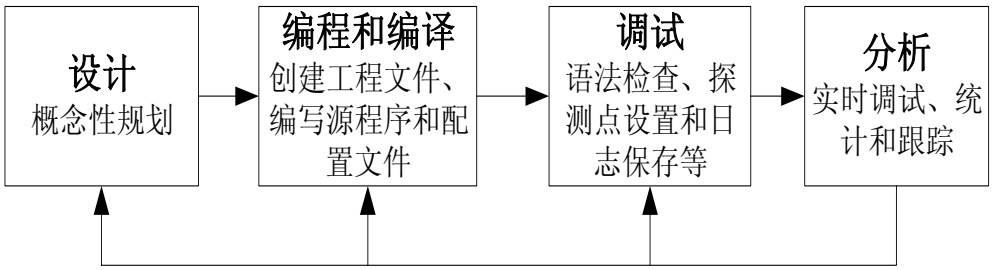
第一章　CCS 概述

本章概述 CCS（Code Composer Studio)软件开发过程、CCS 组件及 CCS 使用的文件和变量。

CCS 提供了配置、建立、调试、跟踪和分析程序的工具，它便于实时、嵌入式信号处理程序的编制和测试，它能够加速开发进程，提高工作效率。

1.1 CCS 概述

CCS 提供了基本的代码生成工具，它们具有一系列的调试、分析能力。CCS 支持如下所示的开发周期的所有阶段。



在使用本教程之前，必须完成下述工作：

- **安装目标板和驱动软件。**按照随目标板所提供的说明书安装。如果你正在用仿真器或目标板，其驱动软件已随目标板提供，你可以按产品的安装指南逐步安装。
- **安装 CCS.**遵循安装说明书安装。如果你已有 CCS 仿真器和 TMS320c54X 代码生成工具，但没有完整的 CCS，你可以按第二章和第四章所述的步骤进行安装。
- **运行 CCS 安装程序 SETUP.** 你可以按步骤执行第二章和第四章的实验。SETUP 程序允许 CCS 使用为目标板所安装的驱动程序。

CCS 包括如下各部分：

- CCS 代码生成工具：参见 1.2 节
- CCS 集成开发环境（IDE)：参见 1.3 节
- DSP/BIOS 插件程序和 API：参见 1.4 节

- RTDX 插件、主机接口和 API：参见 1.5 节

CCS 构成及接口见图 1-1。

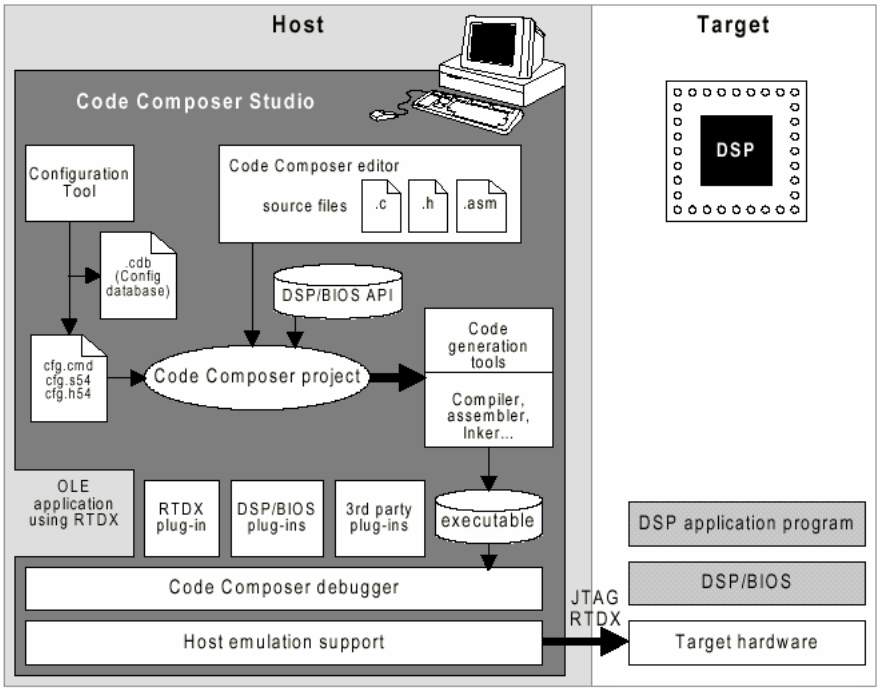


图 1-1　CCS 构成及接口

1.2 代码生成工具

代码生成工具奠定了 CCS 所提供的开发环境的基础。图 1-2 是一个典型的软件开发流程图，图中阴影部分表示通常的 C 语言开发途径，其它部分是为了强化开发过程而设置的附加功能。

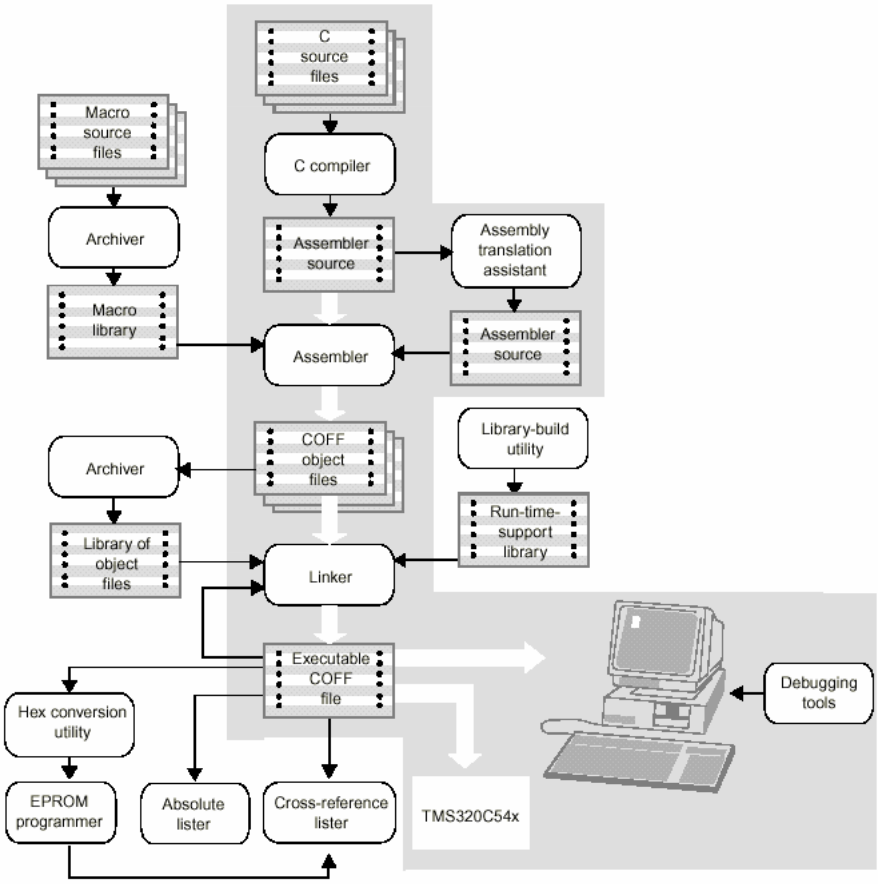


图 1-2 软件开发流程

图 1-2 描述的工具如下：

- **C 编译器(C compiler)** 产生汇编语言源代码，其细节参见 TMS320C54x 最优化 C 编译器用户指南。
- **汇编器(assembler)** 把汇编语言源文件翻译成机器语言目标文件，机器语言格式为公用目标格式（COFF），其细节参见 TMS320C54x 汇编语言工具用户指南。
- **连接器(linker)** 把多个目标文件组合成单个可执行目标模块。它

一边创建可执行模块，一边完成重定位以及决定外部参考。连接器的输入是可重定位的目标文件和目标库文件，有关连接器的细节参见 TMS320C54x 最优化 C 编译器用户指南和汇编语言工具用户指南。

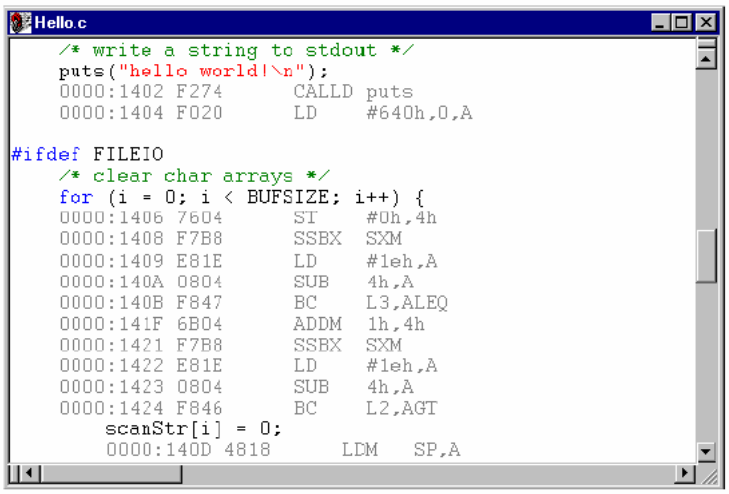
- **归档器**（archiver）允许你把一组文件收集到一个归档文件中。归档器也允许你通过删除、替换、提取或添加文件来调整库，其细节参见 TMS320C54x 汇编语言工具用户指南。
- **助记符到代数汇编语言转换公用程序**（mnimonic_to_algebric assembly translator utility）把含有助记符指令的汇编语言源文件转换成含有代数指令的汇编语言源文件，其细节参见 TMS320C54x 汇编语言工具用户指南。
- 你可以利用**建库程序**（library_build utility）建立满足你自己要求的“运行支持库”，其细节参见 TMS320C54x 最优化 C 编译器用户指南。
- **运行支持库**(run_time_support libraries) 它包括 C 编译器所支持的 ANSI 标准运行支持函数、编译器公用程序函数、浮点运算函数和 C 编译器支持的 I/O 函数，其细节参见 TMS320C54x 最优化 C 编译器用户指南。
- **十六进制转换公用程序**(hex conversion utility) 它把 COFF 目标文件转换成 TI-Tagged、ASCII-hex、 Intel、 Motorola-S、或 Tektronix 等目标格式,可以把转换好的文件下载到 EPROM 编程器中，其细节参见 TMS320C54x 汇编语言工具用户指南。
- **交叉引用列表器**（cross_reference lister）它用目标文件产生参照列表文件，可显示符号及其定义，以及符号所在的源文件，其细节参见 TMS320C54x 汇编语言工具用户指南。
- **绝对列表器**（absolute lister）它输入目标文件，输出.abs 文件，通过汇编.abs 文件可产生含有绝对地址的列表文件。如果没有绝对列表器，这些操作将需要冗长乏味的手工操作才能完成。

1.3 CCS 集成开发环境

CCS集成开发环境（IDE）允许编辑、编译和调试DSP目标程序。

1.3.1 编辑源程序

CCS允许编辑C源程序和汇编语言源程序，你还可以在C语句后面显示汇编指令的方式来查看C源程序。

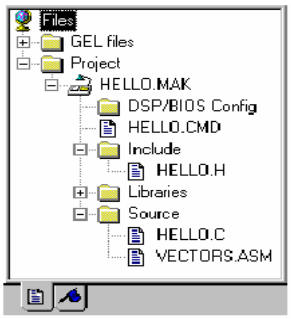


集成编辑环境支持下述功能：

- 用彩色加亮关键字、注释和字符串。
- 以圆括弧或大括弧标记C程序块，查找匹配块或下一个圆括弧或大括弧。
- 在一个或多个文件中查找和替代字符串，能够实现快速搜索。
- 取消和重复多个动作。
- 获得“上下文相关”的帮助。
- 用户定制的键盘命令分配。

1.3.2 创建应用程序

应用程序通过工程文件来创建。工程文件中包括C源程序、汇编源程序、目标文件、库文件、连接命令文件和包含文件。编译、汇编和连接文件时，可以分别指定它们的选项。在CCS中，可以选择完全编译或增量编译，可以编译单个文件，也可以扫描出工程文件的全部包含文件从属树，也可以利用传统的makefiles文件编译。



1.3.3 调试应用程序

CCS提供下列调试功能：

- 设置可选择步数的断点
- 在断点处自动更新窗口
- 查看变量
- 观察和编辑存储器和寄存器
- 观察调用堆栈
- 对流向目标系统或从目标系统流出的数据采用探针工具观察，并收集存储器映象
- 绘制选定对象的信号曲线
- 估算执行统计数据
- 观察反汇编指令和C指令

CCS提供GEL语言，它允许开发者向CCS菜单中添加功能。

1.4 DSP/BIOS 插件

在软件开发周期的分析阶段，调试依赖于时间的例程时，传统调试方法效率低下。

DSP/BIOS插件支持实时分析，它们可用于探测、跟踪和监视具有实时性要求的应用例程，下图显示了一个执行了多个线程的应用例程时序。

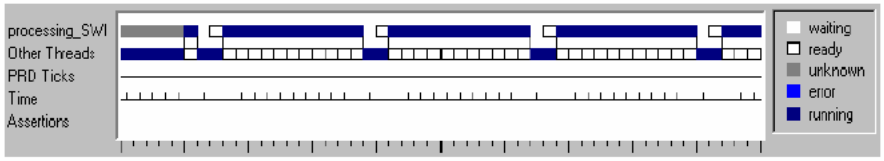


图1-3　应用例程中各线程时序

DSP/BIOS API 具有下列实时分析功能：

- **程序跟踪**（Program tracing）显示写入目标系统日志（target log）的事件，反映程序执行过程中的动态控制流。
- **性能监视**（Performance monitoring）跟踪反映目标系统资源利用情况的统计表，诸如处理器负荷和线程时序。
- **文件流**（File streaming）把常驻目标系统的I/O对象捆绑成主机文档。

DSP/BIOS 也提供基于优先权的调度函数，它支持函数和多优先权线程的周期性执行。

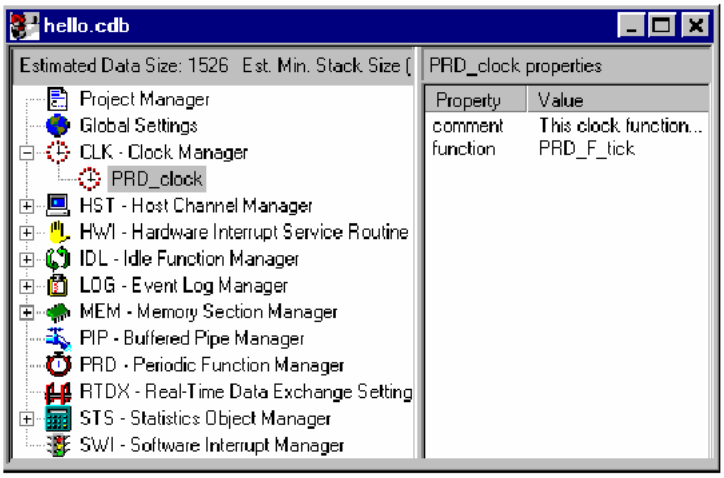
1.4.1 DSP/BIOS 配置

在CCS环境中，可以利用DSP/BIOS API定义的对象创建配置文件，这类文件简化了存储器映象和硬件ISR矢量映象，所以，即使不使用DSP/BIOS API时，也可以使用配置文件。

配置文件有两个任务：

- 设置全局运行参数。
- 可视化创建和设置运行对象属性，这些运行对象由目标系统应用程序的DSP/BIOS API函数调用，它们包括软中断，I/O管道和事件日志。

在CCS中打开一个配置文件时，其显示窗口如下：



DSP/BIOS对象是静态配置的，并限制在可执行程序空间范围内，而运行时创建对象的API调用需要目标系统额外的开销（尤其是代码空间）。静态配置策略通过去除运行代码能够使目标程序存储空间最小化，能够优化内部数据结构，在程序执行之前能够通过确认对象所有权来及早地检测出错误。

保存配置文件时将产生若干个与应用程序联系在一起的文件，这些文件的细节参见1.7.2。

1.4.2 DSP/BIOS API 模块

传统调试（debuging）相对于正在执行的程序而言是外部的，而DSP/BIOS API 要求将目标系统程序和特定的 DSP/BIOS API 模块连接在一起。通过在配置文件中定义 DSP/BIOS 对象，一个应用程序可以使用一个或多个 DSP/BIOS 模块。在源代码中，这些对象声明为外部的，并调用 DSP/BIOS API 功能。

每个 DSP/BIOS 模块都有一个单独的 C 头文件或汇编宏文件，它们可以包含在应用程序源文件中，这样能够使应用程序代码最小化。

为了尽量少地占用目标系统资源，必须优化（C 和汇编源程序）DSP/BIOS API 调用。

DSP/BIOS API 划分为下列模块，模块内的任何 API 调用均以下述代码开头。

- CLK。片内定时器模块控制片内定时器并提供高精度的 32 位实时逻

辑时钟，它能够控制中断的速度，使之快则可达单指令周期时间，慢则需若干毫秒或更长时间。

- HST。主机输入/输出模块管理主机通道对象，它允许应用程序在目标系统和主机之间交流数据。主机通道通过静态配置为输入或输出。
- HWI。硬件中断模块提供对硬件中断服务例程的支持，可在配置文件中指定当硬件中断发生时需要运行的函数。
- IDL。休眠功能模块管理休眠函数，休眠函数在目标系统程序没有更高优先权的函数运行时启动。
- LOG。日志模块管理 LOG 对象，LOG 对象在目标系统程序执行时实时捕捉事件。开发者可以使用系统日志或定义自己的日志，并在 CCS 中利用它实时浏览讯息。
- MEM。存储器模块允许指定存放目标程序的代码和数据所需的存储器段。
- PIP。数据通道模块管理数据通道，它被用来缓存输入和输出数据流。这些数据通道提供一致的软件数据结构，可以使用它们驱动 DSP 和其它实时外围设备之间的 I/O 通道。
- PRD。周期函数模块管理周期对象，它触发应用程序的周期性执行。周期对象的执行速率可由时钟模块控制或 PRD__tick 的规则调用来管理，而这些函数的周期性执行通常是为了响应发送或接收数据流的外围设备的硬件中断。
- RTDX。实时数据交换允许数据在主机和目标系统之间实时交换，在主机上使用自动 OLE 的客户都可对数据进行实时显示和分析，详细资料参见 1.5。
- STS。统计模块管理统计累积器，在程序运行时，它存储关键统计数据并能通过 CCS 浏览这些统计数据。
- SWI。软件中断模块管理软件中断。软件中断与硬件中断服务例程（ISRs）相似。当目标程序通过 API 调用发送 SWI 对象时，SWI 模块安排相应函数的执行。软件中断可以有高达 15 级的优先级，但这些优先级都低于硬件中断的优先级。
- TRC。跟踪模块管理一套跟踪控制比特，它们通过事件日志和统计累积器控制程序信息的实时捕捉。如果不存在 TRC 对象，则在配置文件中就无跟踪模块。

有关各模块的详细资料，可参见 CCS 中的在线帮助，或 TMS320C54× DSP/BIOS 用户指南。

1.5 硬件仿真和实时数据交换

TI DSPs提供在片仿真支持，它使得CCS能够控制程序的执行，实时监视程序运行。增强型JTAG连接提供了对在片仿真的支持，它是一种可与任意DSP系统相连的低侵扰式的连接。仿真接口提供主机一侧的JTAG连接，如TI XSD510。为方便起见，评估板提供在板JTAG仿真接口。

在片仿真硬件提供多种功能：

- DSP的启动、停止或复位功能
- 向DSP下载代码或数据
- 检查DSP的寄存器或存储器
- 硬件指令或依赖于数据的断点
- 包括周期的精确计算在内的多种记数能力
- 主机和DSP之间的实时数据交换（RTDX）

CCS提供在片能力的嵌入式支持；另外，RTDX通过主机和DSP　APIs提供主机和DSP之间的双向实时数据交换，它能够使开发者实时连续地观察到DSP应用的实际工作方式。在目标系统应用程序运行时，RTDX也允许开发者在主机和DSP设备之间传送数据，而且这些数据可以在使用自动OLE的客户机上实时显示和分析，从而缩短研发时间。

RTDX由目标系统和主机两部分组成。小的RTDX库函数在目标系统DSP上运行。开发者通过调用RTDX软件库的API函数将数据输入或输出目标系统的DSP，库函数通过在片仿真硬件和增强型JTAG接口将数据输入或输出主机平台，数据在DSP应用程序运行时实时传送给主机。

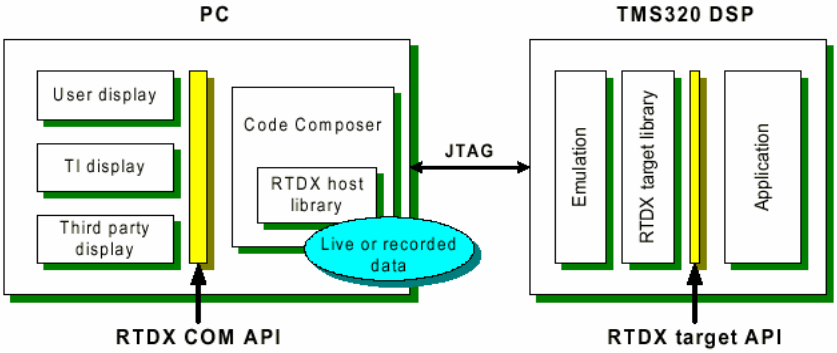


图1-4　RTDX系统组成

在主机平台上，RTDX库函数与CCS一道协同工作。显示和分析工具可以通过COM API与RTDX通信，从而获取目标系统数据，或将数据发送给DSP应用例程。开发者可以使用标准的显示软件包，诸如National Instruments' LabVIEW，Quinn-Curtis' Real-Time Graphics Tools，或Microsoft Excel。同时，开发者也可研制他们自己的Visual Basic或Visual C++应用程序。

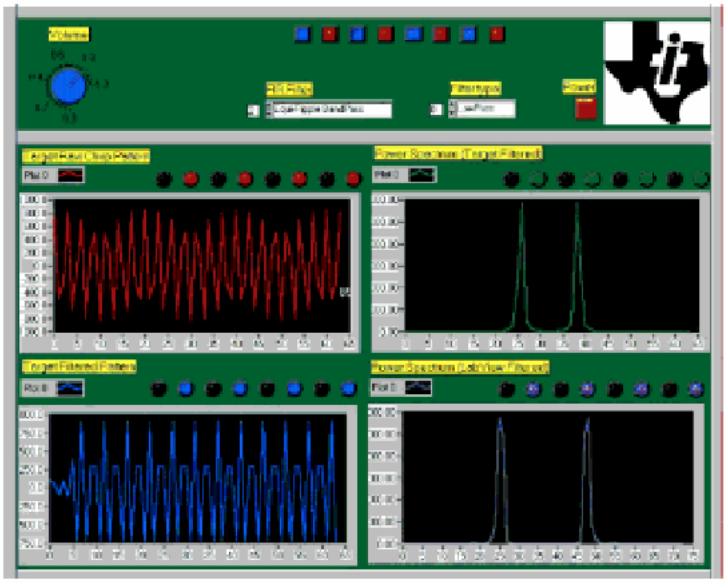


图1-5　RTDX实例

RTDX能够记录实时数据，并可将其回放用于非实时分析。下述样本由National Instruments' LabVIEW 软件产生。在目标系统上，一个原始信号通过FIR滤波器，然后与原始信号一起通过RTDX发送给主机。在主机上，LabVIEW显示屏通过RTDX COM API获取数据，并将它们显示在显示屏的左边。利用信号的功率谱可以检验目标系统中FIR滤波器是否正常工作。处理后的信号通过LabVIEW，将其功率谱显示在右上部分；目标系统的原始信号通过LabVIEW的FIR滤波器，再将其功率谱显示在右下部分。比较这两个功率谱便可确认目标系统的滤波器是否正常工作。

RTDX适合于各种控制、伺服和音频应用。例如，无线电通信产品可以通过RTDX捕捉语音合成算法的输出以检验语音应用程序的执行情况；嵌入式系统也可从RTDX获益；硬磁盘驱动设计者可以利用RTDX测试他们的应用软件，不会因不正确的信号加到伺服马达上而与驱动发生冲突；引擎控制器设计者可以利用RTDX在控制程序运行的同时分析随环境条件而变化的系数。对于这些应用，用户都可以使用可视化工具，而且可以根据需要选择信息显示方式。未来的 TI DSPs 将增加RTDX的带宽，为更多的应用提供更

强的系统可视性。关于RTDX的详细资料，请参见CCS中RTDX在线帮助。

1.6 第三方插件

第三方软件提供者可创建AxtiveX插件扩展CCS功能，目前已有若干第三方插件用于多种用途。

1.7 CCS 文件和变量

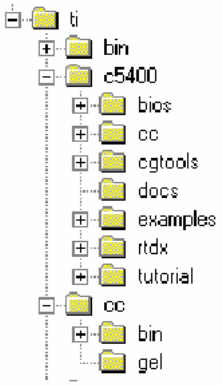
本节简述 CCS 文件夹、CCS 的文件类型及 CCS 环境变量。

1.7.1 安装文件夹

安装进程将在安装CCS的文件夹（典型情况为：c:\ti）中建立子文件夹。此外，子文件夹又建立在Windows目录下（c:\windows or c:\winnt）。

C:\ti包含以下目录：

- bin.各种应用程序
- c5400\bios。DSP/BIOS API的程序编译时使用的文件
- c5400\cgtools.Texas instruments源代码生成工具
- c5400\examples.源程序实例
- c5400\rtdx.　RTDX文件
- c5400\tutorial.本手册中使用的实例文件
- cc\bin.关于CCS环境的文件
- cc\gel.与CCS一起使用的GEL文件
- docs.PDS格式的文件和指南
- myprojects.用户文件夹



1.7.2 文件扩展名

以下目录结构被添加到Windows目录：

- ti\drivers.各种DSP板驱动文件
- ti\plugins.和CCS一起使用的插件程序
- ti\uninstall.支持卸载CCS软件的文件

当使用CCS时，你将经常遇见下述扩展名文件：

- project.mak.CCS使用的工程文件
- program.c.C程序源文件

- program.asm. 汇编程序源文件
- filename.h.C程序的头文件，包含DSP/BIOS API模块的头文件
- filename.lib.库文件
- project.cmd.连接命令文件
- program.obj.由源文件编译或汇编而得的目标文件
- program.out.（经完整的编译、汇编以及连接的）可执行文件
- project.wks. 存储环境设置信息的工作区文件，
- program.cdb.配置数据库文件。采用DSP/BIOS API的应用程序需要这类文件，对于其它应用程序则是可选的。

保存配置文件时将产生下列文件：

- ◆ programcfg.cmd.连接器命令文件
- ◆ programcfg.h54.头文件
- ◆ programcfg.s54.汇编源文件

1.7.3 环境变量

安装程序在autoexec.bat文件中定义以下变量（对Windows 95和98）或环境变量（对于Windows NT）：

表1-1　环境变量

变　量	描　　述
C54X_A_DIR	由汇编程序使用的搜索表和用于DSP/BIOS、RTDX以及代码生成工具的包含文件。可参见TMS320C54X汇编语言工具用户指南。
C54X_C_DIR	由编译程序和连接程序使用的搜索表和用于DSP/BIOS、RTDX以及代码生成工具的包含文件。可参见TMS320C54X 最佳C编译器用户指南。
PATH	添加到路径定义中的文件夹列表。缺省将添加文件夹 c:\ti\c5400\cgtools\bin 和 c:\ti\bin。

1.7.4 增加 DOS 环境空间

如果使用的是Windows 95，你可能需要增加DOS界面的环境空间，以便支持建立一个CCS应用所需的环境变量。

把下一行添加到config.sys文件中，然后重新启动计算机：

shell=c:\windows\commad.com /e:4096 /p

第二章　开发一个简单的应用程序

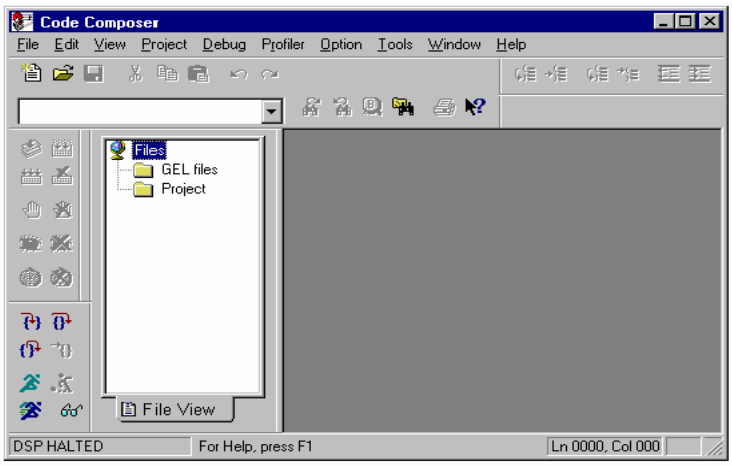
本章使用hello world实例介绍在CCS中创建、调试和测试应用程序的基本步骤；介绍CCS的主要特点，为在CCS中深入开发DSP软件奠定基础。

在使用本实例之前，你应该已经根据安装说明书完成了CCS安装。建议在使用CCS时利用目标板而不是仿真器。如果没有CCS而只有代码生成工具和Code Composer或者是利用仿真器在进行开发，你只要按第二章和第四章中的步骤执行即可。

2.1 创建工程文件

在本章中，将建立一个新的应用程序，它采用标准库函数来显示一条 hello world 消息。

1. 如果 CCS 安装在 c:\ti 中，则可在 c:\ti\myprojects 建立文件夹 hello1。（若将 CCS 安装在其它位置，则在相应位置创建文件夹 hello1。）
2. 将 c:\ti\c5400\tutorial\hello1 中的所有文件拷贝到上述新文件夹。
3. 从 Windows Start 菜单中选择 Programs→Code Composer Studio ‘C5400 →CCStudio。(或者在桌面上双击 Code Composer Studio 图标。)



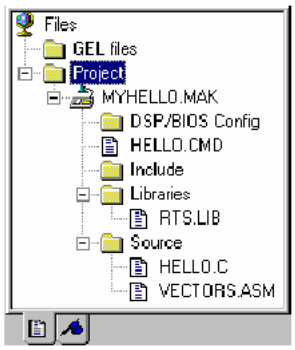
注：CCS 设置

如果第一次启动 CCS 时出现错误信息，首先确认是否已经安装了 CCS。如果利用目标板进行开发，而不是带有 CD-ROM 的仿真器，则可参看与目标板一起提供的文档以设置正确的 I/O 端口地址。

4. 选择菜单项 Project→New。
5. 在 Save New Project As 窗口中选择你所建立的工作文件夹并点击 Open。键入 myhello 作为文件名并点击 Save，CCS 就创建了 myhello.mak 的工程文件，它存储你的工程设置，并且提供对工程所使用的各种文件的引用。

2.2 向工程添加文件

1. 选择 Project→Add Files to Project，选择 hello.c 并点击 Open。
2. 选择 Project→Add Files to Project，在文件类型框中选择*.asm。选择 vector.asm 并点击 Open。该文件包含了设置跳转到该程序的 C 入口点的 RESET 中断（c_int00）所需的汇编指令。(对于更复杂的程序，可在 vector.asm 定义附加的中断矢量，或者，可用 3.1 节上所说明的 DSP/BIOS 来自动定义所有的中断矢量)
3. 选择 Project→Add Files to Project，在文件类型框中选择*.cmd。选择 hello.cmd 并点击 Open，hello.cmd 包含程序段到存储器的映射。
4. 选择 Project→Add Files to Project，进入编译库文件夹（C:\ti\c5400\cgtools\lib）。在文件类型框中选择*.o*，*.lib。选择 rts.lib 并点击 Open，该库文件对目标系统 DSP 提供运行支持。
5. 点击紧挨着 Project、Myhello.mak、Library 和 Source 旁边的符号+展开 Project 表，它称之为 Project View。



注：打开 Project View

如果看不到 Project View，则选择 View→Project。如果这时选择过 Bookmarks 图标，仍看不到 Project View，则只须再点击 Project View 底部的文件图标即可。

6. 注意包含文件还没有在 Project View 中出现。在工程的创建过程中，CCS 扫描文件间的依赖关系时将自动找出包含文件，因此不必人工地向工程中添加包含文件。在工程建立之后，包含文件自动出现在 Project View 中。

   如果需要从工程中删除文件，则只需在 Project View 中的相应文件上点击鼠标右键，并从弹出菜单中选择 Remove from project 即可。

   在编译工程文件时，CCS 按下述路径顺序搜索文件：

   - 包含源文件的目录
   - 编译器和汇编器选项的 Include Search Path 中列出的目录（从左到右）
   - 列在 C54X_C_DIR（编译器）和 C54X_A_DIR（汇编器）环境变量定

义中的目录（从左到右）。

2.3 查看源代码

1. 双击Project View中的文件hello.c，可在窗口的右半部看到源代码。
2. 如想使窗口更大一些，以便能够即时地看到更多的源代码，你可以选择Option→Font使窗口具有更小的字型。

```
/* ======== 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
}
```

当没有定义FILEIO时，采用标准puts()函数显示一条hello world消息，它只是一个简单程序。当定义了FILEIO后（见2.5节），该程序给出一个输入提示，并将输入字符串存放到一个文件中，然后从文件中读出该字符串，并把它输出到标准输出设备上。

2.4 编译和运行程序

CCS 会自动将你所作的改变保存到工程设置中。在完成上节之后，如果你退出了 CCS，则通过重新启动 CCS 和点击 Project→Open，即可返回到你刚才停止工作处。

注：重新设置目标系统 DSP

如果第一次能够启动 CCS，但接下来得到 CCS 不能初始化目标系统 DSP 的出错信息则可选择 Debug→Reset DSP 菜单项。若还不能解决上述问题，你可能需要运行你的目标板所提供的复位程序。

为了编译和运行程序，要按照以下步骤进行操作：

1. 点击工具栏按钮 或选择 Project→Rebuild All，CCS 重新编译、汇编和连接工程中的所有文件，有关此过程的信息显示在窗口底部的信息框中。
2. 选择 File→Load Program，选择刚重新编译过的程序 myhello.out(它应该在 c:\ti\myprojects\hello1 文件夹中，除非你把 CCS 安装在别的地方)并点击 Open。CCS 把程序加载到目标系统 DSP 上，并打开 Dis_Assembly 窗口，该窗口显示反汇编指令。(注意，CCS 还会自动打开窗口底部一个 标有 Stdout 的区域，该区域用以显示程序送往 Stdout 的输出。)
3. 点击 Dis_Assembly 窗口中一条汇编指令（点击指令，而不是点击指令的地址或空白区域）。按 F_1 键。CCS 将搜索有关那条指令的帮助信息。这是一种获得关于不熟悉的汇编指令的帮助信息的好方法。
4. 点击工具栏按钮 或选择 Debug→Run。

注：屏幕尺寸和设置

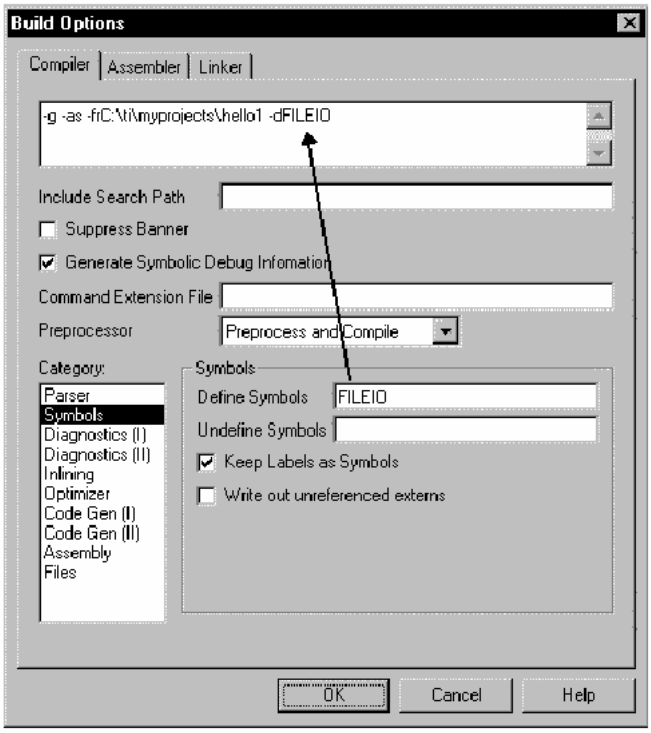
工具栏有些部分可能被Build窗口隐藏起来，这取决于屏幕尺寸和设置。为了看到整个工具栏，请在Build窗口中点击右键并取消Allow Docking选择。

当运行程序时，可在Stdout窗口中看到hello world消息。

2.5 修改程序选项和纠正语法错误

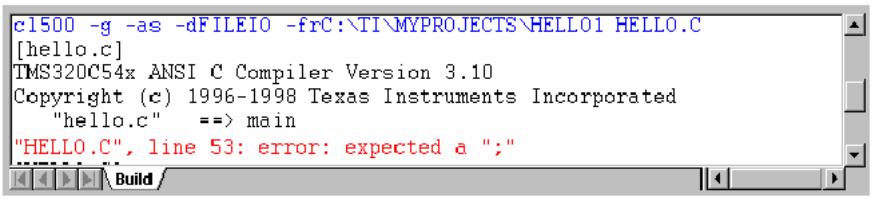
在前一节中，由于没有定义 FILEIO，预处理器命令（#ifdef 和#endif）之间的程序没有运行。在本节中，使用 CCS 设置一个预处理器选项，并找出和纠正语法错误。

1. 选择 Project→Options。
2. 从 Build Option 窗口的 Compiler 栏的 Category 列表中选择 Symbles。在 Define Symbles 框中键入 FILEIO 并按 Tab 键。
   注意，现在窗口顶部的编译命令包含-d 选项，当你重新编译该程序时，程序中#ifdef FILEIO 语句后的源代码就包含在内了。（其它选项可以是变化的，这取决于正在使用的 DSP 板。）



3. 点击 OK 保存新的选项设置。
4. 点击(Rebuild All)工具栏按钮或　　选择 Project → Rebuild All。无论何时，只要工程选项改变，就必须重新编译所有文件。
5. 出现一条说明程序含有编译错误的消息，点击 Cancel。在 Build tab 区

域移动滚动条，就可看到一条语法出错信息。



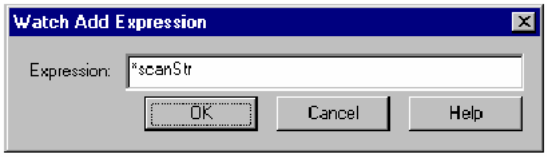
6. 双击描述语法错误位置的红色文字。注意到 hello.c 源文件是打开的，光标会落在该行上：　　fileStr[i] = 0
7. 修改语法错误（缺少分号）。注意，紧挨着编辑窗口题目栏的文件名旁出现一个星号（*），表明源代码已被修改过。当文件被保存时，星号随之消失。
8. 选择 File→Save 或按 Ctrl+S 可将所作的改变存入 hello.c。
9. 点击(Incremental Build)工具栏按钮或　　选择 Project→Build，CCS 重新编译已被更新的文件。

2.6 使用断点和观察窗口

当开发和测试程序时，常常需要在程序执行过程中检查变量的值。在本节中，可用断点和观察窗口来观察这些值。程序执行到断点后，还可以使用单步执行命令。

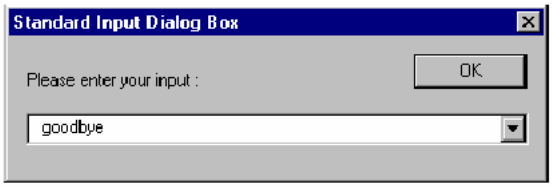
1. 选择 File→Reload Program.
2. 双击 Project View 中的文件 hello.c。可以加大窗口，以便能看到更多的源代码。
3. 把光标放到以下行上：

   fprintf(fptr, “%S”, scacStr);
4. 点击工具栏按 钮或按 F9，该行显示为高亮紫红色。（如果愿意的话，可通过 Option→Color 改变颜色。）
5. 选择 View→Watch Window。CCS 窗口的右下角会出现一个独立区域，在程序运行时，该区域将显示被观察变量的值。
6. 在 Watch Window 区域中点击鼠标右键，从弹出的表中选择 Insert New Expression。



7. 键入表达式*scanStr 并点击 OK。
8. 注意局部变量*scanStr 被列在 Watch window 中，但由于程序当前并未执行到该变量的 main()函数，因此没有定义。

9. 选择 Debug→Run 或按 F5。
10. 在相应提示下，键入 goodbye 并点击 OK。注意，Stdout 框以蓝色显示输入的文字。



还应注意，Watch Window 中显示出*scanStr 的值。

在键入一个输入字符串之后，程序运行并在断点处停止。程序中将要

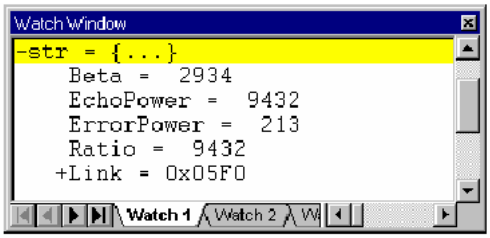
执行的下一行以黄色加亮。

11. 点击(Step Over)工具栏按钮或　　按 F10 以便执行到所调用的函数 fprintf()之后。
12. 用 CCS 提供的 step 命令试验：
    - Step Into (F2)
    - Step over (F10)
    - Step Out (Shift F7)
    - Run to Cursor (Ctrl F10)
13. 点击工具栏按钮　　或按 F5 运行程序到结束。

2.7 使用观察窗口观察 structure 变量

观察窗除了观察简单变量的值以外，还可观察结构中各元素元素的值。

1. 在 watch Window 区域中点击鼠标右键，并从弹出表中选择 Insert New Expression。
2. 键入 str 作为表达式并点击 OK。显示着+str={...}的一行出现在 Watch Window 中。+符号表示这是一个结构。回顾 2.3，类型为 PARMS 的结构被声明为全局变量，并在 hello.c 中初始化。结构类型在 hello.h 中定义。
3. 点击符号＋。CCS 展开这一行，列出该结构的所有元素以及它们的值。



4. 双击结构中的任意元素就可打开该元素的 Edit Variable 窗口。
5. 改变变量的值并点击 OK。注意 Watch Window 中的值改变了，而且其颜色也相应变化，表明已经该值已经人工修改了。
6. 在 Watch Window 中选择 str 变量并点击右键，从弹出表中选择 Remove Cuurent Expression。在 Watch Window 中重复上述步骤。
7. 在 Watch Window 中点击右键，从弹出表中选择 Hide 可以隐藏观察窗口。
8. 选择 Debug→Breakpoits。在 Breakpoints tab 中点击 Delete All，然后点击 OK，全部断点都被清除。

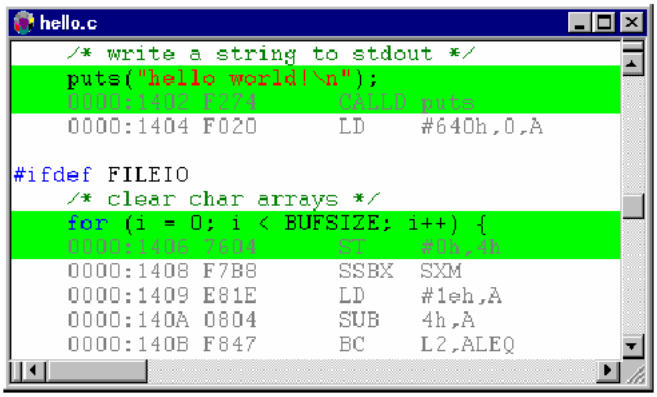
2.8 测算源代码执行时间

在本节中，将使用 CCS 的 profiling 功能来统计标准 puts()函数的执行情况，可以把这些结果与 3.4 节中采用 DSP/BIOS API 显示 hello world 消息的相应结果相比较。

1. 选择 File→Reload Program。
2. 选择 Profiler→Enable Clock。标记“√”出现在 Profile 菜单 Enable Clock 项的旁边，该选项使能就可计算指令周期。
3. 在 Project View 中双击文件 hello.c.
4. 选择 View→Mixed Source/ASM，灰色的汇编指令紧随在 C 源代码行后面。
5. 将光标放在下述行上：

   puts(“hello world!\n”);
6. 点击工具栏按钮 (Toggle Profile_point)，该 C 源代码行和第一条汇编指令被用绿色加亮。
7. 向下移动滚动条，将光标停在以下行上：

   for (i = 0; i<BUFSIZE;i++);{
8. 点击工具栏按钮 或者在该代码行上点击右键并从弹出菜单中选择 Toggle Profile Pt。



有关测试点的统计数据报告显示自前一个测试点或程序开始运行以来到本测试点所需的指令周期数。本例中，第二个测试点的统计数据报告显示自 puts()开始执行到该测试点所需的指令周期数。

9. 选择Profile→View Statistics，窗口底部出现一个显示测试点统计数据的区域。

10. 通过拖拽该区域的边缘可调整其大小。

Location	Count	Average	Total	Maximum	Minimum
HELLO.C line 47	0	0.0	0	0	0
HELLO.C line 51	0	0.0	0	0	0

注：上图中的line数可能会不同

本手册中屏幕上所显示的line数可能会和当前所使用的软件版本显示的line数不同。

11. 点击(RUN)工具栏按钮　　或按F5键运行该程序并在提示窗口中键入一串字符。
12. 注意对第二个测试点所显示的指令周期数，它应该大约为2800个周期（显示的实际数目可能会变化），这是执行puts()函数所需的指令周期数。由于这些指令只执行了一次，所以平均值、总数、最大值和最小值都是相同的。

Profile Statistics

Location	Count	Average	Total	Maximum	Minimum
HELLO.C line 47	1	931.0	931	931	931
HELLO.C line 51	1	2818.0	2818	2818	2818

注：目标系统在测试点处于暂停状态

只要程序运行到一个测试点，它就会自动暂停。所以，当使用测试点时，目标系统应用程序可能不能满足实时期限的要求。(用RTDX则可能实现实时监控，这可参见1.5节。)

13. 在进入下一章之前（完成2.9节以后），执行以下步骤释放测试期间所占用的资源：
    - 进入profiler菜单并撤消 Enable Clock使能。
    - 点击鼠标右键从弹出菜单中选择Hide从而关闭Profile Statistcs窗口。
    - 进入profiler→profile_points，　选择Dlete All并点击OK。
    - 进入View菜单，并撤消 Mixed Source/ASM使能。

2.9 进一步探索

为了进一步探究 CCS，可作如下尝试：

- 在 Build Option 窗口中，检查与编译器、汇编器和连接器有关的域，注意这些域中值的变化是怎样影响所显示的命令行的，可在 CCS 中参见在线帮助了解各种命令行开关。
- 设置某些断点。选择 Debug→Breakpoints，注意在 Breakpoints 输入框中可以设置条件断点，只有当表达式的值为真时，程序才会在断点处暂停。也可以设置各种硬件断点。

2.10 进一步学习

为了掌握关于使用 CCS 的更多的技巧，可参见有关 CCS 的在线帮助或 CCS 用户指南（PDF 格式）。

第三章　开发 DSP/BIOS 程序

本章通过使用 DSP/BIOS 优化第二章中的 hello world 实例介绍 DSP/BIOS 及如何创建、编译、调试和测试使用 DSP/BIOS 编写的程序。

基本要求：CCS 的 DSP/BIOS 组件，目标板。

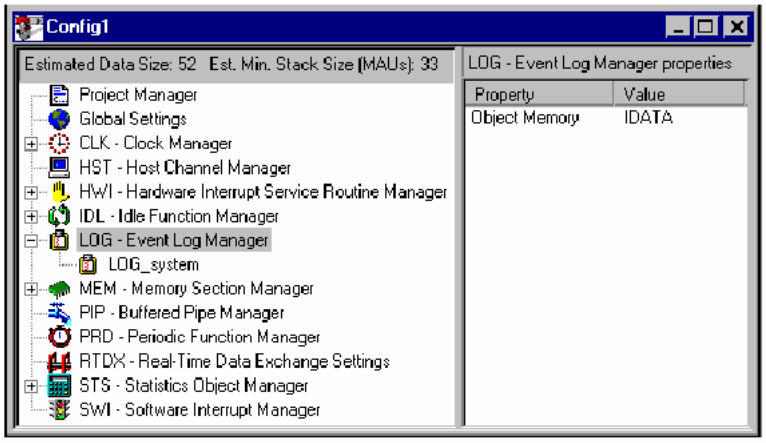
3.1 创建配置文件

实现 hello world 程序的另一种方法是使用 DSP/BIOS API 的 LOG 模块，它能在嵌入式程序中提供基本运行服务。对于实时 DSP 上的应用而言,API 模块是最优的。与诸如 put()这样的 C 库函数调用不同，API 无需中止目标板中运行的应用程序就能进行实时分析。此外，API 代码比标准 C 库函数的 I/O 占用空间少且运行快， 根据程序需要可使用一个或多个 DSP/BIOS 模块。

本章使用 DSP/BIOS API 修改第二章中的应用程序（如果要跳过第二章，则须从 2.1 和 2.2 节开始）。

在使用 DSP/BIOS API 的程序中必须创建一个配置文件，它定义了程序中使用的所有 DSP/BIOS 对象。本节介绍如何创建配置文件。

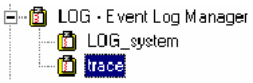
1. 如果已经关闭了 CCS，则重新开始。选择 Project→Open 重新打开 c:\ti\myprojects\hello1 文件夹中的 myhello.mak 项目（如你安装其它地方，则在所安装的地方打开含有 myprojects 的文件夹。）



2. 选择 File→New→DSP/BIOS Config，弹出一个含有“c54xx.cdb”和

“sd54.cdb”的窗口。

3. 在此窗口中选择与你的系统板相适应的 DSP 模板，然后点击 OK（TMS320C54X DSP/BIOS 用户指南阐述了怎样创建一个用户模板），将出现上面这样一个窗口，点击左边的+和-字符能扩张和收缩列表单，窗口右边显示窗口左边选中对象的属性。
4. 在 LOG-Event Log Manager 处点击鼠标右键，从弹出菜单中选择 Insert LOG， 这时创建一个名为 LOG0 的 LOG 对象。
5. 在 LOG0 处点击鼠标右键，从弹出菜单中选择 Rename，键入 trace 即改变此对象名称为 trace。

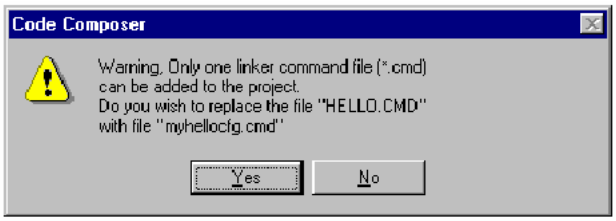


6. 选择 File→Save。在弹出窗口中选择你的工作路径（通常是 c:\ti\myprojects\hello 1），并将此配置保存为 myhello.cdb，实际上创建了下述文件：
   - **myhello.cdb**　　保存配置设置
   - **myhellocfg.cmd**　连接命令文件
   - **myhellocfg.s54**　汇编语言源文件
   - **myhellocfg.h54**　由 myhellocfg.h54 包含的汇编语言头文件

3.2 向工程添加 DSP/BIOS 文件

回顾上节所建立的配置文件，它实际上包括四个新文件 myhello.cdb、myhellocfg.cmd、myhellocfg.s54、myhellocfg.h54。本节介绍如何向工程添加这些文件并删除被取代的文件。

1. 选择 Project→Add Files to Project　在弹出窗口的文件类型框中选择配置文件（*.cdb），然后选择 myhello.cdb 并点击 Open。注意此时在 Project View 中的 DSP/BIOS Config 文件夹下面包含配置文件 myhello.cdb。另外，myhellocfg.s54 作为源文件出现在 source 文件夹中。注意在编译工程文件的过程中，CCS 在扫描文件间的依赖关系时自动向工程中添加包含文件（在此添加的是 myhellocfg.h54）。
2. 输出文件名必须与.cdb 文件名匹配（myhello.out 和 myhello.cdb）。选择 Project→Options 将出现 Build Option 窗口，然后选择 Linker，在 Output Filename 栏中确认输出文件名为 myhello.out，点击 OK。
3. 再次选择 Project→Add Files to Project，在弹出窗口的文件类型栏中选择 Linker Command File(*.cmd)，再选择文件 myhellocfg.cmd 并点击 Open，随之产生如下消息框：



4. 点击 Yes，则加入新生成的配置文件 myhellocfg.cmd 并取代 hello.cmd。
5. 在 Project View 中的 vectors.asm 源文件上点击鼠标右键，然后从弹出菜单中选择 Remove from project。DSP/BIOS 配置文件将自动定义硬中断矢量。
6. 在 RTS.lib 库文件处点击鼠标右键将它从 project 中删除。该库已经由 myhellocfg.cmd 文件自动包含。
7. 双击程序 hello.c 打开并编辑该文件，在弹出的代码框中如果显示了汇编指令，则选择 View→Mixed Source/ASM 可隐藏汇编代码。
8. 源文件中需修改的内容如下。（可以从 c:\ti\c5400\tutorial\hello2\hello.c 中复制和粘贴）由于 puts()

和 LOG_printf 使用同样的资源，你必须确保使用下面的主函数取代当前存在的主函数。

```
/* ======== hello.c ======== */
/* DSP/BIOS header files*/
#include <std.h>
#include <log.h>
/* Objects created by the Configuration Tool */
extern LOG_Obj trace;
/* ======== main ======== */
Void main()
{
LOG_printf(&trace, "hello world!");
/* fall into DSP/BIOS idle loop */
return;
}
```

9. 注意源程序中的下述几点：

（1）C 源程序中包含 std.h 和 log.h 头文件。所有使用 DSP/BIOS API 的程序都必须包含头文件 std.h 和 log.h。在 LOG 模块中头文件 log.h 定义了 LOG_Obj 的结构并阐述了 API 的功能。源代码中必须首先包含 std.h，而其余模块的顺序并不重要。

（2）源程序中声明了配置文件中创建的 LOG 对象。

（3）主函数中，通过调用 LOG_printf，将 LOG 对象的地址（&trace）和 hello world 信息传到 LOG_printf。

（4）主函数返回时，程序进入 DSP/BIOS 空循环，DSP/BIOS 在空循环中等待软中断和硬中断信号，第五、六、七章将阐述这些内容。

10. 选择 File→Save 或按 Ctrl+S 保存修改后的源程序。
11. 选择 Project→Optins，在弹出窗口中选择 Compiler，然后选择 Category 中的 Symbols，并在 define symbols 中删除 FILEIO， 然后点击 OK。
12. 点击工具栏按钮 或选择 Project→Rebuild All 。

3.3 用 CCS 测试

由于使用 LOG 的程序只写了一行，没有更多的内容需要分析。在第五、六、七章中将用更多的方法分析程序功能。

1. 选择 File→Load Program 选取 myhello.out 并点击 open。
2. 选择 Debug→Go Main。
3. 选择 Tools→DSP/BIOS→Message Log，在 CCS 窗口底部出现一个 Message Log 窗口。
4. 在 Message Log 窗口中点击鼠标右键，从弹出的菜单中选择 Property Page。
5. 选择 trace 作为监视对象，然后点击 OK。缺省的刷新频率为 1 秒。（如果要修改刷新频率，可选择 Tools→DSP/BIOS→RTA Control Panel。在 RTA Control Panel 处点击鼠标右键，选择 Property Page 并选取一个新的刷新频率，点击 OK。）
6. 选择 Debug→Run 或按 F5。 hello world 信息将出现在 Message Log 区域内。

7. 选择 Debug→Halt 或按 Shift F5 暂停程序运行。主函数返回后，程序在 DSP/BIOS 空循环中等待中断信号，欲了解空循环的更多信息，请参见 3.5 节。
8. 在 Message Log 中点击鼠标右键，然后选择 Close 关闭 Message Log。在下一节中将使用 Profiler，因此必须关闭 Message Log。
9. 选择 Tools→RTDX 启动 RTDX 插件，并从下拉的菜单中选取 RTDX disable，然后点击鼠标右键并选择 Hide。

注意：在某些目标系统中 Profiling 和 RTDX 不能同时使用。

在使用 Profiling 前，关闭使用 RTDX 的工具，如 Message Log 或其它的 DSP/BIOS 插件。特别是在使用 DSP/BIOS 插件后，必须确保 RTDX 无效，选 Tools→ RTDX 启

动 RTDX 插件，并从下拉菜单中选取 RTDX　disable，然后点击鼠标右键再选择 Hide。反之亦然，如 2.8 节所述。

当试图同时使用 Profiling 和 RTDX 时将导致错误信息，见上图。

3.4 测算 DSP/BIOS 代码执行时间

LOG_printf 所需的指令周期数，可像前面的 puts()一样，利用 CCS 的功能来测算。

1. 选择 File→Reload Program。
2. 选择 Profiler→Enable Clock，在 Enable Clock 旁边可见到√。
3. 在 Project View 中，双击 hello.c 文件。
4. 选择 View→Mixed Source/ASM，则灰色的汇编指令紧随 C 源程序。
5. 将光标放在 LOG_printf(&trace, "hello world!")行上。
6. 点击工具栏按钮 (Toggle Profile-point)，则这一行和其下一行的汇编指令变为绿色高亮显示。
7. 向下移动滚动条，把光标放在程序结尾的大括号所在的行上，然后点击工具栏按钮 (Toggle Profile-point)，你可能会认为在程序的 return 行上设置了第二个测试点。但是，要注意直到大括号后一直没有相应的汇编语言显示出来。如果在 return 这一行上设置了测试点，CCS 将在程序运行时自动纠正这一问题。
8. 选择 Project→View Statistics。
9. 点击（Run）工具栏按钮或 按 F5 运行程序。
10. 注意在第二个测试点显示的指令周期数为 58（实际中可能稍有不同），

Location	Count	Average	Total	Maximum	Minimum
HELLO.C line 33	1	58.0	58	58	58
HELLO.C line 29	1	5273.0	5273	5273	5273

这是执行 LOG_printf 函数需要的指令周期数。由于字符串的格式化在 PC 主机上、而不是在目标系统 DSP 上完成，因此调用 LOG_printf 的效率很高。LOG_printf 需要的 58 个指令周期，而在第二章结束时测试的 put()则需要 2800 个指令周期。在应用程序中调用 LOG_printf 监视系统状态对程序执行几乎没有影响。

11. 点击工具栏按钮 或按 Shift F5 暂停程序运行。
12. 在进行下一章的工作之前（3.5 节结束之后）做下述工作，释放测试时占用的资源。

    - 进入 Profiler 菜单，撤消 Enable Clock 前的“√”。

- 在 Profiler Statistics 窗口中点击鼠标右键，并从打开的菜单选择 Hide
- 选取 Profiler→Profile-points，然后选择 Delete All ，点击 OK。
- 进入 View 菜单，撤消 Mixed Source/ASM 前的“√”。
- 关闭所有的源文件和配置窗口。
- 选择 Project→Close 关闭 Project

3.5 进一步探索

为进一步了解 CCS，试作如下工作：

- 加载 myhello.out 并在 LOG_printf 行设置断点，选取 Debug→Breakpoints 并在 IDL_F_loop 上设置断点。（在弹出对话框的 Location 栏中键入 IDL_F_loop，并点击 Add）。

运行程序　在第一个断点处，使用 View→CPU Registers→CPU Registers 观察寄存器值。注意：当主函数执行时，INTM＝1 表明中断非使能。

运行到下一个断点　注意现在INTM＝0，表明中断使能。注意在执行程序时将重复遇到该断点。

启动进程和主函数执行完毕后，DSP/BIOS应用程序将进入空循环的后台线程。空循环由IDL 模块管理，　直到程序暂停时才结束工作；它在中断使能有效时循环，且允许响应任一ISR中断信号，能满足实时任务处理的要求。第五、六、七章将进一步阐述ISRs和DSP/BIOS的软中断。

- 在MS-DOS窗口中，键入以下命令行可运行sectti.exe程序。如果安装路径不是c:\ti，则须将路径修改为安装了CCS 的目录路径。

```
cd c:\ti\c5400\tutorial\hello1
    sectti hello.out > hello1.prn
    cd ..\hello2
    sectti hello.out > hello2.prn
```

比较hello1.prn和 hello2.prn文件可以发现使用stdio.h和DSP/BIOS时存储器段和空间大小的差别。与使用stdio中的puts()函数相比，DSP/BIOS调用LOG_printf时.text段占用的空间小。有关sectti工具的其它信息可参见TMS320C54x DSP/BIOS 用户指南。

3.6 进一步学习

进一步学习使用CCS 和DSP/BIOS，请参见CCS中 的在线帮助，也可参见CCS 用户指南和TMS320C54x　DSP/BIOS 用户指南。

第四章　　算法和数据测试

本章说明创建和测试一个简单算法的过程并介绍CCS附加功能。

本章将创建一个完成基本信号处理的程序，并在下两章继续介绍。

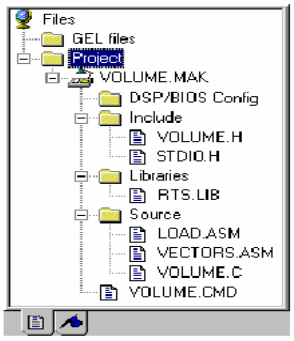
可用存储在PC机文件中的数据来建立和测试一个简单的算法，也可利用CCS的探针断点、图形显示、动态运行和GEL文件。

4.1 打开和查看工程

在CCS中打开一个工程文件并查看此工程中源文件和库文件。

1. 若CCS安装在c:\ti，那么就在c:\ti\myprojects下创建文件夹volume1（若CCS安装在其它位置，那么就在相应位置创建文件夹volume1）。
2. 将文件夹c:\ti\c5400\tutorial\volume1中的所有文件复制到新文件夹。
3. 如果CCS应用程序还未运行，则在开始菜单中选择Program→Code Composer Studio'C5400→CCStudio。
4. 选择Project→Open并在volume1中选择volume.mak文件并点击Open。
5. 由于Project已经移动，CCS将显示一个对话框指示没找到库文件。点击Browse键，按路径c:\ti\c5400\cgtools\lib找到并选中rts.lib。(如果CCS安装在其它位置，那么在安装文件夹里找c5400\cgtools\lib)。

6. 点击符号＋展开Project View，在Project中包含VOLUME.MAK、 Include、Libraries、Source。该Project中的主要文件有：
   - **volume.c**　包含main()函数的C源程序
   - **volume.h** 源程序volume.c包含的头文件，其中定义了各种常数和结构。
   - **load.asm** 此文件包含load子程序，该子程序



是一个能从C函数中调用的简单汇编循环子程序,该函数有一个入口参数、执行所需的指令周期为(31*argument）+13。

- vectors.asm 此文件在第二章使用过，它定义了DSP的中断向量表
- volume.cmd 连接命令文件，它将各段映射到存储器中。
- rts.lib　为DSP目标系统提供运行支持。

4.2 查看源程序

在Project View窗口中双击volume.c文件，源程序就显示在CCS窗口的右边。

注意实例中的下面几部分：

- 主函数打印完信息后，应用程序处于无限循环状态。在此循环中，主函数调用dataIO和processing()函数。
- processing()函数将增益与输入缓存区中的各值相乘并将结果存入输出缓存区；同时也调用汇编Load子程序，该子程序占用的指令周期取决于传递给它的processingLoad值。
- dataIO函数是一个空函数，它的作用类似于return语句。可利用CCS中的探针（Probe Point）功能把主机文件中的数据读取到inp_buffer缓存区，此法优于用C代码来完成I/O功能。

```
#include <stdio.h>
#include "volume.h"
/* Global declarations */
int inp_buffer[BUFSIZE]; /* processing data buffers */
int out_buffer[BUFSIZE];
int gain = MINGAIN; /* volume control variable */
unsigned int processingLoad = BASELOAD; /* processing load */
/* 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 using a Probe Point connected to a host file. */
dataIO();
/* 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 output signal.
* PARAMETERS: none.
* RETURN VALUE: none. */
static void dataIO()
{
/* do data I/O */
return;
}
```

4.3 为 I/O 文件增加探针断点

本节介绍探针断点(Probe Point)的使用方法。探针可以从PC机的文件中读取数据，它是开发算法的一个有效工具。其使用方法如下:

- 将来自PC主机文件中的输入数据传送到目标系统的缓存器中供算法使用。
- 将来自目标系统缓存器中的输出数据传送到PC主机的文件中供分析。
- 用数据更新窗口，如图形窗口。

与断点类似，它们都挂起目标系统来完成自己的动作，但存在如下几个方面的差别 ：

- 探针立即中止目标系统，完成一个操作后，再恢复目标系统的运行。
- 断点暂停CPU直到人工恢复其运行为止，且更新所有打开的窗口。
- 探针允许自动执行文件的输入或输出，而断点则不行。

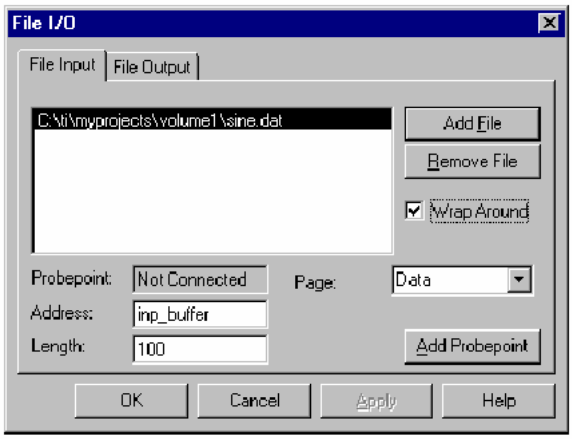
本章介绍如何利用探针把PC机文件内容传送到目标系统中作为测试数据使用。当到达探测点时，同时使用断点更新所有打开的窗口，这些窗口包括输入和输出数据的图形窗口。第七章将阐述管理输入和输出数据流的两种方法。

1. 点击工具栏按钮　　或选择Project→Rebuild All 。
2. 选择File→Lode Program并选取volume.out ，然后点击Open。
3. 在Project View窗口中，双击volume.c文件。
4. 将光标置于主函数中的 dataIO( )这一行上。DataIO函数起占位符作用。现在，它是一个很好的与探针断点相连接的地方，以便于从PC机文件输入数据。
5. 点击工具栏按钮　　(Toggle Probe Point)，则光标所在行变为兰色高亮。
6. 选择File→File I/O,在File I/O 对话窗中可选择输入和输出文件。
7. 在File Input 栏中，点击Add File。
8. 选择sine.dat 文件。

注意：在文件类型框中可以选择数据格式，sine.dat 文件包含正弦波形的16进制值。

9. 点击Open，将该文件添加到File I/O对话框的列表上，接着出现sine.dat文件控制窗口（CCS窗口可以覆盖它）。在运行程序时，可用这个窗口开始、停止、重复、或快速前进来控制数据文件。

10. 在File I/O对话框中，将Address修改为 inp_buffer，Length修改为100，选中Wrap Around。



- Address栏中的值指定来自文件的数据将要存放的位置，inp_buffer是由volume.c文件声明为BUFSIZE的整数数组。
- Length栏中的值指定每次探针到达时读入多少个数据样点，使用100是因为BUFSIZE常数已由volume.h(0x64)设置为100。
- 当探针到达文件结尾时，Wrap Around选项使CCS从文件的开始读数据。即使数据样点只含有1000个值且每次探针到达时读取100个值，也可将数据看作连续的数据流。

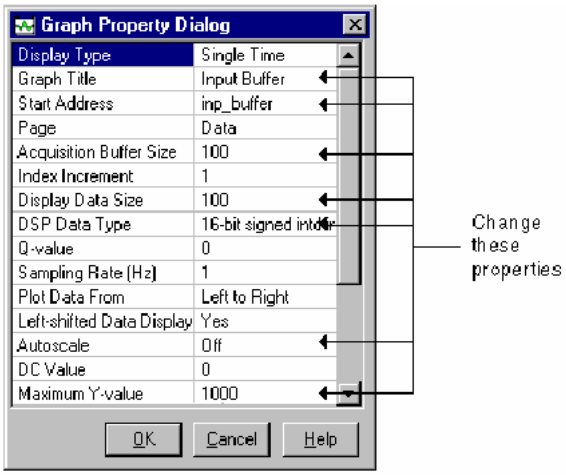
11. 点击Add Probe Point，Break\Probe\Profile Points 对话窗的Probe Point栏就会出现。
12. 加亮（对话框中）显示的第五步的断点设置。
13. 点击Connect栏尾处的下箭头，在其下拉菜单中选择sine.dat 文件。
14. 点击Replace。Probe Point列表将显示探测点已连接到sine.dat 文件。
15. 点击OK。 File I/O对话框则显示文件现已被连接到探测点。
16. 在File I/O对话框，点击OK。

4.4 显示图形

如果现在就运行程序的话，你将无法了解到更多的程序运行时的信息。可以在 inp_buffer 和 out_buffer数组的地址范围内设置观察变量，但需要设置很多变量，而且是按数字形式显示而非图形形式。

CCS 提供了多种用图形处理数据的方法。在本例中，你将看到一个基于时间绘制的图形。本节介绍图形的打开，下节介绍程序的运行。

1. 选择View→Graph→Time/Frequency。
2. 在弹出的Graph Property Dialog对话窗中， 将 Graph Title，Start Address，Acquisition Buffer Size，Display Data Size，DSP Data Type，Autoscale和Maximum Y-value 的属性改变为如下图所示。向下滚动或调整 dialog 框的大小可看到所有的属性。

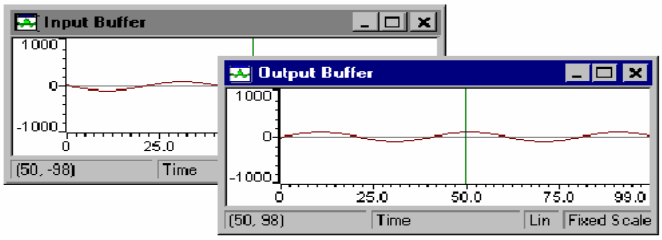


3. 点击OK，出现输入缓存的一个图形窗。
4. 在上述窗中右击鼠标，从弹出的菜单中选择Clear Display。
5. 再次选择View→Graph→Time/Frequency。
6. 改变Graph Title的属性为Output Buffer，改变Start Address的属性为Out_buffer，其余的设置都不变。
7. 点击OK，又出现一个图形窗，在该图形窗内右击鼠标，从弹出的菜单中选择Clear Display。

4.5 执行程序和绘制图形

到目前为止，你已经放置好了一个探针，它可临时暂停目标系统，将数据从PC主机传送到目标系统，并将恢复目标系统应用程序的运行。但是，探针不能刷新图形。在本节中，将设置一个可刷新图形的断点，使用Animate命令在遇到断点后，自动恢复目标系统应用程序的运行。

1. 在C源程序volume.c窗口中，将光标放置在dataIO行。
2. 点击（Toggle　Breakpoint）工具栏按钮　或按F9，该行显示为红色和蓝色高亮（除非用Option→Color改成其它颜色），表明在这一行已经设置了断点和探针。在同一行上既放置探针又放置断点，它能够使目标系统只暂停一次而完成两个操作：数据传输和图形刷新。
3. 重新安排窗口以便能同时能看到这两个图形。
4. 点击工具栏按钮　或按F12运行程序。Animate命令与Run命令相似，它使目标系统应用程序一直运行到断点，随后，目标系统应用程序暂停并刷新窗口。但是，与Run命令不同的是，Animate命令恢复目标系统应用程序运行到下一个断点，而且此过程是连续的，直到目标系统被人工停止。所以，Animate命令可看作运行－中断－继续（run-break-continue）过程。
5. 注意每个图形包含2.5个周期的正弦波形，且在程序运行过程中两个图形反向。每次到达探测点时，CCS 从sine.dat文件中得到100个值，再将这100个值写入inp_buffer地址。符号相反是因为输入缓冲区包含的值是从sine.dat文件中读取的，而输出缓冲区最后的值是经过函数处理后得到的。



注意：目标系统运行程序在探测点暂停

不管何时到达探测点，CCS都能暂时停止目标系统。因此，如果使用了探针，目标系统应用程序就不能满足实时期限的要求。在本开发阶段只测试算法，以后将用RTDX和DSP/BOIS分析实时性能。

只使用测试点和Run命令也能刷新图形，4.10节将详细介绍。

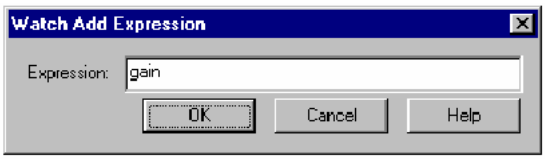
4.6 调节增益

回顾 4.2 节，processing 函数将增益与输入缓存区中的各值相乘并将结果存入输出缓存区；在一个 While 循环中用如下语句完成此功能。

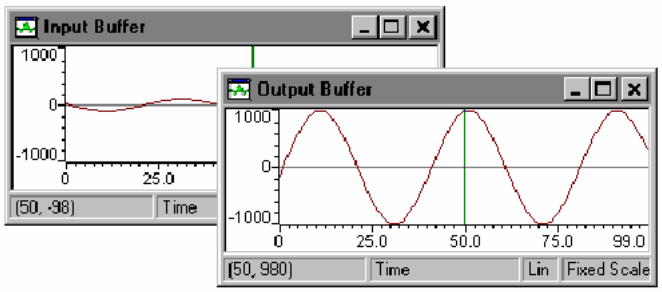
```
*output++ = *input++ * gain;
```

该语句将增益与输入缓存区中的各值相乘并将结果存放在 out_buffer 中相应的位置上。gain 初始化为 MINGAIN，而 MINGAIN 已经在 volume.h 中定义为 1。要改变输出幅度得修改 gain 值。修改 gain 值的一种方法是采用观察变量。

1. 选择 view→Watch Window
2. 在 Watch 窗中右击鼠标，并从弹出菜单中选择 Insert New Expression。
3. 在 Expression 区敲入 gain 并单击 OK，变量值出现在 Watch 窗口。



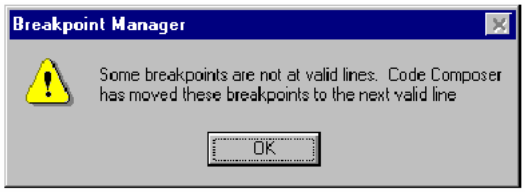
4. 如果程序已暂停，点击工具栏按钮 (Aminate)，重新开始运行程序。
5. 在 Watch 窗口中双击 gain。
6. 在 Edit Variable 窗口中修改 gain 值为 10，并点击 OK。
7. 注意在 Output Buffer 图形中信号幅度变化反映了增益的提高。



4.7 观察范围外变量

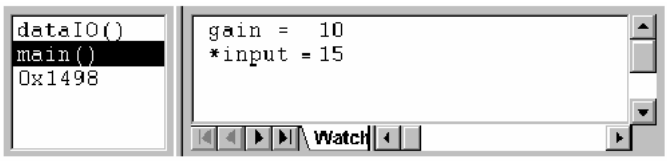
你曾使用 Watch Window 观察过变量并改变变量的值。但当你想查看的变量的作用域不在当前设置的断点范围内时，则可使用访问堆栈来查看。

1. 点击工具栏按钮　　或按 Shift F5 暂停程序运行。
2. 用 CCS 重新查看 volume.c 程序。注意在主函数与 processing 函数中已经定义了变量*input，但它没有在数据输入输出函数中定义。
3. 在显示源程序 volume.c 的窗口中，把光标放置在 dataIO()函数的 return 行上。
4. 点击工具栏按钮　　或按 F9，该行变为红色高亮显示（除非你用 Open→Color 改变颜色）。
5. 点击 F5 执行程序，CCS 将自动把断点移到相应函数的下一条汇编指令处。对话框通知你将把断点移到下一行。



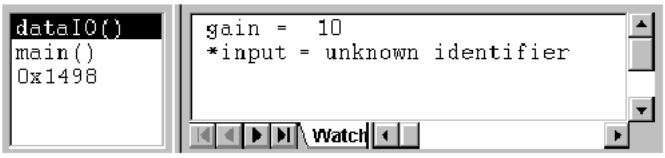
6. 点击 OK
7. 按下 F5，程序运行直到 dataIO()函数末尾的断点处暂停。
8. 在 Watch 窗口中点击鼠标右键并从弹出菜单中选择 Insert New Expression。
9. 在 Expression 区键入*input 并点击 OK。

10. 注意在 Watch 窗口中显示该变量是一个未知符号，这说明*input 没有在 dataIO()函数内定义。



11. 选择 View→Call Stack，则可看到相邻的堆栈窗与观察窗。

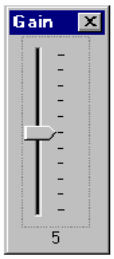
12. 在堆栈窗中点击 main()就可在主函数范围内查看*input 的值。



13. 可以点击堆栈窗底部的地址以便看清楚 gain 是全局变量，而*input 则不是。（地址的变化依赖于正使用的 DSP）
14. 在堆栈窗中点击鼠标右键并从弹出菜单中选择 Hide。
15. 将光标放置在 datdIO()函数中的 return 之后的行上，撤消第 4 步中放置的断点，点击工具栏按钮或 单击 F9。

4.8 使用 GEL 文件

CCS 提供了修改变量的另一种方法，该方法使用一种扩展语言 GEL 来创建可修改变量的小窗口。



1. 选择 File→Load GEL。在 Load GEL File 对话框中选择 volume.gel 文件并点击 Open，这个选项是在上步加载 GEL 文件时自动增加的。
2. 选择 GEL→Application Control→Gain，弹出如右图所示的小窗口。
3. 如果程序已经暂停，点击工具栏按钮 （Animate）。注意即使在弹出的 gain 小窗口中值为零，其实 gain 的当前值并未改变。只有滑动指针时 gain 值才发生变化。
4. 在 gain 窗口中用滑动指针改变 gain 值，则在 Output Buffer 窗口中的正弦波形幅度相应改变。此外，无论任何时候移动滑动指针，在 Watch 窗口的变量 gain 的值将随之改变。
5. 点击工具栏按钮 或按下 Shift F5 暂停程序运行。
6. 为了了解 Gain GEL 函数的工作情况，点击 Project View 中 GEL 文件前的 +符号，然后在 VOLUME.GEL 文件上双击鼠标便可查看其内容：

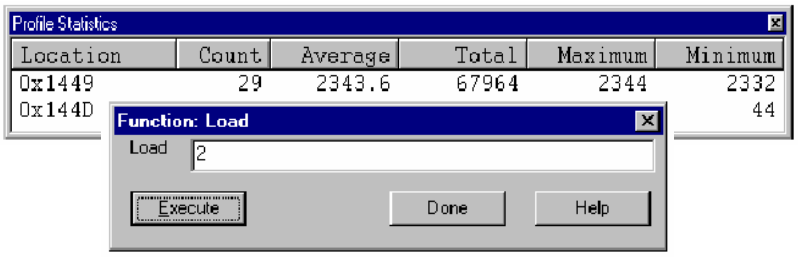
```
menuitem "Application Control"
dialog Load(loadParm "Load")
{
processingLoad = loadParm;
}
slider Gain(0, 10 ,1, 1, gainParm)
{
gain = gainParm;
}
```

Gain 函数定义的滑动指针范围：0 到 10，其步长为 1。当移动滑动指针时，变量 gain 的值将随滑动指针的改变而改变。

4.9 调节和测试 processing 函数

在第二章中曾使用测试点（profile-points）测试 puts()所需的指令周期数。现在，来使用测试点查看变量 processingLoad 改变后的结果，此结果将传递给汇编 Load 程序。 ProcessingLoad 初始化为 BASELOAD， 而 BASELOAD 在 volume.h 中定义为 1。

1. 选择 Profiler→Enable Clock，确保 Enable Clock 使能。
2. 在 Project View 中，双击 volume.c 文件。
3. 选择View→Mixed Source/ASM使能，查看C源程序及其相应汇编指令。
4. 把光标放置在 load(processingLoad) 行后的汇编指令上。
5. 点击工具栏按钮 或点击鼠标右键选择Toggle Profile Pt。
6. 把光标放置在 return(true) 行后的汇编指令上。
7. 点击工具栏按钮 (Toggle Profile-point)。
8. 选择Profiler→View Statistics。在Profile Statistics 窗口中的location栏显示了新增测试点对应的汇编指令或地址。可以通过改变Statistics area区域的大小查看更多内容；或者在Statistics area区域内点击鼠标右键，选择Allow Docking可在一个单独窗口中显示Statistics。
9. 点击工具栏按钮 或按F12。
10. 注意在第二个测试点处显示的最大周期数大约为44(真实值可能不同)，当processingLoad＝1时，它表明执行Load程序所需的指令周期数。
11. 选择GEL→Application Control→Load。



12. 在load域输入2，然后点击Execute，则对应于第二个测试点的最大周期数改变为75。每当processingLoad增加1时，指令周期数就增加31。这些指令周期数表明load函数的执行时间，load函数包含在load.asm文件中。
13. 在Profile Statistics窗口中点击鼠标右键，从弹出菜单中选择Clear

All，这将把Statistics复位为0。平均值、最大值和最小值都等于当前processingLoad的指令周期数。

14. 点击工具栏按钮　　或按Shift F5暂停程序运行。
15. 在第五章开始之前，先执行下列步骤释放在本节中使用的资源
    - 关闭Load、Gain控制窗口、sine.dat控制窗口以及Time/Frequency图形窗。
    - 选择File→File I/O并点击Remove File删除sine.dat文件。
    - 选择Profiler菜单撤消Enable Clock前的“√”。
    - 在Profile Statistics窗口中点击鼠标右键并在弹出菜单中选择Hide。
    - 选择Debug→Breakpoints，然后选择Delete All并点击OK。
    - 选择Debug→Probe Points，然后选择Delete All并点击OK。
    - 选择Profiler→profile-point，然后选择Delete All并点击OK。
    - 选择View菜单撤消MixedSource/ASM前的“√”。
    - 关闭打开的窗口和工程(Project→Close)。
    - 在project View中的volume.gel上击鼠标右键并选择Remove。
    - 在Watch Window中删除表达式并隐藏Watch Window。

4.10 进一步探索

为了进一步了解CCS，试做如下工作：

- 把processingLoad加到Watch Window，当使用Load GEL 控制时，在Watch Window中processingLoad的值将被更新。
- 在Watch Window中点击鼠标右键，在弹出菜单中选择Insert New Expression，并点击Help按钮则可得到你可以使用的数据格式。试用各种不同的格式。例如：你可键入*input，x作为Expression观察正弦输入数据的16进制格式。
- 在volume.h文件中把BUFSIZE修改为0x78(或120)并重新编译，然后重新加载程序。选择File→File I/O，将对话框中Length值修改为0x78。选择View→Graph→Time/Frequency，在弹出的图表中将Acquisition Buffer Size和Display Data Size均修改为0x78，这将使缓存器存入3个完整的正弦波而不是2.5个。点击工具栏按钮 (Animate)，注意缓冲器输入输出波形是同相的。
- 试着使用时钟数来实现测试点的统计计数。用断点代替测试点，选择Profiler→View Clock。将程序运行到第一个断点， 双击clock使统计数清零。再次运行程序，时钟则显示了程序运行到第二个断点的周期数。
- 使用探针重复4.3节到4.5节，这次只使用探针和Run命令。由于一个探测点只能对应一个操作，所以需要设置三个探测点。现在，有两个图形需要刷新，一个文件作为输入，每一个操作都有各自的探针。

  注意每一个探测点必须设置在不同的代码行上。结果，目标系统需要暂停三次，而且在目标系统运行时，操作不能在同一个探测点完成。由于上述原因，本节中将探针和断点结合比只使用探针更有效。
- 为了练习用CCS编译工程文件。将c:\ti\c5400\tutorial\volume1文件夹中的所有文件拷贝到一个新文件夹中。首先，删除volume.mak文件，然后使用CCS的Project 菜单项中的Project→New 和Project→Add File重建工程。可参见4.1节。

4.11 进一步学习

进一步学习关于Probe Points、graphs、animation和GEL文件的知识，请参见CCS中的在线帮助或CCS用户指南。

第五章 程序调试

本章介绍程序调试技术和几个 DSP/BIOS 插件模块。

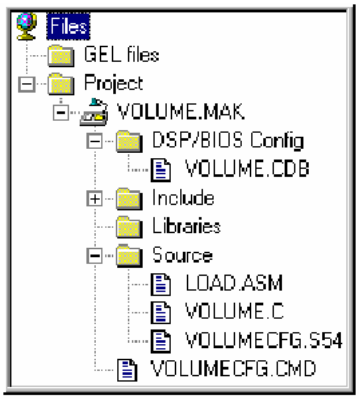
在本章中，将修改第四章的应用程序实例创建一个多线程的实例并为调试目的查看执行性能，还可以了解有关 DSP/BIOS 的更多的功能，其中包括 Execution Graph、实时分析控制板(RTA Control Panel)、Statistics View 和 CLK、SWI、STS、TRC 模块。

基本要求：目标板和 CCS 的 DSP/BIOS 组件。

5.1 打开和查看工程

首先在 CCS 中打开工程，查看工程中的源文件和库文件。

1. 如果 CCS 安装在 c:\ti 目录下，就创建 c:\ti\myprojects\volume2 目录。(如果 CCS 安装在其它位置，就在相应位置创建 volume2 目录)
2. 将目录 c:\ti\c5400\tutorial\volume2 下的所有文件拷贝到新目录下。
3. 从 WINDOWS“开始”菜单中选择“程序”→Code Composer Studio 5400 →CCStudio
4. 选择 Project→Open，在文件夹中选择 volume.mak 文件并点击 Open。点击 Project、VOLUME.MAK、 DSP/BIOS Config 和 Source 后面的+号展开 Project View。根据配置文件创建的 volumecfg.cmd 文件包含许多 DSP/BIOS 头文件（不需要检查所有的头文件）



工程中的文件包括：

- **volume.cdb** 配置文件。
- **volume.c** 包含 main()函数的 C 源程序
- **volume.h** 包含在 volume.c 中的头文件，它定义了各种常数和结构，它与前一章所用的 volume.h 文件相同。
- **load.asm** 此文件包含 Load 子程序，该子程序是一个能从 C 函数

中调用的简单汇编循环子程序。它与前一章中所用的 load.asm 相同。

- volumecfg.cmd　连接命令文件，在保存配置文件时创建
- volumecfg.s54　汇编源文件，在保存配置文件时创建
- volumecfg.h54　头文件，在保存配置文件时创建

5.2 查看源程序

本章通过由第四章的应用程序修改而得的程序实例来介绍实时操作。该实例采用片内定时器中断模拟周期性的外部中断实现数据的输入/输出，只需对其稍加修改就可真正通过外部中断实现数据的输入/输出。

1. 在 Project View 中双击 volume.c，则源程序显示在 CCS 窗口的右半部分。
2. 注意本实例中的下述几个方面：
   - 数据类型的变化。DSP/BIOS 提供的数据类型适用于其它处理器，它的绝大部分数据类型与 C 语言的数据类型相对应。
   - C 源程序中包括三个 DSP/BIOS 头文件：std.h、 log.h 和 swi.h，而且 std.h 必须放在其它 DSP/BIOS 头文件之前。
   - 配置文件中创建的对象声明为外部变量。你可在下一节查看配置文件。
   - 主函数不再调用 dataIO 和 processing 函数，而仅仅是在调用 LOG_printf 显示信息后返回,这将使应用程序进入 DSP/BIOS 空循环，而后由 DSP/BIOS 处理各线程。
   - processing 函数由 processing_SWI 软中断调用，软中断的优先级低于所有硬件中断。但也可以用硬件中断 ISR 直接完成信号处理。然而,信号处理可能需要大量机器周期而无法在下一次中断信号到达之前完成，这将妨碍中断信号的处理。
   - dataIO 函数调用 SWI_dec，SWI_dec 利用软中断作计数器减法。当计数器为 0 时，软中断就安排函数的执行并复位计数器。

     dataIO 函数仿真基于硬件的数据 I/O,一个典型的程序就是在缓存区积累数据，直到有足够的处理数据为止。本实例中，每当 processing 函数运行一次,dataIO 函数就执行 10 次,计数器的减计数由 SWI_dec 控制。

```
#include <std.h>
#include <log.h>
#include <swi.h>
#include "volume.h"
/* Global declarations */
```

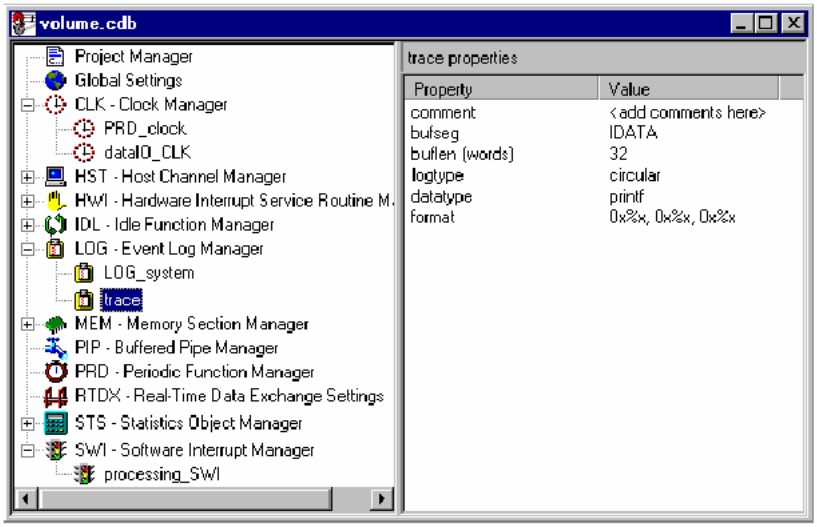
```
Int inp_buffer[BUFSIZE]; /* processing data buffers */
Int out_buffer[BUFSIZE];
Int gain = MINGAIN; /* volume control variable */
Uns processingLoad = BASELOAD; /* processing load value */
/* Objects created by the Configuration Tool */
extern LOG_Obj trace;
extern SWI_Obj processing_SWI;
/* Functions */
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;
Debugging Program Behavior /* ======== 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);
```

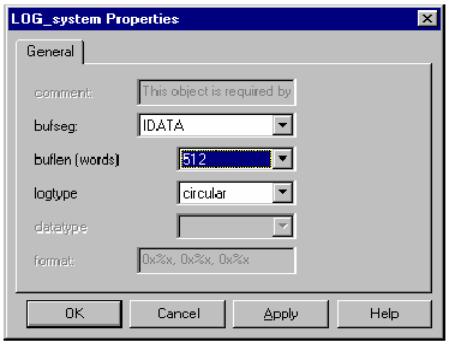
5.3 修改配置文件

对于本实例，DSP/BIOS 的配置文件已经创建。在本节中，你可以查看缺省配置中的对象。

1. 在 Project View 双击 volume.cdb 文件（在 DSP/BIOS 的 Config 文件夹中）
2. 点击 CLK、 LOG 和 SWI managers 旁边的符号+。
3. 点击 LOG 中的 trace 项，你可以从窗口的右半部分查看它的特性。这些特性与 3.1 节创建的 trace LOG 是一样的。volume.c 调用 LOG_printf 将 volume example started 写入这个 log 中。
4. 在 LOG 对象 LOG_system 上鼠标右键，从弹出菜单选择 Properties。你可以看到该对象的属性对话窗。在程序运行时，它记录各种 DSP/BIOS 模块的系统跟踪事件。



5. 将 buflen 域的值修改为 512，点击 OK。
6. 高亮度显示 CLK 目标 dataIO_CLK。注意，当 CLK 目标被激活时才调用_dataIO 函数，它是 volume.c 中的数据 I/O 函数。



注：前缀与 C 函数名称

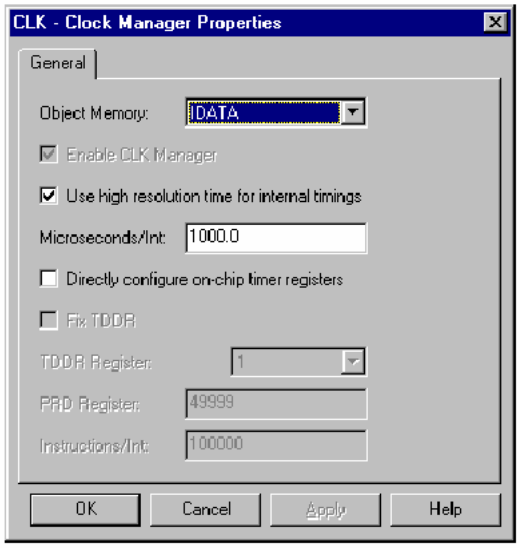
由于保存配置文件时会产生汇编语言文件，所以 C 函数名称要加一下划线作前缀。此下划线前缀是约定由汇编转而访问 C 函数的一种标记。（其细节可参见 TMS320C54x 最佳化 C 编译器用户指南有关 C 语言和汇编语言接口部分）。

此规则仅适用于用户编写的 C 函数，对于配置文件产生的对象或 DSP/BIOS API 调用则无须加下划线前缀，因为相应的两种名称会自动建立，一种会被加上前缀，一种则不会有前缀。

7. 由于 dataIO 函数不再在主函数中被调用，那由什么事件来激活该 CLK 对象呢？要想找到答案，在 CLK-Clock Manager 对象上点击鼠标右键并从弹出菜单中选择 Properties，你将看到 Clock Manager Properties 对话窗口。

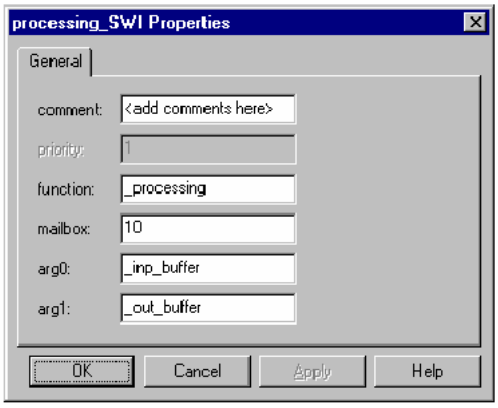
   注意，在 Enable CLK Manager 项前有可选标记，当选中时由定时器中断驱动 CLK 管理器。

8. 不作任何变动仅点击 Cancel 关闭 Clock Manager Properties 对话窗口。



9. 展开 HWI 对象查看 HWI_TINT 的属性，其中断源是 DSP 定时器，当片内定时器引起中断时，它运行 CLK_F_isr 函数。CLK 对象函数的运行是由 CLK_F_isr 硬件中断服务函数引发的，它的优先级高于软中断，一旦运行便不会被打断。（由于 CLK_F_isr 已经保护了寄存器现场，所以 CLK 函数不需要象硬件中断服务程序中正常情况下须做的那样保存和恢复现场。）

10. 在软中断对象 processing_SWI 上点击鼠标右键并从弹出菜单中选择 Properties。



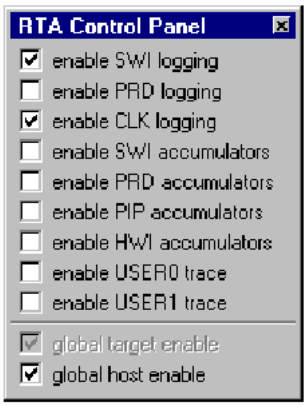
- function　软中断激活时运行 processing 函数，见 5.2 节。
- mailbox　mailbox 域的值可控制何时运行软中断。有几种 API 调用会影响 mailbox 的值，并且所产生的值将决定是否登记软中断。当软中断被登记时，具有最高优先级的软中断或硬中断服务例程将运行。
- arg0, arg1　传递给 processing 函数的 inp_buffer 和 out_buffer 的地址。

11. 不作任何改变仅点击 Cancel 关闭 Properties 对话窗口，。
12. 由于 processing 函数不再在主函数中运行，那什么事件将导致 SWI 对象运行其函数？在源程序 volume.c 中，SWI_dec 被 dataIO 函数调用，它递减 mailbox 域中的值，当 mailbox 域中的值为 0 时，则登记软中断。所以，data_CLK 对象运行 dataIO 函数 10 次，SWI 对象就运行其函数一次。
13. 选择 File→Close，将询问是否保存修改过的 volume.cdb 文件。点击 Yes 保存，保存时将产生 volumecfg.cmd、volumecfg.s54 和 volumecfg.h54。
14. 点击工具栏按钮　　或选择 Project→Build。

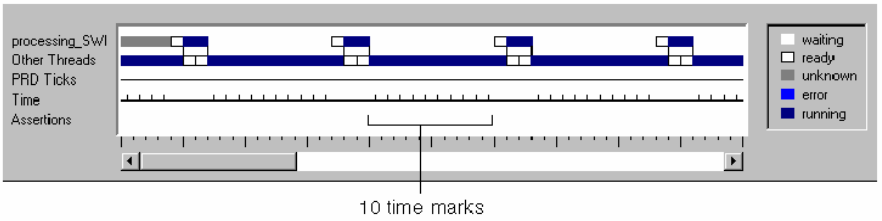
5.4 用 Execution Graph 查看任务执行情况

当在 processing 函数中设置探测点并使用图形方式观察输入输出结果时（见前一章），你已经完成了信号处理算法的测试。本阶段注意的焦点应该是明确任务可以满足实时期限的要求。

1. 选择 File→Load 并选取 volume.out，然后点击 OK。
2. 选择 Debug→Go Main，程序运行到主函数的第一条语句。
3. 选择 Tools→DSP/BIOS→RTA Control Panel，在 CCS 窗口底部将出现几个可选项。
4. 在几个可选项所在窗口区域内点击鼠标右键，取消 Allow Docking 显示方式或选择 Float in Main Windows 方式显示 RTA Control Panel。调整窗口尺寸以便看到所有选项。
5. 在选择框内放置选中标志使能 SWI 和 CLK，并使能 global host enable，如右图示。



6. 选择 Tools→DSP/BIOS→Execution Graph。Execution Graph 出现在 CCS 窗口底部，并可调整其显示方式和大小。
7. 在 RTA Control Panel 上点击鼠标右键并从弹出菜单中选择 Property Page。
8. 确认 Message Log/Execution Graph 的刷新频率为 1 秒，然后点击 OK。
9. 点击 或选择 Debug→Run，则 Execution Graph 显示如下。



10. 在 Time 行的标记给出了 Clock Manager 运行一次 CLK 函数的时间。按标记计算 processing_SWI 执行的时间间隔。在此有 10 个标记表明：dataIO_CLK 对象每执行 10 次，processing_SWI 对象就执行一次。这正如所预料的一样，因为被 dataIO 函数递减的 mailbox 域的起始值为 10。

5.5 修改和查看 load 值

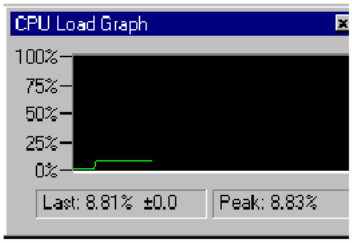
使用 Execution Graph，你可以看到该程序满足其实时期限的要求。然而，一个典型应用程序的信号 processing 函数所要完成的任务会比将数据乘以一个系数并将结果拷贝到另一缓冲区中更复杂、耗费的周期数更多。为此，可以通过增加 Load 函数占用的指令周期来模拟这样的复杂任务。

注：在本实例中 Load 值

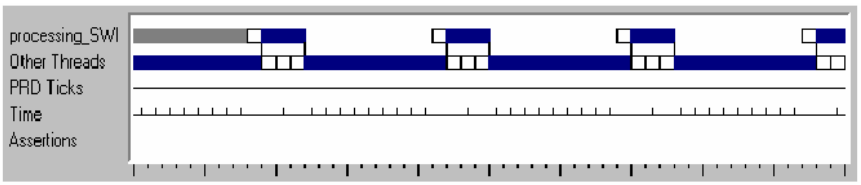
下面的 Load 值适用于以 100MIPS 运行的 C549 的。如果所采用的 C54x 是以不同速率的运行的，则需要将值乘以该 C54x 的 MIPS/100。如果不知道该 MIPS 值，可打开 volume.cdb 查看称之为 DSP MIPS 的 Global Setting 属性（CLKOUT）。

1. 选择 Tools→DSP/BIOS→CPU Load Graph，将出现一个空白的 CPU Load Graph 窗口。
2. 在 RTA Control Panel 上点击鼠标右键并从弹出菜单选择 Property Page。
3. 将 Statistics View/CPU Load Graph 中的 Refresh Rate 修改为 0.5 秒并点击 OK。注意当前 CPU 的 load 值非常低。

   由于Statistics View和CPU Load仅将少量数据从目标板传送到主机，因此你可以频繁地刷新这些窗口数据而不会对运行程序造成大的影响。Message Log和Execution Graph传送的数据是有关配置文件中定义的LOG对象的缓存长度属性的，其数据量大，因此不能频繁地刷新这两个窗口。

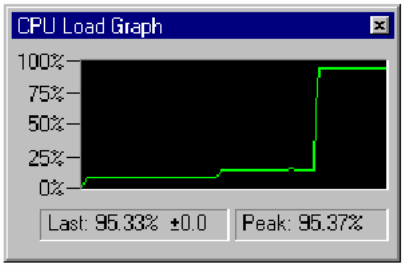


4. 选择 File→Load GEL 并选择 volume.gel，然后点击 Open。
5. 选择 GEL→Application Control→Load。
6. 键入 3000 作为新的 load 值，然后点击 Execute，CPU 负荷增加到 7%左右。
7. 在 Execution Graph 项点右键，从弹出菜单选择 Clear，注意此时程序仍满足其实时期限的要求。在 processing_SWI 函数各次执行之间存在 10 个时间标记。
8. 使用 GEL 控制修改 load 值为 5000，然后点击 Execute。
9. 在 Execution Graph 区域内点击鼠标右键并从弹出菜单中选择 Clear。

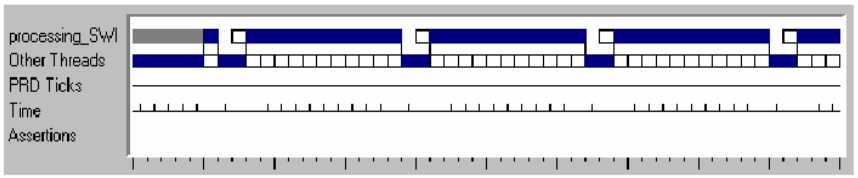


在 processing_SWI 函数执行期间出现了一个时间标记，这意味着程序不满足实时期限的要求吗？不，它只表明运行程序的功能正确。能够引起 CLK 对象的服务例程运行的硬中断能够中断软中断服务例程的执行，而在软中断服务例程再次运行之前，它能够完成自己的任务。

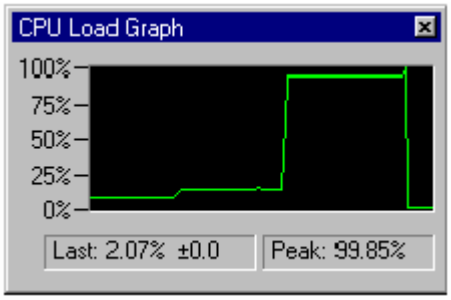
10. 使用 GEL 控制修改 load 值为 3000，然后点击 Execute。CPU 的负荷增加到 95%左右。



11. 在 Execution Graph 区域内点击鼠标右键并从弹出菜单中选择 Clear。因为 processing_SWI 在 10 个时间标记发生之前完成，所以程序仍满足实时期限的要求。



12. 使用 GEL 控制修改 load 值为 35000，然后点击 Execute。由于刷新过程在空闲任务中实现，而空闲任务在该程序中具有最低优先级，所以 CPU Load Graph 窗口 和 Execution Graph 窗口将停止频繁刷新甚至于可以停止刷新。又由于其它高优先级线程占用了 CPU 的全部处理时间，因此无足够的时间用于主机控制完成更新。此时，程序不满足其实时期限的要求。
13. 选择 Debug→Halt，这将暂停程序的运行并刷新 Execution Graph 窗口。当应用程序不满足实时期限的要求时，窗口中的 Assertions 行将显示出错信息。
14. 使用 GEL 控制修改 load 值为 10，然后点击 Execute，则 CPU Load Graph 和 Execution Graph 窗口将再次刷新。



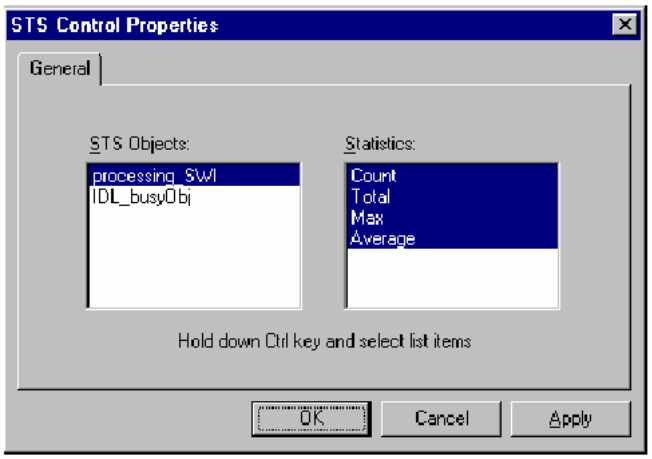
注：采用 RTDX 修改 Load

采用 Load GEL 控制可暂停目标系统。如果正在分析一个实时系统，又不想影响系统性能，则可采用 RTDX 修改 Load。下一章将说明如何用 RTDX 修改实时 Load。

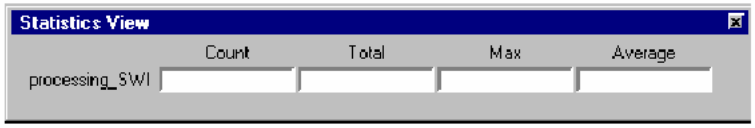
5.6 分析任务的统计数据

可使用 DSP/BIOS 的其它控制功能查看 DSP 的 Load 值和 processing_SWI 对象的处理过程的统计数据。

1. 选择 Tools→DSP/BIOS→Statistics View 将出现一个 Statistics View 区域，它显示 Load DSP/BIOS 程序和/或设置使用控制功能的属性。
2. 在 Statistics View 区域中点击鼠标右键并从弹出菜单中选择 Property Page，按下图加亮相应项并点击 OK。



3. 你将看到 processing_SWI 对象的统计域。(在该统计域上点击鼠标右键并从弹出菜单中撤消 Allow Docking) 可使其成为一个单独的窗口，也可调整窗口大小使得全部四个域均可见。



4. 在 RTA Control Panel 中，在 SWI 累积器前放置选中标记“√”。
5. 若已暂停程序，点击工具栏按钮 （Run）。
6. 注意 Statistics View 中的 Max 值，SWI 的统计计数单位为指令周期。
7. 使用 GEL 控制增加 load 值，然后点击 Execute。注意 Max 值的改变，这是为从 processing_SWI 运行开始到结束所完成的指令数增加了。
8. 使用不同的 load 值试验。减小 load 值，在 Statistics View 区域内点击鼠标右键并从弹出菜单中选择 Clear，这将把所有统计域复位到它们各自的最小可能值，这样你可以从 Max 域中观察到当前的指令周期

数。

9. 点击工具栏按钮 (Halt)，关闭所有已经打开的 DSP/BIOS 和 GEL 控制项。

5.7 增加 STS 显式测试

在前面的章节中，曾用 Statistics View 观察了软件中断服务例程执行期间完成的指令周期数。如果使用配置文件，DSP/BIOS 自动支持统计功能，这称为隐式测试。也可使用 API 调用收集其他统计数据，这称为显式测试。

1. 在 Project View 中，双击 volume.cdb 文件（该文件在 DSP/BIOSConfig 文件夹中）以便打开它。
2. 在 STS manager 上点击鼠标右键并从弹出菜单中选择 Insert STS。
3. 将新增加的对象 STS0 更名为 processingLoad_STS。该对象的缺省属性完全是正确的。
4. 选择 File→Close，将询问你是否保存修改过的 volume.cdb，点击 YES。
5. 在 Project View 中，双击 volume.c 打开它进行编辑。对程序作如下修改：
   - 在包含 swi.h 的文件的那一行下中增加下列内容：

```
#include <clk.h>
#include <sts.h>
#include <trc.h>
```

   - 在标有注释 “Objects created by the Configuration Tool”的程序段中添加下述声明语句：

```
extern STS_Obj processingLoad_STS;
```

   - 在 load 函数调用前的 processing 函数中添加以下内容：

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

   - 在 load 函数调用后的 processing 函数中的添加如下语句：

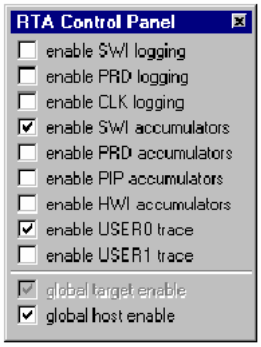
```
if (TRC_query(TRC_USER0) == 0) {
STS_delta(&processingLoad_STS, CLK_gethtime());
}
```

6. 选择 File→Save 保存修改过的 volume.c 文件。
7. 点击工具栏按钮 或选择 Project→Build。

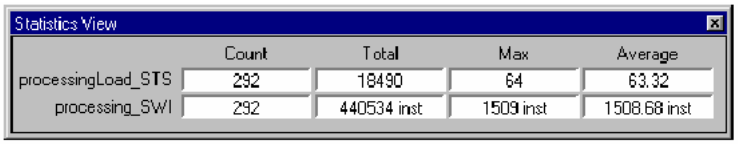
5.8 观察显式测试统计数据

要观察由新增 STS 显式测试所提供的信息，可使用 Statistics View 和 RTA Control Panel。

1. 选择 File→Load Program，选择你刚刚建立的程序 volume.out，然后点击 Open。
2. 选择 Tools→DSP/BIOS→RTA Control Panel。
3. 在 RTA Control Panel 区域点鼠标击右键，然后取消 Allow Docking 选项使 RTA Control Panel 显示为一个独立的窗口。调整窗口尺寸到能够看到所有选项。



4. 将选中标记“√”放置在 enable SWI accumulators、enable USER0 trace 和 global host enable 前面的选择框中 。
   使能 USER0 跟踪，使得 TRC_query(TRC_USER0)调用的返回值为 0。
5. 选择 Tools→DSP/BIOS→Statistics View。
6. 在 Statistics View 区域点击鼠标右键并从弹出菜单中选择 Property Page，然后加亮对象 processing_SWI 和 processingLoad_STS，并加亮所有的四个统计选项。
7. 点击 OK，可看到两个对象的统计域。在该域上点击鼠标右键并从弹出菜单中撤消 Allow Docking，可使该区域成为一个单独的窗口，亦可调



   整窗口大小使全部域可见。
8. 点击工具栏按钮 或选择 Debug→Run。
9. 用 processingLoad_STS 的 Max 值减去 processing_SWI 的 Max 值，其结果约为 1445 条指令（实际显示的值可能有变化）。SWI 的统计数据是用指令周期来度量的。因为曾用 CLK_gethtime 函数作为处理 load 值的基准，所以 processingLoad_STS 由片内定时计数器计数，它等同于指令周期数。这些指令是在 processing 函数中执行的，而不是在 STS_set 和 STS_delta 调用之间执行的，如下所示。

processing_SWI

processingLoad_STS

```
/*  ======== processing ======== */
Int processing(Int *input, Int *output)
{
    Int size = BUFSIZE;
    while(size--){
        *output++ = *input++ * gain;
    }
    /* enable instrumentation 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_gethtime());
    }
    return(TRUE);
}
```

例如，当 load 值为 10 时，processingLoad_STS 的 Max 值约为 203，而 processing_SWI 的 Max 值约为 1648。在 STS_set 函数和 STS_delta 函数之外的 processing 函数中所执行的指令周期数的计算公式为：1648 - 203 = 1445

10. 选择 GEL→Application Control→Load（如果已关闭 CCS 而后又重新启动它，则必须重装 GEL 文件）
11. 修改 load 值，然后点击 Execute。
12. 注意当两个 Max 都增加时，而它们的差值却保持不变。
13. 从 RTA Control Panel 中取消 enable USER0 trace 前的选中标记“√”。
14. 在 Statistics View 区域内点击鼠标右键并从弹出菜单中选择 Clear。
15. 注意，processingLoad_STS 的统计值没有改变，这是因为没有选中 USER0 跟踪选项，使下面这条语句的条件不能满足：

    if (TRC_query(TRC_USER0) = = 0)

    因此，不执行对 STS_set 和 STS_delta 的调用。
16. 完成下面的步骤，以便为下一节做准备。
    - 点击工具栏按钮 或按 Shift F5 暂停程序的运行。
    - 关闭所有 GEL 对话窗口、DSP/BIOS 插件和源程序窗口。

5.9 进一步探索

为了进一步探索 DSP/BIOS，试做下述工作：

- 在配置文件中，将 SWI Manager 的 Statistics Units 的属性修改为毫秒或微秒，重新编译和加载应用程序并请注意观察 Statistics View 中统计值的变化情况。
- 修改 volume.c 源文件，用 CLK_getltime 函数代替 CLK_gethtime 函数。重新编译和加载应用程序并观察 Statistics View 中统计值的变化情况。函数 CLK_getltime 使 Execution Graph 窗口中时间标记分辨率变低。当使用 CLK_getltime 时，必须明显地增加 load 值以改变 Statistics View 中的数值。

5.10 进一步学习

欲了解有关 CLK、SWI、STS 和 TRC 模块的知识，请参见 CCS 中的在线帮助或 TMS320C54x　DSP/BIOS 用户指南。

第六章 实时分析

本章介绍程序的实时分析技术并纠正程序中存在的错误。

本章中，将用第五章的例子来作实时分析并纠正与实时性有关的问题。将采用 RTDX 对目标系统作实时改动，将使用 DSP/BIOS 周期函数并设置软中断优先级等。

基本要求：CCS 的 DSP/BIOS 和 RTDX 组件、目标板

6.1 打开和查看工程

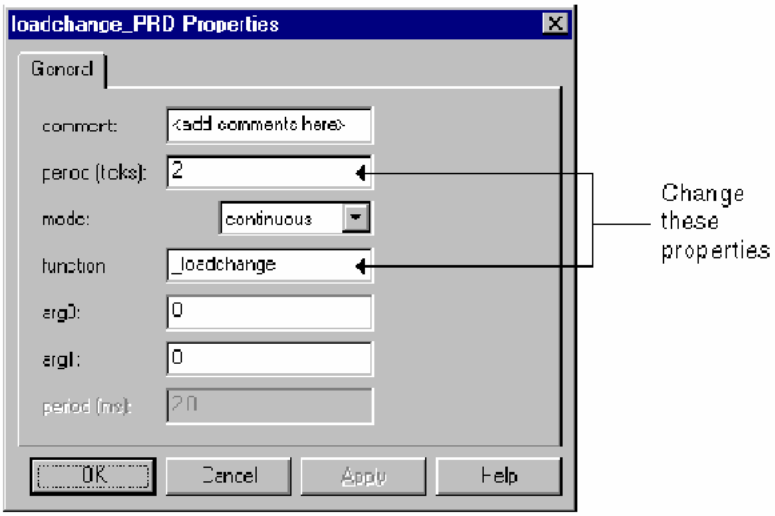
本章的工作是在前一章工作基础上进行的，如果没有做前一章的工作，可以把目录 c:\ti\c5400\tutrial\volume3 中的文件拷贝到你的工作文件夹下。

1. 从目录 c:\ti\c5400\tutrial\volume4 下拷贝下述文件到你的工作文件夹中(注：不要拷贝所有文件，特别不要拷贝 volume.cdb)
   - volume.c　采用 RTDX 的源程序，它无需中止运行程序即可修改 load 值。
   - loadctrl.exe　用 VB5.0 编写的 Windows 应用程序，它采用 RTDX 实时发送 load 值到目标板中。
   - loadctrl.frm，loadctrl.frx，loadctrl.vbp　如果你拥有 VB5.0 开发环境，你可以用它来查看 loadctrl.exe 应用程序的源文件。
2. 从 Windows 开始菜单中，选择 Programs→Code Composer Studio ’C5400 →CCStudio。
3. 选择 Project→Open 并从你的工作文件夹中打开 volume.mak。

6.2 修改配置文件

在本例中，你需要增加一个新对象到配置文件 volume.cdb 中（该文件已经存在于 c:\ti\c5400\tutorial\volume4　目录中）

1. 在 Project View 的 volume.cdb 处双击鼠标左键。
2. 选择 LOG_system，将属性值 buflen 修改为 512 并点击 OK。
3. 在 PRD　magager 处点击鼠标右键，并从弹出菜单中选择 Insert PRD。
4. 将 PRD0 改名为 loadchange_PRD。
5. 在 loadchange_PRD 处，点击鼠标右键，并从弹出菜单中选择 Properties。
6. 将 loadchange_PRD 属性设置为下述值并点击 OK。
   - 周期值设为 2。缺省状态下，PRD　manager 使用 CLK　maganer 驱动 PRD。CLK 类触发 PRD 的缺省值为 1ms，因此 PRD 对象每 2ms 执行一次其服务例程。
   - 函数名修改为 _loadchange。PRD 对象以所选定的周期执行 loadchange 函数。在下一节中，可查看该函数。



7. 点击 SWI　maganer 旁的‘＋’号，可以发现已经自动增加了一个名为 PRD_swi 的 SWI 对象。在运行时，软中断执行周期函数，因此，所有 PRD 函数均在软中断服务例程中被调用，且调用时须保护现场。相应地，CLK 函数被硬中断服务例程调用。
8. 点击 CLK　manager 旁的‘＋’号，可以发现名为 PRD_clock 的 CLK 对

象调用 PRD_F_tick 函数，该函数促使 DSP/BIOS 系统时钟计时（通过调用 PRD_tick API 函数），如果有 PRD 函数需要运行将产生 PRD_swi 软中断。

9. 在 PRD manager 处，点击鼠标右键，并从弹出菜单中选择 Properties。选择 Use CLK Manager to drive PRD。若在你的工程文件中没有选择该属性，则 PRD_clock 对象将自动删除，你的程序可以从其他事件（如硬中断）中调用 PRD_tick 来驱动周期函数。
10. 回忆一下 processing_SWI 对象，它的 mailbox 值为 10，而且 mailbox 值在 dataIO_CLK 对象中每经 1ms 就减 1，因此 processing_SWI 每 10ms 运行 1 次，而 loadchange_PRD 每 2ms 运行 1 次。
11. 选择 File→Close，根据提示选择 Yes 保存 volume.cdb，这将自动产生 volumecfg.cmd，volumecfg.s54，volumecfg.h54 文件。
12. 选择 Project→Rebuild All 或者点击工具栏按钮 (Rebuild All)。

6.3 查看源程序

在 Project View 的 volume.c 处双击鼠标左键，源文件将显示在 CCS 窗口的右半部分。由于文件 volume.c 是从 c:\ti\c5400\tutorial\volume4 目录中拷贝到你的工作文件夹中的，因此它与前一章的源程序存在下述差别：

- 增加了一个头文件：

#Include <rtdx.h>

- 增加了两条声明语句：

RTDX_CreateInputChannel(contro_channel);

Void loadchange(Void);

- 在主函数中增加了下列函数调用：

RTDX_enableInput(&control_channel)

- 下述函数为 PRD 对象调用的函数

```
/* ======== loadchange ========
* FUNCTION: Called from loadchange_PRD to
* periodically update load value.
* PARAMETERS: none.
* RETURN VALUE: none.
sizeof(control));
if ((control < MINCONTROL) || (control > MAXCONTROL)) {
*/
Void loadchange()
{
static Int control = MINCONTROL;
/* Read new load control when host sends it */
if (!RTDX_channelBusy(&control_channel)) {
RTDX_readNB(&control_channel, &control,
LOG_printf(&trace,"Control value out of range");
}
else {
processingLoad = BASELOAD << control;
```

```
LOG_printf(&trace,"Load value = %u",processingLoad);
}
}
```

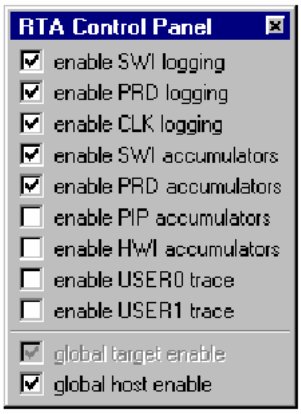
该函数使用 RTDX　API 函数改变实时处理信号的 load 值，其改变表现在下述方面：

- 调用 RTDX_enableInput，名为 control_channel 的输入通道变为使能，这样数据流可以从主机传送到目标板中。运行时，VB 程序将 load 值写入该通道并将它发送给目标板。
- 调用 RTDX_readNB 函数，请求主机方通过 control_channel 发送 load 值并将该值存入名为 control 的变量中。该函数调用后立即返回，无需等待主机发送数据。从 RTDX_readNB 函数调用到数据被写入变量 control，control_channel 一直处于忙状态，不再响应其他请求，此期间会调用 RTDX_channelBusy 函数并返回 TRUE。
- 语句“processingLoad = BASELOAD << control;”使 BASELOAD（1）中的比特向左移位，移动的位数由变量 control 指定。由于 MAXCNTROL 设置为 31，因此 load 最大值为 32 位变量能存储的最大值。

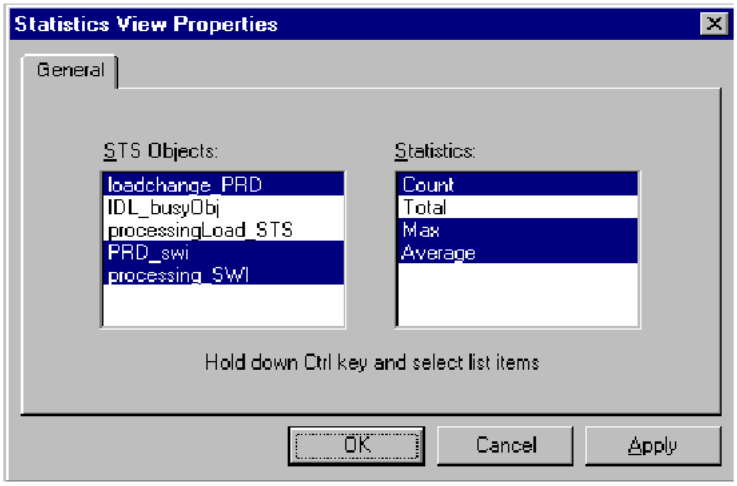
6.4 使用 RTDX 控制修改运行时的 load 值

当你在 processing 函数中放置测试点测试程序、观察输入输出数据的图形时（见 4.3 节），你已经测试了数字处理算法。在开发的现阶段，注意的焦点应是搞清楚新增加的任务仍旧能够满足其实时期限的要求。同样，测试点会中止目标程序并对实时测试带来影响。

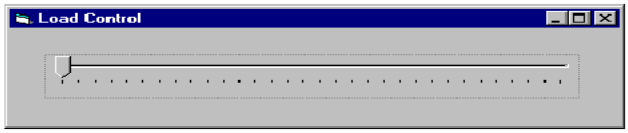
1. 选择 File→Load Program，打开刚编译过的 volume.out
2. 选择 Tools→DSP/BIOS→RTA Control Panel。
3. 在 RTA Control Panel 处点击鼠标右键，取消 Allow Docking 选定以便在将一个单独的窗口中显示 RTA Control Panel。调整窗口大小以便能看到全部选项。
4. 放置“√”，使 SWI、PRD 和 CLK logging 处于使能状态，使 SWI 和 PRD accumulators 处于使能状态，使 global tracing 处于使能状态。



5. 选择 Tools→DSP/BIOS→Execution Graph。Execution Graph 区域显示在 CCS 窗口的底部，可调整其大小或将它显示为一个单独的窗口。
6. 选择 Tools→DSP/BIOS→Statistics View。
7. 在 Statistics View 区域点击鼠标右键并从弹出菜单中选择 Property Page，加亮图示选项。



8. 点击 OK。
9. 调整 Statistics 区域的大小以观察选择的统计数据域。
10. 在 RTA　Control Panel 处点击鼠标右键并从弹出菜单中选择 Property Page。
11. 设置 Message Log/Execution Graph 的刷新频率为 1s，设置 Statistics View/CPU Load Graph 的刷新频率为 0.5s，然后点击 OK。
12. 选择 Tools→RTDX。
13. 注意 RTDX 已经处于使能状态，这是在步骤 2 中当打开 DSP/BIOS 控制时在后台完成的。通过配置 DSP/BIOS 控制项使 RTDX 工作于连续模式。在连续模式下，RTDX 不在日志文件中记录从目标板接收的数据。这允许数据流连续传送。（如果你的程序没有使用 DSP/BIOS，你可以直接使用 RTDX 区域配置和使能 RTDX）
14. 使用 Windows 浏览器运行你的工作文件夹下的 loadctrl.exe 程序，就会出现一个 Load Control 窗口。该应用程序使用 VB 和 RTDX 编制而成，如果你拥有 VB 开发环境，你可以查看 loadctrl.exe 的 VB 源程序，它位于 c:\ti\c5400\tutorial\volume4 目录下。

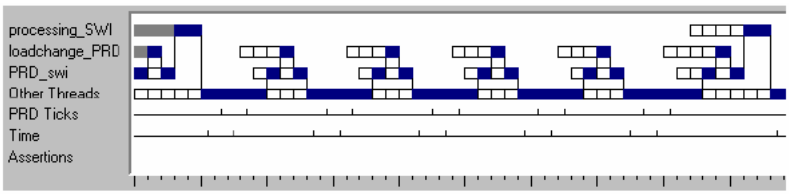


该应用程序使用了下述 RTDX 函数：

- rtdx.Open(“control_channel”,”W”) 当打开应用程序时，打开向目标板写入信息的控制通道。
- rtdx.Close() 当关闭应用程序时，关闭控制通道。
- Rtdx.Writel2(datal2,bufstate) 把滑动控制条的当前值写入控制通道，目标程序可以读取该值并使用它刷新 load 值。

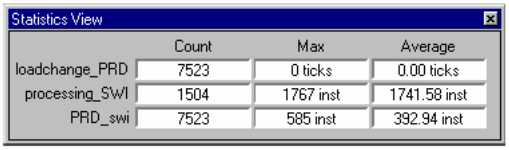
15. 选择 Debug→Run 或者点击工具栏按钮 (Run)。

注意:Processing_SWI 每 10 个计数单位(按 PRD ticks 计)出现一次，PRD_SWI 每 2 个计数单位运行一次，正如所预料的一样。loadchange_PRD 是被 PRD_swi 调用的。

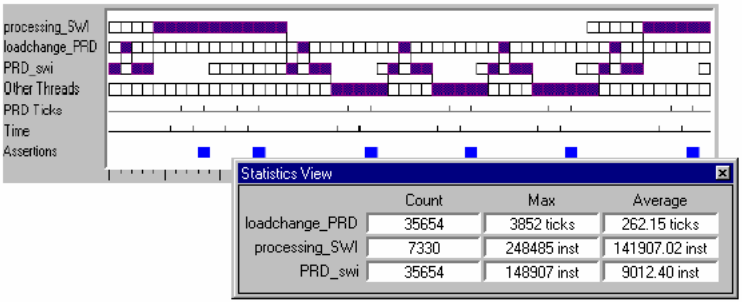


PRD 统计是用 PRD ticks 来计量的，SWI 统计用指令周期来计量的，

loadchange_PRD 的 Max 和 Average 域显示表明，在该函数开始运行到运行结束所需的时间之间不足一个完整的 PRD 周期。



16. 使用 Load Control 窗口逐渐增加处理 load 值。(如果在 DSP 应用程序停止工作的情况下滑动 Load Control 窗口的控制条，则 RTDX 将新的 load 控制值缓存在主机上。这些值直到 DSP 应用程序重新运行、并调用 RTDX_readNB 以请求从主机刷新 load 值时才会有影响)。
17. 重复步骤 16，直到 loadchange_PRD 中的 Max 和 Average 值增加并在 Execution Graph 的 Assertion 行出现蓝色方块，Assertion 表明一个线程不满足实时期限的要求。



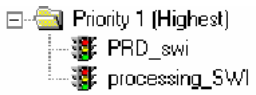
为什么？当 load 值超过某个值时，loadchange_PRD 的 Max 值开始增加，随着 load 值的增加，processing_SWI 需要占用的运行时间长得致使 loadchange_PRD 在超过实时期限很长时间才能开始运行。

当 load 值增加到一定程度时，低优先级的 idle 循环就不再执行，主机停止接收实时分析数据，DSP/BIOS 插件停止刷新，暂停目标程序用排队数据刷新插件。

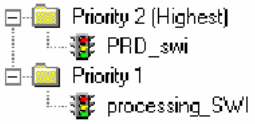
6.5 修改软中断优先级

为了便于理解程序为什么不满足实时期限的要求，你需要检查软中断任务的优先级。

1. 选择 Debug→Halt 中止目标程序。
2. 在 Project View 中双击文件 volume.cdb。
3. 加亮 SWI manager，注意 SWI 对象的优先级显示在窗口的右半部分。



4. 由于 PRD_swi 和 processing_SWI 具有相同的优先级，PRD_swi 不能先处理 processing_SWI。processing_SWI 每 10ms 运行一次而 PRD_swi 每 2ms 运行一次。当 load 值较高时，processing_SWI 运行时间超过 2ms，它使得 PRD_swi 不满足实时期限的要求。
5. 为了解决上述问题，需要把 PRD_swi 设置成最高优先级。降低 processing_SWI 的优先级。这增加了一个第二级的优先级别，现在 PRD_swi 具有最高优先级。



6. 选择 File→Save 保存你所作的修改。
7. 选择 File→Close 关闭 volume.cdb。
8. 选择 Project→Build 或者点击工具栏按钮 (Incremental Build)
9. 选择 File→Reload Program。
10. 选择 Debug→Run 重新运行目标程序。在程序运行时使用 RTDX 使能的 Windows 应用程序 loadctrl.exe 应用程序窗口改变 load 值。
11. 注意:现在可增加 load 值而不会使 PRD_swi 不满足实时期限的要求。
12. 在进行下一章之前（完成 6.6 节之后），需要完成下述操作：
    - 点击工具栏按钮 或者按 Shift+F5 中止程序运行。
    - 关闭 GEL 对话框、DSP/BIOS 插件和源程序窗口。

6.6 进一步探索

为了进一步研究 DSP/BIOS，试做下述工作：

- 当增加 load 值时，Execution Graph 表明 processing_SWI 需要的运行时间超过 1 个 PRD 计数周期。这是否意味着 processing_SWI 不满足实时期限的要求？请回忆一下，PRD ticks 以每毫秒为时间间隔出现的同时，processing_SWI 必须每 10ms 运行一次。
- 如果直接从硬中断服务例程中调用 processing 函数而不是从软中断中调用，那将发生什么呢？由于硬中断优先级低于软中断优先级，那将使得程序不满足实时期限的要求。请回忆一下，当 Load 值很高时，PRD_swi 需要先于 processing_SWI 执行。如果 processing_SWI 是一个硬中断，PRD_swi 则不能先于它执行。
- 观察 CPU Load Graph。使用 RTA Control Panel 打开和关闭统计累积器，注意这对 CPU Load Graph 无影响，这表明统计累器在处理器上放置了一个很小的 Load 值。

统计累积器对 processing_SWI 的统计数据有多大影响呢?可一边打开和关闭统计累积器，一边观察统计数据，这之间的差别是各累计器所需的指令数的精确计量。为了观察效果，记住对统计数据窗口点击鼠标右键并清除统计计数。

- 就象在 5.7 节中所做的那样，在 loadchange 函数中增加函数 STS_set 和 STS_delta 调用。这样的修改对 CPU load 影响如何?在 dataIO 函数中增加 STS_set 和 STS_delta 的调用,这又将对 CPU Load 有何影响？为什么?试考虑各函数执行时的频率。对于经常执行的函数而言，它需要的处理时间的细微的增加，都可能对 CPU Load 造成很大的影响。

6.7 进一步学习

想要更多地了解软中断属性、RTDX 和 PRD 模块，请参见 CCS 和 RTDX 的在线帮助或者参考 TMS320C54x DSP/BIOS User’s Guide(用户指南)。

第七章　I/O

本章介绍用 DSP/BIOS 和 RTDX 技术实现 I/O。

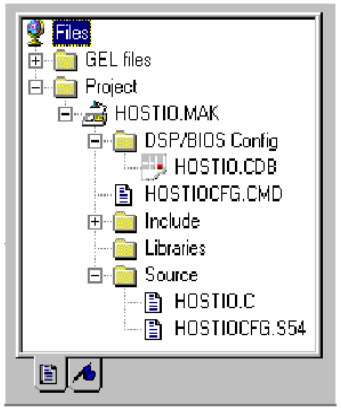
在本章中，将采用 RTDX 和 DSP/BIOS 使应用程序与 I/O 设备相连接。也可以采用 DSP/BIOS API 的 HST、PIP 和 SWI 模块。

基本要求：DSP/BIOS 和 RTDX 组件，目标板

7.1 打开和查看工程

在 CCS 中打开工程文件并查看它包含的源程序及库文件。

1. 如果你把 CCS 安装在 c:\ti 目录，可以创建一个新目录 c:\ti\myproject；如果 CCS 安装在其他目录，则可在相应的位置建立 myproject 目录。
2. 将 c:\ti\c5400\tutorial\hostio1 下的所有文件拷贝到新目录下。
3. 从 Windows 开始菜单中，选择 Program→Code Composer Studio ’C5400→CCStudio。
4. 选择 Project→Open 并打开 hostio.mak。
5. 依次点击 Project、HOSTIO.MAK、Source 旁的’+’，可以看到工程文件中包含的各种文件。hostiocfg.cmd 及 include 中的头文件是在保存配置文件时创建的。本例程需要的文件有：



   - hostio.c 主程序
   - signalprog.exe　VB 应用程序，它产生正弦波并显示输入/输出信号
   - slider.exe　　VB 应用程序，它控制输出信号幅度
   - hostiocfg.cmd　连接命令文件，它仅增加了名为 trace 的 LOG 对象
   - hostiocfg.s54　汇编源文件
   - hostiocfg.h54　头文件

7.2 查看源程序

本章中的例子模拟一个能数字化音频信号、调整音量、产生幅度可调的模拟输出的 DSP 应用程序。

为简单起见，该例没有使用收、发模拟信号的设备，而是使用主机产生的数字信号测试算法。数据的输入/输出及音量控制信号是采用 RTDX 在主机和目标板之间传送的。

在主机上运行的 VB 应用程序使用 RTDX 产生输入信号并显示输入/输出信号，该程序允许开发者不中止程序运行即进行算法测试。

1. 双击 Project View 中的 hostio.c 源程序。
2. 注意源程序的下述方面：
   - 三个 RTDX 通道声明为全局的。第一个输入通道控制音量，第二个输入通道接收主机发送来的信号，输出通道用于从目标板向主机发送的输出信号。（是站在目标板应用程序的角度来称输入和输出信道的，即：输入信道从主机接收数据，输出信道向主机发送数据。）
   - 当通道当前不是处于等待输入状态时，调用 RTDX_channelBusy 函数将返回 FALSE，它表明数据已到达可供读取。如第六章所述，调用 RTDX_readNB 无需等待接收数据就可返回 DSP 应用程序。主机将数据异步写入控制通道。
   - RTDX_Poll 用于 RTDX 下层应用之间的数据读/写的。
   - 如果输入通道是使能的，RTDX_read 将等待数据的到来。
   - 如果输出通道是使能的，RTDX_write 将缓冲区的数据写入到输出的 RTDX 通道中。
   - 当目标板通过调用 RTDX_enableInput 使控制通道 control_channel 处于使能状态时，则该例程中的其他 RTDX 通道将处于非使能状态。而下一节描述的主机程序将使能这些通道。使用 control_channel 的滑动控制被视为应用程序的有机组成部分，在目标程序中使能该通道会使人们在应用程序运行时亦清楚该通道是处于使能状态的；而 A2D 通道和 D2A 通道是用于算法测试的。所以这些通道是通过主机应用程序设定为使能状态或非使能状态的。

```
#include <std.h>
#include <log.h>
```

```
#include <rtdx.h>
#include "target.h"
#define BUFSIZE 64
#define MINVOLUME 1
typedef Int sample; /* representation of a data sample from A2D */
/* Global declarations */
sample inp_buffer[ BUFSIZE];
sample out_buffer[ BUFSIZE];
Int volume = MINVOLUME; /* the scaling factor for volume control */
/* RTDX channels */
RTDX_CreateInputChannel(control_channel);
RTDX_CreateInputChannel(A2D_channel);
RTDX_CreateOutputChannel(D2A_channel);
/* Objects created by the Configuration Tool */
extern LOG_Obj trace;
/*
* ======== main ========
*/
Void main()
{
sample *input = inp_buffer;
sample *output = out_buffer;
Uns size = BUFSIZE;
TARGET_INITIALIZE(); /* Enable RTDX interrupt */
LOG_printf(&trace,"hostio example started");
/* enable volume control input channel */
     RTDX_enableInput(&control_channel);
Connecting to I/O Devices while (TRUE) {
/* Read a new volume when the hosts send it */
if (!RTDX_channelBusy(&control_channel)){
RTDX_readNB(&control_channel, &volume, sizeof(volume));
}
while (!RTDX_isInputEnabled(&A2D_channel)){
RTDX_Poll(); /* poll comm channel for input */
```

```
}
/*
* A2D: get digitized input (get signal from the host through
* RTDX). If A2D_channel is enabled, read data from the host.
*/
RTDX_read(&A2D_channel, input, size*sizeof(sample));
/*
* Vector Scale: Scale the input signal by the volume factor to
* produce the output signal.
*/
while(size--){
*output++ = *input++ * volume;
}
size = BUFSIZE;
input = inp_buffer;
output = out_buffer;
/*
* D2A: produce analog output (send signal to the host through
* RTDX). If D2A_channel is enabled, write data to the host.
*/
RTDX_write(&D2A_channel, output, size*sizeof(sample));
while(RTDX_writing){
RTDX_Poll(); /* poll comm channel for output */
}
}
}
```

7.3 Signalprog 应用程序

VB 编制的应用程序 signalprog.exe 的源程序可用于文件 signalfrm.frm。有关该应用程序的详细说明可见文件 signalprog.pdf。可查看几个对该例而言很重要的例程和函数：

- Test_ON。点击 Test_ON 按钮时运行该例程。首先，它为输入通道和输出通道创建 RTDX 接口实例；接着，它打开输入/输出通道并使它们处于使能状态。该例程同时清除图形并启动 Transmit_Signal 和 Receive_Signal 的定时器。VB 源程序中的全局变量声明把 VB 应用程序中的通道和 hostio.c 应用程序中相应通道联系到一起，如下：

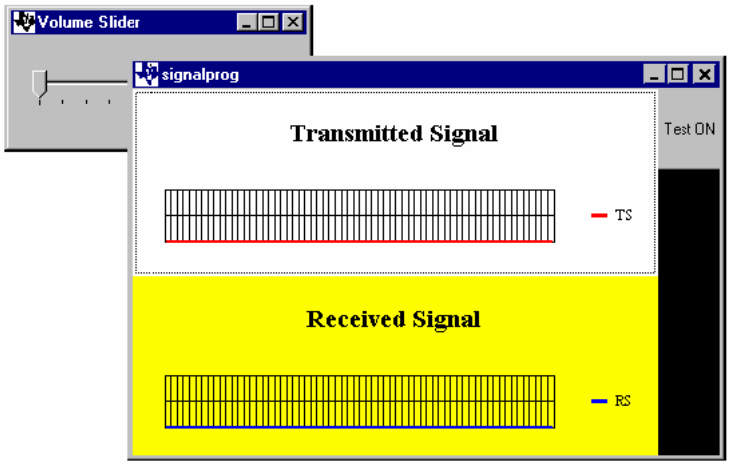
```
' Channel name constants
Const READ_CHANNEL = "D2A_channel"
Const WRITE_CHANNEL = "A2D_channel"
```

- Test_OFF　该例程废止、关闭和释放 Test_ON 例程创建的 RTDX 对象，并使定时器处于非使能状态。
- Transmit_Signal 首先，该函数产生正弦波信号并把正弦信号显示在 Transmitted Signal 图中；然后，它试图使用‘写’方式将数据传送到目标板。
- Receive_Signal　该函数使用 ReadSAI2 方式从目标板读取数据，并将信号显示在 Received Signal 图中。
- tmr_MethodDispatch_Timer　该例程调用 Transmit_Signal 和 Receive_Signal 函数，该例程在定时器被 Test_On 例程使能后每隔 1ms 被调用一次。

7.4 运行应用程序

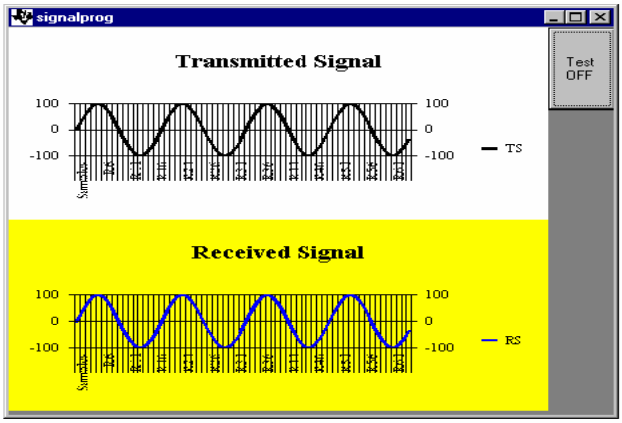
1. 点击工具栏按 钮或选择 Project→Build。
2. 选择 File→Load Program 并双击 hostio.out。
3. 选择 Tools→RTDX。
4. 选择窗口中 RTDX 区域的 Configure，在 RTDX 属性对话框的 General Settings 中选择 Continuous RTDX 模式并点击 OK。
5. 将 RTDX 域的 RTDX Disable 改为 RTDX Enable，此时按钮 Configure 变为 Diagnostics。
6. 选择 Tools→DSP/BIOS→Message Log。在 Message Log 区域点击鼠标右键并从弹出菜单中选择 Property Page，选择名为 trace 的 Log 对象并点击 OK。
7. 点击工具栏按钮 或选择 Debug→Run。
8. 使用 Windows 浏览器，运行 signalprog.exe 和 slider.exe，你可以看到两个 VB 应用程序。

   由于 slider.exe 应用程序将创建和打开 RTDX 控制通道，因此它必须在 RTDX 使能以后运行，否则它不能打开控制通道。Signalprog 应用程序只有在点击 Test On 按钮后才使用到 RTDX，因此它可以在任何时候运行。



9. 调整 signalprog 窗口高度。

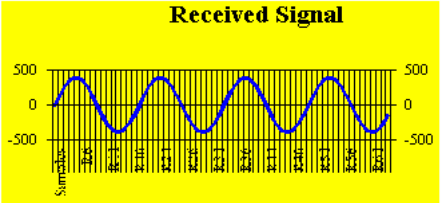
10. 在 signalprog 窗口中的点击 Test ON，这就启动了输入/输出通道。



11. 滑动 Volume Slider 窗口的控制钮，这将改变输出信号幅度，观察 Received Signal 图中信号幅度的变化。

注：Volume Slider 初始化设置

Volume Slider 的初始化设置与其应用程序的运行不是同步进行的，它们在滑动条第一次移动时同步。



12. 关闭 Volume Slider 应用程序，这将废止输入/输出通道。
13. 点击 signalprog 窗口中的 Test OFF，这将关闭控制通道。
14. 点击工具栏按钮　　或按 Shift+F5 中止运行。
15. 现在你可在 Massage Log 窗口中看到调用 LOG_prinft 产生的信息“hostio example started”。由于整个程序的运行都是在主函数中进行的，因此你不能更早地看到上述信息。DSP/BIOS 和主机在 DSP 处于 idle 状态时才进行通信联系。直到应用程序从主函数中返回，DSP 才会处于 idle 状态。所以，如果你想看到运行中 DSP/BIOS 的影响，你的程序应当在从主函数返回后执行 DSP/BIOS 的函数。下一节所用的 hostio.c 的修改版会说明该技术。

7.5 使用 HST 和 PIP 模块修改源程序

现在用 DSP/BIOS 提供的 HST 和 PIP 修改该实例。修改后的程序仍能实时测试 DSP 算法。这次测试数据来自一个主机文件，而不是前述的正弦信号。

HST 模块为执行数据 I/O 提供了与外设接口更直接的通道。HST 模块使用 PIP 模块实现主机 I/O。一旦 I/O 设备和 ISRs 已经做好测试准备，就可使用 PIP 模块的 API，这只需对源程序稍加修改即可。

1. 将目录 c:\ti\c5400\tutorial\hostio2 中的下述文件拷贝到你的工作文件夹下。（不要全部拷贝，特别不能拷贝 hostio.cdb 文件）。
   - hostio.c　已经修改过的源程序，它使用 DSP/BIOS　API 的 HST 和 PIP 模块代替 RTDX 传送输入/输出信号。
   - input.dat　该文件包含输入数据
2. 在 Project View 中的 hostio.c 处，双击鼠标左键，该文件就显示在 CCS 窗口的右半部分，它与前述源程序的区别如下：
   - 增加了两个头文件

     #include <hst.h>

     #include <pip.h>
   - 删除了 BUFSIZE 定义、inp_buffer 和 out_buffer 的全局声明以及 RTDX 的输入和输出通道声明。
   - 将输入/输出函数从主函数中的 while 循环中移到了 A2DscaleD2A 函数中。

```
/* ======== A2DscaleD2A ======== */
/* FUNCTION: Called from A2DscaleD2A_SWI to get digitized data
* from a host file through an HST input channel,
* scale the data by the volume factor, and send
* output data back to the host through an HST
* output channel.
* PARAMETERS: Address of input and output HST channels.
    * RETURN VALUE: None. */
Void A2DscaleD2A(HST_Obj *inpChannel, HST_Obj *outChannel)
{
```

```
PIP_Obj *inp_PIP;
PIP_Obj *out_PIP;
sample *input;
sample *output;
Uns size;
inp_PIP = HST_getpipe(inpChannel);
out_PIP = HST_getpipe(outChannel);
if ((PIP_getReaderNumFrames(inp_PIP) <= 0) ||
(PIP_getWriterNumFrames(out_PIP) <= 0)) {
/* Software interrupt should not have been triggered! */
error();
}
/* Read a new volume when the hosts send it */
if (!RTDX_channelBusy(&control_channel))
RTDX_readNB(&control_channel, &volume, sizeof(volume));
/* A2D: get digitized input (get signal from the host
* through HST). Obtain input frame and allocate output
* frame from the host pipes. */
PIP_get(inp_PIP);
PIP_alloc(out_PIP);
input = PIP_getReaderAddr(inp_PIP);
output = PIP_getWriterAddr(out_PIP);
size = PIP_getReaderSize(inp_PIP);
/* Vector Scale: Scale the input signal by the volume
* factor to produce the output signal. */
while(size--){
*output++ = *input++ * volume;
}
/* D2A: produce analog output (send signal to the host
* through HST). Send output data to the host pipe and
* free the frame from the input pipe. */
PIP_put(out_PIP);
PIP_free(inp_PIP);
```

A2DscaleD2A函数由A2DscaleD2A_SWI对象调用，在下一节中将创建该

SWI对象并使它调用A2DscaleD2A函数。

A2DscaleD2A 函数接收 A2DscaleD2A_SWI 对象传送来的两个 HST 对象，它接着调用 HST_getpipe 函数获取各 HST 对象所采用的内部 PIP 对象的地址。

调用 PIP_getReaderNumFrames 函数和 PIP_getWriteNumFrames 函数可确定是否在输入流水线中至少有一帧数据可读取，输出管道中是否至少有一帧数据可写入。

使用与 7.2 节中同样的 RTDX 调用，该函数获取由 RTDX 控制通道中设置的幅度值。

调用 PIP_get 可从输入流水线中获取完整的一帧数据。调用 PIP_getReaderAddr 获取输入流水线数据帧的起始地址。调用 PIP_getReaderSize 获取输入流水线中帧的字长。

调用 PIP_alloc 从输出流水线中获取空帧。调用 PIP_getWriterAddr 获取输出流水线中输出数据帧的起始地址。

接下来该函数将输入信号乘以幅度值并将结果写入 PIP_getWriterAddr 提供的地址中。

调用 PIP_put 将一个完整帧写入到输出流水线中。调用 PIP_free 重复利用输入帧以便函数下次运行时使用。

- 增加了错误检测函数，它将出错信息写入 trace 日志中并将程序置于无限循环状态。如果 A2DscaleD2A 函数无有效的数据可供处理时，该函数将被调用。

7.6 HST 和 PIP 资料

每个主机通道都在内部都使用一个流水线。当使用主机通道时，目标应用程序管理流水线得一端，主机通道控制插件管理流水线得另一端。

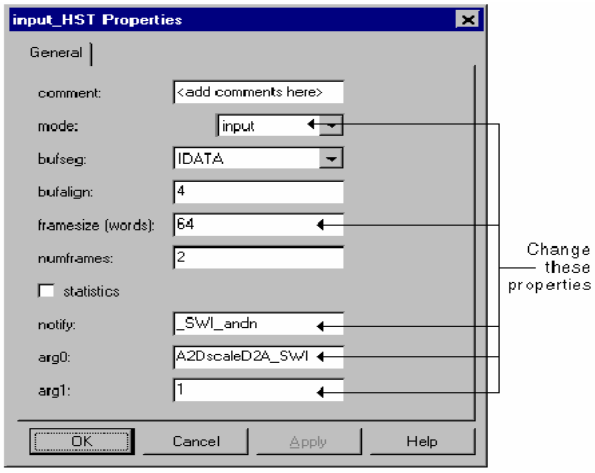
当准备使用外设而不是使用主机修改程序时，可保留管理流水线目标板端的源代码，并在函数中增加处理管理流水线另一端的 I/O 设备的源代码。

7.7 在配置文件中增加通道和 SWI

A2DscaleD2A 函数被 SWI 对象调用并使用两个 HST 对象，本节将创建这些对象（c:\ti\c5400\tutorial\hostio2\hostio.cdb 文件中已经包含了这些对象）。

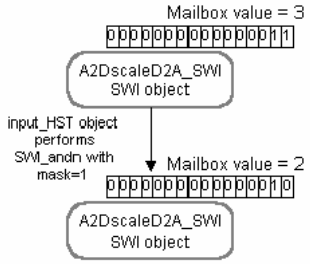
A2DscaleD2A 函数还与两个 PIP 对象有关，不过它们是在你创建 HST 对象时自动生成的。HST_getpipe 函数获取相应 HST 对象的内部 PIP 对象的地址。

1. 在 Project View 的 HOSTIO.CDB 处，双击鼠标左键打开该文件。
2. 在 HST manager 处点击鼠标右键并选择 Insert HST。
   注意，存在名为 RTA_fromHost 和 RTA_toHost 的 HST 对象，它们是内部用来修改 DSP/BIOS 控制的。
3. 将对象名 HST0 修改为 input_HST。
4. 在 input_HST 处点击鼠标右键并从弹出菜单中选择 Properties，然后设置该对象的属性如下图所示并点击 OK。



- mode　该属性决定目标程序和主机通道控制插件各管理的流水线的哪一端。输入通道从主机向目标板发送数据，输出通道从目标板向主机发送数据。
- framesize　它设置每帧的长度，等同于 7.2 节定义的 BUFSIZE，长度为 64 字
- notify,arg0,arg1　当输入通道中包含一个完整的数据帧时，这些属性指定调用的函数及该函数的输入参数，SWI_andn 函数提供了操

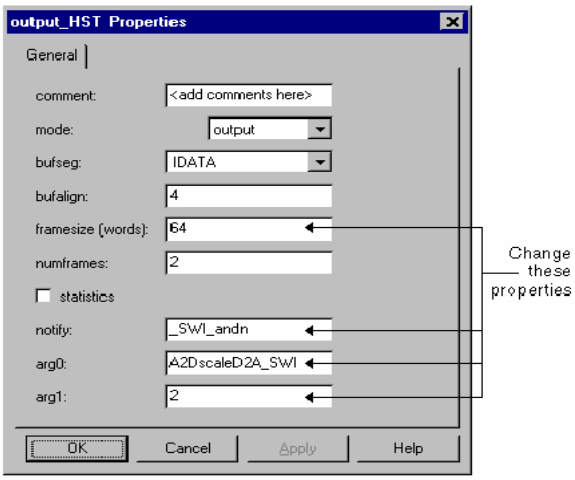
作 SWI 对象的 mailbox 的另一方法。



在第五章，你使用 SWI_dec 函数递减 mailbox 值，并在 mailbox 值抵达 0 时运行 SWI 对象的函数。

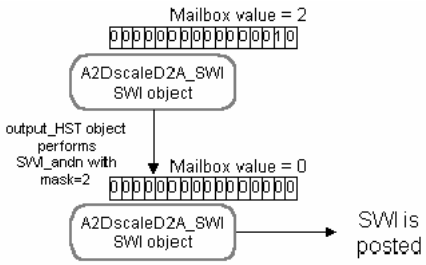
SWI_andn 函数把 mailbox 值当作掩码，它将传递给函数的第 2 个参数所指定的比特位清 0。因此，当输入通道包含一个完整帧时，A2DscaleD2A_SWI 对象的 SWI_andn 函数被调用且 mailbox 的 bit0 被清零。

5. 插入另一个 HST 对象并命名为 output_HST。
6. 为 output_HST 对象设置如下属性并点击 OK。



当输出通道中含有一个空帧时，它用 SWI_andn 清除 mailbox 的 bit1。

7. 在 SWI manager 处点击鼠标右键并选择 Insert SWI。
8. 将新对象 SWI0 命名为 A2DscaleD2A_SWI。
9. 设置 A2DscaleD2A_SWI 属性并点击 OK。



- function　当软中断被登记并执行时，该属性使该对象调用函数 A2DscaleD2A。
- mailbox　mailbox 初始值。input_HST 对象清除掩码的第一比特，output_HST 清除掩码的第二比特，当运行 A2DscaleD2A 函数时，mailbox 值被复位为 3
- arg0,arg1　两个 HST 对象，它们作为 A2DscaleD2A 函数的输入参数

10. 选择 File→Close，你将被询问是否保存对 hostio.cdb 的修改，点击 Yes 保存配置，同时自动产生 hostiocfg.cmd、hostiocfg.s54 和 hostiocfg.h54。

7.8 运行修改后的程序

1. 点击工具栏按钮　　或选择 Project→Rebuild All。
2. 选择 File→Load Program。选择 hostio.out 并点击 Open。
3. 选择 Tools→DSP/BIOS→Host Channel Control，Host Channel Control 列出了 HST 对象，它允许你把它们和 PC 机上的文件捆绑在一起，也可以启动和废止通道。

Host Channel Control

Channel	Transfer...	Limit	State	Mode	Binding
input_HST	0 B	0 KB	Unbound	Input	<unbound>
output_HST	0 B	0 KB	Unbound	Output	<unbound>

4. 点击工具栏按钮　　或选择 Debug→Run。
5. 在 input_HST 通道处点击鼠标右键并从弹出菜单在选择 Bind。
6. 从你的工作文件夹下选择文件 input.dat 并点击 Bind。
7. 在 output_HST 通道处点击鼠标右键并从弹出菜单在选择 Bind。

Host Channel Control

Channel	Transfer...	Limit	State	Mode	Binding
input_HST	0 B	0 KB	Stopped	Input	C:\ti\myprojects\hostio\input.dat
output_HST	0 B	0 KB	Stopped	Output	C:\ti\myprojects\hostio\output.dat

8. 在 File Name 框中敲入 output.dat 并点击 Bind。
9. 在 input_HST 通道处点击鼠标右键并从弹出菜单在选择 Start。
10. 在 output_HST 通道处点击鼠标右键并从弹出菜单在选择 Start，注意在 Transferred 栏中显示数据正在被传送。
11. 当数据传送完毕，点击　　或者按 Shift+F5 中止运行程序。

7.9 进一步学习

有关 RTDX、HST、PIP 和 SWI 模块信息，请参见在线帮助或 TMS320C54x DSP/BIOS User's Guide（用户指南）。

第一章　CCS 概述 1

1.1 CCS 概述 1
1.2 代码生成工具 3
1.3 CCS 集成开发环境 5
1.3.1 编辑源程序 5
1.3.2 创建应用程序 6
1.3.3 调试应用程序 6
1.4 DSP/BIOS 插件 7
1.4.1 DSP/BIOS 配置 7
1.4.2 DSP/BIOS API 模块 8
1.5 硬件仿真和实时数据交换 10
1.6 第三方插件 13
1.7 CCS 文件和变量 14
1.7.1 安装文件夹 14
1.7.2 文件扩展名 14
1.7.3 环境变量 15
1.7.4 增加 DOS 环境空间 16

第二章　开发一个简单的应用程序 17

2.1 创建工程文件 17
2.2 向工程添加文件 19
2.3 查看源代码 21
2.4 编译和运行程序 23
2.5 修改程序选项和纠正语法错误 24
2.6 使用断点和观察窗口 26
2.7 使用观察窗口观察 STRUCTURE 变量 28
2.8 测算源代码执行时间 29
2.9 进一步探索 31
2.10 进一步学习 31

第三章　开发 DSP/BIOS 程序 32

3.1 创建配置文件 32
3.2 向工程添加 DSP/BIOS 文件 34
3.3 用 CCS 测试 36

3.4 测算 DSP/BIOS 代码执行时间.....38
3.5 进一步探索.....40
3.6 进一步学习.....40

第四章 算法和数据测试.....41

4.1 打开和查看工程.....41
4.2 查看源程序.....43
4.3 为 I/O 文件增加探针断点.....45
4.4 显示图形.....47
4.5 执行程序和绘制图形.....48
4.6 调节增益.....50
4.7 观察范围外变量.....51
4.8 使用 GEL 文件.....53
4.9 调节和测试 PROCESSING 函数.....54
4.10 进一步探索.....56
4.11 进一步学习.....57

第五章 程序调试.....58

5.1 打开和查看工程.....58
5.2 查看源程序.....60
5.3 修改配置文件.....63
5.4 用 EXECUTION GRAPH 查看任务执行情况.....66
5.5 修改和查看 LOAD 值.....67
5.6 分析任务的统计数据.....70
5.7 增加 STS 显式测试.....72
5.8 观察显式测试统计数据.....73
5.9 进一步探索.....75
5.10 进一步学习.....75

第六章 实时分析.....76

6.1 打开和查看工程.....76
6.2 修改配置文件.....77
6.3 查看源程序.....79
6.4 使用 RTDX 控制修改运行时的 LOAD 值.....81
6.5 修改软中断优先级.....84

6.6 进一步探索 85
6.7 进一步学习 85

第七章 I/O 86

7.1 打开和查看工程 86
7.2 查看源程序 87
7.3 SIGNALPROG 应用程序 90
7.4 运行应用程序 91
7.5 使用 HST 和 PIP 模块修改源程序 93
7.6 HST 和 PIP 资料 96
7.7 在配置文件中增加通道和 SWI 97
7.8 运行修改后的程序 100
7.9 进一步学习 100