

实时调试集成环境 DSP/BIOS的应用

讲演： 管庆

主要内容

电子科技大学 

- DSP/BIOS的概述- Introduction
- DSP/BIOS的线程管理/调度-Real Time Scheduling
- DSP/BIOS提供的实时分析工具- Real Time Analysis Tools
- DSP/BIOS高级应用：线程、通讯、同步、硬件中断、数据交换、内核评估、优化
- DSP/BIOS应用的例子-Example

DSP/BIOS的概述

Part 1 - Introduction

DSP/BIOS 概述



- DSP/BIOS是一个可升级的实时内核。它主要是为需要任务的实时调度和同步，主机-目标系统通讯和实时监测的应用而设计的。
- DSP/BIOS 集成到 CCS 中的, 不需要额外的费用。
- DSP/BIOS 是 TI's eXpressDSP 技术 的重要组成部分。

DSP/BIOS的组件

- 抢先式多任务内核

- 配置工具



- 实时分析工具



- **DSP/BIOS API :**
提供近200个DSP/BIOS API给用户。

继续

DSP/BIOS 配置工具

Estimated Data Size: 7593 Est. Min. Stack Size (MAUs): 528

TSK - Task Manager objects by priority

- System
 - Global Settings
 - MEM - Memory Section Manager
 - SYS - System Settings
 - Instrumentation
 - LOG - Event Log Manager
 - STS - Statistics Object Manager
 - Scheduling
 - CLK - Clock Manager
 - PRD - Periodic Function Manager
 - HWI - Hardware Interrupt Service Routine Manager
 - SWI - Software Interrupt Manager
 - TSK - Task Manager**
 - task0
 - task1
 - task2
 - TSK_idle
 - IDL - Idle Function Manager
 - Synchronization
 - SEM - Semaphore Manager
 - MBX - Mailbox Manager
 - QUE - Atomic Queue Manager
 - LCK - Resource Lock Manager
 - Input/Output
 - RTDX - Real-Time Data Exchange Settings
 - HST - Host Channel Manager
 - PIP - Buffered Pipe Manager
 - SIO - Stream Input and Output Manager
 - CSL - Chip Support Library

Priority 15 (Highest)

- Priority 14
- Priority 13
- Priority 12
- Priority 11
- Priority 10
- Priority 9
- Priority 8
- Priority 7
- Priority 6
- Priority 5
- Priority 4
- Priority 3
- Priority 2
- Priority 1
 - task0
 - task1
 - task2
- Priority 0 (Reserved for the idle task)
 - TSK_idle
- Priority -1 (Suspended tasks)

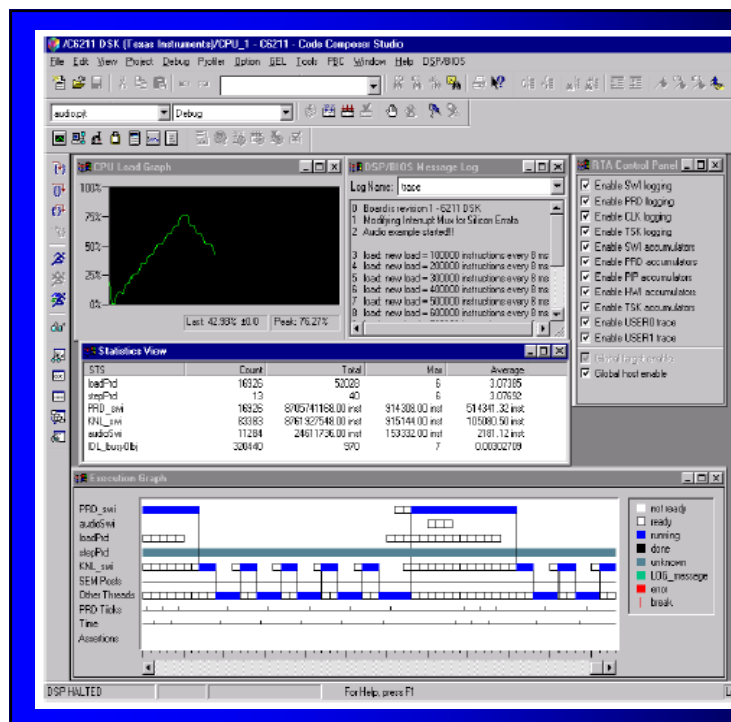
返回



电子科技大学

图形界面的静态配置工具

- 设定DSP/BIOS库中的各种参数
- DSP/BIOS的裁减控制
- 创建目标应用程序使用的对象（object），以便使用DSP/BIOS提供的API函数。
- 使用CSL配置外设
- 对目标系统的初始化配置



实时分析工具

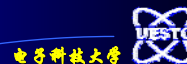
返回

DSP/BIOS 的 API模块



- ◆ **CLK**模块：用于片上的定时器管理，设置定时器中断的间隔时间。
- ◆ **HST**模块：用于实现主机与目标系统间数据的输入或输出。
- ◆ **HWI**模块：用于硬件中断管理，可设置相应的中断服务子程序。
- ◆ **IDL**模块：用于管理后台idle函数，该类函数具有最低优先级。
- ◆ **LOG**模块：用于事件的记录显示。例如，可以通过该API输出调试信息。

DSP/BIOS 的 API模块



- ◆ **MEM**模块：用于定义目标系统的内存使用。系统根据此信息自动产生.cmd文件。
- ◆ **PIP**模块：用于数据管道管理，可以实现线程间的数据交换。
- ◆ **PRD**模块：用于实现周期性的函数。该类函数的执行频率，可以由CLK模块或自己调用PRD_tick函数决定。
- ◆ **RTDX**模块：用于主机与DSP目标系统间的实时数据传递。
- ◆ **DEC**模块：设备驱动程序接口。

DSP/BIOS 的 API模块



- ◆**STS模块**：用于状态统计管理，可以在CCS下查看这些统计参数。
- ◆**SWI模块**：用于管理软件中断。CCS将运行队列中的软件中断，并可以设置15个优先级，但都比硬件中断低。
- ◆**SIO模块**：流式I/O管理模块，可用于设备驱动模块与任务或软件中断之间的数据交换。
- ◆**MXB模块**：管理邮箱，实现任务间同步或通讯。
- ◆**QUE模块**：用于任务或线程的队列管理。
- ◆**SEM模块**：旗语管理，用于任务或线程间的同步。

几个与型号有关的模块



- **MEM模块**
- **GBL模块**
- **C54XX, C62XX, C64XX**
- **CSL片级支持库(Chip Support Library)**

DSP/BIOS 的 API模块



- 几种DSP/BIOS API函数可以触发SWI线程:

- SWI_andn

- SWI_dec

- SWI_inc

- SWI_or

- SWI_post

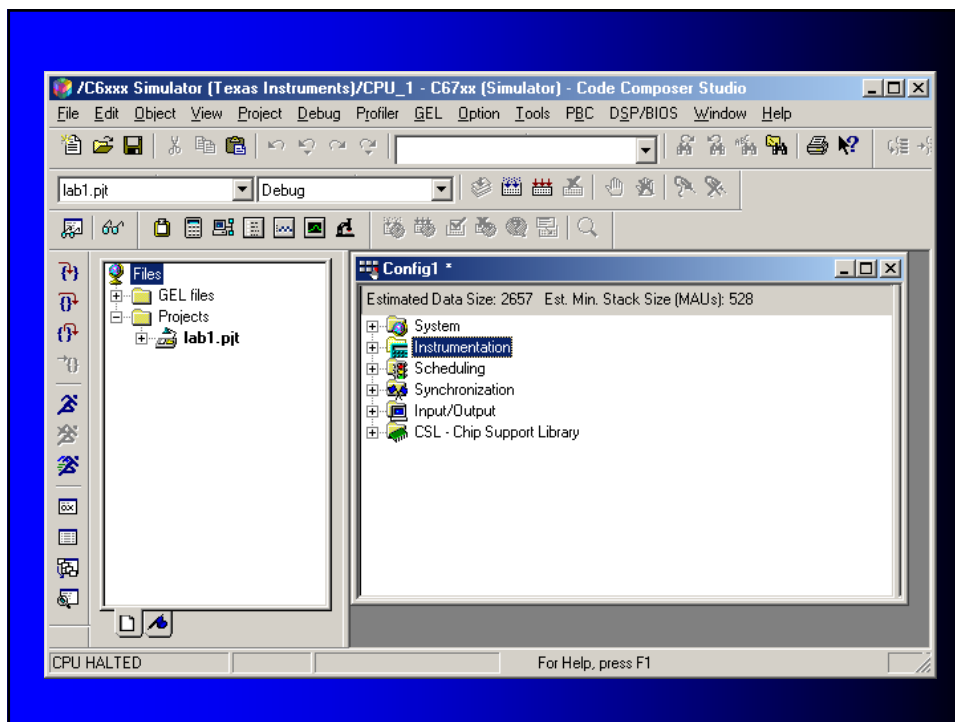
条件启动,与邮箱初始值做“与”、“或”、“加”、“减”运算后决定是否启动该线程。

→直接启动,与邮箱初始值无关。

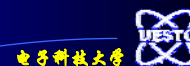
创建DSP/BIOS配置



- 从已有的DSP/BIOS配置文件中修改获得
- 在“File”菜单中新建配置文件, CCS 提供了许多模板共选择
- 将创建的配置文件保存到你的工作目录下
- 将创建的配置文件 (*.cdb) 添加到project工程文件中。



创建DSP/BIOS配置



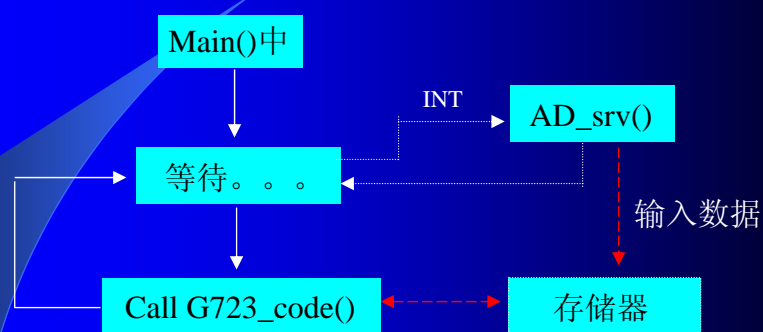
- 配置工具将**自动创建**下面的文件（‘C6000为例’）：
 - 当保存配置文件（如my.cdb）
- | | |
|-------------|--------------------------|
| • mycfg.s62 | Assembly file |
| • mycfg_c.c | C file |
| • mycfg.h | Header file for C |
| • mycfg.h62 | Header file for assembly |
- “mycfg.cmd”也会自动生成，但需要手工添加到工程文件中。

注意：用户需要**自己将 *.cdb 和 *.cmd 添加**到工程文件中。

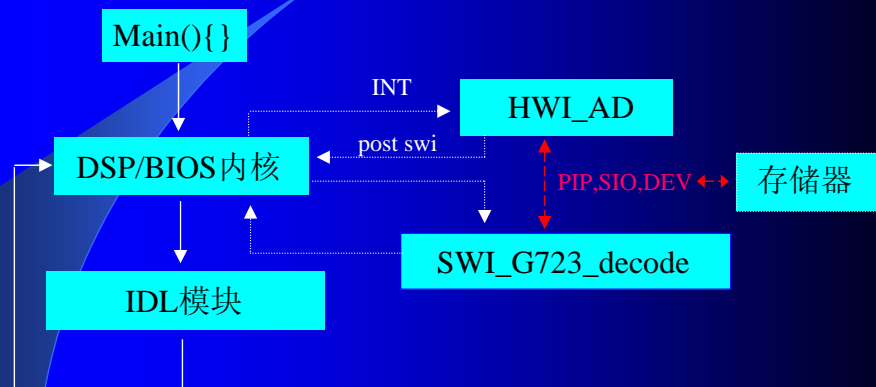
DSP/BIOS的线程管理与调度

Part 2 - Real Time Scheduling

不使用DSP/BIOS时的程序结构



使用DSP/BIOS时的程序结构



DSP/BIOS应用程序的结构

- DSP/BIOS API提供可伸缩的实时核，还提供了有优先级的多线程处理。它是专为那些需要实现实时调度、同步以及通讯的应用程序而设计。在一个包含DSP/BIOS内核的应用程序按优先级从低到高有四种主要线程：

- 后台线程（IDL线程）
- 任务（TSK模块）
- 软件中断（SWI模块）
- 硬件中断（HWI模块）

DSP/BIOS 的线程类型



- ◆ **HWI** 的优先级由硬件中断决定
每个硬件中断提供一个ISR
- ◆ **SWI** 有14个优先级
同一优先级可以有多个SWI线程.
- ◆ **TSK** 有15个优先级
同一优先级可以有多个TSK线程.
- ◆ **IDL** 线程包括多个IDL函数
这些函数会连续反复调用.

HWI 线程由硬件中断触发！

IDL 为后台运行的线程.

线程选择的一般原则

- **严格的实时性**：如果线程的执行需要严格的实时性，而线程执行需要的时间又很少时，你可以使用**硬件中断或时钟函数**来完成。CLK时钟函数也是在硬件中断中执行的。
- **部分实时性**：执行时间较长，使用**SWI软件中断或TSK任务线程**来完成一些非实时性的处理任务。这样可以减少中断的潜伏期，提高响应实时性请求的能力。

线程选择的一般原则



- 周期性的服务：需要周期性或在固定的时间间隔内完成处理任务，使用**PRD周期性函数**来完成。
- 不需要实时性：线程只需要在**后台进行一些不关键的处理**，比如收集统计数据、与自己交换检测数据等等。这种情况，我们建议使用**IDL线程**。

线程之间的不同



- 线程的**等待**和执行的**速率**。
- 线程的不同状态：**HWI**只有运行状态；**SWI**有就绪和运行状态；**TSK**有就绪（**Ready**），挂起/等待（**Pending**），运行（**Running**），结束（**Done**）
- 线程的**优先级**

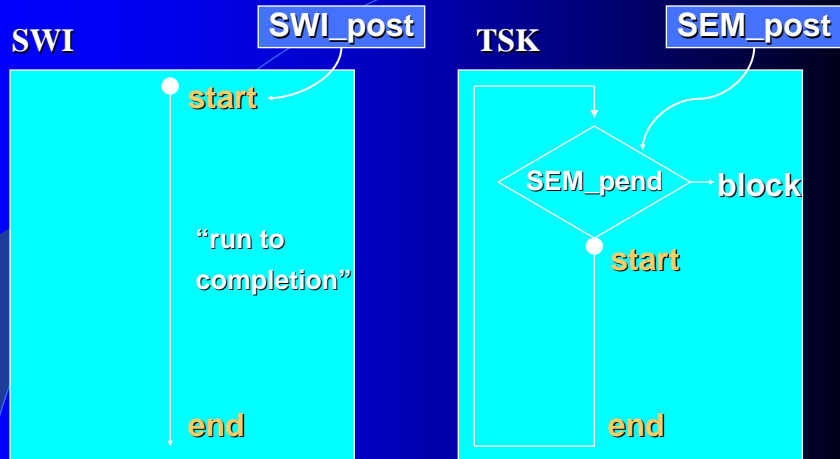
线程之间的不同

- 对堆栈的需求
 - 要使用系统堆栈，请使用SWI线程.
 - 需要独立的堆栈时，应该使用TSK线程.
- 线程之间的同步要求和通讯方式
 - SWI 和TSK使用不同的方法 .
- 用户自己的习惯

任务线程的特点

- 所有任务线程的优先级都低于硬件中断和软件中断。
- 任务线程和软件中断不同的是：一个任务线程可以中断自己的运行，转而运行其它的任务。当某些条件满足后又恢复继续执行。
- 任务的切换不是任何情况下都被允许，只有在中断发生（有更高优先级的线程要运行），或某些任务模块的API函数调用时才会发生。

SWI 或 TSK 的区别



SWI 不能被挂起 (pend)。
SWI 一般总是由该线程函数返回。

TSK 仅在任务线程结束是返回，
或者是一个无限循环！

```
void main (void)
{
    /* Put all your setup code here */
    return; /*DSP BIOS starts after the return */
}

/* Hardware Interrupt */
void timerIsr (void)
{
    /* Put your code here */
    SWI_post (&SWI_for_algorithm_1);
    SEM_post (&taskOneSem);
}

/*Software Interrupt */
void algorithm_1 (void)
{
    /* Put your code here */
}

/* Task */
void ProcessTask (void)
{
    while (1)
    {
        SEM_pend (&taskOneSem, SYS_FOREVER);
        /* Insert your code here */
    }
}
```

DSP/BIOS的实时 分析工具

Part 3 - Real Time Analysis Tools

Introduction



- 传统的代码分析（调试）方法是在处理器暂停运行后，通过观察变量和存储单元来实现。
- 实时分析需要不停止处理器的运行而同时获得观测数据
- DSP/BIOS中的 API模块和CCS中的 Plug-ins插件能帮助程序员实现实时分析。

Introduction

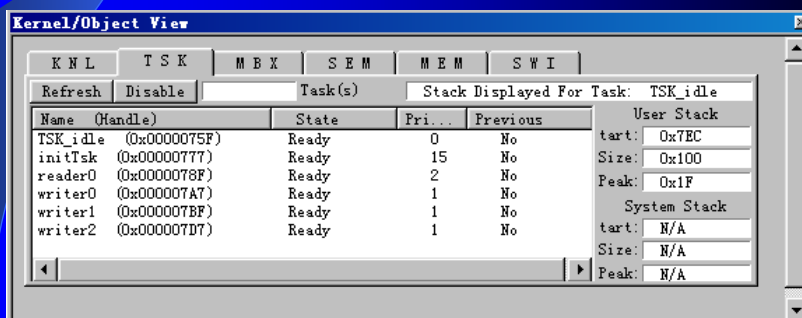


- 如何实现不停止处理器运行而获得需要的观测数据呢？
 - DSP目标系统与host主机的通讯是在DSP/BIOS的IDL线程中完成的。
 - Host主机在接收到数据后立即处理这些数据，从而得到需要的结果。处理软件可以通过CCS的插件（Plug-in）实现。
- For more details see Chapter 3 of the DSP/BIOS Users Guide ([Links\SPRU303.pdf](#)).

内核/模块查看窗口



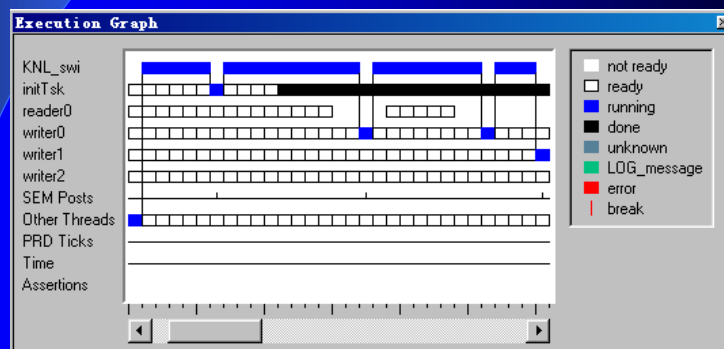
- ▼ 通过内核/模块查看调试工具观察运行的DSP/BIOS模块的当前状态。例如一个DSP/BIOS程序的所有任务模块运行状态。通过该窗口，你可以了解这些任务的优先级，运行状态，堆栈使用情况等



程序模块执行状态图



- 在这个窗口中，我们可以看见程序中的各个线程运行状态图。其中HWI硬件中断服务程序，SWI软件中断，TSK任务，旗语模块，周期函数，以及时钟模式信号。另外，还包括了内核线程核其他的IDL空闲线程。



状态统计窗口

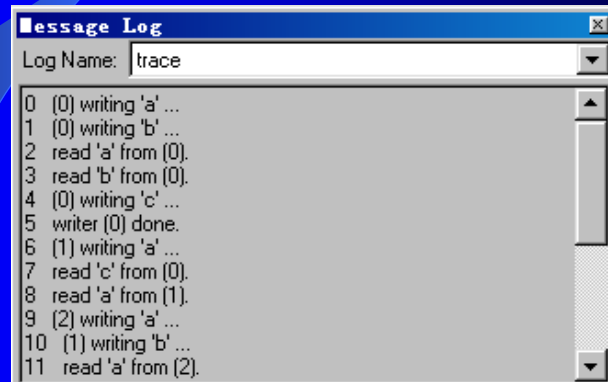


- 搜集并报告内核各个模块的执行情况。每个模块类型收集的状态统计数据代表了不同的值，有不同的单位，如指令数、时间等。

Statistics View					
STS	Count	Total	Max	Average	
processing_SWI	0	0 inst	-2147483648 inst	0.00	
TSK_idle	0	0 inst	-2147483648 inst	0.00	
IDL_busyObj	835	-170602	-136	-204.314	

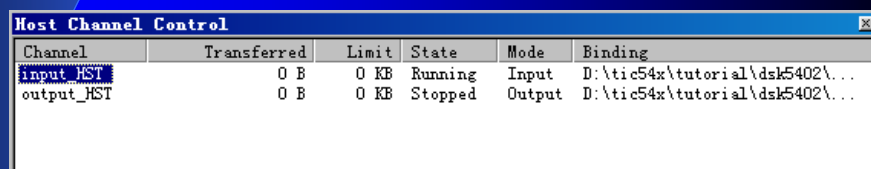
信息显示窗口

- ▼ DSP/BIOS内核专门提供了一个信息输出窗口，用来替代标准C下的stdout窗口，使用DSP/BIOS内核提供的LOG_printf函数比标准C中的printf函数有更高的效率。



主机通道控制

- ▼ 在这个窗口内，我们可以看到由我们的程序定义的主机通道，如input_HST, output_HST，用户可以使用这个窗口来指定与这些通道相连接的数据文件，这些文件被存放在主机（如PC机）的磁盘中。（等效“探针工具”）



Channel	Transferred	Limit	State	Mode	Binding
input_HST	0 B	0 KB	Running	Input	D:\tic54x\tutorial\dsk5402\...
output_HST	0 B	0 KB	Stopped	Output	D:\tic54x\tutorial\dsk5402\...

DSP/BIOS的高级应用

Part 4 – 同步、通信、硬件中断、数据交换、优化

任务的通讯和同步

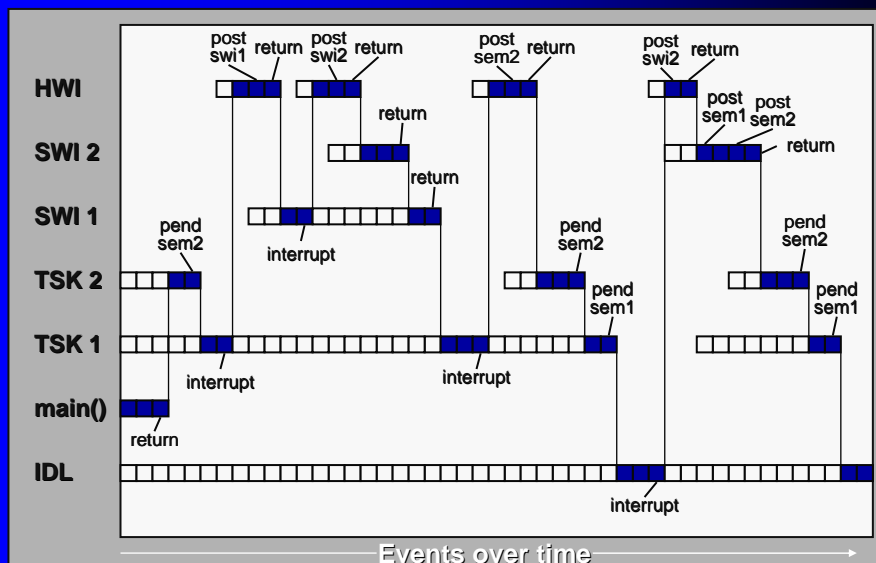


- MBX邮箱管理模块：两个任务线程可以通过邮箱来完成数据的交换。
- SEM旗语管理模块：DSP/BIOS内核提供的旗语实际上是信号量旗语，该旗语管理模块通过对旗语的计数来完成任务线程的同步和相互作用。
- QUE队列管理模块：在任务间或任务与软件中断、硬件中断间共用队列

硬件中断管理

- 硬件中断管理器包含了所有的硬件中断，这些硬件中断根据具体的目标系统按照优先级从高到低的顺序排列。
- 通过这个硬件中断管理器，能够为每个DSP中的硬件中断配置中断服务程序（ISR）。
- 需要特别提醒的是若使用DSP/BIOS内核开发应用程序，用户便不能随意修改中断向量表的位置
- 推荐使用DSP/BIOS的硬件中断调度功能。

Thread Preemption Example



对DSP/BIOS内核的评估

- 开发者只需要**根据分析**，计算DSP/BIOS对象以及调用哪一个函数对象的总数和出现频率来确定**估算开销**。
- **手册给出了DSP/BIOS内核中主要API函数所消耗时间的计算方法以及具体的指令数。**
- 通常情况可使用CCS中提供的DSP/BIOS分析工具确定DSP/BIOS的开销。例如，实时分析工具中的**CPU负载图**就是常用的工具之一。

具体评估的例子

- 在一个应用程序中存在一个**HWI硬件中断**，并在该硬件中断中启动（post）一个**SWI软件中断**。HWI硬件中断与SWI软件中断之间通过一个**PIP管道**传递数据。硬件中断出现的**频率**为250次/秒。

具体评估的例子

- 在C54X平台上硬件中断的进入和退出至少需要 $88+82=170$ 个指令周期数，SWI软件中断的启动至少需要98个指令周期，而PIP管道操作需要452个指令周期。
- 这样整个每秒中DSP/BIOS内核至少会消耗 $250 \times (170+98+452) = 180000$ 个指令周期。如果我们在100MIPS的TMS320VC5410上运行该程序，DSP/BIOS的最小开销为180000个指令周期/秒或0.18MIPS，相当于CPU负载的0.18%。

利用配置工具对DSP/BIOS进行优化

- 对DSP/BIOS的优化一般从两个角度来实现：对速度的优化和对程序大小的优化。
- 提高DSP/BIOS的应用程序的执行性能的建议：
 1. 使用不同的程序函数仔细选择线程的类型。
 2. 把系统堆栈安置在片上（on-chip）内存中。
 3. 减小时钟中断频率。
 4. 增加流式输入输出缓冲器的大小。

减小DSP/BIOS大小

- 对DSP/BIOS后台IDL循环的优化
- 关闭DSP/BIOS的任务管理功能
- 禁止使用动态堆
- 禁止CLK时钟管理
- 禁止实时数据交换（RTDX）功能
- 关闭实时分析功能
- 去掉CSL片级支持库

减小DSP/BIOS大小

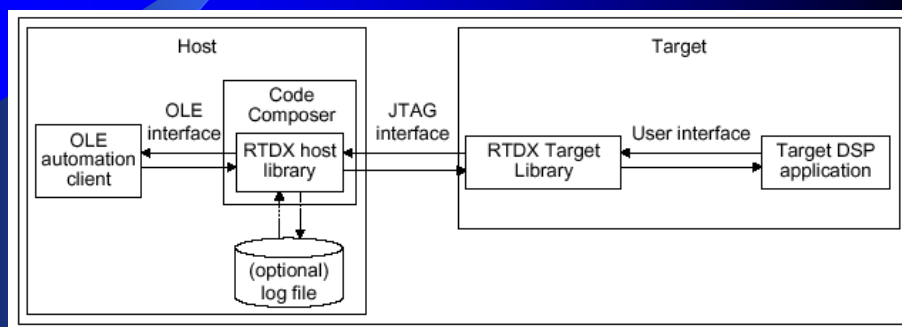
- 去除系统处理函数
 - 最小化数据存储器
 - 选择静态或动态对象创建
- 便于**DSP/BIOS**分析工具读取访问
简化代码长度
改善运行时间性能

RTDX: 实时数据交换

- ◆ RTDX 允许主机与DSP目标系统之间交换数据 (**during IDL**).
- ◆ RTDX在IDL线程(**by default**)中运行, 它的优先级最低, 一般不会对用户程序产生影响.
- ◆ DSP/BIOS的实时分析工具也使用RTDX.
- ◆ 速度受限于:
 - ◆ JTAG connection type (parallel, PCI, etc.).
 - ◆ DSP activity level.

RTDX: Flow of Data

- Code Composer Studio控制在主机与DSP目标板之间数据交换.



DSP/BIOS实例

**Part 5 – 线程优先级、硬件中断、
旗语同步、RTDX的应用以及
DTMF产生/解码的DSP/BIOS实现**

DSP/BIOS示例内容



- DSP/BIOS基础示例
- SWI软件中断以及PRD周期函数优先级控制
- TSK任务优先级控制
- TSK任务的通讯与同步
- HWI使用实例：不同线程的应用
- DTMF 在DSP/BIOS下的实现

DSP/BIOS基础示例



- 这个例子（`bios_ccs2`）包括了SWI软件中断线程、TSK任务线程、RTDX技术以及STS统计对象等DSP/BIOS基本功能的应用。
- 涉及到SWI软件中断对象的启动，定时器模块的应用，PRD周期模块的应用以及TSK任务模块函数调用。
- 关于DSP/BIOS配置文件（.CDB），特别是Global Settings 和MEM的设置。
- 从C5410实验板 -> C5402实验箱移植。

不同线程的应用



- HWI、SWI、TSK以及IDL是DSP/BIOS下的四种基本线程。
- 本节给出了一个音频处理应用的例子，并使用SWI、TSK、以及IDL三个不同线程实现。
- 分别在`g723_idl`、`g723_swi`以及`g723_tsk`三个目录下。
- HWI硬件中断线程一般不会直接用来处理算法。（P325 § 6.3.2）

TSK任务优先级控制

- TSK线程可以中断自己,转而运行其他任务。
- TSK被中断后,当条件满足后,又可恢复。
- TSK线程的优先级比HWI和SWI低。
- TSK任务的切换不是任何时间都允许,只有当中断发生;或某些任务调用一些API函数时。
- 本小节的例子用来说明任务的切换(调用TSK_yield函数),以及优先级对TSK任务对象的影响。

工程文件: **tsktest**

Answer A ↵	Answer B ↵	Answer C ↵
Loop 0: Task 0 Working	Loop 0: Task 0 Working	Loop 0: Task 0 Working
Loop 1: Task 0 Working	Loop 0: Task 1 Working	Loop 1: Task 1 Working
Loop 2: Task 0 Working	Loop 0: Task 2 Working	Loop 2: Task 2 Working
Loop 3: Task 0 Working	Loop 1: Task 0 Working	Loop 3: Task 0 Working
Loop 4: Task 0 Working	Loop 1: Task 1 Working	Loop 4: Task 1 Working
Task 0 DONE↵	Loop 1: Task 2 Working	Task 0 DONE↵
Loop 0: Task 1 Working	Loop 2: Task 0 Working	Task 1 DONE↵
Loop 1: Task 1 Working	Loop 2: Task 1 Working	Task 2 DONE↵
Loop 2: Task 1 Working	Loop 2: Task 2 Working	
Loop 3: Task 1 Working	Loop 3: Task 0 Working	
Loop 4: Task 1 Working	Loop 3: Task 1 Working	
Task 1 DONE↵	Loop 3: Task 2 Working	
Loop 0: Task 2 Working	Loop 4: Task 0 Working	
Loop 1: Task 2 Working	Loop 4: Task 1 Working	
Loop 2: Task 2 Working	Loop 4: Task 2 Working	
Loop 3: Task 2 Working	Task 0 DONE↵	
Loop 4: Task 2 Working	Task 1 DONE↵	
Task 2 DONE↵	Task 2 DONE↵	

TSK任务的通讯与同步

- TSK线程可以通过MBX、QUE等通讯。
- TSK线程可以通过SEM、MBX等实现同步。
- [例6-2-1]使用一个旗语（SEM）使三个任务轮流对一个队列进行写操作。 `\semtest`
- [例6-2-2]使用旗语使两个任务共享同一数据结构，实现互斥访问（Mutual Exclusion）
`\mutex`
- [例6-2-3]使用邮箱在两个任务之间发送消息。
`\mbxtest`

例[6-2-1]

Answer A

semtest example started

```
(0) writing 'a' ...
read 'a' from (0).
(0) writing 'b' ...
read 'b' from (0).
(0) writing 'c' ...
read 'c' from (0).
writer (0) done.
(1) writing 'a' ...
read 'a' from (1).
(1) writing 'b' ...
read 'b' from (1).
(1) writing 'c' ...
read 'c' from (1).
writer (1) done.
(2) writing 'a' ...
read 'a' from (2).
(2) writing 'b' ...
read 'b' from (2).
(2) writing 'c' ...
read 'c' from (2).
reader done.
writer (2) done.
```

Answer B

semtest example started

```
(0) writing 'a' ...
read 'a' from (0).
(1) writing 'a' ...
read 'a' from (1).
(2) writing 'a' ...
read 'a' from (2).
(0) writing 'b' ...
read 'b' from (0).
(1) writing 'b' ...
read 'b' from (1).
(2) writing 'b' ...
read 'b' from (2).
(0) writing 'c' ...
read 'c' from (0).
writer (0) done.
(1) writing 'c' ...
read 'c' from (1).
writer (1) done.
(2) writing 'c' ...
read 'c' from (2).
reader done.
writer (2) done.
```

Answer C

semtest example started

```
(0) writing 'a' ...
(0) writing 'b' ...
(0) writing 'c' ...
read 'a' from (0).
read 'b' from (0).
read 'c' from (0).
writer (0) done.
(1) writing 'a' ...
(1) writing 'b' ...
(1) writing 'c' ...
read 'a' from (1).
read 'b' from (1).
read 'c' from (1).
writer (1) done.
(2) writing 'a' ...
(2) writing 'b' ...
(2) writing 'c' ...
read 'a' from (2).
read 'b' from (2).
read 'c' from (2).
reader done.
writer (2) done.
```

[例6-2-3]

Answer A

```
(0) writing 'a' ...
(0) writing 'b' ...
(0) writing 'c' ...
writer (0) done.
read 'a' from (0).
read 'b' from (0).
read 'c' from (0).
(1) writing 'a' ...
(1) writing 'b' ...
(1) writing 'c' ...
writer (1) done.
read 'a' from (1).
read 'b' from (1).
read 'c' from (1).
(2) writing 'a' ...
(2) writing 'b' ...
(2) writing 'c' ...
writer (2) done.
read 'a' from (2).
read 'b' from (2).
read 'c' from (2).
timeout expired
for MBX_pend()
reader done.
```

Answer B

```
(0) writing 'a' ...
read 'a' from (0).
(0) writing 'b' ...
read 'b' from (0).
(0) writing 'c' ...
writer (0) done.
read 'c' from (0).
(1) writing 'a' ...
read 'a' from (1).
(1) writing 'b' ...
read 'b' from (1).
(1) writing 'c' ...
writer (1) done.
read 'c' from (1).
(2) writing 'a' ...
read 'a' from (2).
(2) writing 'b' ...
read 'b' from (2).
(2) writing 'c' ...
writer (2) done.
read 'c' from (2).
timeout expired
for MBX_pend()
reader done.
```

Answer C

```
(0) writing 'a' ...
(0) writing 'b' ...
read 'a' from (0).
read 'b' from (0).
(0) writing 'c' ...
writer (0) done.
(1) writing 'a' ...
read 'c' from (0).
read 'a' from (1).
(2) writing 'a' ...
(1) writing 'b' ...
read 'a' from (2).
read 'b' from (1).
(2) writing 'b' ...
(1) writing 'c' ...
writer (1) done.
read 'b' from (2).
read 'c' from (1).
(2) writing 'c' ...
writer (2) done.
read 'c' from (2).
timeout expired
for MBX_pend()
reader done.
```

DTMF 在DSP/BIOS下的实现

- 已经在CCS下完成DTMF码的产生和解调。
- 分析实现过程，主要有以下几个任务：
 1. **DTMF码的产生**：分双音与静音，分别持续50ms。
 2. **DTMF的解码运算**：每收到一个A/D数据的处理并完成DTMF码的判别。
 3. **A/D及D/A中断服务程序**：DA完成双音码的数模转换；AD完成数据的接收并设置数据到达标志（全局变量）。
 4. **'AC01'芯片的初始化**：设置采样率等参数，需要使用串口发送中断。

DSP/BIOS下线程考虑



- 拨号任务：定义周期性模块（PRD）每5秒运行一次FXN_Dial_Number()函数。
- 该函数将需要拨出的号码缓冲设置好，初始化全局计数变量和指针，并开启串口发送中断，开始拨号。
- 拨号缓冲中的数字为拨出的号码，0xff表示静音期，当遇到0xffff时，拨号结束，关闭串口发送中断。
- 注意：FXN_Dial_Number将反复运行！

拨号任务相关函数



transmit(): 串口发送中断服务函数，并完成对发送数据的计数以便确定拨号音与静音的时间。

set_freq_coff(NowTel): 当前拨号数字所需要的频率系数。

iir_to_dtmf(): 根据前面的频率系数，产生响应的正弦波形。

接收任务的考虑

- 将整个`de_dtmf()`函数定义为一个TSK对象（`_FXN_TSK_de_dtmf`）。
- 该任务线程由McBSP的接收中断每收到一个新的数据就发出一个旗语（`semaphore`）（`SEM_New_Sample`）触发。
- 同时在中断服务程序内对收到数据计数，并使用全局变量`IsNew_N`，决定是否判断输出一个收到的号码。

接收相关函数

de_dtmf(): 对每个收到的数据做DFT，并对N点数据做能量累加。

receive(): 接收中断服务子程序。

choose_code(): 判断输出一个号码。

init_mem(): 初始化接收运算的一些变量。该函数应该在`FXN_Dial_Number`中运行。

初始化AC01的考虑



```
/*The following code are used to setup AC01*/

C54_plug(20,&_start_ac01);
/* change interrupt vector */
oldmask=C54_disableIMR(0xffff);
/* close all IMR */
start_ac01(); /* in dtmf.asm, used to init
               AC01 & McBSP0 */
C54_enableIMR(oldmask);
/* restore old IMR */
/*-----End setup AC01 !-----*/
```

中断函数的插入和打开



```
#include "DTMF5410_BIOScfg.h"
#include <hwi.h>

.....
HWI_dispatchPlug(20,&plx100,NULL);
C54_enableIMR(0x20);
/* #20h: bit5->BXINT0,
   bit4->BRINT0, enable BXINT0 ,
   其他中断不影响! */
HWI_enable(); .....
```


下图是**CPU负载图**，整个占用约**60%**，
几乎都是解码消耗的。图1中**VC5410A**工
作在**100MHz**！

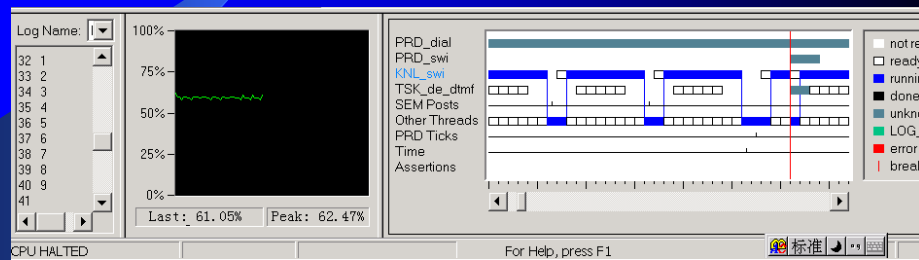


图1 任务拨号+解码任务，采样频率
 $f_s=8.333\text{kHz}$ ，拨号任务5秒/次

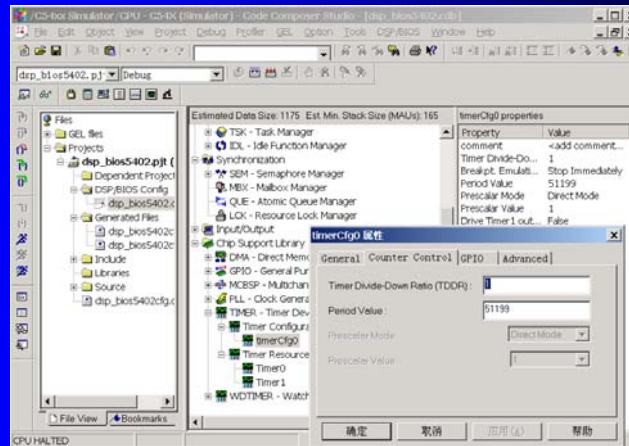
定时器触发的硬件中断

- (1) 由定时器触发的硬件中断.
- (2) 由硬件中断启动（POST）的软件中断
- (3) 由硬件中断控制的任务线程
- (4) 用于控制的旗语
- (5) 使用CSL配置定时器1

请自己完成! (参考代码dsp_bios5402)

(1) 使用CSL配置Timer2

- (1) 创建工程文件“dsp_bios5402.prj”
- (2) 添加DSP/BIOS配置文件 “dsp_bios5402.cdb”
- (3) 在 “Timer Configuration Manager”中添加一个定时器对象



(2) Setting the Hardware Interrupt

- (1) 打开CDB配置文件
- (2) 选择“HWI - Hardware Interrupt Service ...”.
- (3) 选择HWI_SINT7，并进入属性设置



(2) Setting the Hardware Interrupt

(4) 用C编写中断服务子程序.

```
void timerIsr (void)
{
    /* Put your code here */
}
```

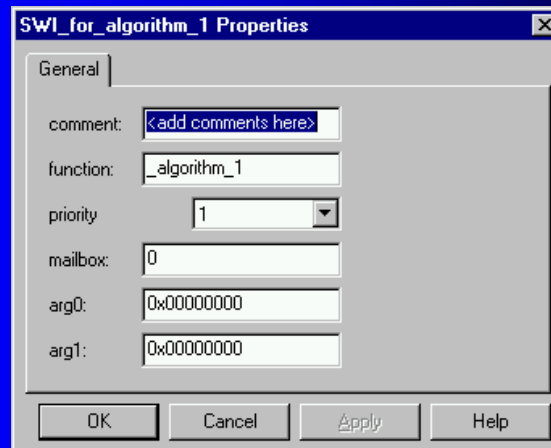
(3) Creating a Software Interrupt

(1) 在CDB配置工具中选择“SWI - Software Interrupt Manager”并创建软件中断对象“SWI_for_algorithm_1”.



(3) Creating a Software Interrupt

(2) 修改 “SWI_for_algorithm_1” 的属性参数:



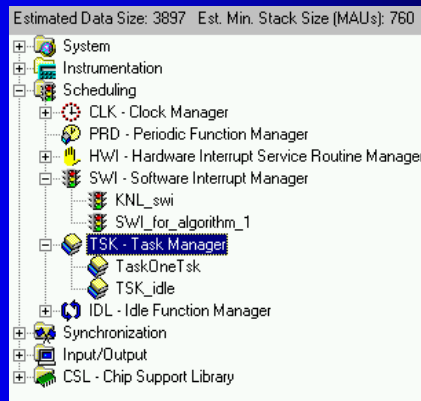
(3) Creating a Software Interrupt

(3) 编写软件中断C代码.

```
void algorithm_1 (void)
{
    /* Put your code here */
}
```

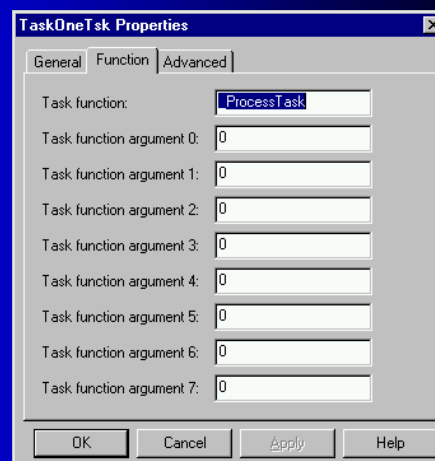
(4) Creating a Task

- (1) 打开“TSK – Task Manager”，创建一个新任务对象“TaskOneTsk”。



(4) Creating a Task

- (2) 修改属性参数:



(4) Creating a Task

(3) 编写任务函数的C代码

```
void ProcessTask (void)
{
    /* Put your algorithm here */
}
```

(5) Creating a Semaphore

(1) 选择 “SEM - Semaphore Manager”，创建一个名为 “taskOneSem” 的旗语对象

(2) 修改 旗语对象 (object) “taskOneSem” 的属性

The screenshot shows the 'taskOneSem Properties' dialog box with the 'General' tab selected. The 'comment' field contains '<add comments here>' and the 'Initial semaphore count' is set to 0. The background shows a project tree with the following structure:

- System
 - Instrumentation
 - Scheduling
 - CLK - Clock Manager
 - PRD - Periodic Function Manager
 - HWI - Hardware Interrupt Service Routine Manager
 - SWI - Software Interrupt Manager
 - KNL_swi
 - SWI_for_algorithm_1
- TSK - Task Manager
 - TaskOneTsk
 - TSK_idle
- IDL - Idle Function Manager
- Synchronization
 - SEM - Semaphore Manager
 - taskOneSem
 - MBX - Mailbox Manager
 - QUE - Atomic Queue Manager
 - LCK - Resource Lock Manager
- Input/Output
- CSL - Chip Support Library

Posting Software Interrupts and Tasks

(1) 触发软件中断线程

```
SWI_post (&SWI_for_algorithm_1);
```

(2) 触发任务线程

```
SEM_post (&taskOneSem);
```

More on Tasks...



- 当SWI软件中断线程运行时，任务线程可以被挂起。
- 任务线程可以是一个无限循环，并在循环中检测旗语。
- 任务可以被抢先

```
void ProcessTask (void)
{
    while (1)
    {
        SEM_pend (&taskOneSem, SYS_FOREVER);
        /* Insert your code here */
    }
}
```

DSP/BIOS 总结



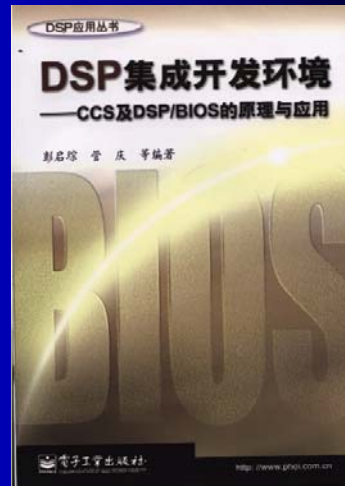
- 仅仅是一个内核，没有如TCP/IP协议栈、文件系统等外挂。
- 目前，**DSP/BIOS**不支持多处理器
- **WindRiver Systems** 公司的“**VSPWorks**”操作系统可以支持多处理器
 - www.windriver.com

DSP/BIOS总结



- 更多的参考文档（以‘C6000为例’）
 - (1) Use the help provided with the CCS (Help also includes tutorials c:\ti\tutorial\dsk6711).
 - (2) TMS320C6000 DSP/BIOS: User Guide.
 - (3) TMS320C6000 DSP/BIOS: Application Programming Interface (API) [SPRU403.](#)
 - (4) Application Report: DSP/BIOS II Timing Benchmarks on the TMS320C6000 DSP [SPRA662.](#)
 - (5) Application Report DSP/BIOS II Sizing Guidelines for the TMS320C62x DSP [SPRA667.](#)
 - (6) Other

▼ “**DSP集成开发环境-CCS及DSP/BIOS的原理与应用**”已经由电子工业出版社出版。



Thanks

欢迎登陆

<http://www.dspsolution.com>

银杏科技

—— DSP解决方案供应商

EMAIL: qguan@uestc.edu.cn