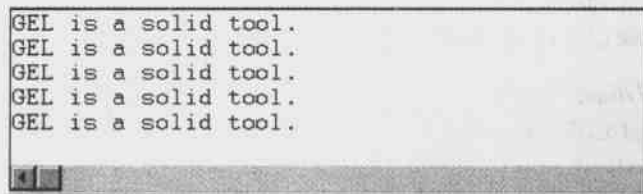
```
menuItem "GEL Welcome Tool";
hotmenu Using_Local_Variables()
{
    int i;
    for (i = 0; i < 10; i++)
        GEL_TextOut("GEL is a solid tool.\n");
}
```

(3) 在 File 菜单中, 执行 Save As 命令。

(4) 浏览文件夹 D:\application\ccs\tutorial\sim54xx\gelsolid\, 在保存对话框中, 选择 GEL 类型的文件, 把文件存为 localvariables.gel。

(5) 在 File 菜单中, 执行 Load GEL 命令, 将刚才新建立的 localvariables.gel 文件载入。

(6) 在 GEL 菜单中, 选择 GEL Welcome Tool→ Using Local Variables 命令, 下面的文字将 10 次出现在 output 窗口, 如图 9.6 所示。



```
GEL is a solid tool.
GEL is a solid tool.
GEL is a solid tool.
GEL is a solid tool.
GEL is a solid tool.
```

图 9.6 output 窗口的输出

9.8.3 GEL 函数的自动执行

本节将用 GEL 自动执行一些任务, 如打开存在的工程及编译并运行 DSP 程序等, 所使用的 GEL 函数都是标准的 GEL 库函数。

(1) 在 File 菜单中, 执行 Open 命令。

(2) 浏览文件夹 D:\application\ccs\tutorial\sim54xx\gelsolid\, 打开 projectmanagement.gel 文件, 查看如下源代码:

```
// A GEL Function showcasing GELs ability
// to automate common tasks
menuItem "Project Management Tool";
hotmenu Run_Project()
{
    // Call standard GEL library function to load project
    GEL_ProjectLoad("C:\\source\\HelloDSP.pjt");
    // Call standard GEL library function to build project
    GEL_ProjectBuild();
    // Call standard GEL library function to load the DSP program
    GEL_Load("C:\\Source\\Debug\\HelloDSP.out");
```

```
// Call standard GEL library function to run the project
GEL_Run();
// Indicate that the GEL script is finished
GEL_TextOut("Finished Executing GEL Script!");
}
```

(3) 在源代码中,做如下两处改动(加黑显示),修改后的源代码如下所示:

```
// A GEL Function showcasing GELs ability
// to automate common tasks
menuitem "Project Management Tool";
hotmenu Run_Project()
{
// Call standard GEL library function to load project
GEL_ProjectLoad("D:\\application\\ccs\\tutorial\\sim54xx\\gelsolid\\HelloDSP. pjt ");
// Call standard GEL library function to build project
GEL_ProjectBuild();
// Call standard GEL library function to load the DSP program
GEL_Load("D: \\application\\ccs\\tutorial\\sim54xx\\gelsolid\\Debug\\HelloDSP. out");
// Call standard GEL library function to run the project
GEL_Run();
// Indicate that the GEL script is finished
GEL_TextOut("Finished Executing GEL Script!");
}
```

(4) 在 File 菜单中,执行 Save As 命令。

(5) 浏览文件夹 D:\application\ccs\tutorial\sim54xx\gelsolid\,在保存对话框中,选择 GEL 类型的文件,把文件存为 projectmanagement. gel。

(6) 在 File 菜单中,执行 Load GEL 命令,将刚才修改好的 projectmanagement. gel 文件载入。

(7) 在 GEL 菜单中,选择 Project Management Tool→Run_Project 项。该命令将载入 hellodsp 工程,编译并执行 hellodsp. out 程序,程序运行将输出如图 9.7 所示的信息。

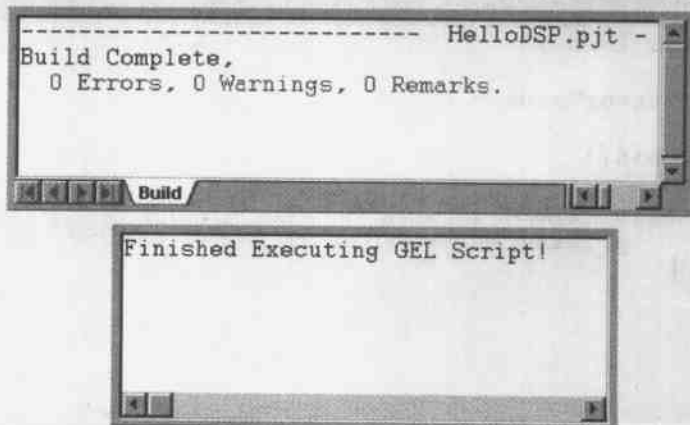


图 9.7 程序运行输出的信息

9.8.4 用 GEL 控制 DSP 变量

本节介绍如何用 GEL 控件去动态控制 DSP 变量。首先建立一个定义了 Slider 控件和对话框的 GEL 文件,并用该文件去控制 counterValue 变量。

(1) 在 File 菜单中,执行 New→Source File 命令。

(2) 在源代码编辑窗口中,输入如下代码:

```
menuitem "Set Counter Value"
dialog Set_Counter(counterParam "Load")
{
    counterValue = counterParam;
}

menuitem "Vary Counter Value"
slider Vary_Counter(0, 10, 1, 1, counterParam)
{
    counterValue = counterParam;
}
```

(3) 在 File 菜单中,执行 Save As 命令。

(4) 浏览文件夹 D:\application\ccs\tutorial\sim54xx\gelsolid\,在保存对话框中,选择 GEL 类型的文件,把文件存为 Counter.gel。

对话框控件是用关键词 dialog 定义的,带有一个 Load 参数。在程序中,将 counterParam 的值传给 counterValue。在 GEL 文件中定义的变量,一定要与 DSP 程序中定义的变量一致。

Slider 控件是用关键词 slider 定义的,从左到右的参数是:移动的最小值、最大值、每次单击移动的值和最终移动的值。

(5) 在 File 菜单中,执行 Load GEL 命令,将刚才新建立的 Counter.gel 文件载入。

(6) 在工程视图中,双击 multidspwelcome.c 以打开它,在源代码中设置如图 9.8 所示的断点。设置断点的目的是当程序运行到断点时,自动地从 DSP 应用程序跳到 GEL 文件中执行。

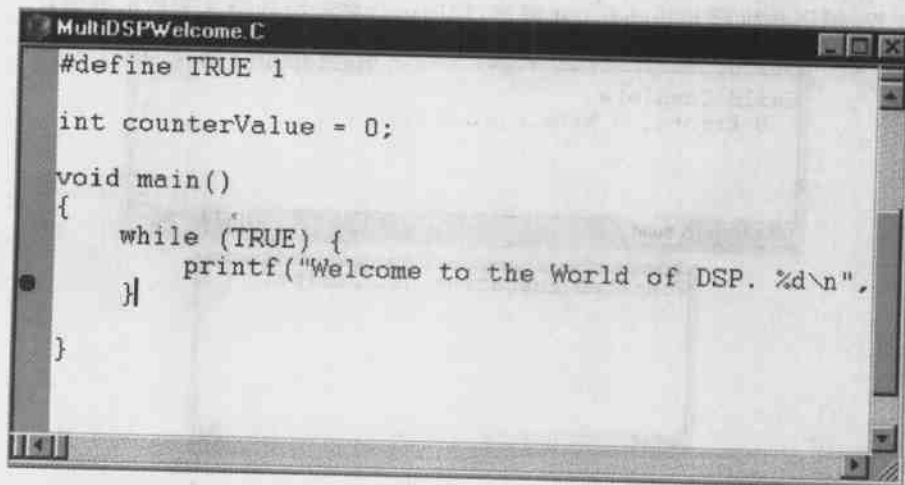


图 9.8 设置断点

(7) 在 File 菜单中,执行 Load Program 命令,在 D:\application\ccs\tutorial\sim54xx\gelsolid\Debug 下找到 hellodsp.out 文件并载入之。

(8) 在 Debug 菜单中执行 Run 命令,运行程序。

(9) 当程序在断点处停止执行时,在 GEL 菜单下选择 Set Counter Value→Set_Counter,这时将出现如图 9.9 所示的对话框,在其中填入 20,然后单击 Execute 按钮。

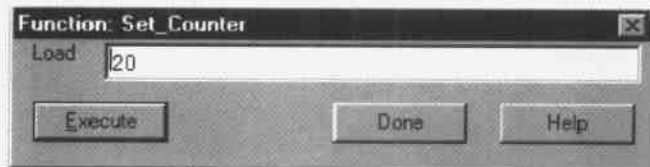


图 9.9 Set Counter 的值

(10) 在 Debug 菜单中,执行 Run 命令,这时,在 output 窗口将输出如图 9.10 所示的信息。

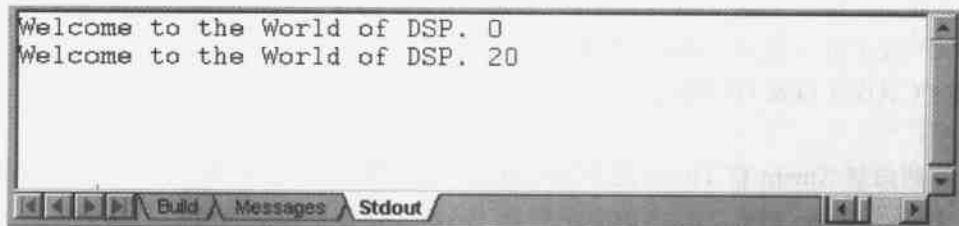


图 9.10 output 窗口输出的信息

(11) 当程序再次停留在断点处时,执行菜单命令 GEL→Vary Counter Value→Vary_Counter,这时将出现如图 9.11 所示的滑动条控件。

(12) 上下调节控件的位置,以改变变量的值。

(13) 在 Debug 菜单中,执行 Run 命令,继续执行程序,这时,CounterValue 的值被改变,如图 9.12 所示。



图 9.11 slider 控件

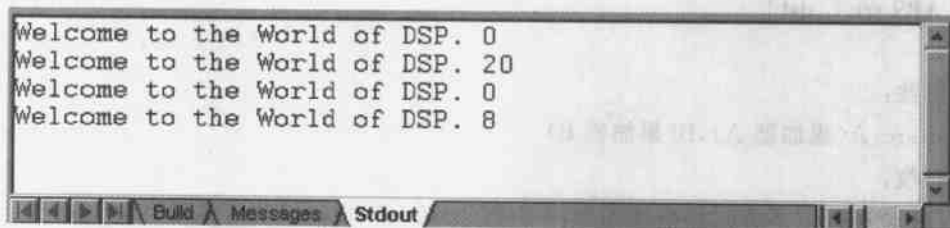


图 9.12 counterValue 的值

附录 指令详解

1. ABDS Xmem, Ymem

操作数:

Xmem, Ymem: 双数据存储操作数

执行:

$(B) + |(A(32-16))| \rightarrow B$

$((Xmem) - (Ymem)) << 16 \rightarrow A$

状态位:

被 OVM、FRCT 以及 SXM 位影响。

影响 C、OVA 以及 OVB 位。

说明:

计算两向量 Xmem 和 Ymem 之差的绝对值。累加器 A 的高端(位 32~16)的绝对值加到累加器 B 中。Xmem 减去 Ymem 的差左移 16 位, 然后存在累加器 A 中。如果分数方式位为 1, 即(FRCT = 1), 则该绝对值需要乘 2。

例: ABDST * AR3+, * AR4 +

执行前		
A	FF	ABCD 0000
B	00	0000 0000
AR3	0100	
AR4	0200	
FRCT	0	
数据存储器		
0100h	0055	
0200h	00AA	

执行后		
A	FF	FFAB 0000
B	00	0000 5433
AR3	0101	
AR4	0201	
FRCT	0	
0100h	0055	
0200h	00AA	

2. ABS src [, dst]

操作数:

dst, src: A(累加器 A), B(累加器 B)

状态位:

OVM 位按照以下方式影响该指令:

如果 OVM=1, 80 0000 0000h 的绝对值是 00 7FFF FFFFh。

如果 OVM=0, 80 0000 0000h 的绝对值是 80 0000 0000h。

影响 C 以及 Ovdst 位(如果 dst=src,影响 Ovsrc 位)。

说明:

计算 src 的绝对值,然后装入 dst。如果没有定义 dst,绝对值装入 src 中。

例:ABS A,B

执行前			执行后				
A	FF	FFFF	FECB	A	FF	FFFF	FECB
B	FF	FFFF	FC18	B	00	0000	0035

3. ADD

语法:

- (1) ADD Smem,src
- (2) ADD Smem,TS,src
- (3) ADD Smem,16,src[,dst]
- (4) ADD Smem[,SHIFT],src[,dst]
- (5) ADD Xmem,SHFT,src
- (6) ADD Xmem,Ymem,dst
- (7) ADD #1k[,SHFT],src[,dst]
- (8) ADD #1k,16,src[,dst]
- (9) ADDsrc[,SHIFT],[,dst]
- (10) ADDsrc, ASM[,dst]

执行:

- (1) (Smem) + (src) → src
- (2) (Smem) << (TS) + (src) → src
- (3) (Smem) << 16 + (src) → dst
- (4) (Smem) [<< SHIFT] + (src) → dst
- (5) (Xmem) << SHFT + (src) → src
- (6) ((Smem) + (Smem)) << 16 → dst
- (7) 1k << SHFT + (src) → dst
- (8) 1k << 16 + (src) → dst
- (10) (src or [dst]) + (src) << ASM → dst

状态位:

被 SXM 位和 OVM 位影响。

影响 C 位和 Ovdst 位(如果 dst=src,影响 Ovsrc 位)。

说明:

把一个 16 位的数加到选定的累加器,或一个采用双数据存储器操作数寻址的 16 位操作数 Smem 中。这个 16 位数可以是以下情况中的一个:

- (1) 单数据存储器操作数(Smem);
- (2) 双数据存储器操作数(Ymem);

(3) 一个 16 位的立即操作数 ($\neq 1k$):

(4) src 中的移位数。

如果定义了 dst, 结果就存在 dst 中; 否则, 结果存在 src 中。大部分第二操作数要移位。左移时低位添 0; 右移时高位情况为

如果 SXM=1, 进行符号扩展;

如果 SXM=0, 添 0。

例: ADD *AR3+, 14, A

执行前		
A	00 0000 1200	
B		1
AR3		0100
SXM		1
数据存储器		
0100h		1500

执行后		
A	00 0540 1200	
B		0
AR3		0101
SXM		1
数据存储器		
0100h		1500

4. ADDC Smem, src

操作数:

Smem: 单数据存储器操作数

src: A(累加器 A), B(累加器 B)

执行:

$(Smem) + (src) + (C) \rightarrow src$

状态位:

被 OVM 位和 C 位影响。

影响 C 位和 Ovsrsrc 位。

说明:

将 16 位单数据存储器操作数 Smem 和进位位(C)的值加到 src 中, 结果存在 src 中。无论 SXM 位的值是多少, 都不进行符号扩展。

例: ADDC *AR2(5)+, A

执行前		
A	00 0000 0013	
B		1
AR2		0100
数据存储器		
0105h		0004

执行后		
A	00 0000 0018	
B		0
AR2		0105
数据存储器		
0105h		0004

5. ADDM #1k, Smem

操作数:

Smem:单数据存储器操作数

— $32\,768 \leq 1k \leq 32\,767$

执行:

$\#1k + (\text{Smem}) \rightarrow \text{Smem}$

状态位:

被 OVM 位和 SXM 位影响。

影响 C 位和 OVA 位。

说明:

16 位单数据存储器操作数 Smem 与 16 位立即数 1k 相加,结果存在 Smem 中。注意:该指令不能循环执行。

例:ADDM #0123Bh, *AR4+

执行前		执行后	
AR4	0100	AR4	0101
数据存储器			
0100h	0004	0100h	123F

6. ADDS Smem,src

操作数:

Smem:单数据存储器操作数

src:A(累加器 A),B(累加器 B)

执行:

$\text{unsign}(\text{Smem}) + (\text{src}) \rightarrow \text{src}$

状态位:

被 OVM 位影响。

影响 C 位和 Ovsrsrc 位。

说明:

把不带符号的 16 位单数据存储器操作数 Smem 加到 src 中,结果存在 src 中。无论 SXM 为何值,都不进行符号扩展。

例:ADDS *AR2-, B

执行前		执行后	
B	00 0000 0003	B	00 0000 F009
C	x	C	0
AR2	0100	AR2	00FF
数据存储器			
0104h	F006	0104h	F006

7. AND

语法:

- (1) AND Smem,src
- (2) AND #1k [,SHFT],src [,dst]
- (3) AND #1k, 16,src [,dst]
- (4) AND src [,SHIFT],[,dst]

操作数:

Smem:单数据存储操作数

src:A(累加器 A),B(累加器 B)

$-16 \leq \text{SHIFT} \leq 15$

$0 \leq \text{SHFT} \leq 15$

$0 \leq 1k \leq 65\,535$

执行:

- (1) (Smem) AND(src)→src
- (2) $1k \ll \text{SHFT}$ AND(src)→dst
- (3) $1k \ll 16$ AND(src)→dst
- (4) (dst)AND(src) $\ll \text{SHIFT}$ →dst

状态位:无。

说明:

该指令功能是使以下几种数据和 src 相“与”:

- (1) 一个 16 位单数据存储操作数 Smem;
- (2) 一个 16 位立即数 1k;
- (3) 源或目的累加器(src 和 dst)。

如果确定了移位,操作数在移位后才进行与操作数。对于左移,低位添 0;高位不进行符号扩展。对于右移,高位不进行符号扩展。

例:AND * AR3+,A

执行前	
A	00 00FF 1200
AR3	0100
数据存储器	
0100h	1500

执行后	
A	00 0000 1000
AR3	0101
0100h	1500

8. ANDM #1k,Smem

操作数:

Smem:单数据存储操作数

$0 \leq 1k \leq 65\,535$

执行:

1k AND(Smem)→Smem

状态位:无。

说明:

16 位单数据存储器操作数 Smem 和一个 16 位立即数 1k 相“与”,结果存在原 Smem 所在单元中。注意:该指令不可循环执行。

例:ANDM #00FFh, * AR4+

执行前		执行后	
AR4	0100	AR4	0101
数据存储器			
0100h	0444	0100h	0044

9. B[D] pmad

操作数:

$0 \leq \text{pmad} \leq 65\,535$

执行:

pmad→PC

状态位:无。

说明:

指令指针指向指定的程序存储器地址(pmad),该地址可以是符号或一个数字。如果是延迟转移(指令后缀 D 确定),紧接着转移指令的两条单字指令或双字指令从程序存储器中取出先执行。注意:该指令不能循环执行。

例:BD 1000h

执行前		执行后	
PC	1F45	PC	1000

10. BAC C[D] src

操作数:

src:A(累加器 A),B(累加器 B)

执行:

(src(15-0))→PC

状态位:无。

说明:

程序指针 PC 指向 src 的低位(位 15 ~ 0)所确定的 16 位地址。如果是延迟转移(由指令后缀 D 确定),紧跟着转移指令的 2 条单字或 1 条双字指令从程序存储器中取出先执行。注意:该指令不能循环执行。

11. BANZ[D] pmad,Sind

操作数:

Sind:单间接寻址操作数

$0 \leq \text{pmad} \leq 65\,535$

执行:

If((ARx) \neq 0)

Then

pmad \rightarrow PC

Else

(PC)+2 \rightarrow PC

状态位:无。

说明:

如果当前辅助寄存器 ARx 不为 0,程序指针转移到指定的程序存储器地址(pmad);否则 PC 指针加 2。如果是延迟转移(由指令后缀 D 决定),紧接着转移指令的 2 条单字指令或 1 条双字指令从程序存储器中取出先执行。注意:该指令不能循环执行。

例: BANZ 2000h, *AR3—

	执行前		执行后
PC	1000	PC	1002
AR3	0000	AR3	FFFF

12. BC[D] pmad,cond [,cond [,cond]]

操作数:

$0 \leq \text{pmad} \leq 65\,535$

执行:

If(cond(s))

Then

pmad \rightarrow PC

Else

(PC)+2 \rightarrow PC

状态位:

影响 OVA 位或 OVB 位。

说明:

如果满足特定的条件,指令转移到程序存储器地址(pmad)上,特定的条件见表附.1。采用延迟方式,紧接着该指令的 2 条单字指令或 1 条双字指令从程序存储器中取出先执行。如果条件满足,那么这 2 个字将从流水中冲掉,程序从 pmad 开始执行;如果条件不满足,PC 加 2 且紧接着该指令的 2 个字继续执行。

表附.1 条件代码对应的条件

条 件	说 明	条 件	说 明
BIO	BIO 为低	NBIO	BIO 为高
C	$C = 1$	NC	$C = 0$
TC	$TC = 0$	NTC	$TC = 0$
AEQ	$(A) = 0$	BEQ	$(B) = 0$
ANEQ	$(A) \neq 0$	BNEQ	$(B) \neq 0$
AGT	$(A) > 0$	BGT	$(B) > 0$
AGEQ	$(A) \geq 0$	BGEQ	$(B) \geq 0$
ALT	$(A) < 0$	BLT	$(B) < 0$
ALEQ	$(A) \leq 0$	BLEQ	$(B) \leq 0$
ALV	A 溢出	BOV	B 溢出
ANOV	A 没有溢出	BNOV	B 没有溢出
UNC	无条件		

指令在把控制权交给程序的另一部分之前,可对多个条件进行测试。指令可测试相互独立的条件或者是相互关联的条件;但多个条件只能出自同一组的不同类中。条件的分组和分类见表附.2。

例:BC 2000h, AGT

执行前				执行后			
A	00	0000	0053	A	00	0000	0053
PC				PC			
	1000				2000		

表附.2 指令的条件代码的分组和分类

第一组			第二组		
A 类	B 类		A 类	B 类	C 类
EQ NEQ LT	OV		TC	C	BIO
LEQ GT GEQ	NOV		NTC	NC	NBIO

13. BIT Xmem, BITC

操作数:

Xmem: 双数据存储操作数

$0 \leq \text{BITC} \leq 15$

执行:

$(\text{Xmem}(15 - \text{bitC})) \rightarrow \text{TC}$

状态位:

影响 TC 位。

说明:

把双数据存储器操作数 Xmem 的指定位复制到状态寄存 ST0 的 TC 位。

例: BIT * AR5+, 15-12

执行前		执行后	
T	<div>C</div>	T	<div>C</div>
TC	<div>1</div>	TC	<div>1</div>
AR0	<div>0008</div>	AR0	<div>0008</div>
AR7	<div>0100</div>	AR7	<div>0108</div>
数据存储器			
0100h	<div>0008</div>	0100h	<div>0008</div>

14. BITF Smem, #1k

操作数:

Smem: 单数据存储器操作数

$0 \leq 1k \leq 65535$

执行:

If((Smem) AND 1k) = 0

Then

0 → TC

Else

1 → TC

状态位:

影响 TC 位。

说明:

测试单数据存储器值 Smem 中指定的某些位, 假如指定的一位或多位为 0, 状态寄存器 ST0 的 TC 位清 0; 否则 TC 置 1。1k 常数在测试一位或多位时起屏蔽作用。

例: BITF 5, 00FFh

执行前		执行后	
TC	<div>x</div>	TC	<div>x</div>
DP	<div>004</div>	DP	<div>004</div>
数据存储器			
0100h	<div>5400</div>	0100h	<div>5400</div>

15. BITT Smem

操作数:

Smem:单数据存储操作数

执行:

$(\text{Smem}(15 - T(3-0))) \rightarrow \text{TC}$

状态位:

影响 TC 位。

说明:

把单数据存储器 Smem 的指定位复制到 ST0 的 TC 位,T 寄存器的低 4 位(位 3~0)值确定了被复制的位代码,高 12 位(位 15~T(3~0))对应着位地址。

16. CALA[D] src

操作数:

src: A(累加器 A), B(累加器 B)

执行:

非延时

$(\text{SP}) - 1 \rightarrow \text{SP}$

$(\text{PC}) + 1 \rightarrow \text{TOS}$

$(\text{src}(15-0)) \rightarrow \text{PC}$

延时

$(\text{SP}) - 1 \rightarrow \text{SP}$

$(\text{PC}) + 3 \rightarrow \text{TOS}$

$(\text{src}(15-0)) \rightarrow \text{PC}$

状态位:无。

说明:

程序指针转移到 src 的低位所确定的 16 位地址单元,返回地址压入栈项。如果是延迟调用,紧接着该指令的两条单字指令或一条双字指令从程序存储器中取出来先执行。本指令不可循环执行。

例:CALA A

执行前				执行后			
A	00	0000	3000	A	00	0000	3000
PC				PC			
SP				SP			
数据存储器				数据存储器			
0100h				0100h			

17. CALL[D] pmad

操作数:

$0 \leq \text{pmad} \leq 65\ 535$

执行:

非延时

$(SP) - 1 \rightarrow SP$

$(PC) + 2 \rightarrow TOS$

$pmad \rightarrow PC$

延时

$(SP) - 1 \rightarrow SP$

$(PC) + 4 \rightarrow TOS$

$pmad \rightarrow PC$

状态位:无。

说明:

程序指针指向确定的程序存储器地址($pmad$),返回地址在 $pmad$ 装入 PC 之前压入栈顶保存。如果是延迟调用,紧接着该指令的 2 条单字指令或 1 条双字指令从程序存储器中取出来先执行。注意:该指令不能循环执行。

例:CALLD 1000h

ANDM #4444h, *AR1+

执行前		执行后	
PC	0025	PC	1000
SP	1111	SP	1110
数据存储器			
1110h	4567	1110h	0029

18. CC[D] pmad, cond [,cond [,cond]]

操作数:

$0 \leq pmad \leq 65535$

执行:

非延时

If(cond(s))

Then

$(SP) - 1 \rightarrow SP$

$(PC) + 2 \rightarrow TOS$

$pmad \rightarrow PC$

Else

$(PC) + 2 \rightarrow PC$

延时

If(cond(s))

Then

$(SP) - 1 \rightarrow SP$

(PC)+4 → TOS

pmad → PC

Else

(PC)+2 → PC

状态位:

影响 OVA 位或 OVB 位。

说明:

本指令的条件代码和指令 BC [D] 的条件代码相同。当满足确定的条件时, 程序指针 PC 指向程序存储器地址 (pmad); 如果不满足条件, 程序指针 PC 加 2。如果是延迟调用, 该指令的后两个字指令先取出执行, 且不会影响被测试的条件。该指令可测试多个条件, 情况同 BC [D] 指令。

例: CC 2222h, AGT

执行前			执行后		
A	00	0000 3000	A	00	0000 3000
PC		0025	PC		2222
SP		1111	SP		1110
数据存储器					
1110h		4567	1110h		0027

19. CMPLsrc [,dst]

操作数:

src,dst: A(累加器 A), B(累加器 B)

执行:

(src) → dst

状态位: 无。

说明:

计算 src 反码(逻辑“反”), 结果存放在 dst 中; 若没有指定 dst, 就存放在 src 中。

例: CMPL A, B

执行前			执行后		
A	FC	DFFA	A	FC	DFFA
B	00	0000 7899	B	03	2005 5155

20. CPM Smem, # 1k

操作数:

Smem: 单数据存储操作数

-32 768 ≤ 1k ≤ 32 767

执行:

If(Smem)→1k

Then

1→TC

Else

0→TC

状态位:

影响 TC 位。

说明:

比较 16 位单数据存储器操作数 Smem 和 16 位常数 1k 是否相等。若相等,TC 置 1;否则清 0。

例:CMPPM * AR4+, 0404h

	执行前		执行后
TC	1	TC	0
AR4	0100	AR4	0101
数据存储器 0100h	4444	0100h	4444

21. CMPR CC, ARx

操作数:

$0 \leq CC \leq 3$

ARx; AR0~AR7

执行:

If(cond)

Then

1→TC

Else

0→TC

状态位:

影响 TC 位。

说明:

比较指定的辅助寄存器(ARx)和 AR0 的值,然后根据结果决定 TC 的值。比较由条件代码 CC 的值决定,可见表附.3。若条件满足 TC 置 1;否则清 0。所有的条件都以无符号运算计算。

表附.3 条件代码的说明

条 件	条件代码(CC)	说 明
EQ	00	测试 ARx 是否等于 AR0
LT	01	测试 ARx 是否小于 AR0
GT	10	测试 ARx 是否大于 AR0
NEQ	11	测试 ARx 是否不等于 AR0

例:CMPR 2,AR4

	执行前		执行后
TC	1	TC	0
AR0	FFFF	AR0	FFFF
AR4	7FFF	AR4	7FFF

22. CMPS src,Smem

操作数:

Smem:单数据存储操作数

src:A(累加器 A),B(累加器 B)

执行:

If((src(31—16))>(src(15—0)))

Then

(src(31—16))→Smem

(TRN)<<1 →TRN

0→TRN(0)

0→TC

Else

(src(15—0))→Smem

(TRN)<<1 →TRN

1→TRN(0)

1→TC

状态位:

影响 TC 位。

说明:

比较位于源累加器的高端和低端两个 16 位二进制补码值的大小,把较大值存在单数据存储单元 Smem 中。如果源累加器的高端(位 31~16)较大,过渡寄存器(TRN)左移一位,最低位添 0,TC 位清 0;反之,如果源累加器的低端(位 15~0)较大,过渡寄存器(TRN)左移一位,最低位添 1,TC 位置 1。该指令不遵从标准的流水操作。流水操作中的比较是在读操作数阶段完成的。因而,源累加器的值是指令执行前一个阶段的值。TRN 寄存器和 TC 位在执行阶段被修改。

例:CMPS A, * AR4 +

	执行前		执行后
A	00 2345 7899	A	00 2345 7899
TC	0	TC	1
AR4	0100	AR4	0101
TRN	4444	TRN	8889
数据存储器			
0100h	0000	0100h	7899

23. DADD Lmem,src [, dst]

操作数:

Lmem:长数据存储操作数

src,dst: A(累加器 A),B(累加器 B)

执行:

If C16=0

Then

(Lmem)+(src) →dst

Else

(Lmem(31-16))+(src(31-16))→dst(39-16)

(Lmem(15-0))+(src(15-0))→dst(15-0)

状态位:

只有在 C16=0 时被 SXM 位和 OVM 位影响。

影响 C 位和 Ovdst 位。

说明:

源累加器的内容加到 32 位长数据存储操作数 Lmem 中。如果定义了目的累加器,就把结果存在其中;否则存在源累加器中。C16 的值决定了指令执行的方式。当 C16 = 0 时,指令以双精度方式执行,40 位源累加器的值加到 Lmem 中,饱和度和溢出位都是根据运算的结果来设置。当 C16 = 1 时,指令以双 16 位方式执行,src 的高端(位 31~16)与 Lmem 的高 16 位相加;src 的低端(位 15~0)与 Lmem 的低 16 位相加,饱和度和溢出位在此方式下不受影响,并且无论 OVM 的状态是什么,结果都不进行饱和运算。

例:DADD * AR3+,A,B

执行前				执行后			
A	00	5678	8933	A	00	5678	8933
B	00	0000	0000	B	00	6BAC	BD89
C16			0	C16			0
AR3			0100	AR3+			0100
数据存储器							
0100h			1534	0100h			1534
0101h			3456	0101h			3456

24. DADST Lmem, dst

操作数:

Lmem:长数据存储操作数

dst: A(累加器 A),B(累加器 B)

执行:

If C16 = 1

Then(Lmem(31-16))+(T)→dst(39-16)

(Lmem(15-0))-(T)→dst(15-0)

Else

(Lmem)+(T)≤≤16→dst

状态位:

被 SXM 位和 OVM 位影响。

影响 C 位和 Ovdst 位。

说明:

本指令把 T 寄存器的值加到 32 位长数据存储器操作数 Lmem 中。C16 的值决定了指令执行的方式。当 C16=0 时,指令以双精度方式执行;T 寄存器的值和其左移 16 位得到的值相结合所组成的 32 位数据加到 Lmem 中;结果存在目的累加器中。当 C16=1 时,指令以双 16 位方式执行;Lmem 的高 16 位与 T 寄存器的值相加,存在目的累加器的前 24 位;同时,从 Lmem 的低 16 位减去 T 寄存器的值,结果存在目的累加器的低 16 位。无论 OVM 位的状态是什么,结果都不进行饱和运算。注意:该指令仅当 C16 置 1 时(双 16 位方式)才有意义。

例:DADST * AR3-,A

执行前		执行后	
A	00 0000 0000	A	00 3879 1111
T	2345	T	2345
C16	1	C16	1
AR3	0100	AR3	00FE
数据存储器			
0100h	1534	0100h	1534
0101h	3456	0101h	3456

25. DELAY Smem

操作数:

Smem:单数据存储器操作数

执行:

(Smem)→Smem+1

状态位:无。

说明:

该指令把一单数据存储器单元 Smem 的内容复制到紧接着的较高地址单元中去。数据复制完后,原单元内容保持不变。该功能在数字信号处理应用中实现一个 Z 延迟是相当有用的。这种延迟操作数在 LTD 和 MACD 指令中可以见到。

例:DELAY * AR3

	执行前		执行后
AR3	0100	AR3	0100
数据存储器			
0100h	6CAC	0100h	6CAC
0101h	0000	0101h	6CAC

26. DLD Lmem, dst

操作数:

Lmem: 长数据存储操作数

dst: A(累加器 A), B(累加器 B)

执行:

If C16=0

Then

(Lmem)→ds

Else

(Lmem(31-16))→dst(39--16)

(Lmem(15-0))→dst(15-0)

状态位:

被 SXM 位影响。

说明:

该指令是一个 32 位的长操作数 Lmem 装入目的累加器 dst 中, C16 的值决定了指令执行的方式。当 C16=0 时, 指令以双精度方式执行, Lmem 装入到目的累加器 dst 中。当 C16=1 时, 指令以双 16 位方式执行, Lmem 的高 16 位装入目的累加器 dst 的前 24 位; 同时, Lmem 的低端(位 15~0)装入目的累加器的低端。

例: DLD *AR3+, B

	执行前		执行后
B	00 0000 0000	B	00 6CAC BD90
AR3	0100	AR3	0102
数据存储器			
0100h	6CAC	0100h	6CAC
0101h	BD90	0101h	BD90

27. DRSUB Lmem, src

操作数:

Lmem: 长数据存储操作数

src: A(累加器 A), B(累加器 B)

执行:

If C16=0

Then

(Lmem)-(src)→src

Else

(Lmem(31-16))-(src(31-16))→src(39-16)

(Lmem(15-0))-(src(15-0))→src(15-0)

状态位:

被 SXM 位和 OVM 位影响。

影响 C 位和 Ovdst 位。

说明:

该指令是把一个 32 位的长操作数 Lmem 减去 src 的内容,结果装入 src 中,C16 的值决定了指令执行的方式。当 C16=0 时,指令以双精度方式执行。Lmem 减去 src 的内容(32 位)。结果存放在 src 中。当 C16=1 时,指令以双 16 位方式执行,Lmem 的高 16 位减去 src 的高端(位 31~16),结果存放在 src 的前 24 位(位 39~16);同时,Lmem 的低端(位 15~0)减去 src 的低端,结果存放在 src 的低端。在这种方式下,不管 OVM 的状态如何,都不进行饱和运算。

例:DRSUB * AR3 +,A

执行前			
A	00	5678	8933
B	X		
C16	0		
AR3	0100		

执行后			
A	FF	BEBB	AB23
B	0		
C16	0		
AR3	0102		

28. DSADT Lmem,dst

操作数:

Lmem:长数据存储操作数

dst:A(累加器 A),B(累加器 B)

执行:

If C16=1

Then

(Lmem(31-16))-(T)→dst(39-16)

(Lmem(15-0))-(T)→dst(15-0)

Else

(Lmem)-(T)<<16→dst

状态位:

只有 C16=0 时被 SXM 位和 OVM 位影响。

影响 C 位和 Ovdst 位。

说明:

该指令是把一个 32 位的长操作数 Lmem 减或加 T 寄存器的值,结果装入 dst 中。C16 的

值决定了指令执行的方式。当 $C16=0$ 时,指令以双精度方式执行,T 寄存器的值与其左移 16 位的值连在一起组成的 32 位的值与 Lmem 相减,结果装入 dst 中。当 $C16=1$ 时,指令以双 16 位方式执行,Lmem 的高 16 位减去 T 寄存器的值,结果装入 dst 的高端;同时把 T 寄存器的值加到 Lmem 的低端,结果装入 dst 的低端。在这种方式下,不管 OVM 的状态如何,都不进行饱和运算。注意:该指令只有当 C16 置 1 时才有意义。

例:DSADT * AR3+,A

执行前			执行后		
A	00	0000 0000	A	FF	F1EF 1111
T		2345	T		2345
C		0	C		0
C16		0	C16		0
AR3		0100	AR3		0102
数据存储器					
0100h		1534	0100h		1534
0101h		3456	0101h		3456

29. DST src,Lmem

操作数:

src:A(累加器 A),B(累加器 B)

Lmem:长数据存储操作数

执行:(src(31-0)→Lmem

说明:

把源累加器的内容存放在一个 32 位的长数据存储器单元 Lmem 中。

例:DST B, * AR3+

执行前			执行后		
B	00	6CAC BD90	B	00	6CAC BD90
AR3		0100	AR3+		0102
数据存储器					
0100h		0000	0100h		6CAC
0101h		0000	0101h		BD90

30. DSUB Lmem,src

操作数:

Lmem:长数据存储操作数

src:A(累加器 A),B(累加器 B)

执行:

If C16=0

Then

$(src) - (Lmem) \rightarrow src$

Else

$(src(31-16) - (Lmem(31-16))) \rightarrow src(39-16)$

$(src(15-0) - (Lmem(15-0))) \rightarrow src(15-0)$

状态位:

只有在 C16=0 时被 SXM 位和 OVM 位影响。

影响 C 位和 Ovdst 位。

说明:

该指令是从源累加器中减去 32 位长数据存储器操作数 Lmem 的值,结果装入 src 中。C16 的值决定了指令执行的方式。当 C16=0 时,指令以双精度方式执行,从源累加器中减去 Lmem 的值,结果装入 src 中。当 C16=1 时,指令以双 16 位方式执行,从源累加器 src 的高端(位 31~16)减去 Lmem 的高 16 位,结果装入 src 的前 24 位(位 39~16);同时,从源累加器 src 的低端(位 15~0)减去 Lmem 的低端,结果装入 src 的低端。

例: DSUB *AR3, A

执行前			执行后		
A	00	5678 8933	A	00	4144 54DD
C16		0	C16		0
AR3		0100	AR3		0102
数据存储器					
0100h		1534	0100h		1534
0101h		3456	0101h		3456

31. DSUBT Lmem, dst

操作数:

Lmem: 长数据存储器操作数

dst: A(累加器 A), B(累加器 B)

执行:

If C16=1

Then

$(Lmem(31-16)) - (T) \rightarrow dst(39-16)$

$(Lmem(15-0)) - (T) \rightarrow dst(15-0)$

Else

$(Lmem) - (T) + (T \ll 16) \rightarrow dst$

状态位:

只有在 C16=0 时被 SXM 位和 OVM 位影响。

影响 C 位和 Ovdst 位。

说明:

该指令的功能是从 32 位的长数据存储器操作数 Lmem 减去 T 寄存器的值,结果装入 src

中。C16 的值决定了指令执行的方式。当 C16=0 时,指令以双精度方式执行,T 寄存器的值与其左移 16 位的值连在一起组成的一 32 位数与 Lmem 相减,结果装入 dst 中。当 C16=1 时,指令以双 16 位方式执行,从 Lmem 的高 16 位值中减去 T 寄存器的内容,结果装入目的累加器的高端(位 39~16);同时,从 Lmem 的低 16 位值中减去 T 寄存器的值,结果存放在目的累加器 dst 的低端(位 15~0)。在这种方式下,不管 OVM 的状态如何,都不进行饱和运算。注意:该指令只有当 C16 置为 1 时(双 16 位方式)才有意义。

例:DSUBT *AR3+,A

执行前				执行后			
A	00	0000	0000	A	FF	F1EF	1111
T			2345	T			2345
C16			0	C16			0
AR3			0100	AR3			0102
数据存储器							
0100h			1534	0100h			1534
0101h			3456	0101h			3456

32. EXP src

操作数:

src:A(累加器 A),B(累加器 B)

执行:

If(src)=0

Then 0→T

Else

(src 的引导位数)-8→T

状态位:无。

说明:

计算指数值并把结果存在 T 寄存器中,该值是一个范围在-8~31 之间的带符号的二进制补码值。指数值是通过源累加器 src 的引导位数然后减 8 得到的,引导位数等于消除 40 位源累加器 src 中除符号位以外的有效位所需要左移的位数。指令结束后,源累加器 src 没有被修改。如果引导位数减去 8 的值为负,这是因为累加器的保护位中有有效位。

例:EXP A

执行前				执行后			
A	FF	FFFF	FFCB	A	FF	FFFF	FFCB
T			0000	T			0019

33. FB[D] extpmad

操作数:

$0 \leq \text{extpmad} \leq 7\text{FFFFFF}$

执行:

 $(\text{pmad}(15 \sim 0)) \rightarrow \text{PC}$
 $(\text{pmad}(22 \sim 16)) \rightarrow \text{XPC}$

状态位:无。

说明:

程序指针 PC 指向由 pmad 的位 22~16 决定的页中 pmad 的位 15~0 所确定的程序存储器地址。pmad 可以是一个符号或一个具体的数字。如果是延迟转移,紧接着该指令的 2 条单字指令或 1 条双字指令从程序存储器中取出先执行。注意:该指令不能循环执行。

例:FB 012000h

执行前			执行后		
PC		1000	PC		2000
XPC		00	XPC		01

34. FBACC[D] src

操作数:

src:A(累加器 A),B(累加器 B)

执行:

 $(\text{src}(15 \sim 0)) \rightarrow \text{PC}$
 $(\text{src}(22 \sim 16)) \rightarrow \text{XPC}$

状态位:无。

说明:

该指令是把源累加器 src 的位 22~16 值装入 XPC,并且程序指针指向 src 的低端(位 15~0)所确定的 16 位地址。如果是延迟转移,紧接着该指令的 2 条单字指令或 1 条双字指令从程序存储器中取出先执行。注意:该指令不能循环执行。

例:FBACC A

执行前			执行后		
A	00	0001 3000	A	00	0001 3000
PC		1000	PC		3000
XPC		00	XPC		01

例:FBACCD B

ANDM 4444h *AR1+

执行前			执行后		
B	00	007F 2000	B	00	007F 2000
XPC		01	XPC		7F

35. FCALA[D] src

操作数:

src: A(累加器 A), B(累加器 B)

执行:

非延时

$(SP) - 1 \rightarrow SP$

$(PC) + 1 \rightarrow TOS$

$(SP) - 1 \rightarrow SP$

$(XPC) \rightarrow TOS$

$(src(15-0)) \rightarrow PC$

$(src(22-16)) \rightarrow XPC$

延时

$(SP) - 1 \rightarrow SP$

$(PC) + 3 \rightarrow TOS$

$(SP) - 1 \rightarrow SP$

$(XPC) \rightarrow TOS$

$(src(15-0)) \rightarrow PC$

$(src(22-16)) \rightarrow XPC$

状态位: 无。

说明:

把源累加器 src 的位 22~16 的值装入 PC, 并且指向 src 的低端位 15~0 所确定的 16 位地址。如果是延迟调用, 紧接着该调用指令的 2 条单字指令或 1 条双字指令从程序存储器中取出先执行。注意: 该指令不能循环执行。

例: FCALA A

执行前			执行后		
A	00	007F 3000	A	00	007F 3000
PC		0025	PC		3000
XPC		00	XPC		7F
SP		1111	SP		110F
数据存储器					
1110h		4567	1110h		0026
110Fh		4567	110Fh		0000

例: FCALAD B

ANDM #4444h *AR1+

	执行前		执行后
B	00 0020 2000	B	00 0020 2000
PC	0025	PC	2000
XPC	7F	XPC	20
SP	1111	SP	110F
数据存储器			
1110h	4567	1110h	0028
110Fh	4567	110Fh	007F

36. FCALL[D] extpmad

操作数:

$0 \leq \text{extpmad} \leq 7F\text{ FFFF}$

执行:

非延时

$(\text{SP}) - 1 \rightarrow \text{SP}$

$(\text{PC}) + 2 \rightarrow \text{TOS}$

$(\text{SP}) - 1 \rightarrow \text{SP}$

$(\text{XPC}) \rightarrow \text{TOS}$

$(\text{pmad}(15-0)) \rightarrow \text{PC}$

$(\text{pmad}(22-16)) \rightarrow \text{XPC}$

延时

$(\text{SP}) - 1 \rightarrow \text{SP}$

$(\text{PC}) + 4 \rightarrow \text{TOS}$

$(\text{SP}) - 1 \rightarrow \text{SP}$

$(\text{XPC}) \rightarrow \text{TOS}$

$(\text{pmad}(15-0)) \rightarrow \text{PC}$

$(\text{pmad}(22-16)) \rightarrow \text{XPC}$

状态位:无。

说明:

程序指针 PC 指向由 pmad 的位 22~16 所确定的页中 pmad 的位 15~0 所确定的程序存储器地址。返回地址在 pmad 前被压入堆栈。如果是延迟调用,紧接着调用指令的 2 条单字指令或 1 条双字指令从程序存储串中取出先执行。注意:该指令不能循环执行。

例:FCALL 013333h

	执行前		执行后
PC	0025	PC	3333
XPC	00	XPC	01
SP	1111	SP	110F
数据存储器			
1110h	4567	1110h	0027
110Fh	4567	110Fh	0000

例:FCALLD 301000h

ANDM #4444h *AR1+

执行前		执行后	
PC	3001	PC	1000
XPC	7F	XPC	30
SP	1111	SP	110F
数据存储器			
1110h	4567	1110h	3005
110Fh	4567	110Fh	007F

37. FIRS Xmem,Ymem,pmad

操作数:

Xmem,Ymem:双数据存储操作数

$0 \leq pmad \leq 65535$

执行:

$pmad \rightarrow PAR$

$While(RC) \neq 0$

$(B) + A(32-16) \times (\text{由 } PAR \text{ 中的内容所决定的 } Pmem \text{ 的地址单元}) \rightarrow B$

$((Xmem) + (Ymem)) \ll 16 \rightarrow A$

$(PAR) + 1 \rightarrow PAR$

$(RC) - 1 \rightarrow RC$

状态位:

被 SXM 位、FRCT 位和 OVM 位影响。

影响 C 位、OVA 位和 OVB 位。

说明:

该指令实现一个对称的有限冲激响应滤波器。累加器 A 的高端(位 32~16)和由 pmad (存放在程序地址寄存器 PAR 中)寻址的一 pmem 值相乘,结果加到累加器 B 中,同时,存储器操作数 Xmem 和 Ymem 相加,结果左移 16 位,然后装入到累加器 A 中。在下一个循环中, pmad 加 1。一旦循环流水开始,该指令就成为单周期指令。

例:FIRS *AR3+, *AR4+,COEFS

执行前		执行后	
A	00 0077 0000	A	00 00FF 0000
B	00 0000 0000	B	00 0008 762C
FRCT	0	FRCT	0
AR3	0100	AR3	0101
AR4	0200	AR4	0201
数据存储器			
0100h	0055	0100h	0055

0200h	<div></div>	00AA	0200h	<div></div>	00AA
程序存储器					
COEFFS	<div></div>	1234	COEFFS	<div></div>	1234

38. FRAME K

操作数:

$-128 \leq K \leq 127$

执行: $(SP) - K \rightarrow SP$

状态位: 无。

说明:

把一个短立即数偏移 K 加到 SP 中。在编译方式 (CPL=1) 下的地址产生或紧接着该指令的下一条指令做堆栈处理都不会产生等待。

例: FRAME 10h

	执行前		执行后
SP	<div>1000</div>	SP	<div>1010</div>

39. FRET[D]

操作数: 无

执行:

$(TOS) \rightarrow XPC$

$(SP) + 1 \rightarrow SP$

$(TOS) \rightarrow PC$

$(SP) + 1 \rightarrow SP$

状态位: 无。

说明:

把堆栈单元的低 7 位值装入 XPC 中; 把下一个单元的 16 位值装入 PC 中, 堆栈指针在每一操作数完成后自动加 1。如果是延迟返回, 紧接着该指令的 2 条单字指令或 1 条双字指令取出先执行。注意: 该指令不能循环执行。

例: FRET

	执行前		执行后
PC	<div>2112</div>	PC	<div>1000</div>
XPC	<div>01</div>	XPC	<div>05</div>
SP	<div>0300</div>	SP	<div>0302</div>
数据存储器			
0300h	<div>0005</div>	0300h	<div>0005</div>
0301h	<div>1000</div>	0301h	<div>1000</div>

40. FRETE[D]

操作数:无。

执行:

(TOS)→XPC

(SP)+1→SP

(TOS)→PC

(SP)+1→SP

0→INTM

状态位:

影响 INTM 位。

说明:

把栈顶单元的低 7 位值装入 XPC 中;把下一个单元的 16 位值装入 PC,并且从新的 PC 值指向的单元继续执行。该指令自动清除 ST1 中的中断屏蔽位(INTM)(清除该位就允许中断)。如果是延迟返回,紧接着该指令的两条单字指令或一条双字指令取出先执行。注意:该指令不能循环执行。

例:FRETE

	执行前		执行后
PC	2112	PC	0110
XPC	05	XPC	6E
ST1	xCxx	ST1	x4xx
SP	0300	SP	0302
数据存储器			
0300h	006E	0300h	006E
0301h	0110		0110

41. IDLE K

操作数:

$1 \leq K \leq 3$

If K=1 NN=00

K=2 NN=10

K=3 NN=10

执行:

(PC)+1→PC

状态位:

被 INTM 位影响。

说明:

强迫程序执行等待操作直至产生非屏蔽中断或复位操作。PC 执行加 1 操作, DSP 芯片保持空闲状态(下拉方式)直到被中断。即使是 $INTM=1$, 只要有一个非屏蔽中断出现, 系统就退出空闲状态。如果 $INTM=1$, 程序继续执行紧接着 IDLE 的指令; 如果 $INTM=0$, 程序转移到相应的中断服务子程序。中断通过中断屏蔽寄存器(IMR)使用, 而不管 $INTM$ 的值。 K 的值所指定的下列选项决定了能让系统从空闲状态中解放出来的中断类型。

$K=1$ 诸如定时器和串口等外围设备在 IDLE 状态时仍有效。外围中断与复位操作以及外部中断一样能使处理器从空闲方式中解放出来。

$K=2$ 诸如定时器和串口等外围设备在空闲状态时无效。复位和外部中断能使处理器从空闲方式中解放出来。因为在正常的设备操作条件下, 中断在空闲方式中没被锁定, 所以它必须持续保持低脉冲几个周期才被响应。

$K=3$ 诸如定时器和串口等外围设备在 IDLE 状态时无效。锁相环 PLL 被禁止。

注意: 该指令不能循环执行。

例: IDLE1

例: IDLE2

例: IDLE3

42. INTR K

操作数:

$1 \leq K \leq 31$

执行:

$(SP) - 1 \rightarrow SP$

$(PC) + 1 \rightarrow TOS$

状态位:

影响 $INTM$ 位和 IFR 位。

说明:

让程序指针指向 K 所确定的中断向量。指令允许使用用户自己的应用软件执行任何中断服务子程序。在指令开始执行时, PC 加 1 并且把它压入栈顶, 然后把 K 指定中断向量装入 PC, 执行该中断服务子程序。对中断标志寄存器(IFR)中相应位清 0, 对应中断就被禁止(当 $INTM=1$ 时)。中断屏蔽寄存器(IMR)不会影响 INTR 指令, 且不管 $INTM$ 的值是什么都能执行 INTR 指令。注意: 该指令不能循环执行。

例: INTR 3

	执行前		执行后
PC	0025	PC	FFBC
INTM	0	INTM	1
IPTR	01FF	IPTR	01FF
SP	1000	SP	0FFF
数据存储器			
0FFFh	9653	0FFFh	0026

43. LD

- (1) LD Smem, dst
- (2) LD Smem, TS, dst
- (3) LD Smem, 16, dst
- (4) LD Smem, [, SHIFT], dst
- (5) LD Xmem, SHFT, dst
- (6) LD #K, dst
- (7) LD #1k[, SHFT], dst
- (8) LD #1k, 16, dst
- (9) LD src, ADM[, dst]
- (10) LD src[, SHIFT], dst

操作数:

Smem: 单数据存储操作数

Xmem: 双数据存储操作数

src, dst: A(累加器 A), B(累加器 B)

$0 \leq K \leq 255$

$-32\,768 \leq 1k \leq 32\,767$

$-16 \leq \text{SHIFT} \leq 15$

$0 \leq \text{SHFT} \leq 15$

执行:

- (1) (Smem) \rightarrow dst
- (2) (Smem) \ll TS \rightarrow dst
- (3) (Smem) \ll 16 \rightarrow dst
- (4) (Smem) \ll SHIFT \rightarrow dst
- (5) (Xmem) \ll SHFT \rightarrow dst
- (6) K \rightarrow dst
- (7) 1k \ll SHFT \rightarrow dst
- (8) 1k \ll 16 \rightarrow dst
- (9) (src) \ll ASM \rightarrow dst
- (10) (src) \ll SHIFT \rightarrow dst

状态位:

在累加器载入时被 SXM 位影响。

在带移位操作时被 OVM 位影响, 影响 Ovdst 位(如果 dst=src, 影响 Ovsrsrc 位)。

说明:

把一个数据存储器值或一个立即数装入累加器, 并且支持各种不同的移位(没有 dst 时为 src), 另外, 指令支持带移位的累加器到累加器的搬移。

例: LD *AR1, AA

执行前			
A	00	0000	0000
SXM	0		
ARI	0200		
数据存储器			
0200h	FEDC		

例:LD *AR1,TS,B

执行前			
B	00	0000	0000
SXM	1		
ARI	0200		
T	8		
数据存储器			
0200h	FEDC		

例:LD A,8,B

执行前			
A	00	7FFE	0040
B	00	0000	FFFF
OVb	0		
SXM	1		
数据存储器			
0200h	FEDC		

执行后			
A	00	0000	FEDC
SXM	0		
ARI	0200		
数据存储器			
0200h	FEDC		

执行后			
B	FF	FFFE	DC00
SXM	1		
ARI	0200		
T	8		
数据存储器			
0200h	FEDC		

执行后			
A	00	7FFE	0040
B	7F	FD00	4000
OVb	1		
SXM	1		
数据存储器			
0200h	FEDC		

44. LD

- (1) LD Smem, T
- (2) LD Smem, DP
- (3) LD #k9, DP
- (4) LD #k5, ASM
- (5) LD #k3, ARP
- (6) LD Smem, ASM

操作数:

Smem:单数据存储器操作数

$0 \leq k9 \leq 511$

$-16 \leq k5 \leq 15$

$0 \leq k3 \leq 7$

执行:

(Smem) → T

(Smem(8-0)) → DP

k9 → DP

k5→ASM

k3→ARP

(Smem(4-0)→ASM

状态位:无。

说明:

把一个数装入 T 寄存器或状态寄存器 ST0 或 ST1 中的 DP、ASM 和 ARP 域。装入的数可以是一个单数据存储器操作数 Smem 或一个常数。该指令代码为 1 个字,但当 Smem 采用了长偏移直接寻址或绝对地址寻址方式时,指令代码为 2 个字。

例:LD *AR4,DP

执行前		执行后	
AR4	0200	AR4	0200
DP	1FF	DP	0C
数据存储器			
0200h	FEDC	0200h	FEDC

例:LD3,ARP

执行前		执行后	
ARP	0	ARP	3

45. LDM MMR,dst

操作数:

MMR:存储器映射寄存器

dst:A(累加器),B(累加器)

执行:

MMR→dst(15-0)

00 0000h→dst(39-16)

状态位:无。

说明:

把存储器映射寄存器 MMR 中的值装入到目的累加器中,不管 DP 的当前内容或 ARX 的高 9 位的值是多少,都把有效地址的高 9 位清 0,以指定为在数据页 0 中。该指令不受 SXM 的影响。

例:LDM AR4,A

执行前		执行后	
A	00 0000 1111	A	00 0000 FFFF
AR4	FFFF	AR4	FFFF

46. LD Xmem,dst||MAS[R] Ymem[,dst_]

操作数:

Xmem, Ymem: 双数据存储操作数

dst: A(累加器), B(累加器)

dst_: 如果 dst = A, 则 dst_ = B; 如果 dst = B, 则 dst_ = A

执行:

$(Xmem) \ll 16 \rightarrow dst(31-16)$

if(Rounding)

$Round((dst_) - ((T) \times (Ymem))) \rightarrow dst_$

Else

$(dst_) - ((T) \times (Ymem)) \rightarrow dst_$

状态位:

被 SXM 位、FRCT 位和 OVM 位影响。

影响 Ovdst_ 位。

说明:

16 位双数据存储操作数 Xmem 左移 16 位后装入目的累加器 dst_ 的高端(位 31~16)。同时并行执行一个双数据操作数 Ymem 与 T 寄存器的值相乘, 再把乘积与 dst_ 中的数据相减, 最后把结果存放在 dst_ 的操作。如果使用了 R 后缀, 指令会对乘法和减法运算的结果进行凑整, 即结果加上 2^{15} , 然后对低端(位 15~0)清 0, 再把新的结果存放在 dst_ 中。

例: LD *AR4+, A || MAS * AR5+, B

执行前		
A	00 0000 1000	
B	00 0000 1111	
T		0400
FRCT		0
AR4		0100
AR5		0200
数据存储器		
0100h		1234
0200h		4321

执行后		
A	00 1234 0000	
B	00 010D 0000	
T		0400
FRCT		0
AR4		0101
AR5		0201
数据存储器		
0100h		1234
0200h		4321

47. LDR Smem, dst

操作数:

Smem: 单数据存储操作数

dst: A(累加器 A), B(累加器 B)

执行:

$(Smem) \ll 16 + 1 \ll 15 \rightarrow dst(31-16)$

状态位:

被 SXM 位影响。

说明:

把单数据存储操作数 Smem 左移 16 位后装入目的累加器 dst 的高端(位 31~16)。

Smem 通过加 2^{15} 再对累加器的低端(位 15~0)清 0 来凑整。累加器的位 15 置为 1。

例: LDR *AR1, A

执行前		执行后	
A	00 0000 0000	A	00 FEDC 8000
SXM	0	SXM	0
AR1	0200	AR1	0200
数据存储器		数据存储器	
0200h	FEDC	0200h	FEDC

48. LD Smem, dst

操作数:

Smem: 单数据存储操作数

dst: A(累加器 A), B(累加器 B)

执行:

$(\text{Smem}) \rightarrow \text{dst}(15-0)$

$000000\text{h} \rightarrow \text{dst}(39-16)$

状态位: 无。

说明:

把单数据存储值 Smem 装入目的累加器 dst 的低端(位 15~0)。dst 的保护位和高端(位 39~16)清 0。因此, 数据被看成是一个不带符号的 16 位数。不管 SXM 位的状态如何都无符号扩展。该指令代码占 1 个字, 但当 Smem 采用长偏移间接寻址或绝对地址寻址方式时就多占 1 个字。

例: LDU *AR1, A

执行前		执行后	
A	00 0000 0000	A	00 0000 0000
AR1	0200	AR1	0200
数据存储器		数据存储器	
0200h	FEDC	0200h	FEDC

49. LMS Xmem, Ymem

操作数:

Xmem, Ymem: 双数据存储操作数

执行:

$(A) + (Xmem) \ll 16 + 2.15 \rightarrow A$

$(B) + (Xmem) \times (Ymem) \rightarrow B$

状态位:

被 SXM 位、FRCT 位和 OVM 位影响。

影响 C 位、OVA 位和 OVB 位。

说明:

该指令是求最小均方值算法。双数据存储器操作数 Xmem 左移动 16 位后再加上 2^{15} , 其结果加到累加器 A 中, 同时并行执行 Xmem 与 Ymem 相乘, 其结果加到累加器 B 中的操作。Xmem 不会冲掉 T 寄存器的值, 因而 T 寄存器总包着用于修改系数的错误值。

例: LMS * AR3+, * AR4+

执行前			执行后		
A	00	7777 8888	A	00	77CD 0888
B	00	0000 0100	B	00	0000 3972
FRCT		0	FRCT		0
AR3		0100	AR3		0101
AR4		0200	AR4		0201
数据存储器					
0100h		0055	0100h		0055
0200h		0200	0200h		0201

50. LTD Smem

操作数:

Smem: 单数据存储器操作数

执行:

(Smem) → T

(Smem) → Smem + 1

状态位: 无。

说明:

把一个单数据存储器单元的内容 Smem 复制到 T 寄存器和紧接着的数据单元中去, 当数据复制完毕后, Smem 单元的内容保持不变。这个功能在数字信号处理应用中实现一个 Z 延迟是相当有用的。该功能在存储器延迟指令中也存在。

例: LTD * AR3

执行前			执行后		
T		0000	T		6CAC
AR3		0100	AR3		0100
数据存储器					
0100h		6CAC	0100h		6CAC
0101h		xxxx	0101h		6CAC

51. MAC

语法:

(1) MAC[R] Smem, src

(2) MAC[R] Xmem, Ymem src [,dst]

(3) MAC 1k,src[,dst]

(4) MAC Smem, #1k,src[,dst]

操作数:

Smem:单数据存储操作数

Xmem, Ymem:双数据存储操作数

src,dst:A(累加器 A),B(累加器 B)

$-32\,768 \leq 1k \leq 32\,767$

执行:

(1) (Smem) \times (T) + (src) \rightarrow src

(2) (Xmem) \times (Ymem) + (src) \rightarrow dst, (Xmem) \rightarrow T

(3) (T) \times 1k + (src) \rightarrow dst

(4) (Smem) \times 1k + (src) \rightarrow dst, (Smem) \rightarrow T

状态位:

被 FRCT 位和 OVM 位影响。

影响 Ovdst 位。

说明:

该指令是完成乘和累加运算,并可凑整。结果按规定存放在 dst 或 src 中。对于第(2)和第(4)种语法,紧接着操作码的数据寄存器的值在读操作数阶段放到 T 寄存器中。如果使用 R 后缀,指令对乘/累加操作的结果凑整。

例:MAC * AR5+,A

执行前	
A	00 0000 1000
T	0400
FRCT	0
AR5	0100
数据存储器	
0100h	1234

执行后	
A	00 0048 E000
T	0400
FRCT	0
AR5	0101
	1234

例:MAC #345h,A,B

执行前	
A	00 0000 0000
B	00 0000 0000
T	0400
FRCT	1

执行后	
A	00 0000 0000
B	00 001A 3800
T	0400
FRCT	1

例:MAC * AR5+,AR6+,A,B

执行前	
A	00 0000 1000
B	00 0000 0004
T	0008

执行后	
A	00 0000 1000
B	00 0CAC 10C0
T	5678

FRCT	1
AR5	0100
AR6	0200
数据存储器	
0100h	5678
0200h	1234

例: MACR * AR5+, A

执行前	
A	00 0000 1000
T	0400
FRCT	0
AR5	0100
数据存储器	
0100h	1234

例: MAC * AR5+, AR6+, A, B

执行前	
A	00 0000 1000
B	00 0000 0004
T	0008
FRCT	1
AR5	0100
AR6	0200
数据存储器	
0100h	5678
0200h	1234

FRCT	1
AR5	0101
AR6	0201
0100h	5678
0200h	1234

执行后	
A	00 0049 0000
T	0400
FRCT	0
AR5	0101
0100h	1234

执行后	
A	00 0000 1000
B	00 0CAC 0000
T	5678
FRCT	1
AR5	0101
AR6	0201
0100h	5678
0200h	1234

52. MACA

语法:

(1) MACA[R] Smem[, B]

(2) MACA[A] T, src[, dst]

操作数:

Smem: 单数据存储操作数

src, dst: A(累加器 A), B(累加器 B)

执行:

(1) $(\text{Smem}) \times (A(32-16)) + (B) \rightarrow B$

(2) $(T) \times (A(32-16)) + (\text{src}) \rightarrow \text{dst}$

状态位:

被 FRCT 位和 OVM 位影响。

影响 Ovdst 位。

说明:

累加器 A 的高端(位 32~16)与一个单数据存储器操作数 Smem 或 T 寄存器中的内容相乘,乘积加到累加器 B 中(语法 1)或源累加器 src 中。语法 1 结果存放在累加器 B 中,语法 2 结果存放在 dst 或 src(如果没有定义 dst)中。累加器 A 的(位 32~16)值用来作为乘法器的一个 17 位操作数。如果使用了 R 后缀,指令要结果进行凑整。该指令代码占 1 个字,但当 Smem 采用长偏移间接寻址或绝对寻址方式时,就会多占 1 个字。

例:MACA * AR5+

执行前				执行后			
A	00	1234	0000	A	00	1234	0000
B	00	0000	0000	B	00	0626	0060
T			0400	T			5678
FRCT			0	FRCT			0
AR5			0100	AR5			0101
数据存储器							
0100h			5678	0100h			5678

例:MACA T,B,B

执行前				执行后			
A	00	1234	0000	A	00	1234	0000
B	00	0002	0000	B	00	009D	4BA0
T			0444	T			0444
FRCT			1	FRCT			1

53. MACD Smem,pmad,src

操作数:

Smem:单数据存储器操作数

src:A(累加器 A),B(累加器 B)

$0 \leq pmad \leq 65\ 535$

执行:

pmad→PAR

If(RC)≠0

Then

(Smem)×(由 PAR 中的内容所决定的 Pmem 的地址单元)+(src)→src

(Smem)→T

(Smem)→Smem+1

(PAR)+1→PAR

Else

(Smem)×(由 PAR 中的内容所决定的 Pmem 的地址单元)+(src)→src

(Smem)→T

$(Smem) \rightarrow Smem + 1$

状态位:

被 FRCT 位和 OVM 位影响。

影响 Ovsrc 位。

说明:

一个单数据存储器值 Smem 与一个程序存储器值 pmad 相乘,乘积和源累加器 src 的值相加,结果存放在 src 中。另外,还把数据存储器值 Smem 装入到 T 寄存器和紧接着 Smem 地址的数据单元中去。当循环执行该指令,在程序地址寄存器 PAR 中的程序存储器地址执行加 1 操作。循环流水一旦启动,指令就变成单周期指令。在存储器延迟指令中也存在该功能。该指令代码占 2 个字,但当 Smem 采用长偏移间接寻址或绝对寻址方式时,就会多占 1 个字。

例:MACD * AR3-,COEFFS,A

执行前			执行后		
A	00	0077 0000	A	00	007D 0B44
T		0008	T		0055
FRCT		0	FRCT		0
AR3		0100	AR3		00FF
程序存储器					
COEFFS		1234	COEFFS		1234
数据存储器					
0100h		0055	0100h		0055
0101h		0066	0101h		0066

54. MACP Smem,pmad,src

操作数:

Smem:单数据存储器操作数

src:A(累加器 A),B(累加器 B)

$0 \leq pmad \leq 65535$

执行:

$pmad \rightarrow PAR$

If(RC) $\neq 0$

Then

$(Smem) \times (\text{由 PAR 中的内容所决定的 Pmem 的地址单元}) + (src) \rightarrow src$

$(Smem) \rightarrow T$

$(Smem) \rightarrow Smem + 1$

$(PAR) + 1 \rightarrow PAR$

Else

$(Smem) \times (\text{由 PAR 中的内容所决定的 Pmem 的地址单元}) + (src) \rightarrow src$

$(Smem) \rightarrow T$

$(Smem) \rightarrow Smem + 1$

状态位:

被 FRCT 位和 OVM 位影响。

影响 Ovsrsrc 位。

说明:

一个单数据存储器值 Smem 与一个程序存储器值 pmad 相乘,乘积和源累加器 src 的值相加,结果存放在 src 中。同时,把数据存储器值 Smem 复制到 T 寄存器中。当循环执行该指令,在程序地址寄存器 PAR 中的程序存储器地址执行加 1 操作。一旦循环流水启动,指令就变成了单周期指令。该指令代码占 2 个字,但当 Smem 采用长偏移间接寻址或绝对寻址方式时,就会多占 1 个字。

例:MACP * AR3-,COEFFS,A

执行前			执行后		
A	00	0077 0000	A	00	007D 0B44
T		0008	T		0055
FRCT		0	FRCT		0
AR3		0100	AR3		00FF
程序存储器					
COEFFS		1234	COEFFS		1234
数据存储器					
0100h		0055	0100h		0055
0101h		0066	0101h		0066

55. MACSU Xmem,Ymem,src

操作数:

Smem:单数据存储器操作数

src:A(累加器 A),B(累加器 B)

执行:

$\text{unsigned}(\text{Xmem}) \times \text{signed}(\text{Ymem}) + (\text{src}) \rightarrow \text{src}$

$(\text{Xmem}) \rightarrow \text{T}$

状态位:

被 FRCT 位和 OVM 位影响。

影响 Ovsrsrc 位。

说明:

一个不带符号的数据存储器 Xmem 与一个不带符号的数据存储器值 Ymem 相乘,乘积与源累加器 src 的值相加,结果放在 src 中。同时,在读操作数阶段把这个 16 位不带符号的数 Xmem 存放 T 寄存器中。由 Xmem 寻址的数据从 D 总线上获得,由 Ymem 寻址的数据从 C 总线上获得。

例:MACSU * AR4+,AR5+,A

执行前				执行后			
A	00	0000	1000	A	00	09A0	AA84
B			0008	B			8765
FRCT			0	FRCT			0
AR4			0100	AR4			0101
AR5			0200	AR5			0201
数据存储器							
0100h			8765	0100h			8765
0200h			1234	0200h			1234

56. MAR Smem

操作数:

Smem:单数据存储操作数

执行:

If(兼容方式打开)(CMPT=1)

then:

If(ARx=AR0)Ar(ARP)被修改

ARP 不被修改

Else

ARx 被修改

X → ARP

Else

(兼容方式关闭)(CMPT=0)

ARx 被修改

ARP 不被修改

状态位:

被 CMPT 位影响。

如果 CMPT=1,影响 ARP 位。

说明:

修改由 Smem 所确定的辅助寄存器的内容。在兼容方式下(CMPT=1),指令会修改 ARx 的内容以及辅助寄存器指针(ARP)的值;在非兼容方式下(CMPT=0),指令只修改辅助寄存器的值,而不改变 ARP。该指令代码占 1 个字,但当 Smem 采用长偏移间接寻址或绝对寻址方式时,就会多占 1 个字。

例:MAR * AR3+

执行前		执行后	
CMPT	1	CMPT	1
ARP	4	ARP	4
AR4	0100	AR4	00FF

例: MAR * AR0—

执行前	
CMPT	1
ARP	0
AR0	0008
AR3	0100

执行后	
CMPT	1
ARP	3
AR0	0008
AR3	0100

例: MAR * AR3

执行前	
CMPT	1
ARP	0
AR0	0008
AR3	0100

执行后	
CMPT	1
ARP	3
AR0	0008
AR3	0100

例: MAR * +AR3

执行前	
CMPT	1
ARP	0
AR3	0100

执行后	
CMPT	1
ARP	3
AR3	0101

57. MAS

(1) MAS[R]Smem,src

(2) MAS[R]Xmem,Ymem,src[,dst]

操作数:

Smem:单数据存储操作数

Xmem,Ymem:双数据存储操作数

src,dst:A(累加器 A),B(累加器 B)

执行:

(1) (src) ← (Smem) × (T) → src

(2) (src) ← (Xmem) × (Ymem) → dst

(Xmem) → T

状态位:

被 FRCT 位和 OVM 位影响。

影响 Ovdst 位(如果 dst=src,影响 Ovsrce 位)。

说明:

一个存储器操作数与 T 寄存器的内容相乘,或者是两个存储器操作数相乘,再从源累加器 src 或目的累加器 dst 中减去该乘积,结果存放在 src 或 dst 中。Xmem 在读操作数阶段装入到 T 寄存器中。如果用了 R 后缀,指令就会对结果进行凑整运算。

例: MAS * AR5+,A

执行前			
A	00	0000	1000
T			0400
FRCT			0
AR5			0100
数据存储器			
0100h			1234

例: MAS * AR5+, * AR6+, A, B

执行前			
A	00	0000	1000
B	00	0000	0004
T			0008
FRCT			1
AR5			0100
AR6			0200
数据存储器			
0100h			5678
0200h			1234

例: MASR * AR5+, A

执行前			
A	00	0000	1000
T			0400
FRCT			0
AR5			0100
数据存储器			
0100h			1234

例: MASR * AR5+, * AR6+, A, B

执行前			
A	00	0000	1000
B	00	0000	0004
T			0008
FRCT			1
AR5			0100
AR6			0200
数据存储器			
0100h			5678
0200h			1234

执行后			
A	FF	FFB7	4000
T			0400
FRCT			0
AR5			0101
数据存储器			
0100h			1234

执行后			
A	00	0000	1000
B	FF	F9DA	0FA0
T			5678
FRCT			1
AR5			0101
AR6			0201
数据存储器			
0100h			5678
0200h			1234

执行后			
A	FF	FFB7	0000
T			0400
FRCT			0
AR5			0101
数据存储器			
0100h			1234

执行后			
A	00	0000	1000
B	FF	F9DA	0000
T			5678
FRCT			1
AR5			0101
AR6			0201
数据存储器			
0100h			5678
0200h			1234

58. MASA

(1) MASA Smem[, B]

(2) MASA[R] T,src[,dst]

操作数:

Smem:单数据存储器操作数

src,dst:A(累加器 A),B(累加器 B)

执行:

(1)(B) ← (Smem) × (A(32-16)) → B;

(Smem) → T

(2)(src) ← (T) × (A(32-16)) → dst

状态位:

被 FRCT 位和 OVM 位影响。

影响 Ovdst 位。

说明:

累加器 A 的高端(位 32~16)与一个单数据存储器操作数 Smem 或 T 寄存器中的内容相乘,再从累加器 B(语法(1))或源累加器 src 中减去该乘积,结果存放在累加器 B 中(语法(1))或 dst 或 src(没有定义 dst 时)中。在读操作数阶段,把 Smem 装入 T 寄存器。如果在语法(2)中使用了 R 后缀,指令对运算结果就会进行凑整运算。该指令代码占 1 个字,但当 Smem 采用长偏移间接寻址或绝对寻址方式时,就会多占 1 个字。

例:MASA * AR5+

执行前				执行后			
A	00	1234	0000	A	00	1234	0000
B	00	0002	0000	B	FF	F9DB	FFA0
T			0400	T			5678
FRCT			0	FRCT			0
AR5			0100	AR5			0101
数据存储器							
0100h			5678	0100h			5678

例:MASA T,B

执行前				执行后			
A	00	1234	0000	A	00	1234	0000
B	00	0002	0000	B	FF	FF66	B460
T			0444	T			0444
FRCT			1	FRCT			1

例:MASAR T,B

执行前				执行后			
A	00	1234	0000	A	00	1234	0000
B	00	0002	0000	B	FF	FF67	0000
T			0444	T			0444
FRCT			1	FRCT			1

59. MAX dst

操作数:

dst: A(累加器 A), B(累加器 B)

执行:

If(A > B)

Then

(A) → dst

0 → C

Else

(B) → dst

1 → C

状态位:

影响 C 位。

说明:

比较两累加器的内容, 并把较大的一个值存放在目的累加器 dst 中。如果最大值是在累加器 A 中, 进位位 C 被清 0; 否则置为 1。

例: MAX A

执行前			
A	00	0000	0055
B	00	0000	1234
C			
			0

执行后			
A	00	0000	1234
B	00	0000	1234
C			
			0

60. MIN dst

操作数:

dst: A(累加器 A), B(累加器 B)

执行:

If(A < B)

Then(A) → dst

0 → C

Else(B) → dst

1 → C

状态位:

影响 C 位。

说明:

比较两累加器值的大小, 把较小值存放在目的累加器 dst 中。如果较小值为累加器 A, 进位位 C 被清 0; 否则置为 1。

例:MAN A

执行前			执行后		
A		FFCB	- 53	A	FFCB
B		FFF6	- 10	B	FFF6
C		1		C	0

61. MPY

语法:

- (1) MPY[R]Smem,dst
- (2) MPY Xmem,Ymem,dst
- (3) MPY Smem,#1k,dst
- (4) MPY #1k,dst

操作数:

Smem:单数据存储操作数

Xmem,Ymem:双数据存储操作数

dst:A(累加器 A),B(累加器 B)

$-32\,768 \leq 1k \leq 32\,767$

执行:

- (1) $(T) \times (Smem) \rightarrow dst$
- (2) $(Xmem) \times (Ymem) \rightarrow dst$
 $(Xmem) \rightarrow T$
- (3) $(Smem) \times 1k \rightarrow dst$
 $(Smem) \rightarrow T$
- (4) $(T) \times 1k \rightarrow dst$

状态位:

被 FRCT 位和 OVM 位影响。

影响 Ovdst 位。

说明:

T 寄存器的值或者一个数据存储器值与另一个数据存储器值或者一个立即数相乘,结果存放在目的累加器 dst 中。在读操作数阶段,把 Smem 或 Xmem 的值装入 T 寄存器中。如果使用了 R 后缀,指令就会对结果进行凑整运算。其步骤是:结果加上 2^{15} ,再对位 15~0 清 0。

例:MPY 13,A

执行前			执行后		
A	00	0000 0036	A	00	0000 0054
T		0006	T		0006
FRCT		1	FRCT		1
DP		008	DP		008
数据存储器					
040Dh		0007	040Dh		0007

例:MPY * AR2-, * AR4+0%,B

执行前		
B	FF	FFFF FFE0
FRCT	0	
AR0	0001	
AR2	01FF	
AR4	0300	
数据存储器		
01FFh	0010	
0300h	0002	

执行后			
B	00	0000	0020
FRCT	0		
AR0	0001		
AR2	01FE		
AR4	0301		
01FFh	0010		
0300h	0002		

例:MPY # OFFFEh,A

执行前			
A	00	0000	1234
T	2000		
FRCT	0		

执行后			
A	FF	FFFF	C000
T	2000		
FRCT	0		

例:MPYR 0,B

执行前		
B	FF	FE00 0001
T	1234	
FRCT	0	
DP	004	
数据存储器		
0200h	5678	

执行后			
B	00	0626	0000
T	1234		
FRCT	0		
DP	004		
0200h	5678		

62. MPYA

语法:

(1) MPYA Smem

(2) MPYA dst

操作数:

Smem:单数据存储器操作数

dst:A(累加器 A),B(累加器 B)

执行:

(1) (Smem) × (A(32-16)) → B

(Smem) → T

(2) (T) × (A(32-16)) → dst

状态位:

被 FRCT 位和 OVM 位影响。

影响 Ovsrsrc 位。

说明:

累加器 A 的高端(位 32~16)与一个单数据存储器操作数 Smem 或 T 寄存器的值相乘,

结果存放在累加器 dst 或累加器 B 中。在读操作数期间,把单数据存储器操作数 Smem 装入 T 寄存器中(语法(1))。该指令代码占 1 个字,但当 Smem 采用长偏移间接寻址或绝对寻址方式时,就会多占 1 个字。

例:MPYA *AR2

执行前				执行后			
A	FF	8765	1111	A	FF	8765	1111
B	00	0000	0320	B	FF	D743	6558
T			1234	T			5678
FRCT			0	FRCT			0
AR2			0200	AR2			0200
数据存储器							
0200h			5678	0200h			5678

63. MPYU Smem dst

操作数:

Smem:单数据存储器操作数

dst:A(累加器 A),B(累加器 B)

执行:unsigned(T)×unsigned(Smem)→dst

状态位:

被 FRCT 位和 OVM 位影响。

影响 Ovdst 位。

说明:

不带符号的 T 寄存器值与不带符号的单数据存储器操作数 Smem 相乘,结果存放在目的累加器 dst 中。乘法器对于该指令来说,相当于是其两个操作数的最高位都为 0 的一个带符号的 17×17 位的乘法器。该指令在计算诸如两个 32 位数相乘得到一个 64 位乘积的多精度乘法时相当有用。该指令代码占 1 个字,但当 Smem 采用长偏移间接寻址或绝对寻址方式时,就会多占 1 个字。

例:MPYU *AR0—,A

执行前				执行后			
A	FF	8000	0000	A	00	3F80	0000
T			4000	T			4000
FRCT			0	FRCT			0
AR0			1000	AR0			0FFF
数据存储器							
1000h			FE00	1000h			FE00

64. MVDD Xmem,Ymem

操作数:

Xmem, Ymem: 双数据存储操作数

执行:

(Xmem) → Ymem

状态位: 无。

说明:

把通过 Xmem 寻址的数据存储器单元的值, 复制到通过 Ymem 寻址的数据存储器单元中去。

例: MVDD * AR3 ←, * AR5 →

执行前		执行后	
AR3	8000	AR3	8001
AR5	0200	AR5	0201
数据存储器			
0200h	ABCD	0200h	1234
8000h	1234		1234

65. MVDK Smem, pmad

操作数:

Smem: 单数据存储操作数

$0 \leq \text{pmad} \leq 65\,535$

执行:

pmad → EAR

If(RC) → 0

Then

(Smem) × (由 EAR 中的内容所决定的 Dmem 的地址单元)

(EAR) + 1 → EAR

Else

(Smem) × (由 EAR 中的内容所决定的 Dmem 的地址单元)

状态位: 无。

说明:

把一个单数据存储操作数 Smem 的内容, 复制到一个通过 dmad(地址在 EAB 地址寄存器 EAR 中)寻址的数据存储器单元。可以循环执行该指令来转移数据存储器中的连续字(使用间接寻址)。实际被转移的字数要比在指令开始执行时循环计数器中的值大 1。一旦启动了流水线, 指令就成为了单周期指令。该指令代码占 2 个字, 但当 Smem 采用长偏移间接寻址或绝对寻址方式时, 就会多占 1 个字。

例: MVDK * AR3 ←, 1000h

执行前		执行后	
AR3	01FF	AR3	01FE
数据存储器			
1000h	ABCD	1000h	1234
01FFh	1234	01FFh	1234

66. MVDM dmad, MMR

操作数:

MMR: 存储器映射寄存器

$0 \leq \text{pmad} \leq 65\,535$

执行:

$\text{dmad} \rightarrow \text{DAR}$

If(RC) \rightarrow 0

Then

(由 DAR 中的内容所决定的 Dmem 的地址单元) \rightarrow MMR

$(\text{DAR}) + 1 \rightarrow \text{DAR}$

Else

(由 EAR 中的内容所决定的 Dmem 的地址单元) \rightarrow MMR

状态位: 无。

说明:

把数据从一个数据存储器单元 dmad(dmad 的值装入 DAB 地址寄存器 DAR 中), 复制到一个存储器映射寄存器 MMR 中。一旦启动了循环流水线, 指令就变成了 1 条单周期指令。该指令代码占 2 个字。

例: MVDM 300h BK

执行前		执行后	
BK	ABCD	BK	1234
数据存储器			
0300h	1234	0300h	1234

67. MVDK Smem, pmad

操作数:

Smem: 单数据存储器操作数

$0 \leq \text{pmad} \leq 65\,535$

执行:

$\text{pmad} \rightarrow \text{PAR}$

If(RC) \neq 0

Then

(Smem)→(由 PAR 中的内容所决定的 Pmem 的地址单元)

(PAR)+1→PAR

Else

(Smem)→(由 PAR 中的内容所决定的 Pmem 的地址单元)

状态位:无。

说明:

把严格的 16 位单数据存储器操作数 Smem,复制到一个由 16 位立即数 pmad 寻址的程序存储器单元中,可以通过循环执行该指令,把数据存储器中的连续字(使用间接寻址)转移到由 16 位立即数寻址的连续的程序存储器空间。源和目的块不必全部在片外或片内。当循环流水开始进行时,该指令就变成了一个单周期指令。另外,当循环执行该指令时,中断被禁止。

例: MVDP 0,0FF00h

	执行前		执行后
DP	004	DP	004
数据存储器			
0200h	0123	0200h	0123
程序存储器			
FE00h	FFFF	FE00h	0123

68. MVKD dmad,Smem

操作数:

Smem:单数据存储器操作数

$0 \leq \text{dmad} \leq 65535$

执行:

dmad→DAR

If(RC)≠0

Then

(由 DAR 中的内容所决定的 Dmem 的地址单元)→(Smem)

(DAR)+1→DAR

Else

(由 DAR 中的内容所决定的 Dmem 的地址单元)→(Smem)

状态位:无。

说明:

把数据从一数据存储器单元转移到另一数据存储器单元中。源数据存储器单元由一个 16 位立即数 dmad 寻址,然后转移到 Smem 中。若以循环执行该指令来转移数据存储器中的连续字(使用间接寻址),转移的字数要比指令开始执行时循环计数器中的值多 1。一旦形成循环流水线,该指令就变成了单周期指令。该指令代码占 2 个字,但当 Smem 采用长偏移间接寻址或绝对寻址方式时,就会多占 1 个字。

例: MVKD 1000h, *+AR5

	执行前		执行后
AR5	01FF	AR5	0200
数据存储器			
0100h	1234	0100h	1234
0200h	ABCD	0200h	1234

69. MVKD MMR, dmad

操作数:

MMR: 存储器映射寄存器

$0 \leq \text{dmad} \leq 65\,535$

执行:

$\text{dmad} \rightarrow \text{EAR}$

$\text{If}(\text{RC}) \neq 0$

Then

$(\text{MMR}) \rightarrow (\text{由 EAR 中的内容所决定的 Dmem 的地址单元})$

$(\text{EAR}) + 1 \rightarrow \text{EAR}$

Else

$(\text{MMR}) \rightarrow (\text{由 DAR 中的内容所决定的 Dmem 的地址单元})$

状态位: 无。

说明:

把数据从一个存储器映射寄存器 MMR 转移到一个数据存储器中。目的数据存储器是通过一个 16 位立即数 dmad 寻址。一旦建立了循环流水线, 该指令就变成了单周期指令。

例: MVMD AR7, 8000h

	执行前		执行后
AR7	1234	AR7	1234
数据存储器			
8000h	ABCD	8000h	1234

70. MVMM MMRx, MMRy

操作数:

$\text{MMRx}, \text{AR0} - \text{AR7}, \text{SP}$

$\text{MMRy}, \text{AR0} - \text{AR7}, \text{SP}$

执行:

$\text{MMRx} \rightarrow \text{MMRy}$

状态位: 无。

说明:

把存储器映射寄存器 MMRx 中的内容转移到另一个存储器映射寄存器 MMRy 中。

MMRx 和 MMRy 只可能为 9 种操作数: AR0~AR7 和 SP。读 MMRx 的操作在译码阶段执行;写 MMRy 的操作在访问阶段执行。注意:该指令不能循环执行。

例: MVMM SP, AR1

	执行前		执行后
AR1	3EFF	AR1	0200
SP	0200	SP	0200

71. MVPD pmad, Smem

操作数:

Smem: 单数据存储器操作数

$0 \leq \text{pmad} \leq 65\,535$

执行:

pmad \rightarrow PAR

If(RC) \neq 0

Then

(由 PAR 中的内容所决定的 Pmem 的地址单元) \rightarrow (Smem)

(PAR) + 1 \rightarrow PAR

Else

(由 PAR 中的内容所决定的 Pmem 的地址单元) \rightarrow (Smem)

状态位: 无。

说明:

把一个字从通过 16 位立即数 pmad 寻址的程序存储器中转移到一个由 Smem 寻址的数据存储器单元。该指令能循环执行, 以把程序存储器中的连续字转移到连续的数据存储器单元去。源和目的块不必全部都在片内或片外。当建立了循环流水线, 该指令就变成了单周期指令。另外, 循环执行该指令时禁止中断。该指令代码占 2 个字, 但当 Smem 采用长偏移间接寻址或绝对寻址方式时, 就会多占 1 个字。

例: MVPD 2000h, * AR7-0

	执行前		执行后
AR0	0002	AR0	0002
AR7	0FFE	AR7	0FFC
程序存储器			
2000h	1234	2000h	1234
数据存储器			
OFFEh	ABCD	OFFEh	1234

72. NEG src[, dst]

操作数:

src,dst;A(累加器 A),B(累加器 B)

执行:

$(src) \times (-1) \rightarrow dst$

状态位:

被 FRCT 位和 OVM 位影响。

影响 Ovdst 位。

说明:

计算源累加器(可能是 A 或 B)值的二进制补码,并把结果存放在 dst 或 src 中。只要累加器值不为 0。指令就对进位位 C 清 0;反之,如果累加器值为 0,进位位置为 1。

如果累加器的值为 8000000000h,该运算将引起溢出,因为 8000000000h 的补码超过了累加器允许的最大值。因此,如果 OVM=1,目的累加器 dst 赋值为 007FFFFFFFh;如果 OVM=0,dst 赋值为 8000000000h。dst 的 OVM 位在这两种情况下都置位,以表明溢出。

例:NEG A

执行前				执行后			
A	80	0000	0000	A	80	0000	0000
OVA				OVA	1		
OVM	0			OVM	0		

例:NEG A

执行前				执行后			
A	80	0000	0000	A	00	7FFF	FFFF
OVA				OVA	1		
OVM	1			OVM	1		

73. NOP

操作数:无。

执行:无。

说明:

该指令除了 PC 执行加 1 操作以外,不执行任何操作,在建立流水线和执行延迟方面比较有用。

74. NORM src[,dst]

操作数:

src,dst;A(累加器 A),B(累加器 B)

执行:

$(src) \ll TS \rightarrow dst$

状态位:

被 SXM 位和 OVM 位影响。

影响 Ovdst 位(如果 dst=src,影响 Ovsrc 位)。

说明:

对源累加器 src 中的带符号数进行归一化,结果存放在 dst 或 src(如果没有明确指 dst)中。定点数的归一化是通过寻找符号扩展数的数量级把这个数分成数值部分和指数部分。该指令允许累加器的单周期归一化,指令 EXP 就是一例。移位数由 T 寄存器的位 5~0 确定,并编码成二进制补码的形式。有效的移位数是-16~31。执行过程中,在读操作数阶段移位器获得移位数(在 T 寄存器中),在执行阶段进行归一化操作。

例:NORM A

执行前			执行后		
A	FF	FFFF F001	A	FF	8008 0000
T		0013	T		0013

例:NORM B,A

执行前			执行后		
A	FF	FFFF F001	A	00	4214 1414
B	21	0A0A 0A0A	B	21	0A0A 0A0A
T		0FF9	T		0FF9

75. OR

语法:

- (1) OR Smem,src
- (2) OR #1k[,SHFT],src[,dst]
- (3) OR #1k,16,src[,dst]
- (4) OR src[,SHIFT],[,dst]

操作数:

src,dst:A(累加器 A),B(累加器 B)

Smem:单数据存储操作数

$0 \leq \text{SHIFT} \leq 15$

$-16 \leq \text{SHIFT} \leq 15$

$0 \leq 1k \leq 65\ 535$

执行:

- (1) (Smem)OR(src(15-0))→src,src(39-16)不被修改
- (2) $1k \ll \text{SHFTOR}(\text{src}) \rightarrow \text{dst}$
- (3) $1k \ll 16 \text{ OR}(\text{src}) \rightarrow \text{dst}$
- (4) $(\text{src or}[\text{dst}]) \text{OR}(\text{src}) \ll \text{SHIFT} \rightarrow \text{dst}$

状态位:无。

说明:

源累加器 src 和一个单数据存储操作数 Smem;一个左移后的 16 位立即数;dst 或它本身相“或”,结果可按指定的方式移位,并存放在 dst 中。如果没有指定 dst,就存放在 src 中。

对于正向移位(左移),低位添 0,高位不进行符号扩展;对于反向移位(右移),高位不进行符号扩展。

例:OR A,+3,B

执行前			执行后		
A	00	0000	200	A	00 0000 1200
B	00	0000	1800	B	00 0000 9800

76. ORM #1k,Smem

操作数:

Smem:单数据存储操作数

$0 \leq 1k \leq 65535$

执行:

$1k \text{ OR}(\text{Smem}) \rightarrow \text{Smem}$

状态位:无。

说明:

一个单数据存储操作数 Smem 与一个 16 位立即数相“或”,结果存放在 Smem 中。该指令实现的是存储器到存储器的操作。注意:该指令不能循环执行。该指令代码占 2 个字,但当 Smem 采用长偏移间接寻址或绝对寻址方式时,就会多占 1 个字。

例:ORM 0404h,*AR4+

执行前		执行后	
AR4	0100	AR4	0101
数据存储器 0100h	4444	0100h	4444

77. POLY Smem

操作数:

Smem:单数据存储操作数

执行:

$\text{Round}(\text{A}(32-16)) \times (\text{T}) + (\text{B}) \rightarrow \text{A}$

$(\text{Smem}) \ll 16 \rightarrow \text{B}$

状态位:

被 FRCT 位、OVM 位和 SXM 位影响。

影响 OVA 位。

说明:

单数据存储操作数 Smem 的内容左移 16 位,结果存放在累加器 B 中;同时并行执行,累加器 A 的高端(位 32~16)与 T 寄存器的值相乘,乘积加到累加器 B 中。再对此运算的结

果凑整,把最后结果存放在累加器 A 中。该指令在多项式计算中为实现每一单项只执行一个周期是很有用的。该指令代码占 2 个字,但当 Smem 采用长偏移间接寻址或绝对寻址方式时,就会多占 1 个字。

例:POLY * AR3+%

执行前		执行后	
A	00 1234 0000	A	00 0627 0000
B	00 0001 0000	B	00 2000 0000
T	5678	T	5678
AR3	0200	AR3	0201
数据存储器			
0200h	2000	0200h	2000

78. POPD Smem

操作数:

Smem:单数据存储器操作数

执行:

(TOS)→Smem

(SP)-1→SP

状态位:无。

说明:

把由 SP 寻址的数据存储器单元的内容转移到由 Smem 确定的数据存储器单元中。然后,SP 执行加 1 操作。

例:POPD 10

执行前		执行后	
DP	008	DP	008
SP	0300	SP	0301
数据存储器			
0300h	0092	0300h	0092
040Ah	0055	040Ah	0092

79. POPM MMR

操作数:

MMR:存储器映射寄存器

执行:

(TOS)→MMR

(SP)-1→SP

状态位:无。

说明:

把由 SP 寻址的数据存储器单元的内容转移到指定的存储器映射寄存器 MMR 中。然后,SP 执行加 1 操作。

例:POPM AR5

	执行前		执行后
AR5	0055	AR5	0060
SP	03F0	SP	03F1
数据存储器			
03F0h	0060	03F0h	0060

80. PORTR PA, Smem

操作数:

Smem:单数据存储操作数

$0 \leq PA \leq 65\,535$

执行:

$(PA) \rightarrow Smem$

状态位:无。

说明:

把一个 16 位数从一个外部 I/O 口 PA(地址为 16 位立即数)读入到指定的数据存储器单元 Smem 中。 \overline{IS} 引脚信号变为低电平,表明在访问 I/O 口; \overline{IOSTRB} 和 READY 的时序和读外部数据存储器的时序相同。该指令代码占 2 个字,但当 Smem 采用长偏移间接寻址或绝对寻址方式时,就会多占 1 个字。

例:PORTR 0.5 INDAT

	执行前		执行后
DP	000	DP	000
I/O 存储器			
0100h	7FFA	0100h	7FFA
数据存储器			
0200h	0000	0200h	7FFA

81. PORTW Smem,PA

操作数:

Smem:单数据存储操作数

$0 \leq PA \leq 65\,535$

执行:

$(Smem) \rightarrow PA$

状态位:无。

说明:

把指定的数据存储器单元 Smem 中的 16 位数据写到外部 I/O 口 PA 中去。 \overline{IS} 引脚信号变为低电平,表明在访问 I/O 口; \overline{IOSTRB} 和 READY 的时序和读外部数据存储器的时序相同。该指令代码占 2 个字,但当 Smem 采用长偏移间接寻址或绝对寻址方式时就会多占 1 个字。

例:PORTW OUTDAT,5h

执行前		执行后	
DP	001	DP	001
I/O 存储器			
0005h	0000	0005h	7FFA
数据存储器			
0087h	7FFA	0087h	7FFA

82. PSHD Smem

操作数:

Smem:单数据存储器操作数

执行:

$(SP) - 1 \rightarrow SP$

$(Smem) \rightarrow TOS$

状态位:无。

说明:

SP 执行减 1 操作后,把存储器单元 Smem 的内容压入到堆栈指针 SP 指向的数据存储器单元中去。在译码阶段读出 SP,并在访问阶段对它进行存储。该指令代码占 1 个字,但当 Smem 采用长偏移间接寻址或绝对寻址方式时,就会多占 1 个字。

例:PSHD *AR3+

执行前		执行后	
AR3	0200	AR3	0201
SP	8000	SP	7FFF
数据存储器			
0200h	07FF	0200h	07FF
7FFFh	0092	7FFFh	07FF

83. PSHM MMR

操作数:

MMR:存储器映射寄存器

执行:

$(SP) - 1 \rightarrow SP$

(MMR)→TOS

状态位:无。

说明:

堆栈指针 SP 执行减 1 操作后,再把存储器映射寄存器 MMR 中的内容压入到 SP 所指的数据存储器单元中去。

例:PSHM BRC

执行前		执行后	
BRC	1234	BRC	1234
SP	2000	SP	1FFF
数据存储器 1FFFh	07FF	1FFFh	1234

84. RC[D] cond[,cond[,cond]]

执行:

If(cond(s))

Then(TOS)→PC

(SP)+1→SP

Else

(PC)+1→PC

状态位:无。

说明:

当满足 cond 所给出的条件时,存放在栈顶的数据存储器的值弹到 PC 中,堆栈针 SP 加 1。如果不满足条件,仅仅执行 PC 加 1 操作。如果是延迟返回,紧接着该指令的两条单字指令或一条双字指令取出先执行。先执行这两个指令字不会影响正在被测试的条件。在把程序指针转移指向另一个单元前,也可对多个条件进行测试。

例:RC AGEQ,ANOV

执行前		执行后	
PC	0807	PC	2002
OVA	0	OVA	0
SP	0308	SP	0309
数据存储器 0308h	2002	0308h	2002

85. READA Smem

操作数:

Smem:单数据存储操作数

执行:

A→PAR

If((RC)≠)

(由 PAR 中的内容所决定的 Pmem 的地址单元)→Smem

(PAR)+1→PAR

(RC)-1→RC

Else(由 PAR 中的内容所决定的 Pmem 的地址单元)→Smem

状态位:无。

说明:

把累加器 A 确定的程序存储单元中的一个字传送到一个数据存储器单元 Smem 中去。一旦建立了循环流水线,指令就变成了单周期指令。可以循环执行该指令,以便把一块连续字(由累加器 A 确定起始地址)转移到一个连续的使用间接寻址方式的数据存储器空间中去。源和目的块不必全部都在片内或片外。

例:READA 6

	执行前		执行后
A	00 0000 0023	A	00 0000 0023
DP	004	DP	004
程序存储器			
0023h	0306	0023h	0306
数据存储器			
0206h	0075	0206h	0306

86. RESET

操作数:无。

执行:

(IPTR)<<7→PC 0→OVA 0→OVb 1→C 1→TC 0→ARP

0→DP 1→SXM 0→ASM 0→BRAf 0→HM 1→XF 0→C16

0→FRCT 0→CMPT 0→CPL 1→INTM 0→IFR 0→OVM

说明:

该指令实现了一个非屏蔽的软件复位,在任何时候都能使用,以便使 C54xx 处于可知状态。当执行这个复位指令时,就会给上面所列的状态位赋值。 $\overline{MP}/\overline{MC}$ 引脚在软件复位期间不被取样。IPTR 和外围寄存器的初始化与使用 \overline{RS} 的初始化有所不同。该指令不受 INTM 的影响,但它对 INTM 置位可以禁止中断。注意:该指令不能循环执行。

例:RESET

	执行前		执行后
PC	0025	PC	0080
INTM	0	INTM	1
IPTR	1	IPTR	1

87. RET[D]

操作数:无。

执行:

$(TOS) \rightarrow PC$

$(SP) - 1 \rightarrow SP$

状态位:无。

说明:

把栈顶 TOS 单元中的 16 位数据弹入到程序指针 PC 中,堆栈指针 SP 加 1。如果是延迟返回,紧接着该指令的 2 条单字指令或 1 条双字指令取出先执行。注意:该指令不能循环执行。

例:RET

执行前		执行后	
PC	2112	PC	1000
SP	0300	SP	0301
数据存储器 0300h	1000	0300h	1000

88. RETE[D]

操作数:无。

执行:

$(TOS) \rightarrow PC$

$(SP) - 1 \rightarrow SP$

$0 \rightarrow INTM$

状态位:

影响 INTM 位。

说明:

把栈顶单元 TOS 中的 16 位数据弹入到程序指针 PC 中,且从这个地址继续执行,堆栈指针 SP 加 1。该指令自动对 ST1 中的中断屏蔽位清 0,即允许中断。如果是延迟返回,紧接着该指令的 2 条单字指令或 1 条双字指令取出先执行。注意:该指令不能循环执行。

89. RETF[D]

操作数:无。

执行:

$(RTN) \rightarrow PC$

$(SP) + 1 \rightarrow SP$

0→INTM

状态位:

影响 INTM 位。

说明:

把快速返回寄存器 RTN 中的 16 位值装入到程序指针 PC 中。RTN 中保存了中断服务子程序返回的地址。RTN 是在返回时,而不是从堆栈中读 PC 时装入到 PC 中去的;然后,SP 执行加 1 操作。该指令自动对 ST1 中的中断屏蔽位 INTM 清 0(清 0 意味着允许中断)。如果是延迟返回,紧接着该指令的两条单字指令或一条双字指令取出先执行。注意:只有在该中断服务子程序执行期间没有调用和其他中断子程序时,才能使用该指令。

例:RETF

	执行前		执行后
PC	01C3	PC	0110
SP	2001	SP	2002
ST1	xCxx	ST1	x4xx
数据存储器			
2001h	0110	2001h	0110

90. RND src[,dst]

操作数:

src,dst:A(累加器 A),B(累加器 B)

执行:

(src)+8000h→dst

状态位:被 OVM 位影响。

说明:

把 2^{15} 加到源累加器 src 中,以对其进行凑整。凑整后的值存放在 dst 或者 src(在没有指定 dst 的情况下)中。注意:该指令不能循环执行。

例:RND A, B

	执行前		执行后
A	FF FFFF FFFF	A	FF FFFF FFFF
B	00 0000 0001	B	00 0000 7FFF
OVM	0	OVM	0

例:RND A

	执行前		执行后
A	00 7FFF FFFF	A	00 7FFF FFFF
OVM	1	OVM	1

91. ROL src

操作数:

src: A(累加器 A), B(累加器 B)

执行:

(C) → src(0)

(src(30-0)) → src(31-1)

(src(31)) → C

0 → src(39-32)

状态位:

被 C 位影响。

影响 C 位。

说明:

源累加器 src 循环左移一位。进位位 C 的值移入 src 的最低位, src 的最高位移入 C 中, 保护位清 0。

例: ROL A

执行前			
A	5F	B000	1234
C			
			0

执行后			
A	00	6000	2468
C			
			1

92. ROLTC src

操作数:

src: A(累加器 A), B(累加器 B)

执行:

(TC) → src(0)

(src(30-0)) → src(31-1)

(src(31)) → C

0 → src(39-32)

状态位:

被 TC 位影响。

影响 C 位。

说明:

源累加器 src 循环左移一位。TC 的值移入 src 的最低位, src 的最高位移到进位位 C 中, 保护位清 0。

例: ROLTC A

执行前				执行后			
A	81	C000	5555	A	00	8000	AAAB
C				x			
TC				1			

93. ROR src

操作数:

src: A(累加器 A), B(累加器 B)

执行:

(C) → src(31)

(src(31-1)) → src(30-0)

(src(0)) → C

0 → src(39-32) _

状态位:

被 C 位影响。

影响 C 位。

说明:

源累加器 src 循环右移一位。进位位 C 的值移入 src 的最高位, src 的最低位移到 C 中, 保护位清 0。

例: ROR A

执行前				执行后			
A	7F	B000	1235	A	00	5800	091A
C				0			

94. RPT

(1) RPT Smem

(2) RPT #K

(3) RPT #1k

操作数:

Smem: 单数据存储操作数

$0 \leq K \leq 255$ $0 \leq 1k \leq 65\,535$

执行:

(1) (Smem) → RC

(2) K → RC

(3) 1k → RC

状态位: 无。

说明:

当指令执行时,首先把循环的次数装入循环计数器(RC)。循环的次数(n)可由一个 16 位单数据存储器操作数 Smem 给定,也可由一个 8 位或 16 位常数 K 或 1k 给定。这样,紧接着的下一条指令会循环执行 n+1 次。RC 在执行减 1 操作时不能被访问。注意:该指令不能循环执行,即不能套用循环。

例:RPT DAT127;DAT127=0FFFh

	执行前		执行后
RC	<div style="border: 1px solid black; padding: 2px; display: inline-block;">0</div>	RC	<div style="border: 1px solid black; padding: 2px; display: inline-block;">000C</div>
DP	<div style="border: 1px solid black; padding: 2px; display: inline-block;">031</div>	DP	<div style="border: 1px solid black; padding: 2px; display: inline-block;">031</div>
数据存储器 0FFFh	<div style="border: 1px solid black; padding: 2px; display: inline-block;">000C</div>	0FFFh	<div style="border: 1px solid black; padding: 2px; display: inline-block;">000C</div>

例:RPT #1111h;重复下一条指令 4 370 次

	执行前		执行后
RC	<div style="border: 1px solid black; padding: 2px; display: inline-block;">0</div>	RC	<div style="border: 1px solid black; padding: 2px; display: inline-block;">1111</div>

95. RPTB[D] pmad

操作数:

$0 \leq \text{pmad} \leq 65\,535$

执行:

1→BRAf

If(延时)then

(PC)+4→RSA

Else

(PC)+2→RSA

Pmad→REA

状态位:影响 BRAf 位。

说明:

循环执行一指令块,循环的次数由存储器映射的块循环计数器(BRC)确定。BRC 必须在指令执行之前被装入值。程序执行时,块循环起始地址寄存器(RSA)中装入 PC+2(如果采用延迟就装入 PC+4);块循环尾地址寄存器(REA)中装入程序存储地址(pmad)。该指令执行时可以被中断。单指令循环也属于这块循环。为了套用该指令,必须保证以下几点:

(1) BRC、RSA 和 REA 寄存器必须作适当的保存。

(2) 块循环有效标志(BRAF)要适当地设置。

在带延迟的块循环中,紧接着该指令的两条单字指令和一条双字指令被取出先执行。注意:块循环可以通过对 BRAf 位清 0 来禁止;该指令不能循环执行,即不能套用循环。

例:ST #99,BRC

RPTB end_block-1

执行前	
PC	1000
BRC	1234
RSA	5678
REA	9ABC

执行后	
PC	1002
BRC	0063
RSA	1002
REA	end block - 1

例: ST #99,BRC; 执行循环块 100 次
MVDM POINTER,AR1

执行前	
PC	1000
BRC	1234
RSA	5678
REA	9ABC

执行后	
PC	1004
BRC	0063
RSA	1004
REA	end block - 1

96. RPTZ dst, #1k

操作数:

dst: A(累加器 A), B(累加器 B)

$0 \leq 1k \leq 65\,535$

执行:

$0 \rightarrow \text{dst}$

$1k \rightarrow \text{RC}$

状态位: 无。

说明:

对目的累加器 dst 清 0, 循环执行下一条指令 $n+1$ 次。其中 n 是循环计数器(RC)中的值。RC 的值是一个 16 位常数 $1k$ 。该指令代码占 2 个字。

例: RPTZ A, 1023; 重复下一条指令 1 024 次

STL A, *AR2+

执行前	
A	0F FE00 8000
RC	0000

执行后	
A	00 0000 0000
RC	03FF

97. RSBX N, SBIT

操作数:

$0 \leq \text{SBIT} \leq 15$

$N=0$ 或 1

执行:

$0 \rightarrow \text{STN}(\text{SBIT})$

状态位: 无。

说明:

对状态寄存器 ST0 和 ST1 的特定位清 0。N 指明了被修改的状态寄存器,SBIT 确定了被修改的位。可直接用状态寄存器中的一个域名作为操作数,而不用 N 和 SBIT。注意:该指令不能循环执行。

例:RSBX SXM

	执行前		执行后								
ST1	<table border="1"><tr><td></td><td></td><td></td><td>35CD</td></tr></table>				35CD	ST1	<table border="1"><tr><td></td><td></td><td></td><td>34CD</td></tr></table>				34CD
			35CD								
			34CD								

例:RSBX 1,8

	执行前		执行后								
ST1	<table border="1"><tr><td></td><td></td><td></td><td>35CD</td></tr></table>				35CD	ST1	<table border="1"><tr><td></td><td></td><td></td><td>34CD</td></tr></table>				34CD
			35CD								
			34CD								

98. SACCD src,Xmem,cond

操作数:

src:A(累加器 A),B(累加器 B)

Xmem:双数据存储操作数

执行:

If(cond)

Then

(src)<<(ASM-16)→Xmem

Else

(Xmem)→(Xmem)

状态位:

被 ASM 位和 SXM 位影响。

说明:

如果满足条件,源累加器 src 左移(ASM-16)位后存放到 Xmem 指定的存储器单元中;如果不满足条件,指令从 Xmem 中读出数据,然后又把它写回到原来的单元中去,即 Xmem 单元的值保持不变。

例:SACCD A,*AR3+0%,ALT

执行前				执行后			
A	FF	FE00	4321	A	FF	FE00	4321
ASM			01	ASM			01
AR0			0002	AR0			0002
AR3			0202	AR3			0204
数据存储器							
0202h			0101	0202h			FC00

99. SAT src

操作数:

src: A(累加器 A), B(累加器 B)

执行:

Saturate(src) → src

状态位:

影响 Ovsrc 位。

说明:

无论 OVM 的值是什么,都把源累加器 src 的值饱和成 32 位。

例: SAT B

	执行前		
B	71	2345	6789
OVB	x		

	执行后		
B	00	7FFF	FFFF
OVB	1		

例: SAT A

	执行前		
A	F8	1234	5678
OVA	x		

	执行后		
A	FF	8000	0000
OVA	1		

例: SAT B

	执行前		
B	00	0012	3456
OVB	x		

	执行后		
B	00	0012	3456
OVB	0		

100. SFTA src, SHIFT[, dst]

操作数:

src, dst: A(累加器 A), B(累加器 B)

$-16 \leq \text{SHIFT} \leq 15$

执行:

If SHIFT < 0

Then

(src((−SHIFT)−1)) → C

(src(39−0)) << SHIFT → dst

If SXM = 1

Then

(src(39)) → dst(39−(39+(SHIFT+1))) [OR src(39−(39−(39+(SHIFT+1))))]

Else

$0 \rightarrow \text{dst}(39 - (39 + (\text{SHIFT} + 1))) \text{ OR } \text{src}(39 - (39 + (\text{SHIFT} + 1)))$

Else

$(\text{src}(39 - \text{SHIFT})) \rightarrow \text{C}$

$((\text{src}) << \text{SHIFT} - 1) - 0 \text{ OR } \text{src}((\text{SHIFT} - 1) - 0))$

状态位:

被 SXM 位 OVM 位影响。

影响 C 位和 Ovdst 位(如果 $\text{dst} = \text{src}$, 影响 Ovsr 位)。

说明:

该指令对源累加器 src 进行算术移位, 结果存放在 dst(如果没有指定 dst 就放在 src)中。
指令的执行由 SHIFT 的值决定。

如果 SHIFT 的值 < 0 , 那么:

(1) $\text{src}((- \text{SHIFT}) - 1)$ 复制到进位位 C 中。

(2) 如果 SXM 为 1, 指令执行算术右移, 源累加器 src 的最高位移入到 $\text{dst}(39 - (39 + (\text{SHIFT} + 1)))$ 中。

(3) 如果 SXM 为 0, 把 0 写进 $\text{dst}(39 - (39 + (\text{SHIFT} + 1)))$ 中。

如果 SHIFT 的值 > 0 , 那么:

(1) $\text{src}(39 - \text{SHIFT})$ 复制到进位位 C 中。

(2) 指令产生一个算术左移。

(3) 把 0 写进 $\text{dst}((\text{SHIFT} - 1) - 0)$ 。

例: SFTA A, -5, B

执行前			
A	FF	8765	0055
B	00	4321	1234
C			
SXM			

执行后			
A	FF	8765	0055
B	FF	FC3B	2802
C			
SXM			

101. SFTC src

操作数:

src: A(累加器 A), B(累加器 B)

执行:

If(src) = 0

Then

$1 \rightarrow \text{TC}$

Else

If($\text{src}(31) \text{ XOR } \text{src}(30)$) = 0

Then

(最高两位为符号位) $0 \rightarrow \text{TC}$

$(\text{src}) << 1 \rightarrow \text{src}$

Else

(最高位为符号位) $1 \rightarrow TC$

说明:

如果源累加器 src 有 2 个有效符号位,就把 32 位的 src 左移 1 位,保护位保持不变,且使测试控制位 TC 清 0;如果只有 1 个符号位,对 TC 置 1。

例: SFTC A

执行前				执行后			
A	FF	FFFF	F001	A	FF	FFFF	E002
TC	x			TC	0		

102. SFTL $src, SHIFT[, dst]$

操作数:

src, dst : A(累加器 A), B(累加器 B)

$-16 \leq SHIFT \leq 15$

执行:

If $SHIFT < 0$

Then

$src((-\mathit{SHIFT})-1) \rightarrow C$

$Src(31-0) \ll SHIFT \rightarrow dst$

$0 \rightarrow dst(39-(31+(SHIFT+1)))$

If $SHIFT = 0$

Then

$0 \rightarrow C$

Else

$src(31-(SHIFT-1)) \rightarrow C$

$src((31-SHIFT)-0) \ll SHIFT \rightarrow dst$

$0 \rightarrow dst((SHIFT-1)-0)$

状态位:

影响 C 位。

说明:

对源累加器 src 进行逻辑移位,结果存放在 dst (如果没有指定 dst 就放在 src)中。指令的执行由 $SHIFT$ 的值决定;

如果 $SHIFT$ 的值 < 0 ,那么:

(1) $src((-\mathit{SHIFT})-1)$ 复制到进位位 C 中。

(2) 指令执行一个逻辑右移。

(3) 把 0 写进 $dst((SHIFT-1)-0)$ 中。

如果 $SHIFT$ 的值 > 0 ,那么:

(1) $src(31-(SHIFT-1))$ 复制到进位位 C 中。

(2) 指令执行一个逻辑左移。

(3) 把 0 写进 $\text{dst}((\text{SHIFT}-1)-0)$ 中。

例: SFTL A, -5, B

执行前				执行后			
A	FF	8765	0055	A	FF	8765	0055
B	FF	8000	0000	B	00	043B	2802
C			0	C			1

103. SQDST Xmem, Ymem

操作数:

Xmem, Ymem: 双数据存储操作数

执行:

$(A(32-16)) \times (A(32-16)) + (b) \rightarrow B$

$((Xmem) - (Ymem)) \ll 16 \rightarrow A$

状态位:

被 OVM 位、FRCT 位和 SXM 位影响。

影响 C 位、OVA 位和 OVB 位。

说明:

计算两个向量距离的平方, 累加器 A 的高端 (位 32~16) 平方后加到累加器 B 中, Xmem 或 Ymem 相减, 差左移 16 位后存放到累加器 A 中。

例: SQDST *AR3+, AR4+

执行前				执行后			
A	FF	ABCD	0000	A	FF	FFAB	0000
B	00	0000	0000	B	00	1BB1	8229
FRCT			0	FRCT			0
AR3			0100	AR3			0101
AR4			0200	AR6			0201
数据存储器							
0100h			0055	0100h			0055
0200h			00AA	0200h			00AA

104. SQUR

(1) SQUR Smem, dst

(2) SQUR A, dst

操作数:

Smem: 单数据存储操作数

dst: A (累加器 A), B (累加器 B)

执行:

(1) $(\text{Smem}) \rightarrow T(\text{Smem}) \times (\text{Smem}) \rightarrow \text{dst}$

(2) $(A(32-16)) \times (A(32-16)) \rightarrow \text{dst}$

状态位:

被 OVM 位和 FRCT 位影响。

影响 Ovsrce 位。

说明:

一个单数据存储器操作数 Smem 或累加器 A 的高端(位 32~16)平方后,结果存放在目的累加器 dst 中。如果用的是累加器 A, T 就不受影响;如果操作数为 Smem,就要把 Smem 放入 T 寄存器中。

例: SQR, B

执行前				执行后			
B	00	0000	01F4	B	00	0000	00E1
T			0003	T			000F
FRCT			0	FRCT			0
DP			006	DP			006
数据存储器							
031Eh			000F	031Eh			000F

例: SQR A, B

执行前				执行后			
A	00	000F	0000	A	00	000F	0000
B	00	0101	0101	B	00	0000	01C2
FRCT			1	FRCT			1

105. SQURA Smem,src

操作数:

Smem; 单数据存储器操作数

src; A(累加器 A), B(累加器 B)

执行:

$(\text{Smem}) \rightarrow T$

$(\text{Smem}) \times (\text{Smem}) + (\text{src}) \rightarrow \text{src}$

状态位:

被 OVM 位和 FRCT 位影响。

影响 Ovsrce 位。

说明:

把数据存储器值 Smem 存放到 T 寄存器中; 平方 Smem, 再把乘积加到源累加器 src 中, 结果存放在 src 中。

例: SQR 30, B

执行前				执行后			
B	00	0320	0000	B	00	0320	00E1
T			0003	T			000F
FRCT			0	FRCT			0
DP			006	DP			006
数据存储器							
031Eh			000F	031Eh			000F

例: SQURA * AR3+, A

执行前				执行后			
A	00	0000	01F4	A	00	0000	02D5
T			0003	T			000F
FRCT			0	FRCT			0
AR3			031E	AR3			031F
数据存储器							
031Eh			000F	031Eh			000F

106. SQURS Smem,src

操作数:

Smem: 单数据存储操作数

src: A(累加器 A), B(累加器 B)

执行:

(Smem) → T

(src) - (Smem) × (Smem) → src

状态位:

被 OVM 位和 FRCT 位影响。

影响 Ovsr 位。

说明:

把数据存储器值 Smem 存放到 T 寄存器中; 平方 Smem, 再从源累加器 src 中减去这个平方值, 结果存放在 src 中。

例: SQURS * AR3, B

执行前				执行后			
B	00	014B	5DB0	B	00	0000	0320
T			8765	T			1234
FRCT			0	FRCT			0
AR3			0309	AR3			0309
数据存储器							
0309h			1234	0309h			1234

107. SRCCD Xmem,cond

操作数:

Xmem: 双数据存储操作数

执行:

If(cond)

Then

(BRC)→Xmem

Else

(Xmem)→Xmem

状态位: 无。

说明:

如果满足条件, 指令把块循环计数器(BBC)中的内容存放到 Xmem 中去; 如果不满足条件, 指令把 Xmem 中的内容读出, 再把它写回去, 即 Xmem 保持不变。

例: SRCCD * AR5--, AGT

执行前				执行后			
A	00	70FF	FFFF	A	00	70FF	FFFF
AR5	0202			AR5	0201		
BRC	4321			BRC	4321		
数据存储器							
0202h	1234			0202h	4321		

108. SSBX N,SBIT

操作数:

$0 \leq \text{SBIT} \leq 15$

N=0 或 1

执行:

$1 \rightarrow \text{STN}(\text{SBIT})$

状态位: 无。

说明:

状态寄存器 ST0 或 ST1 的指定位置为 1。N 指定了所修改的状态寄存器, SBIT 指定了被修改的位。状态寄存器中域名能够用来代替 N 和 SBIT 作为操作数(见例)。

例: SSBX SXM

执行前		执行后	
ST1	34CD	ST1	35CD

例:SSBX1,8

	执行前		执行后
ST1	<div>34CD</div>	ST1	<div>35CD</div>

109. ST

- (1) ST T,Smem
- (2) ST TRN,Smem
- (3) ST #lk,Smem

操作数:

Smem:单数据存储操作数

$-32768 \leq lk \leq 32768$

执行:

- (1) (T)→Smem
- (2) (TRN)→Smem
- (3) lk→Smem

状态位:无。

说明:

把 T 寄存器的内容、过渡寄存器(TRN)的内容或一个 16 位常数 lk 存放到数据存储单元 Smem 中去。

例:ST FFFFh,0

	执行前		执行后
DP	<div>004</div>	DP	<div>004</div>
数据存储器			
0200h	<div>0101</div>	0200h	<div>FFFF</div>

例:ST TRN,5

	执行前		执行后
DP	<div>004</div>	DP	<div>004</div>
TRN	<div>1234</div>	AR5	<div>1234</div>
数据存储器			
0200h	<div>0030</div>	0200h	<div>1234</div>

例:ST T,*AR7—

	执行前		执行后
T	<div>4210</div>	T	<div>4210</div>
AR7	<div>0321</div>	AR7	<div>0320</div>
数据存储器			
0321h	<div>1200</div>	0321h	<div>4210</div>

110. STH

语法:

- (1) STH src, Smem
 (2) STH src, ASM, Smem
 (3) STH src, SHFT, Xmem
 (4) STH src[, SHIFT], Smem

操作数:

src: A(累加器 A), B(累加器 B)

Smem: 单数据存储操作数

Xmem: 双数据存储操作数

 $0 \leq \text{SHFT} \leq 15$ $-16 \leq \text{SHIFT} \leq 15$

执行:

- (1) $(\text{src}(31:16)) \rightarrow \text{Smem}$
 (2) $(\text{src}) \ll (\text{ASM} - 16) \rightarrow \text{Smem}$
 (3) $(\text{src}) \ll (\text{SHFT} - 16) \rightarrow \text{Xmem}$
 (4) $(\text{src}) \ll (\text{SHIFT} - 16) \rightarrow \text{Smem}$

状态位:

被 SXM 影响。

说明:

把源累加 src 的高端(位 31~16)存放到数据存储单元 Smem 中去。src 进行左移,移动位数由 ASM、SHFT 或 SHIFT 决定;然后再把移位后的值(位 31~16)存放到数据存储单元(Smem 或 Xmem)中。如果 SXM=0, src 的位 39 复制到数据存储单元的最高位。如果 AXM=1, 就把移位后进行了符号扩展的位 39 存放到数据存储单元的最高位。

例: STH A, 10

执行前				执行后			
A	FF	8765	4321	A	FF	8765	4321
DP				DP			
数据存储器				数据存储器			
020Ah			1234	020Ah			8765

例: STH B, -8, *AR7-

执行前				执行后			
B	FF	8421	1234	B	FF	8421	1234
AR7				AR7			
数据存储器			0321	数据存储器			0320
0321h			ABCD	0321h			FF84

111. STL

语法:

- (1) STL src, Smem
- (2) STL src, ASM, Smem
- (3) STL src, SHFT, Xmem
- (4) STL src [, SHIFT] Smem

操作数:

src: A(累加器 A), B(累加器 B)

Smem: 单数据存储操作数

Xmem: 双数据存储操作数

$0 \leq \text{SHFT} \leq 15$

$-16 \leq \text{SHIFT} \leq 15$

执行:

- (1) (src(15~0)) → Smem
- (2) (src) << ASM → Smem
- (3) (src) << SHFT → Xmem
- (4) (src) << SHIFT → Smem

状态位:

被 SXM 位影响。

说明:

把源累加 src 的低端(位 15~0)存放到数据存储单元 Smem 中去。src 进行左移操作, 移动位数由 ASM、SHFT 或 SHIFT 决定。然后再把移位后的值的位 15~0 存放到数据存储单元(Smem 或 Xmem)中去。当移位值为正时, 低位添 0。

例: STL A, 11

执行前				执行后			
A	FF	8765	4321	A	FF	8765	4321
DP			004	DP			004
数据存储器							
020Bh			1234	020Bh			4321

例: STL B, -8, * AR7-

执行前				执行后			
B	FF	8421	1234	A	FF	8421	1234
SXM			0	SXM			0
AR7			0321	AR7			0320
数据存储器							
0321h			0099	0321h			2112

112. STLM src,MMR

操作数:

src: A(累加器 A), B(累加器 B)

MMR: 存储器映射寄存器

执行:

$(src(15-0)) \rightarrow MMR$

状态位: 无。

说明:

把源累加 src 的低端(位 15~0)存放到存储器映射寄存器 MMR 中。无论 DP 的当前值或 ARx 的高 9 位是多少,有效地址的高 9 位清 0。指令允许 src 存放在数据页第 0 页中的任何一个存储器单元中,而不必修改状态寄存器 ST0 中的 DP 域。

例: STLM A, BRC

执行前	
A	FF 8765 4321
BRC(1Ah)	1234

执行后	
A	FF 8765 4321
BRC	4321

例: STLM B, *AR1—

执行前	
B	FF 8321 1234
AR1	3B17
AR7(17h)	0099

执行后	
A	FF 8421 1234
AR1	0016
AR7	1234

113. STM #lk,MMR

操作数:

MMR: 存储器映射寄存器

$-32\,768 \leq lk \leq 32\,767$

执行:

$lk \rightarrow MMR$

状态位: 无。

说明:

该指令的功能是把 16 位常数 lk 存放到一个存储器映射寄存器 MMR 或一个在第 0 数据页中的存储器单元,而不必修改状态寄存器中 ST0 中的 DP 域。无论 DP 的当前值或 ARx 的高 9 位是多少,都要对有效地址的高 9 位清 0。

例: STM 0FFFFh, IMR

执行前	
IMR	FF01

执行后	
IMR	FFFF

例:STM 8765h, * AR7 +

	执行前		执行后
AR0	0000	AR0	8765
AR7	8010	AR7	0011

114. ST src, Ymem || ADD Xmem, dst

操作数:

src, dst: A(累加器 A), B(累加器 B)

Xmem, Ymem: 双数据操作数

dst_:

if dst = A

then

dst_ = B;

if dst = B

then

dst_ = A

执行:

(src) << (ASM - 16) → Ymem

(dst_) + (Xmem) << 16 → dst

状态位:

被 OVM 位、SXM 位和 ASM 位影响。

影响 C 位和 Ovdst 位。

说明:

源累加器 src 移动由 (ASM - 16) 决定的位数, 然后存放到数据存储器单元 Ymem 中; 同时并行执行, dst 的内容与左移 16 位后的数所存储器操作数 Xmem 相加, 结果存放在 dst 中。如果 src 等于 dst, 那么存放到 Ymem 中的值是该操作执行之前的 src 的值。

例: ST A, * AR3 || ADD * AR5 + 0%, B

	执行前		执行后
A	FF 8421 1000	A	FF 8021 1000
B	00 0000 1111	B	FF 0422 1000
OVM	0	OVM	0
SXM	1	SXM	1
ASM	1	ASM	1
AR0	0002	AR0	0002
AR3	0200	AR3	0200
AR5	0300	AR5	0302
数据存储器			
0200h	0101	0200h	0842
0300h	8001	0300h	8001

115. ST

语法:

(1) ST src, Ymem || LD Xmem, dst

(2) ST src, Ymem || LD Xmem, T

操作数:

src, dst: A(累加器 A), B(累加器 B)

Xmem, Ymem: 双数据操作数

执行:

(1) (src) << (ASM-16) → Ymem

(Xmem) << 16 → dst

(2) (src) << (ASM-16) → Ymem

(Xmem) → T

状态位:

被 OVM 位和 ASM 位影响。

影响 C 位。

说明:

源累加器 src 移动由 (ASM-16) 所决定的位数, 然后把移位后的值放到数据存储器单元 Ymem 中; 同时并行执行, 把 16 位双数据存储器操作数 Xmem 装入到目的累加器 dst 或 T 寄存器中。如果 src 等于 dst, 存放到 Ymem 中的值是该操作执行之前的 src 的值。

例: ST A, *AR3 || LD *AR4, T

执行前				执行后			
A	FF	8421	1234	A	FF	8421	1234
T			3456	T			80FF
ASM			1	ASM			1
AR3			0200	AR3			0200
AR4			0100	AR4			0100
数据存储器							
0200h			0001	0200h			0842
0100h			80FF	0100h			80FF

116. ST src, Ymem || MAC[R] Xmem, dst

操作数:

src, dst: A(累加器 A), B(累加器 B)

Xmem, Ymem: 双数据操作数

执行:

(src << (ASM-16)) → Ymem

If(Rounding)

Then

$\text{Round}((\text{Xmem}) \times (\text{T}) + (\text{dst})) \rightarrow \text{dst}$

Else

$(\text{Xmem}) \times (\text{T}) + (\text{dst}) \rightarrow \text{dst}$

状态位:

被 OVM 位、SXM 位、ASM 位和 FRCT 位影响。

影响 C 位和 Ovdst 位。

说明:

源累加器 src 移动由 (ASM-16) 所决定的位数, 然后把移位后的值存放于数据存储器单元 Ymem 中; 同时并行执行, T 寄存器的值与数据存储器操作数 Xmem 相乘, 乘积与目的累加器 dst 相加(可以带凑整运算), 结果存放在 dst 中。如果 src 等于 dst, 存放 Ymem 中的值为该操作执行之前 src 的值。如果使用了 R 后缀, 就会对结果进行凑整: 加上 2^{15} 再对低端(位 15~0)清 0。

例: ST A, *AR4-||MAC *AR5,B

执行前			执行后		
A	00	0011 1111	A	00	0011 1111
B	00	0000 1111	B	00	010C 9511
T		0400	T		0400
ASM		5	ASM		5
FRCT		0	FRCT		0
AR4		0100	AR4		00FF
AR5		0200	AR5		0200
数据存储器					
100h		1234	100h		0222
200h		4321	200h		4321

例: ST A, *AR4+||MACR *AR5+,B

执行前			执行后		
A	00	0011 1111	A	00	0011 1111
B	00	0000 1111	B	00	010D 0000
T		0400	T		0400
ASM		10	ASM		1C
FRCT		0	FRCT		0
AR4		0100	AR4		0101
AR5		0200	AR5		0201
数据存储器					
100h		1234	100h		0001
200h		4321	200h		4321

117. ST src, Ymem || MAS[R] Xmem, dst

操作数:

src, dst: A(累加器 A), B(累加器 B)

Xmem, Ymem: 双数据操作数

执行:

$(src(31-16) \ll (ASM-16)) \rightarrow Ymem$

If(Rounding)

Then

$Round((dst) - (Xmem) \times (T)) \rightarrow dst$

Else

$(dst) - (Xmem) \times (T) \rightarrow dst$

状态位:

被 OVM 位、SXM 位、ASM 位和 FRCT 位影响。

影响 C 位和 Ovdst 位。

说明:

源累加器 src 移动由 $(ASM-16)$ 所决定的位数, 然后把移位后的值存放于数据存储器单元 Ymem 中; 同时并行执行, T 寄存器的值与数据存储器操作数 Xmem 相乘, 乘积与目的累加器 dst 相减(可以带凑整运算), 结果存放在 dst 中。如果 src 等于 dst, 存放 Ymem 中的值为该操作执行之前 src 的值。如果使用了 R 后缀, 就会对结果进行凑整: 加上 2^{15} 再对低端(位 15~0)清 0。

例: ST A, *AR4+ || MAS *AR5, B

执行前			执行后		
A	00	0011 1111	A	00	0011 1111
B	00	0000 1111	B	FF	FEF3 8D11
T		0400	T		0400
ASM		5	ASM		5
FRCT		0	FRCT		0
AR4		0100	AR4		0101
AR5		0200	AR5		0200
数据存储器					
0100h		1234	0100h		0222
0200h		4321	0200h		4321

例: ST A, *AR4+ || MASR *AR5+, B

执行前			执行后		
A	00	0011 1111	A	00	0011 1111
B	00	0000 1111	B	FF	FEF4 0000
T		0400	T		0400

ASM	0001	ASM	0001
FRCT	0	FRCT	0
AR4	0100	AR4	0101
AR5	0200	AR5	0201
数据存储器			
0100h	1234	0100h	0022
0200h	4321	0200h	4321

118. ST src, Ymem || MPY Xmem, dst

操作数:

src, dst: A(累加器 A), B(累加器 B)

Xmem, Ymem: 双数据操作数

执行:

$(src(31-16) \ll (ASM-16)) \rightarrow Ymem$

$(Xmem) \times (T) \rightarrow dst$

状态位:

被 OVM 位、SXM 位和 FRCT 位影响。

影响 C 位和 Ovdst 位。

说明:

源累加器 src 移动由 (ASM-16) 所决定的位数, 然后把移位后的值放到数据存储器单元 Ymem 中; 同时并行执行, T 寄存器的值与 16 位双数据存储器操作数 Xmem 相乘, 乘积存放在 dst 中。如果 src 等于 dst, 存放到 Ymem 中的值是该操作执行之前的 src 的值。

例: ST A, *AR3+ || MPY *AR5+, B

执行前			执行后		
A	FF	B421 1234	A	FF	8421 1234
B	xx	xxxx xxxx	B	00	2000 0000
T		4000	T		4000
ASM		00	ASM		00
FRCT		1	FRCT		1
AR3		0200	AR3		0201
AR5		0300	AR5		0301
数据存储器					
0200h		1111	0200h		8421
0300h		4000	0300h		4000

119. ST src, Ymem || SUB Xmem, dst

操作数:

src, dst: A(累加器 A), B(累加器 B)

Xmem, Ymem; 双数据操作数

dst_;

if dst = A

then

dst_ = B;

if dst = B

then

dst_ = A

执行:

$(src(31-16) \ll (ASM-16)) \rightarrow Ymem$

$(Xmem) \ll 16 - (dst_) \rightarrow dst$

状态位:

被 OVM 位、SXM 位和 ASM 位影响。

影响 C 位和 Ovdst 位。

说明:

源累加器 src 移动由 (ASM-16) 所决定的位数, 然后把移位后的值存放到数据存储器单元 Ymem 中; 同时并行执行, 从左移了 16 位的双数据存储器操作数 Xmem 中减去 dst 的值, 并把结果存放在 dst 中。如果 src 等于 dst, 存入 Ymem 中的值是该操作执行之前的 src 的值。

例: ST A, *AR3 - || SUB *AR5 + 0%, B

执行前			执行后		
A	FF	8421 0000	A	FF	8421 0000
B	00	1000 0001	B	FF	FBE0 0000
ASM		01	ASM		01
SXM		1	SXM		1
AR0		0002	AR0		0002
AR3		01FF	AR3		01FE
AR5		0300	AR5		0302
数据存储器					
01FFh		1111	01FFh		0842
0300h		8001	0300h		8001

120. STRCD Xmem, cond

操作数:

Xmem; 双数据操作数

执行:

If(cond)

(T) \rightarrow Xmem

Else

(Xmem) \rightarrow Xmem

状态位:无。

说明:

如果满足条件,就把 T 寄存器的值存放到数据存储器单元 Xmem 中去;如果不满足条件,指令从单元 Xmem 中读出数据,然后再把它写到 Xmem 中去,即 Xmem 中的数据保持不变。

例:STRCD *AR5--,AGT

执行前			执行后				
A	00	70FF	FFFF	A	00	70FF	FFFF
T			4321	T			4321
AR5			0202	AR5			0201
数据存储器							
0202h			1234	0202h			4321

121. SUB

语法:

- (1) SUB Smem,src
- (2) SUB Smem,TS,src
- (3) SUB Smem,16,src[,dst]
- (4) SUB Smem[,SHIFT],src[,dst]
- (5) SUB Xmem,SHFT,src
- (6) SUB Xmem,Ymem,dst
- (7) SUB #lk[,SHFT],src[,dst]
- (8) SUB #lk,16,src[,dst]
- (9) SUB src[,SHIFT],[,dst]
- (10) SUB src,ASM[,dst]

操作数:

src,dst:A(累加器 A),B(累加器 B)

Xmem,Ymem;双数据操作数

Smem;单数据操作数

执行:

- (1) (src)-(Smem)→src
- (2) (src)-(Smem)<<TS→src
- (3) (src)-(Smem)<<16→dst
- (4) (src)-(Smem)<<SHIFT→dst
- (5) (src)-(Smem)<<SHFT→src
- (6) (Xmem)<<16-(Ymem)<<16→dst
- (7) (src)-lk<<SHFT→dst
- (8) (src)-lk<<16→dst
- (9) (dst)-(src)<<SHIFT→dst

(10) $(dst) - (src) \ll ASM \rightarrow dst$

状态位:

被 SXM 位和 OVM 影响。

影响 C 位和 Ovdst 位(如果 $dst = src$, 影响 Ovsrce 位)。

说明:

从选定的累加器或采用双数据存储器寻址方式的 16 位操作数 Xmem 中减去一个 16 位的值。16 位数据可以是单数据操作数、双数据操作数、立即数据或源累加器的移位值。如果指定了目的累加器, 结果就放在其中; 否则, 结果放在源累加器中。大部分指令形式的第二操作数都进行了移位。对于左移, 低位添 0, 高位当 $SXM=1$ 时进行符号扩展, 否则添 0; 对于右移, 高位当 $SXM=0$ 时进行符号扩展, 否则添 0。

例: SUB *AR1+, 14, A

执行前			执行后		
A	00	0000 1200	A	FF	FAC0 1200
C		x	C		0
SXM		1	SXM		1
AR1		0100	AR1		0101
数据存储器					
0100h		1500	0100h		1500

例: SUB A, -8, B

执行前			执行后		
A	00	0000 1200	A	00	0000 1200
B	00	0000 1800	B	00	0000 17EE
C		x	C		1
SXM		1	SXM		1

122. SUBB Smem, src

操作数:

src, dst; A(累加器 A), B(累加器 B)

Smem: 单数据操作数

执行:

$(src) - (Smem) - (\text{进位位 C 的逻辑“反”}) \rightarrow src$

状态位:

被 OVM 位和 C 位影响。

影响 C 位和 Ovsrce 位。

说明:

该指令的功能是: 从源累加器 src 中减去 16 位单数据存储器操作数 Smem 的值和进位位 C 的逻辑“反”, 且不进行符号扩展。

例: SUBB 5, A

执行前				执行后			
A	00	0000	0006	A	FF	FFFF	FFFF
C			0	C			0
DP			008	DP			008
数据存储器							
0405h			0006	0405h			0006

例: SUBB *AR1+, B

执行前				执行后			
B	FF	8000	0006	B	FF	8000	0000
C			1	C			1
OVM			1	OVM			1
AR1			0405	AR1			0406
数据存储器							
0405h			0006	0405h			0006

123. SUBC Smem, src

操作数:

src, dst: A(累加器 A), B(累加器 B)

Smem: 单数据操作数

执行:

$(src) - ((Smem) \ll 15) \rightarrow (ALUoutput)$

if $0 \leq ALUoutput$

Then

$((ALUoutput) \ll 1) + 1 \rightarrow src$

Else

$(src) \ll 1 \rightarrow src$

状态位:

被 SXM 影响。

影响 C 位和 Ovsrsrc 位。

说明:

16 位单数据存储器操作数 Smem 左移 15 位, 然后再从源累加器 src 中减去移位后的值。如果结果大于等于 0, 结果就左移一位, 加上 1, 再存放到 src 中; 否则, 只把 src 的值左移一位, 再存放到 src 中。

除数和被除数在这条指令中都假设为正, SXM 将影响该操作;

如果 SXM=1, 除数的最高位必须为 0;

如果 SXM=0, 任何一个 16 位除数值都可以。

src 中的被除数必须初始化为正(位 31 为 0), 且在移位后也必须保持为正。该指令影响 OVA 或 OVB, 但不受 OVM 影响。所以, 当发生溢出时, src 不进行饱和运算。

例: RPT #15
SUBC *AR1,B

执行前			执行后		
B	00	0000 0041	B	00	0002 0009
C		x	C		1
AR1		1000	AR1		1000
数据存储器					
1000h		0007	1000h		0007

124. SUBS Smem,src

操作数:

src,dst:A(累加器 A),B(累加器 B)

Smem:单数据操作数

执行: $\text{unsigned}(\text{src}) - (\text{Smem}) \rightarrow \text{src}$

状态位:

被 OVM 位影响。

影响 C 位和 Ovsrsrc 位。

说明:

从源累加器 src 中减去 16 位单数据存储器操作数 Smem 的值。无论 SXM 的值为多少, Smem 都被看作是一个 16 位无符号数。

例: SUBS *AR2-,B

执行前			执行后		
B	00	0000 0002	B	FF	FFFF 0FFC
C		x	C		0
AR2		0100	AR2		00FF
数据存储器					
0100h		F006	0100h		F006

125. TRAP K

操作数:

$0 \leq K \leq 31$

执行:

$(\text{SP}) - 1 \rightarrow \text{SP}$

$(\text{PC}) + 1 \rightarrow \text{TOS}$

$K \rightarrow \text{PC}$ 指明的中断向量表

状态位:无。

说明:

让程序指针 PC 指向由 K 指定的中断向量。指令允许使用软件来执行任何中断服务子程序。指令执行时,先把 PC+1 压入由 SP 寻址的数据存储器单元。这使得返回指令在执行完中断服务子程序后,能从 SP 所指的单元中找到要执行的下一条指令的地址。该指令是非屏蔽的,不会受 INTM 的影响,也不会影响 INTM。注意:该指令不能循环执行。

例:TRAP 10h

执行前		执行后	
PC	1233	PC	FFC0
SP	03FF	SP	03FE
数据存储器			
03FFh	9653	03FFh	1234

126. WRITA Smem

操作数:

Smem:单数据操作数

执行:

A→PAR

If(RC)≠0

Then

(Smem)→(由 PAR 中的内容所确定的 Pmem 的地址单元)

(PAR)+1→PAR

(RC)-1→RC

Else

(Smem)→(由 PAR 中的内容所确定的 Pmem 的地址单元)

状态位:无。

说明:

把一个字从一个由 Smem 确定的数据存储器单元传送到一个程序存储器单元。程序存储器的地址由累加器 A 确定,可以通过循环执行该指令,把数据存储器中的连续字(使用间接寻址)转移到由 PAR 寻址的连续的程序存储器空间,PAR 的初始值是累加器 A 的低 16 位值。源和目的块都不必完全在片内或片外。当使用循环时,一旦建立了循环流水线,该指令就变成了单周期指令。

例:WRITA 5

执行前		执行后	
A	00 0000 0257	A	00 0000 0257
DP	032	DP	032
程序存储器			
0257h	0306	0257h	4339
数据存储器			
1005h	4339	1005h	4339

127. XC n,cond[,[,cond]]

操作数:

n=1 或 2

执行:

If(cond)

Then

紧接着的 n 条指令被执行

Else

紧接着的 n 条 nop 指令被执行

状态位:无。

例:XC 1,ALEQ

MAR *AR1+

ADDA,DAT100

		执行前		
A		FF	FFFF	FFFF
AR1		0032		

		执行后		
A		FF	FFFF	FFFF
AR1		0033		

128. XOR

语法:

(1) XOR Smem,src

(2) XOR #lk[,SHFT],src[,dst]

(3) XOR #lk,16,src[,dst]

(4) XOR src[,SHFT][,dst]

操作数:

src,dst:A(累加器 A),B(累加器 B)

Smem:单数据操作数

$0 \leq \text{SHFT} \leq 15$

$-16 \leq \text{SHIFT} \leq 15$

$0 \leq \text{lk} \leq 65\,535$

执行:

(1) (Smem)XOR(src)→src

(2) $\text{lk} \ll \text{SHFT} \text{ XOR}(\text{src}) \rightarrow \text{dst}$

(3) $\text{lk} \ll 16 \text{ XOR}(\text{src}) \rightarrow \text{dst}$

(4) $(\text{src}) \ll \text{SHIFT} \text{ XOR}(\text{dst}) \rightarrow \text{dst}$

状态位:无。

说明:

16 位单数据存储器操作数 Smem(按指令指定移位)与所选定的累加器相“异或”,结果存放在 dst 或 src 中。对于左移,低位添 0,高位不进行符号扩展;对于右移,不进行符号扩展。

例: XOR *AR3+, A

		执行前		
A		00	00FF	1200
AR3		0100		
数据存储器				
0100h		1500		

		执行后		
A		00	00FF	0700
AR3		0101		
数据存储器				
0100h		1500		

例: XOR A, +3, B

		执行前		
A		00	0000	1200
B		00	0000	1800

		执行后		
A		00	0000	1200
B		00	0000	0000

129. XORM #lk, Smem

操作数:

Smem; 单数据操作数

$0 \leq lk \leq 65535$

执行:

$lk \text{ XOR}(Smem) \rightarrow Smem$

状态位: 无。

说明:

一个数据存储器单元 Smem 的内容和一个 16 位常数 lk 相“异或”, 结果写到 Smem 中。

注意: 该指令不能循环执行。

例: XORM 0404h, *AR4—

		执行前		
AR4		0100		
数据存储器				
0100h		4444		

		执行后		
AR4		00FF		
数据存储器				
0100h		4040		

参考文献

- 1 TMS320C54X Code Composer Studio Tutorial. Literature Number: SPRU327C February 2000. Texas Instruments Incorporated
- 2 Code Composer Studio User's Guide. Literature Number: SPRU328A August 1999. Texas Instruments Incorporated
- 3 Code Composer Studio Getting Started Guide. Literature Number: SPRU509C November 2001. Texas Instruments Incorporated
- 4 TMS320C54X DSP/BIOS User's Guide. Literature Number: SPRU326C May 2000. Texas Instruments Incorporated
- 5 Analysis Toolkit for Code Composer Studio User's Guide. Literature Number: SPRU623A. January 2003. Texas Instruments Incorporated
- 6 TMS320C54X Code Composer Studio version 2 help
- 7 张雄伟, 陈亮, 徐光辉. DSP 集成开发与应用实例. 北京: 电子工业出版社, 2002
- 8 汪安民. TMS320C54XX DSP 实用技术. 北京: 清华大学出版社, 2002

《单片机与嵌入式系统应用》月刊

- 北京航空航天大学出版社承办
- 何立民教授主编
- 中央级科技期刊
- 嵌入式系统专业期刊
- 特色: 专业期刊、专家办刊、着眼世界、面向国内、应用为主、读者第一
- 已开辟有 8 个栏目: 业界论坛、专题论述、技术纵横、新器件新技术、应用天地、经验交流、学习园地、编读往来
- 杂志社网址: <http://www.microcontroller.com.cn>,
<http://www.dpj.com.cn>

诚挚欢迎业界人士向本刊投稿, 欢迎广大读者订阅本刊。



杂志社联系方式

通信或汇款地址: 北京市海淀区学院路 37 号《单片机与嵌入式系统应用》杂志社

邮编: 100083

电话: (010)82317029, 82313656

传真: 010-82317043

E-mail: mcu@publica.bj.cninfo.net mcupress@263.net.cn (投稿专用)

广告业务联系人: 王金萍 魏洪亮

开户行: 北京市商业银行学院路支行

户名: 《单片机与嵌入式系统应用》杂志社

账号: 010903391001201110299-36

杂志订阅方式

国内统一刊号: CN 11-4530/V

国际标准刊号: ISSN 1009-623X

每期定价: 8 元, 全年定价: 96 元, 每月 1 日出版。

第一种订阅方式: 通过邮局订阅, 邮发代号: 2-765。

第二种订阅方式: 直接向《单片机与嵌入式系统应用》杂志社订阅, 另加 15% 邮资。

向北航出版社直接邮购图书地址: 北京航空航天大学出版社邮购组(100083) 另加 2 元挂号费。

邮购 E-mail: bhpress@263.net 电话: 010-82316936, 传真: 010-82317031。

单片机与嵌入式系统图书详细介绍请查阅出版社网站: <http://www.buaapress.com.cn>

进入出版社网站主页后, 点击“单片机与嵌入式系统图书专版”

投稿单片机与嵌入式系统图书请联系: 北航出版社 马广云 电话: 010-82317022 传真: 010-82317022

E-mail: pressb@public3.bta.net.cn