

# 第 1 章 DSP/BIOS 概述

DSP/BIOS 是一个尺寸可伸缩的实时内核,它是为那些需要实时线程调度与同步、主机与目标 DSP 间通信或实时监测的应用而设计的。DSP/BIOS 提供了抢占式多线程、硬件抽象、实时分析和配置工具。

## 1.1 DSP/BIOS 的特色与优点

DSP/BIOS 在设计上采用了以下一些技术来最小化目标 DSP 上的存储器需求和 CPU 开销:

- ❑ 所有的 DSP/BIOS 对象都可以在配置工具中静态建立并被绑定到可执行程序中,不仅减小了代码量,还优化了内部数据结构。
- ❑ 监测数据(如日志和统计数据)在主机端(而非目标 DSP 端)被格式化处理。
- ❑ API 函数被模块化,因此只有那些应用程序用到的 API 函数才会被绑定到可执行程序中。
- ❑ 大部分 API 库函数使用汇编语言实现优化,使得其执行所需的指令周期数达到了可能的最小值。

- ❑ 目标 DSP 和主机 DSP/BIOS 分析工具之间的通信在后台空闲循环中完成,以确保分析工具不会影响应用程序所要完成的任务。如果 CPU 太忙以致不能执行后台任务时,DSP/BIOS 分析工具会暂时停止从目标 DSP 接收信息直到 CPU 空闲。
- ❑ 错误检测所需的存储器和 CPU 开销被限制到最小。但作为交换,也引入了一些 API 函数调用时的约束,用户应参考 API 手册中的详细说明,在开发时满足这些约束。

此外,DSP/BIOS API 还为程序的开发提供多种灵活的选择:

- ❑ 一些用于特殊情况下的 DSP/BIOS 对象可以在应用程序中动态建立或删除。应用程序既可以使用动态建立的对象,也可以使用静态建立的对象。
- ❑ 提供了多种类型的线程以满足各种情况的需要:硬件中断、软件中断、任务、空闲函数、周期函数等。用户还可以控制线程的优先级和阻塞(blocking)特性。
- ❑ 提供了实现线程间通信与同步的数据结构,包括信号灯、邮箱和资源锁。
- ❑ 提供了两种 I/O 模型:“管道”和“流”,以提供最大的灵活性和输入/输出能力。其中“管道”用于目标 DSP/主机之间的通信以及线程间的简单数据传输。“流”则用于更为复杂的 I/O 操作并且支持设备驱动。
- ❑ 提供的低级系统原语可以简化错误处理,简化一般数据结构的建立以及简化存储器使用的管理。

DSP/BIOS API 标准化了对 TI 大部分 DSP 器件的编程,并且提供了强大而且易用的

程序开发工具,这些工具通过以下途径缩短了建立 DSP 应用程序的时间:

- ❑ 配置工具可以自动生成代码来静态声明程序中用到的 DSP/BIOS 对象。
- ❑ 配置工具可以在应用程序编译链接之前通过校验属性提前检测出错误。
- ❑ 使用 Tconf 脚本语言可以进一步增强 DSP/BIOS 配置。用户可使用文本编辑器修改配置脚本,如向其中加入条件分支、循环、命令行参数检测等。
- ❑ 在程序运行时自动对 DSP/BIOS 对象信息进行记录和统计,无须额外编程,其他的监测则是根据需要编程实现。
- ❑ DSP/BIOS 分析工具提供了对程序行为的实时监测功能。
- ❑ DSP/BIOS 提供了一个标准 API,使得 DSP 算法开发者开发出的代码更易于整合到其他应用程序中。
- ❑ DSP/BIOS 和 Code Composer Studio IDE 集成在一起,免费且无须任何运行许可证,并由 TI 公司提供全方位的支持。DSP/BIOS 是 TI 的 eXpress DSP™ 实时软件技术的关键组件。

## 1.2 DSP/BIOS 组件

图 1-1 给出了在 CCS 程序生成和调试环境中的 DSP/BIOS 组件。

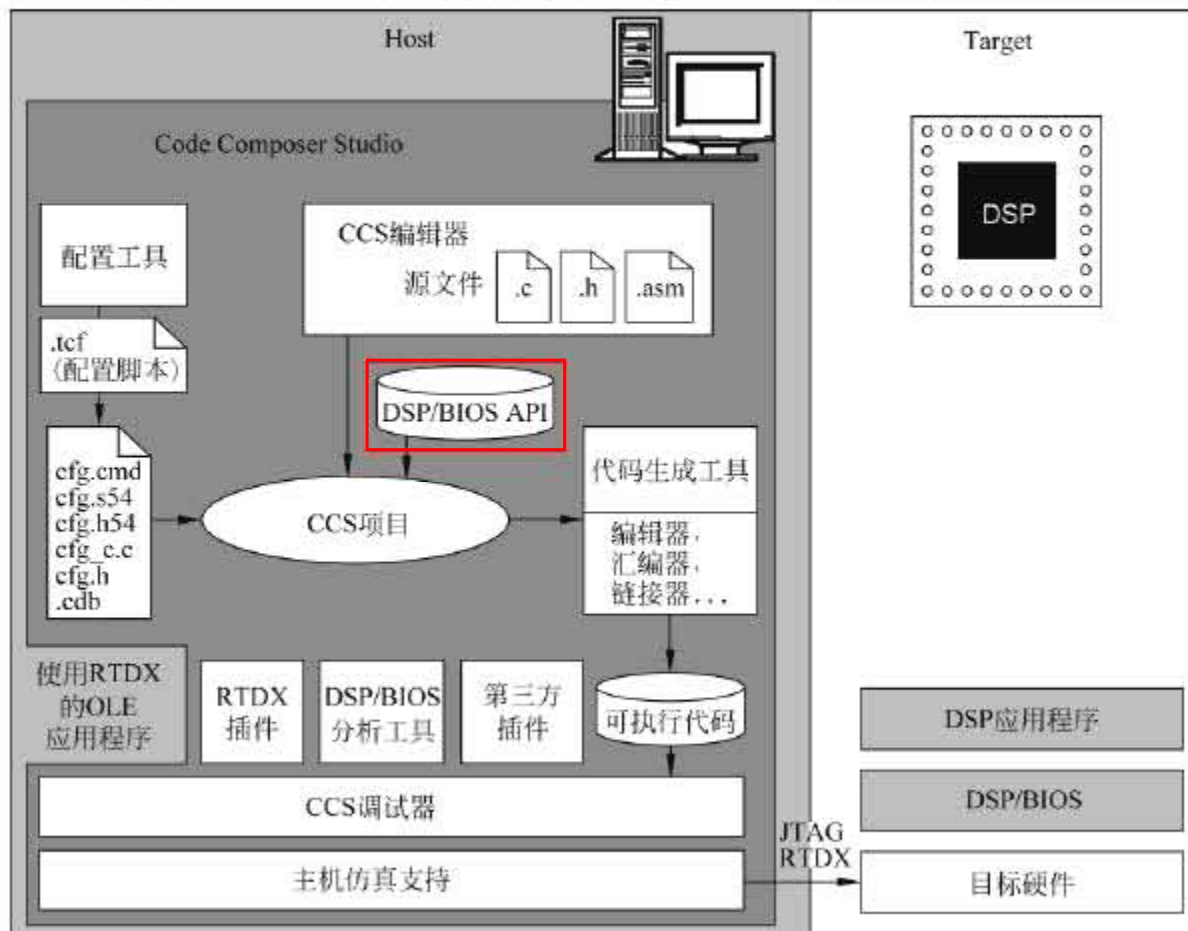


图 1-1 DSP/BIOS 组件



- ❑ **DSP/BIOS API:** 用户在 PC 端使用 C、C++ 或汇编语言编写调用了 DSP/BIOS API 函数的应用程序。
- ❑ **DSP/BIOS 配置:** 用户创建一个 DSP/BIOS 配置, 定义了程序中要使用的静态对象。并且该配置会生成相应的代码文件, 和应用程序一起进行编译链接。
- ❑ **DSP/BIOS 分析工具:** CCS 中的分析工具使用户可以测试和分析目标 DSP 上应用程序的运行, 包括对 CPU 负荷、日志、线程执行情况的监测等。

下面对 DSP/BIOS 组件做简要介绍。

### 1.2.1 DSP/BIOS 实时内核和 API

DSP/BIOS API 是以模块划分的, 根据应用程序中配置和使用的模块的不同, DSP/BIOS 的代码长度为 500 字到 6500 字不等。一个 DSP/BIOS 模块所有的 API 函数都以其模块名为开头, 表 1-1 列出了所有的 DSP/BIOS 模块名。

表 1-1 DSP/BIOS 模块

模块名称	说 明
ATM	使用汇编语言编写的原子(atomic)函数
BUF	定长缓冲池管理器
C28,C54,C55,C62,C64	目标 DSP 特有函数
CLK	时钟管理器
DEV	设备驱动接口
GBL	全局设置管理器
GIO	通用 I/O 管理器
HOOK	钩子函数管理器
HST	主机通道管理器
HWI	硬件中断管理器
IDL	空闲函数管理器
LCK	资源锁管理器
LOG	事件日志管理器
MBX	邮箱管理器
MEM	存储器管理器
PIP	缓冲管道管理器
PRD	周期函数管理器
PWRM	功率管理器(仅 C55x 具有)
QUE	原子队列管理器
RTDX	实时数据交换设置
SEM	信号灯管理器
SIO	流 I/O 管理器
STS	统计对象管理器
SWI	软件中断管理器
SYS	系统服务管理器
TRC	追踪管理器
TSK	多任务管理器

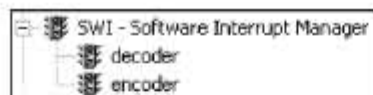
应用程序通过调用 DSP/BIOS API 函数来使用 DSP/BIOS, 所有的 DSP/BIOS 模块都提供 C 调用接口, 此外一些模块还包含优化过的汇编语言宏。在遵循调用规则的情况下, 大多数的 C 调用接口也可以在汇编程序中调用。有一些 C 语言接口实际上是 C 语言宏, 所以汇编程序不能使用。关于所有 DSP/BIOS 模块可用的 C 语言和汇编语言接口的描述, 请查阅相应平台的 TMS320 DSP/BIOS API 函数参考手册。

### 1.2.2 DSP/BIOS 配置

DSP/BIOS 配置允许用户静态地创建对象并设置对象属性, 从而优化用户应用程序, 既改善了程序运行的性能, 也缩短了程序开发的进度。

DSP/BIOS 配置对应的源文件是一个 Tconf 脚本, 其文件扩展名为 .tcf。用户可以通过两种方式得到一个 DSP/BIOS 配置。

- ❑ 图形方式。用户可以使用 DSP/BIOS 配置工具打开并浏览配置脚本。其界面类似于 Windows 的资源管理器, 如图 1-2 所示。当使用图形方式来编辑 DSP/BIOS 配置时, 用户可以在树型视窗中创建对象, 然后在对话框里设置其属性。



- ❑ 文本方式。用户可以使用 CCS 或其他文本编辑器对配置脚本的内容进行编辑。在这种方式下,用户将使用 JavaScript 语法来对配置编程。

```
prog.module("SWI") create("encoder");  
prog.module("SWI") create("decoder");
```

一般情况下,这两种方式可以结合起来使用。图形化的编辑器是创建一个初始配置的好方法,而用文本方式编辑脚本则使得用户可以进一步增强配置。

不论使用哪种方式,用户都可以设置 DSP/BIOS 实时库在运行时所使用的一系列参数。用户创建的对象可以被应用程序的 DSP/BIOS API 调用所使用。这些对象具体可包括软件中断、任务、I/O 流以及事件日志。

当用户对一个配置进行保存时,配置工具自动生成相应的文件并将其包含在当前项目(project)中。当静态配置 DSP/BIOS 时,DSP/BIOS 对象都是在程序运行前被预先设置好并被绑定到可执行程序中的。另外,DSP/BIOS 应用程序也可以在运行时动态地建立和删

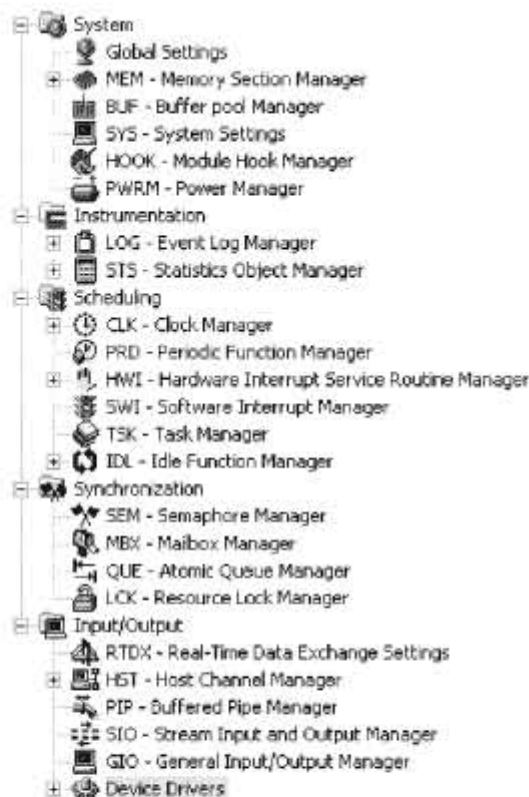


图 1-2 配置工具模块树



### 除对象。

静态地创建对象,除了能通过消除运行时代码和优化内部数据结构来使目标存储器开销减至最小,还可以在程序编译之前通过对象属性有效性检测以及时地发现错误。

更多的细节请参见 DSP/BIOS 在线帮助和 2.2 节“静态配置 DSP/BIOS 应用程序”。

## 1.2.3 DSP/BIOS 分析工具

DSP/BIOS 分析工具可以帮助开发者在 CCS 环境中实现对 DSP/BIOS 应用程序的实时分析,使用户能够以可视化的方式监测一个运行中的 DSP/BIOS 应用程序,而几乎不影响应用程序的实时性能。DSP/BIOS 分析工具可以通过选择 CCS 中的“DSP/BIOS”菜单打开,如图 1-3 所示。

关于每个分析工具的具体细节请参见 DSP/BIOS 在线帮助及第 3 章的内容。

传统的调试方法只能在执行中的程序外部进行,要对程序进行实时调试分析,就需要在目标程序中包含实时监测服务,即在目标 DSP 上运行检测代码。通过使用 DSP/BIOS 的 API 函数和对象,应用程序可以自动地监测目标 DSP,实时采集信息并通过 CCS 分析工具上传到主机。

DSP/BIOS 提供的实时程序分析功能主要包括:

- ❑ 程序追踪:显示被记录到目标日志的事件,反映程序执行过程中的动态控制流。
- ❑ 性能监测:监测能够反映目标资源使用情况的统计信息,如处理器负荷及计时信



图 1-3 CCS 中的 DSP/BIOS 菜单

息等。

□ 文件流：将目标 DSP 端的 I/O 对象绑定到主机文件，形成文件流。

当与 CCS 其他的通用调试功能轮流使用时，DSP/BIOS 实时分析工具能够在执行过程中实时地对目标程序的内部行为进行观察，然而传统的调试方法需要停止目标程序的运行来观察当前的内部行为，因此对程序内部行为的观察能力不足。即使在调试器停止程序之后，用户仍然可以使用主机 DSP/BIOS 分析工具捕获到的历史信息，对程序以往的内部行为进行观察和分析。

在软件开发的后期，当正常的调试手段在那些由时间相关的交互操作所引发的问题面前变得无效时，DSP/BIOS 分析工具则会像一个软件版的硬件逻辑分析器那样，完成一些特殊任务。

图 1-4 给出了若干 DSP/BIOS 分析工具的界面图。图 1-5 给出了 DSP/BIOS 分析工具的工具条，可以通过选择菜单项“View”→“Plug-in Toolbars”→“DSP/BIOS”显示和隐藏该工具条。

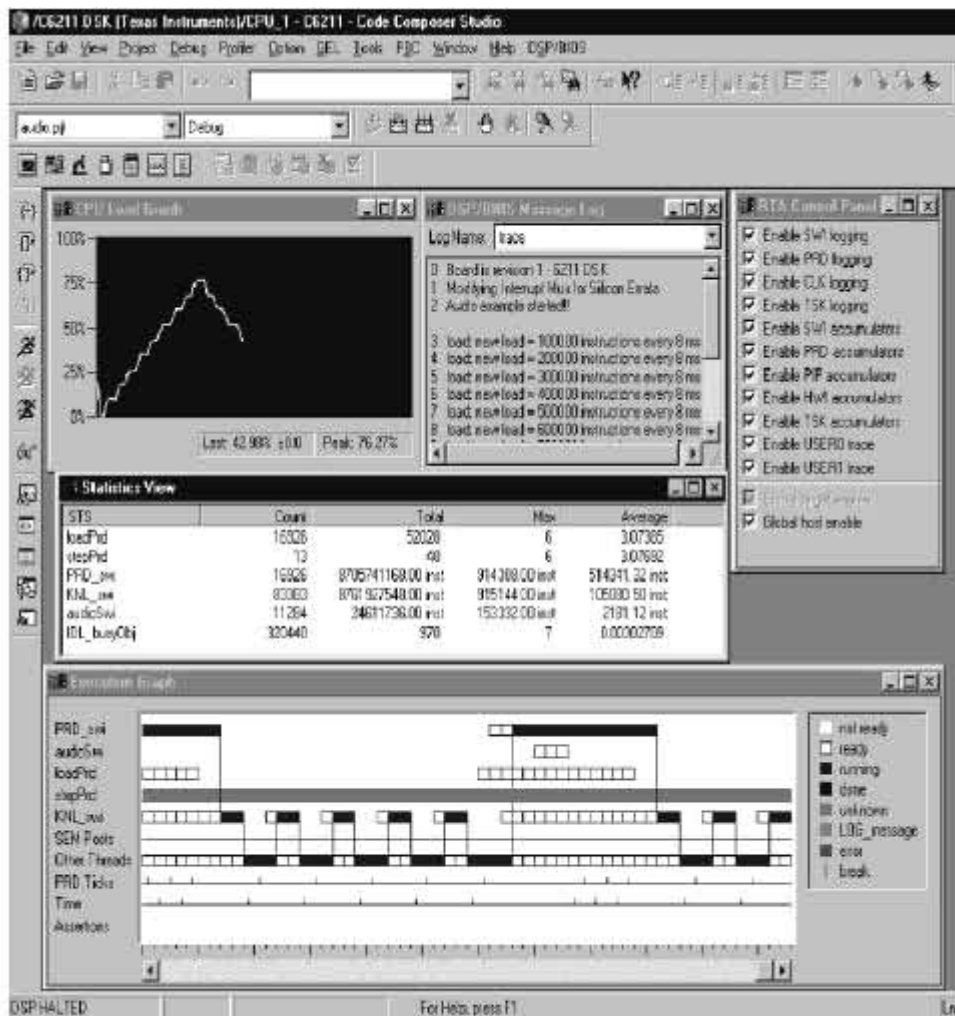


图 1-4 CCS 分析工具面板



图 1-5 DSP/BIOS 分析工具的工具条

## 1.3 命名规则

DSP/BIOS 的每个模块都有一个唯一的名称,用来作为该模块的对象名、操作(函数)名或头文件的前缀。这个名称通常由 3 个或更多个大写字母和数字组成。

文中,DSP/BIOS 生成的源文件名称中的 54 这两个数字代表用户使用的具体 DSP 平台。如果用户 DSP 平台是 C6200,用 62 代替 54 即可。如一个 C6200 平台的 DSP/BIOS 汇编语言头文件的后缀名为, h62。如果用户 DSP 平台为 C55,用 55 代替 54 即可。

所有以 3 个大写字母和一个下划线开始的标识符(XXX\_\*)都被当作保留字。



### 1.3.1 模块头文件名

每个 DSP/BIOS 模块都有两个头文件,包含该模块接口中所有可用的常量、数据类型和函数的声明。

- ❑ **XXX.h:** 为 C 程序提供的 DSP/BIOS API 头文件。用户的 C 语言源文件应该包含 std.h 头文件以及使用到的所有模块的头文件。
- ❑ **XXX.h54:** 为汇编程序提供的 DSP/BIOS API 头文件。用户的汇编源文件应该包含所有使用到的模块的头文件。该头文件中包含了相应器件特有的一些宏定义。

用户应用程序必须在特定的程序源文件中包含每个用到的模块的头文件,另外 C 语言源文件中还必须在模块头文件之前包含 std.h,该头文件中含有标准类型和常量的定义。在 std.h 头文件之后,其他头文件的次序可以任意。例如:

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

DSP/BIOS 有一些模块是在内部使用的,这些模块没有正式的文件说明,并在不断更新之中。但这些内部模块的头文件作为 DSP/BIOS 的一部分,也会在编译和链接 DSP/BIOS 程序时被自动包含进来。

### 1.3.2 对象名称

在 DSP/BIOS 配置中默认包含的系统对象,其名称都以 3 个或 4 个字母的模块名称为开头。如默认配置中包含一个名为 LOG\_system 的 LOG 对象。

注意:由用户静态创建的对象可由用户按照某种共同的规则来命名。用户可以将模块名作为对象名的后缀,例如,可将一个对数据进行编码的 TSK 对象命名为 encoderTsk。

### 1.3.3 操作名

一个 DSP/BIOS API 函数的命名格式为 MOD\_action,其中 MOD 为拥有该操作的模块的字母代码,action 代表该操作具体执行的动作。例如,SWI\_post 函数由 SWI 模块定义,它会触发一个软件中断。

DSP/BIOS API 函数中有许多是系统内建(built-in)的,它们由各种系统内建的对象使用,以下给出一些例子:

- ❑ **CLK\_F\_isr**: 由一个 HWI 对象使用,提供低分辨率 CLK 时钟。
- ❑ **PRD\_F\_tick**: 由 CLK 对象 PRD\_clock 使用,用来管理 PRD\_swi 和系统时钟。

- ❑ **PRD\_F\_swi**: 被 PRD\_tick 触发, 用来运行 PRD 函数。
- ❑ **KNL\_run**: 由最低优先级的 SWI 对象 KNL\_swi 调用, 来运行任务调度。这其实是一个名为 KNL\_run 的 C 函数。如果在汇编代码里进行调用, 就需要在函数名前面加上一个下划线。
- ❑ **IDL\_loop**: 由最低优先级的 TSK 对象 TSK\_idle 调用, 来运行 IDL 函数。
- ❑ **IDL\_F\_busy**: 由 IDL 对象 IDL\_cpuLoad 调用, 来计算当前的 CPU 负荷。
- ❑ **RTA\_F\_dispatch**: 由 IDL 对象 RTA\_dispatcher 调用, 来采集实时分析数据。
- ❑ **LNK\_F\_dataPump**: 由 IDL 对象 LNK\_dataPump 调用, 来管理实时分析数据和主机通道数据向主机的传输。
- ❑ **HWI\_unused**: 并非函数名, 该字符串在配置中用来标记未使用的 HWI 对象。

注意: 用户应用程序代码中不能调用任何以 MOD\_F 开头的内建函数。以 MOD\_和 MOD\_F 形式开头的符号名称被保留由系统内部使用。

### 1.3.4 数据类型名

DSP/BIOS API 不直接使用 C 语言的基本数据类型,如 int 或 char。相反的,为了确保对其他支持 DSP/BIOS API 的处理器可移植性,DSP/BIOS 定义了自己的标准数据类型。这些数据类型如表 1-2 所示,它们都定义在 std.h 头文件里。

表 1-2 DSP/BIOS 标准数据类型

类 型	说 明
Arg	既能携带指针又能携带 Int 类型参数的数据类型
Bool	布尔型数据
Char	字符数据
Fxn	指向函数的指针
Int	带符号整型数据
LgInt	带符号长整型数据
LgUns	无符号长整型数据
Pir	通用指针
String	以 \0 结束的字符串
Uns	无符号整型数据
Void	空类型

在 std.h 中还定义了一些数据类型,但并不被 DSP/BIOS API 使用。

另外,在 DSP/BIOS 中,标准常量 NULL(0)用来表示一个空的指针值,常量 TRUE(1)和 FALSE(0)用来表示布尔类型的值。

DSP/BIOS API 模块使用的对象结构体都使用 MOD\_Obj 形式的命名约定,其中 MOD



为对象所属模块的字母代码。例如 LOG 对象的结构体名称为 LOG\_Obj。如果用户程序代码使用了任何在配置中创建的对象,该对象都应该被声明为一个外部(extern)结构体变量,如声明 LOG 对象 trace 为外部变量:

```
extern LOG_Obj trace;
```

配置工具会自动生成一个头文件(<program>cfg.h)来包含所有由配置工具创建的 DSP/BIOS 对象的声明,所以在应用程序源文件中包含该头文件亦可。



早期的 C54x 平台的 DSP/BIOS 采用 16 位寻址模式,而新型的 C54x 器件支持扩展寻址模式,并且对 DSP/BIOS 进行了相应的改进使其能够工作在新寻址模式下。详细信息请查阅 TI 的应用笔记,“*DSP/BIOS and TMS320C54x Extended Addressing*,SPRA599”。

### 1.3.5 存储器段命名

DSP/BIOS 使用的存储器段名(memory segment),如表 1-3 所示,用户可以改变配置中大多数默认存储器段的名称、首地址和长度。

表 1-3 存储器段名称



## a. C54x 平台

存储器段	说 明
IDATA	内部(片上)数据存储器
EDATA	外部数据存储器的第一块(primary block)
EDATA1	外部数据存储器的第二块(secondary block)(与 EDATA 不连续)
I PROG	内部(片上)程序存储器
EPROG	外部程序存储器的第一块(primary block)
EPROG1	外部程序存储器的第二块(secondary block)(与 EPROG 不连续)
USERREGS	第 0 页的用户存储器(28 字)
BIOSREGS	第 0 页保留寄存器(4 字)
VECT	中断向量段



## b. C55x 平台

存储器段	说 明
IDATA	数据存储器的第一块(primary block)
DATA1	数据存储器的第二块(secondary block)(与 IDATA 不连续)
PROG	程序存储器
VECT	DSP 中断向量表存储器段



c. C6000 EVM 平台

存储器段	说 明
IPRAM	内部(片上)程序存储器
IDRAM	内部(片上)数据存储器
SBSRAM	位于 CE0 空间的外部 SBSRAM
SDRAM0	位于 CE2 空间的外部 SDRAM
SDRAM1	位于 CE3 空间的外部 SDRAM



d. C6000 DSK 平台

存储器段	说 明
SDRAM	外部 SDRAM



e. C2800 DSK 平台

存储器段	说 明
BOOTROM	引导代码存储器
FLASH	内部 FLASH 程序存储器
VECT	中断向量表 (VMAP=0 时)
VECT1	中断向量表 (VMAP=1 时)
OTP	通过 FLASH 寄存器编程的一次性可编程存储器
H0SARAM	内部程序 RAM
L0SARAM	内部数据 RAM
M1SARAM	内部用户和任务堆栈 RAM

### 1.3.6 标准存储段

DSP/BIOS 配置中定义的存储段(memory section)及其分配情况如表 1-4 所示,用户可以使用 MEM 管理器来更改这些默认的分配。详细信息请查阅相应平台的 TMS320 DSP/BIOS API 参考手册中关于 MEM 模块的介绍。

表 1-4 标准存储段



a. C54x 平台

存储段(section)	存储器段(segment)
系统堆栈存储段 (.stack)	IDATA
应用程序参数存储段 (.args)	EDATA
应用程序常量存储段 (.const)	EDATA
BIOS 程序存储段 (.bios)	IPROG
BIOS 数据存储段 (.sysdata)	EDATA
BIOS 存储堆存储段	IDATA
BIOS 启动代码存储段 (.sysinit)	EPROG





### b. C55x 平台

存储段(section)	存储器段(segment)
系统堆栈存储段 (.stack, .sysstack)	DATA
BIOS 内核状态存储段 (.sysdata)	DATA
BIOS 对象,配置存储段 (. * obj)	DATA
BIOS 程序存储段 (.bios)	PROG
BIOS 启动代码存储段 (.sysinit, .gblinit, .trcinit)	PROG
应用程序参数存储段 (.args)	DATA
程序存储段 (.text)	PROG
BIOS 存储堆存储段	DATA
第二个 BIOS 存储堆存储段	DATA1



### c. C6000 平台

存储段(section)	存储器段(segment)
系统堆栈存储段 (.stack)	IDRAM
应用程序常量存储段 (.const)	IDRAM
程序存储段 (.text)	IPRAM
数据存储段 (.data)	IDRAM
启动代码存储段 (.sysinit)	IPRAM
C 初始化记录存储段 (.cinit)	IDRAM
未初始化变量存储段 (.bss)	IDRAM



#### d. C2800 平台

存储段(section)	存储器段(segment)
系统堆栈存储段 (.stack)	MISARAM
程序存储段 (.text)	IPROG
数据存储段 (.data)	IDATA
应用程序常量存储段 (.const)	IDATA
启动代码存储段 (.sysinit)	IPROG
C 初始化记录存储段 (.cinit)	IDATA
未初始化变量存储段 (.bss)	IDATA

## 1.4 更多的信息

有关 DSP/BIOS 组件及 DSP/BIOS API 模块的更多信息,用户可以查看在线帮助系统中的 DSP/BIOS 一节、相应平台的 TMS320 DSP/BIOS API 函数参考手册或 CCS 在线指南中“Using DSP/BIOS”部分。

# 第 2 章 程序生成

这一章介绍 DSP/BIOS 开发程序的过程,并讲解 DSP/BIOS 生成了哪些文件以及如何使用这些文件。

## 2.1 开发过程

DSP/BIOS 支持反复式的(iterative)程序开发过程,用户首先为应用程序建立一个基本框架,并在加入 DSP 算法之前加入一个模拟出来的处理负载来测试程序。这样用户就能方便地对那些完成不同功能的程序线程的优先级和类型进行修改。

一个典型的 DSP/BIOS 程序开发过程包含以下步骤,其中有些步骤可能需要反复进行。

(1) 配置应用程序要用到的静态对象。这可以使用配置工具或 Tconf 脚本语言,或把它们结合起来使用来完成配置。

(2) 在 DSP/BIOS 配置工具中保存配置文件,保存的同时自动生成应用程序编译和链接时需要的文件。

(3) 为应用程序编写一个框架,可以使用 C、C++、汇编语言或这些语言的混合编程。

(4) 将文件添加到项目中,使用 CCS 对项目里的文件进行编译链接。

(5) 使用软件模拟器(simulator)或初始硬件平台和 DSP/BIOS 分析工具来测试应用程序行为,用户可以监测日志、对象的统计信息、时序、软件中断等。

(6) 重复步骤(1)到步骤(5),直至程序运行正确。然后可以对程序结构进行修改以及增加其他的功能函数。

(7) 当正式产品硬件平台开发好后,根据产品硬件平台修改配置文件并且在硬件板上运行程序进行测试。

## 2.2 静态配置 DSP/BIOS 应用程序

DSP/BIOS 配置允许用户静态地创建对象并设置其属性,用户可以选择以图形方式、文本方式或者两种方式的结合来创建一个配置。

DSP/BIOS 在线帮助从各个方面更详细地讲解了如何一步一步对配置进行操作。另外

DSP/BIOS 文本配置 (Tconf) 用户手册 (*DSP/BIOS 5.3 Textual Configuration (Tconf) User's Guide*, SPRU007H) 详细介绍了配置脚本时使用的语法。

## 2.2.1 使用图形化配置工具

使用 DSP/BIOS 图形化配置工具有以下的优势：

- ❑ 树型视图界面能帮助用户方便地查看每个模块和每个对象的可用属性。
- ❑ 该图形化界面提供了包含有效值的下拉列表，剔除无效的命令和域值，从而避免配置过程中产生错误。

用户可以使用文本编辑器修改一个配置脚本，然后将其重新装载到 DSP/BIOS 配置工具中作进一步的图形化编辑。但这种方式有时会有一些限制。

## 2.2.2 使用文本编辑器

使用文本编辑器修改一个配置脚本具有以下优势：

- ❑ 允许用户在配置脚本中使用条件分支、循环及其他结构。
- ❑ 允许用户创建许多类似的对象。这可以通过复制、粘贴操作或将创建函数循环执行多次来实现。

- ❑ 允许用户将一些设置脚本模块化以便在其他一系列应用程序中使用。比如用户所有的应用程序都要使用类似的监测对象,只要创建一个单独配置这些对象的配置脚本文件,然后在所有程序中都包含此配置脚本就可以了。
- ❑ 允许配置脚本像程序源文件那样使用相同的符号定义。这可以通过定义脚本中要使用的变量并从脚本生成一个 C 头文件,再使用 `#include` 将其包含在程序源代码中来实现。
- ❑ 用户可以向脚本传递命令行参数来使配置在某方面进行自动调整。例如,用户可能需要通过改变程序中任务的个数并测试相应的结果,来优化程序,那么只要改变相应的参数就可以方便地创建任务数不同的 DSP/BIOS 配置。
- ❑ 作为标准的代码编辑方式,允许多个开发者一同开发,比较应用程序的配置,在不同应用程序配置间剪切、粘贴。
- ❑ 可以在 UNIX 上使用。

不要将 DSP/BIOS 配置和 CCS 中使用的其他配置项相混淆,如项目配置(Debug 或 Release)、RTDX 配置以及 CCS Setup 工具中的系统配置设置。

### 2.2.3 配置 DSP/BIOS 应用程序的步骤简介

下面给出了采用图形方式配置 DSP/BIOS 的一般步骤。关于每一步更详细的讲解请查阅 DSP/BIOS 在线帮助中的“创建 DSP/BIOS 配置”部分。

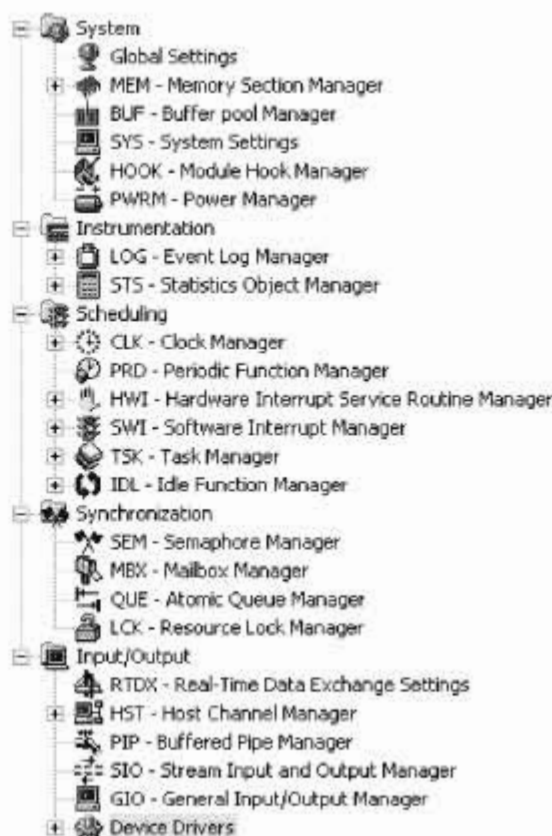
(1) 在 CCS 中选择“File”→“New”→“DSP/BIOS Configuration”打开 DSP/BIOS 图形化配置工具,如图 2-1 所示。

(2) 设置脚本环境变量。

(3) 在“New DSP/BIOS Configuration”窗口中,选择一个 DSP 平台和 DSP/BIOS 配置模板。或者,用户也可以新建一个配置模板。

(4) 在配置(Configuration)窗口中,根据应用程序需要执行下列的操作:

- 创建应用程序中要使用的对象。右键单击相应的模块管理器,选择“Insert MOD”命令(MOD 代表模块名)。
- 命名创建的对象。右键单击该对象,选择“Rename”命令。
- 设置应用程序的全局属性。
- 修改模块管理器属性。右键单击该模块管理器,选择“Properties”命令打开其属性对话框进行设置。





- 修改对象属性。右键单击该对象,选择 “Properties”命令打开其属性对话框进行设置。
- 设置软件中断和任务的优先级。同样在其属性对话框中设置。
- 为 Tconf 脚本添加注释行。

(5) 保存配置,同时会自动生成项目需要的文件。

(6) 将生成的文件添加到项目中。

(7) 编译和链接应用程序并根据需要进行测试。

(8) 可以使用图形方式或文本方式编辑脚本,进一步修改 DSP/BIOS 配置。

## 2.2.4 引用静态创建的 DSP/BIOS 对象

在应用程序中引用静态创建的对象,需要在所有函数体外使用 `extern` 关键字将其声明为外部变量。例如下面的声明使得管道对象 `inputObj` 可以被程序里在该声明之后定义的所有函数引用。

```
extern far PIP_Obj inputObj;           /* C6000 器件 */
```

或者:

```
extern PIP_Obj inputObj;               /* C5000 和 C2800 器件 */
```

配置工具会在一个名称为 `*.cfg.h` 的头文件中自动生成这些声明,其中 `*` 为应用程序

名。所以用户也可以在 C 源文件中使用 `#include` 语句包含该头文件来引用 DSP/BIOS 对象。

#### 2.2.4.1 C6000 的小模式和大模式问题



虽然 DSP/BIOS 是按照小模式编译的,但对于 C6000 平台,用户的 DSP/BIOS 应用程序则可以按照 C6000 编译器的小模式或者各种等级的大模式编译(参见《TMS320C6000 优化编译器用户手册》)。实际上只要保证所有的使用基于 B14(DP 数据页指针)的偏移量访问的全局数据都位于 `bss` 段的起始 32K 字节内,应用程序的代码就可以混合使用不同的编译模式。

DSP/BIOS 也使用 `bss` 段存储全局数据,但静态配置的对象并不放置在 `bss` 段内,这是为了使应用程序数据的放置更加灵活。例如,会被频繁访问的 `bss` 段可以放在片内存储空间,而可以将大量的很少被访问的对象存储在片外存储空间。

然而在小模式编译模式下,编译器会对全局数据的放置位置进行假设(`bss` 段的起始 32K 字节内),目的在于减少存取数据的指令周期数。所以用户如果要使用小模式(默认的编译模式)来优化全局数据的存取,那么可能需要修改代码以保证正确引用静态建立的对象。

有 4 种方法可以解决这些问题,将在后面的小节中详细说明,表 2-1 为这些解决方法的优缺点对照表。

表 2-1 引用 C6000 全局对象的方法

方 法	用 far 关键字声明对象	使用全局对象指针	紧邻着 .bss 段放置对象	使用大模式编译
代码的工作与编译模式无关	是	是	是	是
代码的工作与目标放置位置无关	是	是	否	是
C 代码可以移植到其他编译器	否	是	是	是
静态创建的对象没有 32K 字节的大小限制	是	是	否	是
可以减小 .bss 段的长度	是	是	否	是
可以减少访问操作的指令周期数	否 (3 个指令周期)	否 (2~6 个指令周期)	是 (1 个指令周期)	否 (3 个指令周期)
可以减少每个对象所需的存储空间	否 (12 字节)	否 (12 字节)	是 (4 字节)	否 (12 字节)
易于编程和调试	一般	容易出错	一般	是

#### 2.2.4.2 在小编译模式下引用静态 DSP/ BIOS 对象(C6000 平台)



在

器 B14 被编译器当作一个只读寄存器,并且在程序启动时被初始化为 .bss 段的首地址。全局变量被定位存储于相对 .bss 段首的一个固定偏移地址中,且 .bss 段会被假设最长为 32K 字节。这样,全局变量就可以通过一条指令存取,如下所示:

```
LDW * + DP(_x), A0; load _x into A0 (DP = B14)
```

由于静态创建的对象不是放在 .bss 段内的,所以用户必须确保在小模式编译下的应用程序代码能正确引用这些对象。如下 3 种方法可以保证正确引用:

- ❑ 用 far 关键字声明静态对象, far 关键字指明数据不是在 .bss 段内。例如要引用一个名为 inputObj 的静态建立的 PIP 对象,可以这样声明该对象:

```
extern far PIP_Obj inputObj;  
if (PIP_getReaderNumFrames(&inputObj)) {  
...  
}
```

- ❑ 创建并初始化一个全局对象指针。用户可以创建一个全局变量,并用待引用的对象的地址对其初始化,那么对该对象的所有引用都通过这个指针进行,这样就不必使用 far 关键字了。例如:

```
extern PIP_Obj inputObj;
/* input MUST be a global variable */
PIP_Obj * input = &inputObj;
if (PIP_getReaderNumFrames(input)) {
...
}
```

声明并初始化一个全局指针变量会额外占用一个 32 位的空间(以存放对象的 32 位地址)。

另外,如果该指针被声明为静态变量或自动变量,那么这种方法就会失效。如下代码在小模式编译时将得不到所期望的结果:

```
extern PIP_Obj inputObj;
static PIP_Obj * input = &inputObj; /* ERROR!!!! */
if (PIP_getReaderNumFrames(input)) {
...
}
```

❑ 紧邻着 .bss 段放置所有对象。如果所有对象都紧接着 .bss 段的末端放置,并且这些对象加上 .bss 段内数据的总长度小于 32K 字节,这些变量就会被当作在 .bss 段中的数据一样由用户来引用:

```
extern PIP_Obj inputObj;
if (PIP_getReaderNumFrames(&inputObj)) {
...
}
```

可以通过 MEM 模块的配置来保证上述的对象存放位置,方法如下:

(1) 通过在 MEM 管理器下插入一个新的 MEM 对象来声明一个新的存储器段, 设定其属性(如基地址、长度等), 保证其长度为 32K 字节; 也可以使用现有的 MEM 对象。

(2) 将小编译模式下要引用的所有的对象都放置在该存储器段中。

(3) 右键单击 MEM 管理器选择属性, 将未初始化变量存储段(.bss 段)也放置在该存储器段中。

#### 2.2.4.3 在大编译模式下引用静态 DSP/ BIOS 对象(C6000 平台)



在大编译模式下, 所有经过编译的代码在访问数据时, 都是先将变量的完整 32 位地址装入一个地址寄存器中, 然后通过 LDW 指令的间接寻址能力来装载该数据, 例如:

```
MVKL _x, A0; move low 16-bits of _x's address into A0  
MVKH _x, A0; move high 16-bits of _x's address into A0  
LDW *A0, A0; load _x into A0
```

所以静态对象的存储位置不会对任意等级大模式下编译的代码造成影响。对于静态建立的对象, 不管它们存放在哪里, 如果所有直接引用它们的代码都是在任意等级的大模式选项下编译的, 那么这些代码可以像存取一般数据那样存取这些对象:

```
extern PIP_Obj inputObj;  
if (PIP_getReaderNumFrames(&inputObj)) {  
...  
}
```

-ml0 大模式编译选项和小模式基本是相同的,但会将集合类数据(数组和结构等类型)假设为 far。使用这个选项会将所有的静态对象假设为 far 对象进行存取,而对数值类型(例如 int、char、long)的数据仍按 near 方式存取。虽然性能有所降低,但对于大多数应用程序来说是可以接受的。

## 2.3 动态创建 DSP/BIOS 对象

对于一般的 DSP 应用程序而言,大多数的 DSP/BIOS 对象应该静态地创建,因为在程序执行的整个过程中都需要使用这些对象。在配置模板中也自动地定义了许多默认的对象,静态地创建对象有以下好处:

- ❑ 改善 **DSP/BIOS** 分析工具的存取操作。线程执行图只显示那些静态创建的对象名称。另外,用户只能观察那些静态创建的对象统计信息。
- ❑ 减小代码尺寸。对于一个一般的 DSP/BIOS 模块,XXX\_create() 和 XXX\_delete() 函数包含了实现该模块所需代码量的 50%。如果用户避免了对 TSK\_create() 和



TSK\_delete()的调用,那么这些函数所隐含的代码将不会被包含在应用程序中,其他模块也是如此。所以通过静态创建对象,用户可以显著地减小应用程序的尺寸。

- ❑ 改善运行时性能。除了节省代码空间外,避免动态创建对象也将减少应用程序在执行系统初始化时所花费的时间。

静态地创建对象具有以下局限性:

- ❑ 不管是否会用到,静态对象创建之后一直存在。如果某个对象只被一个很少发生的事件使用,那么动态创建该对象的方式更可取。
- ❑ 静态创建的对象不能使用 XXX\_delete 函数删除。

用户可以使用 XXX\_create 函数创建大部分 DSP/BIOS 对象,而非全部的对象,XXX 为具体的模块名,有些对象只能静态创建。每个 XXX\_create 函数都会为对象的内部状态信息分配存储空间,并返回一个指向新建对象的句柄,XXX 模块所提供的其他函数将使用这个句柄来引用这个新对象。

大多数 XXX\_create 函数的最后一个参数是一个指向 XXX\_Attrs 结构的指针,该结构用于给新创建的对象属性进行赋值。依照惯例,若该参数为空(NULL)时,则使用默认值对新创建对象的属性进行赋值,这些默认值包含于头文件中的 XXX\_Attrs 结构体常量中,这使得用户可以先将这个常量赋予一个 XXX\_Attrs 变量,然后在调用 XXX\_create 函数之前有选择地改变其中的一些由应用程序决定的属性域。例 2-1 给出了使用 TSK\_create 函数创建一个动态对象的例子。

### 例 2-1 创建和引用动态对象

```
#include <tsk.h>

TSK_Attrs attrs;
TSK_Handle task;

attrs = TSK_ATTRS;
attrs.name = "reader";
attrs.priority = TSK_MINPRI;
task = TSK_create((Fxn)foo, &attrs);
```

XXX\_create 函数所返回的句柄实际上是被创建对象的地址, 可以作为参数提供给该模块的其他 API 函数引用该对象, 例如供给 XXX\_delete 函数来删除该对象, 如例 2-2 所示。使用 XXX\_create 函数创建的对象可以调用 XXX\_delete 函数来将其删除, 删除时会释放该对象的内部存储空间给系统作后续使用。

全局常量 XXX\_Attrs 用于保存默认值、更新属性域以及作为 XXX\_create 函数的入口参数。

### 例 2-2 删除一个动态对象

```
TSK_delete(task);
```

动态创建的 DSP/BIOS 对象可在运行时由程序进行调整。

## 2.4 建立 DSP/BIOS 程序使用的文件

图 2-2 给出了建立 DSP/BIOS 程序所需的文件。白色背景的文件由用户自己编写,灰色背景的文件是 DSP/BIOS 自动生成的。单词 program 代表用户项目或应用程序名称。根据用户平台的不同,数字 54 可以被替换为 55、62 或 64。

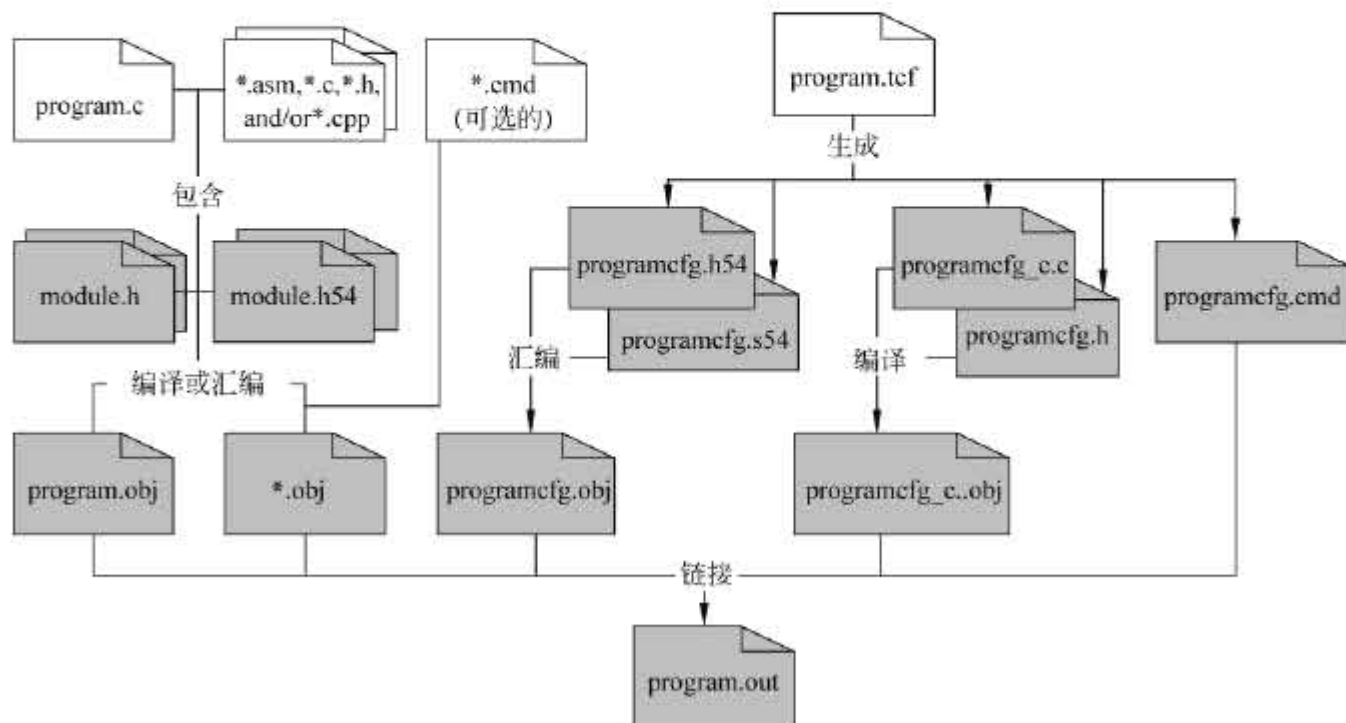


图 2-2 DSP / BIOS 应用程序中的文件

程序文件：

- ❑ **program.c**：包括 main 函数的程序源文件。用户还可以有另外的.c 源文件和.h 头文件。对于用户函数，参见 2.9 节“DSP/BIOS 调用的用户函数”。
- ❑ **\*.asm**：可选的汇编源文件。这些文件中的一个可以用包含 `_main` 的汇编函数来替代 C 或 C++ 的 main 函数。
- ❑ **module.h**：用于 C 或 C++ 程序的 DSP/BIOS API 头文件。用户源程序中需要包含 `std.h` 和任何使用到的模块的 \*.h 头文件。
- ❑ **module.h54**：用于汇编程序的 DSP/BIOS API 头文件。汇编源程序需要包含任何使用到的模块的 \*.h54 头文件。
- ❑ **program.obj**：源文件编译或汇编后生成的目标文件。
- ❑ **\*.obj**：由可选的汇编源文件生成的目标文件。
- ❑ **\*.cmd**：可选的链接命令文件，包含 DSP/BIOS 配置没有定义的附加段的信息。
- ❑ **program.out**：可执行应用程序，可以由 CCS 命令装载并在目标 DSP 上运行的可执

行文件。

静态配置文件：

当用户保存一个 DSP/BIOS 配置时，将自动生成下列文件（其中“program”是配置文件名并且根据用户平台的不同，数字 54 可以被替换为 28、55 或 64。）：

- ❑ **program.tcf**：在运行时建立配置的 Tconf 脚本，是配置的源文件。要使配置成为用户项目的一部分，必须将该文件添加到项目中。
- ❑ **programcfg.cmd**：DSP/BIOS 对象的链接命令文件，该文件定义了 DSP/BIOS 专用链接选项和对象名称，以及通用存储段（如，text、.bss、.data 等）。当添加 \*.tcf 文件到一个 CCS 项目中时，该文件会被自动添加到项目视图的“Generated Files”目录下。
- ❑ **programcfg.h**：该文件包含了 DSP/BIOS 模块头文件，并包含了对配置中创建的对象的外部变量声明。
- ❑ **programcfg.c.c**：定义 DSP/BIOS 相关对象（不同于以前的 DSP/BIOS 版本，新的版本中不再定义 CSL 对象）。
- ❑ **programcfg.s54**：用于 DSP/BIOS 设置的汇编语言源文件，会在添加 \*.tcf 文件的同时被自动添加到项目视图的“Generated Files”目录下。
- ❑ **programcfg.h54**：汇编语言头文件，被 programcfg.s54 文件包含。
- ❑ **program.cdb**：保存由实时分析工具使用的配置设置，在以前的版本中该文件为配置源文件。现在该文件通过运行 \*.tcf 文件生成，该文件被分析工具使用。
- ❑ **programcfg.obj**：由配置源文件生成的目标文件。

## 2.5 编译和链接 DSP/BIOS 程序

用户可以使用 CCS 项目或者自己的 makefile 文件来建立 DSP/BIOS 可执行应用程序。CCS 软件中包含了 GNU 构造工具 gmake.exe 以及用 gmake 构建教学示例时使用的 makefile 文件, 用户可以使用 gmake 来链接这些示例程序。

### 2.5.1 构建 CCS 项目

将一个 DSP/BIOS 配置添加到一个 CCS 项目中的步骤如下:

(1) 如果项目尚未打开, 在 CCS 中选择“Project”→“Open”命令打开项目。

(2) 选择“Project”→“Add Files to Project”。在文件类型列表框中选择配置文件(\*.tcf) (根据 DSP/BIOS 版本的不同, 后缀名可能为 \*.tcf 或 \*.cdb)。选择用户保存的配置文件并单击“Open”。当配置文件被添加到项目中时, 以下文件也会被自动添加到项目中:

■ programcfg.cmd

### ■ programcfg.s54

(3) 如果项目事先已经具有自己的链接命令文件(\*.cmd),用户可以将其移除或者两个\*.cmd文件都使用。

(4) 如果项目中包含有 vectors.asm 源文件,右键点击该文件选择“Remove”命令将其从项目中移除。因为 DSP/BIOS 配置中已经自动定义了硬件中断向量。

(5) 如果项目中包含有 rts.lib 文件,右键点击该文件选择“Remove”命令将其从项目中移除。因为该文件已经自动被包含在配置生成的链接命令文件中。

在 DSP/BIOS 应用程序中,programcfg.cmd 作为用户项目的链接命令文件,它会指示链接器使用合适的库(如 bios.a54、rtdx.lib、rts5401.lib),所以用户不需要将这些库文件添加到项目中。

CCS 软件会自动扫描用户项目中文件的关联性,为配置添加必需的 DSP/BIOS 或 RTDX 头文件到用户项目的 include 目录下。

对于大多数 DSP/BIOS 应用程序,配置生成的链接命令文件 programcfg.cmd 足以描述所有的存储器段(segments)和分配情况。所有的 DSP/BIOS 存储器段及对象都由该链接命令文件处理。对通常的存储段(sections)(如, text、.bss、.data 等)的处理也被包含在 programcfg.cmd 中。存储器段的地址、长度及其他属性可以通过配置中的 MEM 管理器来控制。

在某些情况下,应用程序需要额外的链接命令文件(app.cmd)来描述没有在配置链接命令文件(programcfg.cmd)中描述的应用程序特有的段。关于如何使用自己的 cmd 文件,参见 5.1.3 节“在自己的链接命令文件中定义存储器段”。



## 2.5.2 使用 makefile 建立 DSP/BIOS 应用程序

作为一个可选方式,用户也可以使用 makefile 文件链接自己的应用程序。

在下面这个例程里,包括 C 源文件 volume.c、汇编源文件 load.asm 和配置文件 volume.cdb。CCS 软件自带有 gmake.exe 工具, gmake 工具使用 makefile 文件来链接这些应用程序文件,用户可以在产品 CD 的 PDF 文档中查找 gmake 工具的使用说明。上面提到的 makefile 文件、源文件及配置文件都可以在产品 CD 的 tutorial 目录里的 volume2 子目录里找到。

典型的 makefile 文件如例 2-3 所示,用户可以在该 CCS 所提供的示例 makefile 的基础上作必要的修改,然后使用 gmake.exe 工具建立程序。具体的使用方法可参阅相关文档。

与 CCS 项目不同的是,makefile 文件允许多个链接命令文件。若应用程序需要多个 cmd 文件,用户只需将其他的 cmd 文件添加到例 2-3 的 makefile 文件的 CMDS 变量里。但是这些 cmd 文件必须放在配置生成的 programcfg.cmd 文件之后。

### 例 2-3 DSP/BIOS 程序的 makefile 示例

```
# 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
CC540PTS = -g
AS540PTS =
# -q quiet run
LD540PTS = -q
```

```

# Every DSP/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
OBJS          = $(PROG)cfg.obj load.obj
LIBS          =
CMDS          = $(PROG)cfg.cmd

# Targets:
all:: $(PROG).out

$(PROG).out: $(OBJS) $(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 DSP/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
```

## 2.6 DSP/BIOS 程序中的运行支持库

配置生成的链接命令文件中包含一些命令,可以自动搜索必要的库文件,包括 DSP/BIOS 库、RTDX 库以及运行支持库。运行支持库由 `rts.src` 建立,该文件包含了运行时支持库函数的源代码。这些函数是一些标准 ANSI 函数(例如存储器分配函数,字符串转换和搜索函数)。一些在 `rts.src` 定义的存储器管理函数在 DSP/BIOS 库中也做了定义,这些函数有 `malloc`、`free`、`memalign`、`calloc` 和 `realloc`。在不同的库中这些函数实现的方式是不同的,例如,在 DSP/BIOS 库里这些函数是通过 MEM 模块调用 DSP/BIOS API 函数 `MEM_alloc` 和 `MEM_free` 实现的。因为 DSP/BIOS 库中提供了一些与运行支持库相同的函数,所以 DSP/BIOS 的链接命令文件中包含了一个特殊版本的运行支持库 `rtsbios`,该库不包括表 2-2 中的源文件。

表 2-2 rtsbios 中不包含的文件

C54x 平台	C55x 平台	C6000 平台
memory.c	memory.c	memory.c
autoinit.c	boot.c	sysmem.c
boot.c		autoinit.c
		boot.c

在许多 DSP/BIOS 项目中都需要设置-x 链接选项,以强迫链接器重新读取库文件。例如,在 malloc 函数没有从 DSP/BIOS 库中链接进来的情况下,如果 printf 语句中引用了 malloc 函数,这时-x 链接选项就会强迫链接器重新搜索 DSP/BIOS 库以解决这一引用。

运行支持库是通过断点的方法实现 printf 函数。如果程序频繁调用 printf 函数,printf 函数则有可能影响 RTDX 的功能,也就会影响实时分析工具如消息日志和统计视图使其不能实时更新。这是因为 printf 断点的处理优先级高于 RTDX,所以建议用户在 DSP/BIOS 应用程序中尽可能使用 LOG\_printf 代替 printf 函数。

注意: 建议用户在 DSP/BIOS 应用程序中使用 DSP/BIOS 库的 malloc、free、memalign、calloc 和 realloc 函数,如果用户应用程序中没有直接引用这些函数,但是却调用了其他运行支持库函数,那些函数可能会间接引用这些 malloc 函数,因此需要在链接器选项中添加“-u symbol”(例如添加-u \_malloc)。-u 连接选项可以将一个符号(如

malloc)作为未决符号引入链接符号表。这使得链接器使用 DSP/BIOS 库而不是运行支持库来解决这一符号引用。关于这一点用户可以从 map 文件中验证。

## 2.7 DSP/BIOS 启动序列

当一个 DSP/BIOS 应用程序启动时, boot.s54 (C54x 平台), 或 autoint.c 和 boot.smn (C55x 平台和 C6000 平台) 文件中的代码决定了其启动序列, 这些文件的已编译版本由 bios.amn 和 biosi.amn 库提供, 并且其源代码可以在产品附送的光盘中找到 (*nm* 代表与各平台对应的数字)。这些引导文件源代码给出 DSP/BIOS 启动顺序 (顺序如下说明), 且用户没有必要修改该顺序。

(1) 初始化 **DSP**: DSP/BIOS 程序从 C 或 C++ 环境入口点 `c_int00` 开始运行。复位中断向量被设置为复位后跳转到 `c_int00`。



对于 C54x 平台来说, `c_int00` 开始时, 系统堆栈指针 (SP) 被初始化指向 `stack` 段尾部, 状态寄存器如 `st0`、`st1` 也被初始化。



对于 C55x 平台来说, `c_int00` 开始时, 用户数据堆栈指针 (XSP) 和系统堆栈指针 (XSSP) 被初始化分别指向各自堆栈的末尾, 且 XSP 指向一奇地址边界。



对于 C6000 平台来说, `c_int00` 开始时, 系统堆栈指针 (B15) 和全局页指针 (B14) 被初始化分别指向 `stack` 段的末尾和 `bss` 段的开始。控制寄存器如 `AMR`、`IER`、`CSR` 也同样被初始化。

(2) 用 `.cinit` 段中的记录来初始化 `.bss` 段: 一旦堆栈建立好后, 初始化程序用 `.cinit` 段中的记录初始化全局变量。

(3) 调用 `BIOS_init` 初始化程序中用到的 `DSP/BIOS` 模块: `BIOS_init` 完成基本的模块初始化操作, 为应用程序用到的 `DSP/BIOS` 模块调用 `MOD_init` 宏进行初始化。`BIOS_init` 由配置生成并位于 `programcfg.snn` 文件中。

- `HWI_init`: 设置 `ISTP` 和中断选择寄存器, 设置 C6000 平台的 `IER` 寄存器的 `NMIE` 位, 清除所有平台的 `IFR` 寄存器内容。参考相应平台 TMS320 DSP/BIOS API 函数参考手册中的 `HWI` 模块以查看更详细的说明。



注意: 在配置中断时, `DSP/BIOS` 将中断服务程序插入到中断服务表的相应位置。但是 `DSP/BIOS` 不会在 `IER` 寄存器中使能相应的中断位, 所以用户必须自己在启动序列

或程序执行过程中任何合适的时候使能中断。

- **HST\_init**: 初始化主机 I/O 通道接口,该程序的详细内容取决于特定的主机与目标 DSP 的连接方式。例如在 C6000 平台,如果使用 RTDX,HST\_init 会在 IER 中使能为 RTDX 保留的硬件中断位。
- **IDL\_init**: 进行空闲循环的指令计数。如果在空闲函数管理器里选中了“Auto calculate idle loop instruction count”属性框,IDL\_init 会在启动序列中对空闲循环的指令进行计数。空闲循环指令计数值用于校准 CPU 负荷图里实时显示的 CPU 负载值(参见 3.4.2 节)。

(4) 处理 **.pinit** 表: .pinit 表包含了指向初始化函数的指针。对于 C++ 程序,全局对象的类构造函数是在 .pinit 表的处理过程中执行的。

(5) 调用用户应用程序的 **main** 函数: 在所有 DSP/BIOS 模块初始化过程完成之后,用户 main 函数才会被调用。该函数可以由汇编语言、C、C++ 语言或者多种语言混合编写。由于 C 编译器在其函数名上加了一个下划线作为前缀,所以 C、C++ 及汇编程序都可调用该函数。

由于此时硬件中断和软件中断都没有被使能,用户可以在 main 函数中调用自己的中断初始化函数来使能单个的中断屏蔽位,但不要在这里调用 HWI\_enable 来使能全局中断。



(6) 调用 **BIOS\_start** 启动 **DSP/BIOS**: 同 BIOS\_init 函数一样, BIOS\_start 函数也是由配置生成并包含在 programcfg. smn 文件中, 在用户 main 函数返回后调用 BIOS\_start。BIOS\_start 负责使能 DSP/BIOS 模块并为每一个用到的模块调用 MOD\_startup 宏使其开始工作(MOD 为模块名)。如果 TSK 管理器在配置中被使能, BIOS\_start 将不会返回。MOD\_startup 举例如下。

- CLK\_startup: 设置 PRD 寄存器, 根据在 CLK 管理器中选择的定时器, 在 IER (C6000 平台) 或 IMR (C54x 平台) 使能相应的定时器中断, 并最终启动定时器。(该宏只会在 CLK 管理器被使能的情况下才会被展开执行。)
- PIP\_startup: 为每个建立的管道对象调用 notifyWriter 函数。
- SWI\_startup: 使能软件中断。
- HWI\_startup: 通过设置 CSR 寄存器中的 GIE 位 (C6000 平台) 或清零 ST1 寄存器中的 INTM 位 (C5400 平台) 来使能硬件中断。
- TSK\_startup: 使能任务调度器并启动已就绪的最高优先级的任务。如果应用程序当前没有就绪任务, 则执行会调用 IDLE\_loop 的 TSK\_idle 任务。一旦 TSK\_startup 被调用, 应用程序开始执行并且不会从 TSK\_startup 或者 BIOS\_start 返回。TSK\_startup 只在 TSK 管理器被使能的情况下执行。

(7) 执行空循环: 用户应用程序会在两种情况下进入空闲循环。第一种情况是任务管理器被使能, 任务调度器在运行 TSK\_idle 时会调用 IDL\_loop。第二种情况是任务管理器被禁用, 则当 BIOS\_start 返回时紧接着调用 IDL\_loop。通过调用 IDL\_loop, 引导程序落入

DSP/BIOS 空闲循环,此时发生的硬件中断和软件中断都可以抢占空闲循环的执行。空闲循环控制着和主机的通信,此时主机和目标器之间的数据传输就可以开始了。

## 2.7.1 C5500 平台启动序列



C5500 平台的体系结构允许软件对中断向量表(总长为 256 字节)的首地址进行编程,通过设置寄存器 IVPD 和 IVPH 即可实现。默认情况下,硬件复位时 DSP 会装载 0xFFFF 到这两个寄存器中,复位向量的访问位置为程序地址 0xFF FF00。若要将向量表移动到其他位置,必须在硬件复位后用所希望的地址值设置 IVPD 和 IVPH 寄存器,然后进行一次软件复位,这时 IVPD 和 IVPH 新的值才会起作用。

HWL\_init 宏实现将配置好的向量表地址装载到 IVPD 和 IVPH 寄存器中,但紧接着必须进行一次软件复位来让设置真正起作用。

C5500 平台还支持 3 种堆栈模式(如表 2-3 所示)。为了配置处理器到非默认的堆栈模式,用户需要设置复位向量位置的第 28 和 29 位,然后进行一次软件复位。这可以通过 CCS 调试工具完成。

表 2-3 C5500 平台堆栈模式

堆栈模式	说 明	复位向量设置
2x16 快返回	SP/SSP 相互独立， RETA/CFCT 用于实现快速返回	XX00: XXXX: <24 位 向量地址>
2x16 慢返回	SP/SSP 相互独立， RETA/CFCT 没有使用	XX01: XXXX: <24 位 向量地址>
1x32 慢返回（默认）	SP/SSP 相互同步， RETA/CFCT 没有使用	XX02: XXXX: <24 位 向量地址>

另外,DSP/BIOS 配置文件应该设置 HWI 管理器的 Stack Mode 属性,使其与应用程序使用的模式相一致。详细信息请参考《TMS320C5000 DSP/BIOS API 用户参考手册》。

## 2.8 DSP/BIOS 中使用 C++ 语言

DSP/BIOS 应用程序可以用 C++ 语言编写。在 DSP/BIOS 中使用 C++ 有一些问题需要注意,包括存储器管理、名称改编以及类函数尤其是类构造和类析构函数的调用方法。理解这些有关 C++ 和 DSP/BIOS 的问题有助于更顺利地进行 C++ 应用程序的开发。

## 2.8.1 存储器管理

C++ 操作符 `new` 和 `delete` 分别实现内存的动态分配和释放,在 DSP/BIOS 应用程序中这两个操作符由 DSP/BIOS 存储器管理函数 `MEM_alloc` 和 `MEM_free` 实现。然而,调用这些函数的线程必须预先获得一个存储器资源锁。如果所需的存储器资源锁被别的线程拥有,则会产生阻塞。因此 `new` 和 `delete` 只能由 TSK 对象使用。

`new` 和 `delete` 函数由运行支持库定义,而并非由 DSP/BIOS 库定义。因为 DSP/BIOS 程序链接时首先搜索 DSP/BIOS 库,而这些函数符号是在 `rtsbios` 库(运行支持库)里进行定义的,所以某些应用程序可能出现未定义符号的链接错误。可以使用 `-x` 链接器选项避免这种链接错误,这个选项强制对库文件重新搜索以解决未定义符号引用的链接错误,详细信息请参考 2.6 节。

## 2.8.2 名称改编

在 C++ 中,为了允许操作符重载和函数重载,C++ 编译器往往按照某种规则改写每一个入口点的符号名,以便允许同一个名字有多个用法(具有不同的参数类型或者是不同的作用域),而不会打破现有的基于 C 的链接器。这种技术通常被称为名称改编(name mangling)或者名称修饰(name decoration)。

通过对函数名编码生成其链接名, C++ 编译器能实现函数重载, 操作符重载以及类型安全链接。将函数名编码生成其链接名(linkname)的过程被称为名字改编(name mangling), 而名字改编可能会潜在地影响 DSP/BIOS 应用程序, 这是因为用户一般是使用配置里定义的函数名来引用那些在 C++ 源文件里定义的函数。为了避免名字改编的影响并使得源文件里的函数可以在配置里识别, 用户需要将自己的函数像例 2-4 所示的那样在一个 extern C 块中进行声明。

**例 2-4** 在 extern C 块中声明函数

```
extern "C" {  
    Void function1();  
    Int function2();  
}
```

这样用户就能够在配置里引用这些函数(即在函数名前面加一个下划线前缀, 就和使用其他的 C 函数一样)。例如用户希望每次 SWI 软件中断触发之后运行 function1() 函数, 那么用户只需要在配置工具里该 SWI 对象的函数属性框里填入 function1 就可以。

在 extern C 块里声明的函数不会受到名字改编的影响。由于函数重载是通过名字改编来实现的, 因此对配置文件里调用的函数的重载具有一些限制。extern C 块中只能出现重载函数的一种版本。例 2-5 所示的代码将导致错误发生。

**例 2-5** 函数重载限制

```
extern "C" {
```

```
Int addNums(Int x, Int y);  
Int addNums(Int x, Int y, Int z); // error, only one version  
                                   //of addNums is allowed  
}
```

虽然用户在自己的 DSP/BIOS C++ 应用程序里能使用函数名重载,但配置文件只能调用重载函数的一种版本。

默认参数是 C++ 的一个特点,但是在 DSP/BIOS 配置里调用的函数并不支持这个特性。C++ 允许用户在函数声明里给出参数的默认值,然而在配置里调用的函数必须确定地提供具体的参数值,如果没有给出具体值,则表明实际参数未定义。

### 2.8.3 在配置中调用类的成员函数

我们经常想在配置文件里引用 C++ 的类对象的成员函数,直接从配置文件里调用这些成员函数是不可能的,但是可以通过调用封装函数(wrapper functions)来实现。通过将类的成员函数作为参数编写一个封装函数,从配置文件里调用这个封装函数就可以实现对这个类成员函数的调用。

例 2-6 给出了一

### 例 2-6 类的方法的封装函数

```
Void wrapper (SampleClass myObject)
{
    myObject->method();
}
```

类的成员函数所需的任何其他参数都可以传递给封装函数。

## 2.8.4 类的构造函数和析构函数

在 C++ 类对象初始化建立的任何时候都要调用执行类的构造函数,同样在删除 C++ 类对象的时候都会调用类的析构函数。因此在编写构造函数和析构函数的时候,用户必须考虑这些函数的调用时机并且相应地进行调整。在调用这两个函数时考虑到是在哪种线程环境下运行是非常重要的。

哪些 DSP/BIOS API 函数能在哪种类型的线程中调用是有许多原则和依据的。例如 MEM\_alloc 和 MEM\_calloc 这样的存储器分配函数不能在软件中断 SWI 线程环境中调用。因此如果由一个软件中断来初始化某个类,那么该类的构造函数要避免执行存储器分配操作。同样的,用户也需要时刻注意类的析构函数运行的时机。不仅当对象被显式删除的时候析构函数会被执行,而且当局部对象的生命期结束的时候析构函数也会被自动调用。因此用户必须清楚调用执行类析构函数的线程类型并且在析构函数内只能使用那些适合此线程类型的 API 函数。详细信息请参考相应平台的《TMS320 DSP/BIOS API 参考手册》。

## 2.9 DSP/BIOS 调用的用户函数

由 DSP/BIOS 对象 (IDL、TSK、SWI、PIP、PRD 和 CLK) 调用的用户函数必须符合一定的约定, 以保证正确使用寄存器及在函数调用过程中对寄存器值进行现场保护。



在 C6x 和 C55x 平台上, 所有被 DSP/BIOS 对象调用的用户函数必须遵守相应的 C 编译器寄存器约定。C 语言及汇编语言编写的函数都要求遵守相应的约定。



编译器通过假设 C 函数名以一个下划线开头来区分 C 和汇编函数。在 C54x 平台中, 因为 C 函数和汇编函数遵照两种不同的规则, 这个区别方法显得尤为重要。以一个下划线开头的函数 (可能是 C 函数也可能是以一个下划线开头的汇编函数) 必须遵照 C 编译器的寄存器约定, 而不以一个下划线开头的汇编函数则必须遵守以下规则:

- ☐ 第一个参数通过 AR2 寄存器传递。
- ☐ 第二个参数通过 A 寄存器传递。
- ☐ 返回值通过 A 寄存器传递。



注意：以上规则不适用于由 TSK 对象调用的用户函数。所有被 TSK 对象调用的用户函数（包括 C 函数和汇编函数）都应遵守 C 编译器的寄存器约定。



在 C54x 平台上, 当一个 C 函数(或者以一个下划线开头的汇编函数)执行时, CPL(编译模式)位需要被设置为 1。当一个汇编函数(不以一个下划线开头)开始执行时, CPL 位需要被清 0, 然后在返回前恢复为 1。

有关 C 寄存器约定的更多信息, 请查阅相应平台的优化编译器用户手册 (Optimizing Compiler User's Guide)。

## 2.10 Main 函数中调用 DSP/BIOS API 函数

在 DSP/BIOS 应用程序中, main 函数用于实现一些用户初始化操作, 如配置外围设备、使能单独的硬件中断等。需要特别注意的是, main 函数并不属于任何一种 DSP/BIOS 线程

类型(HWI、SWI、TSK 或 IDL),并且当应用程序执行到 main 函数时,并非所有的 DSP/BIOS 初始化操作都已经完成,这是因为 DSP/BIOS 初始化操作发生在两个阶段:在 main 函数之前运行的 BIOS\_init 中和在 main 函数返回后运行的 BIOS\_start 中。

某些 DSP/BIOS API 函数不能在 main 函数中调用,这是因为此时 BIOS\_start 初始化操作还没有执行。而 BIOS\_start 负责使能全局中断、配置和启动定时器、打开线程调度使 DSP/BIOS 线程开始执行。因此那些假设硬件中断和定时器已被使能的 API,或那些能引起阻塞的 API 都不适合在 main 函数中调用。例如,CLK\_gettime 和 CLK\_gettime 不能在 main 函数中进行调用,因为定时器还没有运行。HWI\_enable 和 HWI\_disable 函数也不能被调用,因为硬件中断还没有被全局使能。有可能引起阻塞的 SEM\_pend 或 MBX\_pend 也不能被调用,因为线程调度器没有被初始化。同样的 TSK\_disable、TSK\_enable、SWI\_disable 和 SWI\_enable 等调度函数也不能在 main 函数里进行调用。

在 main 函数之前运行的 BIOS\_init,主要负责 MEM 模块的初始化,所以在 main 函数中就可以调用实现动态存储器分配的函数。不仅 MEM 模块的 API(如 MEM\_alloc、MEM\_free 等),而且用于动态创建和删除 DSP/BIOS 对象的 API 函数,如 TSK\_create、TSK\_delete 等,都可以在 main 函数中调用。

虽然可能产生阻塞的函数不允许在 main 函数中调用,但是使 DSP/BIOS 线程就绪的调度函数是允许在 main 函数中调用的,例如 SEM\_post 或 SWI\_post。如果 main 函数中调用了这样的函数,就绪的线程就会在 BIOS\_start 执行完毕后直接被调度执行。

更多的关于每一个特定 DSP/BIOS 函数调用的信息请查阅相应平台的 TMS320 DSP/BIOS API 参考手册。调用限制与调用上下文部分指示了该 API 能否被 main 函数调用。



# 第 3 章 监 测

DSP/BIOS 提供了显式和隐式两种方式进行实时程序分析,这些机制对应用程序的实时性能具有最小限度的影响。

## 3.1 实 时 分 析

实时分析是指在一个系统实时运行的过程中对获取的数据进行分析,其目的是帮助用户更容易地判断系统是否是在设计约束下运行的,是否满足性能指标,以及是否有进一步开发的余地。

注意:一个新发布的 DSP 器件或开发板有时候不支持 RTDX。在不支持 RTDX 的平台上,监测数据只有在停止模式下才可以进行更新。也就是说当目标程序运行时,不能和主机 PC 进行数据通信,只有停止目标程序或者当程序执行到断点时,分析数据才被传输到主机并在 CCS 上进行显示。



### 3.1.1 实时调试与循环调试的对比

顺序执行的软件(sequential software)的传统调试方法是一直运行程序直到出现错误,然后停止程序,检查程序状态,插入断点,重新执行程序来收集信息。这种循环调试方法对于非实时的顺序执行的软件是有效的,但很少能在实时系统中起作用,因为实时系统具有连续不断运行、非确定性执行和时序约束严格的特点。

DSP/BIOS 的监测类 API 函数和 DSP/BIOS 分析工具就是为了弥补循环调试工具的不足而设计的,使得用户可以对运行中的实时系统进行监测。得到的实时监测数据可以帮助用户观察实时系统的运行,从而有效地调试和调节系统性能。

### 3.1.2 软件监测与硬件监测的对比

软件监测是指在目标程序代码中编写部分监测代码实现实时系统分析。这些代码在系统运行的时候执行,将感兴趣事件的数据保存在目标系统的存储器中。因此,软件监测不仅使用了目标系统的计算时间,而且还占用其存储空间。

软件监测的优点是灵活且不需要额外的硬件,但是由于监测代码是目标应用程序的一

部分,这样系统的性能和程序行为都可能受到影响。若没有硬件监测器,用户就要面临程序混乱和记录信息是否充分之间的权衡问题。有限的软件监测只能提供不充分的细节,而过多的软件监测又会影响被测系统。

DSP/BIOS 为用户提供了各种各样的机制来精确地控制程序混乱和信息收集之间的平衡。另外 DSP/BIOS 监测操作只需要固定的、很短的执行时间。由于时间开销固定,这样就可以估算出监测对系统的影响了。

## 3.2 监测性能

DSP/BIOS 采用了多种技术来尽量减小实时监测对应用程序性能的影响,在典型应用程序中,所有隐式 DSP/BIOS 监测被使能时,CPU 负荷的增加不到 1%:

- ❑ 目标 DSP 和主机之间的监测通信在后台线程 IDL 中执行,该线程优先级最低,因此监测数据的通信不会影响应用程序的实时行为。
- ❑ 用户能够从主机端控制主机轮询目标 DSP 的频率,如果用户想要消除与目标 DSP 之间所有不必要的交互操作,可以停止主机同目标 DSP 之间的交互操作。
- ❑ 除非追踪(tracing)功能被使能,目标 DSP 不会存储执行图或隐式统计信息。用户还能通过使用 TRC 模块和某一个预留的追踪掩模(TRC\_USER0、TRC\_USER1)来使能或禁止应用程序的显式监测。

- ❑ 日志和统计数据总是在主机上进行格式化的, STS 对象的平均值和 CPU 负荷都是在主机上进行计算的, 显示执行图所需的数据也是在主机上进行计算的。
- ❑ LOG、STS 和 TRC 模块的执行非常快, 且执行时间固定:

C54x

- LOG\_printf 和 LOG\_event: 大约 30 个指令周期
- STS\_add: 大约 30 个指令周期
- STS\_delta: 大约 40 个指令周期
- TRC\_enable 和 TRC\_disable: 大约 4 个指令周期

C55x

- LOG\_printf 和 LOG\_event: 大约 25 个指令周期
- STS\_add: 大约 10 个指令周期
- STS\_delta: 大约 15 个指令周期
- TRC\_enable 和 TRC\_disable: 大约 4 个指令周期

C6000

- LOG\_printf 和 LOG\_event: 大约 32 个指令周期





- STS\_add: 大约 18 个指令周期
- STS\_delta: 大约 21 个指令周期
- TRC\_enable 和 TRC\_disable: 大约 6 个指令周期
- 每个 STS 对象仅占用 8 个字(对于 C5000 平台)或 4 个字(对于 C6000 平台)的数据存储空间。
- 统计数据在目标 DSP 上使用 32 位的变量累加,在主机上则用 64 位的变量累加。当主机每次轮询完目标 DSP 的实时统计数据后,会重新初始化目标 DSP 的统计变量。这使得目标 DSP 上所需的存储空间最小化,同时又能为长时间的测试保留统计信息。
- 用户可以指定 LOG 对象的缓冲区大小。这一缓冲区的大小影响到应用程序数据空间的大小和更新日志数据所耗费的时间。
- 对硬件中断的监测状态默认是禁止的,这样不会影响到应用程序的性能。当硬件中断监测被使能时,对于每个被监测中断的每次中断,更新统计对象数据的操作会耗费 20 到 30 个指令周期。




### 3.2.1 监测内核与非监测内核的对比



通过更改应用程序的全局属性,用户可以禁止掉 DSP/BIOS 内核中的监测功能。在配置工具中,全局设置模块有一个“Enable Real Time Analysis”属性。将其禁止后,用户可以得到最佳的代码长度和执行速度,这是通过链接不支持隐式监测功能的 DSP/BIOS 库实现的。然而这样也去掉了对 DSP/BIOS 分析工具的支持,以及对显式监测(如 LOG、TRC 和 STS 模块的 API 函数)的支持。

表 3-1 给出了使用监测内核和使用非监测内核时应用程序代码的大小。通过对比该表,用户可以预想到使用监测内核时代码的增加量。该表针对 CCS 软件自带的两个 DSP/BIOS 项目给出。第一个应用程序例子 Slice 包含了 TSK、SEM 和 PRD 模块,第二个应用程序例子 Echo 包含了 PRD 和 SWI 模块。

表 3-1 监测内核引入代码大小增加示例

说明 (所有的代码大小都以 MADUs 为单位)	C54x 平台 	C55x 平台 	C6000 平台 
a. 例子: Slice			
使用非监测内核	12 500	32 000	78 900
使用监测内核	14 350	33 800	86 600
代码增量	1 850	1 800	7 700
b. 例子: Echo			
使用非监测内核	11 600	41 200	68 800
使用监测内核	13 000	42 800	76 200
代码增量	1 400	1 600	7 400

### 3.3 监测 APIs

有效地监测需要两种操作：数据采集以及响应程序事件后对数据采集的控制。DSP/BIOS 提供以下三种 API 模块来实现数据采集。

- ❑ **LOG(Event Log Manager, 事件日志管理器)**：日志对象可以实时捕获有关事件的信息。系统事件被记录在系统日志中。用户也可以配置其他的日志。用户应用程序可以向任意一个日志中添加消息。
- ❑ **STS(Statistics Object Manager, 统计对象管理器)**：统计对象可以实时捕获任意变量的计数、最大值和总和等统计数据。对 SWI、PRD、HWI、PIP 和 TSK 对象的统计信息可自动捕获。用户程序也可以创建更多的统计对象来获取其他统计信息。
- ❑ **HST(Host Channel Manager, 主机通道管理器)**：主机通道对象 HST 允许程序将原始数据发送到主机进行分析。详见第 6 章“I/O 概述和管道”。

LOG 和 STS 提供了有效的方法来捕获一个高频率实时事件序列的子集或一个快速变化的数据值的统计摘要。这些事件出现的频率或数值变化的频率有可能过高,以至于不可能将整个事件序列传输到主机(由于带宽限制),或者传输该序列到主机的开销影响到了程序的执行。所以 DSP/BIOS 提供了 TRC(Trace Manager, 追踪管理器)模块来控制由其他监测模块提供的数据采集的机制。TRC 模块控制着哪些事件和统计信息该由目标程序实时采集,哪些该通过 DSP/BIOS 分析工具交互式的采集。

对数据采集的控制是很重要的,这就允许用户可以限制监测对程序行为的影响,保证 LOG 和 STS 对象得到必需的信息,以及在运行时启动或停止对事件或数值的记录。

### 3.3.1 显式监测与隐式监测的对比

监测 API 函数的设计使得应用程序可以显式地调用它们: LOG 模块函数允许用户显式地写入消息到任何日志中,STS 模块函数允许用户存储数据变量或系统性能参数的统计信息,TRC 模块则允许用户程序在响应一个程序事件时使能或禁止日志追踪和统计追踪。

LOG 和 STS 的 API 函数也可以被 DSP/BIOS 内部使用来收集程序执行的信息,这些内部调用提供了对隐式监测的支持。因此即使应用程序没有显式地调用任何 DSP/BIOS 监测 API 函数,也能使用实时分析工具来监测和分析。例如软件中断的执行就被记录在一个名为 LOG\_system 的 LOG 对象中。

此外,软件中断最长的“就绪到完成”的时间以及 CPU 总负荷,也都是通过 STS 对象收集的。系统时钟刻度也会在执行图中显示。关于隐式监测可以获取哪些信息,详见 3.3.4.2 节“对隐式监测的控制”。

### 3.3.2 事件日志管理器(LOG 模块)

该模块管理 LOG 对象,这些对象负责在目标程序执行时实时地捕获事件,用户可以使用执行图(Execution Graph)或者使用自定义的日志来观察事件信息。

用户自定义的日志对象中包含了用户程序使用 LOG\_event 和 LOG\_printf 函数记录到其中的信息,用户可以在 Message Log 窗口中实时观察这些日志中的消息,如图 3-1 所示。选择菜单“DSP/BIOS”→“Message Log”就可以打开 Message Log 窗口。

执行图实际上反映的就是系统日志(LOG\_system 对象)中的信息,它以图形的方式显示应用程序各个组成部分的活动情况。

日志可以是固定式的或循环式的,二者的区别对于使用 TRC 模块使能或禁止日志记录的应用程序是很重要的:

- ❑ 固定的(**Fixed**):从接收的第一个消息开始记录,直到缓冲区满则停止接收消息。因此固定日志对象中保存的是从日志被使能后最先发生的事件集。
- ❑ 循环的(**Circular**):当其缓冲区满时,日志会自动覆盖最早的消息。因此循环日志对象中保存的是最后发生的事件集。

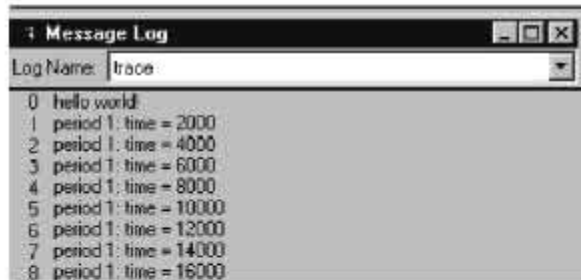


图 3-1 Message Log 窗口

LOG 对象的消息缓冲区长度和位置可以由用户静态配置,其长度以字(word)为单位。每个消息结构占用日志缓冲区中 4 个字的存储空间,第 1 个字中保存消息序列号,剩余的 3 个字包含事件的编码和数据值,它们由用户以参数的形式提供给 LOG 模块的 API 函数,例如负责添加新事件到 LOG 对象中的 LOG\_event 函数。

主机获取日志信息时,目标 DSP 中 LOG 缓冲区的内容会被读取并复制到主机上一个更大的缓冲区中,同时目标 DSP 中被读取过的记录会被标记为空,其过程如图 3-2 所示。

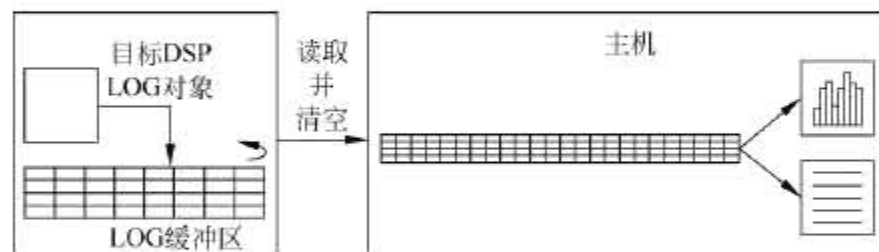


图 3-2 LOG 缓冲区

LOG\_printf 使用消息结构体中的第 4 个字来存放格式化字符串(例如 %d、%c)的地址或偏移量,主机则根据得到的格式化字符串对消息结构体中剩余的两个字进行格式化以显示数据。由于实际的 printf 操作以及对应的代码都由主机处理,因此减少了目标 DSP 上的代码大小以及耗时。

LOG\_event 和 LOG\_printf 在执行时,中断都是被禁止掉的。这使得硬件中断和其他不



同优先级的线程可以向同一个日志写入消息,而不必担心访问同步问题。

使用如图 3-3 所示的 RTA 控制面板属性对话框,用户可以控制主机轮询目标 DSP 日志信息的频率。选择菜单“DSP/BIOS”→“RTA Control Panel”,然后在打开的 RTA 控制面板中单击右键并选择“Property Page”,即可打开 RTA 控制面板属性对话框设置刷新频率。如果用户将刷新频率设为 0,主机将不会轮询目标 DSP 日志信息,除非用户在日志窗口中单击右键并在弹出菜单中选择“Refresh Window”(即手动刷新)。用户也可以在日志窗口中使用右键快捷菜单来暂停和恢复对日志信息的轮询。

消息日志窗口中的日志消息都被编号(编号位于日志追踪窗口第 1 列)来指示事件出现的顺序,这些编号是从 0 开始的递增序列。如果用户的日志缓冲区永远不会满,可以使用一个较小日志缓冲区。如果循环日志的缓冲区不够大或者轮询频率不够高,可能会丢掉某些日志信息,这种情况下看到的编号会有间隔。

DSP/BIOS 在线帮助中描述了 LOG 对象及其参数。有关 LOG 模块的 API 调用,请查阅相应平台的 TMS320 DSP/BIOS API 参考手册中的 LOG 模块部分。

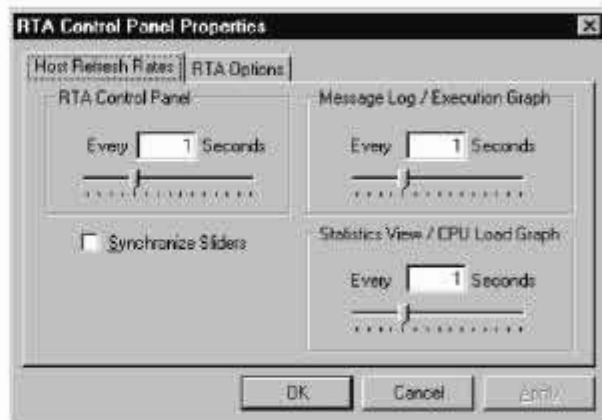


图 3-3 RTA 控制面板属性对话框

### 3.3.3 统计对象管理器(STS 模块)

STS 模块管理统计对象,这些对象负责存储应用程序执行过程中的重要统计数据。

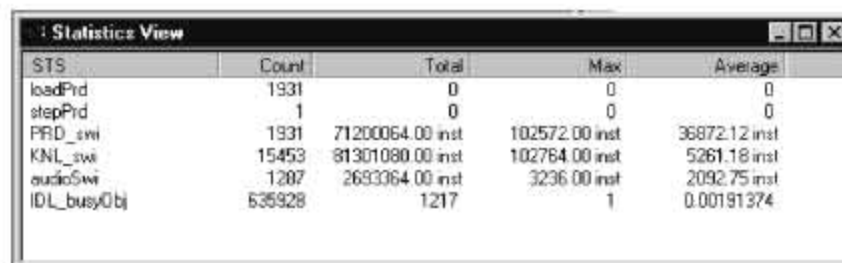
统计对象的创建是在图形化配置工具里静态完成,每个 STS 对象可以收集任意一个长度不大于 32 位(二进制)的数据变量的如下统计信息:

- ❑ **Count:** 目标 DSP 中一个应用程序提供的数据变量被观测的次数。
- ❑ **Total:** 目标 DSP 中该数据变量所有观测值的算术和。
- ❑ **Maximum:** 目标 DSP 中该数据变量的最大观测数值。
- ❑ **Average:** 在主机端,由统计分析工具根据 Count 和 Total 计算得到的观测序列的平均值。

调用 STS\_add 函数可以对监测数据变量的统计对象进行更新。例如,用户可能需要测试一个软件中断处理算法的得失,或测试一个封闭循环控制算法的预期错误和实际错误。

DSP/BIOS 统计对象在执行过程中追踪计算不同例程的 CPU 绝对使用率方面也是很有用的。通过在程序中合适的地方成对地使用 STS\_set 和 STS\_delta 调用,用户可以得到程序某个部分的实时性能统计数据。

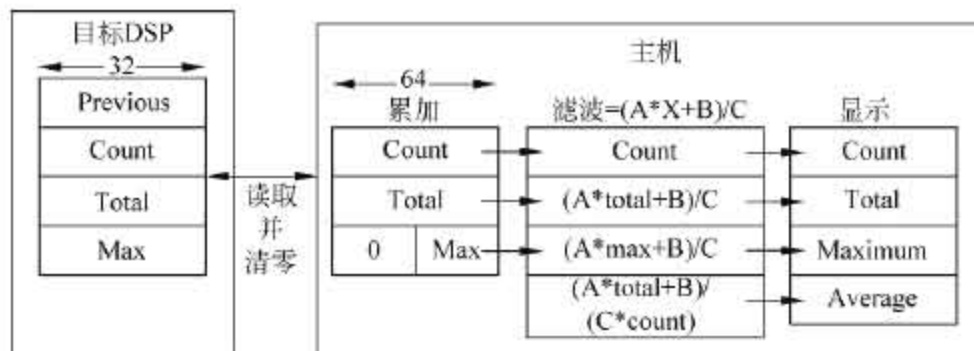
用户可在统计视图窗口中实时地观察这些统计信息,选择菜单“DSP/BIOS”→“Statistics View”即可打开这个统计视图窗口,如图 3-4 所示。



STS	Count	Total	Max	Average
loadPrd	1931	0	0	0
stepPrd	1	0	0	0
PRD_swi	1931	7120064.00 inst	102572.00 inst	36872.12 inst
KNL_swi	15453	81301080.00 inst	102764.00 inst	5261.18 inst
audioSwi	1287	2683364.00 inst	3236.00 inst	2092.75 inst
IDL_busyObj	636928	1217	1	0.00191374

图 3-4 统计视图窗口

虽然对算术和的统计在目标 DSP 上是使用 32 位变量累加的,但在主机端是使用 64 位变量累加的。当主机对目标 DSP 完成一次实时统计数据的查询后,会将目标 DSP 上的累加变量初始化为零。这种方式能够最小化目标 DSP 的存储空间需求,并能实现长时间的统计。主机端的统计视图窗口还可以在输出显示之前可选地对数据进行算术滤波,如图 3-5 所示。



统计变量处理



通过将目标 DSP 端统计变量清零,在主机端显示的数值可以非常大,并且不存在因目标端数值溢出而产生数据丢失的危险。如果对 STS 数据的轮询被禁止掉或频率非常低,则会存在目标端 STS 数据溢出导致信息错误的可能性。

主机对目标端统计数值的清零是自动完成的,而主机端 64 位数值的清零需要通过在 STS 数据窗口中单击右键然后在弹出菜单中选择“Clear”来实现。

主机对目标端统计数值的读取和清零操作都是在中断被禁止的情况下进行的,这使得任何线程都能可靠地对任意 STS 对象进行更新。例如,一个 HWI 函数可以随意调用某个 STS 对象的 STS\_add 函数,而不会丢失任何统计数据。

监测处理对于目标程序的影响是很小的,一个 STS\_add 调用在 C5000 平台大约只需要 20 个指令周期,在 C6000 平台上大约只需要 18 个指令周期。而且,一个统计对象在目标 DSP 上只占用 8 个(C5000 平台)或 4 个(C6000 平台)字的存储空间。数据滤波、格式化显示和平均值计算这些操作都在主机上进行。

用户也可以通过 RTA 控制面板属性对话框设置主机轮询统计信息的频率。如果轮询频率设置为 0,则直到在统计视图窗口的右键快捷菜单中选择“Refresh Window”,主机才会查询目标 DSP 上统计对象里保存的统计信息。

### 3.3.3.1 对变化数值的统计

STS 对象可以用来收集一个 32 位数据值在一段时间内的统计信息。例如,设  $P_i$  为某算法探测出的第  $i$  帧音频数据的音调(pitch),一个 STS 对象可以用来存储序列  $\{P_i\}$  的统计信息。下面的代码片段说明了该序列的音调值是如何被追踪的:

```
pitch = 'do pitch detection'
STS_add(&stsObj, pitch);
```

统计视图窗口会显示出该统计序列中的数值个数、最大值、所有数值的算术和及其平均值。

### 3.3.3.2 对时间段的统计

在任何实时系统中都有相对重要的处理时间段。由于一段时间可用两个连续时间值之间的差表示,用户可以使用 STS 对象对其进行显式测量。例如,设  $T_i$  为某算法处理第  $i$  帧数据时所花的时间,一个 STS 对象可以用来存储时间序列  $\{T_i\}$  的统计信息。下面的代码片段说明了如何使用 CLK\_gettime、STS\_set 和 STS\_delta 来追踪统计该算法所花的时间:

```
STS_set(&stsObj, CLK_gettime());
'do algorithm'
STS_delta(&stsObj, CLK_gettime());
```

STS\_set 将 CLK\_gettime 得到的时间值保存到 STS 对象的 previous value 域(先前值), STS\_delta 会从中减去该先前值, 得到的结果即执行算法所花的时间 ( $T_i$ )。然后 STS\_delta 会调用 STS\_add, 并将该结果值传递给 STS\_add, 实现对  $T_i$  的追踪。

这样主机端就可显示出该算法被执行的次数, 执行该算法所花的最大时间值, 执行该算法所花时间的总和及平均值。

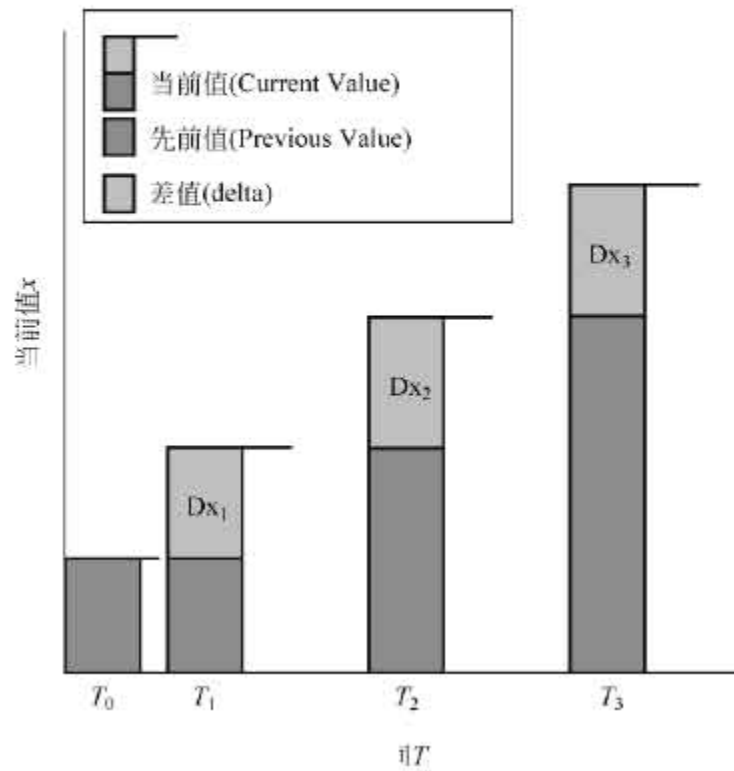
previous value 域是 STS 对象的第 4 个域, 它使得对差值的统计成为可能, 但不支持差值的绝对值统计。

### 3.3.3.3 对差值的统计

STS\_set 和 STS\_delta 都会根据传递给它们的参数更新 STS 对象的 previous 域, 根据它们的调用顺序的不同, 用户可以统计序列中连续数值之间的差值或者数值序列针对某一基值的差值。

例 3-1 给出了采集两个连续数值之间差值的代码示例, 图 3-6 给出了这段代码执行过程中追踪到的数据增量。

例 3-2 给出的代码用来过程中追踪到的数据增量。



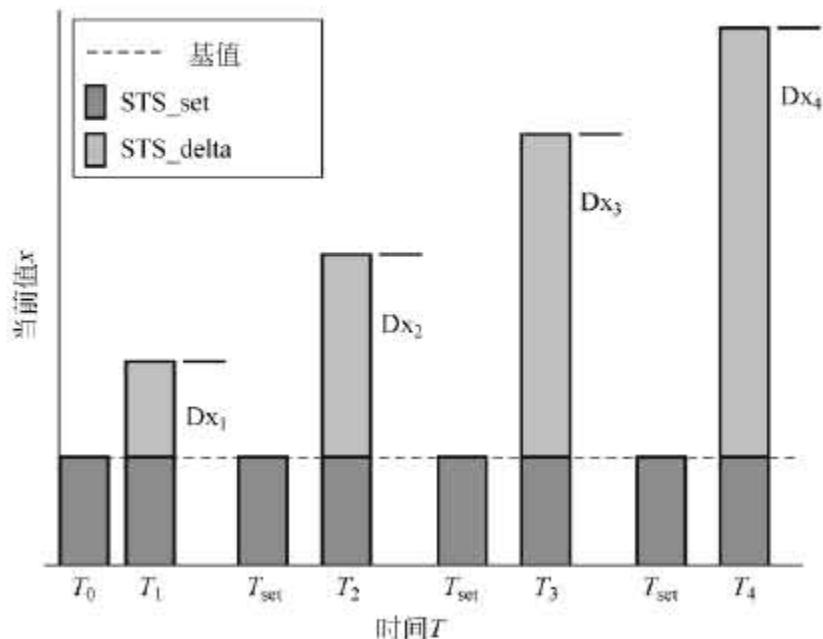


图 3-7 采集当前值相对基值的增量( $Dx_1 \sim Dx_4$ )

### 例 3-1 采集连续差值信息

```
STS_set(&sts, targetValue); /* T0 */
"processing"
STS_delta(&sts, currentValue); /* T1 */
"processing"
STS_delta(&sts, currentValue); /* T2 */
```

```
"processing"  
STS_delta(&sts, currentValue); /* T3 */  
"processing"
```

### 例 3-2 采集相对一个基值的差值

```
STS_set(&sts, baseValue);  
"processing"  
STS_delta(&sts, currentValue);  
STS_set(&sts, baseValue);  
"processing"  
STS_delta(&sts, currentValue);  
STS_set(&sts, baseValue);  
"processing"  
STS_delta(&sts, currentValue);  
STS_set(&sts, baseValue);  
"processing"
```

## 3.3.4 追踪管理器(TRC 模块)

TRC 模块允许应用程序使能或禁止对分析数据的实时追踪采集。例如,目标处理系统可以在发现应用程序行为异常时使用 TRC 模块停止或启动数据采集。

对数据采集的控制是很重要的。它允许用户限制监测对程序行为的影响,保证 LOG 和 STS 对象得到必需的信息,以及在运行时启动或停止事件记录或数值记录。

例如,程序可以在某个事件发生时使能监测,这时一个固定日志对象可以记录日志被使能后最先发生的  $n$  个事件。或者也可以在某个事件发生时禁止监测,这时一个循环日志就可记录关闭日志之前的最后  $n$  个事件。

#### 3.3.4.1 对显式监测的控制

用户可以用下面代码所示的方法使用 TRC 模块控制显式监测:

```
if (TRC_query(TRC_USER0) == 0) {  
    'LOG or STS operation'  
}
```

注意: 如果传递给 TRC\_query 的掩模所指示的所有的追踪(位)都被使能,则其返回值为 0, 否则返回 1。

如果所检查的追踪位没有被使能,这段代码的开销只有几个指令周期。如果应用程序在代码大小方面能够担负这种测试及其包含的 LOG 或 STS 监测调用,在产品程序中保留这些代码来简化开发进程和进行区域诊断是一种非常实用的做法。实际上这种模式也用于 DSP/BIOS 监测内核中。



### 3.3.4.2 对隐式监测的控制

TRC 模块管理着一组追踪位,它们用于控制隐式监测数据的实时获取。为了提高效率,除非追踪被使能,否则目标 DSP 不会存储日志信息或统计信息。(如果用户使用 LOG\_printf 和 LOG\_event 显式地写入消息,或使用 STS\_add 和 STS\_delta 显式地更新统计信息,则不需要为这些操作使能追踪。)

DSP/BIOS 定义了如表 3-2 所示的 TRC 追踪位常量,这些追踪位使得目标 DSP 应用程序可以控制信息收集的启动和停止。当试图捕获关于某个特定事件或者事件集合的信息时,这些追踪位常量就显得比较重要。

表 3-2 TRC 常量

追踪位常量	被使能/禁止的追踪	默认状态
TRC_LOGCLK	记录低分辨率时钟中断	禁止
TRC_LOGPRD	记录系统时钟以及周期函数的启动	禁止
TRC_LOGSWI	记录软件中断函数的触发、启动和完成	禁止
TRC_LOGTSK	记录任务的就绪、启动、阻塞、重新运行以及终止运行等这些事件,这个常量同样用于记录信号灯的发布	禁止
TRC_STSHWI	收集 HWIs 函数中所监测的寄存器值的统计信息	禁止
TRC_STSPIP	计算从数据管道里读取或写入的数据帧的数量	禁止
TRC_STSPRD	收集周期函数执行所占用的系统时钟的统计信息	禁止
TRC_STSSWI	收集软件中断函数从触发到执行完成所用的指令数或者系统时钟数的统计信息	禁止



TRC_STSTSK	收集任务从处于就绪状态直到出现 TSK_deltatime() 函数调用所用时间的统计信息,以定时器中断或 CLK 时钟为单位	禁止
TRC_USER0 和 TRC_USER1	使能或禁止显式监测行为,由 TRC_query 函数检查这些位的设置,并且根据检查结果决定执行还是跳过相应的函数调用,DSP/BIOS 不会设置或使用这些位	禁止
TRC__ GBLHOST	全局启动或停止被使能的追踪类型的采集操作。若要执行任何形式的隐式监测则必须设置该位,通常在主机端通过 RTA 控制面板在运行时设置此位	禁止
TRC__ GBLTARG	控制隐式监测,若要执行任何形式的隐式监测则必须设置该位,并且只能由目标程序来设置	使能

注意：更新任务统计信息。

如果一个任务没有调用 TSK\_deltatime,即使在 RTA 控制面板中使能了 TSK 信息收集,在统计视图中该任务的统计信息也不会被更新。

任务统计信息的处理不同于其他统计信息,因为 TSK 函数一般会运行一个无限循环,并会在等待其他线程时阻塞。相反的,HWI 和 SWI 函数会无阻塞地完成运行。由于这一不同之处,DSP/BIOS 允许程序通过调用 TSK\_settime 来确定一个 TSK 函数处理循环的“开始”,以及通过调用 TSK\_deltatime 来确定该循环的“结束”。

用户可以通过下列途径来使能和禁止这些追踪位：

- ❑ 在主机端,使用 RTA 控制面板,如图 3-8 所示。该控制面板允许用户调整信息采集和运行时间之间的平衡。通过禁止各种隐式监测类型,用户可以减少处理的开销但将会丢失监测信息。默认状态下 RTA 控制面板的所有选项框都是被选中的。
- ❑ 在目标 DSP 代码中,使用 `TRC_enable` 和 `TRC_disable` 函数来使能和禁止追踪位。例如,下面这段 C 代码禁止了对软件中断和周期函数的日志信息追踪：

```
TRC_disable(TRC_LOGSWI | TRC_LOGPRD);
```

又比如：用户有时可能需要将程序运行一整晚或更长的时间来寻找某种特殊情况。当期待的情况发生时,用户应用程序可以执行下面的调用关闭所有的追踪来保存该时刻的监测信息：

```
TRC_disable(TRC_GBLTARG);
```

任何由目标程序对追踪位所作的更改都会反映在 RTA 控制面板中。例如用户可以使目标程序在某一事件发生时禁止全局信息追踪。在主机端,用户就可以简单地等待“Global target enable”复选框被程序自动清除掉时再去检查日志。

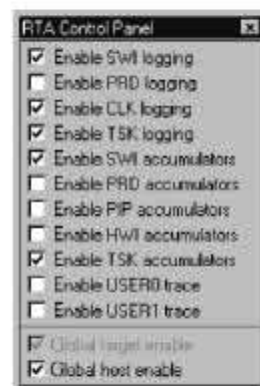


图 3-8 RTA 控制面板对话框

## 3.4 隐式 DSP/BIOS 监测

为了使 DSP/BIOS 分析工具显示执行图 (Execution Graph)、系统统计信息 (System Statistics) 和 CPU 负荷图 (CPU Load), 必要的监测代码会被自动地嵌入 DSP/BIOS 程序来提供隐式监测。用户可以通过使用 CCS 里的 RTA 控制面板使能不同的 DSP/BIOS 隐式监测。

DSP/BIOS 监测的性能是很高效的, 对于一个典型应用程序, 若使能所有类型的隐式监测, CPU 负荷的增加不会超过 1%。

### 3.4.1 执行图

执行图 (Execution Graph) 是一种特殊的图形, 用于显示有关 SWI、PRD、TSK、SEM 和 CLK 的活动信息。用户可以使用 RTA 控制面板或者 TRC 模块的 API 函数来使能或禁止对这些对象的日志记录。执行图中信号灯发布的显示是通过使能或禁止对 TSK 的日志记录来控制的。执行图窗口如图 3-9 所示, 以图形方式显示了各个对象的活动情况。

CLK 和 PRD 事件用于为执行图提供一个时间间隔的度量。为每个日志事件标记时间

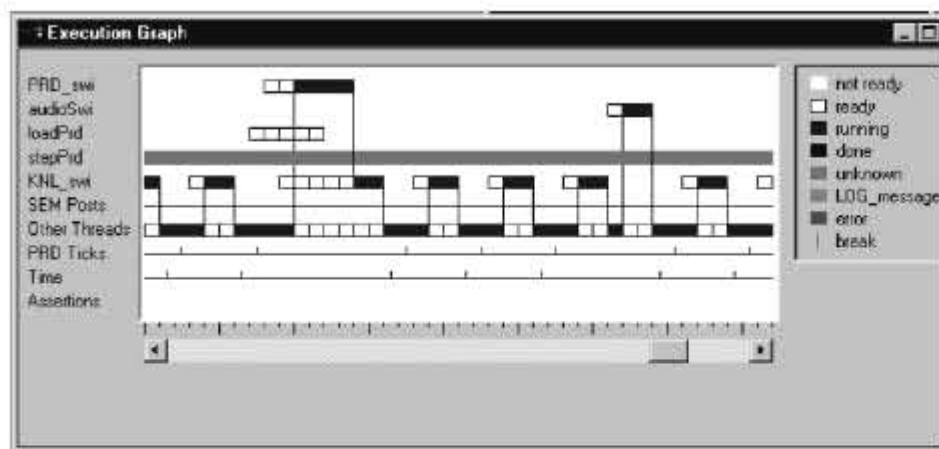


图 3-9 执行图窗口

戳的开销是很大的,因为要获得时间标记就要花费额外的时间和存储空间。所以执行图只是简单地将 CLK 事件和其他系统事件一同记录下来,这使得执行图中时间刻度的显示是非线性的。

除了 SWI、TSK、SEM、PRD 和 CLK 事件之外,执行图还以图形方式显示了一些其他信息。断言(Assertions)用于指示某个实时性限度没有达到,或者侦测到某个无效的状态(可能是由于系统日志崩溃,或是目标 DSP 执行了一个非法操作)。在执行图中以绿色方块显示的“LOG\_message 状态”,会根据用户在应用程序中的 LOG\_message 调用显示在断言追踪线上。内部日志调用产生的错误会以红色方块显示在断言追踪线上。断言追踪线上的红色竖线则指示出了获取系统日志信息过程被打断。

关于执行图中有关程序执行信息的详细解释,请查阅 4.1.5 小节。

### 3.4.2 CPU 负荷图

CPU 负荷(CPU Load)定义为:在整个运行时间中,CPU 用于完成应用程序工作所花的指令周期的百分比,即 CPU 进行如下操作所花时间对总时间的百分比:

- ❑ 运行硬件中断、软件中断、任务和周期函数。
- ❑ 与主机进行 I/O 操作。
- ❑ 运行任何用户定义的例程。
- ❑ 处于节能或硬件空闲模式(只针对 C55x 平台)。

当 CPU 没有做这些工作时,则被认为处于空闲状态。CPU 负荷图窗口如图 3-10 所示。选择菜单“DSP/BIOS”→“CPU Load Graph”即可打开该窗口。

所有 CPU 的活动分为工作时间和空闲时间两部分。要测量一个时间段  $T$  内的 CPU 负荷,需要知道在这段时间内有多少时间用于执行应用程序( $t_w$ ),有多少是空闲时间( $t_i$ ),然后根据下式计算 CPU 负荷:

$$CPU Load = \frac{t_w}{T} \times 100$$

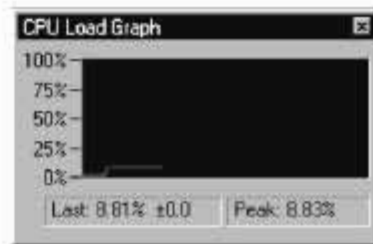


图 3-10 CPU 负荷图



由于 CPU 不是在工作中就是处于空闲状态,即  $T = t_w + t_i$ ,所以也可以写成下式:

$$CPULoad = \frac{t_w}{t_w + t_i} \times 100$$

也可以使用指令周期而非时间来计算 CPU 负荷:

$$CPULoad = \frac{c_w}{c_w + c_i} \times 100$$



对于 C55x 平台,虽然在节能 (power-save) 模式下 CPU 是空闲的,但是这种模式在 C55x 平台上是靠 DSP/BIOS 的 PWRM 模块支持的,所以这时 DSP/BIOS 的空闲循环不能够运行。这就使得节能模式下 CPU 负荷显示为 100%。

### 3.4.2.1 CPU 负荷测量

在一个 DSP/BIOS 应用程序中,当 CPU 进行下列活动中的任何一种活动时,即处于工作状态:

- ☐ 执行硬件中断服务程序。
- ☐ 运行软件中断和周期函数。
- ☐ 运行任务函数。
- ☐ 在空闲循环外面执行用户定义的函数。
- ☐ HST 通道传输数据到主机。
- ☐ 实时分析数据上传到 DSP/BIOS 分析工具。

当 CPU 没有进行以上任何一种活动时,它会进入空闲循环,执行 IDL\_cpuload 函数并调用其他 DSP/BIOS IDL 对象的函数。也就是说,在一个 DSP/BIOS 应用程序中 CPU 的空闲时间就是 CPU 在运行例 3-3 所示的伪代码时所花的时间。

所以要测量一个 DSP/BIOS 应用程序在时间  $T$  内的 CPU 负荷,只要知道有多少时间花费在空闲循环中,有多少时间花费在应用程序的工作上即可。

### 例 3-3 空闲循环

```
'Idle_loop:  
    Perform IDL_cpuLoad  
    Perform all other IDL functions (user/system functions)  
    Goto IDL_loop'
```

在时间  $T$  内,一个  $M$  MIPS(每秒百万条指令)的 CPU 能够执行  $M \times T$  个指令周期。在这些指令周期中,有  $c_w$  个用于执行应用程序工作,剩余的用于执行如例 3-3 所示的空闲循环。如果执行一次该空闲循环所需的指令周期数为  $I_f$ ,且在  $T$  内空闲循环被重复执行了  $N$  次,则总共有  $N \times I_f$  的指令周期花费在空闲循环上,所以有如下的等式成立:

$$MT = c_w + NI_f$$

$$c_w = MT - NI_f$$



### 3.4.2.2 计算应用程序的 CPU 负荷

根据 3.4.2.1 节的等式,可以采用下式计算一个 DSP/BIOS 应用程序的 CPU 负荷:

$$CPULoad = \frac{c_w}{MT} \times 100 = \frac{MT - NI_i}{MT} \times 100 = \left(1 - \frac{NI_i}{MT}\right) \times 100$$

为了计算 CPU 负荷,用户必须知道  $I_i$  和  $N$  的具体数值,然后根据上式进行计算。

在 DSP/BIOS 空闲循环中,函数 IDL\_cpuload 用于更新一个名为 IDL\_busyObj 的 STS 对象,该对象追踪 IDL\_loop 运行次数,以及根据 DSP/BIOS 高分辨率时钟(参考 4.9 节)获得的运行时间。这些信息最终由主机获得并由主机根据上式计算 CPU 负荷。

主机会根据 RTA 控制面板属性对话框中设置的轮询频率来上传目标 DSP 中的统计信息。其中在 IDL\_busyObj 这个 STS 对象里包含的信息用于计算 CPU 负荷,IDL\_busyObj 对象的计数值(count 域)就是空闲循环运行的次数  $N$ ,最大值(maximum 域)不被使用,算术和(total 域)就是以高分辨率时钟为单位的时间  $T$ 。

对于  $I_i$ (在空闲循环中花费的指令周期数),当配置工具中空闲函数管理器的“Auto calculate idle loop instruction count”选项框被选中时,DSP/BIOS 会在调用 BIOS\_init 初始化时计算  $I_i$ 。

这时主机根据获得的  $N, T, I_i$  以及  $M$ ,由下面这个表达式计算 CPU 负荷:

$$CPULoad = \left(1 - \frac{NI_i}{MT}\right) \times 100$$



### 3.4.3 隐式 HWI 监测

通过设置 HWI 对象的隐式监测参数,用户程序可以在每次硬件中断被触发时监测寄存器、数据值以及堆栈指针等。因为 DSP/BIOS 会为每个被监测的 HWI 自动创建一个 STS 对象来获取指定的统计信息,该 STS 对象会自动出现在配置中。图 3-11 和图 3-12 说明了 HWI 监测是如何实现的。

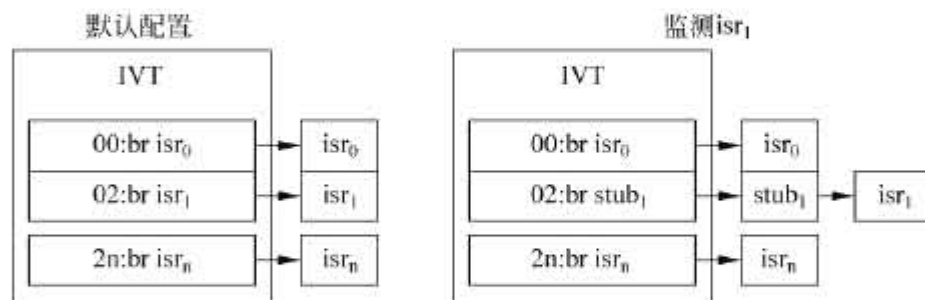


图 3-11 HWI 监测机制(C5000 平台)

对于没有被监测的硬件中断来说,控制权将直接转到 HWI 函数(即中断服务程序),而对于被监测的硬件中断,控制权首先交给一个由配置生成的存根(stub)函数。该存根函数读取指定地址的数据,然后调用选定的 STS 操作以该数据为参数更新自动创建的 STS 对象(数据地址和 STS 操作均由用户在 HWI 属性对话框中设置),最后再跳转到相应的 HWI 函数。

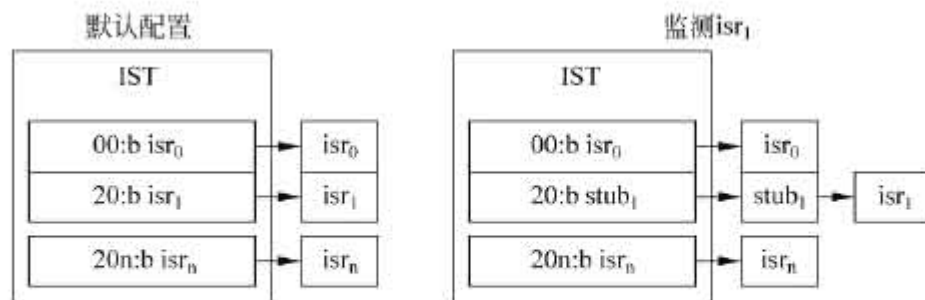


图 3-12 HWI 监测机制(C6000 平台)





RTA 控制面板中的“Enable HWI accumulators”复选框必须是被选中的,这样 HWI 监测才会起作用,否则存根函数不会更新相应的 STS 对象。

这种隐式监测可以对任何一个 HWI 对象进行,在默认状态下这些监测都没有被使能。所以用户应用程序的硬件中断处理性能不会受到影响,除非用户在配置中使能了这种监测。统计对象只在硬件中断处理的一开始被更新,对于每个被监测的硬件中断,每次更新该统计对象只花费 20 到 30 个指令周期。

为一个 HWI 对象使能隐式 HWI 监测的步骤如下:

(1) 在配置工具中打开该 HWI 对象的属性对话框,在 monitor 属性域中选择一个寄存器进行监测。用户可以选择表 3-3 中列出的任何变量进行监测,配置将自动生成一个 STS 对象来采集相应的统计信息。

表 3-3 HWI 对象能够监测的变量

C54x 平台 			C55x 平台 			C6000 平台 			C28x 平台 		
数据值			数据值			数据值			数据值		
系统堆栈栈顶						系统堆栈栈顶					
堆栈指针			堆栈指针			堆栈指针			堆栈指针		
下列通用寄存器：			下列通用寄存器：			下列通用寄存器：			下列通用寄存器：		
ag	ar6	imr	ac0	real	trn0	a0	a12	b6	ah	ph	xar0
ah	ar7	pmst	ac1	rpte	trn1	a1	a13	b7	al	pl	xar1
a1	bg	rea	ac2	rsa0	xar0	a2	a14	b8	idp	st0	xar2
ar0	bh	rsa	ac3	rsa1	xar1	a3	a15	b9	ifr	st1	xar3
ar1	bk	st0	brc0	st0	xar2	a4	a16- a31 (C64x only)	b10	ier	t	xar4
ar2	bl	st1	brc1	st1	xar3	a5		b11		tl	xar5
ar3	brc	t	ifr0	st2	xar4	a6		b12			xar6
ar4	ifr	tim	ifr1	st3	xar5	a7		b13			xar7
ar5		trn	imr0	t0	xar6	a8	b0	b14			
			imr1	t1	xar7	a9	b1	b15			
			reta	t2	xcdp	a10	b2	b16-			
			rea0	t3	xdp	a11	b3	b31			
							b4	(C64x only)			
							b5				



(2) 在 operation 属性域中设置要对该变量值进行的 STS 操作, 用户可以对所选变量的数值进行表 3-4 给出的任何一种操作。对于所有这些操作, STS 对象数据结构的 count 域统计的都是中断被触发的次数。只有最大值和算术和的统计会根据操作的不同有所不同, 所以表中只给出最大值和算术和的操作结果, 平均值由主机进行计算。

表 3-4 STS 操作及结果

STS 操作	结 果
STS_add(* addr)	保存数据值或寄存器值的最大值和总和
STS_delta(* addr)	将数据值或寄存器值与 STS 对象的 prev 属性域的数值进行比较, 保存最大差值和差值总和
STS_add(~ * addr)	将数据值或寄存器值取反, 保存取反后的最大值和总和。执行结果是: maximum 域存储的数值将是最小值的取反值, total 域和 average 域的数值将是实际数值总和及平均值取反的结果
STS_delta(~ * addr)	将数据值或寄存器值取反后, 与 STS 对象的 prev 属性域的数值进行比较, 保存最大差值和差值总和。执行结果是: maximum 域保存的值其实是最小差值取反之后的数值
STS_add(  * addr )	将数据值或寄存器值取绝对值并保存其最大值和总和
STS_delta(  * addr )	将数据值或寄存器值取绝对值后, 与 STS 对象的 prev 属性域的数值进行比较, 保存最大差值和差值总和

(3) 用户也可以设置刚才自动生成的 STS 对象的属性, 在主机端对该 STS 对象的统计值进行滤波。

例如用户可能需要观察系统堆栈的栈顶,判断应用程序对堆栈的使用是否超出所分配的系统堆栈的大小。系统堆栈的栈顶在程序引导时被初始化为 0xBEEF(C5000 平台)或 0xC0FFEE(C6000 平台)。如果该值发生变化,则应用程序要么超出了分配的堆栈,要么某些错误导致应用程序覆盖了堆栈。

下面给出一种观察堆栈是否被超出的方法:

(1) 在配置中,为一个规律性发生的 HWI 使能隐式监测。打开该 HWI 对象的属性对话框,将 monitor 属性域设置为“Top of SW Stack”,并使用 STS\_delta(\* addr)作为 STS 操作。即监测变量为系统堆栈栈顶。

(2) 打开相应的 STS 对象的属性对话框,将 prev 属性域设置为 0xBEEF(C5000 和 C2800 平台)或 0xC0FFEE(C6000 平台)。

(3) 使用 CCS 装载程序,并打开统计观察窗口来观察该 STS 对象。

(4) 运行程序,如果系统堆栈的栈顶发生了变化,该 STS 对象的最大值或算术和将变为一个非零值。

### 3.4.4 最大堆栈深度

用户可以将一个 HWI 对象的 monitor 属性域配置为 stack pointer(堆栈指针),来追踪系统的堆栈指针。这时可以通过 STS 对象的 maximum 域获得当中断发生时系统堆栈出现

的顶部最高位置,这些信息可以用来判断应用程序对系统堆栈的需求(即最大深度)。为了确定系统堆栈的最大深度,可按如下步骤进行:

(1) 使用配置工具,右键点击某个 HWI 对象选择“Properties”打开其属性对话框,将 monitor 属性域设为 stack pointer,同时应将 operation 属性域设为 STS\_add(- \* addr),其他选项保持为默认值。

通过这些改动,用户取反 STS 对象的 maximum 域即可获得堆栈指针的最小值,也即使用堆栈时出现的顶部最高位置,因为堆栈是负向增长的,如图 3-13 所示。

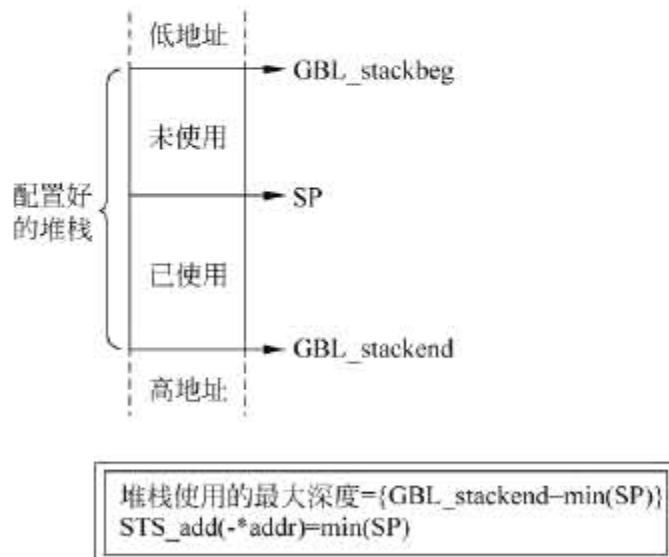


图 3-13 计算堆栈使用的最大深度

(2) 链接应用程序,使用 CCS 的存储器窗口或 map 文件找到由 GBL\_stackend 符号引用的系统堆栈的末尾地址(即栈底)。



(3) 运行应用程序,然后在统计观察窗口中查看为该 HWI 监测自动生成的 STS 对象。该 STS 对象可在配置工具的 STS 模块下找到。

(4) 从系统堆栈的栈底地址值中减去堆栈指针的最小值(STS 对象 maximum 域中的值取反),即可得到堆栈使用的最大深度。

### 3.4.5 中断响应时间

中断响应时间是从中断被触发到执行中断的第一条指令之间的最大时间,用户可以根据下面相应平台的步骤测量定时器中断的中断响应时间:



- (1) 配置 HWI\_TINT 对象来监测 tim 寄存器。
- (2) 设置 HWI\_TINT 对象的 operation 属性域为 STS\_add(~ \* addr)。
- (3) 设置 HWI\_TINT\_STS 对象的 host operation 属性域为  $A * x + B$ 。并设置 A 为 1, B 为 PRD 寄存器的值(在 CLK 模块属性中可以找到)。



注意：现阶段还不能使用 DSP/BIOS 在 C55x 平台上计算中断响应时间，因为 C55x 定时器寄存器的存取地址位于数据空间之外。



(1) 找到 CLK 管理器中 CPU Interrupt 属性域指定的 HWI 对象(默认为 HWI\_INT14)，设置该 HWI 的 monitor 属性域为 Data Value(数据值)。

(2) 查看 CLK 管理器使用的片上定时器(默认为 timer0)，设置 HWI 对象的 addr 参数为该定时器的计数寄存器的地址。

(3) 设置该 HWI 对象的 type 属性域(数据类型)为 unsigned。

(4) 设置该 HWI 对象的 operation 属性域为 STS\_add(\*addr)。

(5) 设置相应的 STS 对象(默认为 HWI\_INT14\_STS)的 host operation 属性域为  $A * x + B$ ，并设置 A 为 4，B 为 0。因为定时器的时钟源是 1/4 的 CPU 主频，所以这里要将 x 乘以 4 得到 CPU 时钟数。




- (1) 配置 HWI\_TINT 对象来监测 tim 寄存器。
- (2) 设置 HWI\_TINT 对象的 operation 属性域为 STS\_add(\*addr)。
- (3) 设置 HWI\_TINT\_STS 对象的 host operation 属性域为  $A * x + B$ 。并设置 A 为 -1, B 为 PRD 寄存器的值。

经过以上设置, STS 对象 HWI\_TINT\_STS(C54x、C28x 平台)或 HWI\_INT14\_STS(C6000 平台)就可以显示出从中断触发到读取定时器计数寄存器之间的最大时钟周期数, 也即定时器中断的中断响应时间。系统实际的中断响应时间至少和该值大小相同。

### 3.5 内核对象观察

内核对象观察工具显示了当前的配置状态以及在目标 DSP 上运行的 DSP/BIOS 对象的状态。

在 CCS 中, 选择菜单“DSP/BIOS”→“Kernel Object View”, 或单击工具栏中的快捷图标, 即可打开如图 3-14 所示的内核对象观察窗口。如果需要, 用户可以打开多个内核对象观察窗口。观察窗口中包含了 3 个区域:

- ☐ 左侧的树型视图区域: 以树型视图显示应用程序中的 DSP/BIOS 对象。
- ☐ 右侧的属性观察区域: 以表格形式显示选中的对象的各种属性。

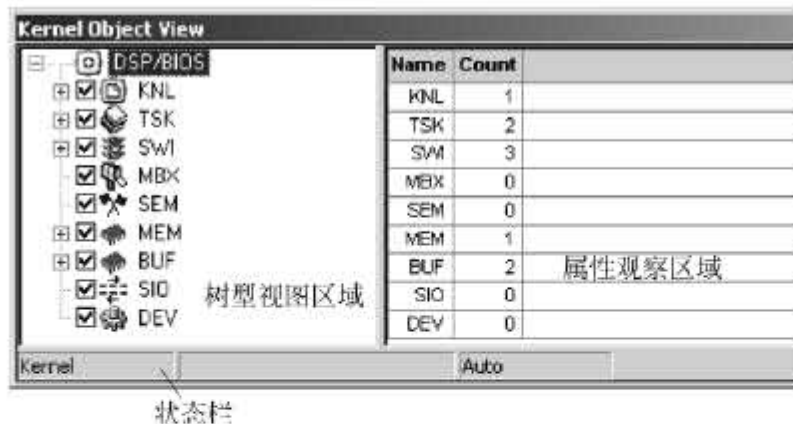


图 3-14 内核对象观察窗口

- ❑ 底部的状态栏：显示了程序正在执行哪个部分（内核或是应用程序）的代码。内核部分（kernel）是用户不能介入的 DSP/BIOS 代码部分；应用程序部分（application）即用户程序代码部分。此外状态栏还会显示当前运行的 TSK 或 SWI 线程，以及数据刷新模式：自动（auto）或手动（manual）。

注意：内核对象观察窗口数据的更新。

与其他 DSP/BIOS 分析工具不同,内核对象观察窗口的数据不会在运行时通过 DSP/BIOS 空闲循环在目标存储器中进行更新,而只有在目标 DSP 暂停时(例如在断点处)或由用户使用右键弹出菜单手动刷新时,目标存储器内的相应数据才被更新,并根据用户的设置(使能或禁止)有选择地上传到主机来刷新内核对象观察窗口。手动刷新实际上也是在短时间内停止目标 DSP 来进行数据的更新和上传的。

### 3.5.1 使用树型视图

用户可以在窗口左侧的树型视图里使能或禁止某类对象的数据显示。视图中列出的对象类型有 KNL(包含系统全局信息)、TSK、SWI、MBX、SEM、MEM、BUF、SIO 和 DEV。

要禁止对某类对象的更新,只要将该对象类型左侧的对勾去掉即可。默认状态下为了减小对性能的影响,SIO 和 DEV 类型是被禁止的。用户也可以禁止掉其他类型的对象来进一步提高性能。

当用户展开某一对象类型时,就可以看到该类型中所有对象的列表。其中包含所有静态创建的对象,以及最新一次刷新时存在的动态创建对象。如果动态创建的对象没有名称,视图会为其生成一个名称并加上尖括号(例如<Task1>)。

当用户单击某个具体的对象时,窗口右侧的属性观察区域会显示出该对象的各种属性。用户也可以使用 Ctrl 或 Shift 键选择多个对象,或直接选中对象类型来观察该类型中所有对象的属性,如图 3-15 所示。也可以同时选择不同类型的对象,但属性表的表头只根据第一个对象的类型显示。



图 3-15 在内核对象视图中观察 TSK 属性

### 3.5.2 使用右键快捷菜单

在内核对象观察窗口的空白处单击右键,会弹出一个快捷菜单,使用其中的命令可以对观察窗口进行控制,如图 3-16 所示。除了窗口定位命令外,该快捷菜单还包含下列命令:

- ❑ **Property Page:** 打开属性对话框。用户可以在该对话框中改变内核对象观察窗口的标题,以及数据输出文件。
- ❑ **Refresh:** 更新窗口中的数据,即在短时间内停止目标 DSP 来执行数据传输操作。当目标 DSP 因为其他原因(如遇到断点)停止时也会执行数据更新。和其他 DSP/BIOS 分析工具不同的是,内核对象观察窗口不在运行状态下更新数据。
- ❑ **Reset Layout:** 重新布局,即设置列宽和排列顺序等。

- ❑ **Global Radix Change:** 改变全局数值(包括地址和计数等)的显示数制(Dec、Hex、Oct)。用户也可以在属性表表头中右键单击某一列,在弹出菜单中更改该列数值的显示数制,如图 3-17 所示。



图 3-16 快捷菜单

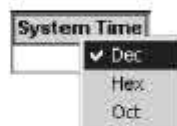


图 3-17 选择显示数制

- ❑ **Save Server Data:** 将当前数据保存到默认文件中。默认文件名为 KOV\_Data.txt。用户可以使用 Property Page 命令打开属性对话框更改该文件。  
该文件中,数据以逗号隔开,可以很方便地导入到电子表格中。对于每种对象类型,数据与视图中属性表相一致,包括表头在内按行显示。  
所有的在最后一次更新时从目标 DSP 读取的数据都被包含在该文件中。如果用户打开了多个观察窗口,并分别使能了不同对象类型的更新,则保存的数据将包括所有在这些窗口中被使能的对象的数据。

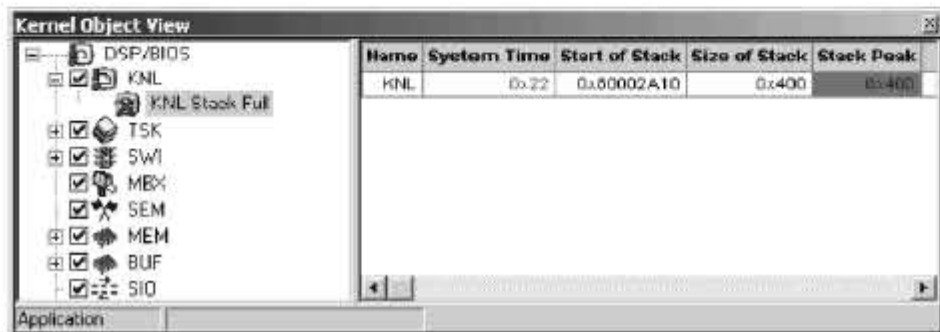


- ❑ **Save Server Data As. . .** : 由用户给出一个文件位置和文件名来保存当前数据。
- ❑ **Manual Refresh Mode/ Auto Refresh Mode**: 在自动刷新模式中(默认),数据会在每次处理器被停止的时候自动更新。在手动刷新模式中,用户必须选择右键快捷菜单的“Refresh”命令来更新数据。状态栏里可以显示内核对象观察窗口处于自动刷新还是手动刷新模式。

### 3.5.3 各种对象类型属性的显示

内核对象观察窗口显示了每类对象的各种属性,下面详细介绍这些属性。

在更新时有变化的数据会以红色显示出来,另外如果堆栈使用量的峰值达到堆栈配置的大小时,堆栈峰值单元格会以红色高亮显示出来,并且一个出错信息会出现在树型视图中的 KNL 类型下,如图 3-18 所示。



### 3.5.3.1 内核

如图 3-19 所示,内核项(KNL 对象)显示了系统全局信息,其属性如下:

Name	System Time	Start of Stack	Size of Stack	Stack Peak
KNL	0x0	0x800021C8	0x400	0x68

图 3-19 内核属性

- ❑ **Name:** 始终为 KNL,在 KNL 类型中系统没有其他对象。
- ❑ **System Time:** CLK 管理器配置使用的时钟的当前值。该时钟用于定时功能和任务警告,并且只有在 TSK 管理器中使能了任务后才会被使用,否则该域始终为 0。
- ❑ **Start of Stack:** 应用程序使用的全局堆栈的起始地址。
- ❑ **Size of Stack:** 全局堆栈的大小。
- ❑ **Stack Peak:** 全局堆栈使用量的峰值。
- ❑ **Start of SysStack:** 系统堆栈的起始地址(C55x 平台)。
- ❑ **Size of SysStack:** 系统堆栈的大小(C55x 平台)。
- ❑ **SysStack Peak:** 系统堆栈使用量的峰值(C55x 平台)。



### 3.5.3.2 任务

如图 3-20 所示,任务(TSK)对象的属性如下:

Name	Handle	State	Priority	Timeout	Blocked On
TSK_idle	0x80002174	Ready	0	0	
task2	0x80002114	Ready	1	0	

Blocked On	Start of Stack	Size of Stack	Stack Peak
	0x80001C48	0x400	0xA0
	0x80002DF8	0x400	0x64

图 3-20 任务属性

- ❑ **Name:** 任务对象的名称。
- ❑ **Handle:** 目标 DSP 中存放任务对象的首地址。
- ❑ **State:** 任务的当前状态,包括就绪(Ready)、运行(Running)、阻塞(Blocked)和终止(Terminated)。
- ❑ **Priority:** 通过
- ❑ **Timeout:** 如果



- ❑ **Blocked On:** 如果任务在进行信号灯(semaphore)同步或邮箱(mailbox)同步时产生阻塞,该域将显示出对应的信号灯或邮箱对象(SEM 或 MBX)的名称。例如,如图 3-21 所示,1 个任务阻塞超时,3 个任务被信号灯阻塞,1 个任务在运行中。用户可以右键单击阻塞任务的信号灯或邮箱,然后在弹出菜单中选择“Go To”命令,来查看 SEM 或 MBX 对象的信息。

Name	Handle	State	Priority	Timeout	Blocked On	Start of Stack
tskControl	0x8000DC6C	Blocked	2	5		0x8000ED20
tskProcess	0x8000DC0C	Blocked	2	0	SEM: 0x8001BEDC	0x8000E920
tskRxSplit	0x8000DB4C	Blocked	2	0	SEM: semIn	0x8000E120
tskTxJoin	0x8000DBAC	Blocked	2	0	SEM: 0x8001BE1C	0x8000E520
TSK_idle	0x8000DCCC	Running	0	0		0x8000DD20

图 3-21 任务属性示例

- ❑ **Start of Stack:** 任务堆栈的起始地址。
- ❑ **Size of Stack:** 任务堆栈的大小。
- ❑ **Stack Peak:** 任务堆栈使用量的峰值。
- ❑ **Start of SysStack:** 系统堆栈的起始地址(C55x 平台)。
- ❑ **Size of SysStack:** 系统堆栈的大小(C55x 平台)。
- ❑ **SysStack Peak:** 系统堆栈使用量的峰值(C55x 平台)。

### 3.5.3.3 软件中断

如图 3-22 所示,软件中断(SWI)的属性如下:

Name	Handle	State	Priority	Mailbox Value	Function	arg0	arg1	Function Address
buffSwi	0x284	Inactive	0x1	0	buffAllocate	0x0	0x0	0x25B4
KNL_swi	0x258	Inactive	0x0	0	KNL_run	0x0	0x0	0x5FC0
PRD_swi	0x22C	Inactive	0x1	0	PRD_F_swi	0x0	0x0	0x5540

图 3-22 软件中断属性

- ❑ **Name:** 软件中断对象的名称。
- ❑ **Handle:** 目标 DSP 中存放软件中断对象的首地址。
- ❑ **State:** 软件中断的当前状态,有效状态包括非活动(Inactive)、就绪(Ready)和运行(Running)。
- ❑ **Priority:** 通过静态配置或动态创建设置的软件中断的优先级,有效优先级范围为 0 到 15。
- ❑ **Mailbox Value:** 软件中断当前的邮箱值。
- ❑ **Function:** 软件中断调用的函数名称。
- ❑ **arg0,arg1:** 传递给该函数的参数,在配置里或创建时设置。
- ❑ **Function Address:** 该函数在目标 DSP 中的地址。

### 3.5.3.4 邮箱

如图 3-23 所示,邮箱(MBX)的属性如下:

Name	Handle	# Tasks Pending	Tasks Pending	#Tasks Blocked Posting	Tasks Posting	# Msgs
mbx	0x800037AC	0		3		0

Max Msgs	Msg Size	Mem Segment
2	8	0

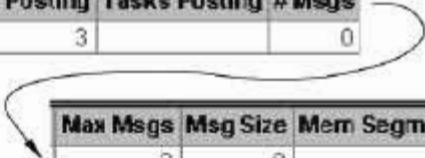


图 3-23 邮箱属性

- ❑ **Name:** 邮箱对象的名称。
- ❑ **Handle:** 目标 DSP 中存放邮箱对象的首地址。
- ❑ **# Tasks Pending:** 当前为了从邮箱读取信息而阻塞的任务的个数,即为了从该邮箱读取一个信息而等待中的任务的个数。
- ❑ **Tasks Pending:** 该域是一个下拉列表,包含当前为了从邮箱读取信息而阻塞的任务的名称。用户可以在该列表中选择一个任务,右键单击该任务后选择“Go To”命令,在观察窗口中显示出该任务的属性。
- ❑ **# Tasks Blocked Posting:** 当前为了向邮箱写入信息而阻塞的任务的个数,即为了向邮箱写入一个信息而等待中的任务的个数。
- ❑ **Tasks Posting:** 该域是一个下拉列表,包含当前为了向邮箱写入信息而阻塞的任务

的名称。用户可以在该列表中选择一任务,右键单击该任务后选择“Go To”命令,在观察窗口中显示出该任务的属性。

- ❑ **# Msgs:** 当前邮箱中包含的消息的个数。
- ❑ **Max Msgs:** 邮箱中能够包含的消息数的最大值,在配置时或动态创建时进行设置。
- ❑ **Msg Size:** 每个消息的大小,以目标 DSP 的最小可寻址数据单元(MADUs)为单位,在配置里或创建时进行设置。
- ❑ **Mem Segment:** 邮箱所处的存储器段的名称,右键单击该域选择“Go To”命令,可在观察窗口中显示出该存储器段(MEM)的属性。

### 3.5.3.5 信号灯

如图 3-24 所示,信号灯(SEM)的属性如下:

Name	Handle	Count	# Tasks Pending	Tasks Pending
0x8000e0dc	0x8000E0DC	0x2	0	
0x8000e004	0x8000E004	0x0	0	
0x8000dfdc	0x8000DFDC	0x4	0	

图 3-24 信号灯属性

- ❑ **Name:** 信号灯对象的名称。
- ❑ **Handle:** 目标 DSP 中存放信号灯对象的首地址。
- ❑ **Count:** 当前的信号灯计数值,即在阻塞发生前能够等待的次数。
- ❑ **# Tasks Pending:** 当前正在等待该信号灯的任务的个数。
- ❑ **Tasks Pending:** 该域是一个下拉列表,包含当前正在等待该信号灯的任务的名称。  
用户可以在该列表中选择任务,右键单击该任务后选择“Go To”命令,在观察窗口中显示出该任务的属性。

### 3.5.3.6 存储器

DSP/BIOS 允许用户配置目标 DSP 上的存储器段(memory segment)对象。一个存储器段中可以包含(也可以不包含)一个存储堆(heap),该存储堆被用于动态存储分配。内核对象观察窗口主要显示存储器段对象中存储堆的属性。

如图 3-25 所示,存储器(MEM)的属性如下:

Name	Largest Free Block	Free Mem	Used Mem	Total Size	Start Address	End Address
IDRAM_base	0x7FF8	0x7FF8	0x8	0x8000	0x80005000	0x8000CFFF

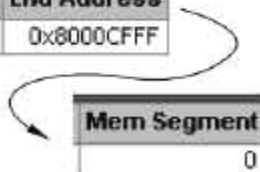


图 3-25 存储器属性

- ❑ **Name:** 配置好的存储器段对象的名称。
- ❑ **Largest Free Block:** 该存储器段的存储堆中可用于动态分配的最大连续存储容量

(以 MADUs 为单位)。

- ❑ **Free Mem:** 存储堆中没有被应用程序使用并能自由分配的存储空间的总容量(以 MADUs 为单位)。
- ❑ **Used Mem:** 存储堆中已分配使用的存储空间的总容量(以 MADUs 为单位)。如果这个值和 Total Size 值相等,则该域的值以红色显示提醒警示。
- ❑ **Total Size:** 存储堆的总容量(以 MADUs 为单位)。
- ❑ **Start Address:** 存储堆的起始地址。
- ❑ **End Address:** 存储堆的结束地址。
- ❑ **Mem Segment:** 存储堆所处的存储器段的编号。

### 3.5.3.7 缓冲池

如图 3-26 所示,缓冲池(BUF)的属性如下:

Name	Segment ID	Size of Buffer	# Buffers in Pool	# Free Buffers	Pool Start Address	Max Buffers Used
BUF0	ISRAM	8	2	2	0x254C	0

图 3-26 缓冲池属性

- ❑ **Name:** 缓冲池对象的名称。
- ❑ **Segment ID:** 缓冲池所处的存储器段的名称。
- ❑ **Size of Buffer:** 缓冲池中单个缓冲区的大小(以 MADUs 为单位)。

- ❑ **# Buffers in Pool:** 缓冲池中缓冲区的个数。
- ❑ **# Free Buffers:** 缓冲池中当前可用的缓冲区个数。
- ❑ **Pool Start Address:** 缓冲池的起始地址。
- ❑ **Max Buffers Used:** 缓冲池中缓冲区使用数量的峰值。

### 3.5.3.8 流 I/O 对象

内核对象观察窗口提供的关于流 I/O(SIO)对象的信息分多个层次显示,如图 3-27 所示。树状层次的第一级列出了由配置静态生成的或由程序动态创建的所有 SIO 对象。在每个 SIO 对象下,列出了流中两个帧队列的信息:“Frames To Device”和“Frames From Device”,分别包含当前处于 device->todevice 和 device->fromdevice 队列中的数据帧列表(详见第 7 章“流 I/O 和设备驱动”)。

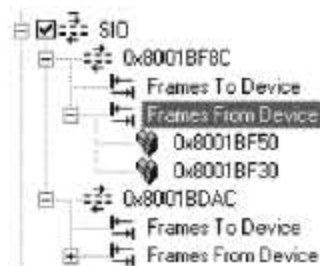


图 3-27 SIO 多层次显示

如图 3-28 所示,SIO 对象的属性如下:

- ❑ **Name:** SIO 对象的名称。
- ❑ **Handle:** 目标 DSP 中存放 SIO 对象的首地址。
- ❑ **Mode:** SIO 对象模式,输入(Input)或输出(Output)。
- ❑ **I/O Model:** I/O 模型,标准(Standard)模型或发放/回收(Issue/Reclaim)模型。

Name	Handle	Mode	I/O Model	Device...	Device ID	Num Frames to ...
inStreamSrc	0x80003D60	Input	Standard	scale	DTR_multiply	0x0
outStreamSrc	0x80003DD0	Output	Standard	pipe0		0x0

Num Frames from ...	Timeout	Buffer size	Number of buf...	Buffer Align..	Mem seg...
0x0	0xFFFFFFFF	0x80	0x3	0x1	0xFFFFFFFF
0x0	0xFFFFFFFF	0x80	0x3	0x1	0xFFFFFFFF

图 3-28 流 I/O 属性

- ❑ **Device Name:** 联结到 SIO 对象的设备的名称,DEV 被禁止时该域为空。右键单击设备名称选择“Go To”命令,可以在观察窗口中显示出该设备的属性。
- ❑ **Device ID:** 被联结的设备的 ID。
- ❑ **Num Frames to Device:** device->todevice 队列中数据帧的个数。
- ❑ **Num Frames from Device:** device->fromdevice 队列中数据帧的个数。
- ❑ **Timeout:** I/O 操作的超时时限。
- ❑ **Buffer Size:** 缓冲区大小。
- ❑ **Number of Buffers:** 流中使用的缓冲区的个数。
- ❑ **Buffer Alignment:** 标准模型下缓冲区的边界限制。
- ❑ **Mem Segment:** 标准模型下,流的缓冲区所处的存储器段的名称。



如果用户选中了某个 SIO 对象的“Frames To Device”或“Frames From Device”队列项。会显示出如图 3-29 中所示的有关这些队列的属性：

Name	Handle	Number of Elements
Frames To Device	0x800054F0	0x0
Frames From Device	0x800054F8	0x3

图 3-29 流传输队列的属性

- ❑ **Name:** 队列名称,“Frames To Device”或“Frames From Device”。
- ❑ **Handle:** 队列中第一个元素(数据帧)的地址。
- ❑ **Number of Elements:** 当前队列中的元素(数据帧)个数。

如果用户选中了某个 SIO 队列中的一个或多个数据帧,会显示出如图 3-30 中所示的有关这些帧的属性:

Name	Handle	Previous	Next
0x8001BF50	0x8001BF50	0x8001BF70	0x8001BF30
0x8001BF30	0x8001BF30	0x8001BF50	0x8001BF70

图 3-30 SIO 帧属性

- ❑ **Name:** 数据帧的名称,实际上显示的是该数据帧在目标 DSP 中的地址。
- ❑ **Handle:** 该数据帧在目标 DSP 中的地址。

- ❑ **Previous:** 队列中前一个数据帧的地址。
- ❑ **Next:** 队列中后一个数据帧的地址。

### 3.5.3.9 DEV 设备

如图 3-31 所示,设备(DEV)对象的属性如下:

Name	Device ID	Type	Stream	Device Function	Device Functions	Device Parameters
printData	DGN_user	DEV_Fxns	0x0	DGN_FXNS	DGN_close [...]	0x80002950
sineWave	DGN_isine	DEV_Fxns	0x0	DGN_FXNS	DGN_close [...]	0x80002978
pipe0		DEV_Fxns	0x0	DPI_FXNS	text [...]	0x0
scale	DTR_multiply	DEV_Fxns	0x0	DTR_FXNS	0x6020 [...]	0x800007F4

图 3-31 设备对象属性

- ❑ **Name:** 设备对象的名称。
- ❑ **Device ID:** 设备的 ID。
- ❑ **Type:** 设备类型,DEV\_Fnx 或 IOM\_Fnx。
- ❑ **Stream:** 联结该设备的 SIO 对象。
- ❑ **Device Function:** 设备函数的地址。
- ❑ **Device Functions:** 一个下拉列表,包含了设备函数表中的函数,如图 3-32 所示。
- ❑ **Device Parameters:** 设备特有参数的地址。

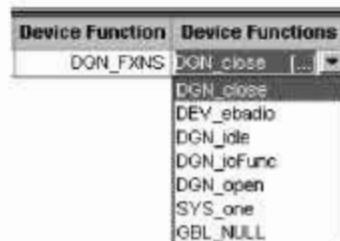


图 3-32 “Device Functions”下拉列表

## 3.6 线程级调试

CCS 中具有线程级(Thread-Level)调试的功能,用户可以使用该功能对 DSP/BIOS 的 TSK 和 SWI 类线程进行逐个单独调试。当用户使用线程级调试时,CCS 会为所选的调试线程打开一个单独的窗口,在其中可以进行如下操作:

- ☐ 运行/暂停线程。
- ☐ 在线程中设置/移除断点来暂停执行。
- ☐ 在线程代码中单步执行。
- ☐ 显示线程的当前状态信息。
- ☐ 读/写存储器。

### 3.6.1 使能线程级调试

DSP/BIOS 是线程级调试的默认 DSP 操作系统,用户可以根据需要重新选择 DSP/BIOS 的版本。CCS 会记住用户最后一次的操作系统选择,更改 DSP 操作系统的步骤如下:

(1) 选择菜单“Tools”→“OS Selector”打开操作系统选择窗口。

(2) 在“Current OS”下拉列表中选择正确版本的 DSP/BIOS。

(3) 右键单击该窗口,在弹出菜单中选择“Close”关闭窗口。

用户可以在装载程序之前或之后使能线程级调试,只要选择菜单命令“Debug”→“Enable Thread-Level Debugging”。打开内核对象视图时也会自动使能线程级调试。

### 3.6.2 打开线程控制窗口

在使用线程调试之前,需要完成以下步骤:

(1) 在 CCS 中装载并运行程序。

(2) 选择菜单“View”→“Threads”,打开一个标题栏为“CPU”的 CCS 窗口,即 CPU 控制窗口,用户可使用其调试整个应用程序。CPU 控制窗口显示了应用程序中 SWI 和 TSK 线程的列表。

- 在目标 DSP 暂停的时候,该列表会被更新,用户也可以使用 Refresh 命令随时进行手动更新。
- 如果应用程序中的线程超过 10 个,将会弹出一个线程选择对话框。
- 选择 Current Thread 则过滤显示出当前运行的线程,这些线程被归为当前线程。
- 列表中包括静态配置和动态创建的所有 TSK 和 SWI 对象。
- 列表中不包括 DSP/BIOS 内核线程,例如 KNL\_swi、PRD\_swi 和 TSK\_idle。

(3) 当用户选中一个线程时,将会打开一个单独的 CCS 窗口,即该线程的控制窗口。新打开的这个 CCS 窗口标题栏显示的是所选线程的名字,被称为线程控制窗口,使得用户可以单独调试此线程。

### 3.6.3 使用线程控制窗口

在 CPU 控制窗口中,用户可对运行在目标 DSP 上的单个应用程序进行调试,但这时 CCS 调试器不能调试目标程序中的任何线程。当遇到任何断点时,CPU 都会停止直到用户选择继续执行。

在线程控制窗口中,可以使用任何的 CCS 调试操作调试一个单独的线程,然而用户需要记得这些操作是在该线程的上下文环境里执行的。比如在线程控制窗口中,CCS 调试器会首先检查一个断点是否作用于当前线程,若不是则自动继续执行到下一个断点处。

由于线程级调试采用停止仿真模式(4.2.3 小节对该仿真模式作了简单介绍),即遇到调试打断事件时(例如执行到断点),CPU 会被暂停运行,这时将阻止 DSP/BIOS 去执行更高优先级的线程,因此 DSP/BIOS 的线程调度可能会偏离正常的实时行为。

只有当所调试的线程活动时该线程调试窗口才是激活的,如果此线程终止运行,则此窗口的调试工具命令就变为灰色不可用。如果此线程被重新触发或者被重新创建,这个窗口也就被重新激活。

可在线程控制窗口中进行的调试活动有：

- ❑ **运行(Run)和单步执行(Step)**：当目标 DSP 由于线程级调试操作被暂停后，只有在 CPU 控制窗口或当前线程控制窗口才能执行运行和单步执行命令。用户不能进入 DSP/BIOS 的 API 函数内进行单步执行，调试器会一次性执行完这些函数。
- ❑ **断点(Breakpoints)**：用户可以在任何线程控制窗口中设置和移除断点，不管该线程是否为当前线程。在线程控制窗口中设置或清除的断点只会对该线程起作用。如果断点所处的代码也被其他的线程运行，断点也不会其他线程控制窗口中被显式地触发，实际上只会简单的暂停一下然后自动接着运行。
- ❑ **停止(Halt)**：在目标 DSP 运行时，任何控制窗口都可以发出停止命令。根据线程是否为当前线程(即当前正在运行中)，停止命令有如表 3-5 所示的不同结果。当目标程序被停止后，目标端的数据便会被读取到主机以更新类似内核对象视图这类的停止模式调试工具。
- ❑ **寄存器操作(Register manipulation)**：当目标 DSP 被暂停时，线程控制窗口提供如表 3-6 所示的寄存器访问方式(不可用的寄存器值用 0xBEEF 标记，向不可用的寄存器写入时会发生错误)。

- ❑ **存储器操作(Memory manipulation):** 对存储器的修改是全局性的,能应用到所有的控制窗口中。在线程控制窗口中修改一个存储器位置和 CPU 控制窗口中修改该存储器位置具有相同的效果。

表 3-5 停止命令的不同结果

线程类型	当 前	非 当 前
SWI	停止于当前执行点	重新进入该 SWI 线程时停止。直到该线程变为当前,线程控制窗口的状态栏显示才会显示“Thread Halting”
TSK	停止于当前执行点	当该 TSK 线程恢复执行时停止。直到该线程变为当前,线程控制窗口的状态栏显示才会显示“Thread Halting”

表 3-6 寄存器访问方式

线程类型	当 前	非 当 前
SWI	对所有寄存器具有完全的读写访问权	无法对寄存器进行访问
TSK	对所有寄存器具有完全的读写访问权	仅能访问现场环境保护时被保存下来的那些寄存器

- ❑ **调用堆栈观察(Call stack viewing):** 任务线程控制窗口中的函数调用堆栈里不显示 TSK\_exit,该调用显示在 CPU 控制窗口的函数调用堆栈里。当前线程的线程控制窗口的调用堆栈与 CPU 控制窗口的相同。非当前线程的控制窗口的调用堆栈中则显示其恢复执行时所在的函数。

### 3.7 用于现场测试的监测

DSP/BIOS 实时运行库和 DSP/BIOS 分析工具能够支持新型的测试和诊断工具,这些工具能和运行在产品系统上的程序进行交互。由于 DSP/BIOS 监测的高效率,最终的产品代码中能够保留这些显式监测代码用来与生产测试和现场诊断工具协同工作。

### 3.8 实时数据交换(RTDX)

实时数据交换(RTDX)提供了实时、连续地观察 DSP 应用程序的行为的能力。RTDX 插件允许系统开发者在不影响应用程序的前提下,进行主机与 DSP 设备之间数据的实时交换。数据可以在主机上使用 OLE 自动化客户应用程序(OLE automation client)进行分析和显示。RTDX 通过真实再现用户系统的行为缩短了开发时间。



RTDX 包括主机模块和 DSP 目标模块两个部分,在 DSP 处理器上运行了一个较小的 RTDX 软件库,DSP 应用程序调用该库里的 API 函数来交换数据。该库通过 JTAG 接口使用基于扫描的仿真器来移动数据。和主机间的数据交换是在 DSP 应用程序运行时实时进行的。

在主机平台上,这个小 RTDX 软件库和 CCS 联合工作。主机显示工具和分析工具通过一个易用的 COM API 接口与 RTDX 通信,实现和目标 DSP 程序间的数据发送与接收。设计者可以选择使用以下的一些标准软件包显示数据:

- ☐ 美国国家仪器的 LabVIEW 工具。
- ☐ Quinn-Curtis 公司的实时图形工具。
- ☐ 微软的 Microsoft Excel。

或者,用户也可以开发自己的 VB 或 VC++ 应用程序,这样除了实现获取数据的目的之外,还能自行设计更形象的数据显示方式。

### 3.8.1 RTDX 应用

RTDX 特别适合于控制、伺服和音频方面的应用。例如,无线通信厂商可以通过采集声音合成算法的输出结果来测试语音应用程序的执行情况。

嵌入式控制系统也可以得益于 RTDX。硬盘驱动器厂商可以在不影响伺服器驱动电机的情況下测试其应用程序的运行情况。发动机控制设计者可以在控制程序运行时分析一些可变因素(如热度或环境因素)。

对于所有这些应用,用户都可以选择对自己最方便的方式来显示这些通过 RTDX 采集到的信息。

### 3.8.2 RTDX 实例

RTDX 可以结合 DSP/BIOS 使用,也可以不结合 DSP/BIOS 使用。在 CCS 安装目录的 C:\ti\tutorial 子目录里 volume4、hostio1 和 hostio2 都是 RTDX 和 DSP/BIOS 结合使用的程序实例,而 C:\ti\examples\target\rtdx 子目录里都是没有结合 DSP/BIOS 使用的 RTDX 使用实例。

### 3.8.3 RTDX 数据流

主机(PC)和目标 DSP(TI DSP)之间的 CCS 数据流如图 3-33 所示。

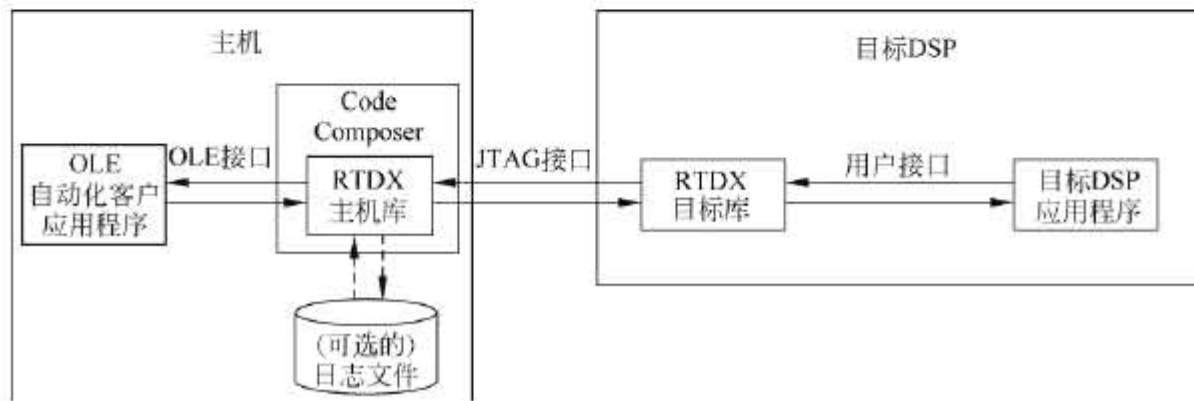


图 3-33 主机和目标 DSP 之间的 RTDX 数据流

### 3.8.3.1 目标 DSP 到主机的数据流

要在目标 DSP 上记录数据,应首先声明一个输出通道,然后使用由用户接口定义的函数向该通道写入数据。数据首先被缓冲存储到一个由 RTDX 目标库定义的目标 DSP 缓冲区,然后通过 JTAG 接口传送到主机。

在另外一端,RTDX 主机库从 JTAG 接口接收这些数据并进行记录。主机将数据记录到一个内存缓冲区中或记录到一个文件中(取决于 RTDX 主机记录模式的设置)。

这些数据流可以由任何 OLE 自动化客户程序获得,一些典型的 OLE 自动化客户程序包括:

- ☐ Visual Basic 应用程序。
- ☐ Visual C++ 应用程序。
- ☐ LabVIEW。
- ☐ Microsoft Excel。

### 3.8.3.2 主机到目标 DSP 的数据流

目标 DSP 为了从主机接收数据,首先需要定义一个输入通道,然后使用用户接口定义

的函数从该输入通道请求数据,该数据请求首先被存储在 RTDX 目标缓冲区,然后通过 JTAG 接口传递给主机。

OLE 客户应用程序能使用 OLE 接口向目标 DSP 发送数据,所有发往目标 DSP 的数据都被写到 RTDX 主机库定义的缓冲区中。当 RTDX 主机库收到一个目标 DSP 发来的读取请求时,主机缓冲区中的数据会通过 JTAG 接口发往目标 DSP,并被实时写入目标 DSP 的请求地址。这些操作完成后,主机会通知 RTDX 目标库。

### 3.8.3.3 RTDX 目标库的用户接口

用户接口提供了一种最安全的方法实现 DSP 应用程序和 RTDX 主机库之间的数据交换。在用户接口中定义的数据类型和函数可以实现如下功能:

- ❑ 允许目标应用程序向 RTDX 主机库发送数据。
- ❑ 允许目标应用程序向 RTDX 主机库请求数据。
- ❑ 提供目标 DSP 上的数据缓冲。数据在被发送给主机之前首先存储在目标 DSP 的缓冲区里,这样的处理方式可以确保数据的完整性以及将实时冲突减少到最小。
- ❑ 提供安全中断机制,可以在中断服务程序中安全地调用这些用户接口函数。
- ❑ 保证正确地使用通信机制。在主机和目标 DSP 之间,要求一次只能使用 JTAG 接口交换一个数据资料。用户接口中定义的函数例程会处理调用时序问题。

### 3.8.3.4 RTDX 主机的 OLE 接口

OLE 接口描述了 OLE 客户程序与 RTDX 主机库进行通信的方法。OLE 接口定义的函数可以实现如下功能：

- ❑ 使得 OLE 客户程序能够访问 RTDX 日志文件里的数据或 RTDX 主机库缓冲区中的数据。
- ❑ 使得 OLE 客户程序能够通过 RTDX 主机库向目标 DSP 发送数据。

### 3.8.4 RTDX 运行模式

RTDX 主机库提供以下两种从目标 DSP 接收数据的模式：

- ❑ 非连续模式：将数据写入到主机的日志文件中，非连续模式一般用于采集有限长度数据以及使用日志文件记录数据的场合。
- ❑ 连续模式：简单的将数据缓存在 RTDX 主机库缓冲区里，并不写入日志文件中。连续模式一般用于连续采集数据并进行显示的场合。

注意：为了排空主机缓冲区并使数据顺畅地从目标 DSP 流向主机，OLE 自动化客户软件必须连续地从目标 DSP 各个输出通道读取数据。否则会导致数据流的中断，降低数据速率，并可能导致通道不能获取数据，另外，OLE 自动化客户软件必须在目标程序启动时就打开所有的输出通道以避免数据丢失。

### 3.8.5 编写汇编代码时的特殊注意事项

由汇编代码编写的应用程序也可以访问 RTDX 目标库的用户接口。关于其调用约定请查阅相应平台的优化编译器用户手册 (Optimizing Compiler User's Guide)。

### 3.8.6 RTDX 目标缓冲区大小

RTDX 目标缓冲区用于临时存放等待发往主机的数据。如果需要发送的数据量很小，可以减小缓冲区的大小；相反地如果数据量很大，则应增加缓冲区的大小。

缓冲区大小的改变通过配置工具实现。在配置工具中用右键单击“RTDX”模块，然后选择“Properties”后进行相应设置即可。

### 3.8.7 RTDX 数据的发送

RTDX 函数库定义了一些数据类型和函数用于完成以下操作：

- ❑ 从目标 DSP 发送数据到主机。
- ❑ 从主机发送数据到目标 DSP。

这些数据类型和函数都在 `rtdx.h` 头文件里定义。

- ❑ 声明宏：
  - RTDX\_CreateInputChannel
  - RTDX\_CreateOutputChannel
- ❑ 函数：
  - RTDX\_channelBusy
  - RTDX\_disableInput
  - RTDX\_disableOutput
  - RTDX\_enableInput
  - RTDX\_enableOutput
  - RTDX\_read
  - RTDX\_readNB
  - RTDX\_sizeofInput
  - RTDX\_write
- ❑ 宏定义：
  - RTDX\_isInputEnabled
  - RTDX\_isOutputEnabled

关于这些 RTDX 函数的详细信息请查看相应平台的 TMS320 DSP/BIOS API Reference Guide 手册。

# 第4章 线程调度

本章描述了 DSP/BIOS 程序可用的线程类型,描述了在程序执行过程中它们的行为和优先级。

## 4.1 线程调度概述

许多实时 DSP 应用都需要同时执行许多看起来不相关的函数,这些函数通常是对外部事件的响应,比如响应特定数据的可用性或某个控制信号的出现。

这些函数被称为线程。不同的系统对线程有不同的定义,要么是狭义的要么是广义的。在 DSP/BIOS 中,线程被广义地定义为由 DSP 执行的任何独立的指令流。一个线程可以是一个子程序、一个中断服务程序(ISR)或一个函数调用。

DSP/BIOS 使得用户应用程序可以由一个线程集合构筑起来,每一个线程执行一个模块化的功能。通过允许高优先级线程抢占低优先级线程,以及允许阻塞、同步、通信等各种



线程间的交互方式,使得多线程的应用程序可在一个处理器上运行。

实时应用程序以这样一种模块化的方式组织起来,而不是由一个单一的集中的轮流检测循环组成。这样更易于设计、实现和维护。

DSP/BIOS 支持多种不同优先级的线程,每种线程类型都有不同的执行和抢占特性。这些线程按照优先级从高到低的顺序排列如下:

- ❑ 硬件中断(HWI),包括 CLK 函数。
- ❑ 软件中断(SWI),包括 PRD 函数。
- ❑ 任务(TSK)。
- ❑ 后台线程(IDL)。

下面分别详细介绍这些不同类型的线程。

### 4.1.1 线程类型

DSP/BIOS 程序中主要有以下四种线程:

- ❑ 硬件中断(HWI):用于响应外部异步事件。当一个硬件中断被触发后,一个 HWI 函数(也被称为中断服务程序或 ISR)会被执行来完成一个有严格时间限制的关键作业。HWI 函数是 DSP/BIOS 应用程序中优先级最高的一类线程,用于完成那些

以 200kHz 左右的频率发生并且必须在  $2\mu\text{s}\sim 100\mu\text{s}$  内完成的应用程序作业。

- ❑ **软件中断(SWI):** 和 HWI 由硬件中断触发不同,软件中断是通过在程序中调用 SWI 函数而触发的。软件中断提供了一个介于 HWI 和 TSK 之间的额外的优先级,用于处理那些时间限制比 TSK 严格但比 HWI 宽松的作业。HWI 线程和 SWI 线程都会一直运行直到完成。软件中断应该用于处理那些执行时限为  $100\mu\text{s}$  或更长的应用程序作业。SWI 使得 HWI 可以将一些不太关键的处理委托给一个优先级比它低的 SWI 线程,从而减少 CPU 在中断服务程序中花费的时间,使其他的 HWI 可以得到运行。
- ❑ **任务(TSK):** 任务的优先级高于后台线程 IDL 而低于软件中断 SWI。任务与软件中断的不同之处在于任务在运行过程中可以等待(阻塞),直到所需的资源可用。DSP/BIOS 提供了许多任务间同步和通信的结构体,如队列、信号灯和邮箱。
- ❑ **后台线程(IDL):** 在 DSP/BIOS 应用程序中,后台线程以最低的优先级执行空闲循环(IDL)。main 函数返回后,DSP/BIOS 应用程序为每个 DSP/BIOS 模块调用启动程序,然后落入空闲循环。空闲循环会连续循环地调用执行所有 IDL 对象的函数,除非有高优先级的线程抢占。只有那些没有时限要求的函数才应该在空闲循环中执行。

在 DSP/BIOS 和  
的环境中执行的。

- ❑ **时钟(CLK)函数**: 该类函数按照片上定时器中断的频率被触发执行。默认情况下, 这些函数由硬件定时器中断触发, 并在与定时器对应的 HWI 函数的环境中执行。
- ❑ **周期(PRD)函数**: 该函数的触发周期基于片上定时器中断周期的倍数或其他事件发生周期的倍数。周期函数是一种特殊的软件中断。
- ❑ **数据通知函数**: 当用户使用管道(PIP)或主机通道(HST)传输数据时执行该函数。当一帧数据被读出或写入时, 数据通知函数被触发, 用来通知写入者或读取者。如果某类线程调用了管道操作函数 `PIP_alloc`、`PIP_get`、`PIP_free` 或 `PIP_put`, 那么数据通知函数会被隐式地触发, 在该线程的环境中执行。

## 4.1.2 线程类型的选择

应用程序中线程类型和优先级的选择将会影响线程是否能及时地被调度并正确地执行, 用户使用 DSP/BIOS 静态配置工具可以很容易地改变线程的类型和优先级。

用户可以根据一些依据来决定程序执行的每个任务该使用哪种类型的线程对象来完成, 这些依据如下:

- ❑ **SWI 或 TSK 与 HWI 对比**: 在硬件中断服务程序中只执行关键处理。对于处理时限在  $5\mu\text{s}$  范围内的硬件中断请求(IRQ), 特别是当时限没有被满足时会产生数据覆盖的情况, 应该考虑使用硬件中断 HWI。对于时间限制为  $100\mu\text{s}$  左右或更长的事件, 应该考虑使用软件中断 SWI 或任务 TSK。用户应该在 HWI 函数中触发软件

中断或者任务去执行较低优先级的处理,这样可以减少硬件中断被禁止的时间,使其他硬件中断得到及时响应。

- ❑ **SWI 与 TSK 对比:** 如果函数的独立性相对较强,数据共享的需求相对较简单,就使用软件中断 SWI。如果数据共享需求比较复杂,就使用任务 TSK。只有任务可以等待其他事件的发生,如资源可用。另外在使用共享数据时 TSK 比 SWI 更灵活:当程序触发一个 SWI 时,该软件中断函数所需的所有输入都应该准备好。SWI 的存储器使用效率更高,因为所有的 SWI 使用同一个堆栈,每个 TSK 线程则拥有各自的堆栈。
- ❑ **IDL:** 当再没有进行其他必需的处理时,就创建若干后台函数来执行非关键的常规作业。IDL 函数没有严格的时间限制,它们只在系统出现处理器空闲时运行。
- ❑ **CLK:** 如果希望一个函数直接由一个定时器中断触发,那就使用 CLK 函数。DSP/BIOS 配置中默认存在的 CLK 对象 PRD\_clock 用来为周期函数产生计数时钟。用户可以添加另外的 CLK 对象以定时器中断的频率运行,但必须保证执行这些 CLK 函数所花的时间足够少,因为它们都在一个内建的 HWI 函数中运行。
- ❑ **PRD:** 如果希望一个函数被触发的周期基于片上定时器中断周期的整数倍,或其他事件(比如外部中断)产生周期的整数倍,那就使用 PRD 函数。这些函数被当作 SWI 函数来执行。
- ❑ **PRD 与 SWI 对比:** PRD 函数的调度由一个 SWI 对象 PRD\_swi 负责,所以所有的 PRD 函数都运行在同一个 SWI 优先级上,这就使 PRD 函数之间不能互相抢占。

但 PRD 函数可以触发更低优先级(比 PRD\_swi 的优先级更低)的软件中断来执行处理时间长的程序,这样在下一次 PRD\_swi 被触发时调度执行的 PRD 函数就可以抢占这些程序。

### 4.1.3 线程特点比较

表 4-1 给出了 DSP/BIOS 提供的线程类型之间的比较。

表 4-1 线程特点比较

特征	HWI	SWI	TSK	IDL
优先级	最高	第二高	第二低	最低
优先级个数	视 DSP 器件而定	15 个,周期函数按照 SWI 对象 PRD_swi 的优先等级运行。任务调度器按照最低的 SWI 优先等级运行	16 个,(其中有一个用于 IDL 循环)	1 个
能否被阻塞	否,运行到结束除非被抢占	否,运行到结束除非被抢占	是	不应被阻塞,否则会阻止 PC 机获取目标 DSP 的信息
执行状态	非活动,就绪,运行	非活动,就绪,运行	就绪,运行,阻塞,终止	就绪,运行

特征	HWI	SWI	TSK	IDL
调度器被如何禁止	HWI_disable	SWI_disable	TSK_disable	程序退出
如何被触发或变为就绪	硬件中断发生	SWI_post, SWI_andn, SWI_dec,SWI_inc, SWI_or	TSK_create	已从 main() 退出且没有其他线程在当前运行
堆栈使用	系统堆栈(1个/应用程序)	系统堆栈(1个/应用程序)	任务堆栈(1个/任务)	使用默认的一个任务堆栈(见注1)
抢占其他线程时的环境保护	用户可制定	固定的一些寄存器被保存到系统堆栈(见注2)	整个环境被保存到任务堆栈	不可用
阻塞时的环境保护	不可用	不可用	保存 C 寄存器组(见相应平台的优化编译器手册)	不可用
与线程共享数据的途径	流, 队列, 管道, 全局变量	流, 队列, 管道, 全局变量	流, 队列, 管道, 资源锁, 邮箱, 全局变量	流, 队列, 管道, 全局变量
与线程同步的途径	不可用	SWI 邮箱	信号灯, 邮箱	不可用
函数钩子	否	否	是: 初始化, 创建, 删除, 退出, 任务切换, 就绪	否

静态创建	包括于默认配置模板中	是	是	是
动态创建	是（见注 3）	是	是	否
动态改变优先级	否（见注 4）	是	是	否
隐式日志记录内容	没有	触发和完成事件	就绪，启动，阻塞，恢复和终止事件	没有
隐式统计内容	被监测的数值	执行时间	执行时间	没有

注：

1. 如果用户在 TSK 管理器属性对话框中禁止了 TSK 管理器，IDL 线程则使用系统堆栈。
2. 见 4.3.7 小节在软件中断抢占时保存寄存器，查看被保存的寄存器的列表。
3. HWI 对象不能被动态创建，因为他们与 DSP 硬件中断是一一对应的，但是中断函数可以在运行时改变。
4. 当一个 HWI 函数调用 HWI\_enter 时，可以向其传递一个位掩模来指示在该 HWI 函数运行时使能哪些中断。一个被使能的中断可以抢占当前 HWI 函数，即使该中断的优先级低于当前中断。

## 4.1.4 线程优先级

线程优先级如图 4-1 所示。在 DSP/BIOS 中，硬件中断的优先级最高。DSP/BIOS 配置中 HWI 对象的个数及其优先级都是固定的，并与 DSP 硬件中断保持一致。在配置工具中 HWI 对象会根据其优先级由高到低排列。只有在一个给定的 CPU 时钟周期内有多个硬

件中断就绪时,这些优先级才被用于决定 CPU 服务这些中断的顺序,其他情况下优先级是无效的。硬件中断会互相抢占,除非通过复位 CSR 寄存器的 GIE 比特(或置位 IER 寄存器的相应比特)来禁止掉其他硬件中断。

软件中断的优先级低于硬件中断,具有 14 个可用的优先级。软件中断可以被任何硬件中断或更高优先级的软件中断抢占,但不能阻塞。

任务的优先级比软件中断低,具有 15 个可用的优先级。任务可以被任何更高优先级的线程抢占,也可以在等待某个资源有效或在等待低优先级线程时被阻塞。

后台空闲循环是优先级最低的线程,当处理器 CPU 空闲时循环运行。



图 4-1 线程优先级

### 4.1.5 让出和抢占

DSP/BIOS 调度器会运行处于就绪状态的优先级最高的线程,除非发生下列情况:

- ❑ 运行中的线程暂时禁止了部分或全部的硬件中断(调用 HWI\_disable),阻止了相应硬件 ISR 的运行。
- ❑ 运行中的线程暂时禁止了软件中断(调用 SWI\_disable),阻止了任何更高优先级的



软件中断抢占当前线程,但并不阻止硬件中断抢占当前线程。

- ❑ 运行中的线程暂时禁止了任务调度(调用 TSK\_disable),阻止了更高优先级的任务抢占当前线程,但并不阻止硬件中断或软件中断抢占当前线程。
- ❑ 最高优先级的线程是一个被阻塞的任务线程,当任务调用了 TSK\_sleep、LCK\_pend、MBX\_pend 或 SEM\_pend 时会发生阻塞。

软件中断和硬件中断都能影响 DSP/BIOS 任务调度。当一个任务被阻塞时,通常是因为该任务在等待一个信号灯,而信号灯可以由软件中断、硬件中断或任务来发布(posting)。当一个 HWI 或 SWI 发布一个信号灯使一个等待中的任务退出阻塞状态时,如果这个任务具有比当前运行任务更高的优先级,那么处理器会切换到该任务去执行。

当运行一个 HWI 或 SWI 时,DSP/BIOS 会使用一个专用的系统中断堆栈,称为系统堆栈(system stack)。而每个 TSK 任务则使用自己的私有堆栈,因此当系统中没有 TSK 任务时,所有线程共享一个系统堆栈。因为 DSP/BIOS 为每个任务使用单独的堆栈,系统以及任务的堆栈都可以设得很小,这样就可以将系统堆栈分配到珍贵的快速存储器中。

表 4-2 给出了当某类线程正在运行时,另一类线程进入就绪状态后,任务调度是如何进行的。调度结果首先取决于该类就绪线程是被使能还是被禁止的。

图 4-2 是一个线程抢占的执行图,其中 SWI 和 HWI 都是被使能的(默认的),当第一个硬件中断(HWI 2)产生时,相应的 ISR 触发了一个比当前运行的软件中断线程(SWI B)优先级更高的软件中断(SWI A)。并且当第一个 ISR 运行时,第二个硬件中断(HWI 1)产生。因为第一个 ISR 没有屏蔽(即禁止)掉第二个硬件中断,因此第二个 ISR 抢占了第一个 ISR。

表 4-2 线程抢占

被触发的线程	正在运行的线程			
	HWI	SWI	TSK	IDL
被使能的 HWI	抢占	抢占	抢占	抢占
被禁止的 HWI	等到被重新使能	等到被重新使能	等到被重新使能	等到被重新使能
被使能的、更高优先级的 SWI	——	抢占	抢占	抢占
被禁止的 SWI	等待	等到被重新使能	等到被重新使能	等到被重新使能
较低优先级的 SWI	等待	等待	——	——
被使能的、较高优先级的 TSK	——	——	抢占	抢占
被禁止的 TSK	等待	等待	等到被重新使能	等到被重新使能
较低优先级的 TSK	等待	等待	等待	——

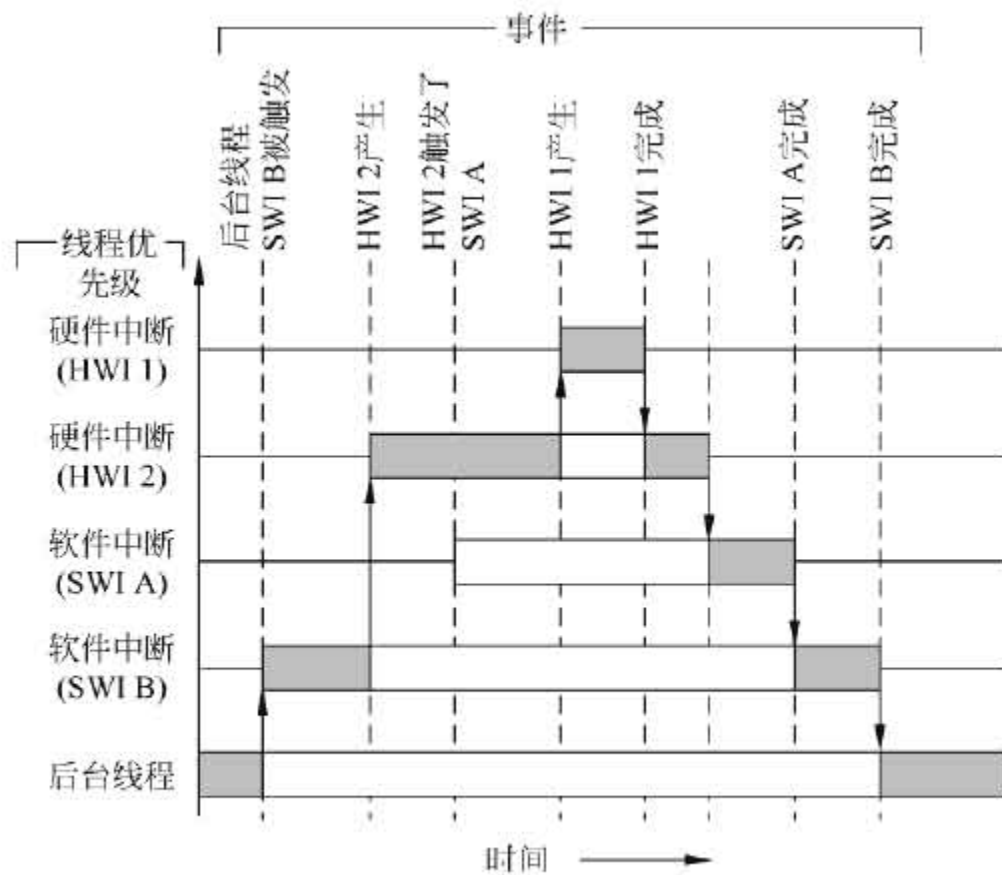


图 4-2 抢占过程示例

从图 4-2 看到,由于优先级较低的 SWI 可以被硬件中断异步抢占,因此第一个 ISR 触发的软件中断在两个硬件中断都执行完毕后才被调度执行。

## 4.2 硬 件 中 断

硬件中断是用来处理应用程序响应外部异步事件时必须执行的关键任务。DSP/BIOS HWI 模块负责管理硬件中断。

在典型的 DSP 应用中,硬件中断是由片上外设或 DSP 外部的设备触发的。中断发生后处理器会通过中断向量跳转到相应的 ISR 地址。一个中断的跳转由一个 DSP/BIOS HWI 对象来负责,跳转的地址可以是一个用户函数或通用的系统 HWI 调度程序(dispatcher)的入口地址。

硬件中断 ISR 可以使用汇编语言、C 语言或者两种语言混合编写。HWI 函数通常采用汇编语言编写以提高效率。如果要完全使用 C 语言编写一个 HWI 函数,应该使用系统 HWI 调度程序,HWI 调度程序会在调用用户 C 函数的前后分别进行现场环境保护与恢复。

所有的硬件中断都会一直运行到结束(在此过程中可以被抢占)。如果一个硬件中断在其 ISR 得到运行机会之前被多次触发,该 ISR 也只运行一次。因此,用户必须尽量减小 HWI 函数执行的代码量。如果 GIE 位被使能,一个硬件中断可能会被任何其他被使能的中断抢占。

如果一个 HWI 函数调用了任何一个 PIP API 函数(PIP\_alloc、PIP\_free、PIP\_get 或 PIP\_put 中之一),那么管道的通知函数(notifyWriter 或 notifyReader)也会在该 HWI 环境中运行。

## 4.2.1 配置硬件中断

在基本 DSP/BIOS 配置中,HWI 管理器包含了与用户 DSP 的每个硬件中断对应的 HWI 对象。

用户可以使用配置工具为 DSP 中的每个硬件中断配置 ISR,即 HWI 函数。只要在 HWI 对象的属性对话框中的 function 属性域中输入 ISR 的函数名即可。DSP/BIOS 会负责建立中断向量表以使每个硬件中断都能被相应的 ISR 处理。用户还可以对中断向量表所在的存储器段(memory segment)进行配置。

DSP/BIOS 在线帮助描述了 HWI 对象及其属性,另外有关 HWI 模块 API 调用的参考资料,请查阅相应平台的 TMS320 DSP/BIOS API 参考手册。

## 4.2.2 禁止和使能硬件中断

用户有时需要在软件中断或在任务执行一段关键处理的过程中暂时禁止硬件中断,通过成对使用 HWI\_disable 和 HWI\_enable/HWI\_restore 函数即可禁止和重新使能硬件中断。

当调用 HWI\_disable 时,中断被全局禁止。在 C6000 平台,HWI\_disable 清零控制状态寄存器(CSR)的 GIE 位,在 C5000 和 C2800 平台,HWI\_disable 置位 ST1 寄存器的 INTM 位,这都将阻止 CPU 接受任何可屏蔽中断。调用 HWI\_enable 或 HWI\_restore 则可以重新使能中断: HWI\_enable 只使能 GIE 位或清零 INTM 位,而 HWI\_restore 可以恢复 HWI\_disable 调用之前每个中断的使能/禁止状态。

## 4.2.3 实时仿真模式对 DSP/BIOS 中断的影响

TI 仿真技术支持两种类型的调试执行控制模式：

- ❑ 停止模式。
- ❑ 实时模式。

这两种模式都会在调试暂停事件发生时挂起程序，比如在软件断点指令出现时，或在访问指定的程序或数据空间时。停止模式提供对程序执行的完全控制，允许禁止掉所有的中断。然而实时模式则允许在其他代码的执行被停止时继续执行有严格时限的中断服务程序。

因此在实时模式下，后台代码在遇到中止事件而被挂起时，CPU 仍然可以继续执行有严格时限的中断服务程序(也可称其为前台代码)。

### 4.2.3.1 实时仿真模式下 C28x 的中断行为

C28x 的实时仿真模式具有以下 3 种不同的状态定义：

- ❑ 调试停止状态(Debug Halt State)。
- ❑ 单指令状态(Single Instruction State)。
- ❑ 运行状态(Run State)。

调试停止状态：由于发生一个中止事件而进入该状态，如一个软件断点指令的出现、一个分析断点/观测点的出现，或一个来自于主机处理器的请求。

当进入该调试停止状态时,时限严格的中断仍能被接收。如果这时一个中断在 IER 和 DBGIER 寄存器中被使能,那么就称该中断为时限严格中断/实时中断。注意在这种情况下 INTM 位被忽略掉了。

然而 DBGM 位可用于阻止 CPU 从某段不应被打断的代码区域进入该停止状态,如果将 INTM 和 DBGM 位一起使用,则可保护此段代码区域不会受到任何类型中断的打断,这同样使得调试器对寄存器/存储器的更新不会发生在该代码区域。

```
SETC INTM, DEGM  
/ Uninterruptable, unhaltable region of code  
CLRC INTM, DBGM
```

如果在实时运行中遇到断点,CPU 会停止并进入调试停止(Debug Halt)状态,这一行为为和停止模式下遇到断点的行为相同。注意到软件断点代替了原来的指令——所以软件断点的执行不可能被安全的忽略或延迟;用户也不能执行想要执行的指令序列。然而其他致使 CPU 停止的打断形式则可以被延迟。所以用户在放置软件断点时一定要考虑周到,需要考察停止发生的位置,反之若使用其他停止 CPU 的形式(如通过 CCS 的 Halt 命令或一个观测点或其他触发事件)时,则不需要了解停止会在哪里发生。

任何时候用户都不应该将软件断点放置在不允许发生中断或停止的地方。但是 CCS 的 Halt 命令或观测点则能够在 CPU 处于不可被中断或不可停止的代码区域使用,因为这



种情况下停止操作可以被延迟到 DBGM 被清零后执行,就像一个中断会被延迟到 INTM 被清零。

例如有一个名为 Semaphore 的变量在一个 ISR 中递增,而在 main 循环中递减。根据中断和调试访问被处理的方式,中断和调试访问都不会发生在如下的斜体字代码区域里。

#### 例 4-1 实时仿真模式下 C28x 的中断行为

```
MAIN_LOOP:
; Do some stuff
SETC INTM, DBGM
/ Uninterruptible, unhaltable region of code
MOV ACC, @Semaphore
SUB ACC, #1; Let's do " * Semaphore--; " really inefficiently!
MOV @Semaphore, ACC
CLRC INTM, DBGM
; Do some more stuff
B MAIN_LOOP
; By default, INTM and DBGM are set in an ISR so you can't halt or interrupt
RT_ISR:
; Do some stuff
MOV ACC, @Semaphore
ADD ACC, #1; Let's do " * Semaphore--; " really inefficiently!
MOV @Semaphore, ACC
```

*; Do .*

*IRET*

注意：如果调试器发出停止指令，以上代码是安全的；程序不会在斜体代码区域停止，所以程序指针 PC 将总是停在“B MAIN\_LOOP”指令处。如果用户将一个观测点设置在 Semaphore 的地址被访问的时刻，直到“CLRC INTM, DBGM”指令被执行 CPU 才会被停止。如果用户在 RI\_ISR 中设置一个硬件断点则会发生同样的结果。如果用户在上面的斜体区域设置了一个软件断点，CPU 将被停止，但是调试器会弹出错误报告指出该操作非法。在这种情况下，应使用一个 C28x 原子(atomic)指令，例如 DEC 或者 INC。

调试停止状态下的中断处理顺序如图 4-3 所示。

单指令状态：当用户通过使用 RUN 1 或 STEP 1 指令让调试器逐个执行单条指令时，将进入单指令状态。CPU 首先执行程序指针 PC 指向的单条指令然后返回到调试停止状态。如果使用 RUN 1 进入该状态且有一个中断发生，则 CPU 能够处理该中断。但是如果是使用 STEP 1 进入该状态，CPU 则不能处理该中断。这对于停止模式和实时模式都是一样的。

请注意：用户可以放心地假设在单步执行时 INTM 不会被改变。如果用户在上面的例子中单步执行代码，那么所有的不可被中断、不可被停止的代码都将会被视为一条指令执行：

**PC initially here** -> SETC INTM, DBGM

```
; Uninterruptible, unhaltable region of code
MOV ACC, @Semaphore
SUB ACC, #1; Let's do " * Semaphore--; " really inefficiently!
MOV @Semaphore, ACC
CLRC INTM, DBG
; Do some more stuff
PC will stop here -> B MAIN_LOOP
```

运行状态：当用户从调试界面使用运行(Run)命令时，将进入运行状态。CPU 会根据 INTM 位和 IER 寄存器的设置处理所有的中断。





图 4-3 调试停止状态下的中断处理顺序

DSP/BIOS 有些代码段需要进行保护以防止被中断,这些代码段被称为关键段。如果这些代码段被中断,且这些中断调用了某些 DSP/BIOS API 函数,就很有可能破坏程序结果。因此使用“SET INTM, DBG M”和“CLRC INTM, DBG M”指令将这些代码保护起来是十分重要的。例 4-2 给出了两个保护代码区域不被中断的例子。

## 例 4-2 不可被中断的代码区域

### (a) 汇编代码

```
.include hwi.h54
...
HWI_disable A; disable all interrupts, save the old intm value in reg A
    'do some critical operation'
HWI_restore A0
```

### (b) C 代码

```
.include hwi.h
Uns oldmask;
oldmask = HWI_disable();
    'do some critical operation; '
    'do not call TSK_sleep(), SEM_post, etc.'
HWI_restore(oldmask);
```

使用 HWI\_restore 而非 HWI\_enable 可以允许 HWI\_disable/HWI\_restore 函数对被嵌套调用。如果嵌套发生,最外层的 HWI\_disable 调用会将中断关闭,而内层的 HWI\_disable 调用将不进行任何操作,直到最外层的 HWI\_restore 被调用时中断才会被重新使能。当在嵌套内层使用 HWI\_enable 时要小心,因为即使那些在外层被禁止的中断也会被该调用使能。

注意：由 HWI\_disable 和 HWI\_enable 包围的代码块内应该避免出现可能引起任务调度的内核调用（如 SEM\_post 和 TSK\_sleep），这是因为如果引起任务调度则可能导致中断被禁止一段不确定的时间长度。

运行状态中的中断处理顺序如图 4-4 所示。

## 4.2.4 中断环境管理

当硬件中断抢占一个正在执行的函数时，HWI 函数必须保存和恢复它将要使用的或修改的寄存器，即进行现场环境保护。DSP/BIOS 提供了汇编语言宏 HWI\_enter 和 HWI\_exit 分别来保存和恢复寄存器。使用这些宏可以为被抢占的函数提供和以前相同的环境，使其正确地恢复运行。除了寄存器环境保存和恢复功能之外，HWI\_enter 和 HWI\_exit 宏还执行以下的系统级操作：

- ❑ 保证 SWI 和 TSK 调度程序在合适的时间被调用。
- ❑ 当 ISR 执行时禁止或恢复某些指定的中断。

如果 ISR 调用了任何可能影响软件中断或信号灯的 DSP/BIOS API 函数，则必须在调用这些函数之前调用 HWI\_enter 宏，而 HWI\_exit 宏则必须在 ISR 函数代码的最后被调用。

为了支持完全使用 C 语言编写的中断服务程序，DSP/BIOS 提供了一个 HWI 调度程序来为中断服务程序执行以上两个宏。一个 HWI 可以使用 HWI 调度程序或显式地调用







图 4-4 运行状态中的中断处理顺序

HWI\_enter 和 HWI\_exit 宏来进行环境保护和中断禁止。用户可以使用配置工具选择是否为 HWI 对象使用 HWI 调度程序。HWI 调度程序是处理中断的首选方式。

HWI 调度程序实际上是在一对 HWI\_enter/HWI\_exit 宏之间调用了用户配置的 HWI 函数,从而允许 HWI 函数完全用 C 语言编写。如果 HWI 调度程序调用了包含 HWI\_enter 和 HWI\_exit 的函数,将会导致系统崩溃。所以使用 HWI 调度程序时只允许存在一个 HWI\_enter 和 HWI\_exit 代码实体。

注意:当 HWI 对象与 C 函数联结时(即使用 C 函数作为 ISR),不要使用 interrupt 关键字或 INTERRUPT pragma 预处理指令来指出该 C 函数为中断服务函数,因为 HWI\_enter 或 HWI\_exit 宏以及 HWI 调度程序包含了这一功能,否则将产生灾难性后果。

不论 HWI\_enter 和 HWI\_exit 宏被显式地调用还是被 HWI 调度程序调用,都会为一个 ISR 调用任何 C 函数做好准备。特别的该 ISR 也做好准备在 HWI 环境下调用任何允许被

调用的 DSP/BIOS API 函数。(关于这些 API 函数的完整列表,请查阅相应平台的 TMS320 DSP/BIOS API 参考手册的“*Functions Callable by Tasks, SWI Handlers, or Hardware ISRs*”部分。)

注意: 在 C6000 和 C54x 平台使用 HWI 调度程序时,HWI 函数不能调用 HWI\_enter 和 HWI\_exit。

不管使用哪种 HWI 调度方式,DSP/BIOS 在执行 HWI 和 SWI 时都使用系统堆栈,如果系统中没有 TSK 任务,那么所有线程将使用同一个系统堆栈。如果有 TSK 任务,每个 TSK 使用自己的私有堆栈。当任务被 HWI 或 SWI 抢占时,DSP/BIOS 在整个中断线程中使用系统堆栈。



在 C54x 平台中,HWI\_enter 和 HWI\_exit 都接受两个参数:

- ❑ 第一个参数为 MASK: 用于指定要被 ISR 保存和恢复的 CPU 寄存器。
- ❑ 第二个参数为 IMRDISABLEMASK,即那些要在 HWI\_enter 和 HWI\_exit 宏调用之间禁止掉的中断的掩模。

当一个硬件中断被触发时,处理器会禁止掉全局中断(通过置位 ST1 寄存器的 INTM 控制位),然后跳转到中断向量表中建立的 ISR 中。HWI\_enter 宏会通过清零 INTM 控制位重新使能全局中断,但在此之前 HWI\_enter 会根据掩模参数(IMRDISABLEMASK)有

选择地禁止掉一些中断,通过清零中断屏蔽寄存器(IMR)中相应的屏蔽位实现。通过使用 HWI\_enter,用户可以有选择地控制哪些中断能抢占当前 HWI 函数,哪些中断不能。

当调用 HWI\_exit 时,用户也可以为其提供一个 IMRRESTOREMASK 掩模参数,这时该掩模参数的值将决定哪些中断会被 HWI\_exit 恢复,通过置位 IMR 中相应的屏蔽位实现。但在 IMRRESTOREMASK 所指定的中断中,HWI\_exit 只会恢复被 HWI\_enter 禁止的那一部分中断。如果在退出 ISR 时用户不希望恢复某个被 HWI\_enter 禁止的中断,不要置位 HWI\_exit 的 IMRRESTOREMASK 参数中该中断对应的掩模位即可,HWI\_exit 不会影响 IMRRESTOREMASK 中未指定的中断屏蔽位的状态。



在 C55x 平台中,HWI\_enter 和 HWI\_exit 能接受 7 个参数,前面 5 个用于指定哪些 CPU 寄存器需要被保存起来,最后两个用来提供两个中断掩模。



在 C6000 平台中,HWI\_enter 和 HWI\_exit 共接受 4 个参数:

- ❑ 前两个参数为 ABMASK 和 CMASK,用于指定要被 ISR 保存和恢复的 A 组寄存器、B 组寄存器和控制寄存器。
- ❑ 第 3 个参数为 IEMASK,即那些要在 HWI\_enter 和 HWI\_exit 宏调用之间禁止掉

的中断的掩模。

当一个硬件中断被触发时,处理器会禁止掉全局中断(通过清零 CSR 寄存器的 GIE 控制位),然后跳转到中断向量表中建立的 ISR 中。HWI\_enter 宏通过置位 GIE 控制位重新使能全局中断,但在此之前 HWI\_enter 会根据中断掩模参数(IEMASK)有选择地禁止掉一些中断,通过清零中断使能寄存器(IER)中相应的使能位实现。通过 HWI\_enter,用户可以有选择地控制哪些中断能抢占当前 HWI 函数,哪些中断不能。

当 HWI\_exit 被调用时,掩模 IEMASK 参数的值将决定哪些中断会被 HWI\_exit 恢复,这通过置位 IER 中相应的使能位实现。但是在 IEMASK 所指定的中断中,HWI\_exit 只会恢复被 HWI\_enter 禁止的那一部分中断。如果在退出 ISR 的时刻用户不希望恢复某个被 HWI\_enter 禁止的中断,不要置位 IEMASK 中该中断对应的掩模位即可。HWI\_exit 不会影响 IEMASK 中未指定的中断屏蔽位的状态。

- ❑ 第 4 个参数为 CCMASK,用于指定放入 CSR 寄存器中缓存控制域的值。该缓存状态在 HWI\_enter 和 HWI\_exit 宏调用之间一直保持有效,该掩模的一些典型常量值在 c62.h62 中定义(例如 C62\_PCC\_ENABLE)。用户可以将 PCC 码和 DCC 码进行逻辑或以生成 CCMASK 参数。如果用户使用 0 值,一个默认值会被使用。该默认值可由用户在配置中的全局设置属性中设置。

当 CLK 时钟管理器被使能时,用来处理片上定时器中断的 CLK\_F\_isr 也会使用配置中设置的缓存状态值。HWI\_enter 在设置缓存控制域之前会保存当前的 CSR 状态。HWI\_

exit 会恢复 CSR,回到被中断的环境。



预定义的掩模 C62\_ABTEMPS 和 C62\_CTEMPS(C62x 系列)或 C64\_ABTEMPS 和 C64\_CTEMPS(C64x 系列)分别指定了所有的 C 语言临时 A 组/B 组寄存器和所有的临时控制寄存器。这些掩模可以用来保存所有能被 C 函数自由使用的寄存器。当在 C6000 平台使用 HWI 调度程序时,用户不能指定一个自己的寄存器集合,所以由这些掩模指定的所有寄存器都被保存和恢复。

如果用户 HWI 函数调用了一个 C 函数,应该这样使用 HWI\_enter 和 HWI\_exit 宏:

```
HWI_enter C62_ABTEMPS, C62_CTEMPS, IEMASK, CCMASK  
'isr code'  
HWI_exit C62_ABTEMPS, C62_CTEMPS, IEMASK, CCMASK
```

在调用任何 C 或 DSP/BIOS 函数之前,HWI\_enter 应该将所有的 C 运行时环境寄存器保存起来,然后使用 HWI\_exit 恢复这些寄存器。

除了保存和恢复寄存器外,HWI\_enter 和 HWI\_exit 宏还会确保当嵌套中断发生时,DSP/BIOS 调度程序只会被最外层的 ISR 调用。如果最外层 HWI 或某个内层嵌套的 HWI 通过调用 SWI\_post 触发了一个 SWI,或是通过调用 SEM\_ipost 或 TSK\_itick 使一个较高优先级的 TSK 就绪,只有最外层的 HWI\_exit 才会调用 SWI 和 TSK 调度程序。SWI 调度



在 C2800 平台中,HWI\_enter 和 HWI\_exit 都接受 4 个参数:

- ❑ 第 1 个参数为 AR\_MASK,指定要被 ISR 保存和恢复的 CPU 寄存器(xar0~xar7)。
- ❑ 第 2 个参数为 ACC\_MASK,指定要保存和恢复的 ACC、p 和 t 寄存器的掩模。
- ❑ 第 3 个参数为 MISC\_MASK,用来指定 ier、ifr、DBGIER、st0、st1 和 dp 寄存器的掩模。
- ❑ 第 4 个参数为 IERDISABLEMASK,用来指定 IER 中的那些使能位被关闭。

当一个硬件中断被触发时,处理器会关闭 IER 中的使能位并禁止掉全局中断(通过置位 ST1 寄存器的 INTM 控制位),然后跳转到中断向量表中建立的 ISR 中。HWI\_enter 宏会通过清零 INTM 控制位重新使能全局中断,但在此之前 HWI\_enter 会根据中断掩模参数 (IERDISABLEMASK)有选择地禁止掉一些中断,通过清零中断使能寄存器 (IER) 中相应的使能位实现。

当调用 HWI\_exit 时,用户也可以为其提供一个 IERRESTOREMASK 掩模参数,这时该掩模参数的值将决定哪些中断会被 HWI\_exit 恢复,通过置位 IER 中相应的屏蔽位实现。但在 IERRESTOREMASK 所指定的中断中,HWI\_exit 只会恢复被 HWI\_enter 禁止那一部分中断。如果在退出 ISR 时用户不希望恢复某个被 HWI\_enter 禁止的中断,不要置位 HWI\_exit 的 IERRESTOREMASK 参数中该中断对应的掩模位即可,HWI\_exit 不会影响

IERRESTOREMASK 中未指定的中断屏蔽位的状态。

关于一个 ISR 中能够调用的 DSP/BIOS 函数的完整列表,可参见相应平台的 TMS320 DSP/BIOS API 参考手册。

注意:任何引用了 DSP/BIOS 函数的汇编或 C 语言 HWI 函数都必须被 HWI\_enter 和 HWI\_exit 包围起来。使用 HWI 调度程序能够满足这一要求。

例 4-3、例 4-4、例 4-5 和例 4-6 给出了不使用 HWI 调度程序而使用汇编语言构造一个最小 ISR 的代码示例。例 4-3 用于 C6000 平台,例 4-4 和例 4-5 分别用于 C54x 平台和 C55x 平台,例 4-6 用于 C28x 平台。



#### 例 4-3 在 C6000 平台中构造 ISR

```
;
; ===== myclk.s62 =====
;
    .include "hwi.h62"      ; macro header file
IEMASK    .set 0
CCMASK    .set c62_PCC_DISABLE
    .text
```

```

;
; ===== myclkisr =====
;

    global _myclkisr
_myclkisr:
    ; save all C run-time environment registers
    HWI_enter C62_ABTEMPS, C62_CTEMPS, IEMASK, CCMASK

    b      TSK_itick      ; call TSK itick (C function)
    mvkl   tiret, b3
    Mvkh   tiret, b3

    Nop    3
tiret:
    ; restore saved registers and call DSP/BIOS scheduler
    HWI_exit C62_ABTEMPS, C62_CTEMPS, IEMASK, CCMASK
    .end

```



例 4-4 在 C54x 平台中构造 ISR



```

;
; ===== _DSS_isr =====
;
; Calls the C ISR code after setting cpl
; and saving C54_CNOTPRESERVED
;
    .include "hwi.h54" ; macro header file
_DSS_isr:
    HWI_enter    C54_CNOTPRESERVED, 0fff7h
    ; cpl = 0
    ; dp = GBL_A_SYSPAGE
    ; We need to set cpl bit when going to C
    ssbx    cpl
    nop                ; cpl latency
    nop                ; cpl latency
    call    _DSS_c_isr
    rsbx    cpl    ; HWI_exit precondition
    nop                ; cpl latency
    nop                ; cpl latency
    ld    #GBL_A_SYSPAGE, dp
    HWI_exit    C54_CNOTPRESERVED, 0fff7h

```

## 例 4-5 在 C55x 平台中构造 ISR

```

;
; ===== _DSS_isr =====
;
_DSS_isr:
    HWI_enter C55_AR_T_SAVE_BY_CALLER_MASK,
              C55_ACC_SAVE_BY_CALLER_MASK,
              C55_MISC1_SAVE_BY_CALLER_MASK,
              C55_MISC2_SAVE_BY_CALLER_MASK,
              C55_MISC3_SAVE_BY_CALLER_MASK,
              0FFF7h,0
              ; macro has ensured 'C' convention,
              ; including SP alignment!

    call _DSS_cisr

    HWI_exit C55_AR_T_SAVE_BY_CALLER_MASK,
            C55_ACC_SAVE_BY_CALLER_MASK,
            C55_MISC1_SAVE_BY_CALLER_MASK,
            C55_MISC2_SAVE_BY_CALLER_MASK,
            C55_MISC3_SAVE_BY_CALLER_MASK.

```



#### 例 4-6 在 C28x 平台中构造 ISR

```

;
; ===== _DSS_isr =====
;
_DSS_isr:
    HWI_enter                AR_MASK, ACC_MASK, MISC_MASK, IERDISABLEMASK
    lcr                      _DSS_cisr
    HWI_exit                  AR_MASK, ACC_MASK, MISC_MASK, IERDISABLEMASK

```

### 4.2.5 寄存器

DSP/BIOS 应用程序中现场环境处理时保存和恢复的寄存器同样遵照标准 C 编译器代码编写规范。关于被寄存器保存及恢复的更多信息请查阅相应平台的优化编译器手册 (Optimizing Compiler User's Guide)。

## 4.3 软件中断

DSP/BIOS 的 SWI 模块提供了软件中断的能力。通过在程序中调用一个 DSP/BIOS API 函数,例如 SWI\_post,即可触发软件中断。软件中断的优先级介于硬件中断和任务之间。

DSP/BIOS 的 SWI 模块有别于任何处理器本身特有的软件中断指令。

SWI 线程适用于处理发生频率较低的或者实时限制没有硬件中断严格的应用程序作业。

能够触发软件中断的 DSP/BIOS API 函数有:

- ☐ SWI\_andn。
- ☐ SWI\_dec。
- ☐ SWI\_inc。
- ☐ SWI\_or。
- ☐ SWI\_post。

SWI 管理器控制所有软件中断的执行。当应用程序调用上面任意一个 API 函数时,SWI 管理器将调度执行与该软件中断对应的 SWI 函数。SWI 管理器使用 SWI 对象来处理应用程序中的所有软件中断。

当软件中断被触发时,它会在所有等待中的硬件中断都执行完后才开始执行。正在运行的 SWI 线程在任何时刻都可以被硬件中断抢占,并且 SWI 线程会在硬件中断执行完毕后才恢复执行。另一方面,SWI 线程总能抢占任务的执行。所有等待中的 SWI 线程执行完之后具有最高优先级的任务才可以被调度执行。在效果上,软件中断就像是一个最高优先级的任务。

注意:关于 SWI 有以下两点需要注意:

除非被硬件中断或更高优先级的 SWI 抢占,否则软件中断将一直执行到完毕。

如果要在 HWI 中断服务程序内部调用任何可能触发软件中断的 SWI 函数,那么调用该函数的代码序列要么必须被一对 HWI\_enter 和 HWI\_exit 调用“包装”起来,要么使用 HWI 调度程序。

### 4.3.1 创建 SWI 对象

像其他 DSP/BIOS 对象一样,用户可以动态地(调用 SWI\_create)或静态地(在配置中)创建 SWI 对象。动态创建的 SWI 对象可以在程序执行过程中被动态地删除。

要添加一个新软件中断到配置中,只要在配置工具里为 SWI 管理器创建一个新的 SWI

对象。在每个 SWI 对象的属性对话框中的 function 属性域,用户可以设置此软件中断被触发时要运行的函数,即 SWI 函数。用户还能向每个 SWI 函数传递最多两个参数。

在 SWI 管理器的属性对话框中,用户可以决定 SWI 对象被分配到哪个存储器段(memory segment)。当软件中断被触发并且被调度执行时,SWI 管理器会访问相应的 SWI 对象。

DSP/BIOS 在线帮助描述了 SWI 对象及其参数。有关 SWI 模块 API 函数的信息请查阅相应平台的 TMS320 DSP/BIOS API 参考手册。

要动态建立一个软件中断,使用如下的语法调用 SWI\_create:

```
swi = SWI_create(attrs);
```

其中,swi 是一个句柄,参数 attrs 指向 SWI 的属性结构体。SWI 的属性结构体(结构体类型为 SWI\_Attrs)包含了一个 SWI 所有可以静态配置的属性。如果 attrs 为 NULL,则使用一套默认的属性。一般情况下,attrs 中至少会为该句柄包含一个函数名,即 SWI 函数。

注意: SWI\_create 只能在任务级调用,而不能在 HWI 或其他 SWI 中调用。

调用 SWI\_getattrs 可以获得 SWI 对象所有的 SWI\_Attrs 属性,其中一些属性可能在程序执行过程中被修改,但它们一般包含的是对象创建时所赋予它们的值。

```
SWI_getattrs(swi, attrs);
```

### 4.3.2 在配置工具里设置软件中断优先级

软件中断共有 15 个不同优先级,用户可以在不超过存储器限制的情况下,为每个等级创建足够多的软件中断。用户可以为实时性要求苛刻的软件中断设置较高的优先级,给时间要求不严格的软件中断设置较低的优先级。

在配置工具里设置软件中断的优先级,可按照以下步骤进行:

(1) 在配置工具里选中 SWI 管理器 (software interrupt manager),窗口右侧会出现一个窗格,其中按优先级顺序排列出所有的 SWI 对象,如图 4-5 所示。(如果窗格中没有显示 SWI 对象列表,右键单击 SWI 管理器然后选择“View Ordered collection”)

(2) 要改变一个 SWI 对象的优先级,只需用鼠标将 SWI 对象拖动到相应的优先级文件夹中即可。例如,为了将对象 SWI0 的优先级改为 3,首先用鼠标选中该对象然后拖动到 Priority 3 文件夹里即可。

软件中断有 15 个优先级,从最低的 SWI\_MINPRI(0)到最高的 SWI\_MAXPRI(14)。优先级 0 保留给 KNL \_swi 对象的,它用来运行任务调度程序。有关优先级和堆栈大小限制之间的关系,请查看 4.3.3 节。用户不能对同一优先级中的多个软件中断进行

排序。

用户也可以在 SWI 对象的属性对话框中设置对象的优先级,如图 4-6 所示。只要在 priority 下拉列表里选择相应的数字(从 0 到 14,最高优先级为 14)即可。

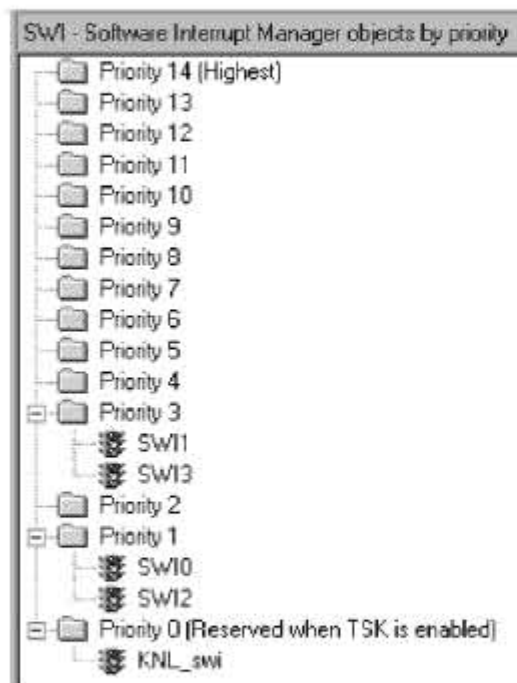


图 4-5 软件中断管理器



图 4-6 SWI 属性对话框

### 4.3.3 软件中断优先级和应用程序堆栈大小



除了任务线程,所有 DSP/BIOS 中的线程使用同一个系统堆栈。当一个软件中断抢占其他线程时,系统堆栈会用于保存寄存器现场环境。为了满足同一时刻出现的最大数量的线程抢占,每增加一个软件优先级别,所需的系统堆栈大小会随之增加。所以从堆栈大小的角度来看,给所有的软件中断对象设置相同的优先级别相比给每个软件中断对象设置不同优先级别效率会更高。

在 MEM 模块中设置的系统堆栈默认大小为 256 字,用户可以在配置中改变其大小。所需系统堆栈大小的估计值会显示在配置工具顶部的状态栏上。

用户应用程序中最多可以有 15 个软件中断优先级,每多一个优先级别所需的系统堆栈就更大。如果看到一个弹出的消息框指示“the system stack size is too small to support a new software interrupt priority level(系统堆栈太小,不能支持新的软件优先级别)”,那么就应该在 MEM 管理器属性对话框中增加系统堆栈的大小。

当创建第一个 PRD 对象时,会自动生成一个新的 SWI 对象 PRD\_swi(参见 4.10 节“周期函数管理器(PRD)和系统时钟”)。如果在创建第一个 PRD 对象之前没有创建任何的 SWI 对象,那么 PRD\_swi 对象会使用优先级 1,并且使所需的系统堆栈大小有所增加。

当 TSK 管理器被使能时,TSK 调度程序(由 SWI 对象 KNL\_swi 来运行)占用最低的 SWI 优先级,其他的任何 SWI 对象都不能使用这个优先级。

### 4.3.4 软件中断的执行

通过调用 `SWI_andn`、`SWI_sec`、`SWI_inc`、`SWI_or` 和 `SWI_post` 可以使软件中断被调度执行。这些函数本身可以在程序的任何地方调用——中断服务程序 ISR 中，周期函数中，空闲函数中或其他软件中断函数中。

当一个 SWI 对象被触发时，SWI 管理器将该软件中断添加到一个被触发软件中断的列表中等待执行，然后 SWI 管理器检查软件中断当前是否被使能。如果没有被使能，例如程序正处于一个 HWI 函数中，SWI 管理器会将执行权交给当前 HWI 线程。

如果被使能，SWI 管理器将该 SWI 对象的优先级和当前运行线程的优先级进行比较。若当前运行线程是后台空闲循环 IDL 或是一个更低优先级的 SWI，那么 SWI 管理器将这个 SWI 对象从被触发 SWI 对象列表中移除，并将 CPU 控制权从当前线程转交给此 SWI 对象，开始执行 SWI 函数。

如果当前运行的线程的优先级大于等于被触发的 SWI，SWI 管理器仍将控制权交给当前线程。当以前被触发的优先级大于等于该 SWI 对象的所有其他 SWI 运行结束后，该 SWI 对象的函数才能运行。

注意：有关 SWI 要记住两点：

当一个 SWI 开始执行后，它必须无阻塞地运行到结束。

当在 HWI 中调用时，调用任何会触发软件中断的 SWI 函数的代码序列必须被包装在一个 `HWI_enter`/`HWI_exit` 宏调用对中，或者由 HWI 调度程序来调用。

SWI 函数可以被一个更高优先级的 SWI 或者一个 HWI 抢占,但不会阻塞,用户不能在一个软件中断等待某些资源(比如一个设备)变得可用时,将其挂起。

如果一个软件中断在 SWI 管理器将其从被触发 SWI 对象列表中移除之前,被触发了多次,其 SWI 函数只会执行一次。这一点类似硬件中断的特性:即在 CPU 清除中断标志寄存器中相应的中断标志位之前,如果该硬件中断被触发了多次,对应的 HWI 只会执行一次。

应用程序不应该对优先级相同的 SWI 对象被调用的顺序进行任何假设。一个 SWI 对象可以很安全地触发自身(或被其他中断触发)。如果有多个 SWI 对象处于等待中,它们都会在任何任务线程运行之前被调用。

### 4.3.5 使用 SWI 对象的邮箱

每个 SWI 对象都有一个 32 位(C6000)或 16 位(C5400)的邮箱,可以用于决定是否触发该软件中断,也可作为考察 SWI 函数的数值。

用于触发 SWI 的 API 函数可以对邮箱值作不同的操作,并根据不同的限制条件触发 SWI 对象。其中 SWI\_post、SWI\_or 和 SWI\_inc 可以无条件地触发 SWI 对象:

- ❑ 在触发 SWI 时,SWI\_post 不改变 SWI 对象邮箱的值。

❑ SWI\_or 在触发 SWI 之前,会根据传递给它的掩模参数置位邮箱的某些位。

❑ SWI\_inc 在触发 SWI 之前将邮箱值加 1。

SWI\_andn 和 SWI\_dec 只在邮箱值变为 0 时触发 SWI 对象,它们对邮箱值的操作为:

❑ SWI\_andn 会根据传递给它的掩模参数清零邮箱的某些位。

❑ SWI\_dec 将邮箱值减 1。

表 4-3 给出了这些函数的区别。

使用 SWI 邮箱可以更好地控制触发 SWI 的条件,也可以控制软件中断被触发后 SWI 函数应该被执行的次数。

要访问一个 SWI 对象的邮箱值,可以在该 SWI 函数中调用 SWI\_getmbox 获得。SWI\_getmbox 只能在 SWI 函数中调用,其返回值是该 SWI 对象从被触发 SWI 对象队列中移除之前其邮箱的值。

表 4-3 SWI 对象触发函数

动 作	将邮箱看作位掩模	将邮箱看作计数器	不改变邮箱值
无条件触发	SWI_or	SWI_inc	SWI_post
为 0 时触发	SWI_andn	SWI_dec	——

当 SWI 管理器从被触发 SWI 对象列表中移除一个 SWI 对象时,其邮箱值被复位为其

时调用 `SWI_getmbox` 返回的邮箱值,该值是在 SWI 对象被从被触发 SWI 队列中移除那一刻锁存起来的邮箱值。SWI 被从等待列表中删除并被调度运行之后,其邮箱值立刻被复位,邮箱值的自动复位使得应用程序有能力在新的触发产生时对 SWI 邮箱的值进行有效地更新,即使这时该 SWI 函数还没有执行完毕。

例如,一个 SWI 对象在被移出队列之前,如果被多次触发,一般情况下,SWI 管理器只会调度其函数执行一次,然而,如果应用程序需要该 SWI 函数在这种情况下执行多次,可以利用 `SWI_inc` 按照图 4-7 那样来触发该 SWI 对象。

如果使用 `SWI_inc` 触发一个 SWI 对象,一旦 SWI 管理器调用并执行对应的 SWI 函数,就可以在函数中访问该 SWI 对象的邮箱,获知该 SWI 在运行之前被触发了多少次,然后将相同的一段程序执行同样的次数来满足应用程序的需求。

如果一个软件中断必须在多个事件都发生的条件下才能被触发,应该使用 `SWI_andn` 来触发该 SWI 对象,如图 4-8 所示。例如一个软件中断在执行之前必须等待两个不同设备的输入数据,其邮箱应该被配置有两个被置 1 的比特位。当提供输入数据的两个程序都完成了它们的任务时,都应该调用 `SWI_andn` 以互补的位掩模参数来清除邮箱默认值中的置位比特。这样该软件中断就只会两端的数据都准备好时被触发。

程序配置		邮箱值	SWI_getmbox()的返回值
SWI对象myswi 函数myswiFxn()		0	
程序执行	<ul style="list-style-type: none"> <li>调用SWI_inc(&amp;myswi)</li> <li>myswi被触发</li> </ul>	1	
	<ul style="list-style-type: none"> <li>调用SWI_inc(&amp;myswi)</li> <li>myswi在被调度执行之前又一次被触发</li> </ul>	2	
	<ul style="list-style-type: none"> <li>SWI管理器从已触发SWI列表中移除myswi对象</li> <li>myswiFxn()被调度</li> </ul>	0	2
	<ul style="list-style-type: none"> <li>myswiFxn()开始执行</li> </ul>	0	2
	<ul style="list-style-type: none"> <li>myswiFxn()被调用</li> <li>SWI_inc(&amp;myswi)的ISR抢占</li> <li>myswi被添加到已触发SWI列表中</li> </ul>	1	2

```

myswiFxn()
{...
  repetitions=SWI_getmbox();
  while(repetitions--){
    'run SWI routine'
  }
}

```

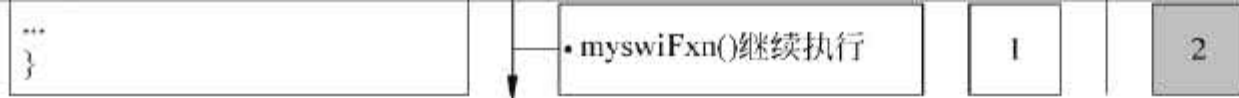
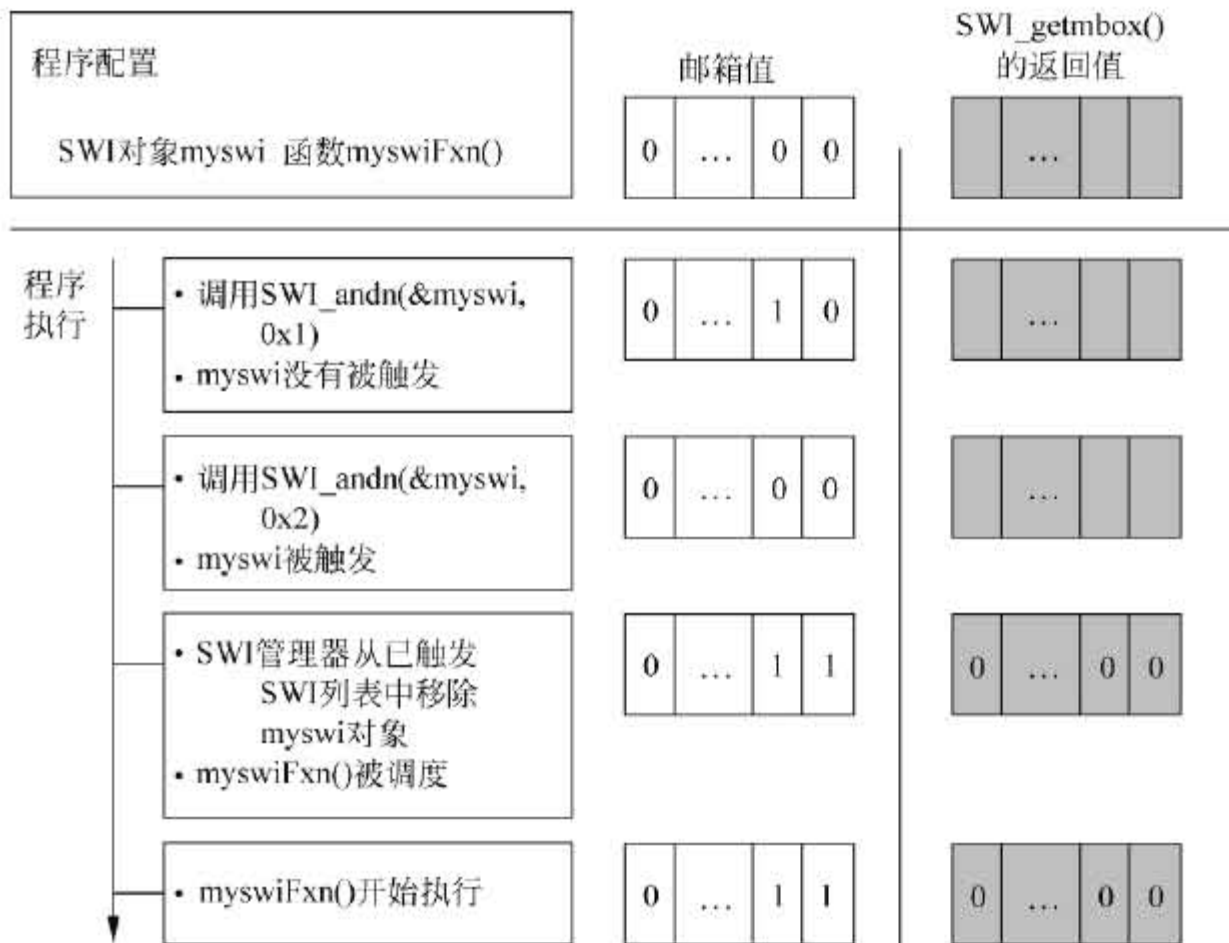
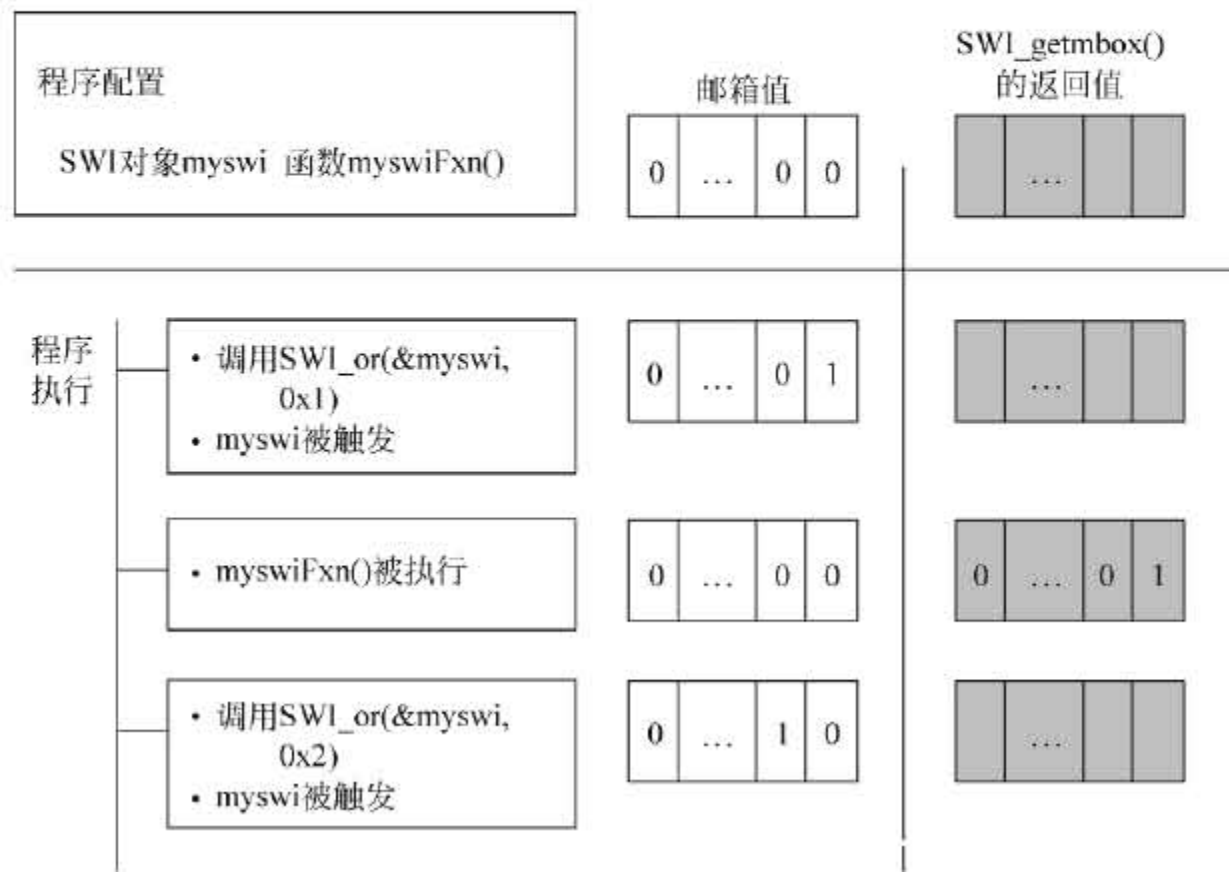


图 4-7 使用 SWI\_inc 触发一个 SWI



在某些情况下,SWI 函数需要根据触发事件的不同来调用不同的函数。这时程序可以在一个事件发生时使用 SWI\_or 触发该 SWI 对象,如图 4-9 所示。由 SWI\_or 使用的位掩模值来对触发事件的类型进行编码,SWI 函数可以将该位掩模值作为标志来识别事件并选择不同的函数来执行。





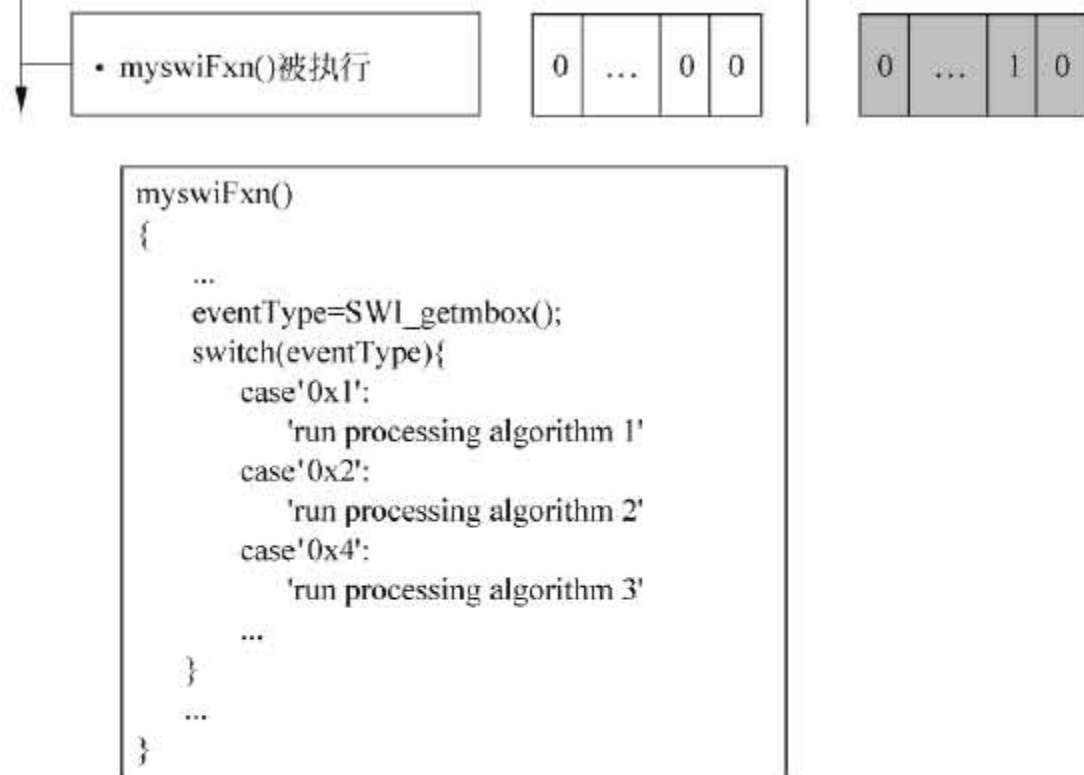


图 4-9 使用 SWI\_or 触发一个 SWI

如果应用程序需要同一个事件必须发生多次之后才触发一个 SWI,可以使用 SWI\_dec 来实现,如图 4-10 所示。只要将 SWI 邮箱的值配置为该事件发生的次数,然后在每次事件发生时调用 SWI\_dec,仅当邮箱值减小到 0 时,也即事件发生了与邮箱值相同的次数后,该 SWI 才会被触发。

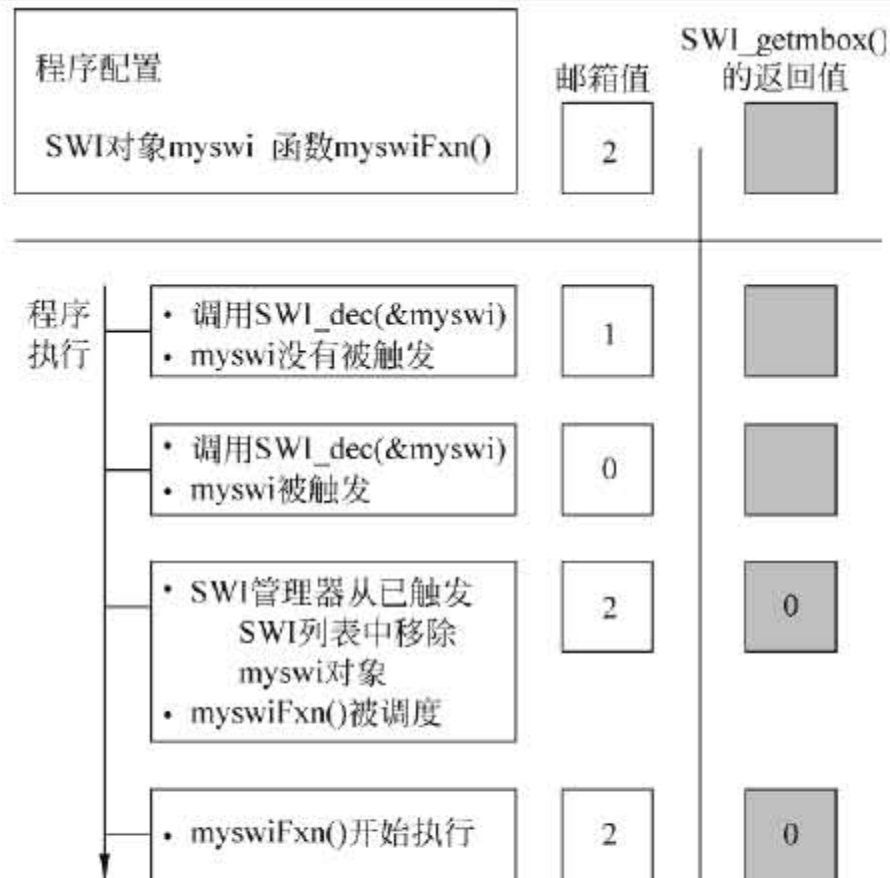


图 4-10 使用 SWI\_dec 触发一个 SWI

### 4.3.6 使用 SWI 的优缺点

使用软件中断而非硬件中断主要有以下两点好处：

首先,SWI可以在所有硬件中断都被使能时执行,不会影响到硬件中断对事件的实时响应。例如,当一个 HWI 和一个任务想要以互斥方式访问同一个数据结构时,任务 TSK 需要在存取数据结构之前关闭硬件中断。很明显,禁止硬件中断会存在延长中断响应时间从而降低系统实时性能的隐患。相反地,如果由 SWI 和 TSK 互斥访问此数据结构,TSK 访问该数据结构时只需禁止软件中断,这不会影响到系统使用硬件中断响应实时事件的能力。

将较长的 ISR 分割为两部分是很有用的: HWI 用来处理时限要求非常苛刻的操作,然后将时限要求相对不苛刻的后续处理交给 SWI 完成。

其次,SWI 可以调用一些 HWI 不能调用的函数,这是因为 SWI 能够保证不在 DSP/BIOS 更新内部数据结构时运行。这是 DSP/BIOS 一个非常重要的特点,用户应该熟知哪些函数可以在哪种线程中调用,这些信息可参考相应平台的 TMS320 DSP/BIOS API 参考手册。





注意: SWI 函数中不能调用任何会导致阻塞的 DSP/BIOS 函数。例如,SEM\_pend 可能引起阻塞,所以在 SWI 函数内部不能调用 SEM\_pend 函数,而且在 SWI 函数内部也不能有调用 SEM\_pend 函数的函数(如 MEM\_alloc 和 TSK\_sleep)。

此外,一个 SWI 函数必须在任何被阻塞的任务运行之前执行完毕。即使会引入额外的开销,有些情况下使用任务线程更适合整个系统的设计。

## 4.3.7 软件中断抢占时的寄存器保存

当一个软件中断抢占另一个线程时,DSP/BIOS 会自动在系统堆栈中保存表 4-4 所示的所有寄存器,以保护现场环境。

表 4-4 软件中断过程中保存的 CPU 寄存器

C54x 平台 			C55x 平台 			C6000 平台 		C28x 平台 	
ag	ar5	pmst	ac0	real	t0	a0-a9	b16- b31 (仅 C64x 有)	al	xt
ah	ar6	rea	ac1	rptc	t1	a16-a31 (仅 C64x 有)		ah	ph
al	ar7	rsa	ac2	rsa0	trn1			xar0	pl
ar0	bg	sp	ac3	rsa1	xar1			xar4	dp
ar1	bh	st0	brc1	st0	xar2	b0-b9	CSR	xar5	
ar2	bk	st1	brs1	st1	xar3		AMR	xar6	
ar3	bl	t	csr	st2	xar4			xar7	
ar4	brc	trn	rea0	st3					

不管是用 C 语言还是汇编语言编写的用户 SWI 函数都没有必要保存任何寄存器。由于未来的 DSP/BIOS 实现这一自动环境保护时可能不会保存“save on entry”寄存器,因此在使用汇编语言编写 SWI 函数时,最安全的做法还是按照寄存器约定将这些“save on

entry”寄存器保存起来。这些“save on entry”寄存器有 C54x 平台的 ar1、ar6 和 ar7，C6000 平台的 a10 到 a15 以及 b10 到 b15（有关 C 寄存器约定的细节，请查阅相应平台的优化编译器用户手册 Optimizing Compiler User's Guide）。



如果一个 SWI 函数修改了 IER 寄存器，应该自行将其保存起来并在返回时恢复。否则，对 IER 的修改就会持续下去，其他开始运行的线程或该 SWI 返回后恢复执行的线程就会继承这种改变。

在 HWI 函数中抢占环境不会被自动保存，用户必须在一个 HWI 被触发时使用 HWI\_enter 和 HWI\_exit 宏或者 HWI 调度程序来保护中断环境。

### 4.3.8 禁止和恢复 SWI

在空闲循环函数、任务或软件中断函数内部，用户可通过调用 SWI\_disable 临时阻止被更高优先级的软件中断抢占。要重新使能 SWI 抢占，调用 SWI\_enable 即可。

SWI\_disable 会禁止掉所有的软件中断，SWI\_enable 则相应地使能所有软件中断，这两个调用必须成对使用。另外，一个软件中断不能使能或禁止其自身。

当 DSP/BIOS 完成初始化并在第一个任务被调用之前，软件中断是被允许的。如果应

用程序希望禁止软件中断,可以这样调用 SWI\_disable:

```
key = SWI_disable();
```

对应的使能函数为:

```
SWI_enable(key);
```

变量 key 由 SWI 模块用来判断 SWI\_disable 是否被多次调用。这样可以实现 SWI\_disable /SWI\_enable 的嵌套调用,这样只有最外层的 SWI\_enable 调用才真正地使能软件中断。换句话说,一个任务在禁止或允许软件中断时,不用去判断 SWI\_disable 是否已在其他地方被调用。

当软件中断被禁止时,被触发的软件中断不会立刻运行,而是以软件方式被“锁存”起来,并在软件中断被重新使能且该软件中断是等待运行的最高优先级的线程才会被运行。

注意: SWI\_disable 的一个重要副作用是任务抢占也被禁止了,这是因为 DSP/BIOS 是使用软件中断来管理信号灯和时基的。

## 4.4 任 务

DSP/BIOS 任务对象是由 TSK 模块进行管理的线程,其优先级高于空闲循环但低于硬

件中断和软件中断。

TSK 模块根据任务的优先等级和任务的当前执行状态动态地进行调度和抢占,保证将处理器交给处于就绪状态且优先级最高的线程。任务线程共有 15 个可用的优先级,最低优先级(0)被预留来运行空闲循环。

TSK 模块提供了一组函数来操作 TSK 对象,这些函数通过 TSK\_Handle 类型的句柄来访问 TSK 对象。

DSP/BIOS 内核为每个任务对象都维护着一份处理器寄存器的拷贝,每个任务对象都有自己的运行时堆栈,用于保存局部变量以及函数的嵌套调用环境。

每个 TSK 对象的堆栈大小可分别指定。每个堆栈必须足够大,来处理普通子程序调用以及能为单次任务抢占保护现场环境。如果任务发生阻塞,只有那些 C 函数必须保存的寄存器才会被保存到任务堆栈中。为了配置准确的堆栈大小,可以先把堆栈设置的较大,然后使用 CCS 软件来记录堆栈的实际使用情况。

同一个应用程序里执行的所有任务线程共享一套相同的全局变量,并根据为 C 函数定义的标准规则进行存取访问。

## 4.4.1 创建任务对象

用户可以调用 TSK\_create 动态地或在配置中静态地创建 TSK 对象。动态创建的软件也可以在程序执行过程中动态地被删除。

#### 4.4.1.1 动态创建和删除任务

用户可以通过调用 `TSK_create` 函数来生成一个 DSP/BIOS 任务,该函数的参数包含一个 C 函数地址(即新建任务的起始执行地址),该函数的返回值是一个 `TSK_Handle` 类型的句柄,可传递给其他 TSK 函数作为参数。

TSK\_create 函数接口如下:

```
TSK_Handle TSK_create(fxn, attrs, [arg],...)  
    Fxn fxn;  
    TSK_Attrs * attrs  
    Arg arg
```

只有当任务被创建生成并且因为其优先级更高而抢占了当前运行的线程时,该任务才进入运行激活的状态。

调用 `TSK_delete` 将动态删除一个 TSK 对象,并且回收此对象占用的存储器和堆栈空间。`TSK_delete` 会从所有内部队列中移除该任务,并调用 `MEM_free` 来释放任务对象及其堆栈。

但任务占用的任何信号灯、邮箱和其他资源都不会被释放。因此动态删除一个占用这些资源的任务可能引起错误,如果这些资源不被其他线程使用的话,应该在删除任务之前先释放这些资源。

TSK\_delete 函数接口如下:



注意：当一个任务和系统里的其他任务共享某些资源时，用户动态删除正占用这些资源的任务对象可能引起灾难性错误。有关 TSK\_delete 函数的详细介绍请查阅相应平台的 TMS320 DSP/BIOS API 函数参考手册。

#### 4.4.1.2 静态创建任务

用户也可以静态地创建任务，如使用配置工具。DSP/BIOS 配置工具允许用户为每个任务以及为 TSK 管理器自身设置一系列属性值。在运行时，静态创建的任务与动态创建的任务的行为是完全一致的，但用户不能用 TSK\_delete 删除静态创建的任务。有关静态创建对象的优点，参见 2.3 节“动态创建 DSP/BIOS 对象”。

默认的 DSP/BIOS 配置模板中定义了 TSK\_idle 任务，此任务的优先级最低，当没有更高优先级的任务或中断就绪时，该任务会调用执行为 IDL 对象定义的函数，即用来管理后台线程。

#### 4.4.1.3 在配置工具中设置任务属性

在 TSK 管理器的属性对话框中可以查看 TSK 对象的一些默认属性，包括任务优先级、堆栈大小和堆栈所处的存储器段。每次插入一个新的 TSK 对象时，其属性都会被设置为这

些默认值,用户也可以为每个 TSK 对象单独修改这些属性。对所有 TSK 属性的完整描述请查阅相应平台的 TMS320 DSP/BIOS API 参考手册的 TSK 模块部分。

改变一个任务的优先级的方法如下:

(1) 在配置工具中选中 TSK 管理器,窗口右侧会出现一个窗格,其中按优先级顺序排列出所有的 TSK 对象,如图 4-11 所示。

(2) 要改变一个任务对象的优先级,只需用鼠标将该任务对象拖放到相应的优先级文件夹中即可。

(3) 用户也可以在任务对象的属性对话框中改变其优先级,右键单击任务对象并在弹出菜单中选择属性命令即可打开该对话框。

具有相同优先级的任务对象,任务调度器会根据它们在配置工具里列出的顺序进行调用。任务有 16 个优先级,最低为 0,最高为 15。且优先级 0 保留给系统空闲任务。

如果用户希望一个任务在初始时被挂起(阻塞),将其拖放到优先级为 -1 的文件夹中即可。这类任务直到其优先级在运行时得到提升才会被调度运行。

任务的属性窗口显示了其优先级数字(从 -1 到 15,最高优先级为 15),在 Priority 下拉列表里选择相应的数字即可设置任务的优先级,如图 4-12 所示。

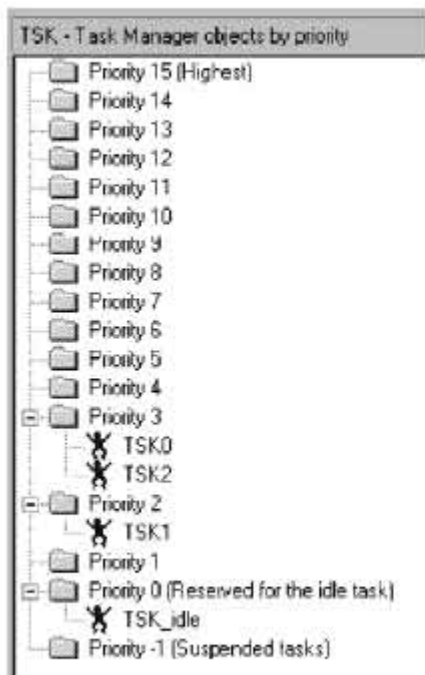


图 4-11 任务优先级列表



图 4-12 TSK 对象属性对话框

## 4.4.2 任务的执行状态和调度

每个 TSK 对象总是处于如下四种可能的执行状态之一：

- (1) 运行态(**Running**)：代表该任务当前正在系统处理器上执行。
- (2) 就绪态(**Ready**)：代表任务已经被调度并在等待处理器可用。
- (3) 阻塞态(**Blocked**)：代表任务必须等到某个事件发生或某些资源可用时才能执行。
- (4) 终止态(**Terminated**)：代表任务已经被终止，不会再执行。

任务根据其优先级顺序被调度执行,同一时刻只能有一个任务在执行,并且所有处于就绪态的任务优先级都小于等于当前运行任务的优先级,因为一旦优先级更高的任务进入就绪状态,就会立刻抢占当前任务,这与许多基于时间片分配策略的操作系统不同。

最高的任务优先级是 TSK\_MAXPRI(15),最低的任务优先级是 TSK\_MINPRI(1)。如果任务优先级小于 0,那么会被禁止执行直到其优先级在以后被其他任务提升。如果任务的优先级为 TSK\_MAXPRI,除非有硬件或软件中断发生,否则该任务将独占 CPU。

在程序执行过程中,每个任务的状态会由于一些原因而改变。图 4-13 说明了任务的状态是如何改变的。



图 4-13 任务状态变化

TSK、SEM 和 SIO 模块的函数可以改变任务的状态：阻塞或终止当前运行的任务，使之前被挂起(阻塞)的任务就绪，以及重新调度当前线程等。

任何时候只有一个任务处于运行态。如果所有的任务都被阻塞，并且没有硬件中断或软件中断运行，TSK 管理器将执行优先级为 0 的 TSK\_idle 任务。如果一个任务被硬件中断或软件中断抢占，由于当抢占结束时该任务将继续运行，因此调用 TSK\_stat 返回的该任务的状态仍然是 TSK\_RUNNING。

注意：不要在 IDL 函数中调用任何会引起阻塞的函数，如 SEM\_pend 或 TSK\_sleep，这样会阻止 DSP/BIOS 分析工具收集运行时信息(因为 DSP/BIOS 是在 IDL 函数中收集信息的，而 IDL 函数被阻塞的话就不会执行任何操作)。

如图 4-13 所示：当 TSK\_RUNNING 状态的任务转换到任何其他三种状态时，处理器的控制权会交给当前处于就绪态(TSK\_READY)的优先级最高的任务。一个 TSK\_RUNNING 状态的任务可以按下面三种方式之一转换到其他状态：

- ❑ 运行态的任务调用 TSK\_exit 转入终止态 TSK\_TERMINATED。当一个任务从其顶层函数返回时会自动调用 TSK\_exit。当所有的任务都返回后，TSK 管理器通过以状态码 0 调用 SYS\_exit 来终止程序执行。
- ❑ 运行态的任务调用了引起其阻塞的函数(如 SEM\_pend 或 TSK\_sleep)而转入阻塞态 TSK\_BLOCKED。当一个任务正在执行某种 I/O 操作，等待某些资源有效或资

源空闲时都有可能转入阻塞态。

- ❑ 当具有更高优先级的任务进入就绪状态时,当前运行的任务被抢占,并转入就绪态 TSK\_READY。如果当前运行的任务通过调用 TSK\_setpri 函数使得本身的优先级不再是系统里的最高优先级时,便会引起这种状态的切换。一个任务也可以调用 TSK\_yield 将控制权交给优先级相同的其他任务,自己变为就绪态。

一个阻塞态的任务在响应了它所等待的事件(如 I/O 操作的完成,共享资源的有效,一个指定时间段的消逝等等)之后就转入就绪态。一旦进入就绪状态,如果其优先级高于当前运行任务,就会立刻转入运行态,否则该任务被放入相同优先级的先入先出队列中将等待调度。

### 4.4.3 检测堆栈溢出

当一个任务使用的存储空间大于为其分配的堆栈时,可能改写其他任务或数据占用的存储区,引起不可意料的严重错误,所以提供一种堆栈溢出的检测方法是很有用的。

TSK\_checkstack 和 TSK\_stat 这两个函数用于观察堆栈大小。函数 TSK\_stat 返回的数据结构体包含了堆栈的大小以及堆栈曾被使用到的最大值(均以 MADU 为单位)。下面这个代码段给出了这个函数的用法,当检测到堆栈使用量超过 90%时,用 LOG\_printf 给出警告。有关这两个函数的详细描述请查阅相应平台的 TMS320 DSP/BIOS API 函数参考手册。

```
TSK_Stat statbuf;           /* declare buffer */
```

```
TSK_stat(TSK_self(), &statbuf);    /* call func to get status */  
if (statbuf.used > (statbuf.attrs.stacksize * 9 / 10))  
{  
    LOG_printf(&trace, "Over 90 % of task's stack is in use.\n")  
}
```

#### 4.4.4 任务钩子

应用程序可能需要在各种有关任务的事件(如任务创建、删除、退出、就绪等)产生时触发执行一些用户自定义的函数,这些函数被称为钩子(Hook)函数。钩子函数可在程序初始化、任务创建(TSK\_create)、任务删除(TSK\_delete)、任务退出(TSK\_exit)、任务就绪和任务环境切换(TSK\_sleep, SEM\_pend)时被自动调用。这些钩子函数也可用于实现额外环境保护。

用户可以为 TSK 模块管理器指定一组钩子函数。要创建额外的钩子函数组,可以使用 HOOK 模块完成。例如,一个集成了第三方软件的应用程序除了执行自己的钩子函数以外,还需要执行第三方软件所需的钩子函数。另外,每个 HOOK 对象都为每个任务维护一个私有的数据环境。

当用户在配置第一个 HOOK 对象时,之前由用户为 TSK 模块配置的所有钩子函数都会自动地被放入一个名为 HOOK\_KNL 的 HOOK 对象中。用户可以在 HOOK\_KNL 对象



的属性域中设置其初始化函数,而其他属性参数则需要在 TSK 模块中设置。如果用户只为 TSK 模块管理器配置了一组钩子函数,则 HOOK 模块不会被使用。

用 C 语言编写的函数在配置工具里必须以一个下划线为开头,后面是 C 函数名。对于钩子函数的详细信息请参考相应平台的 TMS320 DSP/BIOS API 用户手册里的 TSK 模块和 HOOK 模块部分。

#### 4.4.5 用于额外环境保护的任务钩子

一个系统可能需要为每个任务保存一些特殊的硬件寄存器(如扩展寻址寄存器),那么就需要钩子函数来实现这些功能。在例 4-7 中函数 doCreate 用于给每个任务分配缓冲区来保存这些寄存器,函数 doDelete 用于释放这些缓冲空间,而函数 doSwitch 则用于保存和恢复这些寄存器。例 4-7 假设任务对象是动态创建的。

**例 4-7** 任务扩展环境保护

```
#define CONTEXTSIZE    'size of additional context'

Void doCreate(TSK_Handle task)
{
    Ptr    context;
    context = MEM_alloc(0, CONTEXTSIZE, 0);
    TSK_setenv(task, context);    /* set task environment */
}
```

```
Void doDelete(TSK_Handle task)
```

```
{
```

```
    Ptr    context;
```

```
    context = TSK_getenv(task); /* get register buffer */
```

```
    MEM_free(0, context, CONTEXTSIZE);
```

```
}
```

```
Void doSwitch(TSK_Handle from, TSK_Handle to)
```

```
{
```

```
    Ptr    context;
```

```
    static Int first = TRUE;
```

```
    if (first) {
```

```
        first = FALSE;
```

```
        return;
```

```
    }
```

```
    context = TSK_getenv(from); /* get register buffer */
```

```
    *context = 'hardware registers'; /* save registers */
```

```
    context = TSK_getenv(to); /* get register buffer */
```

```
    'hardware registers' = *context; /* restore registers */
```

```
}
```

```
Void doExit(Void)
```

```
{
```

```
    TSK_Handle  usrHandle;
```

```
    /* get task handle, if needed */
```

```
    usrHandle = TSK_self();
```

```
    'perform user-defined exit steps'
```

```
}
```

注意：对于 C55x 和 C28x 平台，非指针类型变量在被 LOG\_printf 引用时，需要将其类型转化为 Arg，例如：

```
LOG_printf(& trace, "Task %d Done", (Arg) id);
```

## 4.4.6 任务让出与时间片调度

例 4-8 给出了一个由用户管理的时间片调度的模型,该模型是可抢占的且不需要任何来自任务的协作。虽然任何一个应用程序中的 DSP/BIOS 任务都可以具有不同优先级,但时间片模型只适用于优先级相同的任务。

在这个例子中,PRD 对象 prd0 被配置为每隔 1ms 调用一次 TSK\_yield() 函数。PRD 对象 prd1 被配置为每隔 16ms 运行一次 SEM\_post(&sem) 函数。图 4-14 给出了这个例子的追踪结果,图 4-15 则是该例的执行图。

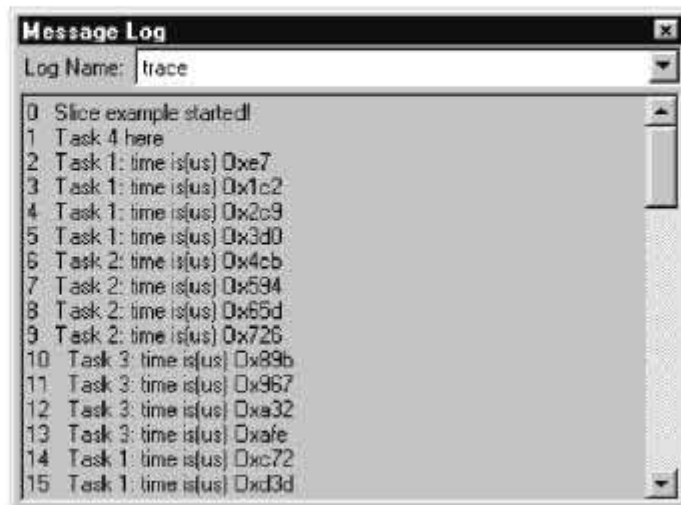


图 4-14 例 4-8 的日志追踪结果窗口

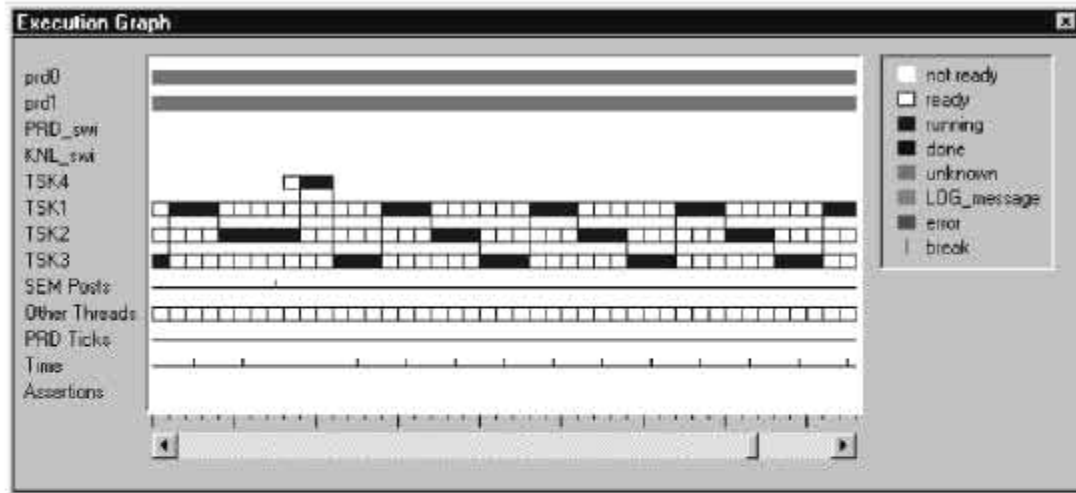


图 4-15 例 4-8 的执行图

#### 例 4-8 任务的时间片调度

/\*

\* ===== slice.c =====

\* This example utilizes time-slice scheduling among three tasks of

```
* equal priority. A fourth task of higher priority periodically
* preempts execution.
*
* A PRD object drives the time-slice scheduling. Every
* millisecond, the PRD object calls TSK_yield() which forces the
* current task to relinquish access to the CPU. The time
* slicing could also be driven by a CLK object (as long as the
* time slice was the same interval as the clock interrupt), or by
* another hardware interrupt.
*
* The time-slice scheduling is best viewed in the Execution Graph
* with SWI logging and PRD logging turned off.
*
* Because a task is always ready to run, this program does not
* spend time in the idle loop. Calls to IDL_run() are added to
* force the update of the Real-Time Analysis tools. Calls to
* IDL_run() are within a TSK_disable(), TSK_enable() block because
* the call to IDL_run() is not reentrant.
* /
```

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

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

```

#include <idl.h>
#include <log.h>
#include <sem.h>
#include <swi.h>
#include <tsk.h>
#include "slicecfg.h"

Void task(Arg id_arg);
Void hi_pri_task(Arg id_arg);
Uns counts_per_us;    /* hardware timer counts per microsecond */

/* ===== main ===== */
Void main()
{
    LOG_printf(&trace, "Slice example started!");
    counts_per_us = CLK_countspms() / 1000;
}

/* ===== task ===== */
Void task(Arg id_arg)
{
    Int id = ArgToInt(id_arg);

```

```

LgUns time;
LgUns prevtime;
/*
 * The while loop below simulates the work load of
 * the time sharing tasks
 */
while (1) {
    time = CLK_gettime() / counts_per_us;
    /* print time only every 200 usec */
    if (time >= prevtime + 200) {
        prevtime = time;
        LOG_printf(&trace, "Task %d: time is(us) 0x%x", id, (Int)time);
    }
    /* check for rollover */
    if (prevtime > time) {
        prevtime = time;
    }
    /*
     * pass through idle loop to pump data to the Real-Time
     * Analysis tools
     */
    TSK_disable();

```



```

        IDL_run();
        TSK_enable();
    }
}

/* ===== hi_pri_task ===== */
Void hi_pri_task(Arg id_arg)
{
    Int id = ArgToInt(id_arg);
    while (1) {
        LOG_printf(&trace, "Task %d here", id);
        SEM_pend(&sem, SYS_FOREVER);
    }
}

```

## 4.5 空闲循环

空闲循环是 DSP/BIOS 的后台线程，只在没有硬件中断、软件中断和任务运行的时候才会循环运行。其他的任何线程都可以在任何时候抢占空闲循环。

IDL 管理器允许用户配置自定义的 IDL 函数,在进入空闲循环时这些 IDL 函数会被执行。IDL\_loop 会依照 IDL 对象的创建顺序调用与每个对象相关联的函数,每次运行一个且当最后一个对象函数执行完之后 IDL\_loop 重新调用第一个 IDL 函数不断循环。空闲循环经常用来轮询不产生中断的非实时设备、监测系统状态或执行其他后台操作。

空闲循环是 DSP/BIOS 中优先级最低的线程。目标 DSP 和主机 DSP/BIOS 分析工具间的通信通常是在空闲循环中执行的。这保证了 DSP/BIOS 分析工具不会影响应用程序的处理。如果目标 DSP 的 CPU 十分繁忙而无法执行后台操作,那么主机分析工具就会停止从目标 DSP 接受信息,直到 CPU 空闲。

默认情况下,空闲循环执行下列 IDL 对象的函数:

- **LNK\_dataPump** 用来管理目标 DSP 处理器和主机之间的实时分析数据(例如 LOG 和 STS 数据)和 HST 数据的传输。这通过使用 RTDX 来处理。

在 C55x 和 C6000 平台上,主机 PC 通过触发一个硬件中断来和目标 DSP 交换数据,优先级高于 SWI、TSK 和 IDL 函数。因为实际的 HWI 函数仅运行很短的时间,因此 LNK\_dataPump 函数负责在空闲循环内完成准备 RTDX 缓冲区以及执行调用等耗时的的工作,只有实际的数据传输工作在高优先级 HWI 线程中执行。这种数据传输方式对应用程序的实时行为产生的影响较小,特别是 LOG 数据很多的时候。

而在 C54x 平台上 IDL 对象 RTDX\_dataPump 会调用 RTDX\_Poll 在目标 DSP 和主机之间传输数据,所以数据传输是在最低优先级的空闲循环中进行。

- **RTA\_dispatcher** 用于在目标 DSP 端接收主机实时分析工具的命令,收集目标 DSP 的监测信息并实时上传给主机 PC。RTA\_dispatcher 处在两个专用主机通道的末

端,它通过 LNK\_dataPump 传输命令和数据。

- **IDL\_cpuload** 使用一个 STS 对象 IDL\_busyObj 来计算目标 DSP 负荷。该对象包含的内容会通过 RTA\_dispatcher 对象上传给 DSP/BIOS 分析工具来显示 CPU 负荷。
- **RTDX\_datapump** 在 C54x 平台上调用 RTDX\_Poll 函数在空闲时间内传输目标 DSP 和主机 PC 之间的数据。而对于 C55x 和 C6000 平台,RTDX 是一个由中断驱动的接口(见上面 LNK\_dataPump 对象的说明),没有 RTDX\_datapump 对象。
- **PWRM\_idleDomains** 在 DSP/BIOS 空闲循环中调用一个函数来闲置各种 DSP 时钟域。用户可以在 PWRM 模块的配置里选择要闲置的时钟域。当一个 HWI、SWI 或 TSK 线程就绪时,被闲置的时钟域会重新恢复到原来的状态。

## 4.6 功率管理



DSP/BIOS 功率管理模块(PWRM)允许用户降低其应用程序的功耗,当前阶段该模块在 C5510 DSP 上可用,并对其他 C55x 系列器件提供部分支持。用户可以查看 DSP/BIOS

版本说明来了解该模块对不同的器件都提供哪些模块特性的支持。

PWRM 模块的功能包括：

- ❑ 闲置时钟域：用户通过闲置指定的时钟域来降低功耗。
- ❑ 引导时节省功耗：用户可以指定一个节能函数，该函数在器件引导时自动运行，可以随意闲置一些耗费功率的外设。
- ❑ 电压和频率尺度调整：用户可以动态地改变 CPU 的工作电压和频率，称为 V/F 尺度调整。由于功率使用与频率成线性正比，与电压的平方成正比，所以使用动态电压和频率尺度调整可以显著地降低功耗。
- ❑ 使用睡眠模式：用户可以设置自定义的睡眠模式来在非活动阶段内节省功耗。睡眠模式可静态配置，也可以在运行时设置。
- ❑ 睡眠模式与尺度调整的协调：用户可以使用由 PWRM 模块提供的注册和通知机制来协调睡眠与 V/F 尺度调整。

注意：PWRM 模块并不保证应用程序能够满足其调度时序安排要求。开发者必须自行考虑应用程序行为以满足这种要求。

## 4.6.1 闲置时钟域

TI DSP 具有一个“IDLE”指令用来关闭 DSP/BIOS 时钟以降低功耗，这是在运行时降低功耗的最主要的机制。在 C55x 平台，时钟被划分为以下几个时钟域：CPU、CACHE、

DMA、EMIF、PERIPH 和 CLKGEN。通过在闲置配置寄存器(ICR)中置位相应的比特位,然后执行 IDLE 指令,即可将这些时钟域闲置起来。

当闲置时钟域时,需要注意避免对应用程序调度时序产生不利影响。例如,一个线程想要闲置 CPU 直到获取足够的外部数据,那么直到下一次中断发生时,其他优先级相同或更低的线程才能被运行,即该线程无意地阻塞了其他任务的调度。为了避免这种情形的发生,DSP 的 CPU 时钟域只应该在 DSP/BIOS 空闲循环中被闲置。

为了方便用户,PWRM 模块使得用户可以在 DSP/BIOS 空闲循环中自动地闲置被选中的时钟域。用户可以在 PWRM 模块属性对话框的 Idling 标签页里静态地配置需要闲置的时钟域,也可以使用 PWRM\_configure 函数来动态更改配置。

当 PWRM 被配置为在 IDL 循环中闲置时钟域,那么 IDL 循环的处理就不会像之前那样有规律地运行。例如,当实时分析被使能时,空闲循环会运行函数来计算 CPU 负荷,从 DSP 获取实时分析数据,以及输送数据到 CCS。当 PWRM 闲置被使能时,PWRM\_F\_idleDomains 函数会被添加到空闲循环函数列表中。如果 PWRM 闲置了 CPU 域,那么每次该函数运行后,CPU 域将被挂起直到下一个中断产生。因此,CCS 中实时分析数据的动态更新将被延迟而变得不连续。

在空闲循环中闲置时钟域是为了在脱离了 CCS 的最终应用系统中使用的。

函数 PWRM\_idleClocks 用于直接地且无限期地闲置时钟域。比如一个完全在片内存储器上运行的应用程序,可以调用 PWRM\_idleClocks 来闲置掉 EMIF 时钟域。

## 4.6.2 引导时节省功耗

DSP 器件通常会以最高时钟速率和最高功耗来引导。然而总有一些功耗源是在初始化时不需要的,或者应用程序从来未曾使用的。

PWRM 模块提供了一种钩子机制,使得用户可以指定一个在引导时调用的函数来关闭或闲置功耗源,直到真正需要使用的时候才将其打开。例如应用程序可以稍晚一点再打开一个设备驱动来为底层物理设备提供动力。

在引导函数中,用户可以进行时钟域闲置和控制外部设备运行于低功耗模式等操作。虽然这些功能可以直接在主函数中实现,但是引导钩子机制允许用户将和功率有关的代码和功率管理器联系得更紧密一些。

## 4.6.3 电压和频率尺度调整

一个基于 CMOS 的 DSP 器件,其功耗与时钟速率(频率)成正比,与工作电压的平方成正比,而工作电压又决定了可以达到的最大时钟速率。

因此如果一个应用程序可以降低 CPU 时钟速率而又能满足其处理时限,就可以按线性正比节省功耗。降低 CPU 时钟速率可能会按比例地增加程序的执行时间,因此用户应该仔细地对应用程序进行分析以确保降低时钟速率后仍然可以满足其实时性要求。

如果时钟频率可以被降低,且新的频率也与 DSP 能够支持的较低的工作电压相匹配,则通过降低电压就可根据平方正比关系得到额外的相当显著的功耗节省。

PWRM 模块允许应用程序调用 PWRM\_changeSetpoint 来改变工作电压和频率(V/F)。所以,当应用程序切换到一个处理时限要求降低的模式时,就可以逐步降低电压和频率来减少功耗。或者那些从外部获取数据然后再处理的应用程序,可以在“闲散”时间(数据未准备好时)内通过降低 V/F 来在低功耗状态下运行。

通过使用 PWRM\_getCurrentSetpoint、PWRM\_getNumSetpoints、PWRM\_getSetpointInfo 和 PWRM\_getTransitionLatency 函数,用户可以了解到 V/F 尺度调整的其他特性。

PWRM 模块还支持 V/F 尺度调整在整个应用中的协调实施,这是通过注册和通知机制实现的。当客户程序在 PWRM 中注册以求在 V/F 尺度调整事件发生时被通知时,这些客户程序应指出其支持的 V/F 设置点(setpoints)。例如一个设备驱动不能工作在某频率以下,其客户程序就可以在 PWRM 模块中注册时指出这一点,这样只要该客户一直保持被注册的状态就不会转换到更低的工作频率。

PWRM 模块使用一个与平台相关的功率尺度调整库(power scaling library,PSL)来实现 V/F 尺度调整。该库只能适用于某些平台。有关该库的详细信息请查阅 TI 应用笔记 *Using the Power Scaling Library on the TMS320C5510*(SPRA848)。



#### 4.6.3.1 V/F 调整对 DSP/ BIOS CLK 模块的影响

在 C5510 DSP 上,被 V/F 尺度调整影响的时钟(CPU 时钟)同样驱动着定时器,而 DSP/BIOS 正是使用定时器来提供时钟服务的(CLK 模块)。这意味着改变 V/F 设置点会使 DSP/BIOS 时钟服务陷入混乱。为了将这种影响降至最小,PWRM 模块允许 DSP/BIOS CLK 模块进行注册以得到 V/F 尺度调整通知。当 CLK 模块被通知转换到一个新的 V/F 设置点时,就会重新对定时器编程,保持与尺度调整之前相同的计时频率。

所以低分辨率时间(CLK\_gettime)在频率调整之后可以继续起作用,然而一小部分绝对时间会在定时器重新编程的过程中被丢失。时间的丢失是因为在 V/F 尺度调整发生之前的最后一步 DSP/BIOS 定时器会暂时停止计时,V/F 尺度调整操作一旦完成定时器就立刻保持以前的频率开始计时。所以在尺度调整操作过程中,时间对 DSP/BIOS 和应用程序来说都是“静止不动”的。

高分辨率时钟(CLK\_gettime)在与 V/F 尺度调整一起使用时应注意:

- ❑ 跨越设置点变换使用 CLK\_gettime 获取高分辨率时间增量会得到一个错误的结果。但在两次设置点变换之间,仍然可以使用 CLK\_gettime 获取时间增量。
- ❑ 定时器递增或递减的速率通常在不同的 V/F 设置点下有所不同。

#### 4.6.4 使用睡眠模式



PWRM 模块允许应用程序处于睡眠模式,即将 DSP 置为一种低功耗状态。睡眠模式的实现根据目标 DSP 平台的不同而不同,并且会涉及到闲置时钟、降低工作电压以及关闭子系统电源等机能。当前阶段,C5510 器件支持两种睡眠模式:深度睡眠和睡眠到重启。

- ❑ **深度睡眠(deep sleep)**: 允许 DSP 进入一种最小功耗状态同时等待一个外部中断。当该中断发生时,DSP 会快速而平稳地恢复被打断的处理。默认情况下,所有 C5510 的时钟域都会在深度睡眠中被闲置。但用户可以使用 PWRM 重新配置 C5510 器件在深度睡眠中闲置的时钟域。
- ❑ **睡眠到重启(sleep until restart)**: 则是一个更彻底的模式。这种模式会使 DSP 进入一种不会恢复的最小功耗状态,直到 DSP 被重新引导。

PWRM 模块还提供了一种协调机制来使睡眠状态在整个应用程序中被配合实施,这是通过注册和通知机制实现的。例如一个控制外部编解码器的设备驱动经过注册后,就会在 DSP 进入深度睡眠时被通知,所以该驱动就可以告诉外部设备也进入低功耗状态。当 DSP 从深度睡眠中恢复时,该驱动又得到通知,即可发送适当的指令来唤醒编解码器。

#### 4.6.5 睡眠及尺度调整的协调

PWRM 模块允许那些关心功率事件的代码进行注册,以使它们能够在特定的功率事件发生时得到通知。同样,代码也可以在不需要通知时进行注销。

客户程序可以调用 PWRM\_registerNotify 进行注册,以求在下列类型事件发生时得到通知:

- 功率事件:
  - V/F 设置点将要改变。
  - V/F 设置点已被改变。
- 睡眠事件:
  - DSP 将要进入深度睡眠。
  - DSP 已从深度睡眠中被唤醒。
  - DSP 将要进入睡眠到重启模式。

图 4-16 给出了客户程序注册以及得到功率调整事件通知的顺序。

被编号的步骤说明如下：

1. 应用程序代码进行注册,希望在 V/F 设置点变化时得到通知。例如不同的设置点可能需要不同的 EMIF 设置,所以应用程序需要将 EMIF 控制代码注册到 PWRM 模块中,以使其能在设置点变化时改变 EMIF 设置。

2. 一个使用 DMA 访问外部存储器的 DSP/BIOS 链接驱动进行注册,希望在 V/F 设置点变化时得到通知。例如在设置点变化之前,该驱动可能需要暂时停止 DMA 对外部存储器的操作。

3. 类似的, TI 或第三方的目标代码也进行了注册,希望在 V/F 设置点变化时得到通知。

4. 应用程序决定改变 V/F 设置点,调用 PWRM\_changeSetpoint 对设置点改变作初始化。例如一个设备工作模式的变化可能引起这个操作。

5. 在改变设置点之前, PWRM 会验证改变请求的有效性,然后通知所有已注册的客户程序即将改变设置点。客户程序会根据其注册顺序依次得到通知(先进先出)。

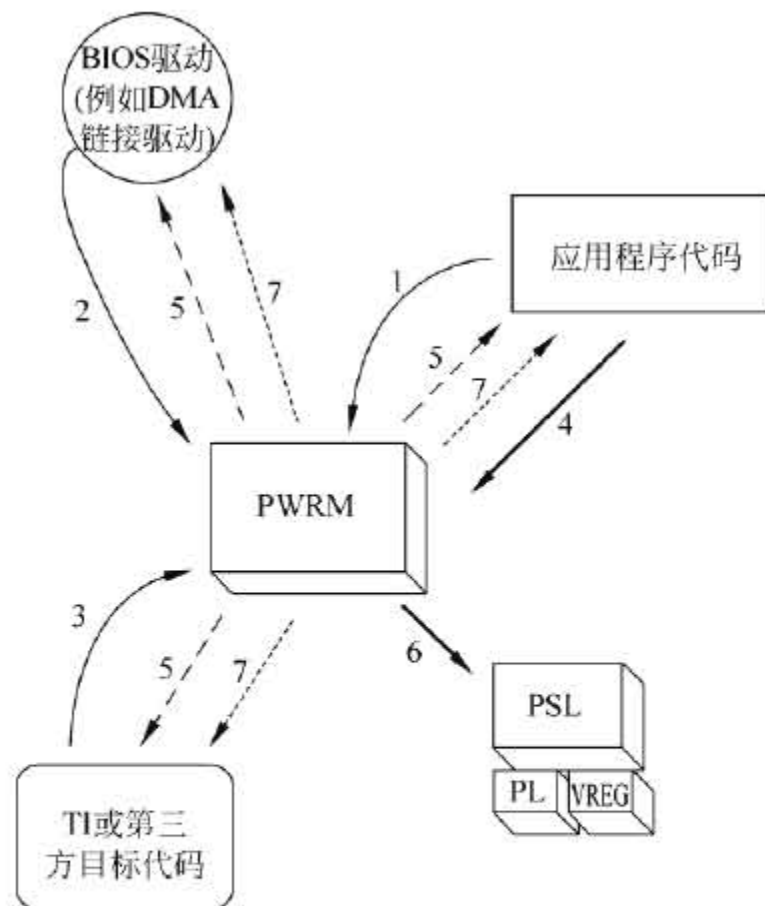


图 4-16 功率事件通知

6. PWRM 调用功率尺度调整库(PSL)来改变 V/F 设置点。

7. 在设置点改变后,PWRM 通知注册客户程序设置点已被改变。

如果一个客户程序的通知函数能立即完成其工作,应返回 PWRM\_NOTIFYDONE。如果该通知函数因为需要等待而不能立即行动,应返回 PWRM\_NOTIFYNOTDONE,稍后当该客户程序完成了其所需的操作后,应该调用由函数 PWRM\_registerNotify 为该客户程序指派的 delayedCompletionFunction 函数。PWRM 模块会等待所有客户返回 PWRM\_NOTIFYDONE 后再继续执行。如果有客户程序在指定的超时参数(timeout)值之内没有给出其已经完成的信号,PWRM 返回 PWRM\_ETIMEOUT 指示产生了系统故障。

在将功率事件通知给客户程序之前,PWRM 模块首先会禁止掉 SWI 和 TSK 调度以保证在处理功率事件时不会发生抢占。下表说明了在处理功率事件过程中什么时候禁止和重新使能 SWI 和 TSK 调度。

事件类型	禁止 SWI 和 TSK 调度	重新使能调度
V/F 尺度调整	在向客户程序发出即将改变设置点的通知之前	在完成设置点改变并向客户程序发出设置点改变已完成的通知之后
睡眠	在向客户程序发出即将进入某睡眠模式的通知之前	在唤醒并向客户程序发出已从深度睡眠模式中唤醒的通知之后

由于在 PWRM 执行通知时 SWI 和 TSK 调度被禁止,所以客户程序不能依靠 SWI 或 TSK 调度来完成 PWRM 事件处理。但客户程序可以利用 HWI 来完成功率事件处理并发出信号。例如,要让一个运行中的 DMA 操作完成,可以让该 DMA 的 ISR 照旧运行,并且在该 ISR 内调用 delayedCompletionFunction 函数向 PWRM 发出该客户程序已完成事件处理的信号。

## 4.7 信 号 灯

DSP/BIOS 提供了一组函数来实现任务间的基于信号灯(semaphores)的同步和通信。信号灯经常用于协助一组相互竞争的任务来访问共享资源。SEM 模块提供了一组操纵信号灯对象的函数,对信号灯对象的访问通过 SEM\_Handle 类型的句柄实现。

SEM 对象是一种计数信号灯,可以实现任务同步和互斥。计数信号灯对象有一个内部计数器,计数值对应有效的资源数。如果计数值大于 0,则任务在请求该信号灯时不会被阻塞。

函数 SEM\_create 和 SEM\_delete 分别用于动态地创建和删除信号灯对象,见例 4-9。用户也可以由配置静态地创建信号灯对象。

#### 例 4-9 函数 SEM\_create 和 SEM\_delete

```
SEM_Handle SEM_create(count, attrs);  
    Uns          count;  
    SEM_Attrs   * attrs;  
Void SEM_delete(sem);  
    SEM_Handle sem;
```

信号灯被动态创建时,其计数值被初始化为参数 count 的值。通常,计数值被设置为某同步资源的数目。

函数 SEM\_pend 用于等待一个信号灯,函数接口见例 4-10。如果信号灯计数值大于 0,则 SEM\_pend 只是简单地将其计数值减 1 并返回。否则 SEM\_pend 等待 SEM\_post 来发布该信号灯。

#### 例 4-10 函数 SEM\_pend

```
Bool SEM_pend(sem, timeout);  
    SEM_Handle sem;  
    Uns      timeout;      /* return after this many system clock ticks */
```

注意：当在 HWI 函数内调用时，SEM\_post 或 SEM\_ipost 代码必须包含在汇编宏 HWI\_enter/HWI\_exit 之间，或者使用 HWI 调度程序。

SEM\_pend 函数的超时参数 timeout 允许任务一直等待直到超时，或无限等待（取值 SYS\_FOREVER），或不等待（取值 0）。SEM\_pend 的返回值代表请求信号灯是否成功。

函数 SEM\_post 用于发布信号灯，函数接口见例 4-11。如果有任务正在等待该信号灯，SEM\_post 会从等待队列中将该任务删除，并将其放入就绪任务队列等待调度。如果没有任务等待这个信号灯，SEM\_post 简单地将计数值加 1 并返回。

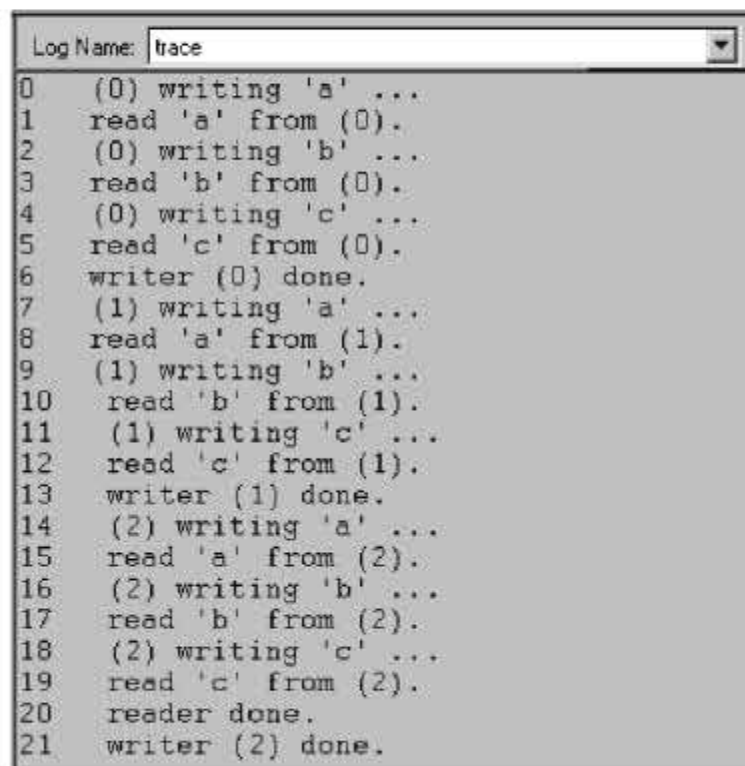
#### 例 4-11 函数 SEM\_post

```
Void SEM_post(sem);  
    SEM_Handle sem;
```

下面的例 4-12 是一个使用信号灯的例子：3 个写操作任务创建消息并放到 1 个队列中由 1 个读操作任务读取。写操作任务调用 SEM\_post 来指示有新的消息写入到队列中。读操作任务调用 SEM\_pend 来等待消息队列的消息。当有可读取的消息时，SEM\_pend 函数才会返回。读操作任务使用 LOG\_printf 函数将消息内容显示出来。

该例中的 3 个写操作任务、1 个读操作任务、信号灯和队列这些对象都是在配置工具中静态生成的。

由于该例涉及到多个任务,所以使用计数信号灯对访问队列的操作进行同步。图 4-17 显示了该实例的执行结果。因为读操作任务的优先级高于写操作任务的优先级,所以消息一旦进入队列就被读走。



```
Log Name: trace
0 (0) writing 'a' ...
1 read 'a' from (0).
2 (0) writing 'b' ...
3 read 'b' from (0).
4 (0) writing 'c' ...
5 read 'c' from (0).
6 writer (0) done.
7 (1) writing 'a' ...
8 read 'a' from (1).
9 (1) writing 'b' ...
10 read 'b' from (1).
11 (1) writing 'c' ...
12 read 'c' from (1).
13 writer (1) done.
14 (2) writing 'a' ...
15 read 'a' from (2).
16 (2) writing 'b' ...
17 read 'b' from (2).
18 (2) writing 'c' ...
19 read 'c' from (2).
20 reader done.
21 writer (2) done.
```

图 4-17 例 4-12 的日志追踪结果



例

```
/*  
 * ===== semtest.c =====  
 *  
 * Use a QUE queue and SEM semaphore to send messages from  
 * multiple writer() tasks to a single reader() task. The  
 * reader task, the three writer tasks, queues, and semaphore  
 * are created by the Configuration Tool.  
 *  
 * The MsgObj's are preallocated in main(), and put on the  
 * free queue. The writer tasks get free message structures  
 * from the free queue, write the message, and then put the  
 * message structure onto the message queue.  
 * This example builds on quetest.c. The major differences are:  
 * - one reader() and multiple writer() tasks.  
 * - SEM_pend() and SEM_post() are used to synchronize  
 * access to the message queue.  
 * - 'id' field was added to MsgObj to specify writer()  
 * task id.  
 *  
 * Unlike a mailbox, a queue can hold an arbitrary number of  
 * messages or elements. Each message must, however, be a  
 * structure with a QUE_Elem as its first field.
```

\* /

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

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

```
#include <mem.h>
```

```
#include <que.h>
```

```
#include <sem.h>
```

```
#include <sys.h>
```

```
#include <tsk.h>
```

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

```
#define NUMMSGs 3    /* number of messages */
```

```
#define NUMWRITERS 3 /* number of writer tasks created with */
```

```
/* Config Tool */
```

```
typedef struct MsgObj {
```

```
    QUE_Elem    elem; /* first field for QUE */
```

```
    Int          id;   /* writer task id */
```

```
    Char         val;  /* message value */
```

```
} MsgObj, *Msg;
```

```
Void reader();
```

```
Void writer();
```

```

/
* The following semaphore, queues, and log, are created by
* the Configuration Tool.
* /
extern SEM_Obj sem;
extern QUE_Obj msgQueue;
extern QUE_Obj freeQueue;
extern LOG_Obj trace;

/ *
* ===== main =====
* /
Void main()
{
    Int i;
    MsgObj * msg;
    Uns mask;

    mask = TRC_LOGTSK | TRC_LOGSWI | TRC_STSSWI | TRC_LOGCLK;
    TRC_enable(TRC_GBLHOST | TRC_GBLTARG | mask);

```

```

        SYS_abort("Memory allocation failed!\n");
    }

    /* Put all messages on freequeue */
    for ( i = 0; i < NUMMSGs; msg++, i++ ) {
        QUE_put(&freeQueue, msg);
    }
}

/*
 * ===== reader =====
 */
Void reader()
{
    Msg msg;
    Int i;

    for ( i = 0; i < NUMMSGs * NUMWRITERS; i++ ) {
        /*
         * Wait for semaphore to be posted by writer().
         */
    }
}

```

```

    /* dequeue message */
    msg = QUE_get(&msgQueue);

    /* print value */
    LOG_printf(&trace, "read '%c' from (%d).", msg->val, msg->id);

    /* free msg */
    QUE_put(&freeQueue, msg);
}
LOG_printf(&trace, "reader done.");
}

/*
 * ===== writer =====
 */
Void writer(Int id)
{
    Msg msg;
    Int i;

    for (i = 0; i < NUMMSGS; i++) {

```

```

/*
 * Get msg from the free queue. Since reader is higher
 * priority and only blocks on sem, this queue is
 * never empty.
 */
if (QUE_empty(&freeQueue)) {
    SYS_abort("Empty free queue!\n");
}
msg = QUE_get(&freeQueue);

/* fill in value */
msg->id = id;
msg->val = (i & 0xf) + 'a';
LOG_printf(&trace, "( %d) writing '%c' ...", id, msg->val);

/* enqueue message */
QUE_put(&msgQueue, msg);

/* post semaphore */
SEM_post(&sem);

```

```
    }  
    LOG_printf(&trace, "writer ( %d) done.", id);  
}
```

注意：对于 C55x 和 C28x 平台，非指针类型变量在被 LOG\_printf 引用时，需要将其类型转化为 Arg，例如：

```
LOG_printf(& trace, "Task %d Done", (Arg) id);
```

## 4.8 邮 箱

MBX 模块提供了一组函数来管理邮箱(mailboxes)。MBX 邮箱可以用来在任务间传递消息。由一个固定长度的共享邮箱来实现任务间的同步，可以保证消息流的输入不会超出系统处理这些消息的能力。本节里给出的实例就是一个很好的说明。

由 MBX 模块管理的邮箱对象和 SWI 对象里包含的邮箱数据结构体是不同的。

函数 MBX\_create 和 MBX\_delete 分别用于动态地创建和删除邮箱，用户也可以静态创建邮箱对象。有关静态创建对象的优点，参见 2.3 节，动态创建 DSP/BIOS 对象。用户创建邮箱时可指定邮箱的长度和单个消息的大小等属性，如例 4-13。

```

MBX_Handle MBX_create(msgsize, mbxlength, attrs)
    Uns      msgsize;
    Uns      mbxlength;
    MBX_Attrs * attrs;
Void MBX_delete(mbx)
    MBX_Handle  mbx;

```

函数 MBX\_pend 用于等待(从邮箱中读取)一个消息,函数接口见例 4-14。如果没有可读取的消息(也就是邮箱为空),则 MBX\_pend 阻塞。这种情况下根据超时参数 timeout 来确定任务等待的时间长度:等待到超时,或无限等待,或根本不等待。

#### 例 4-14 函数 MBX\_pend

```

Bool MBX_pend(mbx, msg, timeout)
    MBX_Handle  mbx;
    Void        * msg;
    Uns         timeout; /* return after this many system clock ticks */

```

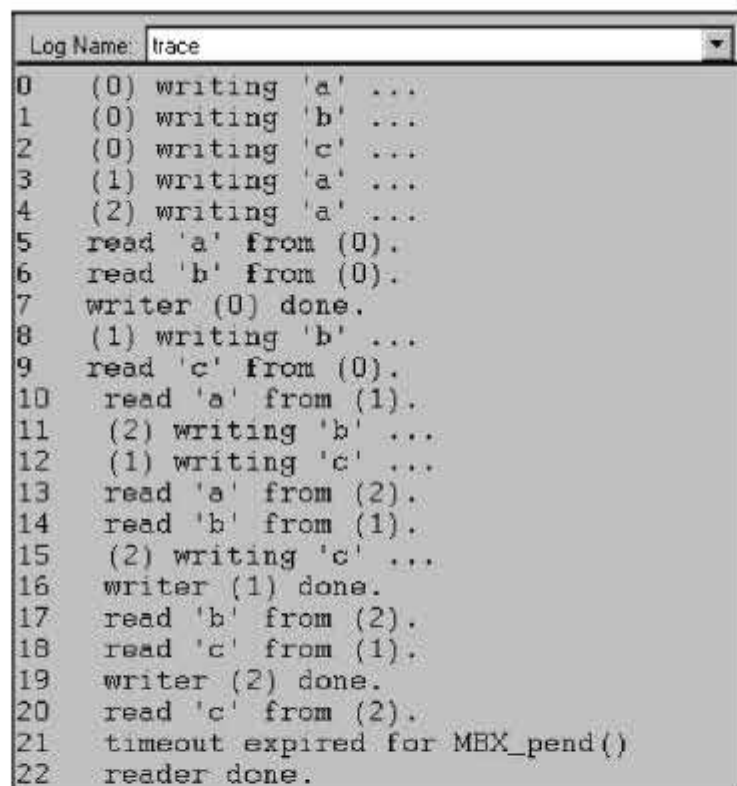
相反地,函数 MBX\_post 用于向邮箱中发布(写入)一个消息,函数接口见例 4-15。如果邮箱中没有可用的消息槽(即邮箱已满),则 MBX\_post 阻塞。这种情况下根据超时参数 timeout,任务可以一直等待到超时,或无限等待,或不等待。



#### 例 4-15 函数 MBX\_post

```
Bool MBX_post(mbx, msg, timeout)
    MBX_Handle  mbx;
    Void        * msg;
    Uns         timeout; /* return after this many system clock ticks */
```

下面的例 4-16 是一个使用邮箱的例子：多个写操作任务插入消息到邮箱中，一个读操作任务从邮箱中删除消息，图 4-18 显示了该例的执行结果。



```
Log Name: trace
0 (0) writing 'a' ...
1 (0) writing 'b' ...
2 (0) writing 'c' ...
3 (1) writing 'a' ...
4 (2) writing 'a' ...
5 read 'a' from (0).
6 read 'b' from (0).
7 writer (0) done.
8 (1) writing 'b' ...
9 read 'c' from (0).
10 read 'a' from (1).
11 (2) writing 'b' ...
12 (1) writing 'c' ...
13 read 'a' from (2).
14 read 'b' from (1).
15 (2) writing 'c' ...
16 writer (1) done.
17 read 'b' from (2).
18 read 'c' from (1).
19 writer (2) done.
20 read 'c' from (2).
21 timeout expired for MBX_pend()
22 reader done.
```

图 4-18 例 4-16 的日志追踪结果

注意：当在 HWI 函数内调用时，MBX\_post 代码必须包含在 HWI\_enter/HWI\_exit 之间，或者使用 HWI 调度程序。

#### 例 4-16 MBX 程序示例

```
/*  
 * ===== mbxtest.c =====  
 * Use a MBX mailbox to send messages from multiple writer()  
 * tasks to a single reader() task.  
 * The mailbox, reader task, and 3 writer tasks are created  
 * by the Configuration Tool.  
 *  
 * This example is similar to semtest.c. The major differences  
 * are:  
 * - MBX is used in place of QUE and SEM.  
 * - the 'elem' field is removed from MsgObj.  
 * - reader() task is * not * higher priority than writer task.  
 * - reader() looks at return value of MBX_pend() for timeout  
 * /
```

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

#define NUMMSGs 3 /* number of messages */
#define TIMEOUT 10

typedef struct MsgObj {
    Int      id;      /* writer task id */
    Char     val;      /* message value */
} MsgObj, *Msg;

/* Mailbox created with Config Tool */
extern MBX_Obj mbx;

/* "trace" Log created with Config Tool */
extern LOG_Obj trace;

Void reader(Void);
Void writer(Int id);

/*
```

```

* ===== main =====
* /
Void main()
{
    /* Does nothing */
}

/*
* ===== reader =====
* /
Void reader(Void)
{
    MsgObj msg;
    Int i;
    for (i=0;; i++) {
        /* wait for mailbox to be posted by writer() */
        if (MBX_pend(&mbx, &msg, TIMEOUT) == 0) {
            LOG_printf(&trace, "timeout expired for MBX_pend()");
            break;
        }

        /* print value */

```

```

        LOG_printf(&trace, "read '%c' from (%d).", msg.val, msg.id);
    }
    LOG_printf(&trace, "reader done.");
}

/*
 * ===== writer =====
 */
Void writer(Int id)
{
    MsgObj msg;
    Int i;
    for (i = 0; i < NUMMSGS; i++) {
        /* fill in value */
        msg.id = id;
        msg.val = i % NUMMSGS + (Int)('a');
        LOG_printf(&trace, "(%d) writing '%c' ...", id, (Int)msg.val);

        /* enqueue message */
        MBX_post(&mbx, &msg, TIMEOUT);
        /* what happens if you call TSK_yield() here? */
        /* TSK_yield(); */
    }
}

```

```
}  
LOG_printf(&trace, "writer ( %d) done. ", id);  
}
```

用户在创建邮箱对象时指定的邮箱长度决定了初始创建时邮箱中可用的消息槽的数目。为了和邮箱的写入任务进行同步,该例创建了一个计数信号灯,并将其计数值初始化为邮箱长度(即消息槽数目)。当一个任务进行一次 MBX\_post 操作,该计数值减 1。该例还创建了另一个信号灯来同步任务对邮箱的读取操作,其计数值被初始化为 0 以使读操作任务被空邮箱阻塞。当有消息发布到邮箱中时,该信号灯的计数值会加 1。

在例 4-16 中,所有任务具有相同的优先级,所有的写操作任务都尽力将其全部消息发布出去,但是邮箱一满它们就会被无限阻塞。读操作任务则不断读取消息直到在邮箱变空时被阻塞。这样的过程会一直重复直到写操作任务发布完所有的消息。

读操作任务的等待时间可根据下式计算(TIMEOUT 为超时值):

$$\text{TIMEOUT} * \text{1ms} / (\text{clock ticks per millisecond})$$

在超时发生之后,等待中的读操作任务会继续执行然后结束。

所以最好通过实验来测试调度顺序和优先级,参与同步的任务数目、邮箱长度、等待时间这些因素之间的相互影响。这需要通过以下的几方面对代码进行修改:

- ❑ 任务的创建顺序和优先级。
- ❑ 读操作任务和写操作任务的个数。

- ❑ 邮箱长度参数(MBXLENGTH)。
- ❑ 添加代码来处理写操作任务超时。

## 4.9 定时器、中断和系统时钟

大多数 TMS320 DSP 片内都具有一个或多个定时器,按照一定的时间周期产生硬件中断。DSP/BIOS 通常使用其中一个定时器作为其自身系统时钟的时钟源。使用这种片上定时器硬件,CLK 模块支持的时间分辨率可以精确到单指令周期。

用户可以在 DSP/BIOS 配置中定义 DSP/BIOS 系统时钟参数。除了系统时钟之外,用户还可以建立其他时钟对象,实现在每次定时器中断发生时触发某函数的执行。

在 C6000 平台上,用户还可以为 CLK 模块使用的 HWI 对象定义参数,该 HWI 对象被预先配置为使用 HWI 调度程序,这使得用户可以操纵 CLK ISR 的中断屏蔽位掩模和缓存控制位掩模。

DSP/BIOS 提供两种计时方法——高、低分辨率时间和系统时钟。在默认配置中,系统时钟和低分辨率时间是相同的。然而用户应用程序可以使用其他事件(如数据的有效)来驱动系统时钟。用户可以使能或禁止 CLK 管理器对片上定时器的使用来驱动或关闭高分辨率时钟和低分辨率时钟。驱动系统时钟的时钟源可以设置为低分辨率时钟、其他事件,或不



设置。这两种计时方法的相互联系如图 4-19 所示。

	CLK 模块驱动 系统时钟	其他事件驱动 系统时钟	没有事件驱动 系统时钟
CLK 管理器 被使能时	默认配置： 低分辨率时钟与 系统时钟一致	低分辨率时钟与 系统时钟不同	只有低分辨率和高分 辨率时钟可用； API 函数的超时参数 不起作用(不会流逝)
CLK 管理器 被禁止时	不可能	只有系统时钟可用； CLK 函数不能运行	没有计时方法； CLK 函数不能运行； API 函数的超时参数 不起作用(不会流逝)

图 4-19 两种计时方法的相互联系

### 4.9.1 高分辨率和低分辨率时钟

使用配置中的 CLK 管理器,用户可以设置 DSP/BIOS 是否使用片上定时器来驱动高分辨率和低分辨率时钟。

C6000 平台的 DSP 器件有多个通用定时器,而 C5400 平台的 DSP 器件只有一个通用定时器。所以在 C6000 平台上,用户可以通过配置选择一个片上定时器供 CLK 管理器使用。

在所有的平台上,用户都可以配置定时器中断触发的周期时间值。只要直接输入定时器中断的周期时间值,DSP/BIOS 会自动计算并设置周期寄存器的值。有关 CLK 管理器属性的详细介绍请查阅相应平台的 TMS320 DSP/BIOS API 函数参考手册。图 4-20 给出了 C54x 平台的 CLK 管理器属性对话框。



图 4-20 CLK 管理器属性对话框

增一次。在 C5400 平台上,当 CLK 管理器被使能时,定时器计数寄存器的值会根据下式的速率递减,其中 CLKOUT 是以 MHz 为单位的 DSP 时钟速率,TDDR 为定时器分频寄存器(divide-down register)的值:

$$\text{CLKOUT} / (\text{TDDR} + 1)$$

当计数寄存器的值递减到 0 时(C5400 和 C2800 平台)或递增到周期寄存器中设置的数值时(C6000 平台),计数寄存器则重新复位到周期寄存器的数值(C5400 和 C2800 平台)或复位到 0(C6000 平台),与此同时产生一个定时器中断。当定时器中断发生时,相应的 HWI 对象将运行 CLK\_F\_isr 函数,致使下列事件发生:

- ❑ 低分辨率时间计数值递增加 1(C6000,C2800 和 C5000 平台)。
- ❑ 所有 CLK 对象的函数在该 ISR 环境中被依次执行。

因此低分辨率时钟按照定时器中断发生的速率来变化,其计时时间值等于定时器中断发生的次数。通过调用 CLK\_gettime 可以得到低分辨率时间。

CLK 函数是在产生系统时钟的硬件中断的环境中执行的,所以 CLK 函数中进行的处理一定要尽量少,并且只能调用那些允许在 HWI 函数内调用的 DSP/BIOS 函数。

注意:因为 DSP/BIOS 运行的 CLK\_F\_isr 其内部已经调用了 HWI\_enter 和 HWI\_exit,所以 CLK 函数内不能调用这对宏。另外也不能使用 interrupt 关键字或 pragma 伪指令来指出 CLK 函数为中断服务函数。

高分辨率时钟计数则是按照片上定时器计数寄存器变化的速率(C6000 平台上按其递增速率,C5400 和 C2800 平台上按其递减速率)进行的。因此高分辨率时间等于计数寄存器值递增或递减的次数。



在 C6000 平台上,32 位的高分辨率计时时间等于低分辨率计时时间(即中断次数)乘以定时器周期寄存器的值,然后再加上定时器计数寄存器当前的值。通过调用 CLK\_gettime 可以得到高分辨率时间。当达到 32 位最大值时,高低分辨率时钟都从 0 开始重新计数。



在 C54x 和 C28x 平台上,32 位高分辨率计时时间等于低分辨率计时时间(中断次数)乘以定时器周期寄存器值,然后再加上定时器周期寄存器值和计数寄存器当前值之间的差值。通过调用 CLK\_gettime 可以得到高分辨率时间。当达到 0 值时,高分辨率时钟从周期寄存器值开始重新计数。

CLK 模块的 API 函数 CLK\_getprd 可返回定时器周期寄存器的值,CLK\_countspms 可以得到定时器计数寄存器在每毫秒内变化(递增或递减)的次数。

可以在配置中改变 CLK 管理器的属性来配置低分辨率时钟。例如,要配置低分辨率计数值每毫秒变化 1 次,在 CLK 管理器属性对话框的 Microseconds/Int 域输入 1000 即可,配置会自动计算定时器周期寄存器的值。



在图 4-20 中选中“Directly configure on-chip timer registers”选项框，则用户可以自己设置定时器周期寄存器的值。例如在 C6000 平台上，要在 160MHz 的处理器上使用四分之一 CPU 时钟产生一个 1ms 的系统时钟，那么周期寄存器值应设为：

$$\text{Period} = 0.001 \text{ sec} * 160,000,000 \text{ cycles per second} / 4 \text{ cycles} = 40,000$$



C5400 和 C2800 平台上，要在一个 40MHz 的处理器使用 CPU 驱动一个相同的系统时钟，那么周期寄存器应设为：

$$\text{Period} = 0.001 \text{ sec} * 40,000,000 \text{ cycles per second} = 40,000$$

## 4.9.2 系统时钟

很多 DSP/BIOS API 函数都有一个超时(timeout)参数。DSP/BIOS 使用系统时钟来判断是否超时。系统时钟可由低分辨率时钟驱动，也可以由外部时钟源驱动。

TSK\_sleep 就是一个具有超时参数的函数，在被调用之后，当系统时钟的变化次数达到超时参数的值时，该函数的超时时间即被耗尽。如果系统时钟的分辨率是  $1\mu\text{s}$ ，并且希望当

前任务阻塞 1ms 的时间,那么应该这样调用 TSK\_sleep:

```
/* block for 1000 ticks * 1 microsecond = 1 msec */  
TSK_sleep(1000)
```

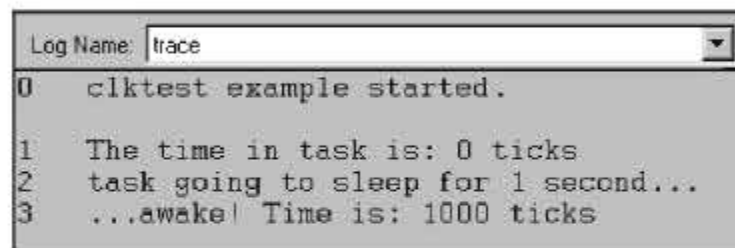
注意: 如果应用程序配置中 PRD 模块没有被任何驱动源驱动,那么在调用 TSK\_sleep 或 SEM\_post 函数时超时参数只能使用 0 或 SYS\_FOREVER。在默认配置中,PRD 模块由 CLK 模块驱动。

如果用户使用默认的 CLK 设置,那么系统时钟和低分辨率时间是相同的,因为系统时钟是由一个名为 PRD\_clock 的 CLK 对象驱动的,该 CLK 对象函数被触发执行的频率和低分辨率时间变化的频率是一致的。

系统时钟并不是一定要使用片上定时器作为其时钟源,也可以使用一个外部时钟作为其时钟源,如一个被某数据流速率驱动的时钟。如果用户不想使用片上定时器驱动系统时钟,可在配置脚本中删除名为 PRD\_clock 的 CLK 对象。当一个外部时钟被使用时,其响应程序可以调用 PRD\_tick 使系统时钟变化 1 次。另一种方案是使用片上外设(如 CODEC)按一定的时间间隔触发一个中断,在该中断的 HWI 函数中调用 PRD\_tick 使系统时钟变化 1 次。在这种情况下,系统时钟的分辨率与调用 PRD\_tick 函数的中断的频率一致。

### 4.9.3 系统时钟的实例

clktest.c 是一个简单的使用 DSP/BIOS 函数 TSK\_time 和 TSK\_sleep 的源程序,这两个函数都用到了系统时钟,如例 4-17 所示。其中被命名为 task 的任务在重新被调度程序唤醒之前要休眠 1000 个系统时钟。因为没有其他任务,所以程序在 task 被阻塞时会运行空闲循环。该程序假设系统时钟是由 PRD\_clock 对象驱动的,该程序位于 C:\ti\examples\target\bios\clktest 目录中,target 代表用户选择的平台。图 4-21 是该例的日志追踪记录输出。



```
Log Name: trace
0  clktest example started.
1  The time in task is: 0 ticks
2  task going to sleep for 1 second...
3  ...awake! Time is: 1000 ticks
```

图 4-21 例 4-17 的日志追踪输出

#### 例 4-17 使用系统时钟来控制任务

```
/* ===== clktest.c =====
 * In this example, a task goes to sleep for 1 sec and
 * prints the time after it wakes up.
 * /

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

```

#include <clk.h>
#include <tsk.h>
extern LOG_Obj trace;

/* ===== main ===== */
Void main()
{
    LOG_printf(&trace, "clktest example started.\n");
}
Void taskFxn()
{
    Uns ticks;

    LOG_printf(&trace, "The time in task is: %d ticks", (Int)TSK_time());
    ticks = (1000 * CLK_countspms()) / CLK_getprd();
    LOG_printf(&trace, "task going to sleep for 1 second... ");
    TSK_sleep(ticks);
    LOG_printf(&trace, "...awake! Time is: %d ticks", (Int)TSK_time());
}

```

注意：对于 C55x 平台，非指针类型变量在被 LOG\_printf 引用时，需要将其类型转化为 Arg，例如：



## 4.10 周期函数管理器(PRD)和系统时钟

许多应用程序都需要根据 I/O 可用性或其他可编程的事件来调度函数。其余的应用程序则基于实时时钟来调度函数。

PRD 管理器允许用户创建 PRD 对象来实现程序函数的周期性调度执行。DSP/BIOS 使用系统时钟来驱动 PRD 模块。系统时钟是一个 32 位的计数器,在每次 PRD\_tick 被调用时加 1。用户也可以使用定时器中断或其他周期性的事件来调用 PRD\_tick 以驱动系统时钟。

用户可以创建多个 PRD 对象,但它们都由同一个系统时钟驱动。每个 PRD 对象的周期参数决定着其函数被调用的频率,该周期参数以系统时钟为单位。

要基于某事件来调度函数的执行,分为两种情况:

- ❑ 基于一个实时时钟:使用配置工具,在 PRD 管理器的属性对话框中选中“Use CLK Manager to Drive PRD”选项框,也就是使用 CLK 管理器的片上定时器中断来驱动系统时钟。这时一个名为 PRD\_clock 的 CLK 对象会自动出现在 CLK 模块下,在每次定时器中断发生时这个对象调用 PRD\_tick 使系统时钟加 1。

注意：当使用 CLK 管理器来驱动 PRD 时，驱动 PRD 函数的系统时钟和低分辨率时钟是一致的。

- ❑ 基于 I/O 可用性或其他事件：清除上述选项框的选中标记。在用户应用程序内自己调用 PRD\_tick 来驱动系统时钟。这时系统时钟的分辨率与 PRD\_tick 函数被调用的频率是一致的。

## 4.10.1 调用 PRD 对象的函数

当 PRD\_tick 被调用时：

- ❑ 系统时钟计数器 PRD\_D\_tick 加 1，也即系统时钟变化 1 次。
- ❑ 当 PRD\_tick 被调用的次数等于所有 PRD 对象周期参数满足 2 的整数次幂的最大公因子时，会触发一个名为 PRD\_swi 的软件中断。例如有 3 个 PRD 对象，其周期参数分别是 12、24 和 36，那么 PRD\_swi 每 4 个系统时钟单位运行一次。因为 6 或 12 不是 2 的次幂数。

当一个 PRD 对象被静态创建时，一个名为 PRD\_swi 的新 SWI 对象会自动地被添加到配置中。当 PRD\_swi 运行时，其 SWI 函数会执行下面这类循环：

```
for ("Loop through period objects") {  
    if ("time for a periodic function")  
        "run that periodic function";  
}
```

因此,周期函数的执行是被延迟到 PRD\_swi 软件中断的环境中,而不是在调用 PRD\_tick 的硬件中断的环境中,这样在系统时钟变化时刻和 PRD 函数执行时刻之间会有一个延迟。如果用户希望 PRD 函数在系统时钟计数到周期值时立即执行,应将 PRD\_swi 软件中断设置为应用程序中优先级最高的线程。

## 4.10.2 PRD 和 SWI 的统计信息

实时系统中的许多任务都是周期性的,即它们以固定的时间间隔不断地运行。因此保证这些作业在下一次运行时间到达之前能够完成执行是很重要的,否则将超出系统的实时处理时限。尽管可以利用内部数据缓冲来恢复偶然的超时,但频繁的超时将导致不可恢复的错误。

为 SWI 函数收集的隐式统计信息可以测量一个软件中断从就绪到执行完毕所需的时间长度。因为处理器实际上要执行许多的硬件和软件中断,所以这种时间测量是很关键的。如果一个软件中断就绪后还要等待很长时间让别的软件中断完成,或者任务开始执行后被中断太多次而且在很长的一段时间内不能恢复执行,那么都有可能超出实时处理时限。

就绪到完成时间(ready-to-complete time)的最大值可以用于判断系统是否有失败的潜在可能。一个软件中断的最大就绪到完成的时间越接近软件中断的周期,系统就越有可能在突发数据计算负荷下失败。最大就绪到完成时间还可以显示出系统在将来的产品性能增强上还有多大的空间。

注意：DSP/BIOS 并没有隐式地测量每个软件中断的执行时间。实际的执行时间可以通过软件模拟(simulator)或硬件仿真(emulator)方式单独执行软件中断来计算精确的时钟周期数。

另外需要注意的是,即使系统里所有线程所需的 MIPS 总和远远小于 DSP 的 MIPS 指标,系统仍然有可能达不到实时要求。一个 CPU 负荷小于 70%的系统因为优先级问题而不能满足实时要求的情况并不少见。利用 DSP/BIOS 监测最大就绪到完成时间有助于立即发现这类问题。

当 SWI 和 PRD 对象的统计信息收集被使能时,主机会自动采集下列类型的统计数据的统计次数、总和、最大值和平均值:

- ❑ **SWI**: 统计软件中断从被触发到完成所经过的系统时钟数。
- ❑ **PRD**: 统计周期函数从就绪到完成所经过的系统时钟数。根据定义,周期函数在系统时钟计数达到其周期参数值时就绪。

用户可以在 CLK 管理器的属性对话框中设置统计数据的测量单位。如果选中“Use high resolution time for interval timings”选项框(默认为选中),那么时间值是以时钟周期为单位进行测量的,如果未选中则会以定时器中断周期为时间单位。用户也可以在统计信息视图属性对话框中选择毫秒或微秒作为时间单位。

如果一个 PRD 对象完成时间的最大值在不断地增加,该对象有可能不满足其实时要求。一个 PRD 对象完成时间的最大值应该小于等于该对象的周期参数值(以系统时钟为单位),如果大于该周期参数值(如图 4-22),PRD 函数将超出实时时间限制。

Statistics View				
STS	Count	Total	Max	Average
loadPrd	1931	0	0	0
stepPrd	1	0	0	0
PRD_swi	1931	71200064.00 inst	102572.00 inst	36872.12 inst
KNL_swi	15453	81301080.00 inst	102764.00 inst	5261.18 inst
audioSwi	1287	2693364.00 inst	3236.00 inst	2092.75 inst
IDL_busyObj	635928	1217	1	0.00191374

图 4-22 一个 PRD 对象的统计信息视图

## 4.11 使用执行图观察程序的执行情况

在 CCS 中选择“DSP/BIOS”→“Execution Graph”菜单选项,即可打开程序执行图来以可视化的方式观察线程的行为。

### 4.11.1 执行图中的状态指示

程序执行状态图会检查系统日志(LOG\_system 对象)中的信息,将各个线程的状态根据定时器中断(图 4-23 中的 Time)和系统时钟计数(图 4-23 中的 PRD Ticks)为基准显示出来。

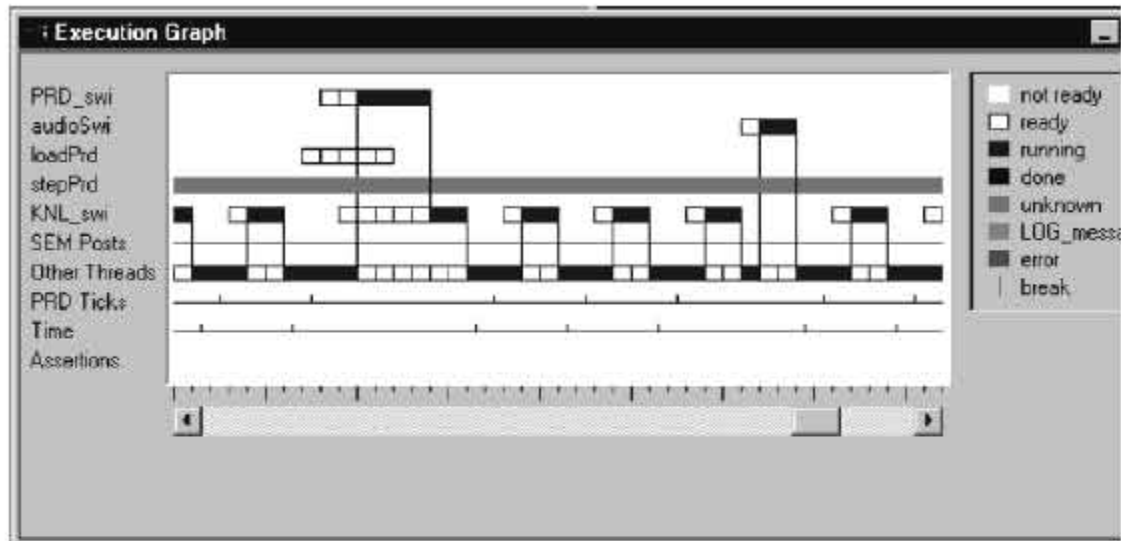


图 4-23 程序执行状态图

执行图使用不同颜色指示出线程的不同状态：

- ❑ 白色方块(**ready**)：代表线程已经被触发并且处于就绪状态,准备运行。
- ❑ 蓝绿色方块(**unknown**)：代表当前时刻主机没有从系统日志里收到任何关于该线程状态的信息。
- ❑ 暗蓝色方块(**running**)：代表线程正在运行。
- ❑ 亮红色方块(**error**)：代表有错误发生。例如一个线程没有满足其实时处理时限，或检测到无效的日志记录,执行图都会给出错误指示。无效的日志记录可能由程序对系统日志的覆盖引起,用户可以双击一个错误指示查看其细节。

## 4.11.2 执行图中的线程

在执行图窗口左侧按优先级大小排列了 SWI 函数和 PRD 函数。出于对性能考虑，执行图中没有显示出硬件中断和后台线程的信息(除了可能使用定时器中断的 CLK 对象)。没有在 SWI、PRD 和 TSK 线程内花费的 CPU 时间必定是在 HWI 或后台线程中花费的，所以这部分时间在“Other Threads”行(见图 4-23)上显示的。

PIP 通知函数作为调用它的线程的一部分运行，执行图不会将其作为单独的线程显示。LNK\_dataPump 对象(会运行一个函数来管理一个 HST 对象的主机端操作)和其他的 IDL 对象都在 IDL 后台线程中运行，所以被包含在“Other Threads”中。

注意：Time 标记和 PRD Ticks 标记并非等间隔地出现，但这并不意味着时间间隔是不相等的。因为执行图会为每个线程显示出一个方框，所以在相邻两次定时器中断(Time)或系统时钟计数(PRD Ticks)之间发生的事件越多，两个标记之间的距离就会越宽。

## 4.11.3 执行图中的序列号

执行图窗口中位于底部滑动条上方的刻度标记指示了事件执行的顺序，如图 4-23 所示。

注意：环形日志(执行图默认采用的 LOG 类型)只包含了最近发生的  $n$  个事件。通常有些事件没有被列出来,这是因为它们发生在上次主机查询记录之后,但又在下一次主机查询之前被覆盖了。执行图窗口中会显示一个红色的竖线并且在日志序列号中产生一个断裂来标记这种情况。

用户可以通过增加日志缓冲区的长度观察到更多的事件,也可以设置 RTA 控制面板只记录感兴趣的事件。

#### 4.11.4 使用 RTA 控制面板设置执行图

TRC 模块允许用户来选择在程序执行过程中需要被记录到执行图中的事件。执行图中 SWI、PRD 和 CLK 事件的记录既可以通过主机来控制(使用 RTA 控制面板,如图 4-24 所示,通过选择 CCS 菜单选项“DSP/BIOS”→“RTA Control Panel”),也可以通过目标代码来控制(调用 API 函数 TRC\_enable 和 TRC\_disable)。因此在程序运行过程中,TRC 模块也可



以用来控制需要在执行图中显示哪些对象事件。关于如何控制隐式监测,详见 3.3.4.2 小节。

当使用执行图时,关闭自动轮询就会停止日志的频繁更新,使用户可以有时间来分析执行图。用户可以采用下列方法关闭自动轮询:

- ❑ 在执行图窗口中单击右键,从快捷菜单中选择“Pause”命令。
- ❑ 在 RTA 控制面板中单击右键,从快捷菜单中选择“Property Page”打开属性对话框。将“Event log/Execution Graph refresh rate”参数域设为 0,单击“OK”按钮。

任何时候用户都可以在执行图窗口单击右键从快捷菜单里选择“Refresh”命令从目标 DSP 查询日志记录的数据来更新执行状态图,也可将窗口刷新多次来观察额外的数据。用户也可以在右键快捷菜单里选择“Clear”命令清除执行图里显示的数据。

如果用户程序的执行序列比较复杂而又想要使用执行图,那么最好在配置中加大执行图记录缓冲区的长度。在配置工具中右键单击 LOG 对象 LOG\_system,选择“Properties”打开属性对话框并增加 buflen 属性域(缓冲区长度)的值。因为每个记录消息占用 4 个字,因此 buflen 的值至少等于需要记录的事件个数乘以 4。

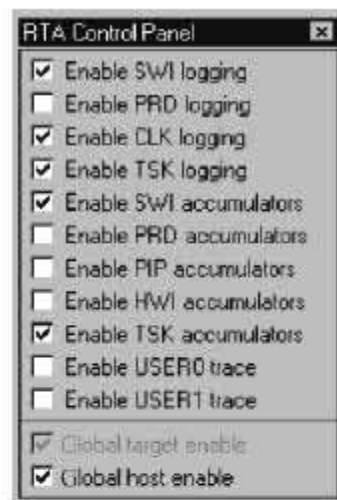


图 4-24 RTA 控制面板对话框



在 C55x 平台上,大存储器模型的数据指针的长度为 23 位,且所有的长字(long word)访问需要满足奇地址边界限制。这使得事件记录缓冲区的大小增加一倍(即每个消息占用 8 个字的空间)。

# 第 5 章 存储器和低级函数

本章描述了 DSP/BIOS 实时多任务内核中的低级函数。这些函数包含在以下的 DSP/BIOS 软件模块中：

- ❑ **MEM** 和 **BUF** 模块：分别用于管理可变长度和固定长度存储空间的分配。
- ❑ **SYS** 模块：用于提供多种系统服务。
- ❑ **QUE** 模块：用于管理队列。

本章也给出了一些使用这些模块的简单的程序示例，在相应平台的 TMS320 DSP/BIOS API 函数参考手册中有对这些模块的 API 函数更为详细的描述。

## 5.1 存储器管理

存储器管理器(MEM 模块)用于管理与物理存储器区域相对应的已命名的存储器段(memory segments)。每个 MEM 对象就对应一个存储器段，MEM 对象的名称就是该存储器段的名称，该存储器段的起始地址和长度可通过该对象的属性进行设置。

用户可以使用 MEM 模块将程序的不同存储段(memory sections)(如, text、. data 等)分配到不同的存储器段中,实现 CMD 文件的功能。实际上 DSP/BIOS 就是根据用户对 MEM 模块的设置自动生成 CMD 文件的。如果想对存储器段进行更多的控制,用户可以创建自己的链接命令文件并将其添加到项目中。

MEM 模块还提供了一套 API 函数用于在存储器段中动态分配以及释放可变长度的存储块,而 BUF 模块的 API 函数则用于固定长度的存储块的动态管理。

与 malloc 等标准 C 函数不同, MEM 的 API 函数使得用户可以具体指定使用哪一个存储器段来满足特定的存储器分配需求。实时 DSP 硬件平台一般会包含几种不同类型的物理存储器:快速片内 RAM、零等待状态外部 SRAM、大数据容量慢速 DRAM 等。为了满足许多 DSP 应用程序的实时性约束,对哪一个存储器段存储哪一种数据块的精确控制是十分必要的。

MEM 模块不会设置那些与 DSP 存储器子系统有关的硬件寄存器,这些配置工作需要由用户完成,通常是在引导程序中加入代码或在 CCS 下通过 GEL 语言实现。例如,为了在 C6000 平台上访问外部存储器,外部存储器接口(EMIF)的控制寄存器必须在访问前正确设置。DSP/BIOS 应用程序中用户初始化 EMIF 的最早机会是在用户初始化函数中(见 TMS320 DSP/BIOS API 函数参考手册中的全局设置)。

MEM 函数可以分配和释放可变长度的存储块,当使用 MEM 模块时内存的分配与释放具有不确定性,因为 MEM 模块会为每个具体的存储器段(MEM 对象)保存并维护一个空闲存储块的链表,而调用 MEM\_alloc 与 MEM\_free 函数进行内存分配与释放时是在链表中进行搜索和操作的。

### 5.1.1 配置存储器段

DSP/BIOS 提供的配置模板预先定义了一系列存储器段(MEM 对象),这些存储器段根据具体的 DSP 硬件平台而有所不同(详见 1.3.5 小节“存储器段命名”)。一般来说,用户在自己的硬件平台上开发 DSP/BIOS 应用程序时,需要在 DSP/BIOS 配置工具中通过改动定制自己的 MEM 对象以及它们的属性。方法有:

- ❑ 插入一个新 MEM 对象并定义其属性。关于 MEM 对象属性的详细信息,请查阅相应 DSP 平台的 TMS320 DSP/BIOS API 函数参考手册。
- ❑ 修改某个现有的存储器段的属性。
- ❑ 删除一些 MEM 对象,特别是那些对应于外部存储空间的存储器段。但是必须首先修改其他 DSP/BIOS 对象及其模块管理器对该存储器段的引用。为了找出对该 MEM 对象的依赖关系,右键单击该 MEM 对象,然后在快捷菜单中选择“Show Dependencies”。注意,用户不能删除或重命名 IPRAM 和 IDRAM 对象(C6000 平台)或 IRROG 和 IDATA 对象(C5000 平台)。

❑ 重命名某些 MEM 对象。步骤如下：

(1) 按照上一种方法查看并删除对该存储器段的依赖关系。

(2) 重命名此对象。可以右击该对象,在弹出菜单中选择“Rename”来编辑新名称。

(3) 根据需要重新建立对该存储器段的依赖关系,即在其他相关对象的属性对话框中选择该存储器段。

## 5.1.2 禁止动态存储分配

如果小的代码尺寸在用户的应用中很重要,可以通过禁止动态分配与释放存储块的功能来达到减小代码尺寸的目的。在 MEM 管理器的属性对话框中选中“No Dynamic Memory Heaps”选项框,则禁止动态分配与释放存储块的能力。这时,用户程序将不能调用任何 MEM 函数或任何对象创建函数(例如 TSK\_create)。用户只能使用配置工具静态创建所有程序中用到的对象。

如果动态存储块分配被禁止而用户程序调用了一个 MEM 函数(或者调用了一个对象创建函数导致对 MEM 函数的间接调用),就会出现错误。这时如果传递给 MEM 函数的 segid 参数是存储器段的名称,则会产生链接错误;如果 segid 参数是整数,则 MEM 函数会调用 SYS\_error 函数告知发生错误。

### 5.1.3 在自己的链接命令文件中定义存储器段

MEM 管理器允许用户为不同类型的代码与数据选择合适的存储器段,如果用户想对这些代码与数据的存储作更多控制,可以选中 MEM 管理器属性对话框中的“User .cmd file for non-DSP/BIOS segments”选项框。

接下来用户需要建立自己的链接命令文件,并要注意在文件最开始包含由配置工具自动创建的链接命令文件,例如:

#### 例 5-1 链接命令文件(C6000 平台)

```
/* First include DSP/BIOS generated cmd file. */
-l designcfg.cmd
SECTIONS {
    /* place high-performance code in on-device ram */
    .fast_text: {
        myfastcode.lib* (.text)
        myfastcode.lib* (.switch)
    } > IPRAM
    /* all other user code in off device ram */
    .text:      {} > SDRAM0
    .switch:    {} > SDRAM0
    .cinit:     {} > SDRAM0
```

```

.pinit:      {} > SDRAM0
/* user data in on-device ram */
.bss:        {} > IDRAM
.far:        {} > IDRAM
}

```

## 例 5-2 链接命令文件(C5000 和 C28x 平台)

```

/* First include DSP/BIOS generated cmd file. */
-l designcfg.cmd
SECTIONS {
    /* place high-performance code in on-device ram */
    .fast_text: {
        myfastcode.lib* (.text)
        myfastcode.lib* (.switch)
    } > IPROG PAGE 0

    /* all other user code in off device ram */
    .text:      {} > EPROG0 PAGE 0
    .switch:    {} > EPROG0 PAGE 0
    .cinit:     {} > EPROG0 PAGE 0
    .pinit:     {} > EPROG0 PAGE 0
}

```



```

/* user data in on-device ram */
.bss:      {} > IDATA PAGE 1
.far:      {} > IDATA PAGE 1
}

```

## 5.1.4 动态存储分配

DSP/BIOS 提供了 MEM 和 BUF 两类模块函数实现动态存储块分配：MEM 模块分配长度可变的存储块，BUF 模块则分配固定长度的缓冲区。

### 5.1.4.1 MEM 模块的存储分配

调用 MEM\_alloc 函数完成基本的存储块分配，其函数参数有存储器段标识符、存储块大小以及存储块地址的边界条件。如果分配成功，MEM\_alloc 返回存储块的起始地址；如果失败则返回 MEM\_ILLEGAL，其函数接口如例 5-3。

#### 例 5-3 MEM\_alloc 函数

```

Ptr MEM_alloc(segid, size, align)
    Int segid;
    Uns size;
    Uns align;

```

参数 segid {

名称或一个整数。

参数 size 指示了要分配的存储块的大小,其单位为最小可寻址数据单元(MADUs)。由 MEM\_alloc 返回的存储块中至少包含 size 个 MADU。一个处理器的最小可寻址数据单元是其可以读取或存储的最小数据单元,例如 C5000 平台的 MADU 是一个 16 位字,而 C6000 平台为 8 位字节。

由 MEM\_alloc 返回的存储块起始地址必须是边界参数 align 的整数倍。如果 align 参数为 0 则没有起始地址的约束,但 MEM 模块在实现分配时还是会用存储 MEM\_Header 结构体所需的字(word)数作为边界条件。而当 align 值为其他值时,函数执行时会保证存储块的起始地址位于 align 个字的边界,align 为 2 的次幂数。

许多数字信号处理算法都使用的是环形缓冲,在大多数 DSP 平台上当环形缓冲区位于 2 的整数次幂的地址边界时,这种缓冲区的边界限制使代码能更有效地利用循环寻址模式。

下面给出的这个实例使用 MEM\_alloc 函数给一个结构体数组分配存储块。

#### 例 5-4 分配一个数据结构数组

```
typedef struct Obj {  
    Int field1;  
    Int field2;  
    Ptr objArr;
```

```
} Obj;  
objArr = MEM_alloc(SRAM, sizeof(Obj) * ARRAYLEN, 0);
```

调用 MEM\_free 函数能够释放先前由 MEM\_alloc、MEM\_calloc 或 MEM\_valloc 动态分配的存储块。MEM\_free 的参数 segid、ptr 和 size 分别指定存储器段名、存储块指针和存储块的大小,这些参数必须与存储块分配时使用的参数一致。其调用示例如例 5-5 所示。

#### 例 5-5 MEM\_free 函数

```
Void MEM_free(segid, ptr, size)  
    Int segid;  
    Ptr ptr;  
    Uns size;
```

#### 例 5-6 释放例 5-4 中为结构体队列分配的存储块

```
MEM_free(SRAM, objArr, sizeof(Obj) * ARRAYLEN);
```

### 5.1.4.2 BUF 模块的存储分配

BUF 模块用来管理若干个缓冲池,每个缓冲池包含有若干个固定长度的缓冲区。这些缓冲池既能静态创建也能动态创建。动态创建的缓冲池是从 MEM 模块管理的动态存储器堆(heap)中进行分配的。调用函数 BUF\_create 即可动态创建一个缓冲池。当在应用程序设计阶段已经知道缓冲池长度和边界约束等条件时,缓冲池通常在配置工具里被静态地创

建。当这些限制条件会随着程序执行发生变化时可采用实时创建。

在缓冲池内,所有的缓冲区都具有相同的大小和边界条件。从应用程序的使用角度来讲可以把每个缓冲区称为一帧,虽然每帧的长度是相同的,但是应用程序可以将数量不等的数据存放到这些帧中,只要不超过帧的长度。因此用户通常创建帧长不同的多个缓冲池适应不同大小的分配需要。

根据需要,用户可以使用 BUF\_alloc 和 BUF\_free 在运行时从已创建的缓冲池中分配和释放缓冲区。相对于 MEM 模块提供的直接从动态存储堆中分配存储块的方法,从缓冲池中分配缓冲区具有以下一些优势:

- ❑ 分配时间具有确定性。BUF\_alloc 和 BUF\_free 函数执行时需要的时间是固定的,而在存储堆中分配和释放存储空间的时间则不确定。
- ❑ 能被所有类型的线程调用。分配和释放缓冲区是原子操作不会被阻塞或中断,因此 BUF\_alloc 和 BUF\_free 可以被所有类型的 DSP/BIOS 线程所调用,包括 HWI、SWI、TSK 和 IDL。相反地,HWI 和 SWI 线程则不能调用 MEM\_alloc。
- ❑ 针对固定长度的分配进行了优化。相比较而言,MEM\_alloc 则是针对可变长度的分配进行了优化。
- ❑ 没有存储器碎片  
碎片。

### 5.1.5 获得一个存储器段的状态

用户通过调用 `MEM_stat` 可以获得一个存储器段的状态,调用该函数可以获得指定存储器段的长度值、已经使用的存储块总长度以及可分配的最大连续存储块的长度。这些长度值都是以最小可寻址数据单元(MADUs)为单位给出的。这些值包含在一个 `MEM_stat` 类型的结构体中。

如果用户使用的是 `BUF` 模块,则可以调用 `BUF_stat` 来获得缓冲池的状态信息,也可以调用 `BUF_maxbuff` 来得到一个缓冲池中已经使用的缓冲区的最大数目。

### 5.1.6 减小存储器碎片

上文提到,调用 BUF 模块函数从缓冲池中分配和释放定长的缓冲区能够减少存储器碎片。而调用 MEM 模块函数频繁分配和释放可变大小的存储块会使存储器段变得支离破碎,从而产生存储器碎片。这时,很可能没有足够长度的连续空闲存储区来满足较长的分配需求,调用 MEM\_alloc 会返回 MEM\_ILLEGAL,有时甚至当空闲存储区总长度大于请求长度时,也会分配失败。

为了在分配可变长度的存储块时将存储器碎片影响降至最小,用户可以如图 5-1 那样使用不同的存储器段分配长度不同的存储块。

注意:为了减小内存碎片,在第一个存储器段中分配那些等尺寸的小存储块(例如用来存放消息),在第二个存储器段中分配那些等尺寸的大存储块(用来存放数据流)。



图 5-1 使用不同存储器段分配不同尺寸的存储块

### 5.1.7 MEM 模块使用举例

在例 5-7 与例 5-8 中,调用 MEM\_alloc 从 IDATA 与 IDRAM 存储器段中分配存储块,并随后由 MEM\_free 释放。printmem 用来将存储器段状态写入日志对象 trace 的缓冲区,存储器段的最终状态(即“释放后”的状态)应该与初始状态值相匹配。图 5-2 显示的是例 5-7 和例 5-8 的日志追踪结果。

#### 例 5-7 存储块分配(C5000 和 C28x 平台)

```
/* ===== memtest.c =====  
* This code allocates/frees memory from different memory  
* segments.  
* /
```

```

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

#define NALLOCS 2      /* of allocations from each segment */
#define BUFSIZE 128    /* size of allocations */

/* "trace" Log created by Configuration Tool */
extern LOG_Obj trace;
#ifdef -54-
    extern Int IDATA;
#endif
#ifdef -55-
    extern Int DATA;
#endif
static Void printmem(Int segid);

/*
 * ===== main =====
 */
Void main()
{
    Int i;

```



```
Ptr ram[NALLOCS];  
LOG_printf(&trace, "before allocating ...");  
  
/* print initial memory status */  
printmem(IDATA);  
LOG_printf(&trace, "allocating ...");  
  
/* allocate some memory from each segment */  
for (i = 0; i < NALLOCS; i++) {  
    ram[i] = MEM_alloc( IDATA, BUFSIZE, 0);  
    LOG_printf(&trace, "seg %d: ptr = 0x%x", IDATA, ram[i]);  
}  
LOG_printf(&trace, "after allocating ...");  
  
/* print memory status */  
printmem(IDATA);  
  
/* free memory */  
for (i = 0; i < NALLOCS; i++) {  
    MEM_free(IDATA, ram[i], BUFSIZE);  
}  
LOG_printf(&trace, "after freeing ...");
```

```

        /* print memory status */
        printmem(IDATA);
    }

    /*
    * ===== printmem =====
    */
    static Void printmem(Int segid)
    {
        MEM_Stat statbuf;
        MEM_stat(segid, &statbuf);
        LOG_printf(&trace, "seg %d: 0 0x%x", segid, statbuf.size);
        LOG_printf(&trace, "\tU 0x%x\tA 0x%x", statbuf.used,
                    statbuf.length);
    }

```

注意：对于 C55x 平台，非指针类型变量在被 LOG\_printf 引用时，需要将其类型转化为 Arg，例如：

```
LOG_printf(&trace, "Task %d Done", (Arg) id);
```

#### 例 5-8 存储块分配(C6000 平台)

```

/* ===== memtest.c =====
* This program allocates and frees memory from

```

```

* different memory segments.
* /

#include <std.h>
#include <log.h>
#include <mem.h>
#define NALLOCS 2      /* # of allocations from each segment */
#define BUFSIZE 128    /* size of allocations */

/* "trace" Log created by Configuration Tool */
extern LOG_Obj trace;
extern Int IDRAM;
static Void printmem(Int segid);

/*
* ===== main =====
* /
Void main()
{
    Int i;
    Ptr ram[NALLOCS];
    LOG_printf(&trace, "before allocating ...");

```

```

/* print initial memory status */
printmem(IDRAM);
LOG_printf(&trace, "allocating ...");

/* allocate some memory from each segment */
for (i = 0; i < NALLOCS; i++) {
    ram[i] = MEM_alloc( IDRAM, BUFSIZE, 0);
    LOG_printf(&trace, "seg %d: ptr = 0x%x", IDRAM, ram[i]);
}
LOG_printf(&trace, "after allocating ...");

/* print memory status */
printmem(IDRAM);

/* free memory */
for (i = 0; i < NALLOCS; i++) {
    MEM_free( IDRAM, ram[i], BUFSIZE);
}
LOG_printf(&trace, "after freeing ...");

/* print memory status */
printmem(IDRAM);
}

```

```

/*
 * ===== printmem =====
 */
static Void printmem(Int segid)
{
    MEM_Stat statbuf;
    MEM_stat(segid,&statbuf);
    LOG_printf(&trace,"seg %d: 0 0x%x",segid,statbuf.size);
    LOG_printf (&trace,"\\tU 0x%x\\tA 0x%x",statbuf.
                used,statbuf.length);
}

```

例 5-7 与例 5-8 中程序执行的结果与其面向的平台有关。图 5-2 所示的窗口中,O 显示了原始存储器段的大小,U 为已经使用的 MADUs,A 为空闲 MADU 中最大连续存储块长度。用户验证时所看到的地址可能与图 5-2 中的显示有所不同。

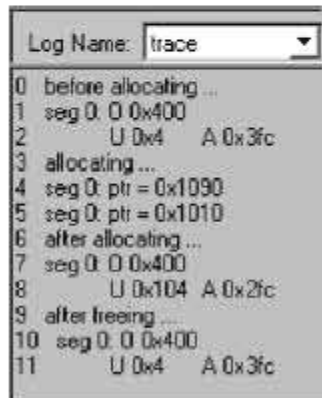


图 5-2 存储块分配日志追踪窗口

## 5.2 系统服务

SYS 模块提供了一些基础的系统服务函数,其形式按照标准 C 程序运行库中的同类函数设计。依照惯例,DSP/BIOS 软件模块会在需要同类 C 库函数的场合使用 SYS 模块所提供的服务。

用户可以为某个 SYS API 配置一个函数,当应用程序在任何时候调用该 SYS API 时,这个函数会被触发执行。详细信息请参见相应平台的 TMS320 DSP/BIOS API 函数参考手册中有关 SYS 的章节。

### 5.2.1 停止程序执行

例 5-9 中给出 SYS 提供的两个停止程序运行的函数。SYS\_exit 用来安全地有秩序地退出程序;SYS\_abort 用于灾难性情况下的异常中止。由于程序退出或中止时执行的具体操作通常与具体的系统硬件平台具有相关性,因此用户需要根据自己的硬件平台修改 DSP/BIOS 配置,使得在调用 SYS\_exit 和 SYS\_abort 时满足自己的具体系统的相关需要。

**例 5-9** SYS\_exit 和 SYS\_abort 函数

```
Void SYS_exit(status)
```

```
    Int status;
```

```
Void SYS_abort(format,[arg,] ...)
    String format;
    Arg arg;
```

用户可以在 SYS 模块的“Exit Function”和“Abort Function”两个属性域中分别指定自己的退出函数和中止函数,例 5-9 给出 SYS API 函数会分别调用它们来停止程序执行,默认的退出函数是 UTL\_halt;默认的中止函数是 \_UTL\_doAbort(它会将错误信息记录下来并调用 \_halt 函数, \_halt 在 boot.c 文件中定义,它会禁止所有中断,然后进入无限循环)。

SYS\_abort 函数的参数包括一个字符串和一组可选的数值参数(例如一个诊断消息),SYS\_abort 会将这些参数传递给“Abort Function”属性中指定的中止函数(见例 5-10)。

参数 vargs 的类型为 va\_list,它代表了传递给 SYS\_abort 的多个 arg 参数组成的序列。而“Abort Function”属性中指定的中止函数可以在中止程序之前将参数 format 和 vargs 直接传递给 SYS\_vprintf 或 SYS\_vsprintf 以打印出这些参数。为避免 SYS\_vprintf 与 SYS\_vsprintf 增加代码长度,也可以使用 LOG\_error 将错误信息简单地打印出来。

**例 5-10** SYS\_abort 中对中止函数的调用

```
( * (Abort_function)) (format, vargs)
```

类似地, `SYS_exit` 会调用由“Exit Function”属性域指定的退出函数, 并把自己接收的参数 `status` 传递给它。在执行退出函数之前, `SYS_exit` 首先会执行一系列通过 `SYS_atexit` 函数注册到系统中的清理(clean-up)函数, 完成退出前的清理工作, 实现安全退出。清理函数的注册和调用通过函数句柄完成, 见例 5-11。

`SYS_atexit` 函数提供了某种机制可以让用户将最多 `SYS_NUMHANDLERS`(被设为 8)个清理函数的句柄压入堆栈, 即注册清理函数, 见例 5-12。当内部堆栈已注册满时, 对 `SYS_atexit` 的调用会返回 `FALSE`。

**例 5-11** `SYS_exit` 中通过被注册的函数句柄执行清理函数和退出函数

```
( * handlerN)(status)
...
( * handler2)(status)
( * handler1)(status)
( * (Exit_function))(status)
```

**例 5-12** 使用 `SYS_atexit` 注册清理函数

```
Bool SYS_atexit(handler)
    Fxn handler;
```

## 5.2.2 错误处理



SYS\_error 用来对 DSP/BIOS 的异常错误进行处理, 见例 5-13。DSP/BIOS 内部函数使用 SYS\_error 来处理程序错误。用户应用程序也可以使用它。

### 例 5-13 DSP/BIOS 错误处理

```
Void SYS_error(s, errno, ...)
    String s;
    Uns errno;
```

SYS\_error 调用 SYS 模块“Error Function”属性域指定的错误函数来处理错误状况, 默认的错误函数是 \_UTL\_doError, 它会将错误信息记录下来。例 5-14 给出了一个用户定义错误函数的例子, “Exit Function”属性域指定的函数为 doError, 它使用 LOG\_error 来打印错误信息及相关的字符串说明信息。

### 例 5-14 应用 doError 来打印错误信息

```
Void doError(String s, Int errno, va_list ap)
{
    LOG_error("SYS_error called: error id = 0x%x", errno);
    LOG_error("SYS_error called: string = '%s'", s);
}
```

个用户错误( $\text{errno} \geq 256$ )。DSP/BIOS 错误编码和字符串内容详见对应平台的 TMS320 DSP/BIOS API 函数参考手册。

注意：DSP/BIOS API 函数已将错误检测所带来的存储器及 CPU 需求限制到最小。但是这也带来 DSP/BIOS API 函数调用时的约束，API 参考手册中有详细说明。应用程序开发者必须注意满足这些调用约束。

## 5.3 队 列

QUE 模块提供了一系列函数来管理一个 QUE 元素链表。尽管队列元素可以在链表的任何位置进行插入或删除，但 QUE 模块通常用于实现一个 FIFO 链表——队列元素从表尾插入，从表头删除。队列元素可以是任何种类的结构体，但其第一个域必须为 `QUE_Elem` 类型。QUE 模块用 `QUE_Elem` 类型的参数来完成队列元素的入队出队操作，而实际的数据则包含在队列元素其他域中，见例 5-15。

`QUE_create` 和 `QUE_delete` 用来生成和删除整个队列，由于 QUE 模块像链表一样执行，队列无最大尺寸，如例 5-15 所示。

**例 5-15** QUE 元素的队列管理

```
    struct QUE_Elem * next;
    struct QUE_Elem * prev;
} QUE_Elem;

typedef struct MsgObj {
    QUE_Elem elem;
    Char val;
} MsgObj;

QUE_Handle QUE_create(attrs)
    QUE_Attrs * attrs;
Void QUE_delete(queue)
    QUE_Handle queue;
```

### 5.3.1 原子 QUE 函数

函数 `QUE_put` 与 `QUE_get` 分别用于在队列尾部插入元素和在队列头部移除元素, 这些函数是原子(atomic)函数, 即它们在执行时会禁止中断。因此, 多个线程可以安全地使用这两个函数来修改队列而无须外部同步。

`QUE_get` 函数从队列头部移除元素并将该元素返回, 相反 `QUE_put` 在队列尾部插入一个

元素。对于这两个函数,队列元素应为 Ptr 类型以避免不必要的类型转换,如例 5-16 所示。

#### 例 5-16 队列的原子操作

```
Ptr QUE_get(queue)
    QUE_Handle queue;

Ptr QUE_put(queue, elem)
    QUE_Handle queue;
    Ptr elem;
```

### 5.3.2 其他 QUE 函数

不同于 QUE\_get 和 QUE\_put,另外一些 QUE 函数不能在队列操作时禁止中断。如果队列被多个线程所共享,这些函数调用执行时必须结合某种互斥机制。

QUE\_dequeue 和 QUE\_enqueue 的作用等同于 QUE\_get 与 QUE\_put,但它们在更新队列时没有禁止中断。

QUE\_head 函数仅返回指向队列首元素的指针而不删除该元素,QUE\_next 和 QUE\_prev 用来在队列中搜索元素——QUE\_next 返回队列中指向下一个元素的指针,而 QUE\_prev 返回队列中指向上一元素的指针。

QUE\_insert 和 QUE\_remove 用来在队列中的任一位置插入或删除元素。

#### 例 5-17 其他 QUE 函数

```

Ptr QUE_dequeue(queue)
    QUE_Handle queue;
Void QUE_enqueue(queue, elem)
    QUE_Handle queue;
    Ptr elem;
Ptr QUE_head(queue)
    QUE_Handle queue;
Ptr QUE_next(qelem)
    Ptr qelem;
Ptr QUE_prev(qelem)
    Ptr qelem;
Void QUE_insert(qelem, elem)
    Ptr qelem;
    Ptr elem;
Void QUE_remove(qelem)
    Ptr qelem;

```

注意：由于 QUE 队列是带有头节点的双向链表，QUE\_head、QUE\_next 或 QUE\_prev 可能会返回指向头节点本身的指针（例如，对空队列调用函数 QUE\_head），因此应注意不要调用 QUE\_remove 误删掉头节点。

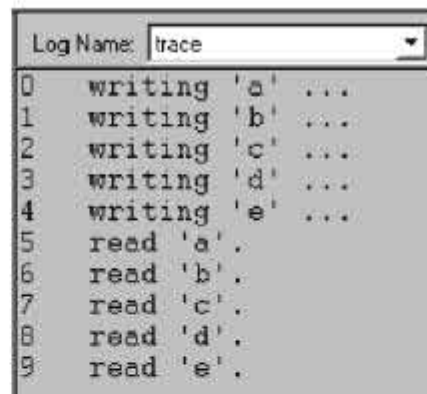
### 5.3.3 QUE 程序示例

例 5-18 用 QUE 队列实现了一个写任务线程到读任务线程的 5 个信息的传递。MEM\_alloc 与 MEM\_free 函数用来为 MsgObj 结构体分配及释放存储块。

例 5-18 程序运行结果如图 5-3 所示,写任务线程调用 QUE\_put 将得到的 5 个信息入队,然后读任务线程用 QUE\_get 来从队列中取出每个信息。

**例 5-18** 使用 QUE 发送信息

```
/*  
 * ===== quetest.c =====  
 * Use a QUE queue to send messages from a writer to a reader  
 *  
 * The queue is created by the Configuration Tool.  
 * For simplicity, we use MEM_alloc and MEM_free to manage  
 * the MsgObj structures. It would be way more efficient to  
 * preallocate a pool of MsgObj's and keep them on a 'free'  
 * queue. Using the Config Tool, create 'freeQueue'. Then in  
 * main, allocate the MsgObj's with MEM_alloc and add them to
```



Log Name:	trace
0	writing 'a' ...
1	writing 'b' ...
2	writing 'c' ...
3	writing 'd' ...
4	writing 'e' ...
5	read 'a' .
6	read 'b' .
7	read 'c' .
8	read 'd' .
9	read 'e' .

图 5-3 例 5-18 日志追踪窗口输出结果

```

* 'freeQueue' with QUE_put.
* You can then replace MEM_alloc calls with QUE_get(freeQueue)
* and MEM_free with QUE_put(freeQueue, msg).
*
* A queue can hold an arbitrary number of messages or elements.
* Each message must, however, be a structure with a QUE_Elem as
* its first field.
* /
#include <std.h>
#include <log.h>
#include <mem.h>
#include <que.h>
#include <sys.h>

#define NUMMSG 5          /* number of messages */
typedef struct MsgObj {
    QUE_Elem    elem;      /* first field for QUE */
    Char        val;       /* message value */
} MsgObj, * Msg;

extern QUE_Obj queue;

```

```

/* Trace Log created statically */
extern LOG_Obj trace;

Void reader();
Void writer();

/* ===== main ===== */
Void main()
{
    /*
     * Writer must be called before reader to ensure that the
     * queue is non-empty for the reader.
     */
    writer();
    reader();
}

/* ===== reader ===== */
Void reader()
{
    Msg        msg;
    Int        i;
    for ( i = 0; i < NUMMSGs; i++ ) {

```



```

    /* The queue should never be empty */
    if (QUE_empty(&queue)) {
        SYS_abort("queue error\n");
    }

    /* dequeue message */
    msg = QUE_get(&queue);
    /* print value */
    LOG_printf(&trace, "read '%c'.", msg->val);
    /* free msg */
    MEM_free(0, msg, sizeof(MsgObj));
}

}

/* ===== writer ===== */
Void writer()
{
    Msg        msg;
    Int        i;
    for (i = 0; i < NUMMSGs; i++) {

```

```

    /* allocate msg */
    msg = MEM_alloc(0, sizeof(MsgObj), 0);
    if (msg == MEM_ILLEGAL) {
        SYS_abort("Memory allocation failed!\n");
    }

    /* fill in value */
    msg->val = i + 'a';
    LOG_printf(&trace, "writing '%c'...", msg->val);

    /* enqueue message */
    QUE_put(&queue, msg);
}
}

```

注意：对于 C55x 平台，非指针类型变量在被 LOG\_printf 引用时，需要将其类型转化为 Arg，例如：

```
LOG_printf(& trace, "Task %d Done", (Arg) id);
```

# 第 6 章 I/O 概述和管道

本章对 DSP/BIOS 数据传输的方法进行概述,并重点讨论管道机制。

## 6.1 I/O 概述

在 DSP/BIOS 应用程序中,数据输入和输出可以采用流(stream)、管道(pipe)和主机通道(host channel)三类对象来处理,每类的对象都有其自己的模块来管理数据的输入和输出。

注意:除了使用管道和流,还可以选择 GIO 类驱动与 IOM 微型驱动相配合的形式来处理 I/O 操作。关于 GIO 类驱动和 IOM 微型驱动模型的详细介绍,请查阅第二部分的 DSP/BIOS 驱动开发中的内容。

用户若采用 IOM 微型驱动与流或管道配合使用的方式,也会用到本章中给出的有关流和管道对象的信息。

流(stream)是应用程序和 I/O 设备之间的一种数据交换通道,流可以是只读(输入)或只写的(输出),如图 6-1 所示。流提供了一种简单的、通用的面向所有 I/O 设备的接口,使



图 6-1 输入/输出流

流的一个重要特点是其具有“异步天性”(asynchronous nature),因为其大部分 I/O 函数可阻塞,可以自动等待直到异步数据可用。数据缓冲区的输入或输出和处理是同时进行的。当应用程序处理当前缓冲数据的同时,下一个新的缓冲区会被填满,而先前的一个缓冲区会被输出。这种高效的 I/O 缓冲区管理将数据的复制减少到最小。流交换缓冲区指针而非数据本身,因此减小了应用程序的开销,使应用程序更容易满足实时性要求。

典型的应用程序是首先得到一个输入数据缓冲区,然后对其进行处理,然后输出一个处理后数据的缓冲区,这一过程不断重复直至应用程序终止。

数模转换器、视频捕获器、传感器和 DMA 通道等都是一些常见的 I/O 设备,流模块(SIO)使用设备驱动(由 DEV 模块管理)与这些不同类型的设备进行交互,驱动程序采用统一的 DSP/BIOS 编程接口。

数据管道(pipe)通过数据缓冲实现数据的异步 I/O 操作,它需要使用通知函数来对异步 I/O 操作进行同步,所以使用管道比使用流复杂一些。数据管道提供了一种一致的软件数据结构,用户可以用它驱动所有类型的实时外围设备和 DSP 之间的 I/O 操作,所有在一个管道上进行的 I/O 操作一次只能针对一帧数据进行。虽然每帧的长度固定,但应用程序可以放入数量不等的的数据到帧中,只要不超过其长度。

每个管道只能有一个写入者(writer)和一个读取者(reader),所以只能提供点到点的数据传输。一般来说管道的一端由 HWI 控制,另一端由 SWI 函数控制。管道也可以在两个应用程序线程间传输数据。

主机通道(host channel)对象使得应用程序可以在目标 DSP 和主机之间进行数据通信。主机通道可被静态地配置为输入或输出。实际上每个主机通道都是在内部使用一个数据管道对象来实现的。

## 6.2 管道与流的对比

DSP/BIOS 为数据传输提供了两种不同的模型:管道模型(pipe model)用于 PIP 和 HST 模块,流模型(stream model)用于 SIO 和 DEV 模块。

这两种模型都要求每个管道或流只能有一个读取者线程和一个写入者线程,并且管道

或流在传输缓冲区时都是通过复制指针而非复制缓冲区的数据实现的。

通常,管道模型支持低层的数据通信,而流模型则支持高层的与设备无关的 I/O 操作,两者的详细区别见表 6-1。

表 6-1 管道与流的比较

管道(PIP 和 HST)	流(SIO 和 DEV)
程序员必须创建自己的驱动结构(driver structure)	提供了一个更结构化的途径来创建设备驱动
读线程和写线程可以是任何线程类型或是主机 PC	一端必须由一个任务(TSK)调用 SIO 函数来控制,另一端必须由 HWI 调用 Dxx 函数来控制
PIP 函数是非阻塞的,应用程序必须确保在读写管道之前缓冲区是可用的。读者线程和写者线程的同步需要由通知函数完成	除 SIO_issue 外,SIO_put、SIO_get 和 SIO_reclaim 都是可阻塞函数,会使任务等待直到缓冲区可用,所以流具有“异步天性”,用户不必关心同步问题
所需的存储空间更小,通常更快	更灵活,通常更易于使用
每个通道拥有自己的缓冲区	缓冲区数据可以在流之间传输而无须复制(实际情况下,因为必须对数据进行处理,复制通常是必需的)
管道必须在配置工具中静态创建	流既能在配置工具中静态创建,也能在运行时创建。流能用对象名称打开
本身不支持堆叠(stacking)设备	提供对堆叠(stacking)设备的支持
和 HST 模块结合使用可以很容易地进行主机和目标 DSP 之间的数据传输	为 DSP/BIOS 提供了许多设备驱动

## 6.3 不同驱动模型的比较

在应用程序底层,DSP/BIOS提供了两种设备驱动模型:IOM模型和SIO/DEV模型,使得应用程序可以和DSP的外围设备进行通信。

- ❑ **IOM模型**: IOM模型的组成部分见图6-2。它将硬件无关层和硬件相关层相分离。其中类驱动是和硬件无关的,用于管理设备实例(instance)以及对I/O请求进行同步和串行化,而底层的微型驱动是和硬件相关的。IOM模型可以通过PIO及DIO适配器分别与管道(PIP APIs)和流(SIO APIs)配合使用。关于IOM模型更多的信息,请查看本书第二部分“DSP/BIOS驱动开发手册”。

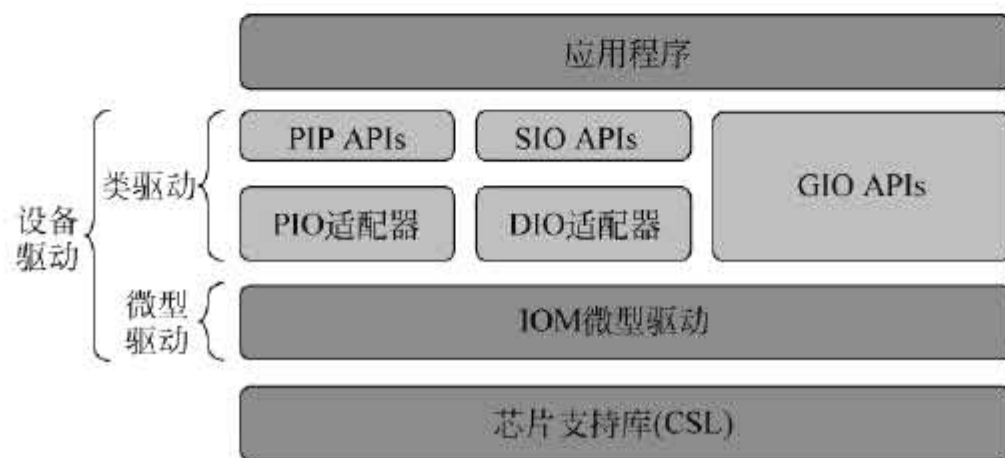


图 6-2 IOM 模型的组成结构

- ❑ **SIO/DEV 模型**：该模型提供了一个流传输(streaming)I/O 接口,应用程序在使用 SIO 模块提供的通用函数时,SIO 函数会间接调用由设备驱动提供的 DEV 函数来管理联结(attached)到流的物理设备。SIO/DEV 模型不能和管道一起使用,第 7 章将对这个模型作更详细地介绍。

对于每种模型,用户都需要使用 DSP/BIOS 的 DEV 模块创建一个自定义的设备对象,该设备对象所使用的模型通过其函数表的类型来识别。函数表若为 IOM\_Fxns 类型则该设备使用的是 IOM 模型,若为 DEV\_Fxns 类型则该设备使用的是 SIO/DEV 模型。

用户可以通过静态配置或通过 DEV\_createDevice 函数动态地创建设备对象,此外还有 DEV\_deleteDevice 和 DEV\_match 函数用于设备对象的管理。

下面的几个小节描述了如何创建使用不同 I/O 驱动对象及模型的设备。关于其中 API 函数调用和配置参数的细节,请查看相应平台的 DSP/BIOS API 函数参考手册。

### 6.3.1 创建一个使用 IOM 微型驱动的设备

如果用户计划采用 IOM 微型驱动和 GIO 类驱动相配合的方式,可以按照如下类似的参数和方法静态地配置一个自定义的设备,或调用 DEV\_createDevice 函数动态地创建一个



```

DEV_Attrs gioAttrs = {
    NULL, /* device id */
    NULL, /* device parameters */
    DEV_IOMTYPE, /* type of the device */
    NULL /* device global data ptr */
};
status = DEV_createDevice("/codec", &DSK6X_EDMA_IOMFXNS,
                          (Fxn)DSK6X_IOM_init, &gioAttrs);

```

## 6.3.2 创建一个使用 SIO 流和 DIO 适配器的设备

如果用户计划采用 IOM 微型驱动和 SIO 流以及 DIO 适配器相互配合的方式,可以按照如下类似的参数和方法静态地配置一个自定义的设备,或调用 DEV\_createDevice 函数动态地创建一个自定义的设备:

```

DIO_Params dioCodecParams = {
    "/codec", /* device name */
    NULL /* chanParams */
};
DEV_Attrs dioCodecAttrs = {

```

```

    NULL, /* device id */
    &dioCodecParams, /* device parameters */
    DEV_SIO_TTYPE, /* type of the device */
    NULL /* device global data ptr */
};

status = DEV_createDevice("/dio_codec",&DIO_tskDynamicFxn,
                        (Fxn)DIO_init,&dioCodecAttrs);

```

在任务线程(TSK)中使用设备时,传递给 DEV\_createDevice 的驱动函数表类型应为 DIO\_tskDynamicFxn,如果在软件中断(SWI)中使用时,则为 DIO\_cbDynamicFxn。

### 6.3.3 创建一个使用 SIO/DEV 模型的设备

如果用户计划采用 SIO/DEV 模型,可以按照如下类似的参数和方法静态地配置一个自定义的设备,或调用 DEV\_createDevice 函数动态地创建一个自定义的设备:

```

DEV_Attrs devAttrs = {
    NULL, /* device id */
    NULL, /* device parameters */
    DEV_SIO_TTYPE, /* type of the device */
    NULL /* device global data ptr */
};

```

```
}  
status = DEV_createDevice("/codec",&DSK6X_EDMA_DEVPXNS,  
                           (Fxn)DSK6X_DEV_init,&devAttrs);
```

传递给 DEV\_createDevice 的设备函数表类型应为 DEV\_Fxns。

### 6.3.4 创建一个使用 DSP/BIOS 提供的软件驱动的设备

DSP/BIOS 提供了多个使用 SIO/DEV 模型的软件驱动,这些软件驱动在相应平台的 DSP/BIOS API 函数参考手册中的 DEV 模块一节有详细描述。创建使用这些驱动的设备对象的方法和创建一个使用 SIO/DEV 模型的设备的方法是一样的。

## 6.4 数据管道管理器(PIP 模块)

管道通过数据缓冲实现异步 I/O 操作。每个管道对象维护着一个缓冲区,并将该缓冲区划分成若干个长度相等的帧,帧的数量和长度分别由 PIP 对象的 numframes 和 framesize 属性指定。所有基于管道的 I/O 操作在一个时刻只能针对一个帧进行,虽然每帧的长度固定,但应用程序装入每帧的数据量可以不等(只要不超过帧长)。

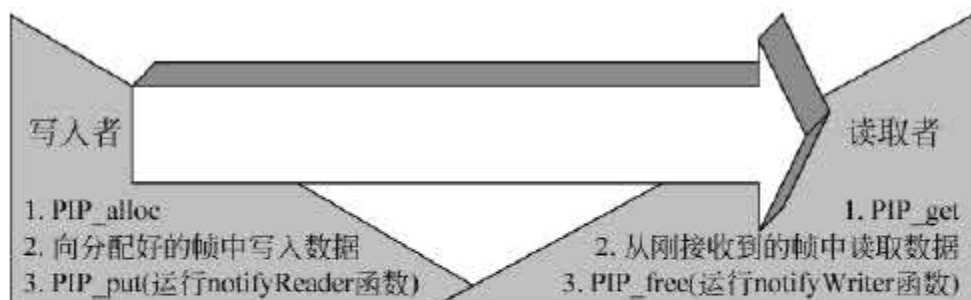


图 6-3 管道的两个端口

数据通知函数(notifyReader 和 notifyWriter)用于对数据传输进行同步。用户可以为每个管道配置自定义的通知函数。当一帧数据被读出或被写入时,这些函数将被触发以告知应用程序有一帧已空闲或有一帧数据可用。通知函数是在调用 PIP\_free、PIP\_put、PIP\_get 或 PIP\_alloc 的线程的环境中执行的。当 PIP\_get 被调用时,DSP/BIOS 检查当前管道中是否还有更多的满帧,如果有则执行 notifyReader 函数通知读取者线程;同样当调用 PIP\_alloc 时,DSP/BIOS 检查当前管道中是否有空帧存在,如果有则执行 notifyWriter 函数通知写入者线程。

一个管道只应有一个读取者和一个写入者。通常管道的一端由 HWI 控制,另一端则由软件中断函数来控制。管道也能用于程序中两个应用线程之间的数据传输。

在 DSP/BIOS 程序启动时(详见 2.7 节“DSP/BIOS 启动序列”),BIOS\_start 函数会启用 DSP/BIOS 模块,在这一步,PIP\_startup 函数会为每个管道对象调用 notifyWriter 函数,因为在启动时,所有的管道都有空帧可以使用。

由管道传输的数据并没有特殊的格式要求和类型要求,DSP/BIOS 在线帮助描述了数据管道对象及其参数。有关 PIP 模块 API 函数的详细信息,请查看相应平台的 DSP/BIOS API 函数参考手册。

## 6.4.1 写入数据到管道

应用程序应按照如下步骤向管道写入数据:

(1) 应用程序首先应该检查管道中空帧的数量,为此必须调用 PIP\_getWriterNumFrames 函数,该函数会返回一个管道对象中空帧的数目。

(2) 如果空帧的数量大于 0,应用程序可调用 PIP\_alloc 从管道中获取一个空帧。

(3) 在函数 PIP\_alloc 返回之前,DSP/BIOS 将自动检查管道中是否有更多可用的空帧,如果有,notifyWriter 函数将在这时被自动调用。

(4) 一旦 PIP\_alloc 返回,应用程序就可以将数据写入得到的空帧中。为此首先需要知道该帧的起始地址和大小。调用 API 函数 PIP\_getWriterAddr 可以得到其起始地址,调用 PIP\_getWriterSize 可以得到能写入的数据量(以 word 为单位),对于一个空帧,该函数的默认返回值就是静态配置好的帧长。

(5) 数据写完后,就可以将该帧返回给管道。如果写入的数据长度小于帧长,应用程序可以通过调用 PIP\_setWriterSize 函数向管道指出写入数据的长度,然后调用 PIP\_put 将数据帧返回给管道。

(6) 调用 PIP\_put 函数会触发 notifyReader 函数的执行,通知读取线程管道中当前有可以读取的有效数据。

例 6-1 说明了向管道写入数据的过程。

#### 例 6-1 向管道写入数据

```
extern far PIP_Obj writerPipe;          /* pipe object created with the Configuration Tool */
writer()
{
    Uns size;
    Uns newsize;
    Ptr addr;

    if (PIP_getWriterNumFrames(&writerPipe) > 0) {
        PIP_alloc(&writerPipe);          /* allocate an empty frame */
    }
    else {
        return;                          /* There are no available empty frames */
    }
}
```

```

        PIP_getWr:
size = PIP_getWr:
' fill up the frame

/* optional */
newsize = 'number of words written to the frame';
PIP_setWriterSize(&writerPipe, newsize);

/* release the full frame back to the pipe */
PIP_put(&writerPipe);
}

```

## 6.4.2 从管道中读取数据

应用程序从管道读取一个满帧需要执行的步骤如下：

(1) 程序首先应先调用 PIP\_getReaderNumFrames 函数检查管道中能被读取的满帧的数量，该函数调用后返回一个管道对象中满帧的个数。

(2) 如果满帧的个数大于 0，应用程序可调用 PIP\_get 从管道中获得一个满帧。

(3) 从函数 PIP\_get 返回之前，DSP/BIOS 会自动检查管道中是否有更多可用的满帧。如果有，则在这时调用执行 notifyReader 函数。

(4) 一旦 PIP\_get 返回,应用程序就可以读取该满帧中的数据了。为此首先需要知道该满帧的起始地址和大小,调用 PIP\_getReaderAddr 函数获得该满帧的起始地址,调用 PIP\_getReaderSize 获得该满帧的有效数据字数(words)。

(5) 当应用程序读出所有数据后,接着调用 PIP\_free 将这一帧返回给管道。

(6) 调用 PIP\_free 会触发 notifyWriter 函数的执行,通知写入线程当前在管道中有一个新的空帧可以使用了。

例 6-2 说明了从管道中读取数据的过程。

#### 例 6-2 从管道中读取数据

```
extern far PIP_Obj readerPipe;          /* created with the Configuration Tool */
reader()
{
    Uns size;
    Ptr addr;
    if (PIP_getReaderNumFrames(&readerPipe) > 0) {
        PIP_get(&readerPipe);           /* get a full frame */
    }
    else {
        return;                          /* There are no available full frames */
    }
}
```



```

    addr = PIP_getReaderAddr(&readerPipe);
    size = PIP_getReaderSize(&readerPipe);
    ' read the data from the frame '
    /* release the empty frame back to the pipe */

    PIP_free(&readerPipe);
}

```

### 6.4.3 使用管道的通知函数

管道的读取和写入线程可以工作在查询方式下,即在获得新的满帧或空帧之前直接检测可用的满帧或空帧的数目。上面的例 6-1 和例 6-2 就是这种查询方式的写和读的操作。

当管道对象用于缓冲外围硬件设备写入(或读取)的实时 I/O 数据时,由于外设本身通常会触发一个 HWI 例程(即 ISR),所以该管道对象常常被作为该 HWI 例程和应用程序读取(或写入)函数两端之间的数据通道。这时,用户可以为管道配置自定义的 notifyReader(或 notifyWriter)函数使其自动触发一个软件中断,由该软件中断负责运行数据的读取(或写入)函数,从而实现应用程序与底层 I/O 数据的有效同步。

当 HWI 例程完成写入(或读取)一帧数据的操作并调用 PIP\_put(或 PIP\_free)时,管道的通知函数自动触发一个软件中断。这时只需在软件中断被触发时运行读取(或写入)函数即可。也就是当管道中的帧可读(或可写)时,管道的读取(或写入)函数会自动运行,而不必

随时查询管道。

用户按照上述方法使用通知函数后,在执行读取(或写入)函数之前就不用检查管道中帧的有效性了。但为了做好预防措施,可以在 SWI 函数一开始保留对管道中数据帧的检测,并在帧无效时执行错误处理函数而非读取(或写入)函数。例如:

```
if (PIP_getReaderNumFrames(&readerPipe) == 0) {  
    error(); /* reader function should not have been posted! */  
}
```

综上所述,管道对象的通知函数实质上可以作为应用程序线程和硬件设备之间的一种流量控制机制。

#### 6.4.4 PIP 模块 API 函数的调用顺序

在每个管道对象的内部,为管道写入端维护着一个空帧列表和一个空帧个数计数器,也为管道读取端维护着一个满帧列表和一个满帧个数计数器。另外管道对象中还包含一个当前写入帧和当前读取帧的描述符(descriptor),当前写入帧也就是最后一个被应用程序分配并正在填充的帧,当前读取帧也就是最后一个被应用程序获得并正在读取的满帧。

当 PIP\_alloc 被调用时,写入端计数器减 1。一个空帧会从写入端列表中删除,并且写入

帧描述符会根据该帧的信息进行更新。当应用程序完成该帧填充之后调用 PIP\_put 函数时,读取端计数器加 1 并且当前写入帧描述符会被 DSP/BIOS 使用,以将这个新的满帧添加到管道的读取端列表中。

注意:每次调用 PIP\_alloc 后,必须在下次调用 PIP\_alloc 之前调用 PIP\_put,因为管道的 I/O 机制不允许连续调用 PIP\_alloc,否则会覆盖掉之前的描述符信息并导致不确定的后果,见例 6-3。

### 例 6-3 使用 PIP\_alloc

<code>/* correct */</code>	<code>/* error! */</code>
<code>PIP_alloc();</code>	<code>PIP_alloc();</code>
<code>...</code>	<code>...</code>
<code>PIP_put();</code>	<code>PIP_alloc();</code>
<code>...</code>	<code>...</code>
<code>PIP_alloc();</code>	<code>PIP_put();</code>
<code>...</code>	<code>...</code>
<code>PIP_put();</code>	<code>PIP_put();</code>

类似地,当 PIP\_get 被调用时,读取端计数器将减 1 并且一个满帧会从读取端列表中删除,读取帧描述符会根据当前读取帧的信息进行更新。当应用程序在读取该帧之后调用 PIP\_free 时,写入端计数器加 1 并且当前读取帧描述符会被 DSP/BIOS 使用,以将这个新的空帧添加到管道的写入端列表中。

管道的 I/O 机制同样不允许连续的 PIP\_get 调用,见例 6-4。

#### 例 6-4 使用 PIP\_get

<pre>/* correct */ PIP_get();  ... PIP_free();  ... PIP_get();  ... PIP_free();</pre>	<pre>/* error! */ PIP_get();  ... PIP_get();  ... PIP_free();  ... PIP_free();</pre>
---	--

#### 6.4.4.1 避免递归问题

当在通知函数中对其所服务的管道调用 PIP API 函数时需要特别注意避免递归问题。

下面以这个例子进行说明:一个 notifyReader 函数对其所服务的管道调用了 PIP\_get 和 PIP\_free 函数。该管道的读取端是一个 HWI 硬件中断服务程序,其写入端是一个 SWI 服务程序。

管道的读取函数(HWI 服务程序)会调用 PIP\_get 和 PIP\_free 从管道中读取并随后释放数据帧;管道的写入函数(SWI 函数)会调用 PIP\_put, PIP\_put 会调用 notifyReader 函数,所以如果用户通过配置在 notifyReader 函数中直接执行 PIP\_get 和 PIP\_free 函数来进

行一次读取端的读取操作,则很有可能发生这种情况:当 SWI 线程刚刚执行完 PIP\_get 后,就被 HWI 线程抢占,这时 PIP\_get 将会在调用 PIP\_free 之前被调用两次。如果这种情况发生将会产生灾难性的后果。

注意:为了避免这种递归问题,作为惯例,用户应该避免在 notifyReader 和 notifyWriter 中调用 PIP 函数,如果必须这样做以保证应用程序的高效性的话,这样的调用应该采取保护措施以避免对同一个管道对象进行,以及避免错误的 PIP API 函数调用次序。

## 6.5 主机通道管理器(HST 模块)

HST 模块用于管理主机通道对象,主机通道使得应用程序可以在目标 DSP 和主机之间传输数据。主机通道可以配置成输入或输出。输入通道从主机读取数据到目标 DSP,输出通道从目标 DSP 传输数据到主机。

注意: HST 通道名不能以下划线“\_”作为开头。

用户可以在 CCS 中动态地将主机通道绑定到主机的文件上,然后开始数据的传输。如图 6-4 所示。

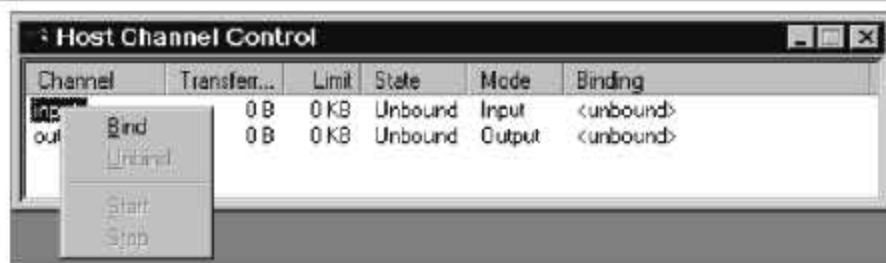


图 6-4 绑定主机通道

每个主机通道在内部使用一个管道对象实现。为了使用一个特定的主机通道,应用程序调用 `HST_getpipe` 获得主机通道中的管道对象,然后通过调用 `PIP_get` 和 `PIP_free` 实现输入操作,或通过调用 `PIP_alloc` 和 `PIP_put` 实现输出操作。

通过主机通道读取数据的代码类似于例 6-5。

#### 例 6-5 通过主机通道读取数据

```
extern far HST_Obj input;
readFromHost()
{
    PIP_Obj * pipe;
    Uns size;
    Ptr addr;
    pipe = HST_getpipe(&input) /* get a pointer to the host
                                channel's pipe object */
    PIP_get(pipe);             /* get a full frame from the host */
}
```

```

    size = PIP_getReaderSize(pipe);
    addr = PIP_getReaderAddr(pipe);
    'read data from frame'
    PIP_free(pipe);          /* release empty frame to the host */
}

```

用户可以为每个主机通道都指定一个数据通知函数,当一个数据帧可用时(对输入通道)或一个数据帧空闲时(对输出通道),该函数就会被调用执行。即该函数会在主机写入或读取一帧数据时被触发。

HST 通道会根据不同的 DSP 平台在主机文件中保存 16 位或 32 位字的原始数据。具体的数据格式是由应用程序指定的,用户需要保证主机和目标 DSP 都采用相同的格式来组织数据。例如,如果用户从主机端读取 32 位的整型数据,则需要保证主机文件中的数据采用正确的字节顺序(byte order)组织。除正确的字节顺序外,主机和目标 DSP 之间传输数据再没有其他的格式或类型上的要求。

在开发应用程序时,用户可以使用 HST 对象来仿真数据流,来测试算法对预定义数据的处理结果。在开发的初级阶段,特别是验证数字信号处理算法时,应用程序往往需要使用输入通道访问主机文件中的数据以此作为算法的输入,并使用输出通道将运算结果记录到文件中,通过比较输出文件和期望的结果就可以判断算法的正确性。在程序开发的后期,如果算法看起来可行,用户可以用 PIP 对象代替 HST 对象与其他线程或实际的 I/O 驱动进行通信,以实现产品功能。

## 6.5.1 传输 HST 数据到主机

尽管目标主机间数据流的实时传输所能使用的带宽最终决定于实际物理链路的选择，但 HST 通道接口仍能保持与物理链路的独立无关性。HST 管理器允许用户在配置工具中选择可用的物理连接。

对于 C54x 平台，目标 DSP 和主机端间实际的数据传输运行在最低优先级的空闲循环中。在 C55x 和 C6000 平台上，则是由主机触发一个中断来进行与目标 DSP 之间的数据传输，且该中断拥有比 SWI、TSK 以及 IDL 函数更高的优先级。该中断的 ISR 函数在很短的时间内运行完成，而准备 RTDX(实时数据交换)缓冲区和执行 RTDX 调用等更耗时的工作则在空闲循环中由函数 LNK\_dataPump 来进行，只有实际的数据传输才会高优先级的中断函数里执行。所以这种数据传输对实时操作的影响很小，特别是在需要传输大量 LOG 数据的时候。

## 6.6 I/O 性能问题

当用户使用 HST 对象时，主机对数据的读或写是通过使用 LNK\_dataPump 对象指定的函数来完成，该对象是一个自动创建的 IDL 对象，其函数在后台线程中运行。由于后台线



程的优先级最低,所以在 C54x 平台上,软件中断和硬件中断可以抢占数据的传输;而在 C55x 和 C6000 平台上,实际的数据传输则在高优先级上运行。

用户在 LOG、STS 和 TRC 的控制面板中设置的轮询率 (polling rates) 不是用来控制 HST 对象数据传输的速率,更快的轮询率实际上只会减慢 HST 对象的数据传输速率,这是因为所需的 LOG、STS 和 TRC 数据传输占用了更多的带宽。

# 第 7 章 流 I/O 和设备驱动

本章涉及采用 DEV\_Fxns 模型的设备驱动,描述了该类设备驱动的编写和使用方法,并给出了一些程序示例。

## 7.1 流 I/O 和设备驱动概述

注意:这一章描述的设备都使用 DEV\_Fxns 结构体类型的函数表。另外,本书第二部分“DSP/BIOS 驱动开发手册”中则描述了一种新的设备驱动模型——IOM 模型,该模型使用 IOM\_Fxns 结构体类型的函数表,其中还对如何创建 IOM 微型驱动器和如何将 IOM 微型驱动整合到用户应用程序中进行了介绍。

如果用户选择 IOM 微型驱动与 SIO 流一起使用,那么本章介绍的有关使用 SIO 流的信息也会对用户有所帮助。

第 6 章从有利于应用程序的方面描述了 DSP/BIOS 所支持的与设备无关的 I/O 操作,指出流提供了一种简单的、通用的面向所有 I/O 设备的接口,使得应用程序可以完全忽略具体 I/O 设备的操作细节。实际上,应用程序在使用 SIO 模块提供的对所有设备都通用的 API 函数时,会间接地触发执行相应的由设备驱动实现的函数,而设备驱动则负责管理联结(attach)到流的具体的物理设备。本章将从流与设备驱动接口层面上描述 DSP/BIOS 中设备无关的 I/O 操作,如图 7-1 所示。

DEV 设备驱动函数用于控制 SIO 模块所联结的各个设备对象。与其他 DSP/BIOS 模块不同,用户应用程序不需要直接调用这些驱动函数。相反,每个驱动模块会给出一个有特定名称的 DEV\_Fxns 类型的结构体(函数表),当应用程序调用 SIO 通用 API 函数时,SIO 模块会使用该函数表触发相应的驱动函数。

如表 7-1 所示,每个 SIO 操作都是通过对照下表来调用相应的驱动函数的。Dxx 代表用户为特定设备编写的驱动。

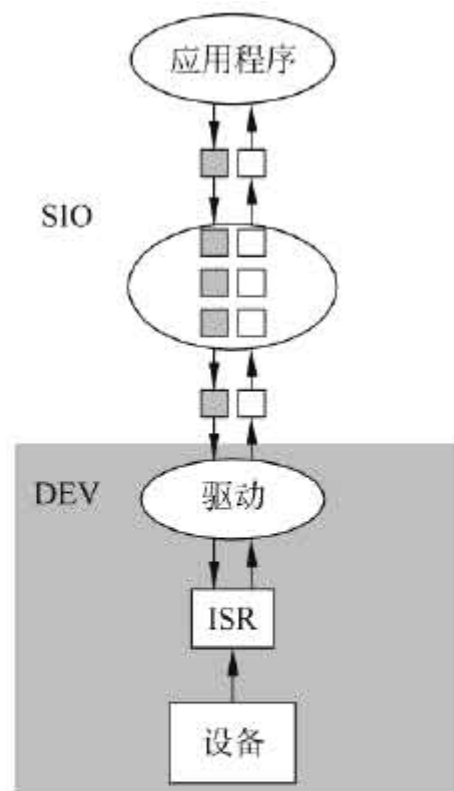


图 7-1 DSP/BIOS 中与设备无关的 I/O 操作

表 7-1 通用 I/O 操作到内部驱动操作

通用 I/O 操作	内部驱动操作
SIO_create(name,mode,bufsize,attrs)	Dxx_open(device,name)
SIO_delete(stream)	Dxx_close(device)
SIO_get(stream,&buf)	Dxx_issue(device) 和 Dxx_reclaim(device)
SIO_put(stream,&buf,nbytes)	Dxx_issue(device) 和 Dxx_reclaim(device)
SIO_ctrl(stream,cmd,arg)	Dxx_ctrl(device,cmd,arg)
SIO_idle(stream)	Dxx_idle(device,FALSE)
SIO_flush(stream)	Dxx_idle(device,TRUE)
SIO_select(streamtab,n,timeout)	Dxx_ready(device,sem)
SIO_issue(stream,buf,nbytes,arg)	Dxx_issue(device)
SIO_reclaim(stream,&buf,&arg)	Dxx_reclaim(device)
SIO_staticbuf(stream,&buf)	无

事实上,DSP/BIOS 所提供的所有功能,从多任务处理到应用级的服务,都能够被这些内部驱动函数所使用。设备驱动还能够通过 DSP/BIOS 提供的与设备无关的 I/O 接口来间接地与其他设备驱动进行通信,尤其是在支持可堆叠(stackable)设备的时候。

图 7-2 描述了硬件设备、Dxx 设备驱动和从该设备接收数据的流(输入流)之间的关系。SIO 操作会自动调用 DEV\_Fxns 结构体(即该设备的函数表)中列举的 Dxx 函数。原子队

列 device->todevice 和  
被设备填满的数据帧。

对每个设备驱动,用户都要自己编写函数 Dxx\_open, Dxx\_idle, Dxx\_input, Dxx\_output, Dxx\_close, Dxx\_ctrl, Dxx\_ready, Dxx\_issue 和 Dxx\_reclaim。

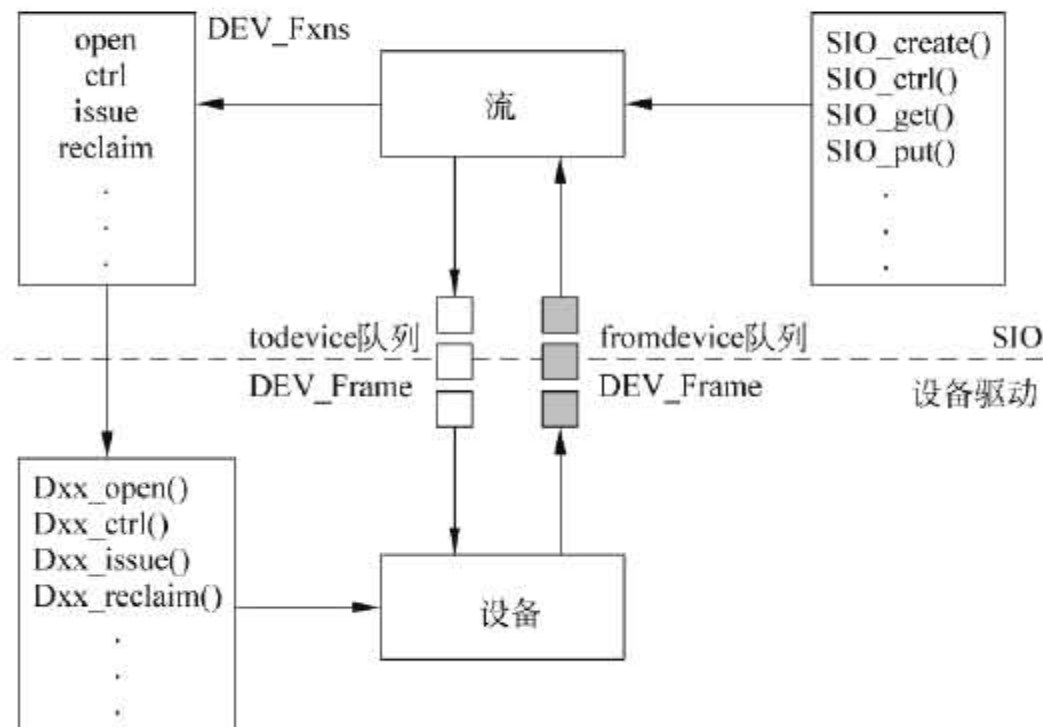


图 7-2 设备、驱动和流的关系

## 7.2 创建和删除流

为了使用户应用程序能够对一个设备实现流输入/输出,首先必须把该设备添加到 DSP/BIOS 配置中。用户可以为任何 TI 提供的或用户开发的设备驱动添加一个或多个设备对象。配置好设备后,还需要在配置工具中静态创建或者在运行时调用 SIO\_create 函数来动态创建一个流对象。

### 7.2.1 静态创建流对象

用户可以在 DSP/BIOS 配置中创建流对象,并设置每个流对象的属性以及 SIO 管理器自身的属性,静态创建的流对象不能用 SIO\_delete 函数来删除。

### 7.2.2 动态创建和删除流对象

用户也可以在程序运行时用 SIO\_create 函数创建一个流对象,其函数接口见例 7-1。

**例 7-1** 用 SIO\_create 创建流对象

```
SIO_Handle SIO_create(name, mode, bufsize, attrs)
    String      name;
```

```
Int          mode;  
Uns          bufsize;  
SIO_Attrs    * attrs;
```

SIO\_create 函数会创建一个流并返回一个 SIO\_Handle 类型的句柄。SIO\_create 函数会触发 DEV 函数打开由 name 参数指定的一个或多个设备,并会根据参数 bufsize 分配相应大小的缓冲区。可选的属性参数(attrs)用于指定缓冲区的数量,缓冲区所在的存储器段(memory segment)和流传输模型(streaming model)等信息。参数 mode 用来指定该流是一个输入流(取值 SIO\_INPUT)还是一个输出流(取值 SIO\_OUTPUT)。

注意:参数 name 必须和在配置中为设备对象设置的名称一致,且必须在前面加上一个斜杠“/”。如对于一个名为 sine 的设备,对应的 name 参数应该为“/sine”。

在创建流对象时,如果将流传输模型(attrs→model)设置为 SIO\_STANDARD(默认),SIO\_create 函数会自动分配指定大小的缓冲区给该流对象,为流的启动做好准备。如果用户将流传输模型设置为 SIO\_ISSUERECLAIM,则不会自动分配缓冲区,这时流所需的缓冲区由用户应用程序负责提供。

SIO\_delete 函数会关闭流对象中联结的所有设备并释放掉该流对象。如果该流对象的流传输模型是 SIO\_STANDARD,则 SIO\_delete 函数也会释放所有保留在流中的缓冲区,而

由用户自行提供的流缓冲区必须由用户代码来显式释放(针对 SIO\_ISSUERECLAIM 流传输模型)。SIO\_delete 函数接口见例 7-2。

**例 7-2** 使用 SIO\_delete 删除流对象

```
Int SIO_delete(stream)
    SIO_Handle stream;
```

## 7.3 流 I/O——读入流和写出流

DSP/BIOS 中数据的流传输有两种模型：标准(Standard)模型和发放/回收(Issue/Reclaim)模型。标准模型提供了使用流的简单方法，而发放/回收模型提供了更多的对流操作的控制。

函数 SIO\_get 和 SIO\_put 用来实现标准流传输模型，见例 7-3。函数 SIO\_get 用于输入流，它通过流中的缓冲区交换，使客户程序从设备端得到一个存放有输入数据的缓冲区。参数 bufp 用来给设备传递一个空缓冲区地址并承载输入缓冲区地址供客户程序使用。函数 SIO\_get 的返回值为得到的输入缓冲区中有效数据的字节数。函数 SIO\_put 用于输出流，它和 SIO\_get 一样也是通过流中的缓冲区交换，输出一个数据缓冲区给设备，并得到一个空缓冲区，参数 nbytes 指定了输出缓冲区中有效数据的字节数。



### 例 7-3 输入和输出数据缓冲区

```
Int SIO_get(stream, bufp)
    SIO_Handle    stream;
    Ptr           * bufp;
Int SIO_put(stream, bufp, nbytes)
    SIO_Handle    stream;
    Ptr           * bufp;
    Uns           nbytes;
```

注意：由于 bufp 所指向的缓冲区将在流中进行交换，该缓冲区的大小、所在存储器段和边界条件必须符合流的相应属性。

函数 SIO\_issue 和 SIO\_reclaim 用于实现发放/回收 (Issue/Reclaim) 流传输模型，见例 7-4。SIO\_issue 用来传递一个缓冲区给流。该 SIO 操作不返回任何缓冲区，并且流会在不产生阻塞的情况下将控制权交还给客户端。DSP/BIOS 自身不会解析参数 arg，但是它为流的客户端（比如一个任务线程）提供了一种和设备通信的服务。arg 会和相关的缓冲区数据一同被传递到每一个设备。流的客户端可以将它作为一种和设备驱动通信的方法来使用。例如，可以用 arg 给输出设备发送一个时间戳 (time stamp)，准确地指出该缓冲区中的数据应在什么时候被输出。

函数 SIO\_reclaim 用来请求流返回一个缓冲区，即从流中回收一个缓冲区。如果没有可

返回的缓冲区,流就会阻塞任务的执行直到有缓冲区可返回或者流的超时时间用完。

#### 例 7-4 实现发放/回收(Issue/Reclaim)流传输模型

```
Int SIO_issue(stream,pbuf,nbytes,arg)
```

```
    SIO_Handle    stream;
```

```
    Ptr           pbuf;
```

```
    Uns           nbytes;
```

```
    Arg           arg;
```

```
Int SIO_reclaim(stream,bufp,parg)
```

```
    SIO_Handle    stream;
```

```
    Ptr           * bufp;
```

```
    Arg           * parg;
```

标准模型和发放/回收模型之间最明显的区别在于发放/回收模型把缓冲区到达流的通知(SIO\_issue)和对流中缓冲区可返回的等待(SIO\_reclaim)分离开。所以,函数对 SIO\_issue/SIO\_reclaim 提供了和调用 SIO\_get(或 SIO\_put)一样的缓冲区交换。

发放/回收流传输模型通过允许流的客户端在运行时控制发放缓冲区的数目,从而提供了更大的灵活性。通过调用 SIO\_issue,客户端可以向流发送多个缓冲区且不会产生阻塞。通过调用 SIO\_reclaim,流可以根据客户端的请求返回缓冲区。这就使得客户端能够适应一个设备的缓冲深度,还可决定何时阻塞并等待一个缓冲区的返回。

此外通过保证客户端的缓冲区按照其发放的顺序被回收,发放/回收流传输模型在缓冲

区管理上也具有更大的决定权,这就允许客户端为流传输使用任何来源的内存。例如,如果一个 DSP/BIOS 任务接收到一个大的缓冲区,该任务可以将其分成小块传递给流——仅仅通过一个指针在大缓冲区中的逐步推进同时并不断调用 SIO\_issue 即可实现。这种方法可行是因为缓冲区的每一小块能够保证按照送出时的顺序返回。

### 7.3.1 缓冲区交换

DSP/BIOS 里流传输模型的一个重要部分就是缓冲区交换。要提供低开销的高效的 I/O 操作,DSP/BIOS 在某些 I/O 操作中尽力避免了数据从一处到另一处的复制。所以 DSP/BIOS 在使用 SIO\_get、SIO\_put、SIO\_issue、SIO\_reclaim 时只是将缓冲区指针在流和设备之间移动。图 7-3 给出了 SIO\_get 的工作原理。

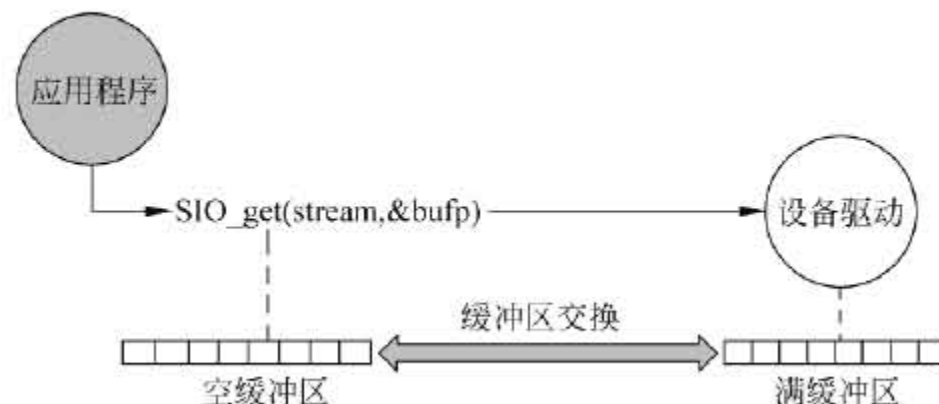


图 7-3 SIO\_get 工作原理

如图 7-3 所示,在数据可用时,和流相联结的设备驱动对一个缓冲区进行填充。此时,应用程序正在处理当前缓冲区。当应用程序调用 `SIO_get` 获取下一个输入缓冲区时,被输入设备填满的新缓冲区和一个缓冲区相交换。这是通过交换缓冲区指针而非通过很耗时的数据复制来实现的。因此,函数 `SIO_get` 的开销和缓冲区长度无关。

每次调用 `SIO_get` 后实际得到的物理缓冲区都会改变,所以用户要保证 I/O 操作中任何对缓冲区的引用都必须在每次操作后得到更新,否则就会使引用出错。

`SIO_put` 用同样的交换指针的方法来在输出流中交换缓冲区。`SIO_issue` 和 `SIO_reclaim` 每次都只向一个方向传递指针,因此 `SIO_issue/SIO_reclaim` 函数对同样可以实现缓冲区指针的交换。

注意:一个流不能同时被几个任务使用。就是说,对用户应用程序中的每一个流,一次只能有一个任务调用 `SIO_get/SIO_put` 或者 `SIO_issue/SIO_reclaim` 来使用它。

### 7.3.2 例子——从 DGN 设备读取输入缓冲区

例 7-5 中的程序举例说明了一些基本的 SIO 函数使用方法并提供了一个从流中读取缓冲区的简单易懂的例子。对于由 DSP/BIOS 提供的 DGN 设备驱动的完整描述,请参考相应平台的 TMS320 DSP/BIOS API 函数参考手册中有关 DGN 的章节。

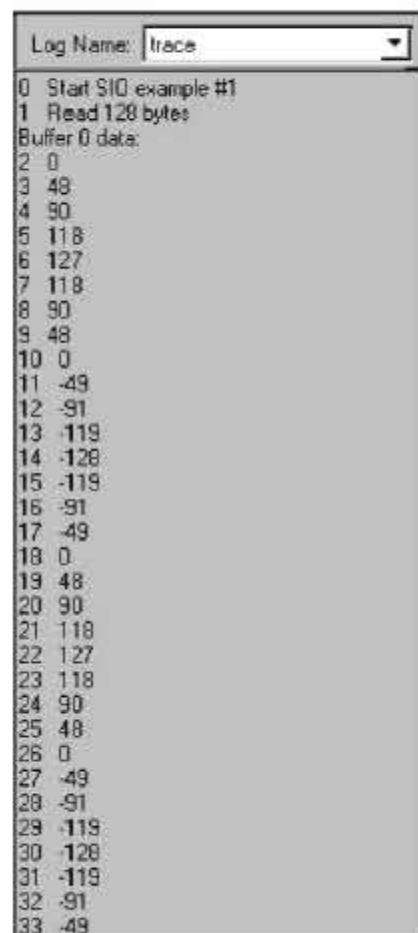
例 7-5 的 DSP/BIOS 配置模板可以在目录 `...\ti\tutorial\target\sioctest` 中找到, `target` 代表用户选择的平台。该例中,一个名为 `sineWave` 的 DGN 设备被用作 SIO 流 `inputStream` 的数据

发生器,任务 streamTask 调用函数 doStreaming 函数从 inputStream 中读取正弦数据并将其打印到日志缓冲区进行程序追踪。图 7-4 中显示出的正弦波形数据为该例的日志追踪输出。

### 例 7-5 基本的 SIO 函数

```
/*
 * ===== sioTest1.c =====
 *
 * In this program a task reads data from a DGN sine device
 * and prints the contents of the data buffers to a log buffer.
 * The data exchange between the task and the device is done
 * in a device independent fashion using the SIO module APIs.
 *
 * The stream in this example follows the SIO __STANDARD
 * streaming
 * model and is created using the Configuration Tool.
 */

#include <std.h>
#include <log.h>
#include <sio.h>
#include <sys.h>
#include <tsk.h>
```



```
Log Name: trace
0 Start SIO example #1
1 Read 128 bytes
Buffer 0 data:
2 0
3 48
4 90
5 118
6 127
7 118
8 90
9 48
10 0
11 -49
12 -91
13 -119
14 -128
15 -119
16 -91
17 -49
18 0
19 48
20 90
21 118
22 127
23 118
24 90
25 48
26 0
27 -49
28 -91
29 -119
30 -128
31 -119
32 -91
33 -49
```

图 7-4 例 7-5 的日志追踪输出

```

extern Int IDRAM1;          /* MEM segment ID defined by Conf tool */
extern LOG_Obj trace;       /* LOG object created with Conf tool */
extern SIO_Obj inputStream; /* SIO object created w Conf tool */
extern TSK_Obj streamTask;  /* pre-created task */

SIO_Handle input = &inputStream; /* SIO handle used below */
Void doStreaming(Uns nloops);     /* function for streamTask */

/*
 * ===== main =====
 */
Void main()
{
    LOG_printf(&trace, "Start SIO example #1");
}

/*
 * ===== doStreaming =====
 * This function is the body of the pre-created TSK thread
 * streamTask.
 */
Void doStreaming(Uns nloops)

```

```

{
    Int i,j,nbytes;
    Int * buf;
    status = SIO_staticbuf(input,(Ptr *)&buf);
    if (status != SYS_ok) {
        SYS_abort("could not acquire static frame: ");
    }

    for (i = 0; i < nloops; i++) {
        if ((nbytes = SIO_get(input,(Ptr *)&buf)) < 0) {
            SYS_abort("Error reading buffer %d",i);
        }
        LOG_printf(&trace,"Read %d bytes\nBuffer %d data: ",
            nbytes,i);
        for (j = 0; j < nbytes / sizeof(Int); j++) {
            LOG_printf(&trace," %d",buf[j]);
        }
    }
    LOG_printf(&trace,"End SIO example # 1");
}

```

### 7.3.3 例子——对 DGN 设备的读和写

例 7-6 在上一个例子的基础上添加了一个新的 SIO 操作，配置中添加了一个输出流 outputStream。streamTask 仍然像前面一样从 DGN 正弦设备中读取缓冲区，但是这次它不是将数据内容打印到记录缓冲区而是把数据缓冲区发送给 outputStream。流 outputStream 则把数据发送给一个名为 printData 的 DGN 用户设备。该设备接收数据缓冲区并用函数 DGN\_print2log 将数据打印到日志缓冲区中，日志缓冲区可在配置工具中由用户进行详细设置。

例 7-6 给例 7-5 加上一个输出流

```
===== Portion of siotest2.c =====  
/* SIO objects created with conf tool */  
extern far LOG_Obj trace;  
extern far SIO_Obj inputStream;  
extern far SIO_Obj outputStream;  
extern far TSK_Obj streamTask;  
SIO_Handle input = &inputStream;  
SIO_Handle output = &outputStream;  
  
...  
Void doStreaming(Arg nloops_arg)
```



```

{
    Int i,nbytes;
    Int * buf;

    Long nloops = (Long) nloops_arg;
    if (SIO_staticbuf(input,(Ptr *)&buf) == 0) {
        SYS_abort("Error reading buffer ");
    }
    for (i = 0; i < nloops; i++) {
        if ((nbytes = SIO_get(input,(Ptr *)&buf)) < 0) {
            SYS_abort("Error reading buffer % d",(Arg)i);
        }
        if (SIO_put(output,(Ptr *)&buf,nbytes) < 0) {
            SYS_abort("Error writing buffer % d",(Arg)i);
        }
    }
    LOG_printf(&trace,"End SIO example # 2");
}

/* ===== DGN_print2log =====
 * User function for the DGN user device printData. It takes as an

```

```
* argument the address of the LOG object where the data stream  
* should be printed  
* /
```

```
Void DGN_print2log(Arg arg, Ptr addr, Uns nbytes)  
{  
    Int i;  
    Int * buf;  
    buf = (Int *)addr;  
    for (i = 0; i < nbytes/sizeof(Int); i++) {  
        LOG_printf((LOG_Obj *)arg, " %d", buf[i]);  
    }  
}
```



注意：对于 C55x 平台，LOG\_printf 的非指针型参数需要用 (Arg) 对其进行类型转换，如下面代码所示：

```
LOG_printf(&trace, "Task %d Done", (Arg)id);
```

例 7-6 的完整源代码和 DSP/BIOS 配置模板 (siotest2.c, siotest2.cdb, dgn\_print.c) 可以在目录... \ti\tutorial\target\siotest 中找到，target 代表用户选择的平台。至于如何在配置工具中添加并配置一个 DGN 设备，请参考相应平台的 TMS320 DSP/BIOS API 函数参考手册中有关 DGN 的章节。

这个例子的输出结果见图 7-5，正弦波形数据这次在 myLog 日志窗口显示出来。

Log Name: myLog	
0	0
1	48
2	90
3	118
4	127
5	118
6	90
7	48
8	0
9	-49
10	-91
11	-119
12	-128
13	-119
14	-91
15	-49
16	0
17	48
18	90
19	118
20	127
21	118
22	90
23	48
24	0

图 7-5 例 7-6 的日志追踪窗口

### 7.3.4 例子——使用发放/回收模型的流 I/O

例 7-7 和例 7-6 在实现的功能上完全相同，所不同的是这里的流采用发放/回收 (Issue/Reclaim) 模型创建，读和写的 SIO 操作使用 SIO\_reclaim 和 SIO\_issue 函数。

在这个模型中，当动态创建流之后并没有初始化分配相应的缓冲区，所以应用程序必须

分配必需的缓冲区并将其提供给流来进行 I/O 操作。对于静态创建的流对象,用户可以在配置工具中选中“Allocate Static Buffer(s)”一项来为 SIO 对象分配静态的缓冲区。

#### 例 7-7 使用发放/回收模型

```
/* ===== doIRstreaming ===== */  
Void doIRstreaming(Uns nloops)  
{  
    Ptr    buf;  
    Arg    arg;
```

```

/* Prime the stream with a couple of buffers */
buf = MEM_alloc(IDRAM1, SIO_bufsize(input), 0);
if (buf == MEM_ILLEGAL) {
    SYS_abort("Memory allocation error");
}

/* Issue an empty buffer to the input stream */
if (SIO_issue(input, buf, SIO_bufsize(input), NULL) < 0) {
    SYS_abort("Error issuing buffer %d", i);
}
buf = MEM_alloc(IDRAM1, SIO_bufsize(input), 0);
if (buf == MEM_ILLEGAL) {
    SYS_abort("Memory allocation error");
}

for (i = 0; i < nloops; i++) {
    /* Issue an empty buffer to the input stream */
    if (SIO_issue(input, buf, SIO_bufsize(input), NULL) < 0)
    {

```

```

    }

    /* Reclaim full buffer from the input stream */
    if ((nbytes = SIO_reclaim(input,&buf,&arg)) < 0) {
        SYS_abort("Error reclaiming buffer %d",i);
    }

    /* Issue full buffer to the output stream */
    if (SIO_issue(output,buf,nbytes,NULL) < 0) {
        SYS_abort("Error issuing buffer %d",i);
    }

    /* Reclaim empty buffer from the output stream to be reused */
    if (SIO_reclaim(output,&buf,&arg) < 0) {
        SYS_abort("Error reclaiming buffer %d",i);
    }
}

/* Reclaim and delete the buffers used */
MEM_free(IDRAM1,buf,SIO_bufsize(input));
if ((nbytes = SIO_reclaim(input,&buf,&arg)) < 0) {
    SYS_abort("Error reclaiming buffer %d",i);
}

```

```

if

}
if (SIO_reclaim(output, &buf, &arg) < 0) {
    SYS_abort("Error reclaiming buffer %d", i);
}

MEM_free(IDRAM1, buf, SIO_bufsize(input));
}

```

该例的完整源代码和 DSP/BIOS 配置模板同样位于... \ti\tutorial\target\siotest 目录中, target 代表用户选择的平台。例 7-7 的输出和例 7-5 的输出一样。

## 7.4 可堆叠设备

SIO 模块所具有的功能对 DSP/BIOS 中的设备无关性起到了重要的作用, 因为用户应用程序在指派一个特定设备时不必关心设备的具体细节, 只需要使用一个逻辑设备名。例如, /dac 就是个逻辑设备名, 它并不对应任何特定的 DAC 硬件设备, 这种设备命名规则为 DSP/BIOS 设备无关的 I/O 增加了一个特有的功能——能够用一个逻辑设备名称来指派一

堆设备。

注意：通过将一些数据的流传输设备或者信息传递设备一个一个堆叠起来，用户就可以建立虚拟 I/O 设备，来进一步将应用程序和底层系统硬件分离开。

考虑一个例子，一个应用程序在实现算法时，使用了一对 14 位 A/D-D/A 转换器来进行定点数据流的输入和输出。如果用户想将输入数据按比例增大到 16 位，那么 A/D-D/A 设备只能选择处理 16 位数据中的高 14 位。

为了使数据转换和数据缓冲能够满足算法要求，用户可以不用增加那些会使程序变得杂乱无章的额外代码，而使用例 7-8 所示的方法，打开一对虚拟设备，它们会对底层真实设备生成和接收的数据隐式地进行一系列变换。

#### 例 7-8 打开一对虚拟设备

```
SIO_Handle input;  
SIO_Handle output;  
Ptr_Handle buf;  
Int_Handle n;  
buf = MEM_alloc(0, MAXSIZE, 0);  
input = SIO_create("/scale2/a2d", SIO_INPUT, MAXSIZE, NULL);  
output = SIO_create("/mask2/d2a", SIO_OUTPUT, MAXSIZE, NULL);  
while (n = SIO_get(input, &buf)) {  
    'apply algorithm to contents of buf'
```



```
}  
SIO_delete(input);  
SIO_delete(output);
```

在例 7-8 中,虚拟输入设备 /scale2/a2d,实际上由一个包含两个设备的设备堆构成,它们都是根据用户配置文件里定义的设备名称加上前缀来命名的。

- ❑ /scale2 指定了一个设备,它将底层设备(/a2d)产生的定点数据流转换成按比例增大的定点数据流。
- ❑ /a2d 指定了一个由 A/D-D/A 设备驱动来管理的设备,负责驱动一个具体的 A/D 转换器产生输入定点数据流。

虚拟输出设备 /mask2/d2a 同样指定了一个包含两个设备的设备堆。图 7-6 给出了应用程序调用 SIO 流传输函数时,通过这些虚拟的源出(source)和吸纳(sink)设备的空帧和满帧的传输流程图。

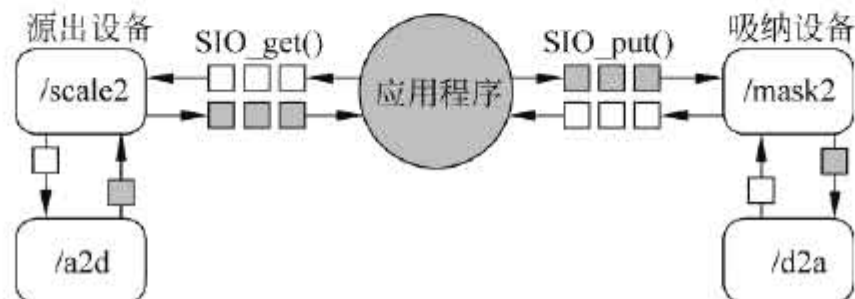


图 7-6 空帧和满帧数据传输流程图

## 7.4.1 例子——SIO\_create 和堆叠设备

例 7-9 给出了两个任务, sourceTask 和 sinkTask, 通过一个管道设备来交换数据的程序示例。

sourceTask 是一个写任务, 负责从一个联结到 DGN 正弦设备的输入流中接收数据, 然后将这些数据导出到一个输出流, 该输出流与一个 DPI 管道设备联结。输入流还有一个在 DGN 正弦设备之上的堆叠设备 scale。来自 DGN 正弦设备的数据流在被 sourceTask 任务接收之前, 首先被 scale 设备处理(给每个数据乘以一个固定的整数值)。

sinkTask 是一个读任务, 它通过一个输入流读取 sourceTask 发送给 DPI 管道设备的数据, 然后通过一个输出流将这些数据导出到一个 DGN printData 设备。

例 7-9 中的设备已经在配置工具中静态配置, 其完整的源代码和配置模板(siotest5.c, siotest5.cdb, dgn\_print.c)可以在目录... \ti\tutorial\target\siotest 中找到。sineWave 和 printData 都是 DGN 设备, pip0 是一个 DPI 设备, scale 是一个 DTR 堆叠设备。关于怎样添加和配置 DPI、DGN 以及 DTR 设备, 请查阅相应平台的 TMS320 DSP/BIOS API 函数参考手册上的介绍。

例 7-9 中使用的流也已添加到配置工具中。sourceTask 任务的输入流为 inStreamSrc, 其配置如图 7-7 所示。例中创建的其他 SIO 流还有 outStreamSrc(sourceTask 任务的输出

流)、inStreamSink(sinkTask 任务的输入流)以及 outStreamSink(sinkTask 任务的输出流)。这些流使用终端设备 pip0 和 printData。

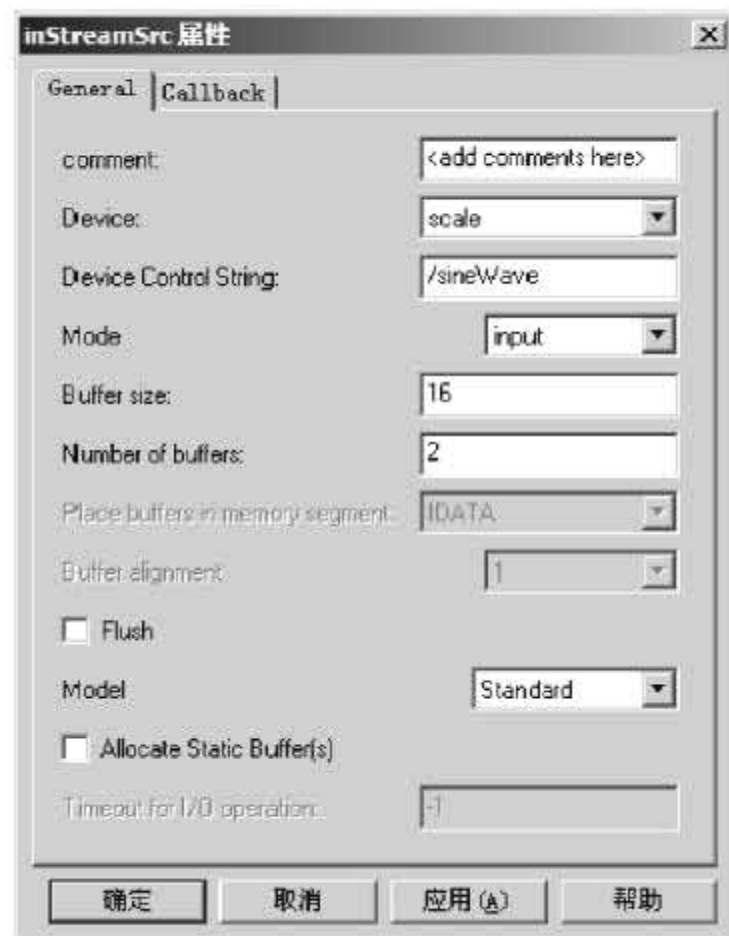


图 7-7 inStreamSrc 属性对话框

在配置工具中添加使用了堆叠设备的 SIO 流时,用户必须首先在 SIO 对象的“Device Control Parameter”属性域中键入一个已配置过的终端设备的名称,并且在名称之前加上斜杠“/”。在该例中 sineWave 是已配置好的 DGN 终端设备的名称,因此我们键入/sineWave; 然后从“Device”属性域的下拉菜单中选择已配置过的堆叠设备(scale)。在没有设定“Device Control Parameter”属性域的情况下,配置工具不会允许用户选择堆叠设备。

### 例 7-9 通过管道设备交换数据

```
/*  
 * ===== siotest5.c =====  
 * In this program two tasks exchange data through a pipe device.  
 */  
  
#include <std.h>  
#include <dtr.h>  
#include <log.h>  
#include <mem.h>  
#include <sio.h>  
#include <sys.h>  
#include <tsk.h>  
  
#define BUFSIZE 128
```

```
# ifdef _62_
# define SegId IDRAM
extern Int IDRAM;      /* MEM segment ID defined with conf tool */
# endif

# ifdef _54_
# define SegId IDATA
extern Int IDATA;      /* MEM segment ID defined with conf tool */
# endif

# ifdef _55_
# define SegId DATA
extern Int DATA;      /* MEM segment ID defined with conf tool */
# endif

extern LOG_Obj trace;   /* LOG object created with conf tool */
extern TSK_Obj sourceTask; /* TSK thread objects created via conf tool */
extern TSK_Obj sinkTask;
extern SIO_Obj inStreamSrc; /* SIO streams created via conf tool */
extern SIO_Obj outStreamSrc;
extern SIO_Obj inStreamSink;
```

```
extern SIO_Obj outStreamSink;
```

```
/* Parameters for the stacking device "/scale" */
```

```
DTR_Params DTR_PRMS = {
```

```
    20,          /* Scaling factor */
```

```
    NULL,
```

```
    NULL
```

```
};
```

```
Void source(Uns nloops);
```

```
/* function body for sourceTask above */
```

```
Void sink(Uns nloops);    /* function body for sinkTask above */
```

```
static Void doStreaming(SIO_Handle input, SIO_Handle output, Uns nloops);
```

```
/*
```

```
 * ===== main =====
```

```
 */
```

```
Void main()
```

```
{
```

```

        LOG_printf(&trace, "Start SIO example #5");
    }

    /*
     * ===== source =====
     * This function forms the body of the sourceTask TSK thread.
     */
    Void source(Uns nloops)
    {
        SIO_Handle input = &inStreamSrc;
        SIO_Handle output = &outStreamSrc;
        /* Do I/O */
        doStreaming(input, output, nloops);
    }

    /*
     * ===== sink =====
     * This function forms the body of the sinkTask TSK thread.
     */
    Void sink(Uns nloops)
    {

```

```

    SIO_Handle input = &inStreamSink;
    SIO_Handle output = &outStreamSink;
    /* Do I/O */
    doStreaming(input, output, nloops);
    LOG_printf(&trace, "End SIO example # 5");
}

/*
 * ===== doStreaming =====
 * I/O function for the sink and source tasks.
 * /
static Void doStreaming(SIO_Handle input, SIO_Handle output, Uns nloops)
{
    Ptr    buf;
    Int    i, nbytes;

    if (SIO_staticbuf(input, &buf) == 0){
        SYS_abort("Error reading buffer % d", i);
    }
    for (i = 0; i < nloops; i++) {
        if ((nbytes = SIO_get (input, &buf)) < 0) {

```



```

        SYS_abort ("Error reading buffer % d", i);
    }
    if (SIO_put (output, &buf, nbytes) < 0) {
        SYS_abort ("Error writing buffer % d", i);
    }
}
}

```

图 7-8 给出了例 7-9 的输出, 经过比例变换的正弦波数据在 myLog 日志窗口中显示出来。

用户可以编辑 siotest5.c 来改变比例变换因子 DTR\_PRMS, 重新链接生成可执行文件, 然后观察输出窗口 myLog 中输出数据和改变前输出结果之间的差别。

例 7-9 的另一个版本, 通过在运行时调用 SIO\_create 动态创建流来实现相同的功能, 其源代码和配置模板 (siotest4.c, siotest.cdb) 可以在目录... \ti\tutorial\target\siotest 中找到。

Log Name: myLog	
0	0
1	480
2	900
3	1180
4	1270
5	1180
6	900
7	480
8	0
9	-490
10	-910
11	-1190
12	-1280
13	-1190
14	-910
15	-490
16	0
17	480
18	900
19	1180
20	1270
21	1180
22	900
23	480
24	0

图 7-8 例 7-9 的正弦波形数据输出

## 7.5 流 控 制

为了完成所要求的操作,实际的物理设备通常需要一个或多个特殊的控制信号。函数 `SIO_ctrl` 能够实现与设备的通信,将命令和参数传递给设备。因为每个设备只接收其所规定的特殊命令,所以用户需要参考每个特定设备驱动的说明文档。例 7-10 给出了 `SIO_ctrl` 的一般调用格式。

**例 7-10** 用 `SIO_ctrl` 和设备进行通信

```
Int SIO_ctrl(stream,cmd,arg)
    SIO_Handle  stream;
    Uns         cmd;
    Ptr         arg;
```

参数 `cmd` 所代表的设备特有命令会被传递给与流联结的设备。通用指针 `arg` 指向该特有命令所含的参数,它也被传递给该设备。设备驱动提供的控制函数负责解释该命令及其参数并进行相应的操作。

假设模数转换设备/a2d 具有能够改变采样速率的控制操作,那么设备/a2d 的采样速率可如例 7-11 那样被设置为 12kHz。

### 例 7-11 改变采样速率

```
SIO_Handle stream;
```

```
stream = SIO_create("/a2d",...);
```

```
SIO_ctrl( stream, DAC_RATE, 12000);
```

在一些情况下,用户需要和进行数据缓冲的 I/O 设备进行同步。和设备同步的方法有两种: SIO\_idle 和 SIO\_flush,这两个函数都用于使设备进入空闲状态。使一个设备空闲意味着所有的缓冲区都返回到它们在设备初始创建时所处的队列中,即设备回到它的初始状态,并且流传输也被停止了。对输入流,这两个函数会有相同的结果:所有未读取的输入数据都被丢弃。对输出流,SIO\_idle 会阻塞,直到所有的缓冲数据都写到设备端。而 SIO\_flush 会丢弃那些还没有来得及写到设备端的数据,不会产生阻塞。

#### 例 7-12 和设备同步的函数

```
Void SIO_idle(stream);  
    SIO_Handle stream;  
Void SIO_flush(stream);  
    SIO_Handle stream;
```

空闲流不与底层的设备进行 I/O 操作,因此,当没有输入或输出操作的需要时,可以调用 SIO\_idle 或者 SIO\_flush 时将流关闭。

## 7.6 流 选 择

函数 `SIO_select` 允许一个 DSP/BIOS 任务处于等待状态,直到可以从一组 SIO 流中找出一个或多个流能够进行不受阻塞的 I/O 操作。下面的应用就需要这种机制。

- ❑ 不受阻塞的 I/O 操作：对一个慢速设备（例如一个磁盘文件）进行数据流传输的实时任务必须保证 SIO\_put 不受阻塞。
- ❑ 多任务：实际上，任何多任务应用程序都有多个收发任务负责对来自多个源的数据进行疏导。SIO\_select 的机制则允许单个任务来处理所有数据源。

调用 `SIO_select` 时需要一个流数组(数组元素为流对象),一个数组长度以及一个超时值,如例 7-13 所示。当超时值不为 0 时,`SIO_select` 可以阻塞直到该组流中的其中一个就绪(和其联结的设备已准备好进行 I/O 操作)或者超时时间耗尽。不管在何种情况下,`SIO_select` 返回的掩模值会指示出哪些流已经就绪(如果掩模的第  $j$  位为 1,则表明 `streamtab[j]` 已经就绪)。

### 例 7-13 找出已就绪的流

[illegible]

在例 7-14 中,两个流被轮询以查看它们的 I/O 操作是否会受到阻塞。

**例 7-14** 轮询两个流

```
SIO_Handle      stream0;
SIO_Handle      stream1;
SIO_Handle      streamtab[2];
Uns             mask;
...
streamtab[0] = stream0;
streamtab[1] = stream1;
while ((mask = SIO_select(streamtab,2,0)) == 0) {
    'I/O would block,do something else'
}
if (mask & 0x1) {
    'service stream0'
}
if (mask & 0x2) {
    'service stream1'
}
```

## 7.7 到多个客户端的流传输

多处理系统中的一个常见问题是如何将一个数据缓冲区同时传送给系统中的多个任务。这种多点传送或者分散传送,都能用 DSP/BIOS SIO 流轻松地完成。读者可以考虑一个处理器向 4 个客户端处理器发送数据的情况,这种数据的流传输和与一个获取设备(例如 A/D 转换器)通过流交换缓冲区有所不同,因为单个数据缓冲区必须发送到一个或多个客户端。

DSP/BIOS 的 SIO 函数 SIO\_get/SIO\_put 可以用来完成这种传输。SIO\_put 会自动进行缓冲区交换,这种交换是在设备端和应用程序端之间进行的。交换后用户将不再对发送出的缓冲区具有控制权,因为它已被送到 I/O 队列,并且这种输入/输出操作是通过中断被异步触发的,这就迫使用户通过复制数据才能将其发送到多个客户端,如例 7-15 所示。

**例 7-15** 用 SIO\_put 发送数据到多个客户端

```
SIO_get(inStream, (Ptr)&bufA, npoints);  
'fill bufB,C,D, ... with data of bufA'  
for ('all data points') {  
    bufB[i] = bufC[i] = bufD[i] = ... = bufA[i];
```

```

}
SIO_put(outStreamA, (Ptr)&bufA, npoints);
SIO_put(outStreamB, (Ptr)&bufB, npoints);
SIO_put(outStreamC, (Ptr)&bufC, npoints);
SIO_put(outStreamD, (Ptr)&bufD, npoints);

```

数据复制不仅会浪费 CPU 时钟周期而且需要更多的存储空间,因为每个流都需要缓冲区。在这种情况下,使用 SIO\_issue 和 SIO\_reclaim 的优势就体现出来,如例 7-16 所示。应用程序不再需要复制数据,而仅仅使用了两个缓冲区。在每次调用中,SIO\_issue 只是简单地将 bufA 指向的缓冲区无阻塞地添加到流 outStream 的 todevice 队列当中。因为没有复制和阻塞,这种方法大大地减少了一个缓冲区从就绪到可以将其传给所有客户的时间。为了从输出设备端回收缓冲区,必须调用相应的 SIO\_reclaim 函数。

**例 7-16** 用 SIO\_issue/SIO\_reclaim 发送数据到多个客户端

```

SIO_issue(outStreamA, (Ptr)bufA, npoints, NULL);
SIO_issue(outStreamB, (Ptr)bufA, npoints, NULL);
SIO_issue(outStreamC, (Ptr)bufA, npoints, NULL);
SIO_issue(outStreamD, (Ptr)bufA, npoints, NULL);
SIO_reclaim(outStreamA, (Ptr)&bufA, NULL);
SIO_reclaim(outStreamB, (Ptr)&bufA, NULL);
SIO_reclaim(outStreamC, (Ptr)&bufA, NULL);
SIO_reclaim(outStreamD, (Ptr)&bufA, NULL, SYS_FOREVER);

```



注意：当设备驱动会改变缓冲区中的数据时，使用 SIO\_issue 向多个设备发送相同的缓冲区将会存在问题，因为这些设备实际使用的是同一个缓冲区。例如，一个用于将打包数据解包的堆叠设备会在另一个设备输出该缓冲区的同时改变该缓冲区。

SIO\_issue 接口提供了一种能让所有的通信设备访问相同缓冲区数据的方法，每个通信设备的驱动可以使用 DMA 来同时读取该缓冲区中的数据。例 7-16 中，应用程序不会从 4 个 SIO\_reclaims 返回，除非得到一个缓冲区可被所有的流使用。

总之，SIO\_issue/SIO\_reclaim 函数提供了最有效的方法来把数据同时传输给多个流。然而这种操作不具有反向性：SIO\_issue/SIO\_reclaim 模型能够非常有效地解决输出数据的分散问题，但是不能在输入时把多个数据源的数据收集到单个缓冲区中。

## 7.8 主机与目标板之间数据的流传输

主机通道对象(HST 对象)允许应用程序在目标板与主机文件之间实现数据的流传输。在 DSP/BIOS 分析工具中，用户可将这些通道绑定(bind)到主机文件，并启动它们。

DSP/BIOS 包含一个主机 I/O 模块(HST)，可以使主机计算机和目标 DSP 程序之间的数据传输变得容易。每个主机通道在内部使用一个 SIO 流对象实现。为了使用一个主机通

道,应用程序调用 `HST_getstream` 获得主机通道中的流对象,然后通过 `SIO` 调用来进行主机和目标 DSP 之间数据的流传输。

## 7.9 设备驱动模板

因为设备驱动直接和各种硬件交互,所以其低层次细节的变化是多种多样的。然而,所有的设备驱动必须对 `SIO` 呈现同样的接口。本节将给出一个名为 `Dxx` 的驱动模板作为示例。这个模板主要包含实现上层操作的 C 语言代码和实现底层操作的伪代码。任何设备驱动都应该和 `Dxx` 函数的接口标准相一致。

用户应该结合一个或多个实际的驱动来学习 `Dxx` 驱动模板,也可以参考相应平台的 TMS320 DSP/BIOS API 函数参考手册对 `Dxx` 函数的说明,xx 代表任何两个字母的结合。关于如何配置设备驱动(包括用户制定的驱动和 DSP/BIOS 本身提供的驱动)的细节,则需要参考详细的设备驱动说明。

### 7.9.1 典型的文件组织

设备驱动通常被分成多个文件。典型的文件组织为:

❑ **dxh.h**: Dxx 头文件

❑ **dxh.c**: Dxx 函数

❑ **dxh\_asm.s** ## : (可选的)汇编语言函数(##代表 DSP 系列,如 54)

大多数设备驱动的代码都可以用 C 语言编写。下面对 Dxx 的描述就没有用到汇编语言,然而为了高效,中断服务程序都是用汇编语言编写的,并且一些硬件控制函数也需要用汇编语言来编写。

我们推荐用户能够先熟悉一个软件设备驱动(例如 DGN)的代码设计和编排,尤其需要注意以下几点:

❑ 头文件(dxh.h)除了包含一些设备特有的定义之外,一般还应包括例 7-17 所示的必需的语句。

❑ 设备参数,例如 Dxx\_Params,会作为配置工具中设备对象的属性。

例 7-17 dxh.h 头文件中必需的语句

```
/*
 * ===== dxh.h =====
 */
#include <dev.h>
extern DEV_Fxns Dxx_FXNS;

/*
 * ===== Dxx_Params =====
```

```
    */  
typedef struct {  
    'device parameters go here'  
} Dxx_Params;
```

必需的设备函数表包含在 dxx.c 中。SIO 模块根据该表来调用特定的设备驱动函数。例如, SIO\_put 会根据这个表格查找并调用 Dxx\_issue/Dxx\_reclaim。设备函数表如例 7-18 所示。

#### 例 7-18 设备函数表

```
DEV_Fxns Dxx_FXNS = {  
    Dxx_close,  
    Dxx_ctrl,  
    Dxx_idle,  
    Dxx_issue,  
    Dxx_open,  
    Dxx_ready,  
    Dxx_reclaim  
};
```

## 7.10 流 DEV 结构体

## 7.10.1 DEV\_Fxns 结构体

DEV\_Fxns 类型的结构体包含了一组函数指针,它们指向与通用 I/O 操作相对应的内部驱动函数,如例 7-19 所示。

例 7-19 DEV\_Fxns 结构体

```
typedef struct DEV_Fxns {  
    Int      (* close)(DEV_Handle);  
    Int      (* ctrl)(DEV_Handle, Uns, Arg);  
    Int      (* idle)(DEV_Handle, Bool);  
    Int      (* issue)(DEV_Handle);  
    Int      (* open)(DEV_Handle, String);  
    Bool     (* ready)(DEV_Handle, SEM_Handle);  
    Int      (* reclaim)(DEV_Handle);  
} DEV_Fxns;
```

## 7.10.2 DEV\_Frame 结构体

设备帧是 DEV\_Frame 类型的结构体,见例 7-20。SIO 和设备驱动使用该结构体进行流缓冲区的入队/出队操作。device->todevice 和 device->fromdevice 队列所包含的队列元素实

实际上就是这种类型的结构体。

### 例 7-20 DEV\_Frame 结构体

```
typedef struct DEV_Frame { /* frame object */
    QUE_Elem    link;      /* queue link */
    Ptr         addr;      /* buffer address */
    Uns         size;      /* buffer size */
    Arg         misc;      /* reserved for driver */
    Arg         arg;       /* user argument */
    Uns         cmd;       /* mini-driver command */
    Int         status;    /* status of command */
} DEV_Frame;
```

DEV\_Frame 结构体中包含如下参数域：

- ❑ **link**：被 QUE API 函数 QUE\_put 和 QUE\_get 使用，进行帧的入队/出队操作。
- ❑ **addr**：指定了流缓冲区的地址。
- ❑ **size**：指定了流缓冲区的逻辑大小（即有效数据的长度）。逻辑大小值可以小于缓冲区的物理大小。
- ❑ **misc**：是一个额外的保留域，供设备使用。
- ❑ **arg**：是一个额外的供用户使用的域，用户可以使用它存放一些和特定帧数据相关的信息。设备驱动不应该改变该域。
- ❑ **cmd**：是一个命令代码，用于 IOM 模型，用于告知微型驱动进行什么样的操作。详见本书第二部分“DSP/BIOS 驱动开发手册”。

- **status:** 用于 IOM 模块, IOM 微型驱动会在调用一个回调函数之前设置该域。详见本书第二部分“DSP/BIOS 驱动开发手册”。

### 7.10.3 DEV\_Obj 结构体

所有设备驱动函数都使用一个 DEV\_Handle 类型的句柄作为第一个参数。DEV\_Handle 实际上是指向一个 DEV\_Obj 类型结构体的指针。该 DEV\_Obj 被 SIO\_create 函数创建和初始化, 并被传递给 Dxx\_open 做进一步的初始化。DEV\_Obj 中包含了指向缓冲区队列的指针, SIO 和设备用这些队列来交换缓冲区。

例 7-21 DEV\_Handle 指向的结构体

```
typedef DEV_Obj * DEV_Handle;
typedef struct DEV_Obj { /* device object */
    QUE_Handle    todevice;    /* downstream frames here */
    QUE_Handle    fromdevice;  /* upstream frames here */
    Uns           bufsize;     /* buffer size */
    Uns           nbufs;       /* number of buffers */
    Int           segid;       /* buffer segment ID */
    Int           mode;        /* DEV_INPUT/DEV_OUTPUT */
    LgInt         devid;       /* device ID */
}
```

```

Ptr      params;      /* device parameters */
Ptr      object;      /* ptr to dev instance obj */
DEV_Fxns  fxns;        /* driver functions */
Uns      timeout;     /* SIO_reclaim timeout value */
Uns      align;       /* buffer alignment */
DEV_Callback *callback; /* pointer to callback */
} DEV_Obj;

```

DEV\_Obj 结构体中包含如下参数域：

- ❑ **todevice**: 用于向设备传递 DEV\_Frame 帧的队列。在使用标准流传输模型时，SIO\_put 会放入满帧到该队列中，SIO\_get 会放入空帧到该队列中。在使用发放/回收流传输模型时，SIO\_issue 发放帧到该队列中。
- ❑ **fromdevice**: 用于向应用程序端传递 DEV\_Frame 帧的队列。在使用标准流传输模型时，SIO\_put 会从该队列中得到空帧，SIO\_get 会从该队列中得到满帧。在使用发放/回收流传输模型时，SIO\_reclaim 从该队列中回收帧。
- ❑ **bufsize**: 指定了帧队列中单个缓冲区的物理大小。
- ❑ **nbufs**: 指定了在标准流传输模型中分配给该设备的缓冲区的数目，或是在发放/回收流传输模型下能够在流中存在的缓冲区的最大数目。
- ❑ **segid**: 指定了设备缓冲区是从哪个存储器段中分配的，用于标准流传输模型。
- ❑ **mode**: 指定了该设备是输入设备还是输出设备。
- ❑ **devid**: 设备 ID。



- ❑ **params:** 是指向设备特有参数的指针。
- ❑ **object:** 指向设备特有对象的指针。大多数设备驱动会创建能够在连续的设备操作中引用的对象,并使用该域存放该对象的指针。
- ❑ **fxns:** 是一个包含驱动函数的 `Dxx_Fxns` 结构体。这个结构体通常是 `Dxx_FXNS` 结构体的一个拷贝,但设备驱动可以在 `Dxx_open` 中动态地改动其中的函数。
- ❑ **timeout:** 指定了 `SIO_reclaim` 最多会花多少系统时钟来等待 I/O 操作完成。
- ❑ **align:** 指定了缓存区的边界条件。
- ❑ **callback:** 是一个指向一个回调结构体——`DEV_Callback` 结构体的指针。该结构体包含一个回调函数和两个函数参数,其中回调函数一般为 `SWI_andnHook` 或者类似的触发 SWI 的函数。回调只可以在发放/回收模型中使用。回调机制使得 SIO 对象可以和非阻塞的 SWI 线程一起使用。

只有 `object` 和 `fxns` 域可以被设备驱动函数(例如被 `Dxx_open`)进一步初始化或修改。

## 7.11 设备驱动初始化

驱动函数表 `Dxx_FXNS` 是在 `dxx.c` 中进行初始化的,参见 7.10 节“流 DEV 结构体”。其余的初始化操作则由 `Dxx_init` 来完成。当其他应用程序级的模块被初始化时,`Dxx` 模块

也被初始化。Dxx\_init 通常会调用硬件初始化程序并初始化静态的驱动结构体,见例 7-22。

### 例 7-22 用 Dxx\_init 初始化

```
/*  
 * ===== Dxx_init =====  
 */  
Void Dxx_init()  
{  
    'Perform hardware initialization'  
}
```

尽管需要使用 Dxx\_init 以保持与 DSP/BIOS 配置和初始化的标准相一致,但 DSP/BIOS 实际上并不要求 Dxx\_init 的内部操作。事实上并不存在硬件初始化的标准,并且对于某些系统来说,在别的 Dxx 函数,比如 Dxx\_open 中执行某些硬件配置操作会更合适。因此在某些系统中,Dxx\_init 可能仅仅是一个空函数。

## 7.12 打 开 设 备

函数 Dxx\_open 用于打开一个 Dxx 设备并返回其状态,见例 7-23。函数 SIO\_create 会在内部触发 Dxx\_open 来打开 Dxx 设备,如例 7-24 所示。

**例 7-23** 用 Dxx\_open 打开设备

```
status = Dxx_open(device, name);
```

**例 7-24** 打开一个输入终端设备

```
input = SIO_create("/adc16", SIO_INPUT, BUFSIZE, NULL)
```

下列步骤说明了例 7-24 中 SIO\_create 函数打开输入终端设备“/adc16”的过程：

(1) 在 DSP/BIOS 内部的 DEV\_devtab 设备表中查找和 /adc 相匹配的字符串。并从对应的 DEV\_Device 结构体中得到该设备的驱动函数、设备 ID 和设备参数。

(2) 分配一个新的 DEV\_Obj 对象并用句柄 device 指向它。

(3) 根据传递给 SIO\_create 的 attrs(一个 SIO\_Attrs 结构体指针, 例中为 NULL) 等参数, 对 DEV\_Obj 中的 bufsize、nbufs、segid 等域进行赋值。

(4) 创建 todevice 和 fromdevice 队列, 并将队列句柄放入 DEV\_Obj 对象中对应的域中。

(5) 如果使用标准流传输模型, 分配 attrs.nbufs 个 BUFSIZE 大小的缓冲区并把它们放入 todevice 队列中。

(6) 使用指向 DEV\_Obj 对象的句柄 device 和设备名剩余字符串 16, 调用 Dxx\_open:

```
status = Dxx_open(device, "16");
```

(7) 验证 device 指向的 DEV\_Obj 中的各个域的有效性。

(8) 分析剩余字符串以确定附加参数(例如 16kHz)。

(9) 分配并初始化一个设备特有对象。

(10) 使 device->object(DEV\_Obj 对象的 object 域)指向该设备特有对象。

其中步骤(7)和步骤(8)在函数 Dxx\_open 中完成。

Dxx\_open 的参数如例 7-25 所示。参数 device 指向一个 DEV\_Obj 对象,其参数域被 SIO\_create 函数初始化。参数 name 是设备名被 SIO\_create 使用 DEV\_match 进行匹配后剩余的字符串。

**例 7-25** Dxx\_open 的参数

```
DEV_Handle device; /* driver handle */  
String name;      /* device name */
```

再来看一看 SIO\_create 调用时接受的参数,见例 7-26。

**例 7-26** SIO\_create 的参数。

```
stream = SIO_create(name,mode,bufsize,attrs);
```

传递给 SIO\_create 的参数 name 通常由一个代表设备的字符串和一个附加后缀组成,该后缀代表着该设备的一些特殊操作模式。例如一个模数转换器可能有一个基本的名称 /adc,而其采样频率就可以通过后缀来指示,如 16 代表 16kHz。所以传给 SIO\_create 的 name 参数就是 /adc16。

SIO\_create 使用 DEV\_match 来从已配置设备的列表中匹配字符串 /adc,从而查找和识

别设备。剩余的字符串 16 将会传给 Dxx\_open 来将 ADC 设置在正确的采样速率上。

Dxx\_open 通常会分配一个设备特有的对象,该对象用来保存设备状态和用于同步的信号灯。对于一个终端设备,这个对象一般包含两个信号灯句柄(SEM\_Handle),一个信号灯用来同步 I/O 操作(例如 SIO\_get、SIO\_put、SIO\_reclaim)。另一个信号灯和 SIO\_select 一起使用来确定设备是否就绪。一个设备特有对象的典型定义如例 7-27 所示。

#### 例 7-27 Dxx\_Obj 结构体

```
typedef struct Dxx_Obj {  
    SEM_Handle    sync;    /* synchronize I/O */  
    SEM_Handle    ready;   /* used with SIO_select() */  
    'other device-specific fields'  
} Dxx_obj, * Dxx_Handle;
```

例 7-28 提供了一个 Dxx\_open 函数的模板,给出了打开终端设备时该函数的典型做法。

#### 例 7-28 打开终端设备时 Dxx\_open 函数的典型做法

```
Int Dxx_open(DEV_Handle device,String name)  
{  
    Dxx_Handle objptr;  
  
    /* check mode of device to be opened */  
    if ('device->mode is invalid') {
```

```

}
/* check device id */
if ('device->devid is invalid') {
    return (SYS_ENODEV);
}
/* if device is already open, return error */
if ('device is in use') {
    return (SYS_EBUSY);
}
/* allocate device-specific object */
objptr = MEM_alloc(0, sizeof (Dxx_Obj), 0);

'fill in device-specific fields'

/* create synchronization semaphore ... */
objptr->sync = SEM_create(0, NULL);
/* initialize ready semaphore for SIO_select()/Dxx_ready() */
objptr->ready = NULL;

```

```
/* assign initialized object */  
device->object = (Ptr)objptr;  
  
return (SYS_OK);  
}
```

前两步用来进行错误检查,例如为输入需求而打开一个只能输出的设备就会产生一个错误信息。在只有 5 个通道的系统中打开通道 10(设备 ID 超出范围)也会产生一个错误信息。

下一步是确定设备是否已经被打开。在许多情况下,已经被打开的设备不能被再次打开,否则就给出一个错误信息。

如果设备能被打开,Dxx\_open 函数的剩余部分主要包含两个操作。首先分配一个设备特有对象,并根据 SIO\_create 传递下来的 device->params 中的一部分设置,初始化该设备特有对象。返回前设置 device->object 指向该对象。最后 Dxx\_open 返回 SYS\_OK 给 SIO\_create,这时 SIO\_create 就拥有了一个已经正确初始化了的 DEV\_obj 对象。

用户可配置的设备参数用来设置硬件的操作参数,DSP/BIOS 并没有限制哪些参数应该在函数 Dxx\_init 中设置,哪些应该在函数 Dxx\_open 中设置。

信号灯对象 objptr->sync 通常用来给一个正在等待 I/O 操作完成的任务发送信号。例如,一个任务可以调用 SIO\_put,它可能为等待 objptr->sync 而阻塞。当所要求的输出完成后,函数 SEM\_post 会被调用来发布信号灯 objptr->sync。这就使得在 Dxx\_output 中阻塞的任务重新就绪运行。

在设备驱动中使用同步信号灯方面,DSP/BIOS 并没有强加任何特殊限制。对这些信号灯的合理使用取决于驱动的本身需求和底层硬件。

信号灯 `objptr->ready`, 会被函数 `Dxx_ready` 使用, 而 `Dxx_ready` 函数由 `SIO_select` 调用来决定设备是否就绪好进行 I/O 操作。对于信号灯, 4.7 节已经作了介绍。

## 7.13 实时 I/O

在 DSP/BIOS 中, 有两个用于实时 I/O 操作的流传输模型: `DEV_STANDARD` 流传输模型和 `DEV_ISSUERECLAIM` 流传输模型。本节将对这两个模型进行详细介绍。

### 7.13.1 DEV\_STANDARD 流传输模型

在 `DEV_STANDARD` 流传输模型中, `SIO_get` 用来从输入流中获取一个满帧。为此, `SIO_get` 会首先在 `device->todevice` 队列中放置一个空帧, 然后调用 `Dxx_issue` 启动 I/O 操作, 接着调用驱动函数 `Dxx_reclaim` 进行等待, 直到 `device->fromdevice` 队列中有一个满帧可用。 `Dxx_reclaim` 中的阻塞是通过调用 `SEM_pend` 等待设备信号灯 `objptr->sync` 来实现的。无论任何时候一个缓冲区被填满该信号灯就会被发布。

`Dxx_issue` 函数会调用一个低层次的硬件函数来初始化输入设备接收外界数据。当设



备已经接收够一帧的数据后,该数据帧就被发送到 device->fromdevice 队列中。通常情况下,当接收到一定数量的数据后,硬件设备会发出一个中断。Dxx 利用一个 HWI(图 7-9 中的 ISR)来处理该中断,HWI 会累计数据量并决定是否需要接收更多的数据,如果 HWI 认为所需数量的数据已全部接收到,就会将该帧发送到 device->fromdevice 队列中,然后 HWI 会调用 SEM\_post 发布设备信号灯。这就使得在 Dxx\_reclaim 中阻塞的任务继续运行。Dxx\_reclaim 最后返回到 SIO\_get。SIO\_get 完成输入操作的过程如图 7-9 所示。图中也给出了 SIO\_put 完成输出操作的过程。



图 7-9 DEV\_STANDARD 流传输模型的流程

注意: objptr->sync 是个计数信号灯,所以任务并不总是会被阻塞在这里。该信号灯的计数值代表着 device->fromdevice 队列中可用帧的个数。

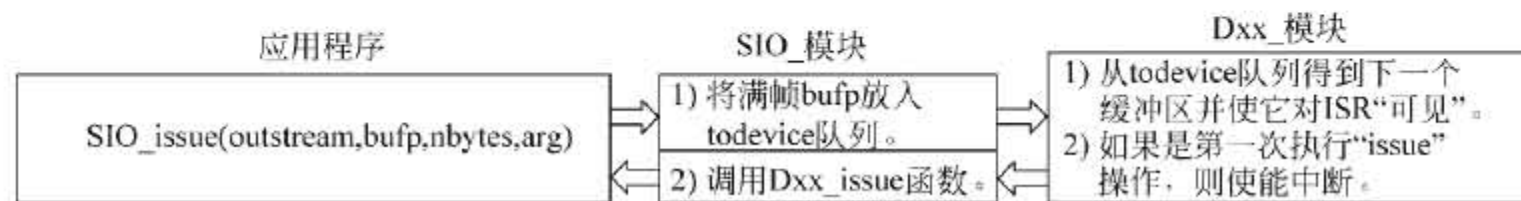
## 7.13.2 DEV\_ISSUERECLAIM 流传输模型

在 DEV\_ISSUERECLAIM 流传输模型中,SIO\_issue 用来向流中发放缓冲区。为此,SIO\_issue 首先把该帧放在 device->todevice 队列上。然后调用 Dxx\_issue,Dxx\_issue 启动 I/O 操作然后返回。

Dxx\_issue 会调用一个低层次硬件函数来初始化 I/O 操作。

SIO\_reclaim 用来回收流中的缓冲区。这通过调用 Dxx\_reclaim 函数完成,而 Dxx\_reclaim 会一直阻塞,直到在 device->fromdevice 队列中出现一个可用帧。该阻塞通过调用 SEM\_pend 函数等待设备信号灯 objptr->sync 来实现。当设备 HWI(在图 7-10 中和图 7-11 中的 ISR)发布信号灯 objptr->sync,Dxx\_reclaim 就不再阻塞并返回到 SIO\_reclaim。紧接着,SIO\_reclaim 会得到 device->fromdevice 队列中的可用帧,并将其返回给任务。

图 7-10 和图 7-11 给出了这种流传输模型的 I/O 操作流程。



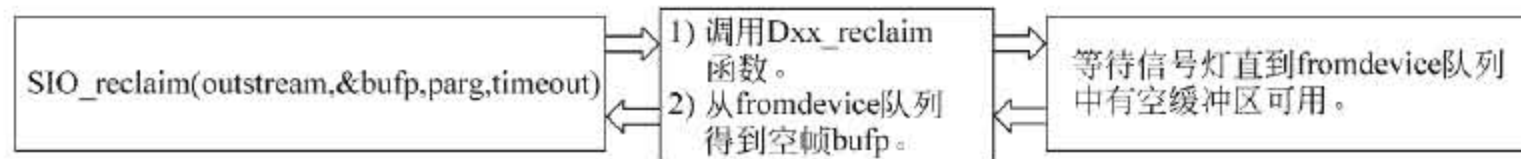


图 7-10 把数据发放到流中

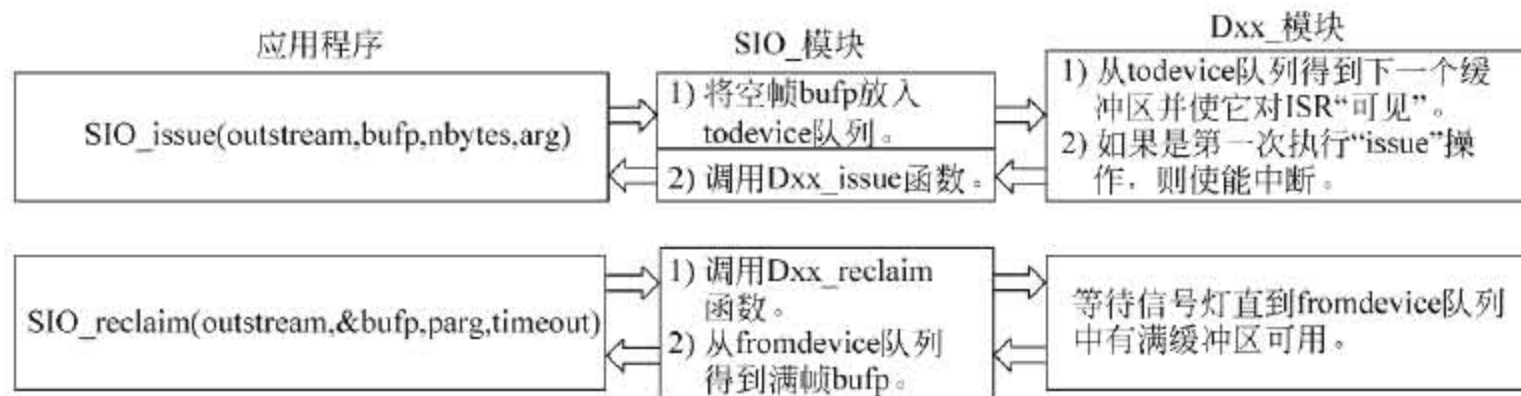


图 7-11 从流中接收数据

例 7-29 给出了对一个典型的终端设备进行操作的 `Dxx_issue` 函数模板。

**例 7-29** 一个典型的终端设备的 `Dxx_issue` 模板

```
/*
 * ===== Dxx_issue =====
 */
```

```

1
{
    Dxx_Handle objptr = (Dxx_Handle) device->object;
    if ('device is not operating in correct mode') {
        'start the device for correct mode'
    }
    return (SYS_OK);
}

```

Dxx\_issue 函数首先启动设备工作于要求的模式：输入(DEV\_INPUT)或输出(DEV\_OUTPUT)。一旦确定设备已经启动，Dxx\_issue 就会返回。实际的数据处理则通过 HWI 来执行。

例 7-30 给出了对一个典型的终端设备进行操作的 Dxx\_reclaim 函数模板。

**例 7-30** 一个典型终端设备的 Dxx\_reclaim 模板

```

/*
 * ===== Dxx_reclaim =====
 * /
Int Dxx_reclaim(DEV_Handle device)
{
    Dxx_Handle objptr = (Dxx_Handle) device->object;
    if (SEM_pend(objptr->sync, device->timeout)) {
        return (SYS_OK);
    }
}

```

```

    }
    else {
        /* SEM_pend() timed out */
        return (SYS_ETIMEOUT);
    }
}

```

Dxx\_reclaim 函数会等待 HWI 在 device->fromdevice 队列中放入一帧, 然后返回。

Dxx\_reclaim 会调用带有超时参数的 SEM\_pend 函数, 该超时值是流在创建(通过配置工具或是 SIO\_create 函数)时指定的。如果该时间在缓冲区可用之前就用尽了, Dxx\_reclaim 返回 SYS\_ETIMEOUT。在这种情况下, SIO\_reclaim 不再尝试从 device->fromdevice 队列获取任何数据, 且返回 SYS\_ETIMEOUT 而非缓冲区。

## 7.14 关闭设备

设备的关闭通过调用 SIO\_delete 函数实现, SIO\_delete 内依次调用 Dxx\_idle 和 Dxx\_close。函数 Dxx\_idle 会将设备返回到其初始状态(设备在打开时的最初状态), 然后 Dxx\_close 会关闭设备, 见例 7-31。

**例 7-31** 关闭设备

```

/*
 * ===== Dxx_idle =====
 */
Int Dxx_idle(DEV_Handle device, Bool flush)
{
    Dxx_Handle objptr = (Dxx_Handle) device->object;
    Uns post_count;

    /*
     * The only time we will wait for all pending data is when
     * the device is in output mode, and flush was not
     * requested.
     */
    if ((device->mode == DEV_OUTPUT) && !flush) {
        /* first, make sure device is started */
        if ('device is not started'
            && 'device has received data')

        {
            'start the device'
        }

        /*

```

```

    * wait for all output buffers to be consumed by the
    * output HWI. We need to maintain a count of how many
    * buffers are returned so we can set the semaphore
    * later.
    * /
post_count = 0;
while (!QUE_empty(device->todevice)) {
    SEM_pend(objptr->sync, SYS_FOREVER);
    post_count++;
}
if ('there is a buffer currently in use by the HWI') {
    SEM_pend(objptr->sync, SYS_FOREVER);
    post_count++;
}
'stop the device'

/*
    * Don't simply SEM_reset the count here. There is a
    * possibility that the HWI had just completed
    * working on a buffer just before we checked, and we
    * don't want to mess up the semaphore count.

```

```

        * /
    while (post_count > 0) {
        SEM_post(objptr->sync);
        post_count--;
    }
}

else {
    /* dev->mode = DEV_INPUT or flush was requested */
    'stop the device'

    /* do standard idling, place all frames in fromdevice
       queue */
    while (!QUE_empty(device->todevice)) {
        QUE_put(device->fromdevice,
                QUE_get(device->todevice));
        SEM_post(objptr->sync);
    }
}
return (SYS_OK);
}

```

Dxx\_idle 的参数如下：



```
DEV_Handle device; /* driver handle */  
Bool flush; /* flush indicator */
```

参数 device 是一个指向该设备的 DEV\_Obj 实体的指针。flush 是个布尔参数,用来指定在返回设备到其初始状态的过程中,如何处理未传输完成的数据。

对处于输入模式的设备,所有未传输完成的数据总是会被丢弃掉,因为没有办法强迫一个任务去重新接收来自设备的数据。因此,flush 参数对用输入模式打开的设备没有影响。

然而对处于输出模式的设备,参数 flush 则很重要。如果 flush 值为真(TRUE),任何未传输完成的数据都被丢弃;如果 flush 值为假(FALSE),直到所有未传输完成的数据都被传递之后,Dxx\_idle 函数才会返回。

## 7.15 设备控制

函数 Dxx\_ctrl 会被 SIO\_ctrl 调用,用以对设备进行控制操作。Dxx\_ctrl 的一个典型使用就是改变设备控制寄存器值或者 A/D、D/A 设备的采样频率值。Dxx\_ctrl 的调用语法如下:

```
status = Dxx_ctrl(DEV_Handle device, Uns cmd, Arg arg);
```

- ❑ `cmd` 是一个设备特有命令。
- ❑ `arg` 提供了可选的命令参数。

如果操作成功, `Dxx_ctrl` 会返回 `SYS_OK`, 否则 `Dxx_ctrl` 会返回一个错误的代码。

## 7.16 设备就绪

`SIO_select` 调用 `Dxx_ready` 来决定一个设备是否准备好进行 I/O 操作。如果设备已经准备就绪, `Dxx_ready` 就返回真(TRUE), 否则返回假(FALSE)。如果下一次调用 I/O 函数从一个设备获取缓冲区时不会发生阻塞, 则该设备已就绪。这通常意味着在 `Dxx_ready` 返回时, 队列 `device->fromdevice` 中至少有一个可用的帧, 如例 7-32 所示。有关 `SIO_select` 更多的信息, 请查阅 7.6 节“流选择”。

### 例 7-32 判断设备是否就绪

```
Bool Dxx_ready(DEV_Handle dev, SEM_Handle sem)
{
    Dxx_Handle objptr = (Dxx_Handle)device->object;

    /* register the ready semaphore */
```

```

objptr->ready = sem;
if ((device->mode == DEV_INPUT) &&
    ((device->model == DEV_STANDARD) &&
     'device is not started')) {
    'start the device'
}

/* return TRUE if device is ready */
return ('TRUE if device->fromdevice has a frame or device won't block');
}

```

如果设备模式为输入(DEV\_INPUT),且流传输模型为标准模型(DEV\_STANDARD),且设备还未被启动时就启动该设备。这是必需的,因为在标准流传输模型中,应用程序可以在第一次调用 SIO\_get 之前调用 SIO\_select。

被 SIO\_select 传递进来的信号灯(sem)被赋予设备特有对象的信号灯(objptr->ready)。为了更好地理解函数 Dxx\_ready,请参考下面对 SIO\_select 的详细介绍。

SIO\_select 的行为可用伪代码概括如例 7-33。

### 例 7-33 SIO\_select 伪代码

```

/*
 * ===== SIO_select =====

```

```

    * /
Uns SIO_select(streamtab,n,timeout)
    SIO_Handle    streamtab[];    /* array of streams */
    Int           n;               /* number of streams */
    Uns           timeout;         /* passed to SEM_pend() */
{
    Int           i;
    Uns           mask = 1;        /* used to build ready mask */
    Uns           ready = 0;       /* bit mask of ready streams */
    SEM_Handle    sem;            /* local semaphore */
    SIO_Handle    * stream;        /* pointer into streamtab[] */

    /*
     * For efficiency, the "real" SIO_select() doesn't call
     * SEM_create() but instead initializes a SEM_Obj on the
     * current stack.
     */
    sem = SEM_create(0, NULL);

    stream = streamtab;

    for (i = n; i > 0; i--, stream++) {
        /*
         * call each device ready function with 'sem'

```

```

    * /
    if ('Dxx_ready(device, sem)') {
        ready = 1;
    }
}

if (!ready) {
    /* wait until at least one device is ready */
    SEM_pend(sem, timeout);
}

ready = 0;

stream = streamtab;

for (i = n; i > 0; i--, stream++) {
    /*
     * Call each device ready function with NULL.
     * When this loop is done, ready will have a bit set
     * for each ready device.
     */
    if ('Dxx_ready(device, NULL)') {
        ready |= mask;
    }
    mask = mask << 1;
}

```

```
    return (ready);  
}
```

SIO\_select 对每个 Dxx 设备都调用了两次 Dxx\_ready。第一次调用将一个可用的信号灯注册到设备特有对象,第二次调用(设置 sem = NULL)则用来注销设备特有对象中的信号灯。

每次,Dxx\_ready 函数都将 sem 保存在设备特有对象中(例 7-32 中,objptr->ready = sem)。当 I/O 操作完成(即一个缓冲区被填满或被清空),且 objptr->ready 不是 NULL,SEM\_post 就会被调用来发布 objptr->ready。

只要至少一个设备就绪或者 SIO\_select 函数的超时参数为零,SIO\_select 就不会阻塞。否则,SIO\_select 就会等待 ready 信号灯,直到至少一个设备就绪,或者超时值耗尽。

考虑设备在超时发生之前就绪的情况:不管哪个设备第一个就绪,ready 信号灯都会被发布。然后 SIO\_select 再次对每个设备调用 Dxx\_ready,这时设置参数 sem=NULL。这有两个作用:第一,任何其他就绪设备都不会发布 ready 信号灯,这就阻止了设备发送一个已经不存在的信号灯,因为 ready 信号灯是 SIO\_select 的一个局部存储器变量;第二,通过第二次查询每个设备,SIO\_select 可以找出在第一次调用 Dxx\_ready 之后就绪的设备,然后在就绪掩模 mask 中为这些设备设置相应的比特位。

## 7.17 设备类型

DSP/BIOS 中的设备有两种主要类型：终端(terminating)设备和可堆叠(stackable)设备。每种类型都提供相同的设备函数,但在实现上又稍有不同。终端设备是一种数据源出(data source)或者数据吸纳(data sink)设备。而堆叠设备则是一种不源出或不吸纳数据的设备,但它可通过使用 DEV 函数与其他设备进行数据传输。从图 7-12 中可以看出堆叠设备和终端设备在流中的位置和作用。

在种类繁多的堆叠设备中,有两种不同的类型,分别是原地(in-place)堆叠设备和拷贝堆叠设备。原地堆叠设备会对缓冲区的数据进行原地操作。而拷贝堆叠设备则在处理数据时将结果数据存放到另一个缓冲区。对于那些处理后产生的数据比所接收的数据更多的设备(例如数据解包设备或者音频解码驱动),或者是那些需要访问整个缓冲区来产生每一点的输出而因此不能覆盖输入数据的设备(例如 FFT 驱动),数据的复制搬移是必需的。与原地堆叠设备相比,拷贝堆叠设备必须有自己的缓冲区,所以它们的实现方式是不同的。

先来看一个终端设备的缓冲区传输过程,如图 7-13。该过程与 DSP/BIOS 的相互作用相对简单,其主要的复杂性在于那些控制和传输数据到物理设备的代码。



图 7-12 堆叠设备和终端设备

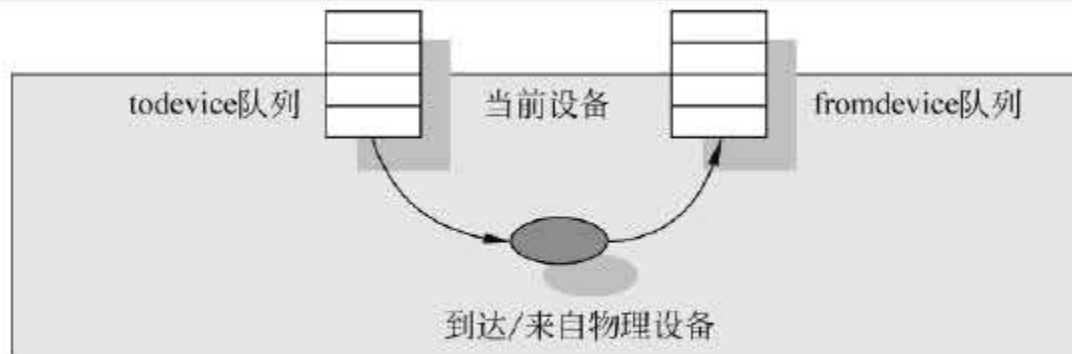
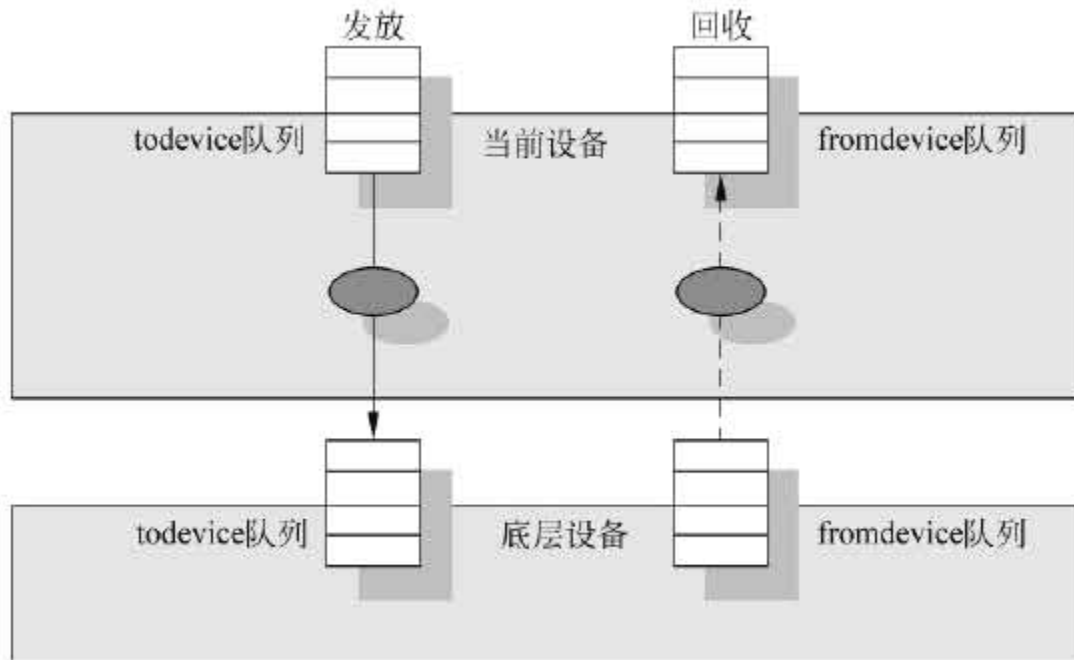


图 7-13 一个终端设备中的缓冲区传输

图 7-14 给出了原地堆叠设备的缓冲区传输流程,所有的数据处理都在原来的缓冲区内进行。原地堆叠设备的缓冲区处理和传输虽然简单,但其通用性不及拷贝堆叠设备。

图 7-15 给出了拷贝堆叠设备的缓冲区传输过程。注意实际上从流中任务端传下来的缓冲区不会移动到流中的设备端,两个缓冲池始终相互独立。这很重要,因为在拷贝堆叠设备中,任务端的缓冲区和设备端的缓冲区的大小可能不同。还要注意缓冲区到达设备端的顺序,必须能够支持发放/回收(SIO\_ISSUERECLAIM)流传输模型。





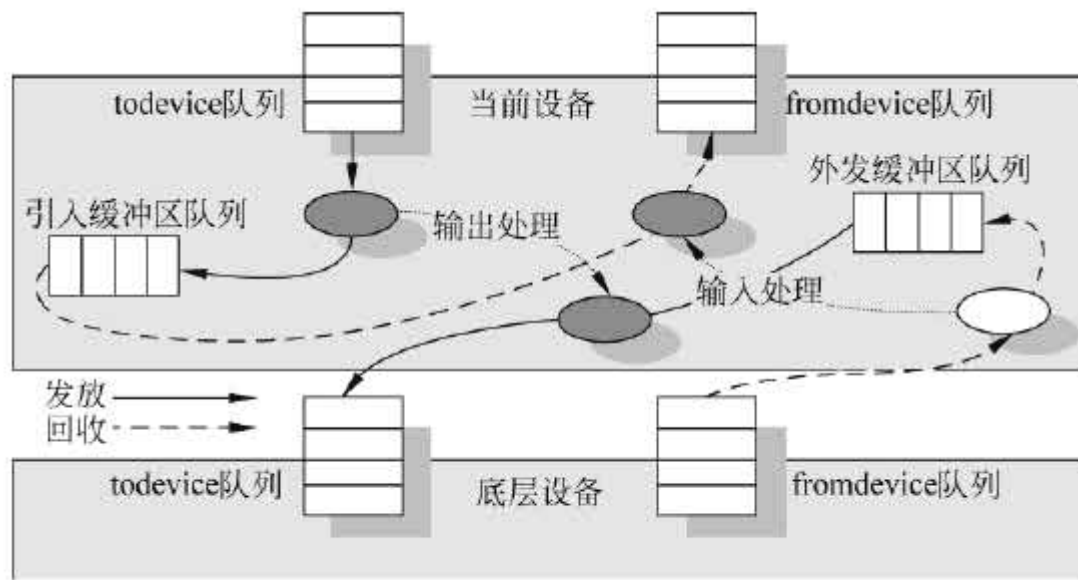


图 7-15 拷贝堆叠驱动