SHARC21469平台的djyos移植笔记

编写：李强 朱海兵

# 开发工具调试

## 工程的建立

基于ADSP-Sharc21469处理器的djyos移植，开发环境是ADI CrossCore Embedded Studio 1.0.2或Visual DSP++5.0。CCES 1.0.2是基于Eclipse的开发环境，djyos在Sharc21469上的移植需要重点关注的问题有两点：linker文件的编写（ldf，而非lds），汇编相关部分源码的改写。

### CCES下工程的建立

在CCES下新建基于Sharc21469的空工程，CCES会在工程中自动增加system文件夹和system.svc文件，主要内容为ADI官方定义的startup文件和ldf文件，其中包含了中断向量表IVT的定义。鉴于djyos移植的特性，这部分代码只能作为参考，不应加入工程的编译。

CCES工程的设置。在path variables中增添DJYOS\_SRC\_LOCATION基地址，同时在Paths and Symbols中增添工程总的include文件目录。如图1-1：

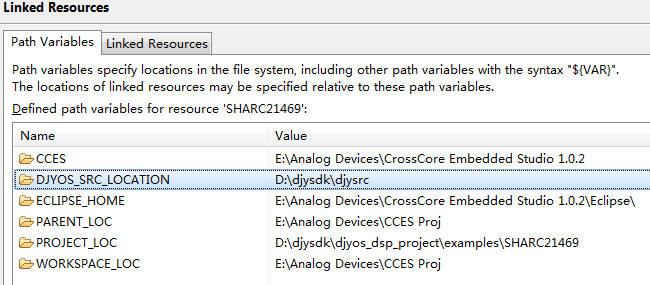


图1-1 Eclipse头文件搜索路径的设置

这里遇到一个问题，当使用宏定义的DJYOS\_SRC\_LOCATION时，总是无法寻找到相关的头文件。后改用绝对路径暂时解决了该问题。[解决方法]在environment下再增添一次DJYOS\_SRC\_LOCATION的定义，这里定义的宏才是后面的Paths and Symbols中查询include文件所能识别的路径宏定义。前面在Linked Resources中定义的路径宏是用于编译器链接源文件所用，因此后面在查询include头文件时不能用，所以需要再次定义。如图1-2所示：

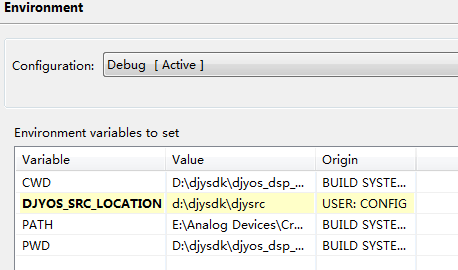


图1-2 工程头文件搜索路径的设置

Djyos中的pci总线和ip协议相关部分暂时不会用到，因此exclude出工程，不参加编译。Bsp目录下则不添加相关部分内容。Bsp目录下还有中断等硬件相关性较大的部分也暂时exclude出工程。

### Visual DSP下工程的建立

在VDSP中，中断向量表已经包含在21469\_hdr.doj文件中，其源码为21469\_hdr.asm，可在VDSP的example目录中查找修改。此处需要注意的是，修改后生产的21469\_hdr.doj应拷贝到lib根目录下，以与ldf文件中的名称对应。

将djyos的系统文件夹及工程文件夹下的源码部分链入工程，在工程文件夹下，新建了基于DSP-Sharc21469处理器的相关工程目录和文件，在系统文件夹下，增添了基于DSP-Sharc21469处理器的相关目录和文件。这部分文件暂时是从PPC的文件中拷贝而来，移植代码时需修改！

VDSP工程设置。在Compile—Preprocessor下和Link—General下都需要设置include search paths，前者是编译用，后者是链接用。两者保持一致即可。

对于在VDSP下建立工程，相对要复杂一些，因为VDSP的文件管理系统比较落后，不能导入文件夹至工程中，因此只好自己建立了。工程内的文件只能一个个链接进来，VDSP中建立的工程如图1-3所示。

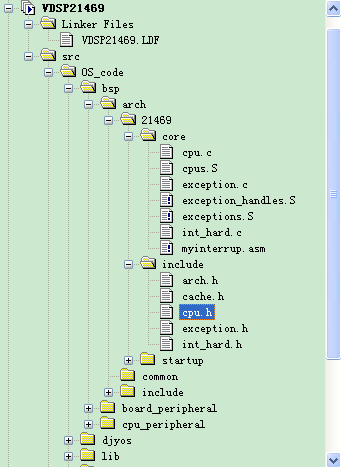


图1-3 VDSP中建立的工程

VDSP头文件搜索路径的设置也比较原始，不能如CCES一般定义公共的基地址路径，而只能在Project Option中一个一个的添加绝对路径。具体设置见图1-4所示。

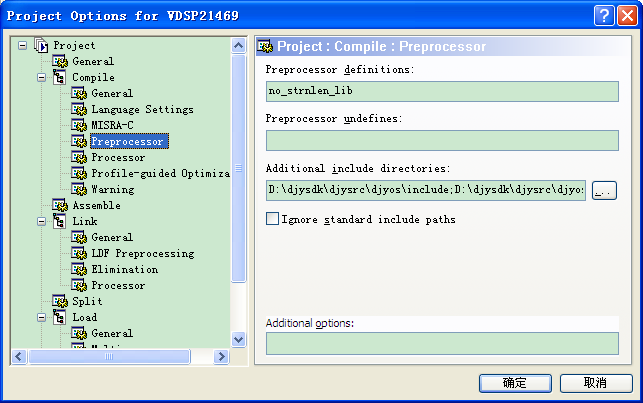


图1-4 头文件搜索路径的设置

在VDSP中还有一个小技巧，可以针对单个源文件（C或者汇编）进行不同于整个工程的特殊设置，具体方法为，在源文件上点右键，选择File Options，选择File-specific setting，然后就可以针对单个文件进行设置了。这在有些时候汇编文件不能正常搜索到头文件时非常有用。在进行优化等级的测试的时候也非常有用。如图1-5所示。

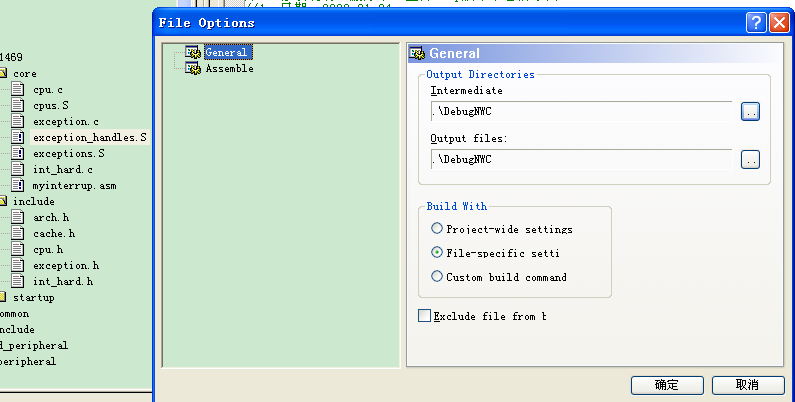


图1-5 单个文件的特殊设置

至此，build project通过，在link时候报错，是因为工程中没有ldf文件所致。

## 链接和加载过程

LDF全称Link Discription File，是由ADI定义的一种链接脚本文件格式，与常用的LDS文件功能类似，语法有些差异。在VDSP和CCES中使用ADI系列的DSP都需要用到LDF文件。对于普通应用，ADI提供了很多可供选择的默认LDF文件，但对于需要自己定义链接脚本内容的用户而言，只能自己修改或者编写新的LDF文件，不能用插件生成。在VDSP中可以用Expert Linker工具直观的观察LDF文件对存储空间的分配。

链接脚本的作用是用于引导链接程序如何确定代码的存储地址和运行地址。它告诉链接器，目标板的rom 和ram 的起始地址和尺寸，它允许定义不连续的多块rom 和ram，哪些代码需要连接，代码在rom 中的存储位置和运行时地址，每段代码都可以单独指定，以及运行时使用的内存地址。另外，连接脚本文件还能输出程序的定位信息，在C 源程序中作为外部符号引用，以完成程序加载功能。在djyos的移植中，自定义链接脚本的内容是必要的。

与常用的LDS文件不同，链接和下载的完整功能是由LDF文件和LDR文件共同完成的。LDR即loader文件，其本质是一个dxe可执行文件，与用户工程的dxe可执行文件一起生产ldr后缀的加载文件，用于下载到DSP中去。LDF指定了程序和数据的运行位置。LDR文件则指定了在FLASH或者ROM中的实际存储位置，LDR文件有自己的特定的格式。代码从FLASH至RAM的搬运是通过LDR来实现的，具体实现通过ADI提供的Boot Kernel代码。Boot Kernel的长度限制为最多256行代码。Boot Kernel拷贝代码是通过每段代码的Header部分来实现的，最后的一个段为Final Block，其中为IVT的定义，在Boot Kernel的最后阶段会将Final Block拷贝覆盖掉Boot Kernel段的代码，从而完成整个加载过程。如图1-6所示。

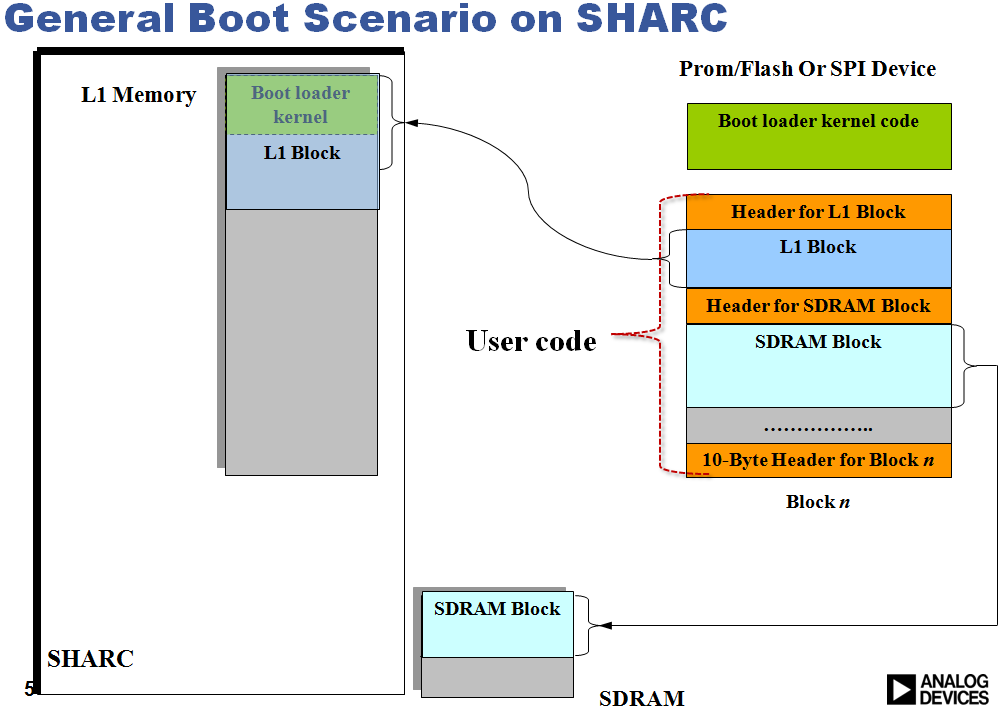


图1-6 SHARC的加载过程

在完成代码搬运后，CPU会自动触发一次RSTI中断，即硬件重启，因此RSTI中断需指向CPU的初始化部分，包括C语言环境的初始化（\_\_\_lib\_setup\_c）、跳转的目的地址（\_main或\_pre\_start）等。其中\_\_\_lib\_setup\_c的源码在set\_c.asm中可以参阅和修改。

LDR文件的源码可以在软件的安装目录下找到。用户可自行选择所使用的LDR文件，可以是默认的，也可以是由用户自行定义的。在特殊情况下可以选择不使用kernel，如在进行二次加载的时候。如图1-7所示。

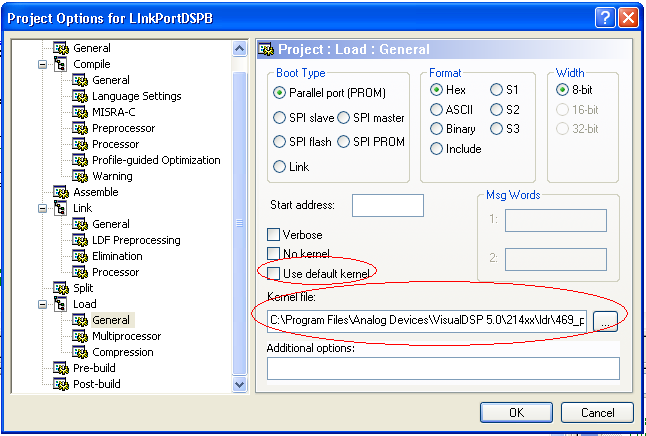


图1-7 Kernel文件的选择

在VDSP的Tools中有一个Flash Programmer工具，可以用于向Flash中烧写ldr加载文件。这个小工具所调用的同样也是dxe可执行文件，在example目录下可以找到该工程。在Flash Programmer这个小工具中可以自定义代码存储在Flash中的起止位置。如图1-8所示。

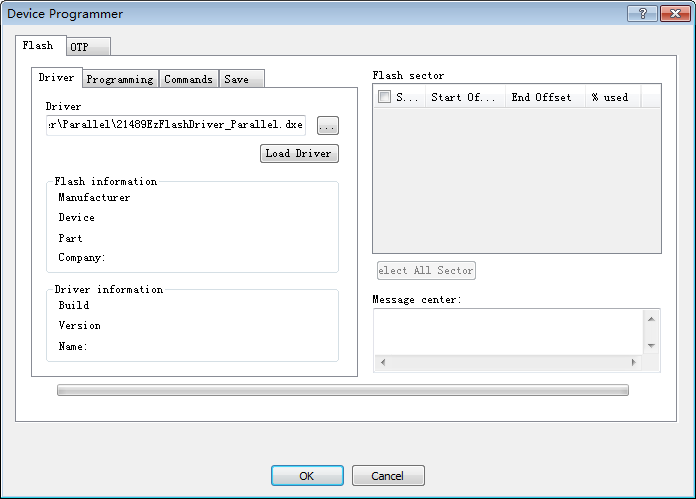


图1-8 Flash Programmer工具

ADI的LDF文件中常用的命令有如下几个。详细可参阅CCES/VDSP帮助文档的**LDF Commands**。

ALIGN()：对齐指令。

MEMORY()：物理段的命名和分配物理空间。

PROCESSOR()：包含SECTION。以及符号的定义。

SECTION()：逻辑段和物理段的对应关系。

ADI的LDF文件中常用的关键字参阅CCES/VDSP帮助文档的**LDF Keywords**。

ADI的LDF文件中常用的几个预定义的宏定义有$OBJECTS和$LIBRARIES。$OBJECTS指向DOJ文件，$LIBRARIES指向DLB文件。在VDSP中默认包含的21469\_hdr.doj包含了IVT的定义；在CCES中默认包含的app\_startup.doj包含了startup相关内容及IVT的定义。在我们的工程中，将之修改为exceptions.doj。

ADI的LDF文件的使用原则是可以适当修改和增加自己需要的内容或段，但最好不要删除原本默认定义的段。

CCES的LDF中相关物理段定义如下：

mem\_iv\_code { TYPE(PM RAM) START(0x0008C000) END(0x0008C0A7) WIDTH(48) }

VDSP的LDF中相关物理段定义如下：

seg\_rth { TYPE(PM RAM) START(0x0008c000) END(0x0008c0ff) WIDTH(48) }

注意，其中的TYPE、WIDTH都只能作为参考使用，并没有实际意义。真正决定位宽的是实际的物理地址。Sharc中的物理地址存在如下转换关系：

SW（short word，16bit）== NW（normal word，32bit）\*2 == LW（long word，64bit）\*4

VDSP的LDF中默认的几个段名如下：

seg\_rth：存放IVT和初始化部分代码的逻辑段名；

seg\_pmco：存放代码的逻辑段名；

seg\_pmda：存放代码中数据的逻辑段名；

seg\_dmda：存放数据的逻辑段名；

seg\_stak：存放栈的逻辑段名；

seg\_heap：存放堆的逻辑段名；

cces\_1-0-2\_comp\_sharc\_man\_rev.1.1.pdf的P413 的**Memory Section Usage**章节描述了各个memory section的命名和使用规范，可参阅。

关于符号的定义和使用。

在VDSP的LDF文件中，定义了与堆栈相关的两组符号，包含堆栈的起始地址、长度等信息。ADI自身的文件中会调用到这些符号，在djyos的内存管理模块也会调用到heap相关的一组符号。原本在loader中需要定义一组符号供loader和pre-loader使用，但鉴于SHARC启动的特殊性，因此从工程中移除了相关部分代码，因此只需要考虑djyos的内存管理模块中与heap相关的一组符号的定义，这组符号是：u8 \*sys\_heap\_bottom\_cnt和u8 \*sys\_heap\_top\_cnt，分别代表堆的顶断和底端地址。鉴于在LDF文件中已经有相似符号的定义，我们只需要给这两个符号赋值并在我们的C源码中extern就可以正常使用了。LDF文件相关部分修改如图1-9：

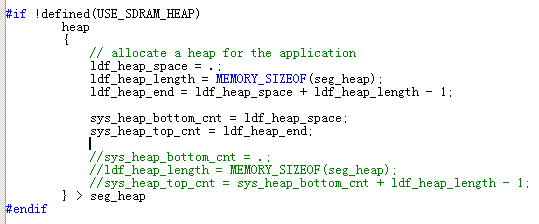


图1-9 修改LDF中的符号

因为ADI本身定义的这组符号也有特定功能，因此我们保留它的这组符号，只是将地址数据赋值给我们需要使用的符号即可。在C源代码中要调用这组符号，需要使用如下形势来声明：



在汇编代码中声明的格式如下：



通过上述方式可以实现符号的定义和声明，但是需要修改\djysrc\djyos中源文件的内容，为保证源码内容不做修改，因此只好采取迂回战术。具体方法如下：

1. 不修改LDF中堆的符号的定义，仍然使用ADI默认的一组符号，如下所示：

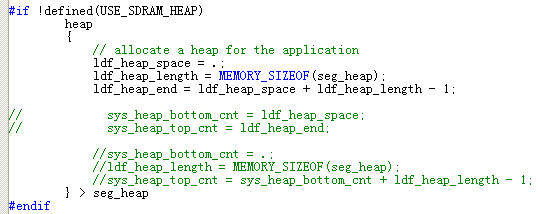


图1-10 LDF中默认的heap所用的符号

1. 因为djyos中用到堆相关符号的名称与ADI默认定义的不一样，因此，在CPU.C文件中，我们重新定义这样一组变量，并用ADI默认定义的符号为他赋初值，如下：

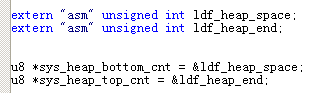


图1-11 LDF符号与djyos符号的转换

在完成以上基本的一些设置和修改后，工程能够正常编译通过，并链接成功。

## ADI汇编指令及寄存器说明

关于ADI汇编的寄存器说明。该部分内容可参阅《ADSP技术与应用》的3、4两章节内容，本处仅记录常用到的内容及容易引起误解的内容。

1. 内核寄存器：

内核寄存器可以在汇编语言中直接访问，但在C语言中不能直接访问，这是不同于IOP寄存器的。因为内核寄存器不能通过I/O总线进行寻址，因而没有物理地址。如果应用程序需要读\写内核寄存器，只能通过调用ADI提供的API函数或者自己编写简短的汇编代码来实现相应功能。如图1-12和1-13所示。

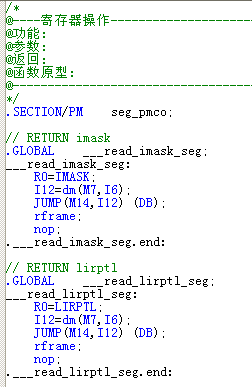
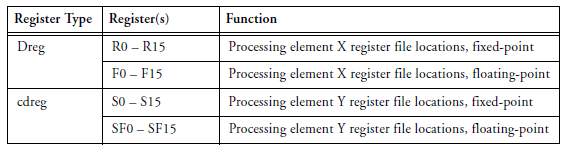
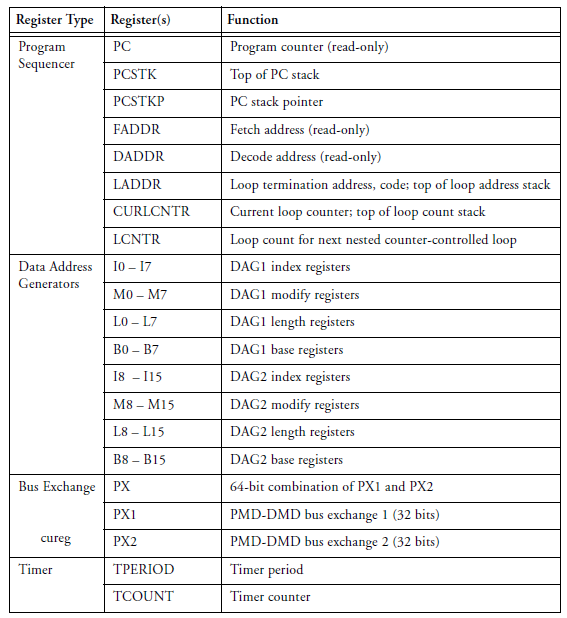


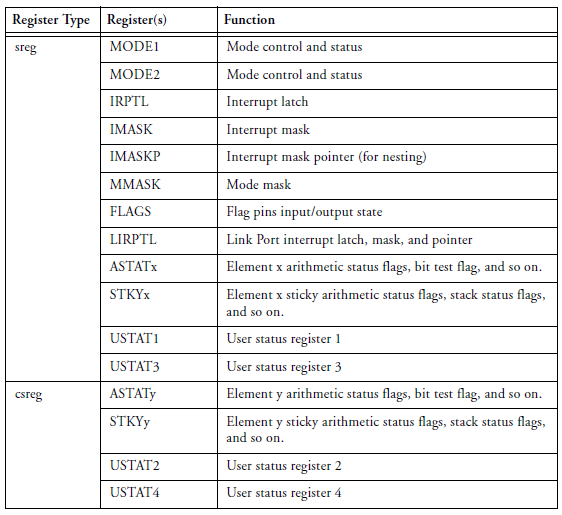
图1-12 汇编操作内核寄存器示例



图1-13 API操作内核寄存器示例







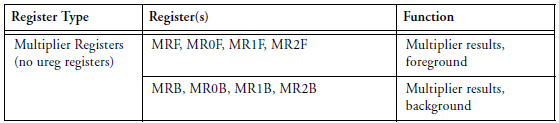


图1-14 内核寄存器

1. IOP寄存器：

IOP的含义为input/output processor。IOP寄存器是指，可以通过I/O总线直接进行访问的寄存器。IOP的作用是帮助CPU管理数据操作相关的一些模块，比如DMA、片内外设等。IOP寄存器包含了除内核寄存器以外的其余寄存器，绝大部分外设模块的寄存器都属于IOP寄存器，具体可以查阅ADSP-214xx\_hwr\_r1-0.pdf的P1191。对IOP寄存器的访问跟一般指针数据的访问方式是类似的，这些寄存器的地址都定义在Cdef21469.h头文件中。调用示例如图1-15所示。

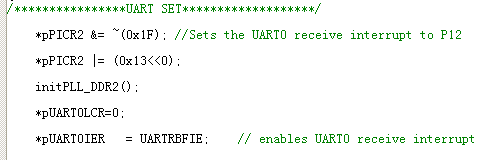


图1-15 IOP寄存器的访问

三、关于ADI汇编的寄存器操作指令。

1. 位操作

系统寄存器（SREG）的快速位操作指令：

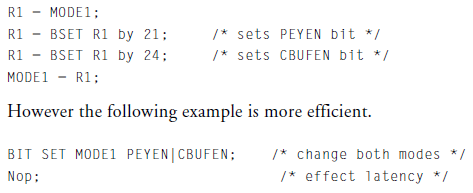


图1-16系统寄存器（SREG）的快速位操作指令

控制寄存器的位操作指令：

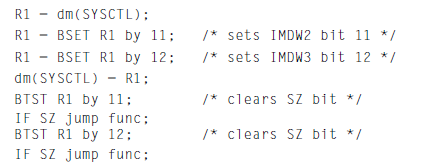


图1-17控制寄存器的位操作指令

利用用户标志寄存器（user status registers USTAT4-1）进行位操作，可以防止破坏相邻位：

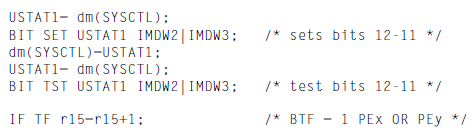


图1-18用户标志寄存器操作指令

1. 总线数据交换

利用总线数据交换寄存器PX（64bit）或PX1/PX2（32bit）实现对PM和DM总线数据的单指令周期操作，相关操作指令及说明如下：

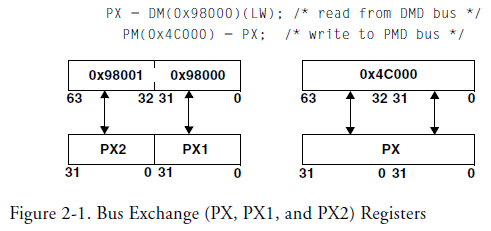


图1-19 PX数据交换指令

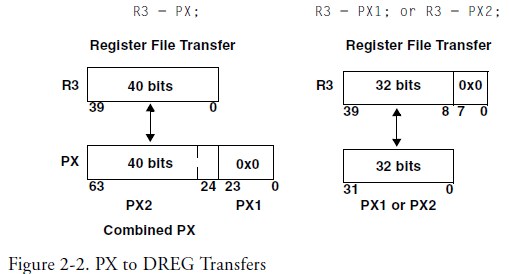


图1-20 PX及寄存器数据交换指令

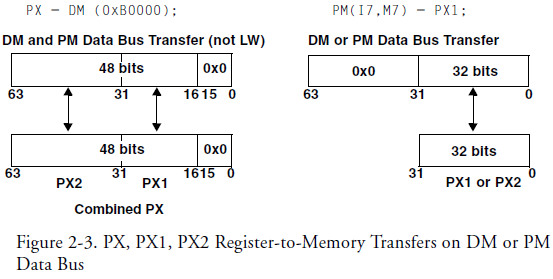


图1-21 PX寄存器值操作指令（DM或PM）

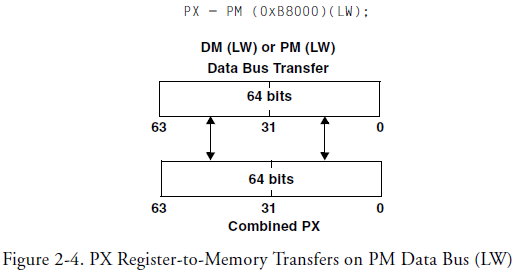


图1-22 PX寄存器值操作指令（PM）

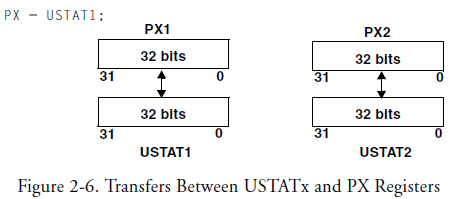


图1-23 PX及USTAT数据操作指令

1. 通过DAG读写数据

对于如下指令，DAG直接解析相应地址的数据：

PM（I0,0）、DM（I4,0）

对于包含了Modify（M）寄存器时的解析方式，可将M寄存器所包含类容理解为偏移量，DAG取出数据后，会根据指令格式判断是否更新I寄存器所对应地址的数据，如下例：

R6 = PM(I15,M12); /\* Post-modify addressing with update \*/

可以理解为：R6=\*I15,I15=I15+M12;

R6 = PM(M12,I15); /\* Pre-modify addressing without update \*/

可以理解为：R6=\*(I15+M12),I15=I15;

同时，M寄存器也可以直接由数字代替，如下例：

R1 = DM(0x40000000,I1); /\* DM address = I1 + 0x4000 0000 \*/

图1-24描述了两种情况下的差异。

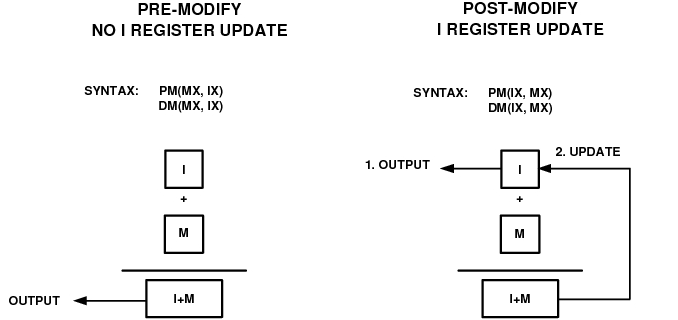


图1-24 栈操作指令说明

下表所列的寄存器值为默认设置，不可更改的。

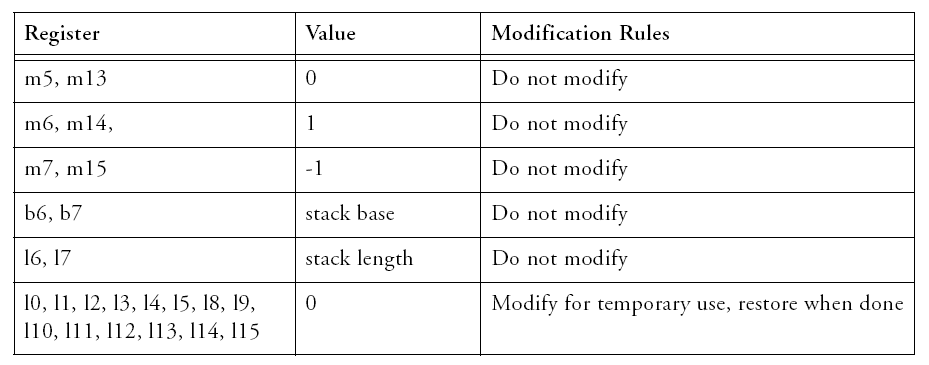
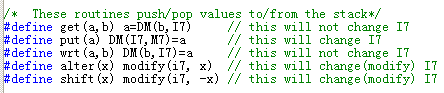


图1-25 栈相关寄存器默认值（不可更改）

以I7和M7为例。I7为默认的 stack pointer，M7 = -1。当使用DM（I7,M7）指令从栈中读取数据时，I7会自动-1，而不需要人为干预。需注意，在修改I7时若B7、L7赋值不正确，则在调用modify(I7,X)指令时可能会出现栈溢出异常。

我们定义了常用的一组栈操作函数，他们的应用环境各不相同：



其中，put与wrt都是向栈中写数据，区别在于wrt不会改变I7指针值，这样可以有助于避免内存不对其问题的产生。put与get，wrt与get，alter与shift都应该是成对出现的。

四、关于ADI汇编预定义的一些宏。

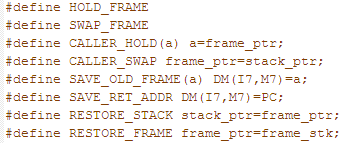
这些预定义的宏可以有助于提高代码的阅读性，但同时也有可能增加代码的阅读难度，在我们的代码中完全可以规避这些宏定义的使用。归纳常用的几个定义供查阅：













五、关于函数调用时寄存器的保存规则

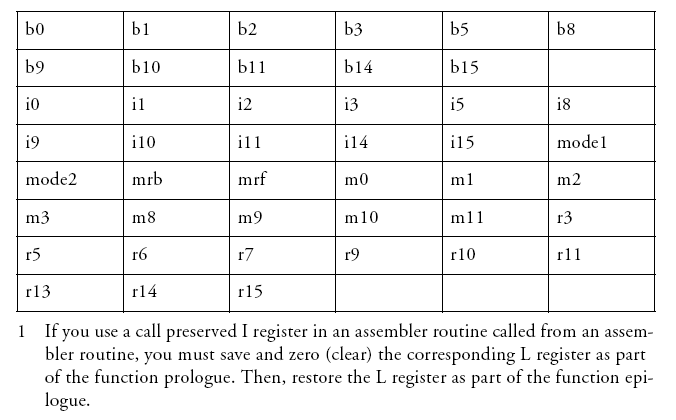


图1-26 由被调用方保存的寄存器

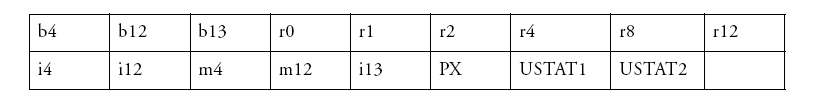
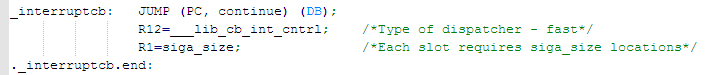


图1-27 由调用方保存的寄存器

六、ADI汇编相关的补充说明。

ADI定义了自己的汇编语法格式，与通用的汇编语法类似，但有差别。现将已经发现的容易引起误会的地方记录在此，今后有新的问题继续补充：

1. 在汇编中定义的函数，若要在C语言中调用，则需要注意函数名称的变化。如在ASM中定义函数：\_\_ASM\_TEST，在C语言中调用时函数名应为：\_ASM\_TEST。但在H文件中申明时，应用\_\_ASM\_TEST，与汇编中一致。
2. 在汇编中的如下函数段：



其中，JUMP（PC,continue） （DB）中DB代表的含义是：执行完本语句的后面两条指令后，再执行本语句。DB指令有可能会引起BUG，参阅第4章中关于WA\_09000015的说明。

# ADSP21469系统初始化

## 2.1 Arch.h中系统参数配置

Include文件夹下arch.h头文件用于定义与CPU相关内核配置，主要有：

**存储器大小端配置**：ADSP21469为小端 (small\_endian);

与CPU字长相关配置：

**处理器字长** cn\_cpu\_bits：ADSP21469处理器字长为32位

**CPU地址位数** cn\_ptr\_bits：ADSP21469 CPU地址位数为32

**CPU字长2进制表示的后缀0个数** cn\_cpu\_bits\_suffix\_zero：同默认值为5

**字节位宽** cn\_byte\_bits: ADSP21469字节位宽为32。

**CPU系统栈类型**：ADSP21469栈类型为向下生长的满栈 cn\_empty\_down\_stack

**代理内核栈深度**cn\_kernel\_stack:同默认值1000

**内存对齐**：2字节对齐，定义如下：

//定义一些与对齐相关的宏

#define mem\_align 1 //如果目标系统没有对齐要求，改为0

#define align\_size 2 //ADSP21469要求2字节对齐

## 2.2 PLL初始化

### 2.2.1 ADSP21469 PLL工作原理

如图 8所示，ADSP21469系统时钟外部输入时钟信号CLKIN，然后经过PPL倍频、分频分别得到内核时钟CCLK、系统时钟PCLK、DDR2时钟DDR2CLK及Link Port时钟LCLK。

|  |
| --- |
| C:\Documents and Settings\Administrator\桌面\QQ图片20131011143543.jpg |

图 8 ADSP21469时钟系统示意图

由CLKIN到CCLK、PCLK等其他时钟频率的转换均需要通过设置PMCTL(Power Management Control)寄存器来实现，PMCTL寄存器如图 9所示。

|  |
| --- |
| C:\Documents and Settings\Administrator\桌面\QQ图片20131011190403.jpg |

图 9 **PMCTL**寄存器示意图

下面结合图1、图2阐述时钟频率的转变原理。

1. PLL初步分频：在图1中CLKIN DIVIDER模块为Pre-Input Divider，它将CLKIN时钟频率分频1/2，由PMCTL寄存器的Bit8(*INDIV*)控制，当Bit8=1时，分频1/2，当Bit8=0时则不分频。
2. PLL倍频：在CPU刚上电时，PLL倍频是由硬件CLK\_CFG 1-0引脚控制，通过向CLK\_CFG 1-0引脚写入指定的状态值来确定倍频倍数，CLK\_CFG 1-0引脚状态值对应PMCTL的Bit17、Bit16，CLK\_CFG [1：0]引脚状态值对应的倍频倍数*CART*对应关系如表格 6所示。

表格 6 ADSP21469 PLL倍频倍数*CART*与CLK\_CFG [1：0]引脚状态值对应表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CLK\_CFG [1：0] | 00 | 01 | 10 | 11 |
| *CART* | 6 | 32 | 16 | Reserved |

当CPU启动后可以通过软件设置**PMCTL**寄存器(Address: 0x2000)的Bit0~Bit5(*PLLM*)获得0~63倍的倍频。

1. fVCO电压控制振荡器，由图1可知fVCO由PLL初步分频(*INDIV*)和PLL分频(*PLLM*)确定，fVCO与寄存器PMCTL的INDIV(Bit8)及PLLM(Bit0~Bit5)关系如表格 7所示。

表格 7 fVCO与寄存器**PMCTL**的INDIV(Bit8)及PLLM(Bit0~Bit5)关系表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | *PLLM*=0 | *PLLM*=1 | *PLLM*=2 | *PLLM*=N(3~62) | *PLLM*=63 |
| *INDIV*=0 | 128 | 2 | 4 | 2N | 126 |
| *INDIV*=1 | 64 | 1 | 2 | N | 63 |

1. PLL分频：如图1中PLL DIVIDER模块，分频倍数可由**PMCTL**寄存器*PLLD(*Bit6~Bit7)确定，如表格 8所示

表格 8 PLL分频值

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **PMCTL[7:6]** | 00 | 01 | 10 | 11 |
| Clock divider | 2 | 4 | 8 | 16 |

**Tips:** fVCO及内核时钟CCLK计算公式如下：

|  |  |
| --- | --- |
|  | (1) |

*PLLM*为寄存器**PMCTL**设置的倍频值，其中在复位/上电时*PLLM*值由CLK\_CFG [1：0]引脚选择的比值确定。

*PLLD*为寄存器**PMCTL**设置的分频值，其中在复位/上电时，默认为2。

为PLL输入频率，当PMCTL寄存器的*INDIV*=0时，；当*INDIV*=1时，。

1. 系统时钟PCLK(IOP Clock)，系统时钟由内核时钟分频1/2得到，即。

|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\QAE~0$TV~}_NB)_~2@C7DEE.jpg |

图 10外设时钟PMCTL设置示意图

1. 外设时钟(DDR2CLK、LCLK)，外设时钟由内核时钟CCLK经过分频后得到，通过设置**PMCTL**寄存器的EPCKR(Bit20~Bit18)和LCKR(Bit22~Bit21)确定，如图3所示。在设置EPCKR(Bit20~Bit18)和LCKR(Bit22~Bit21)前必须要先使能寄存器PMCTL的*DIVEN*位(Bit9)。

选择合适的外部晶振，通过配置寄存器**PMCTL**值即可得到想要的各种时钟频率，几种典型的时钟频率如图 11所示。

|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\K0GK~03A$ML)3H4NHD)Y@~3.jpg |

图 11几种典型的内核频率与CLKIN比例配置图

目前ADSP-21469开发板配置一个25MHz外部晶振。

在配置各种时钟频率要注意一个问题是fVCO值不能越界，fVCO极限值如图 12所示。

|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\WT)MEU1GRBF(]GTVVO2%X_P.jpg |
| 图 12 fVCO极限值示意图 |

其他几个注意事项：

1. 关于旁路模式(Bypass Mode)：CPU运行模式分为Bypass Mode和Normal Mode两种，在Normal Mode模式下时fVCO锁定，固定不变。当在程序中想改变fVCO值时，CPU即进入Bypass Mode，通过设置寄存器**PMCTL**的*PLLBP*=1(Bit15)(使能*PLLBP*过程详见*ADSP-214\*\* SHARC Processor Hardware Reference, Revision 1.0. p877*)，然后设置寄存器**PMCTL**的其他位值，使fVCO被设置为新的值，fVCO由旧值到新值之间需要花4096 CLKIN cycles。在这个时期内，内核频率等于CLKIN。当fVCO到达新值后，寄存器**PMCTL**的*PLLBP*位自动清零。在CPU复位/上电阶段，CPU也需要花4096 CLKIN cycles才能达到设定的时钟频率。
2. 外设时钟默认为使能：CPU上电后，DDR2、SDRAM、S/PDIF及RTC默认使能，其他外设默认未使能，当这些默认使能的外设不需要使用而使其未使能，可通过分别设置DDR2 Controller、SDRAM Controller、S/PDIF Controller及Real-Time Clock Controller，具体方法参考*“ADSP-214\*\* SHARC Processor Hardware Reference, Revision 1.0. p870”。*
3. PLL启动与内核复位关系**：**在内核开始运行或者开始引导程序之前必须要启动PLL，而PLL需要4096 CLKIN cycles(默认)锁定CLKIN。内核复位信号触发示意图如图 13所示。

|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\QZKPR@4AFKR@T85HSJ2CNFD.jpg |
| 图 13内核复位电路图 |

### 2.2.2 ADSP21469时钟初始化具体步骤

下面结合上面阐述的ADSP21469时钟工作原理阐述ADSP时钟初始化具体步骤：

1. 置位*INDIV*位，将CLKIN时钟频率分频1/2；
2. 设置PLL Bypass：刚上电到锁定设定频率期间，CPU 时钟处于旁路模式，需要对*PLLBP*  置位；
3. 刚上电时PLL倍频倍数由硬件CLK\_CFG 1-0引脚控制，通过检测CLK\_CFG 1-0引 脚状态来确定倍频倍数，CPU需花4096 CLKIN cycles才能达到设定的时钟频率。在此处程序应等待4096 CLKIN cycles；
4. 将CPU时钟由旁路状态转换到正常状态：通过对*PLLBP* 位清零，此处应等待16个CLKIN cycles完成状态转换；
5. 清除原有的PLL倍频，设置特定的倍频参数：目前我们拟打算将内核CPU设置为450MHz，外部晶振频率(CLKIN)为25MHz，故可将*PLLM*=18，*PLLD*=2，*INDIV*清零，此时CCLK=(2\*25\*18)/(2\*1)=450MHz。注意此时由于要改变fVCO，需要将CPU时钟重新设置为旁路模式(置位*PLLBP*)；
6. 等待4096CLKIN cycles等待PLL锁定新频率，将*PLLBP* 位清零，CPU时钟回到正常状态；

输出RESETOUT，完成PLL时钟初始化。

|  |
| --- |
| InitPLL:  //Step 1 - change the PLLD to 4  USTAT1 = dm(PMCTL);  bit clr USTAT1 PLLD16 ; //#define PLLD16 (BIT\_6|BIT\_7)  bit set USTAT1 PLLD4 | DIVEN ;  dm(PMCTL) = USTAT1;    //Step 2 - wait for dividers to stabilize  lcntr=16, do first\_div\_delay until lce;  first\_div\_delay: nop;    //Step 3 - set INDIV bit now to bring down the VCO speed and enter the bypass mode  bit clr USTAT1 DIVEN;  bit set USTAT1 INDIV | PLLBP;  dm(PMCTL) = USTAT1;    //Step 4 - wait for the PLL to lock  lcntr=4096, do first\_bypass\_delay until lce;  first\_bypass\_delay:nop;    //Step 5 - come out of the bypass mode  USTAT1 = dm(PMCTL);  bit clr USTAT1 PLLBP;  dm(PMCTL) = USTAT1;    //Step 6 - wait for dividers to stabilize  lcntr=16, do second\_div\_delay until lce;  second\_div\_delay:nop;    //Step 7 - set the required PLLM and INDIV values here and enter the bypass mode  //PLLM=18, INDIV=0, fVCO=2\*PLLM\*CLKIN = 2\*18\*25 = 900 MHz  ustat1 = dm(PMCTL);  bit clr USTAT1 INDIV | PLLM63;  bit set USTAT1 PLLM18 | PLLBP; //--Core CLK is 450MHz  dm(PMCTL) = USTAT1;    //Step 8 - wait for the PLL to lock  lcntr=4096, do second\_bypass\_delay until lce;  second\_bypass\_delay:nop;    //Step 9 - come out of the bypass mode  USTAT1 = dm(PMCTL);  bit clr USTAT1 PLLBP;  dm(PMCTL) = USTAT1;    //Step 10 - wait for dividers to stabilize  lcntr=16, do third\_div\_delay until lce;  third\_div\_delay:nop;    //Step 11 - set the required values of PLLD(=2), LPCKR and DDR2CKR (=2) here  // fCCLK = fVCO/PLLD = 900/2 = 450 MHz, fDDR2CLK = fCCLK/DDR2CKR = 450/2 = 225 MHz, fLPCLK = 450/3 = 150 MHz  USTAT1 = dm(PMCTL);  bit clr USTAT1 PLLD16 | 0x1C0000 | LPCKR4;  bit set USTAT1 DDR2CKR2 | LPCKR3 | PLLD2 | DIVEN;  dm(PMCTL) = USTAT1;    //Step 12 - wait for the dividers to stabilize  lcntr=16, do fourth\_div\_delay until lce;  fourth\_div\_delay: nop; |

## 2.3 DDR2初始化

DDR2初始化步骤如下：

1. On Chip DLL Reset：当DDR2时钟频率改变时，需要此步骤，在Boot Kernel中将DDR2初始化频率为225MHz，当在运行过程中需要重新设定DDR2频率时，需要重新此步骤。该步骤分为两步：①DLL启动：DLL启动通过两个寄存器**DLL0CTL1**(Address: 0x1851)、**DLL1CTL1**(Address: 0x1856)控制，**DLL0CTL1**寄存器如图 14所示：

|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\7L1A]N2W)Z9Q}`V90F6)X6L.jpg |
| 图 14 **DLL0CTL1**寄存器示意图 |

Bit9~Bit11分别代表Reset DLL Control Logic、Reset Data Capture Logic、Reset DQS Phase Calibration Logic只需将这三位置位即可，寄存器**DLL1CTL1**执行同样操作。DLL启动至少需要花9个内核时钟周期(此处应让程序在原地等待9个内核时钟周期)。

②锁定新频率：此处应根据DDR2设定的时钟频率确定一固定的时间(3000内核时钟周期)等待锁定新频率。DDR2的时钟频率(DDR2CLK)最大值为内核时钟频率(CCLK)的一半，即当时钟频率为450MHz时，DDR2CLK为225MHz。

当锁定新频率后DLL Status Registers(**DLL0STAT0**) (Address: 0x1853)的Bit3自动置位。

1. 设置寄存器**EPCTL**(External Port Control)使能DDR2 External Banks。ADSP21469的DDR2外部映射地址如表格 9所示。

表格 9 DDR2地址映射表

|  |  |  |
| --- | --- | --- |
| Bank | Size in words | Address Range |
| Bank0 | 62M | 0x0020 0000~0x03FF FFFF |
| Bank1 | 64M | 0x0400 0000~0x07FF FFFF |
| Bank2 | 64M | 0x0800 0000~0x0BFF FFFF |
| Bank3 | 64M | 0x0C00 0000~0x0FFF FFFF |

寄存器**EPCTL**(Address: 0x1801)如图 15所示，Bit0~Bit3分别使能Bank0~Bank3，初始化阶段使能Bank0。Bit4~Bit5用于设置External Port Bus Priority，默认设定为11(Rotating Priority)，初始化阶段设置为10(内核优先级最高，具体说明详见*ADSP-214\*\* SHARC Processor Hardware Reference, Revision 1.0. p948*)。设置External Port DMA Channel优先级(Bit6~Bit7)，初始化为Channel 0为高优先级，FRZDMA、FRZCR初始化为No Freezing。

|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\SLTPTZCEM77P5[X75[7T6{F.jpg |

图 15寄存器**EPCTL**示意图

1. 初始化DDR2自动刷新频率：DDR2的刷新频率是通过设置寄存器Refresh Rate Control Register(**DDR2RRC**) (Address: 0x181D)实现初始化。寄存器**DDR2RRC**示意图如图 16所示。DDR2RRC寄存器中包含两个重要参数RDIV(Bit13~Bit0)、tRFC(Bit28~Bit21)。

|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\FN5UX7NOHNK]}ON{LJJ9HYU.jpg |

图 16寄存器**DDR2RRC**示意图

*RDIV*指DDR2两次刷新中间间隔的时间。*RDIV*的计算如式1~2所示。

|  |  |
| --- | --- |
|  | (1-2) |

该公式的具体说明详见*ADSP-214\*\* SHARC Processor Hardware Reference, Revision 1.0. p234*。RDIV的范围为0x001~0x3FFF，RDIV的具体值与DDR2\_CLK有关，当内核频率为450MHz时，RDIV设定为0x6CC；当当内核频率为400MHz时，RDIV设定为0x60A。

tRFC为列更新时间，默认值为0x14。

1. 设置DDR2 Control Registers(**DDR2CTL0~DDR2CTL5**) (Address: 0x1812~0x1817)，**DDR2CTL0~DDR2CTL5**具体功能详见*ADSP-214\*\* SHARC Processor Hardware Reference, Revision 1.0. p161*。DDR2CTL1设置DDR2的存储/访问时间，其与DDR2\_CLK密切相关。**DDR2CTL1**寄存器如图 17所示，具体各个参数定义及设置详见*ADSP-214\*\* SHARC Processor Hardware Reference, Revision 1.0. p971，*各参数默认值如表格 10所示。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 表格 10 **DDR2CTL1**寄存器中各参数默认值 | | | | | | | |
| 参数 | tRAS | tRP | tWTR | tRCD | tRTP | tRRD | tFAW |
| 值(clock cycle) | 0x06 | 0x03 | 0x02 | 0x02 | 0x02 | 0x01 | 0x0A |

|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\}6]@N`EX%$N@5K957J3F7S7.jpg |
| 图 17 **DDR2CTL1**寄存器示意图 |

其他的DDR2 Control Registers默认值设置方法详见*ADSP-214\*\* SHARC Processor Hardware Reference, Revision 1.0. p971~p978*。

⑸ 等待并确认DLL完成calibration。当DLL calibration to complete ，**DDR2STAT0**的Bit3自动清零。**DDR2STAT0**的Bit3功能定义具体如下：

|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\T6JDBY9[%QXLLKMSGX`KNCY.jpg |
| 图 18 **DDR2STAT0**的Bit3功能定义示意图 |

## 2.4 系统tick

### 2.4.1．ADSP21469内核时钟(Core Time)

ADSP21469内核时钟与其他处理器类似，内核时钟从一个初始值随着每个时钟减1，当减到零时，触发一个中断。ADSP21469通过三个寄存器控制内核时钟，这三个寄存器分别是：

MODE2： MODE2寄存器的Bit 5(*TIMEN*)使能/屏蔽内核寄存器，当*TIMEN*=1，使能；*TIMEN*=0，屏蔽；

TCOUNT: 内核时钟当前计数值。

TPERIOD: 内核时钟自动加载值，当内核时钟计数值减到零，产生中断后，内核会自动将TPERIOD寄存器中的值加载到计数寄存器TCOUNT中，重新计数。

ADSP21469内核时钟框架图如图 19所示。

|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\G(J24U0_LVKQNWEJX1`F_AA.jpg |
| 图 19ADSP21469内核时钟框架图 |

ADSP21469内核时钟能产生两个中断TMZHI和TMZLI，中断优先级分别为4,32，当计数值减到零，TMZHI和TMZLI均会被锁住(be latched)，但是只有没有被屏蔽(unmasked)的中断才会被响应，当两个中断均未被屏蔽，则先响应TMZHI，再响应TMZLI。

ADSP21469针对内核时钟提供了三个API函数，分别是：

Timer\_on()：使能内核时钟

Timer\_off(): 关闭内核时钟

Timer\_set(uint32 a, uint32 b):设置内核时钟参数，形参a为初始计数值tcount，形参b为自动加载值tperiod。

### 2.4.2. 系统定时函数移植实现

系统定时部分需要移植的函数在cpu.c文件中，包含void \_\_djy\_set\_delay(void)(设定指令延时常数)，void set\_delay\_handler(int sig\_int)(定时中断服务函数)，void \_\_djy\_init\_tick(void)初始化tick，读取当前精密时间s64 djy\_get\_time(void)，读取当前ticks s64 djy\_get\_time\_tick(void)。下面分别阐述这五个函数实现过程。

#### 2.4.2.1.设置指令延时常数

void \_\_djy\_set\_delay(void)(设定指令延时常数)：该函数利用定时器产生精确时间来测量for循环的执行时间，该定时器的时间基准为CPU内核时钟(CCLK，初步设计为450MHz)，其基本思想是测定一固定次数(u32loops)循环所用的时间(counter，时间脉冲数)，然后用循环所用时间除以循环次数，即得到每循环纳秒数u32g\_cycle\_speed。

整个函数流程为：首先初始化参数并设置内核时钟中断线，这里产生高优先级中断TMZLI，并将中断ISR与中断线相关联；然后进入while循环，在while循环中首先配置内核时钟参数及使能内核时钟，然后进入for循环，其他与移植ARM版本相同，在此不赘述。

|  |
| --- |
| void \_\_djy\_set\_delay(void)  {  u32 counter,temp, narrow\_cfg\_mclk,clksum;  u32 u32\_fors=10000;  volatile uint32\_t u32loops;  clksum=(u32)(cfg\_mclk/1000); //1000uS  atom\_high\_t high\_atom;  high\_atom = int\_high\_atom\_start();//don’t be interrupt by something  int\_isr\_connect(cn\_int\_line\_TMZLI,set\_delay\_handler); //将中断ISR与中断线相关联  int\_restore\_asyn\_line(cn\_int\_line\_TMZLI);  while(1)  {  timer\_set(clksum,clksum); //初始计数值和自动加载值均设为clksum  timer\_on(); //开启内核时钟  uf\_delay\_counter=0;  for(u32loops=u32\_fors;u32loops>0;u32loops--); //进入for循环  counter=timer\_off(); //关闭内核时钟  if(uf\_delay\_counter!=0) //若在for循环过程中产生了定时中断，则重新设置clksum，  //直至for循环运行过程不发生定时中断  {  temp = (u32)(uf\_delay\_counter+1) \* clksum;  if(temp < 65536)  {  clksum = (u32)temp;  }  else  {  clksum = 65535;  u32\_fors = u32\_fors / (temp / 65535 +1);  }  }  else  break;  }  counter = clksum - counter; //取实际脉冲数。  narrow\_cfg\_mclk=cfg\_mclk/1000000;  u32g\_cycle\_speed = (uint64\_t)((counter \* 1000) /(u32\_fors\*narrow\_cfg\_mclk));//防溢出，用64位  int\_high\_atom\_end(high\_atom);  return;  } |

void set\_delay\_handler(int sig\_int)(定时中断服务函数)：在该函数中设置一全局计数器uf\_delay\_counter，每发生一次中断，uf\_delay\_counter加1，uf\_delay\_counter用于void \_\_djy\_set\_delay(void)函数中判断在执行for循环过程中是否发生定时中断。

#### 2.4.2.2.初始化tick

void \_\_djy\_init\_tick(void)( 初始化tick)：完成将内核时钟中断配置为异步信号，并将中断服务ISR与中断线关联，配置内核时钟参数及使能内核时钟及时钟中断。

|  |
| --- |
| void \_\_djy\_init\_tick(void)  {  int\_cut\_line(cn\_int\_line\_TMZLI);  int\_isr\_connect(cn\_int\_line\_TMZLI,\_\_djy\_isr\_tick); //将中断ISR与中断线相关联  int\_restore\_asyn\_line(cn\_int\_line\_TMZLI);  timer\_set(cfg\_mclk/1000,cfg\_mclk/1000);  timer\_on();  int\_restore\_asyn\_line(cn\_int\_line\_TMZLI);  return;  } |

#### 2.4.2.3.读取当前时间

s64 djy\_get\_time(void)函数获取启动以来经历的us数，准确讲应该是系统在启动时初始化tick使能内核时钟开始计时。获取当前时间方法与处理器内核时钟操作方法相关，但是其基本原理相同。系统定义了一个全局变量s64g\_os\_ticks，在每次tick中断服务函数中，都将累加一次s64g\_os\_ticks，产生一次tick中断所经过的时间us可在config-prj.h通过宏定义设置。

#define cfg\_tick\_us 1000 //操作系统内核时钟脉冲长度，以us为单位。

由s64g\_os\_ticks\* cfg\_tick\_us所得值再加上不足一个tick部分的us数即为系统从启动至今的时间us数。

不足一个tick部分计算方法如下：读取内核时钟加载值及当前Tcount值，两者之差即为不足一个tick部分所经历的内核脉冲数，将内核脉冲数乘以内核脉冲宽度即可得到此部分时间us数。由于读取内核时钟加载值及当前Tcount值需要读取TPERIOD及TCOUNT寄存器的值，用C语言不方便操作，这部分用汇编语言使用，编写\_asm\_get\_tick汇编函数在s64 djy\_get\_time(void)调用。\_asm\_get\_tick函数代码如下：

|  |
| --- |
| .section/pm seg\_pmco;  .global \_asm\_get\_tick;  .extern \_core\_time;  \_asm\_get\_tick:  //首先将本汇编函数使用到的寄存器入栈保护  shift(2);  wrt(R1,reg\_vm\_offset+2);  wrt(R2,reg\_vm\_offset+1);  r1=TPERIOD;  r2=TCOUNT;  r0=r1-r2;  dm(\_core\_time)=r0; //将返回值写入R0寄存器中  get(R2,reg\_vm\_offset+1); //恢复寄存器值  get(R1,reg\_vm\_offset+2);  alter(2);  FETCH\_RETURN  RETURN (DB);  rframe;  nop;  \_asm\_get\_tick.end: |

s64 djy\_get\_time(void)代码如下：

|  |
| --- |
| s64 djy\_get\_time(void)  {  s64 time;  u32 narrow\_cfg\_mclk;  atom\_low\_t atom\_low;  atom\_low = int\_low\_atom\_start();  //此处调用cpu.s中汇编函数\_asm\_get\_tick获得当前TCOUNT值与加载值之差  core\_time=asm\_get\_tick();  narrow\_cfg\_mclk=cfg\_mclk/1000000;  time = s64g\_os\_ticks\*cfg\_tick\_us + (u32)core\_time/narrow\_cfg\_mclk;  int\_low\_atom\_end(atom\_low);  return time;  } |

读取当前ticks函数s64 djy\_get\_time\_tick(void)，由于ADSP21469为32位处理器，此部分代码无需修改。

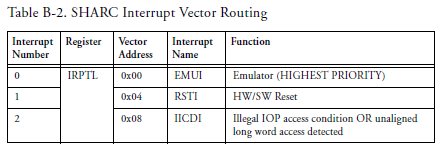
# 中断驱动

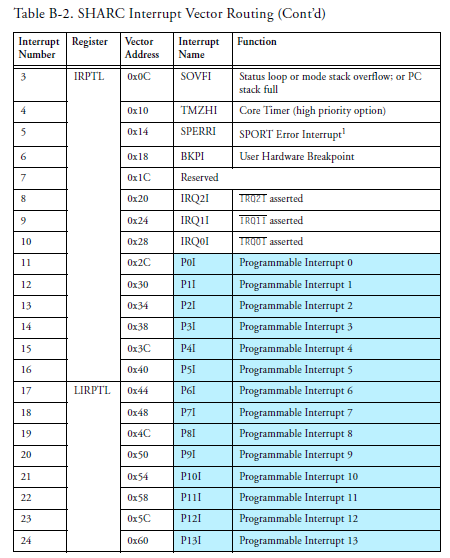
## 中断向量表

名词解释：ISR（Interrupt Service Routine中断服务函数）、IRQ（Interrupt Request中断请求）、RTI（Return from Interrupt中断返回）。

在Sharc21469中，中断向量表中包含的异常向量有硬件异常和算法异常。常见的几类算法异常（Arithmetic Interrupts）在ADSP-214xx\_PRM\_REV2.4.pdf的P116有详细描述。

中断向量表（Interrupt Vector Tables）描述如下：





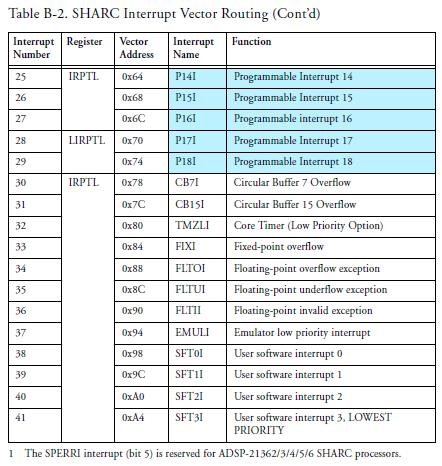


图2-1中断向量表（Interrupt Vector Tables）

在CCES中可供参考的函数和文件有：

interrupt.h：Interrupt macro definitions and dispatched handler support for SHARC（中断宏定义和Dispatcher handler support）

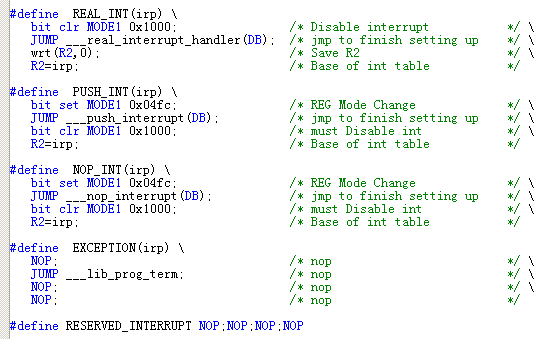
adi\_int.h：Interrupt Handler API header file（ADI的中断API调用函数头）

app\_IVT.s：Interrupt vector code（包含IVT，ISR的定义，可供参考）

若Sharc21469从片外boot（EPROM,SPI等），中断向量表（IVT）必须存放在片内内存，默认为片内RAM空间的开头。CCES中为mem\_iv\_code，VDSP中为seg\_rth。

在CCES工程的system--startup文件夹下，可以找到CCES默认的中断向量表的定义文件app\_IVT.s。参照app\_IVT.s中的语法格式完成IVT的初始化和ISR的定义。本工程的exceptions.S中即为IVT及初始化部分代码的内容，是参照ADI提供的214xx\_hdr.asm文件编写修改而成。

SHARC21469的硬件设计决定了，每个中断在中断向量表中占有4条指令空间，换言之，在中断触发进入IVT之后，每个中断都可以执行4条指令。因此我们利用这4条指令完成由IVT向ISR的跳转，主要工作是禁止全局中断；指定ISR的位置并跳转；判断是否需要切换到第二套寄存器；通过R2传递中断向量号。正是由于有这些不同条件的存在，因此对于不同的中断信号，其响应并不是完全一样的。参见如下程序段：



## SHARC21469的中断模块分析

### Sharc21469中断模块介绍

关于中断部分的说明，详见ADSP-214xx\_hwr\_r1-0.pdf的P89的INTERRUPT CONTROL章节。下面先简要介绍一下SHARC的中断模块。

系统中断控制（即响应中断请求的时间）的时钟为**fPCLK / 4**。

与中断相关的寄存器详见ADSP-214xx\_PRM\_REV2.4.pdf的P624的Interrupt Registers章节。中断相关寄存器罗列如下：

**PICR3-0**（可编程中断控制寄存器）：用于选择Programmable Interrupt（P0I^P18I）的中断源(外设中断源，如UART、SPI等)。其中最高优先级的中断为P0I，最低为P18I。详见ADSP-214xx\_hwr\_r1-0.pdf的P940。同时，外设中断源也被分为两大类——DAI或DPI，他们有各自的中断控制寄存器，能够设置外部中断源的极性、优先级和目标。而且，外部中断源也能够通过DAI或DPI连接至内核中断控制器。DAI和DPI模块自身的中断源也由PICR控制。

**IRPTL**（Interrupt Latch Register）；查中断源。当有中断信号时，IRPTL中相应位置位。注意，无论IMASK是否屏蔽了相应的位，IRPTL都会置位。当IMASK解除屏蔽之后，还是会响应相应中断。

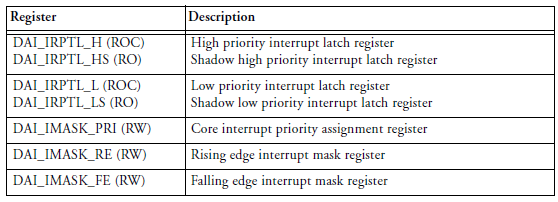
**IMASK**（Interrupt Mask Register）；中断屏蔽。

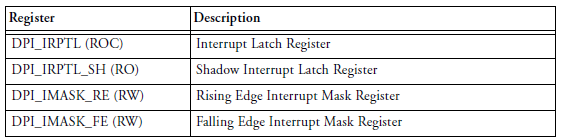
**IMASKP**（Interrupt Mask Pointer Register）；中断嵌套相关，当中断嵌套使能时，可用于在中断响应时屏蔽其余低优先级的中断。

**LIRPTL**（Interrupt Register）；补充的中断控制寄存器，因为上述的IRPTL、IMASK、IMASKP三个寄存器无法涵盖所有的中断号，因此LIRPTL用于控制上述3个寄存器不能覆盖的中断号，该寄存器能够实现上述3个寄存器的功能。

中断屏蔽的总开关在MODE1寄存器中的IRPTEN位，该位能够屏蔽IVT中所有可以被屏蔽的中断（除EMUI和RSTI外的其余所有中断，包括异常）。NESTM为中断嵌套使能位。

此外，DAI和DPI模块同样可以管理片内外设的中断信号，它们可以将各自对应外设中断源的中断信号有选择性的汇总成为DAI\_H、DAI\_L和DPI三个中断信号，并连接至可编程中断信号上。DAI相关寄存器定义详见ADSP-214xx\_hwr\_r1-0.pdf的P944。其中ROC代表Read Only to Clear。

**** DPI相关寄存器定义详见ADSP-214xx\_hwr\_r1-0.pdf的P945。

****

DAI和DPI的特征如图2-3所示：

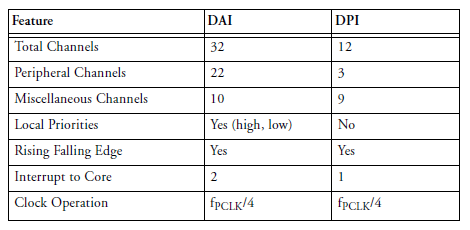


图2-3 DAI\DPI特性

在Sharc21469中，内核中断控制器通过两个系统中断控制器（DAI SIC、DPI SIC）来实现对中断的控制。其中DAI SIC包含两个不同优先级的内核中断源。其控制流程如图2-4所示：

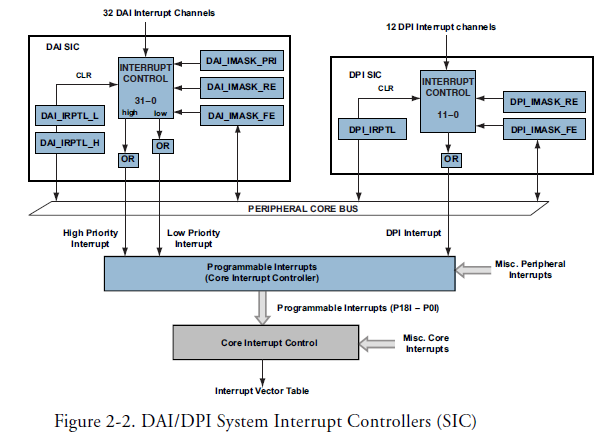


图2-4 DAI\DPI系统内核中断控制流程

DAI high priority和DAI low priority提供32路、DPI interrupt提供12路，总共44路互相独立的、可配置的中断信号源。DAI和DPI的中断源定义详见ADSP-214xx\_hwr\_r1-0.pdf的P96。

总中断的屏蔽是通过MODE1的IRPEN位来实现的。该位可以屏蔽所有能够被IMASK屏蔽的寄存器。即除EMUI和RSTI外的所有中断信号。

中断的执行流程：中断源产生中断请求IRQ——跳转至IVT查询相应的ISR并保存现场——Dispatcher调用用户的中断服务函数——返回至ISR恢复现场——中断返回RTI。

ADI Dispatcher在中断控制中的作用：

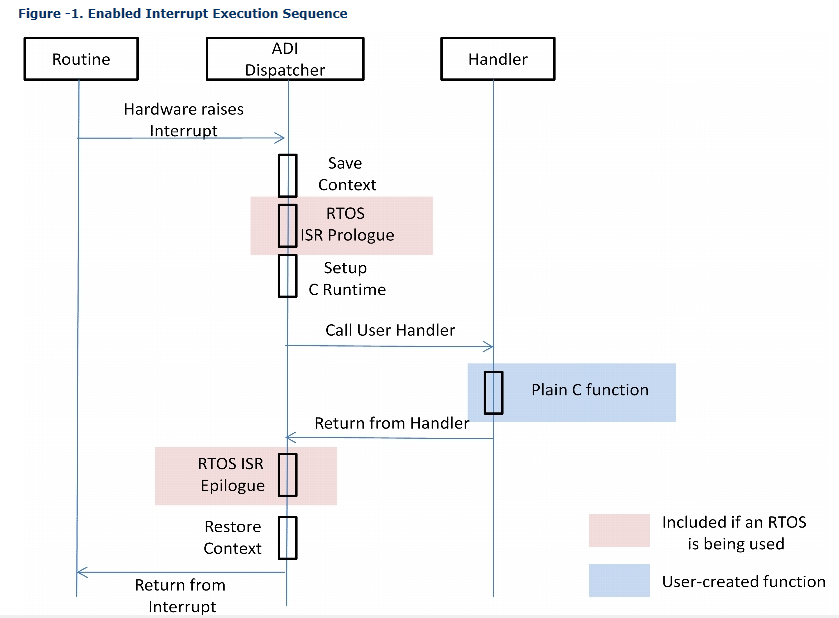


图2-5 ADI Dispatcher在中断控制中的作用

**Interrupt Dispatchers**：VDSP 5.0++中Sharc有5种不同的Interrupt Dispatchers，每种都各自包含两个功能设置函数，一个使用用户自定义代码，一个则不是。这5种模式是：

**1.circular buffer interrupt dispatcher**： interruptcb() or signalcb() functions.

**2.normal interrupt dispatcher**： interrupt() or signal() functions.

**3.fast interrupt dispatcher**： interruptf() or signalf() functions.

**4.super-fast interrupt dispatcher**：interrupts() or signals() functions.

**5.pragma interrupt dispatcher**：use with user-written assembly/C functions compiled using

“#pragma interrupt”. Use the interruptss() or signalss() function to utilize this dispatcher.

**Interrupt Handler Pragmas**：用于绕过Dispatcher，直接使用C语言编写ISR函数，并直接将之注册至IVT中。CCES中包含#pragma flush\_restore\_loop\_stack、#pragma implicit\_push\_sts\_handler、#pragma interrupt\_complete、#pragma interrupt\_complete\_nesting、#pragma interrupt\_dispatched\_handler、#pragma interrupt\_reentrant、#pragma save\_restore\_40\_bits和#pragma save\_restore\_simd\_40\_bits指令。使用这些指令都需要在IVT中修改相应中断源的中断向量，并需要特别注意环境、数据的保存。详见cces\_1-0-2\_comp\_sharc\_man\_rev.1.1.pdf的P264。

在djyos的中断系统中，我们的int\_hard.c中的int\_setto\_real（）、int\_isr\_connect（）等函数实现的功能与ADI Dispatcher实际是类似的。因此在我们的工程中应规避Dispatcher的使用。

### DAI/DPI模块详解

DAI即数字应用接口，DPI即数字外设接口，他们由Sharc21469的片内数字外设以及相应的信号路由单元组成。信号路由单元（SRU）用于控制片内外设与IO引脚之间的连接，通过寄存器的配置来实现IO引脚的自由配置。

DAI/DPI模块的的时钟为**fPCLK / 4**。

DAI模块所包含的片内外设见图2-6所示：

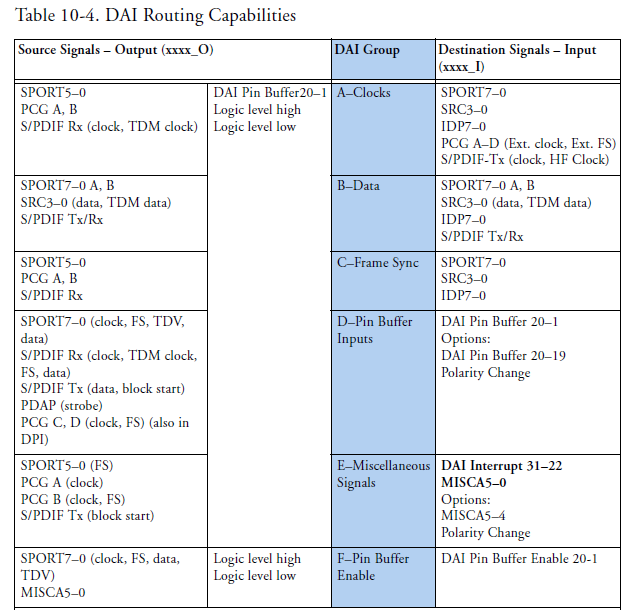
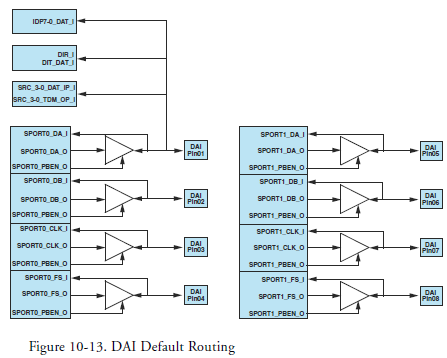


图2-6 DAI包含的片内外设

DAI模块与片内外设的默认连接如下图2-7所示：



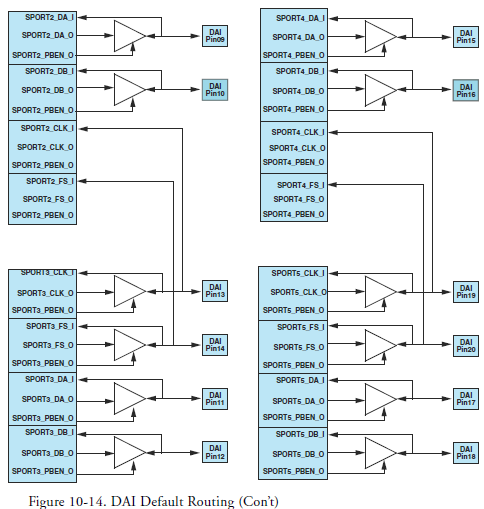


图2-7 DAI模块与片内外设的默认连接

DAI模块相关的寄存器有6个，如下：

**Clock Routing Registers (SRU\_CLKx)**：Group A，时钟信号路由。

**Serial Data Routing Registers (SRU\_DATx)**：Group B，数据路由。

**Frame Sync Routing Control Registers (SRU\_FSx)**：Group C，串行端口的帧同步或者字时钟信号路由，SRC,S/PDIF和IDP。

**Pin Signal Assignment Registers (SRU\_PINx)**：Group D，物理引脚连接路由。

**Miscellaneous Signal Routing Registers (SRU\_MISCx)**：Group E，允许程序控制、选择DAI中断闭锁，PBEN的输入路由，或者输入信号逆转。

**DAI Pin Buffer Enable Registers (SRU\_PBENx)**：Group F，DAI 20个引脚的驱动缓冲区使能。

DPI模块所包含的片内外设见图2-8所示：

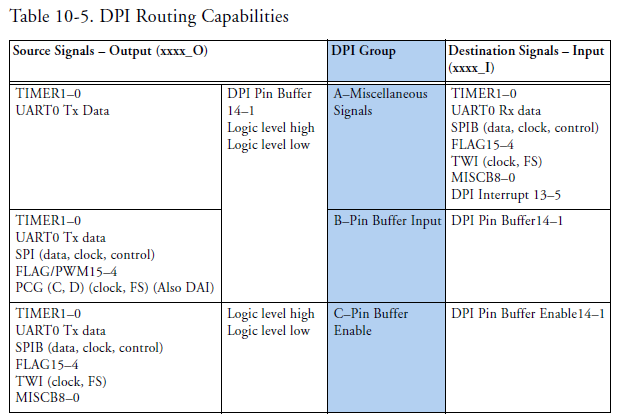


图2-8 DAI包含的片内外设

DPI模块与片内外设的默认连接如图2-9所示：

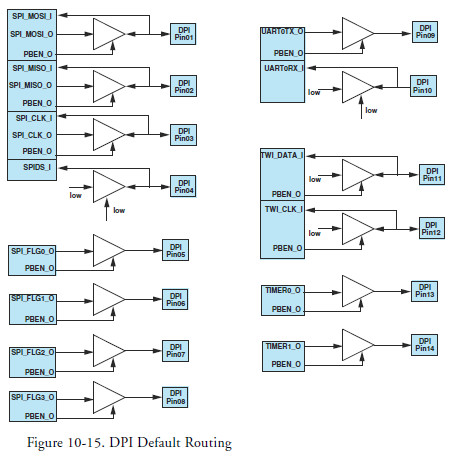


图2-9 DPI模块与片内外设的默认连接

DPI模块相关的寄存器有3个，如下：

**Miscellaneous Signal Routing Registers (SRU2\_INPUTx)**：Group A，14个外部引脚信号至其它外设输入引脚的路由。

**Pin Assignment Signal Routing (SRU2\_PINx)**：Group B，引脚输出信号至DPI引脚的路由。

**Pin Enable Signal Routing (SRU2\_PBENx)**：Group C，设置、使能引脚缓冲区，用以指定某引脚是否作为输入或输出引脚使用。

简单来说，DAI和DPI的SRU单元决定了片内外设与IO口之间的连接、对应关系，控制的时钟信号、数据、物理引脚连接等信息。使得CPU的通用性更强，外设和IO的使用更加灵活。

### 中断的实现过程分析

关于中断控制寄存器及DAI、DPI模块中断部分的基本定义，前面已做说明。首先分析一下djyos中中断控制模块的流程。如图2-10所示：

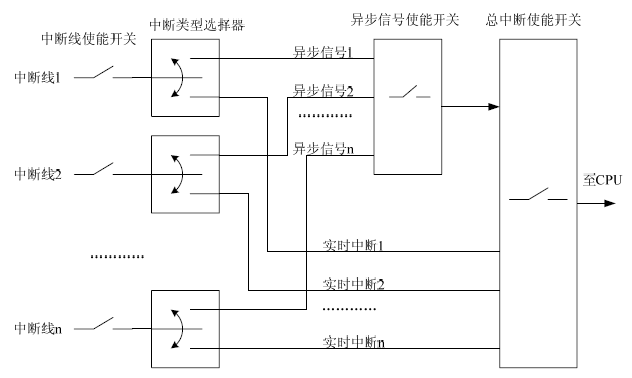


图2-10 djyos中中断控制模块的流程

下面分析一下Sharc21469中Interrupt具体的实现过程。

首先，所有的中断信号被分为了两大类：可编程中断（Programmable Interrupt）和系统中断（System Interrupt）。可编程中断又可以分为软件中断和外设中断两类。

PICR寄存器用于选择可编程中断的中断源，可编程中断本身的优先级已经确定，从P0I至P18I依次递减。

外设中断通过DAI和DPI再划归为两类分别管理，DAI和DPI的IMASK寄存器和IRPTL能够实现对外设中断信号的优先级、锁存控制、触发方式等设置。同时，外设中断信号也可以通过DAI或DPI连接至内核，成为系统中断。

DAI SIC和DPI SIC可以将各自的中断源连接至内核中断控制模块，从而成为系统中断。

针对单个中断的实现而言，首先需要使能相关的中断信号，同时要明确该信号是可编程中断还是系统中断，这涉及到优先级的设定等。

当产生一个中断信号后，ADI定义的中断的执行流程是：中断源产生中断请求IRQ——跳转至IVT查询相应的ISR并保存现场——Dispatcher调用用户的中断服务函数——返回至ISR恢复现场——中断返回RTI。如图2-11所示：

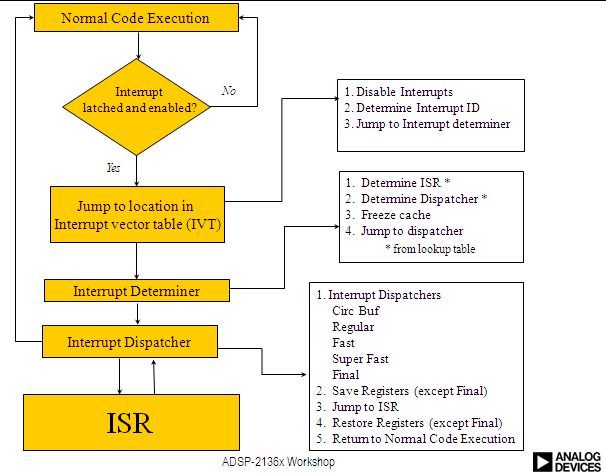


图2-11 ADI中的中断的执行流程

而在djyos中所用到的执行流程应该是：中断源产生中断请求IRQ——跳转至IVT查询相应的ISR并传递中断号（需注意硬件是否push sts）——保存现场——调用djyos的中断入口函数int\_engine\_all——判断中断类别（real或sync）——调用相应的用户中断服务函数——返回至ISR并恢复现场——中断返回RTI。

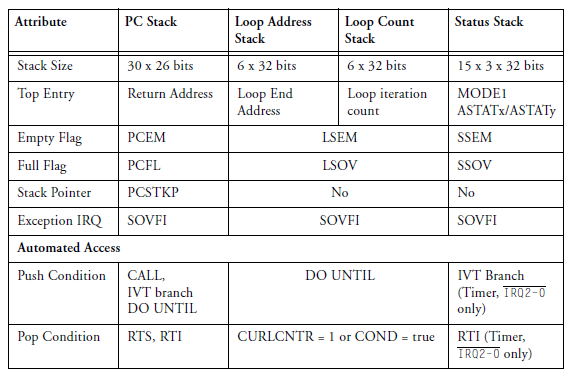
关于中断嵌套的问题，我们约定只默认IRQ2一个中断源入口可以被设置为实时中断，其余的中断入口全部为异步信号。此外，所有的异常与实时中断的处理方式是一致的。

实时中断不可以被打断，也不会进行线程的切换，因此进入实时中断后，即刻关闭总中断直至中断返回。

异步信号可以被实时中断或者异常打断，因此，在完成必要的不能被打断的操作之后，打开全局中端，利用IMASKP寄存器屏蔽所有的异步信号，仅保留被实时中断或者异常打断的可能。在中断返回之前，打开全局中端，清理IMASKP寄存器。

由于SHARC21469硬件上有两套寄存器，包含了通用寄存器R0/S0~R15/S15，DAG1、DAG2 L\M\I\B0~15，MRB\MRF，在一个cycle内可以完成两套寄存器的互换，因此，为了节约中断响应的时间开销，可以通过切换第二套寄存器的方式，来节省极大部分中断响应的时间，只需保存极少的寄存器。

对于PC、Status（MODE1、ASTATx\y）、LOOP，SHARC21469分别有3个硬件栈，用于保存这些重要寄存器的值。具体的可以参见图2-12中各硬件栈的相关属性，包含了硬件自动压栈和推栈的条件，及手动操作的指令。以中断为例，只有在进入Timer中断或者IRQ中断的时候，硬件会自动保存Status Stack，其余情况下不会自动推栈。这些硬件栈同样可以省却中断响应保存现场所需的时间，不同情况下的应用需具体对待。防止因为操作不当导致硬件栈溢出。



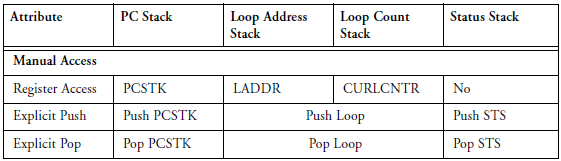


图2-12 SHARC21469硬件栈属性

除开上述的第二套寄存器和硬件栈之外，需要手动保存的寄存器数量锐减，只有MODE2、PX1、PX2等需要手动保存。当然，在进行线程切换的时候，无论是线程之间切换还是通过中断实现线程的切换，都不能使用第二套寄存器，硬件栈的内容也需要手动保存和恢复。对于实时中断和异常的处理，也不使用第二套寄存器，但可以使用硬件栈。在不使用第二套寄存器及硬件栈的情况下，现场的保存和恢复与一般的现场保存的流程是一致的。

对于中断开关，通过分析djyos的中断实现可知，一共有三级中断开关，分别是：单个中断线使能开关，异步信号使能开关和总中断使能开关。其中，单个中断线的使能通过IMASK/LIRPTL可以实现，通过屏蔽掉相应中断的IMASK/LIRPTL位可以实现关闭相应中断线的功能；异步信号使能开关同样通过IMASK/LIRPTL来实现，屏蔽掉所有的异步信号即可；总中断使能开关也不能使用全局中端使能位（IRPTEN），因为该位会关闭所有中断相应，包括异常信号，这是不允许的，因此，总中断使能同样要通过IMASK/LIRPTL来实现。

## 中断号定义

根据djyos的规则，需要在cpu\_peri\_int\_line.h中定义中断号，根据中断向量表可知，SHARC21469的中断向量个数为39个，而实际的中断源有80多个（除去EMUI和RSTI），包含异常。其中S/PDIF、ASRC都是与音频应用相关的接口，我们可以忽略不计。可知，中断源与中断向量并非一一对应，有部分中断源信号是共用的DAI或DPI中断向量作为中断的入口的。我们给所有的中断源分配一个固定对应的中断号。对于DAI或DPI中断，他们各自分别享有一个中断号（DAII和DPII，DAI没有再分为DAIHI和DAILI），对于具体中断源信号，则只能在程序中通过寄存器（DAI\_IRPTL\_HS和DPI\_IRPTL\_SH）来再次判断具体来源了。如图2-13所示。

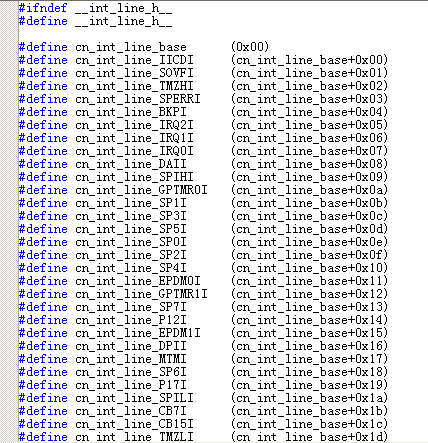


图2-13 中断号的定义

除了中断号的定义外，还需要注意修改在工程文件目录下的int\_config.h文件中对于系统用到的中断进行申明。这里我们申明所有中断都是可用的，共计81个可用中断源（不包含EMUI、RSTI，包含异常）。如图2-14所示。

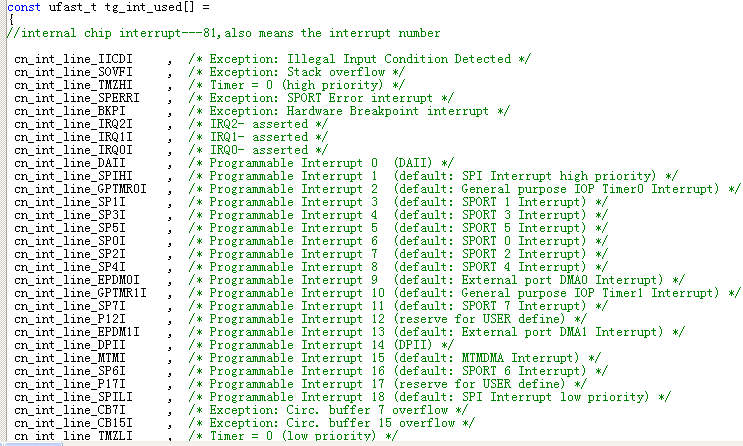


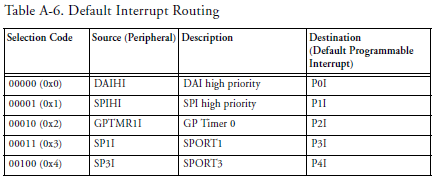
图2-14 可用中断申明

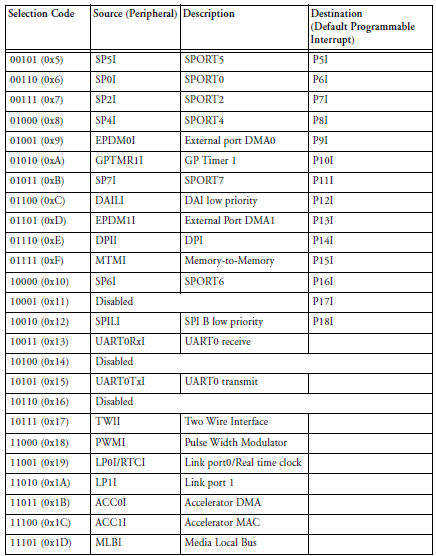
## 中断源与中断号的对应及初始化

考虑到SHARC21469中断系统的特殊性，中断源与IVT中的中断向量并非的固定的一一对应的关系，而是可以通过PICR和DAI、DPI等来实现中断源与中断号的分配，即可以按照实际需求，将指定的中断源链接到IVT中的中断向量上。具体方法可以参见2.2.1及2.2.2中的内容。

中断源与中断号的对应关系是由操作系统完成的，换言之，应用程序员只有API接口函数，而不能决定哪个中断源链接到哪个中断向量上，因此，在移植过程中，需要设定好中断源与中断号的对应关系，并提供不同模块的中断初始化函数。

由2.1小节可知，SHARC21469的IVT中包含了所有的中断及异常向量。其中，19个可编程中断与中断源的默认链接关系如图2-15所示。





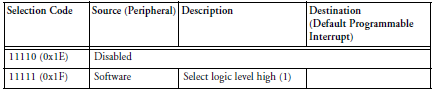


图2-15 可编程中断与中断源的默认链接关系

通过PICR0~3寄存器可以配置中断源与可编程中断向量号的对应关系，PICR0~3的位定义请参阅ADSP-214xx\_hwr\_r1-0.pdf的p492。

DAIH\L、DPI所包含的中断源如图2-16所示。

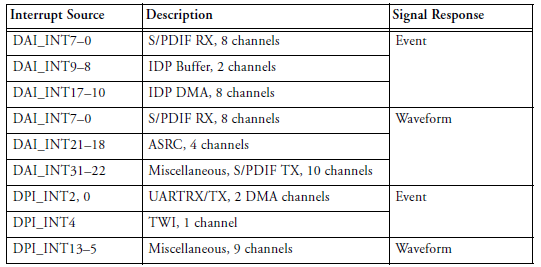


图2-16 DAI\DPI中断源

## 实时中断现场的保存与恢复

实时中断不可被异步信号打断，但可以被异常信号打断。不考虑异常的情况下，我们可以认为实时中断是不会被打断的中断，因此可以不需要考虑中断嵌套的问题。而且，根据我们的约定，只有IRQ2为实时中断，因此我们需要对IRQ2中断进行特别的处理。

实时中断从IVT中直接跳转到\_\_\_real\_interrupt\_handler处，如图2-17所示。

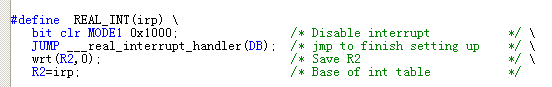


图2-17 实时中断的汇编处理函数

实时中断现场的处理遵照普通的中断处理流程来进行。即我们要确保在退出中断时能够完全恢复现场。因此，我们必须保证所有在实时中断服务函数和汇编处理函数中用到的寄存器，都要压栈保存。

要实现实时中断不被异步信号打断的功能，我们在进入实时中断后，首先修改IMASK/LIRPTL寄存器的屏蔽位，实现将当前值压栈，然后屏蔽掉所有异步信号和实时中断。在退出中断前再恢复IMASK/LIRPTL值。

要实现寄存器的保存和恢复，我们首先要了解SHARC的寄存器使用规则。C语言的规范规定，在进行函数调用时，调用方和被调用方分别保存不同的寄存器值，以确保寄存器值不被错误改变。因此，在进行实时中断的处理的时候，我们不必去管调用方已经保存的寄存器值，而只需要保存调用方不会保存的寄存器值，这样就能实现对所有寄存器值的保存。关于调用方和被调用方分别需要保存的寄存器值，见图1-26和图1-27所示。在程序中，通过调用\_\_\_backup\_real\_int\_regs和\_\_\_restore\_real\_int\_regs这一组子程序来实现实时中断寄存器的保存和恢复。

对于循环栈（硬件栈LOOP STACK），因为在ISR中难免不会使用循环，而SHARC的循环栈的深度是有限的（6层），因此，需要软件将循环栈的值压栈，在退出ISR时恢复。

对于PC栈（硬件栈PC STACK）和状态栈（硬件栈STS STACK），我们可以不必对其进行操作。

实时中断的中断服务函数总引擎为\_\_\_int\_engine\_real。

实时中断的退出可以使用RTI指令来实现。

## 异步信号中断现场的保存与恢复

### 第二套寄存器及硬件栈的使用

对于异步信号的响应，需要使用到第二套寄存器及硬件栈的相关知识。

第二套寄存器的使用可以替代普通的中断处理流程中，寄存器值的保存和恢复过程，因此可以极大的提高中断的响应效率，同时也能避免因为增加程序而引入的BUG。

因为根据djyos的规定，在异步信号中有可能进行线程的切换，因此，这难免就会涉及到对硬件栈的操作。我们在第二套寄存器中执行所有的ISR函数，当需要进行线程切换时，我们可以重新切换为第一套寄存器，根据线程切换的基本流程进行老线程的保存与新线程的恢复工作，然后切回第二套寄存器，通过欺骗中断系统，让中断从新的线程恢复工作。老线程的保存与新线程的恢复都要求保存所有的通用寄存器值和硬件栈。

SHARC的硬件栈有3个，分别是PC栈、STS栈、LOOP栈。其中对STS栈和LOOP栈的读写操作类似，PC栈的有所不同。

PC栈包含两个参数：PCSTK和PCSTKP。PCSTK代表最后一次压入PC栈的PC值，PCSTKP代表PC栈已经使用的深度。对PC栈进行读取时，我们可以直接读取PCSTK和PCSTKP的值，读完一次调用POP PCSTK指令，再进行下一次的读取即可。对PC栈进行写操作时，我们采用先执行PUSH PCSTK指令压栈，然后修改PCSTK值为我们想要写入PC栈的值的方式来实现。

STS栈和LOOP栈的操作类似。读操作首先需要执行一次POP指令，将硬件栈中的值推出，然后再通过读取相应的寄存器值来实现对硬件栈的读取（因为STS中存储的MODE1中的有IRPTEN和NESTM两个很重要的全局中断控制位，因此需要谨慎对待）。写操作是读操作的逆向过程，即通过修改相应寄存器值为所需要的值，再通过PUSH指令来压栈。STS栈和LOOP栈没有类似PCSTKP这样的已用栈深度的计数标志位。要判断这两个栈的满栈或空栈状态，可以通过测试STKYx中相应的标志位来实现。

### 异步信号的响应流程

异步信号能够被实时中断和异常打断。

异步信号从IVT中直接跳转到\_\_\_nop\_interrupt或\_\_\_push\_interrupt处，再从这个跳板跳转到\_\_\_asyn\_interrupt\_handler即异步信号的汇编处理函数处执行。这两个跳板的存在原因是因为，SHARC只会在IRQ中断和TIMER中断处自动push sts，而在其他中断处不会，为了保持硬件栈的一致性，所以才有了这两个跳板函数。如图2-18所示。

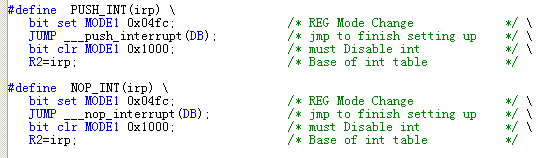


图2-18 异步信号的跳转

异步信号使用了第二套寄存器，不能再共用第一套寄存器所使用的软件栈，因此我们专门为异步信号中断开辟了一个中断栈供其使用。在\_\_\_nop\_interrupt和\_\_\_push\_interrupt这两个跳板中，我们主要做三件事情：第一，对于硬件自动推栈了的，将sts栈再推出来，关闭掉全局中断后，再推进去，对于硬件没有自动推栈的，关闭全局中断后，入栈；第二，重新初始化栈指针（I7）以及栈的范围限定（B7、L7），以使用中断栈；第三，与实时中断一样，我们需要将当前的IMASK/LIRPTL值保存，并屏蔽掉所有异步信号（包括当前正在响应的异步信号），在退出中断时再恢复。注意，上述过程都是在第二套寄存器中进行的。

因为异步信号使用了第二套寄存器，因此不必再有寄存器的保存和恢复过程了。注意，ISR服务函数使用的同样也是第二套寄存器和中断栈。

异步信号的中断服务函数总引擎为\_\_\_int\_engine\_asyn\_signal。

对于循环栈（硬件栈LOOP STACK），因为在ISR中难免不会使用循环，而SHARC的循环栈的深度是有限的（6层），因此，需要软件将循环栈的值压栈，在退出ISR时恢复。

异步信号的退出不使用RTI指令来实现，而是通过从PC栈中取出返回值，然后直接使用JUMP跳转指令进行跳转。不使用RTI指令是因为，RTI指令会根据进入中断时是否推了sts栈来判断是否将sts栈再推出。而我们的在跳板函数中，已经为设计成异步信号都推了一次sts栈，所以这里不能对应。而且在涉及到中断中的线程切换时，我们更加没法保证sts栈所使用的深度了，这时候调用RTI指令是有可能会错误的将sts栈中保存的参数推出的。

## 中断中的线程切换

中断中的线程切换只能在异步信号中发生，在实时中断中是不可以进行切换的。

中断中的线程切换是一个偷梁换柱的过程，ISR函数通过修改寄存器值、中断返回地址等来使中断不从原被中断处返回。因此，中断中的线程切换的主要任务是保存老线程、恢复新线程。

由于我们在异步信号中使用了第二套寄存器，因此，在进入中断中的线程切换函数时，首先要切换到第一套寄存器。第一套寄存器值就是老线程的运行值，我们将之直接压入老线程的线程栈中即可。同时，要将当前的3个硬件栈的值全部推出，压入软件栈中。

在完成了老线程的保护工作之后，我们开始恢复新线程的寄存器及硬件栈。需要注意的是，在调用中断中的线程切换函数时，所传递的新、老线程栈地址是通过第二套寄存器的R4和R8传递的，因此在取用这两个值时，要将R0~R15寄存器切换为第二套，再用Ix寄存器去读取。

中断中的线程切换与线程切换在本质上并没有区别，但要特别注意硬件栈的操作，因为进入中断的时候，会压PC、STS、LOOP三个硬件栈，这三个硬件栈的第一个值是新线程的运行值，是不需要恢复到硬件栈中的，不然会引发硬件栈溢出的BUG。同时，在进行入栈操作时，因为是硬件栈的存在，导致入栈参数的总数难以预知的，所以采用的是压一次栈，修改一次栈指针的方式。而这种方式是无法保证栈指针是2字节对齐的，因此务必要保证该过程不被中断打断，否则有引起BUG的风险。特别是在STS出栈的时候，有可能会直接使能全局中断，导致之前被触发而没有执行的中断开始执行，因此必须要仔细对待。

线程现场的保存和恢复都必须遵循图2-19所定义的栈结构。其中所有通用寄存器的保存和恢复通过调用\_\_\_backup\_all\_regs和\_\_\_restore\_all\_regs函数来实现。

|  |  |  |
| --- | --- | --- |
| 栈指针 | 栈值 | 类别 |
| 1 | PC | PC返回值 |
| 2 | STKYx、USTAT1-4 | 所有的  通用寄存器 |
| 3 | R0~15 |
| 4 | B0~15 |
| … | …… |
| 119 | MRF |
| M | PCSTK | PC STACK |
| M | PCSTKP | PC栈深度 |
| M | MODE1、ASTAT | STS STACK |
| M | n | STS栈深度 |
| M | LADDR、CURLCNTR | LOOP STACK |
| M | n | LOOP栈深度 |

图2-19 保存线程现场的栈结构定义

## Int\_hard.c中函数的实现

Int\_hard.c中各个函数功能的实现要依赖于硬件是否支持，通过分析SHARC21469的硬件系统，我们实现了Int\_hard.c中绝大部分函数的功能，没有实现的函数代表在SHARC21469中不支持或不需要软件实现，仅保留函数名保持一致性。Int\_hard.c中各函数罗列如图2-20所示。

|  |  |  |
| --- | --- | --- |
| 函数名 | 功能 | 是否实现 |
| int\_contact\_asyn\_signal | 接通异步信号开关 | 实现 |
| int\_cut\_asyn\_signal | 断开异步信号开关 | 实现 |
| int\_contact\_trunk | 接通总中断开关 | 实现 |
| int\_cut\_trunk | 断开总中断开关 | 实现 |
| int\_high\_atom\_start | 进入高级原子操作 | 实现 |
| int\_high\_atom\_end | 离开高级原子操作 | 实现 |
| int\_low\_atom\_start | 进入低级原子操作 | 实现 |
| int\_low\_atom\_end | 离开低级原子操作 | 实现 |
| int\_contact\_line | 接通单个中断线开关 | 实现 |
| int\_cut\_line | 断开单个中断线开关 | 实现 |
| int\_clear\_line | 应答中断 | 硬件自动应答 |
| int\_tap\_line | 激发中断 | 实现 |
| \_\_int\_clear\_all\_line | 应答全部中断 | 硬件自动应答 |
| int\_query\_line | 查询中断线请求状态 | 实现 |
| int\_setto\_asyn\_signal | 把指定中断线设置为异步信号 | 不支持 |
| int\_setto\_real | 把指定中断线设置为实时中断 | 不支持 |
| int\_enable\_nest | 使能中断嵌套 | 不支持 |
| int\_disable\_nest | 禁止中断嵌套 | 不支持 |
| int\_set\_prio | 设定嵌套优先级 | 不支持 |
| int\_init | 初始化中断 | 实现 |
| \_\_int\_engine\_all | 总中断引擎 | 不需要 |
| \_\_int\_engine\_real | 实时中断引擎 | 实现 |
| \_\_int\_engine\_asyn\_signal | 异步事件中断引擎 | 实现 |

图2-20 Int\_hard.c中各函数功能的实现

# 线程切换

线程切换，顾名思义就是让系统从老线程转换到新线程执行，其本质是在操作系统调度执行调度策略时，保存/复位/删除老线程上下文，恢复新线程上下文。线程上下文包含了线程执行过程中所包含的特定信息，直接决定了线程切换是否成功。线程上下文包含在线程虚拟机中，下面阐述ADSP21469的线程虚拟机设计方法。

## 4.1．线程虚拟机

参考《都江堰操作系统与嵌入式系统设计》，将ADSP21469平台上线程虚拟机设计成如图所示结构。

线程虚拟机主要包含局部变量区、线程上下文区、空闲区、虚拟机结构体四部分。其中线程上下文区由通用寄存器区及ADSP21469硬件栈相关内容两部分。

|  |
| --- |
|  |
| 图 20线程虚拟机结构 |

线程上下文区依次保存PC，通用寄存器及硬件栈中寄存器内容。硬件栈中内容包括PC Stack, Status Stack, Loop Stack三个栈中保存的寄存器的值，这部分保存的内容应根据相应栈中已保存的深度来确定应保存多少，分别三个变量PC stack number, status stack number, loop stack number来表示三个硬件栈深度。硬件栈寄存器中内容在此处保存的深度是可变的。

通用寄存器除了M5、M6、M7、M13、M14、M15、I7等7个寄存器无需保存，其中M5,M6,M7,M13,M14,M15用于寻址，其值固定，如表所示。

|  |  |  |  |
| --- | --- | --- | --- |
| 表格 11 寻址偏移量寄存器值对应表 | | | |
| 寄存器 | M5 M13 | M6 M14 | M7 M15 |
| 值 | 0 | 1 | -1 |

I6,I7分别为帧指针和栈指针。

通用寄存器部分保存ADSP部分内核寄存器，ADSP21469的内核寄存器如表所示。

这些内核寄存器决定了线程的上下文，考虑到与硬件栈相关的8个寄存器存储比较特殊，其深度可变，特意单独出来，将其他内核寄存器保存在通用寄存器区。

线程上下文通用寄存器压栈顺序表很重要，直接决定了出栈后各个寄存器获得正确的值，为此确定上下文通用寄存器压栈顺序如表所示。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 表格 13上下文通用寄存器区压栈顺序表 | | | | | |
| 编号 | 编号 | 编号 | 编号 | 编号 | 编号 |
| 1 | R1 | 44 | R5 | 85 | I10 |
| 2 | R2 | 45 | R6 | 86 | I11 |
| 3 | I2 | 46 | R10 | 87 | I14 |
| 4 | I13 | 47 | R13 | 88 | I15 |
| 5 | I6 | 48 | R14 | 89 | I12 |
| 6 | MMASK | 49 | R15 | 90 | B12 |
| 7 | MODE2 | 50 | S0 | 91 | M12 |
| 8 | SKTYx | 51 | 92 | L12 |
| 9 | STKYy | 52 | S4 | 93 | I4 |
| 10 | USTAT1 | 53 | 94 | B4 |
| 11 | USTAT2 | 54 | S9 | 95 | M4 |
| 12 | USTAT3 | 55 | 96 | L4 |
| 13 | USTAT4 | 56 | S11 | 97 | B0 |
| 14 | L0 | 57 | 98 | B1 |
| 15 | L1 | 58 | S12 | 99 | B2 |
| 16 | L2 | 59 | 100 | B3 |
| 17 | L3 | 60 | S1 | 101 | B5 |
| 18 | L5 | 61 | S2 | 102 | B8 |
| 19 | L8 | 62 | S3 | 103 | B9 |
| 20 | L9 | 63 | S5 | 104 | B10 |
| 21 | L10 | 64 | S6 | 105 | B11 |
| 22 | L11 | 65 | S7 | 106 | B13 |
| 23 | L13 | 66 | S8 | 107 | B14 |
| 24 | L14 | 67 | S10 | 108 | B15 |
| 25 | L15 | 68 | S13 | 109 | MR0F |
| 26 | PX1 | 69 | S14 | 110 | MR1F |
| 27 | PX2 | 70 | S15 | 111 | MR2F |
| 28 | R0 | 71 | M0 | 112 | MR0B |
| 29 | 72 | M1 | 113 | MR1B |
| 30 | R3 | 73 | M2 | 114 | MR2B |
| 31 | 74 | M3 | 115 | S10 |
| 32 | R4 | 75 | M8 | 116 | S11 |
| 33 | 76 | M9 | 117 | S12 |
| 34 | R7 | 77 | M10 | 118 | S13 |
| 35 | 78 | M11 | 119 | S14 |
| 36 | R8 | 79 | I0 | 120 | S15 |
| 37 | 80 | I1 |  |  |
| 38 | R9 | 81 | I3 |  |  |
| 39 | 82 | I5 |  |  |
| 40 | R11 | 83 | I8 |  |  |
| 41 | 84 | I9 |  |  |
| 42 | R12 |  |  |  |  |
| 43 |  |  |  |  |

线程栈栈底保存虚拟机结构体，其中虚拟机结构定义如下所示。

|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\MM@YF]S(O]1{_S1QDIW]8DN.jpg |

该结构体分别指示该线程栈的栈底，栈深度。在线程切换函数中thread\_vm常作为函数参数，在线程切换函数中常使用到该结构体中三个成员：\*stack、\*stack\_top、stack\_size。\*stack为线程栈指针，线程被抢占时将栈指针保存在\*stack指向的地址处，同时thread\_vm\*vm作为函数传递时，传递的结构体指针也等于\*stack所指向的地址，因为\*stack是struct thread\_vm结构体的第一个成员。\*stack\_top所指向的地址处保存线程虚拟机栈顶地址。stack\_size则为线程虚拟机可使用的栈大小，线程虚拟机实际的栈大小为stack\_size+sizeof(struct thread\_vm)。

|  |
| --- |
| \_\_\_asm\_reset\_thread  //重置线程函数，函数原型:void \_\_asm\_reset\_thread(void (\*thread\_routine)(void),  @ struct thread\_vm \*vm);  ……  dm(I7,M7)=R2; //将返回地址压入软件栈  I4=R8; //根据ADSP21469参数传递规则，R8=stack  R2=DM(M6,I4); //取虚拟机栈顶指针 根据虚拟机结构，第二个元素为栈顶变量  …… |

## 4.2．线程相关函数实现

线程切换相关函数主要有以下6个：创建线程虚拟机函数、复位线程虚拟机函数、上下文切入函数、复位老线程，切入新线程函数、上下文切换函数及从异步信号ISR中返回时上下文切换等。下面分别阐述这6个函数在ADSP21469平台上的实现。

### 4.2.1 创建线程虚拟机

函数原型为：struct thread\_vm\*\_\_create\_thread(struct event\_type\*evtt\_id)

函数功能为为事件类型创建伪虚拟机，初始化上下文环境，安装执行函数，构成完整线程。该函数用C语言实现，在移植该函数时必须要根据处理器栈类型及栈对齐规则等确定线程虚拟机栈深、栈顶等信息。创建线程虚拟机函数移植需要修改的三点如下图黄色标示：

|  |
| --- |
| struct thread\_vm \*\_\_create\_thread(struct event\_type \*evtt,u32 \*stack\_size)  {  struct thread\_vm \*result;  ptu32\_t len;  //计算虚拟机栈:线程+最大单个api需求的栈  ① len = evtt->stack\_size+cn\_kernel\_stack+sizeof(struct thread\_vm);  //栈顶需要对齐，malloc函数能保证栈底是对齐的，对齐长度可以使栈顶对齐 ② len = align\_up\_sys(len);  ③ result=(struct thread\_vm \*)m\_malloc\_stack(len);  \*stack\_size = len;  if(result==NULL)  {  djy\_error\_login(enum\_mem\_tried,NULL); //内存不足，返回错误  return result;  }  #if cfg\_stack\_fill != 0  memset(result,cfg\_stack\_fill,len);  #endif  ④ result->stack\_top = (u32\*)((ptu32\_t)result+len); //栈顶地址，移植敏感  result->next = NULL;  result->stack\_size = len - sizeof(struct thread\_vm); //保存栈深度  result->host\_vm = NULL;  //复位虚拟机并重置线程  \_\_asm\_reset\_thread(evtt->thread\_routine,result);  return result;  } |

①处计算栈虚拟机深度，虚拟机深度等于线程所需栈大小evtt->stack\_size与最大单个api需求的栈大小cn\_kernel\_stack及thread\_vm结构体大小三者之和。其中第一项需根据具体应用程序确定，在用户注册事件类型时需指定执行线程所需的栈大小(不包括系统服务所需的栈)。系统服务所需的最大的栈在Arch.h文件中通过宏定义#define cn\_kernel\_stack 0x1000确定。thread\_vm结构体由sizeof(struct thread\_vm)确定。

②处为根据栈对齐要求重新修正栈深len，栈对齐要求设置同样在Arch.h中通过宏定义设置，ADSP21469栈为2字节对齐，如下所示：

|  |
| --- |
| //定义一些与对齐相关的宏  #define mem\_align 1 //如果目标系统没有对齐要求，改为0  #define align\_size 2 //ADSP21469要求2字节对齐 |

③和④两处与栈生长方向及栈类型相关，ADSP21469为向下生长的满栈，故③和④设置如上所示，当栈生长方向及栈类型为其他三种情形时，上述两处应分别设置如下：

|  |
| --- |
| 向下生长空栈：  result=(struct thread\_vm \*)m\_malloc\_stack(len);  result->stack\_top = (u32\*)((ptu32\_t)result+len)-1;  向上生长的满栈：  result=(struct thread\_vm \*)m\_malloc\_stack(len)+len-sizeof(struct thread\_vm);  result->stack\_top = (u32\*)((ptu32\_t)result)-1;  向上生长的空栈：  result=(struct thread\_vm \*)m\_malloc\_stack(len)+len-sizeof(struct thread\_vm);  result->stack\_top = (u32\*)((ptu32\_t)result); |

### 4.2.2 重置线程

函数原型为：void \* \_\_asm\_reset\_thread(void (\*thread\_routine)(struct event\_script \*),

struct thread\_vm \*vm)

功能: 初始化线程的栈，并挂上线程执行函数，新创建线程时将会调用本函数

参数: thread\_routine，被重置的线程函数；vm，线程指针

返回: 初始化结束后的当前栈指针。

重置线程即需要将线程虚拟机恢复到初始状态，重置线程需要完成以下工作：①将线程引擎\_\_djy\_vm\_engine放置于虚拟机PC位置处；②将通用寄存器及硬件栈中内容初始化，对于寄存器初始状态为随机值情形只需移动栈指针，对于某些特定寄存器或标志位赋予初始值；③将更新后的栈指针写入thread\_vm->vm所指向的位置处。

重置线程函数传递两个参数： event\_script\*和\*vm，根据ADSP21469参数传递规则，R4传递第一个参数event\_script\*，R8传递第二个参数\*vm。对虚拟机初始化必须要先获得栈顶位置，由栈顶位置依次向下初始化。由5.1节可知\*vm为结构体地址，也等于\*stack所指向的地址，vm中第二个成员即为\*stack\_top栈顶地址，通过将\*vm所指向的地址往上移动一位(ADSP21469字长位宽为32位，故只需移动一个位置)，即可获得栈顶位置。另外由5.1节可知在线程虚拟机初始状态下共使用了127位：1(PC)+117(通用寄存器)+9(硬件栈相关)=127，为使栈对齐，对虚拟机重置完成后栈指针应往下移动128个位置，故首先可以将栈指针移动128位。在进行上述操作之前，务必要先保存将要使用的所有寄存器及包含中间有效值的寄存器，具体代码如下所示：

|  |
| --- |
| //首先将本汇编函数使用到的寄存器入栈保护  shift(4);  wrt(R1,reg\_vm\_offset+4);  R1=I2; wrt(R1,reg\_vm\_offset+3);  R1=I4; wrt(R1,reg\_vm\_offset+2);  R1=I13; wrt(R1,reg\_vm\_offset+1);  I4=R8; //根据ADSP21469参数传递规则，R8=stack  R1=I2;DM(I7,M7)=R1; //保存I2，由于下面将使用I2作为栈指针，首先将I2压栈保护  R2=DM(M6,I4);I2=R2; //取虚拟机栈顶指针 根据虚拟机结构，第二个元素为存储栈顶指针  //取得虚拟机栈顶指针后，即按照线程栈结构依次对栈中各个位置初始化  R2=\_\_\_djy\_vm\_engine; //取虚拟机引擎指针  modify(I2,-128); //先申请栈空间  dm(reg\_vm\_offset+128,I2)=R2; //\_djy\_vm\_engine入口作为PC地址，将PC地址存入软件栈，线程栈第一个位置存储PC |

接下来即可按照5.1节所述线程栈顺序将某些特定位置寄存器及变量值压栈初始化。需要初始化寄存器或位置有以下这些：

1.PC位置，将线程引擎置入该处；

2.通用寄存器处需要初始化的STKYx和R4寄存器，STKYx初始值为0x05400000，将thread\_routine指向的地址存入R4寄存器。

3.硬件栈相关的，首先PC stack，此时PC stack应为空，直接将PC Stack number位置处置零；由于MODE1寄存器相关位控制着中断及通用寄存器切换，在操作MODE1寄存器需犹为注意。ASTAT初始化值为零。

|  |
| --- |
| //初始化STATUS STACK在线程栈中值  R1=MODE1;  R2=0x00001000; //全局中断使能位置0  R2=NOT R2;  R2=R1 and R2;  R1=0x00000800; //中断嵌套使能位置1  R2=R1 or R2;  R1=0x04fc; //以免第二套寄存器是使能的，引发BUG  R1=NOT R1;  R2=R1 and R2;  dm(reg\_vm\_offset+8,I2)=R2;  ASTAT=0x00000000;  R2=ASTAT;  dm(reg\_vm\_offset+7,I2)=R2;  R2 = 0x00000000;  dm(reg\_vm\_offset+6,I2)=R2;  R2=1;  dm(reg\_vm\_offset+5,I2)=R2; //status stack 此时压入初始化值并将status number置1 |

接着将Loop Stack栈相关的三个寄存器初始值压栈，LCNTR、LADDR、CURLCNTR初始值分别为0x00000001、0xFFFFFFFF、0xFFFFFFFF。

最后一定要将当前栈指针存入\*vm地址处，并将I2出栈恢复原值，最后即可函数返回，代码如下：

|  |
| --- |
| I4=R8;  R2=I2;  dm(I4,M5)=R2; //将栈指针保存到R8所在的地址处，也即vm的栈底被更新  R1=DM(M6,I7);I2=R1; //恢复I2  FETCH\_RETURN  RETURN (DB);  rframe;  nop; |

### 4.2.3 切入上下文

函数原型：void \_\_asm\_turnto\_context(struct thread\_vm \*new\_vm)

功能: 系统初始化完成后，启动多事件调度并切入第一个事件.

参数: new\_vm，首次启动的线程控制块指针

返回: 无

当事件完成，没有必要保存旧事件的上下文，直接切入新线程时，即可调用该函数，同时在cpu.h文件通过宏定义#define \_\_asm\_start\_thread \_\_asm\_turnto\_context使操作系统启动时，切入的第一个事件也调用该函数。

该函数需要完成的功能使将\*new vm所指向的线程虚拟机中所保存的内容出栈。只要获得new vm虚拟机栈指针，即可按照5.1节虚拟机结构依次出栈即可。由5.3.2可知，vm的栈指针保存在\*vm所指向的地址处，在上下文切入函数中\*new vm参数通过R4传递，只需读取R4所指向的地址即可获知new vm的栈指针。另外在出栈之前必须要给B7(栈底)、L7(栈深)寄存器赋值，设置虚拟机范围。在5.1节中已指出通过thread\_vm\* vm结构体可获知栈底及栈深。具体代码如下：

|  |
| --- |
| I4=R4; //R4传递 new vm结构体首地址，也即new vm->stack  I12=I4;  R2=DM(M6,I4); //取虚拟机栈顶指针 根据虚拟机结构，第二个元素为存储栈顶指针  //接下来要取vm结构体中stack\_size这个变量 vm->stack\_size这个成员在vm结构体中第4个位置  R0=I4;  R1=2;  R0=R1+R0;  I4=R0;  R1=DM(M6,I4); //取得vm->stack\_size值，接下来将vm->stack\_size值赋给L7，同时根据栈顶及栈深L7求出栈底并赋值给B7  L7=R1;  R0=R2-R1;  R0=R0+1;  B7=R0;  I4=I12;  R2=DM(M5,I4);  I7=R2; |

在出栈之前还需要完成一步：需要调用int\_half\_\_enable\_asyn\_signal，此处为汇编调用C函数，要完成顺利调用，要注意栈指针及PC的保存及恢复工作，具体代码如下：

|  |
| --- |
| //开始调用C函数int\_half\_enable\_asyn\_signal,在调用C函数之前需完成两步预备动作  R2=I6;  I6=I7;  pm\_ptr=\_int\_half\_enable\_asyn\_signal;  JUMP(M13,pm\_ptr)(DB);  DM(I7,M7)=R2;  DM(I7,M7)=PC; |

接下来即可依次恢复硬件栈相关内容及通用寄存器。最后将PC位置处内容出栈，并使下一步程序即将跳转到该处，在跳转到该处之前，使能全局中断，代码如下：

|  |
| --- |
| get(pm\_ptr,1);  alter(1);  I12=dm(m7,i6);  jump(m14,I12)(db);  nop;  bit set MODE1 0x1000; // Enable int |

### 4.2.4 复位老线程，切换到新线程

函数原型：void \_\_asm\_reset\_switch(void (\*thread\_routine)(void),

struct thread\_vm \*new\_vm,struct thread\_vm \*old\_vm)

功能: 把old\_vm的上下文重新初始化到新创建的状态，然后切换到新线程的上下文中继续执行。当一个在常驻内存的线程中处理的事件完成后，不能删除该线程，必须复位该线程，然后切换到其他就绪线程，这时候就需要调用本函数。因此本函数在被复位的线程上下文里面执行。

参数: thread\_routine，被重置的线程函数new\_vm，目标线程的线程old\_vm，被复位的线程

返回: 无

此函数可以分为两个过程，首先将当前线程复位到初始状态，然后切入到新线程，这两部分工作可以参考5.2.2及5.2.3中阐述的重置线程及切入上下文两个函数。需要注意的是根据ADSP21469参数传递规则，\_\_asm\_reset\_switch函数的三个参数\*thread\_routine、\*new\_ vm、\*old\_vm分别通过R4、R8、R12传递。

### 4.2.5 上下文切换

函数原型：void \_\_asm\_switch\_context(struct thread\_vm \*new\_vm,struct thread\_vm \*old\_vm)

功能: 保存当前线程的上下文，切换到新线程的上下文中继续执行。

参数: new\_sp，新上下文的栈指针

参数: old\_sp，旧上下文的栈指针的指针，即&vm->stack。无需提供旧上下文栈指针，sp寄存器的当前值就是。

返回: 无

上下文切换函数其本质是保存当前线程上下文，然后切入到新线程上下文区，其关键是制造old\_vm好像从\_\_asm\_switch\_context返回的假象，要做到这一点的关键是将调用\_\_asm\_switch\_context函数的PC下一条指令地址写入当前线程虚拟机PC位置，这样当重新返回到当前线程时，将虚拟机PC位置保存的内容出栈，并使PC跳到该处即可。由1.3.3节可知，在C函数调用上下文切换函数时，编译器自动执行了四行指令：

r2 = i6;

i6 = i7;

dm(i7,m7)=r2;

dm(i7,m7)=PC;

在上下文切换函数中我们只需将已经压栈的PC值读出并且写进old\_vm虚拟机的PC位置处即可。

R0=DM(M7,I6);

put(R0);

# 异常及其处理

异常处理要完成的任务具体是：系统发生异常后，保存异常现场将现场信息写进FLASH中，以便于解析，然后根据异常类型采取相应的措施(如Hardware Reset/Software Reset/Running Reset等)。

## 5.1．ADSP21469异常类型

ADSP21469异常类型有12种，如表格 14 ADSP21469异常中断表所示：

表格 14 ADSP21469异常中断表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Interrupt Number | Exp Type | Vector Address | Interrupt Name | Function |
| 0 | 0x1 | 0x00 | EMUI | Emulator |
| 2 | 0x 2 | 0x08 | IICDI | Illegal IOP access or unaligned long word access detected |
| 3 | 0x 3 | 0x0C | SOVFI | Status loop or mode stack overflow; or PC stack full |
| 5 | 0x 4 | 0x14 | SPERRI | SPORT Error Interrupt |
| 6 | 0x 5 | 0x18 | BKPI | User Hardware Breakpoint |
| 30 | 0x 6 | 0x78 | CB7I | Circular Buffer 7 overflow(DAG1) |
| 31 | 0x 7 | 0x7C | CB15I | Circular Buffer 15 overflow(DAG2) |
| 33 | 0x 8 | 0x84 | FIXI | Fixed-point overflow |
| 34 | 0x 9 | 0x88 | FLTOI | Floating-pointing overflow exception |
| 35 | 0xa | 0x8C | FLTUI | Floating-pointing underflow exception |
| 36 | 0xb | 0x90 | FLTII | Floating-pointing invalid exception |
| 37 | 0xc | 0x94 | EMULI | Emulator low priority interrupt |

上述12个异常在中断向量表中对应12个异常中断号，但是可以归类为异常中断类。上述12个异常中断均通过IRPTL寄存器锁存中断信号，要想在异常发生后进入异常中断服务函数，必须要先使能异常中断，首先使能总中断，然后使能相应的异常中断(对IMASK寄存器相应位置1)。

**IICDI**：IICDI属于内存访问异常，在两个情形下会产生异常：Illegal IOP access和Unaligned 64-bit forced long word access。Illegal IOP access指当MODE2寄存器的*IIRAE* 位置1时，处理器会监视I/O处理器寄存器访问，当检测到有非法的输入时产生IICDI异常中断。Unaligned 64-bit forced long word access指在MODE2寄存器的*U64-MAE*位置1时，处理器会监视不对齐的64位内存访问。An unaligned access is an odd numbered address normal word access that is forced to 64 bits with the LW mnemonic.当检测到发生不对齐64位内存访问时会产生IICDI异常中断。

IICDI发生时，系统会直接崩溃。IICDI异常需要保存的信息：MODE2、MODE1、STKYx、IRPTL、IMASK等寄存器、发生异常时PC指针、栈指针、R/S/I/M/B/L等96个通用寄存器当前值、访问内存区前后50个内存区空间数据。

**SOVFI: S**OVFI指硬件栈溢出，ADSP21469硬件栈属性表如图所示。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 表5 ADSP21469栈属性表 | | | | |
| Attribute | PC Stack | Status Stack | Loop Address Stack | Loop Count Stack |
| Stack Size | 30\*26bit | 15\*3\*32bit | 6\*32bit | 6\*32bit |
| Top Entry | Return Address | MODE1 ASTATx/ASTATy | Loop End Address | Loop iteration count |
| Empty Flag | PCEM | SSEM | LSEM | |
| Full Flag | PCFL | SSOV | LSOV | |
| Stack Pointer | PCSTKP | No | No | |
| Exception IRQ | SOVFI | SOVFI | SOVFI | |
| Automated Access | | | | |
| Push Condition | CALL/IVT branch/DO UNTIL | IVT branch(Timer, only) | DO UNTIL | |
| Pop Condition | RTS/RTI | RTI(Timer, only) | CURLCNTR=1 or COND=true | |
| Manual Access | | | | |
| Register Access | PCSTK | No | LADDR | CURLCNTR |
| Explicit Push | Push PCSTK | Push STS | Push Loop | |
| Explicit Pop | Pop PCSTK | Pop STS | Pop Loop | |

SOVFI发生的条件是：PC stack overflow，Loop stack overflow, Status stack overflow。三个硬件栈溢出的定义如下：

表格 15 SOVFI异常发生条件表

|  |  |  |  |
| --- | --- | --- | --- |
| 栈名 | 栈深 | 栈溢出定义 | 栈溢出及栈空检测位(STKYx) |
| PC Stack | 30 | 当PC Stack已使用29层，即PCSTKP=29时产生SOVFI异常中断，第30层用于存放返回地址。 | PC stack full: Bit 21(*PCFL*) full(if 1), not full(if 0)  PC stack empty: Bit 22(*PCEM*) Empty(if 1), not empty(if 0) |
| Status Stack | 15 | 当15层全部使用完毕后再试图向Status Stack压栈时产生SOVFI异常中断。 | Status Stack overflow: Bit23(*SSOV*) overflow(if 1), not overflow(if 0)  Status Stack empty: Bit24(*SSEM*) empty (if 1), not empty(if 0) |
| Loop Stack | 6 | 当6层全部使用完毕后再试图向Loop Stack压栈时产生SOVFI异常中断。 | Loop stacks overflow: Bit25(*LSOV*) overflow(if 1), not overflow(if 0)  Loop Stack empty: Bit26(*LSEM*) empty (if 1), not empty(if 0) |

测试SOVFI异常(Loop Stack溢出)出现时，系统会报SOVFI异常(SOVFI置1)，STKYx 寄存器LSOV标志位会置1，不会导致程序直接崩溃，程序可以继续往下执行，只是无法再压栈。

|  |
| --- |
| C:\Documents and Settings\Administrator\桌面\QQ图片20131209145445.jpg |

图 21 测试SOVFI异常例

测试SOVFI异常(Status Stack溢出)，发现一个奇怪现象，当向status stack压第16次时，SOVFI置1，但是SSOV没有置1，而理论上SSOV此时应该置1，如图 22所示，why?另外，Status Stack栈溢出时，继续压栈不会导致死机，只是不能继续成功压栈。

|  |
| --- |
| C:\Documents and Settings\Administrator\桌面\EP6](HU]W7FZXQ_P4WVWI3I.jpg |

图 22 Status Stack溢出异常测试例

测试SOVFI异常(PC Stack溢出情形)，发现当压了28次时，SOVFI置1，压第29次时PCFL置1，(注意，这部分测试结果与技术文档不一致，技术文档提到压第29次时才报SOVFI中断。)压第30次时，程序直接Halt(死机)。

|  |
| --- |
| C:\Documents and Settings\Administrator\桌面\QQ图片20131209143930.jpg |
| PC Stack使用28层时寄存器状态 |
| C:\Documents and Settings\Administrator\桌面\QQ图片20131209144014.jpg |
| PC Stack使用29层时寄存器状态 |

图 23 SOVFI异常PC Stack溢出测试例

SOVFI异常中断优先级比实时中断IRQ2高，从测试效果看，当发生SOVFI异常时，应当在相应的ISR中先将当前PC指针保存、当前栈指针、STKYx、MODE1、IRPTL、IMASK等寄存器、R/S/I/M/B/L等96个通用寄存器当前值及三个硬件栈中全部数据等现场信息写进Flash中然后软复位。

**SPERRI: SPERRI**指ADSP21469 Serial Ports(SPORTS)出现异常时报的异常中断。ADSP21469有8个SPORTS(SPORT0~SPORT7)。SPORT支持多种通信协议，可以与多种类型硬件接口相连进行数据类型转换。每个SPORT有独立的控制寄存器和Buffer。每个SPORT有两个双向的通道(A和B)。SPORT异常由两方面组成：帧同步异常和data buffer异常(unexpected Frame Syncs and not meeting real time requirements (data buffer under/overflow))。

每个SPORT由其独立的寄存器控制，SPORT与SPERRI异常相关的寄存器有以下：

SPORT Divisor Registers(**DIVx**):该寄存器用于设置帧同步及时钟分频倍数；

Serial Control Registers(**SPCTLx**):该寄存器用于控制SPORT发射和接收数据方式；

SPORT Control 2 Registers(**SPCTLNx**):该寄存器用于设置对I2S帧同步边缘检测；

SPORT Multichannel Control Registers(**SPMCTLx**):该寄存器用于设置SPORT多通道通信模式；

SPORT Active Channel Select Registers(**SPxCSy**):用于设置SPORT通道选择；

SPORT Compand Registers(**SPxCCSy**):

Error Control Register(**SPERRCTLx**):该寄存器用于控制及报告SPORT产生的中断状态；SPERRCTL寄存器位定义及说明如图 24 SPERRCTL寄存器位说明所示。

|  |
| --- |
| C:\Documents and Settings\Administrator\桌面\QQ图片20131213193613.jpg  C:\Documents and Settings\Administrator\桌面\QQ图片20131213193643.jpg  C:\Documents and Settings\Administrator\桌面\QQ图片20131213193700.jpg |

图 24 SPERRCTL寄存器位说明

SPORT Error Status Register(**SPERRSTAT**):该寄存器包含了SPORT0~SPORT7所有的中断状态，SPERRSTAT寄存器位定义如图 25 SPERRSTAT寄存器位定义图。

|  |
| --- |
| C:\Documents and Settings\Administrator\桌面\QQ图片20131213194516.jpg |

图 25 SPERRSTAT寄存器位定义图

由SPERRCTLx和SPERRSTAT寄存器可知SPERRI异常具体是由哪一个SPORT哪一个原因造成的。由以上分析可知，当报SPORT异常时，需要保存的现场信息除了当前PC指针保存、当前栈指针、STKYx、STKYy、MODE1、IRPTL、IMASK等寄存器、R/S/I/M/B/L等96个通用寄存器当前值外，还需保存上述寄存器。

**CB7I/CB15I:**DAG1/DAG2在环形缓冲区(Circular buffers)溢出时产生CB7I/CB15I异常中断。环形缓冲区溢出指I7(对应DAG1)I15(对应DAG2)值超出了环形缓冲区范围。环形缓冲区范围由B7/B15(指定环形缓冲区起始地址)L7/L15(指定环形缓冲区结束地址)确定。环形缓冲区使能位(*CBUFEN*)位于MODE1寄存器，复位时默认为0。

环形缓冲区使用步骤：

1. Enable circular buffering (BIT SET MODE1 CBUFEN;). This operation is only needed once in a program.

2. Load the buffer’s base address into the B register. This operation automatically loads the corresponding I register. If an offset is required the I register can be changed accordingly.

3. Load the buffer’s length into the corresponding L register. For example, L0 corresponds to B0.

4. Load the modify value (step size) into an M register in the corresponding DAG. For example, M0 through M7 correspond to B0.

Alternatively, the program can use an immediate value for the modifier.

**CB7I/CB15I**发生时程序不会立刻Halt，但可以肯定的是发生异常后访问数据肯定不正确，程序基本无效，必须要软复位。此时首先在Flash中需保存该异常的相关信息应包括：当前PC指针保存、当前栈指针、STKYx、STKYy、MODE1、IRPTL、IMASK等寄存器、R/S/I/M/B/L等96个通用寄存器当前值，然后软复位。

**FIXI/FLTUI/FLTOI/FLTII**：ADSP21469计算处理单元PEx/PEy中ALU和Multiplier对于执行定点和浮点计算时提供异常监测信息，每个单元出现上溢(overflow)/下溢(underflow)/非法操作等时会在产生异常中断。

表格 16 定点/浮点异常表

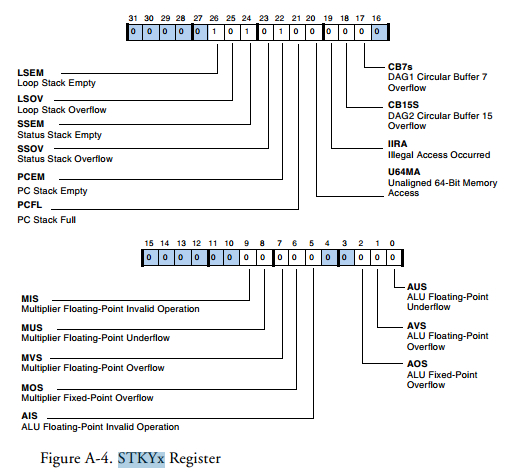
|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\JHK)HZON64A$}RE6J)~IBFH.jpgC:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\@~2$[X7{%([0AJ09LGCR~AV.jpg |

当发生异常时，Arithmetic Status Registers(ASTATx/ASTATy)和Sticky Status Registers会自动在相应标志位置1，如图 22所示。

|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\]C)UOI]RWGPBV@}794R4TLN.jpg |

图 26 ASTATx寄存器

IICDI、SOVFI、CB7I/CB15I、FIXI/FLTUI/FLTOI/FLTII等8个异常在异常发生时会在STKYx/STKYy相应的标志位置1，具体的标志位如图 23 异常标志位图所示。



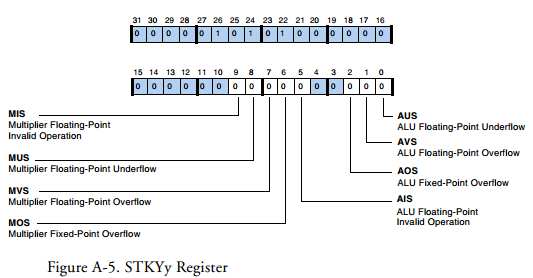


图 27 异常标志位图

关于FLTUI异常测试：

测试背景：最初是在djiyos在ADSP21469初步可以跑线程及中断的时候发现Interrupts Registers中FLTUI置1了，报异常了！经过单步调试，发现问题出在tick中断ISR中的

if((u32)s64g\_os\_ticks %u32g\_RRS\_slice == 0)

这行命令中，u32g\_RRS\_slice我已经宏定义为100，在单步调试时s64g\_os\_ticks=1。

这个时候由于异常处理ISR还未写，直接忽视掉了，程序也貌似没有其他任何异常。最初怀疑是u32g\_RRS\_slice变量定义为static原因(因为我将u32g\_RRS\_slice定义为非static时，没有报FLTUI异常)，经过和ADI AE Frank WU交流，其指出：

但是这个问题与static类型无关，我的例子里就没有使用static类型。关键是做除法的两个变量如果都是外部输入的，则编译器必须要调用定点除法的库函数（本例子中调用了\_\_divrem\_s64()库函数），就会出现此问题。

在SHARC上，将定点除法中被除数小于除数的情况都归为了underflow问题，触发了FLTUI中断（但是这样规定也确实会有歧义）。但是，你们可以看到这时的计算结果始终是正确的，但是要忽略掉FLTUI中断。

他给出了工程例子，发现与static 确实无关。但是我在例子移到djyos-DSP工程上时，发现还是会报FLTUI异常，比较两个工程不同之处，发现问题在于u32g\_RRS\_slice等变量是全局变量和局部变量有关，我在Frank WU工程上将变量作为局部变量时就没有报FLTUI，如图 26 FLTUI测试实例。

|  |
| --- |
| C:\Documents and Settings\Administrator\桌面\QQ图片20131212155702_副本.jpg |
| 图a 全局变量情形 |
| C:\Documents and Settings\Administrator\桌面\QQ图片20131212155920_副本.jpg |
| 图 b 局部变量情形 |

图 28 FLTUI测试实例

我们对ADSP21469关于FIXI/FLTUI/FLTOI/FLTII在什么情形下会报异常以及该如何处理有必要深究一下。

注意：当发生FIXI/FLTUI/FLTOI/FLTII异常时，STKYx/STKYy对应的标志位会置1(具体位稍后阐述)，响应中断后不会自动清零，必须要在相应的中断服务函数ISR中清零。需要注意的是同一个异常可能ALU或Multiplier造成，不同的原因在STKYx/STKYy寄存器中有独立的标志位。通过判断相应的标志位即可获知实际的造成原因。

FIXI/FLTUI/FLTOI/FLTII：这四个异常需要保存的现场信息有：当前PC指针保存、R/S32个通用寄存器值、当前栈指针、STKYx、STKYy、ASTATx、ASTATy、MODE1、IRPTL、IMASK等寄存器。注意在ISR中要清STKYx/STKYy相应标志位。

上述12种异常对应12个异常中断，每个异常中断对应一个中断优先级，需要注意的是IICDI、SOVFI、SPERRI三个异常优先级高于实时中断IRQ2优先级，这三个异常发生时直接软复位，无需考虑保存现场并返回到中断前处。但是CB71、CB15I两个异常中断优先级较低(30、31)，比实时中断及异步信号优先级都要低，直接等异步信号或实时中断执行完到CB71、CB15I异常ISR中执行复位即可。

对于FIXI/FLTUI/FLTOI/FLTII四个异常，是复位还是其他情形，待定。

## 5.2 异常现场信息保存机制

在中断向量表处，所有异常都指向EXCEPTION，在EXCEPTION中，将中断向量号irp保存到R2中，然后跳转到\_\_\_Default\_sys\_general\_exception(异常服务函数ISR)，保存异常现场信息。在\_\_\_Default\_sys\_general\_exception中将现场信息压栈保护，该ISR通过汇编语言实现，在该函数中根据异常类型将异常现场信息写进RAM指定位置处，起始地址为0xe7e00(片上RAM Block3:E8000~FFFFF)，在保存完现场后然后跳到u32 sys\_general\_exp\_default\_dealer(unsigned int exp\_number,unsigned int\*pStartAddress)函数中， sys\_general\_exp\_default\_dealer函数用C语言实现，在该函数中将从0xe7e00地址开始依次将数据读出写进Flash中，写完后根据异常类型采取相应的措施(如Hardware Reset/Software Reset/Running Reset等)，需要注意的是EMUI、BKPI、EMULI三个异常属于调试异常，暂时不考虑，重点考虑其他9种异常。

异常现场信息依次如表格 17 异常通用现场保存信息表所示。由6.1.1可知，对于IICDI/SOVFI/测试发现的问题等异常都有多种异常的原因，通过查询STKYx寄存器可获知具体原因，将异常具体原因以编号形式写进Flash中。另外各异常类型可通过查询异常中断号获知异常类型，异常类型编号及异常原因编号如表格 18 异常类型编号及异常原因编号所示。

表格 17 异常通用现场保存信息表

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信息 | 编号 | 信息 |
| 0 | 异常类型编号 | 32 | R13 |
| 1 | R1 (后面做参数传递使用，先保存再使用) | 33 | R14 |
| 2 | MODE1 | 34 | R15 |
| 3 | 造成异常具体原因编号 | 35 | S0 |
| 4 | MODE2 | 36 |
| 5 | PC指针 | 37 | S1 |
| 6 | 栈指针I7 | 38 | S2 |
| 7 | ASTATx | 39 | S3 |
| 8 | STKYx | 40 | S4 |
| 9 | IRPTL | 41 |
| 10 | IMASK | 42 | S5 |
| 11 | PX1 | 43 | S6 |
| 12 | PX2 | 44 | S7 |
| 13 | R0 | 45 | S8 |
| 14 | 46 | S9 |
| 15 | R3 | 47 |
| 16 | 48 | S10 |
| 17 | R4 | 49 | S11 |
| 18 | 50 |
| 19 | R5 | 51 | S12 |
| 20 | R6 | 52 |
| 21 | R7 | 53 | S13 |
| 22 | 54 | S14 |
| 23 | R8 | 55 | S15 |
| 24 | 56~71 | I0~I15 |
| 25 | R9 | 72~87 | M0~M15 |
| 26 | 88~103 | B0~B15 |
| 27 | R10 | 104~119 | L0~L15 |
| 28 | R11 |  |  |
| 29 |  |  |
| 30 | R12 |  |  |
| 31 |  |  |

表格 18 异常类型编号及异常原因编号

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 异常类型 | 异常中断号 | 异常主编号(majorexp\_type) | 异常原因/分支 | 分支异常编号(minorexp\_type) |
| IICDI | 0x0 | 2 | Illegal IOP access | 1 |
| unaligned long word access detected | 2 |
| SOVFI | 0x1 | 3 | PC stack full | 3 |
| Status stack overflow | 4 |
| Loop stack overflow | 5 |
| SPERRI | 0x3 | 4 | SPORT0\_Unexpected\_Frame\_Syncs | 6 |
| SPORT0\_A\_Data\_Error | 7 |
| SPORT0\_B\_ Data\_Error | 8 |
| SPORT1\_Unexpected\_Frame\_Syncs | 9 |
| SPORT1\_A\_Data\_Error | 10 |
| SPORT1\_B\_ Data\_Error | 11 |
| SPORT2\_Unexpected\_Frame\_Syncs | 12 |
| SPORT2\_A\_Data\_Error | 13 |
| SPORT2\_B\_ Data\_Error | 14 |
| SPORT3\_Unexpected\_Frame\_Syncs | 15 |
| SPORT3\_A\_Data\_Error | 16 |
| SPORT3\_B\_ Data\_Error | 17 |
| SPORT4\_Unexpected\_Frame\_Syncs | 18 |
| SPORT4\_A\_Data\_Error | 19 |
| SPORT4\_B\_ Data\_Error | 20 |
| SPORT5\_Unexpected\_Frame\_Syncs | 21 |
| SPORT5\_A\_Data\_Error | 22 |
| SPORT5\_B\_ Data\_Error | 23 |
| SPORT6\_Unexpected\_Frame\_Syncs | 24 |
| SPORT6\_A\_Data\_Error | 25 |
| SPORT6\_B\_ Data\_Error | 26 |
| SPORT7\_Unexpected\_Frame\_Syncs | 27 |
| SPORT7\_A\_Data\_Error | 28 |
| SPORT7\_B\_ Data\_Error | 29 |
| CB71 | 0x1b | 5 | I7 overflow | 30 |
| CB15I | 0x1c | 6 | I15 overflow | 31 |
| FIXI | 0x1e | 7 | ALU | 32 |
| Multiplier | 33 |
| FLTOI | 0x1f | 8 | ALU | 34 |
| Multiplier | 35 |
| FLTUI | 0x20 | 9 | ALU | 36 |
| Multiplier | 37 |
| FLTII | 0x21 | 10 | ALU | 38 |
| Multiplier | 39 |

对于FIXI/FLTUI/FLTOI/FLTII四个异常，在ISR函数中需要将STKYx寄存器中相应标志位清零，系统不会自动清零。通过查询STKYx相应标志位确定异常原因，将异常原因编号写进Flash，同时将相应标志位清零，代码如下：



上面保存异常通用现场信息，对于SOVFI异常(其他几种异常额外需保存的信息暂时还未写)，还需要保存硬件栈中所有内容，这里与切换中上下文切换保存将硬件栈内容压栈功能类似，但是需要注意的是：当发生SOVFI异常后，具体造成SOVFI异常(PC stack full/ Status stack overflow/ Loop stack overflow)某个栈溢出/满后，再对该栈执行pop指令时，可以出栈但是STKYx寄存器相应标志位不会自动清零，导致无法直接使用上下文切换中硬件栈内容压栈部分代码，需要将保存硬件栈分两种情形：1.非造成SOVFI异常原因；2是造成SOVFI异常原因。当是造成SOVFI异常原因时则跳转到.save\_PC\_exp执行，否则可按正常情形保存，执行.save\_pc\_start\_exp，如图 27 SOVFI异常硬件栈保存部分源码所示。Status stack/Loop stack方法同上。注意：硬件栈保存的顺序如图 26 SOVFI异常中硬件栈内容保存顺序图：

|  |
| --- |
|  |

图 29 SOVFI异常中硬件栈内容保存顺序图

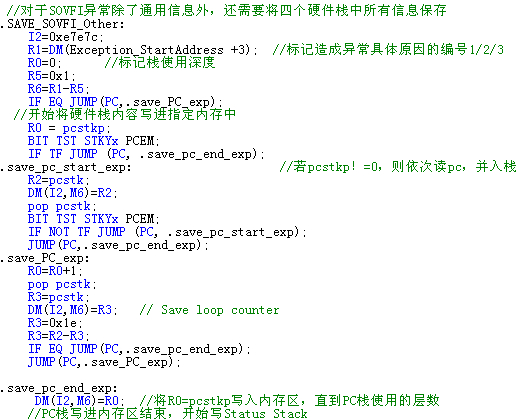


图 30 SOVFI异常硬件栈保存部分源码

## 5.3 复位功能实现原理

ADSP21469有三种复位方式(Emulation Reset这里暂不讨论)，分别是Hardware Reset、Software Reset、Running Reset。这三种方式的区别如表格 17 ADSP21469 Processor Reset方式比较表所示：

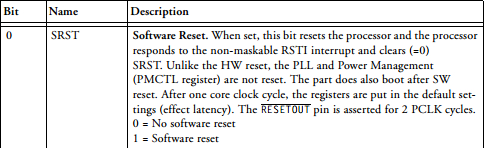
表格 19 ADSP21469 Processor Reset方式比较表

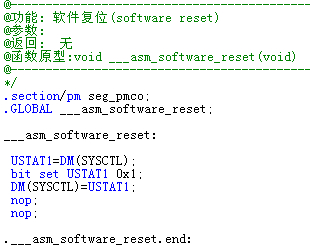
|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\HCY9[$@]B)0O~D5AL7(UV`Y.jpg |

三种复位方式实现

Hardware Reset：毫无疑问，按下复位按键或者通过将某个I/O pin与RESETOUT(ADSP21469复位引脚)引脚相连，通过软件将指定I/O pin拉低电平即可实现软件硬复位。

Software Reset：通过设置**SYSCTL**寄存器Bit0位即可实现，**SYSCTL** Bit0(*SRST*)





Running Reset:通过设置RUNRSTCTL寄存器实现，RUNRSTCTL寄存器位定义如下：



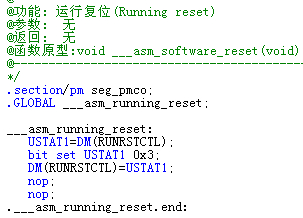
关于Running Reset几个需要注意的事项：

* The core-PLL is NOT reset, and continues to run.
* Internal memory SRAM contents remain unaltered.
* The processor core and peripherals are reset exactly as if a Power-on (hardware) reset is asserted, except:

The SDRAM/DDR2 controllers continue to run and refresh as programmed.

The contents of external SDRAM/DDR2 are unaffected, and retain their values prior to a running reset.

A system boot is NOT initiated. Instead, the program counter is cleared and program execution begins from the very first location of program memory (from the reset interrupt vector table).



## 5.4 查看异常信息

### 查看异常信息表：

使用格式：g\_exp\_info

会告诉你最后一条异常简要信息，包括异常使用标示符、主异常编号、分支异常编号、异常现场信息存储长度、记录异常发生次数、异常存储位置sector、异常存储位置相对偏移位置、异常存储结束地址、发生异常事件ID、发生异常线程ID及异常发生时间。具体定义可见exception.c中struct sys\_exp\_record\_item定义。

### 查看异常信息详情

使用格式：s\_exp\_info 0x3(第三次异常信息)

会告诉你指定异常的详细信息。数据结构详情参见exception.c文件。

### 清除异常信息

如果异常数太多了，你可以清除所有的异常记录。

使用格式：exp\_info\_clear

# SHARC中ASM与C语言混合编程

## 源码分析

以下讨论都是以不使用硬件PC栈为目的的，也就是不使用CALL和RTS指令。

如果调用方与被调用方都是C函数，则这些操作都是由后台完成的。若有一方为汇编，则汇编部分需手动完成。下面以汇编调用C函数为例说明。

若用汇编调用C函数。首先要做两件事：

R2 = I6; // I6用于保存上一次调用时的I7值，因此用R2暂存这个值

I6 = I7; //将当前的I7值赋给I6，为跳转做准备

完成这两步之后，可以JUMP至C函数了。注意，是JUMP，而不是CALL。如下：

JUMP(M5, pm\_ptr) (DB); // M5=0，pm\_ptr是C函数名

DM(I7,M7)=R2; //R2存入当前栈，即将I6的上一次的值存入当前栈

DM(I7,M7)=PC; // 将当前的PC存入栈，用于从C函数返回

至此，就可以进入C函数了。对于用C编写的函数，在进入函数后，编译器会做两件事：

I7 = modify(I7,-4); //这里将I7指针下移，是因为这里的4个位用于存储传递参数，即R4、R8、R12等

dm(-2，I6)=R4; //这里依次将传递参数取回。以下的忽略

完成C函数的调用，从C函数返回调用前函数位置时，编译器会做这几件事：

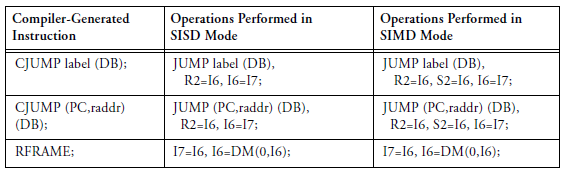
I12 = dm(M7,I6); //这里，I6存储的是调用前的I7值，因此本句正是用于取回调用前所存储的PC值，并赋给I12

JUMP(M14,I12)(DB); //M14=1，I12是调用前所在PC，这里是跳转至PC的下一行

rframe; //通过查阅编程手册PPR可知，rframe可以分解为两行语句：I7 = I6 , I6 = DM(0,I6);这里，首先将调用前所保存的I7值恢复，再从栈中恢复I6值。与上文调用前程序段中的I6 = I7和DM(I7,M7)=R2相对应

nop； //延迟，至此返回调用前的程序处了

下表有所借鉴意义。



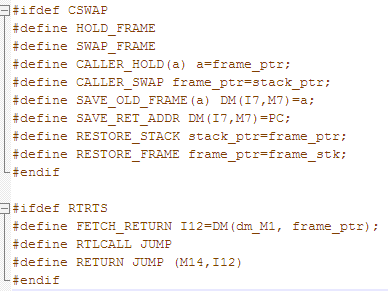
上文所述的流程是在汇编中调用C函数，因为C函数部分的流程是编译器偷偷摸摸在后面为我们做的，是我们无法掌控的，因此我们只能按照C的流程去操作，才能保证函数的调用能够正常执行和返回。但当我们在汇编中调用汇编函数的时候，则可以无需这么麻烦。

如果不需要变换栈指针的位置，而只需要实现调用函数和正常返回的话，我们可以在汇编中使用CALL、RTS这一套指令。使用CALL指令会，硬件会自动压PC栈，而调用RTS指令，硬件会弹出最后一次保存的PC值，并跳转到指定位置。

分析本部分流程的意义在于，因为CALL会自动压PC栈，而在C语言中实现函数调用的时候，VDSP是不会去使用PC栈的，而是存在软件栈中。这点从上文的分析就可以得出了。因此，在进行C和汇编函数之间的调用的时候，要特别注意这个问题，汇编部分的代码必须按照C语言函数调用的格式进行完善，以免函数不能正常返回。因为设计到线程的切换，我们要尽量避免使用PC栈，因此在汇编调用汇编的时候，也尽量不使用CALL，如果使用，要确保PC STK不会压入新值，以免函数不能够正常返回。在使用JUMP指令的时候，是不会出现上述问题的，但会导致函数不能正常返回。

这里顺便提一下RTS和RTI的区别。RTI是指从中断中返回，RTS只会弹出PC值并跳转，RTI除了会做这个工作外，还会做两件事：1、清IRPTL和IMASKP，即清楚中断标志位；2、如果是TIMER或IRQ中断，因为硬件会在进入中断时自动push sts栈，因此RTI会根据需要决定是否pop sts栈。

合理利用ADI在lib\_glob.h中的一组定义，可以更为直观方便的实现函数的互相调用和返回。



## 汇编调用C

调用示例：

pm\_ptr = C\_PROG\_NAME; // Jump to where

CALLER\_HOLD(R2) // R2=I6;

CALLER\_SWAP // I6=I7;

RTLCALL(pm\_0, pm\_ptr) (DB); // JUMP(0,I12)(DB);

SAVE\_OLD\_FRAME(R2) // put(R2);

SAVE\_RET\_ADDR // put(PC);

## 汇编返回C

调用示例：

FETCH\_RETURN // I12=DM(-1,I6);取PC

RETURN (DB); // JUMP (M14,I12)（DB）;

rframe; // I7=I6,I6=DM(0,I6);恢复I7,I6

nop; // nop;

## SHARC参数传递规则

程序通过寄存器和堆栈来传递函数参数，其规则为：

1.最多允许3个参数通过寄存器传递，并且左起三个参数按照次序传递到R4，R8，R12；

2.如果有一个参数传到堆栈，其右边的所有参数都应传到堆栈；

3.字宽大于32位的复数或双精度型数据将传递到堆栈，且占用两个连续的地址单元，次序为实部在低地址、虚部在高地址，或者高32位在低地址、低32位在高地址；

函数参数个数可变时，最后一个有名参数进入堆栈。

例如，C程序为：

foo(int a, float b, char c,float d);

其前三个参数传递到R4，R8，R12，参数d进入堆栈。

bar(int a, double b，char c，float d)；

第一个参数a放在R4，其余入栈。

test（float a, int b，char c，……）；

参数个数可变，则第一个参数a放在R4，第2个参数b放在R8，最后一个有名参数c进入堆栈。

4.当函数参数多于三个时，从第四个参数开始用stack传递。

汇编函数返回时的参数传递：被调用的汇编函数在最后必须完成以下工作：

1. 把返回值写入寄存器R0、R1。返回值为单字时放在R0中；返回值为双精度字时，高32位放在R0，低32位放在R1；返回值为复数时，实部放在R0，虚部放在R1；返回值是大于二个字的结构型时，R1指向结构地址。参数传递表如表格 1所示。

表格 1 ADSP21469混合编程参数传递表

|  |
| --- |
|  |
|  |

# Monitor与Release版本工程

## Monitor与Release版本说明

在实际工程应用中，我们需要在片外存储器中存储两个版本的程序供CPU加载，一个是Monitor版本，一个是Release版本。

Release版本的说明：包含了精简的djyos，及完整的应用程序。Release版本在片内和片外RAM空间中运行，执行效率要求高的代码尽量保证在片内RAM空间运行。其初始化栈、中断栈、系统栈等都位于片内RAM中。Release版本的加载方式为片外FLASH引导，与普通的加载方式无异，CPU上电或复位时，Boot\_Kernel自动默认引导加载Release版本的程序。Release版本作为正常运行时的版本，所有的应用程序都在该版本中运行，同时删减了djyos中无关的部分模块，这些模块在Monitor版本中或有使用。

Monitor版本的说明：包含了完整的djyos程序，如shell等，不包含应用程序。Monitor版本在片外RAM中独占一块区域运行程序，尽量保证不与Release版本的程序在RAM中有交集，在片内RAM中，因为seg\_rth的唯一存在，两个版本的程序共用该段，除此之外，Monitor版本的其余代码全在片外DDR2的独立空间中运行，包括code与stak、heap等。Monitor版本的加载方式为片外FLASH引导，但与Release版本不同的是，CPU在上电或复位时，需要按下某指定的按键以保证Boot\_Kernel引导加载Monitor版本的程序。Monitor版本的主要功能是用于应用程序的辅助调试，在Release版本出现故障时，能通过reset cpu进入Monitor版本执行，这样可以保证Release版本程序的运行环境不被破坏，从而可以在Monitor版本中通过特殊的指令检查出现故障时的RAM运行空间、栈、异常记录等状态。

Release版本和Monitor版本的RAM空间分配如图4-1所示。

Release版本和Monitor版本的FLASH存储空间分配如图4-2所示。

## 程序在DDR2中运行

Djyos的移植一直是以Release版本为基础进行的，Monitor版本的建立有两条途径，第一，建立一个新工程，将Release版本照搬过来，修改LDF链接脚本中的相关内容，使程序运行在片外DDR2中；第二，在原有Release版本工程的基础上，通过增加宏（Monitor\_version、Release\_version）来条件编译LDF链接脚本文件，同时将不需要的源码exclude出工程等方式迅速的重建一个Monitor版本，两者共存于一个工程，大部分编译器都支持这样做，类似于Debug版本与Release版本。我们采用的是第二种方式。

在前面的移植过程中，程序一直是运行在片内RAM空间中的。其所有程序代码、变量、常量等数据也都存储在片内RAM空间中。而Monitor版本程序的程序代码、变量等都在片外DDR2内存中，这会导致Monitor版本程序的执行效率远远低于Release版本。要实现程序全部运行于DDR2中，我们需要在链接文件LDF中将Monitor版本的程序全部指定到片外。

首先，对于LDF中的memory段，我们需要在片外DDR2中单独规划一块独立的空间出来供Monitor使用，这块空间的起始地址如图所示。

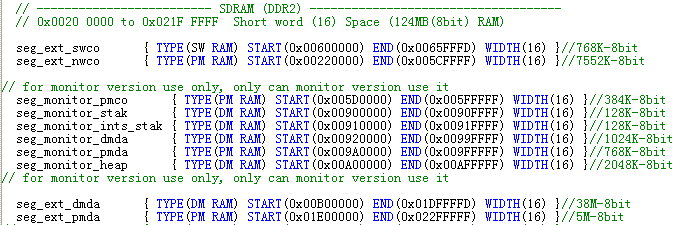


图4-3 DDR2内存空间规划

其中，seg\_monitor\_pmco是用于Monitor程序运行的空间，其位置紧邻Release的程序运行空间，其后依次为启动栈、中断栈、dmda、pmda和堆空间。注意，片内RAM的开始处定义了两个较为特殊的段：seg\_rth和seg\_ldr。seg\_rth段为IVT存储区，其位置只能是在片内，因此这段代码仍然保留在片内执行。seg\_ldr段为二次加载代码的存储区，也是实现Monitor与Release版本切换的关键代码部分，其运行位置理论上来说也可以是在DDR2中，前提是DDR2已经正常初始化，目前暂时将其放在片内，今后可尝试搬到片外。

其次，对于LDF中的section段，我们需要将.obj文件中包含的段指定到特定位置，对于Release而言，只需要在默认LDF文件的内容上稍作修改即可。对于Monitor版本而言，则改动较大，因为默认的程序运行空间都是优先选择片内的。

在这里，我们在工程设置的Link->LDF Preprocessing中定义了一组宏：Monitor\_version和Release\_version，这样，我们就可以在LDF中使用这组宏来控制不同版本程序的section段的预编译了。图4-4所示为Monitor版本下的工程设置示例，Release版本下的设置类同。

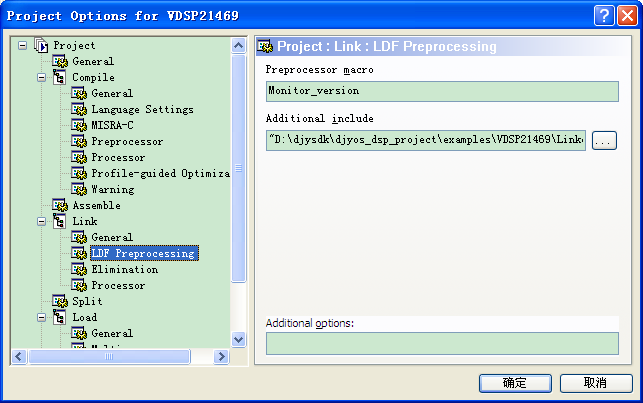


图4-4 Monitor版本下的工程设置示例

对于Monitor版本的section段代码，我们需要保证如下两点原则：1、程序和数据应该尽最大可能放到片外DDR2中运行，而不应该考虑运行效率的问题；2、对于特定的必须在片内执行的少数代码，可以在相应代码或数据段处指定其运行或存储空间为片内，使用seg\_int\_code符号来指定。图4-5所示为原本的片内程序运行空间在Monitor版本下的section中的指定。dmda、pmda空间的指定方式与之类同，不再赘述。

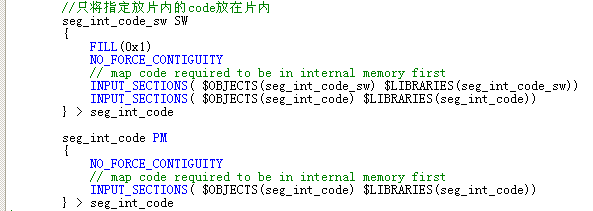


图4-5 Monitor版本LDF中片内空间的指定

相应的，片外DDR2中的空间，我们使用图4-6所示方式来指定。

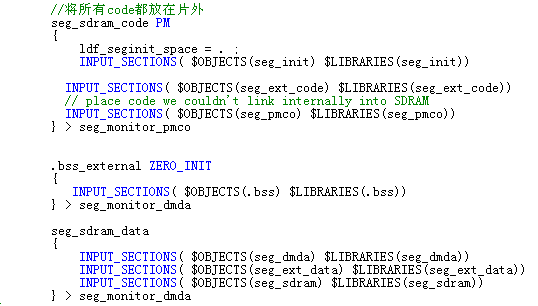


图4-6 Monitor版本LDF中片外空间的指定

其基本原理是：对于指定运行位置为seg\_int\_code的代码，将其放在片内空间运行，其余所有的程序段，无论指定为片内优先的，还是片外优先的，统统放在片外。需要说明的是，VDSP对所有.c及.asm的默认设置都是优先在片内的，我们也可以通过手工的方式来指定，参展图4-7所示。如果工程文件很多的话，这样做的工作量将会非常巨大。当然，因为我们已经修改了LDF，所以在我们的工程中是不需要这样做的。



图4-7 指定当前文件的优先存储（运行）位置的方法

## 二次加载的原理与实现

要实现Monitor和Release两套程序的切换，有两种可选方案：第一，在boot\_kernel中进行引导方式的选择，也就是通过标志位来判断boot\_kernel是需要引导Monitor版本的程序还是Release版本的程序，这样做的缺点是需要按下reset按钮，或需要CPU硬件支持软复位；第二，通过二次加载的方式，也就是将boot\_kernel的引导加载部分代码提取出来，以应用程序的方式放在Release版本的程序中供调用，这样就可以实现Release版本的程序在特定的条件下自动或手动的切换到Monitor版本运行，且CPU一直是处于正常运行状态，不会掉电，也不需要重新初始化DDR2和PLL模块。

综合考虑之后我们选择是第二种方式，也就是二次加载的方式来实现。二次加载的优势在于：1、CPU一直处于运行状态，这样可以避免硬复位或软复位造成的寄存器、RAM空间数据有可能被破坏的危险性；2、不需要重新初始化DDR2和PLL，可以保证DDR2中运行数据不被破坏；3、加载的方式更加灵活，应用更加方便。二次加载的缺点在于：1、因为boot\_kernel引导的特性要求中断向量表中的可编程中断P9I要被占用，具体原因将在后续说明；2、需要在Release版本的程序中添加二次加载的引导程序，这样就会增加系统出错的风险，如非预期的切换到Monitor版本。

对于二次加载存在的这两点缺陷，其中缺陷1可以被完全的避免，缺点2不能完全避免，但可以通过增加条件判断的方式尽可能的降低其出错的概率。首先分析一下缺陷1及其解决方法。缺陷1的产生原因是，可编程中断P9I默认为DMA中断所对应的中断号，而在加载的过程中，程序由片外FLASH向片内RAM或片外DDR2的搬运过程都是通过DMA模块来完成的，boot\_kernel中该部分的设计思路是：在启动DMA搬运指令后，将CPU至于休眠状态（IDLE），然后通过DMA完成搬运后产生一个DMA中断来唤醒CPU。这样就不可避免的需要保证在加载过程中，中断向量P9I所在的位置需要被放置一个rti指令，来引导CPU从休眠模式中退出。在默认的boot\_kernel中，程序在P9I所在的内存位置放置了一个rti指令，在完成加载的最后阶段，也就是final\_init阶段，再将用户程序中的IVT写到seg\_rth段，覆盖掉所有的boot\_kernel代码，包括P9I位置。而我们要实现二次加载，我们不可能保证Release版本应用程序的IVT在P9I处放置的是rti指令，因此只能在开始二次加载之前，通过软件将P9I处改写，在此地址处添加一个rti指令，以保证二次加载程序的正常执行。而在二次加载完成之后，seg\_rth段又会被Monitor版本的IVT所覆盖，所以不必担心P9I处的rti指令会影响加载之后程序的执行。图4-8所示为二次加载之前部分的代码段，因为二次加载的触发条件为IRQ1中断，因此在此处我们还需要屏蔽掉所有的实时中断和异步信号，并清除中断响应状态。该代码段同时即为IRQ1中断的中断服务函数。

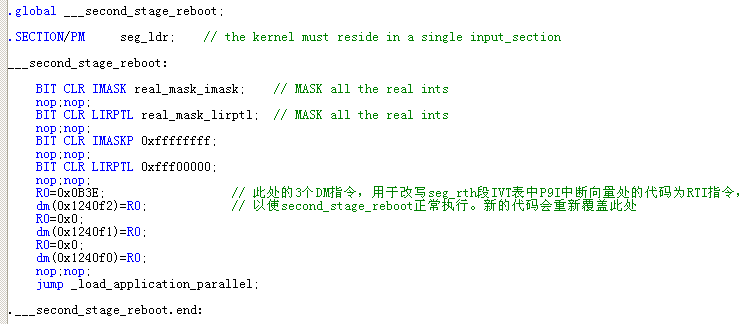


图4-8 二次加载的预处理

再来说说缺陷2的处理方法。要防止缺陷2的产生，我们需要尽可能的避免二次加载程序被错误的触发，因此我们可以通过增加条件判断的方式来尽最大可能的避免程序的错误执行。具体方式为：1、在二次加载被触发后（IRQ1中断被触发），判断某一确定标志位（FLAG）的状态，该标志位的状态由硬件跳线来决定，如果不满足设定条件，则退出二次加载；2、在进入二次加载的流程之后，在开始启动DMA搬运代码至片内RAM或片外DDR2之初，再次判断标志位，如果不满足设定条件，则退出二次加载。通过这样的方式，我们虽然不能百分百的避免由二次加载而导致的错误的产生，但可以在最大的程序上避免这种情况的发生，使其发生的概率降到极低。

前面说过，二次加载部分的代码是参照boot\_kernel的加载引导部分来完成的，其原理也与boot\_kernel的原理比较类似，但也有不同：1、不需要初始化DDR2和PLL，因为CPU上电运行时boot\_kernel已经干过这份工作了；2、不需要再顾忌指令长度的限制，因为二次加载代码已经是一个比较特殊的应用程序了，因此不再受限于CPU硬件设计所决定的boot\_kernel不能超过256指令长度的限制；3、引导程序加载的起始位置与boot\_kernel也有不同，应该指向Monitor版本程序所在的位置。因此，我们就以VDSP默认使用的469\_prom这个boot\_kernel的源码为蓝本来设计二次加载的程序。

两个版本程序的组成与在FLASH中的位置分配的示意图见图4-9所示，其中Release版本使用的是自定义的boot\_kernel，其内容与执行过程与默认的boot\_kernel基本一致，仅优化了其PLL初始化过程，并根据工程要求重新定义了时钟频率。其中，Monitor版本所包含的默认的boot\_kernel在实际执行时并不运行，保留其仅为调试方便。



图4-9 程序的组成及在FLASH中的位置分配示意图

关于引导程序加载的起始位置的计算，根据如下公式计算：

**EIEP register = reset value + start addr + default boot kernel offset**

EIEP register就是供引导程序加载程序时的起始位置的，其初始值reset value= 0x4000000；在实际工程中，Monitor版本程序（包含default boot kernel）在FLASH中的存储起始位置为0x40000，注意这里的0x40000是根据8bit进行计算的，转换成EIEP所需的32bit，故需要将0x40000右移2位，即0x10000，因此start addr=0x10000；default boot kernel为256指令长度，为48bit，因此转换为32bit后，其长度为256\*48/32=0x180，因此default boot kernel offset=0x180；根据上述分析，我们可以计算得出EIEP register = 0x4000000 + 0x10000 + 0x180 = 0x4010180。

在计算出正确的引导程序加载的起始位置之后，我们需要在二次加载程序开始加载之前将EIEP指定为0x4010180，这个步骤必须在开始搬运之前完成，因此，我们在加载程序的USER\_INIT段中完成上述工作，除了需要设置EIEP之外，还有关于DDR2和AMI的两个寄存器需要被设置。程序段参考图4-10所示。

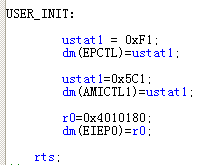
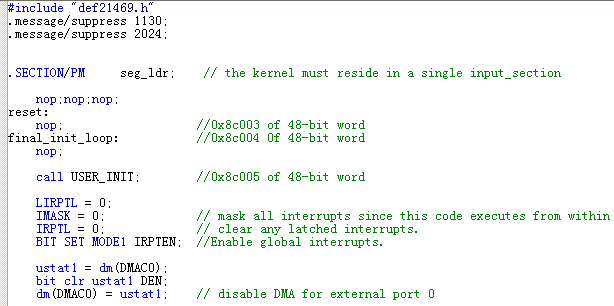


图4-10 二次加载程序开始加载之前的初始化

除了引导程序加载的起始位置需要被重新指定外，还有一个地方是必须要修改的，那就是PLL及DDR2的初始化部分。默认的boot kernel中，在执行完USER\_INIT段之后，会进行PLL和DDR2的初始化，因为我们的程序是在CPU上电后，由默认的boot kernel引导加载的，因此PLL及DDR2都已经是处于初始化正常的状态，所以我们在二次加载的时候，并不需要再重新初始化，这一点是必须注意的，因为一旦在二次加载的过程中重新初始化了PLL或DDR2，都会导致DDR2中的数据现场被破坏。因此，我们将USER\_INIT之后的InitPLL和InitDDR2两段代码删除，并在USER\_INIT的末尾加上一个rts指令指导程序返回被调用位置。

除了上述修改外，在程序的开头部分也需要稍作修改以适应二次加载的需要。图4-11为默认的boot kernel中起始部分的程序段，图4-12为二次加载代码起始部分的程序段。



4-11 默认的boot kernel中起始部分的程序段

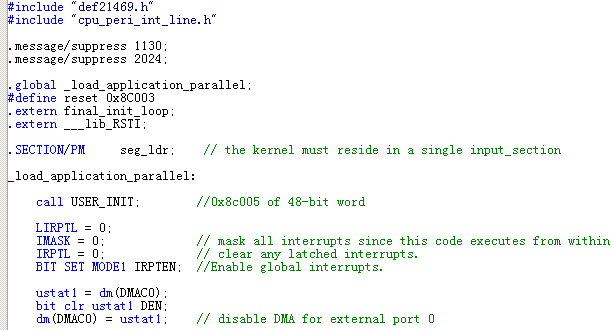


图4-12 二次加载代码起始部分的程序段

其中，需要说明的是，这两者所处的seg\_ldr段是完全不一样的两个概念。默认的boot kernel是一个完整的工程，包含源码、LDF文件等，最终编译生成一个dxe可执行文件供其他的用户程序来调用，最终共同生成ldr可加载文件写入FLASH中，因此，在默认的boot kernel中的seg\_ldr所表示的是boot\_kernel工程自己的LDF文件所定义的一个片内RAM段，这个段在boot kernel的工作完成之后就不再存在了，而在我们的二次加载的程序中，因为二次加载的程序本质上来说是一个应用程序，包含在Release和Monitor版本中，因此他们是和工程共享LDF文件的，因此，这里的seg\_ldr段需要在工程的LDF文件中，在seg\_rth段之后开辟一个约256\*48bit的空间，定义为seg\_ldr供二次加载程序使用。当然，原理上来说，这个空间是可以放在片外DDR2中的，但其具体实现尚未验证。

还有一个需要注意的地方是，reset所代表的0x8C003地址为复位向量表中，CPU复位向量所在的实际物理地址，而\_\_\_lib\_RSTI是用户IVT程序中，复位向量处所定义的一个符号，他们的作用在于二次加载完成之后，需要通过这个符号来引导程序跳转至复位向量处开始执行新加载的Monitor版本程序。所以在Monitor版本的应用工程中，相应的复位中断向量处需要将符号定义为对应的\_\_\_lib\_RSTI。

在完成上述操作后，二次加载的功能就可以实现了。最后，我们在烧写程序到FLASH中时，还需要注意两点：1、在烧写时注意，不可选择在烧写前擦除全片FLASH，而只能擦除需要写入的部分，以防其余版本的程序被破坏；2、Release版本程序的烧写跟正常烧写无异，Monitor版本的程序需要在工程设置中的Load->General下，将Format改为ASCII或Binary，这样我们才可以在写入FLASH时，人为指定其起始位置为0x40000。请参见图4-13及图4-14所示。

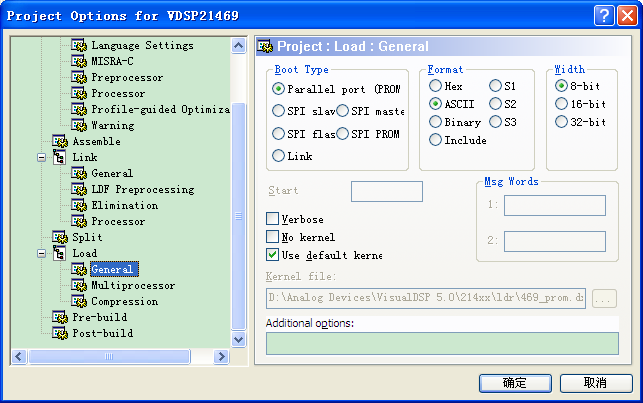


图4-13 Monitor版本的Load选项设置

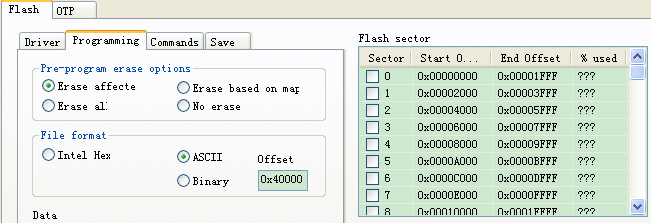


图4-14 Monitor版本的FLASH烧写设置

至此，二次加载的过程就已经实现了。实际上，在实际运行中，我们通过触发IRQ1中断启动二次加载，完成Release版本程序向Monitor版本程序的切换。此时，在Monitor版本程序正常运行的过程中，我们仍然可以通过触发IRQ1中断来实现Monitor版本程序的重新加载，也就是软件复位。

因此，从本质上来说，二次加载的过程与软件复位的实现过程是基本一致的，如果需要在工程中实现软件复位，只需要在二次加载程序的USER\_INIT段中添加控制判断条件，来使二次加载程序决定是从Release版本程序或Monitor版本程序的起始位置处开始加载，从而实现相应版本程序的软件复位过程。SHARC21469本身提供软件复位的功能，其控制位在SYSCTL系统控制寄存器中，但这个软件复位会重新启动booting的过程，因此，这会导致程序只能从默认的Release版本处启动，所以这个软件复位与硬件复位的功能基本是一致的。而通过二次加载的方式启动的软件复位则不会初始化所有寄存器值，不会破坏片内片外RAM空间，而且可以指定需要启动的软件版本，可以是Release，也可以是Monitor，其具有更大的灵活性。

# 简易文件系统

## NorFlash驱动程序设计（demon板）

# 驱动程序设计

本节所讨论的模块都属于SHARC21469的片内外设，其源码都位于src\bsp\cpu\_peripheral\src目录下，其头文件都位于src\bsp\cpu\_peripheral\include目录下。本节所讨论的驱动程序的设计仅只包含对应模块硬件相关部分的驱动设计，模块具体功能的实现同时还要依赖于djyos\driver中的相关模块。驱动程序的设计属于硬件相关的部分，属于底层模块之一，对外设寄存器的直接操作都在该部分完成，而在djyos源码中只调用相关的接口函数。

## 串口驱动程序

SHARC21469包含一个通用的UART模块。UART的基本功能包含接收和发送数据，属于半双工通讯方式。UART模块的使用有如下几个要点，下面将结合具体的驱动程序一一说明。

1. 关键寄存器说明
2. TX、RX引脚的设置
3. 波特率设置
4. 发送和接收中断的设置

# ADSP21469 CPU概述

## CPU架构

图 1 ADSP21469架构图所示，由CORE PROCESSOR，4 BLOCKS OF ON-CHIP MEMORY，EXTERNAL PORT及I/O PROCESSOR四部分组成。

|  |
| --- |
|  |

图 1 ADSP21469架构图

**CORE PROCESSOR**

其中内核包含1个程序序列发生器、2个计算处理单元(PEx、PEy)、1个内核时钟(Timer)，1个32\*48bit的指令Cache、2个数据地址产生器(Data Address Generators, **DAG**)。图 2计算单元结构示意图所示：

|  |  |
| --- | --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\UASCU5{7Y}9SDU5(6MCNK5R.jpg | C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\ZZ%E(K1UJ(ZV33ITTGH(KJW.jpg |

图 2计算单元结构示意图

单个计算处理单元包含1个数据寄存器文件夹、1个算法/逻辑单元(ALU)、1个乘法器(Multiplier)及1个移位器(Shifter)。16个40bit通用寄存器(r0~r15/s0~s15定点；f0~f15/sf0~sf15浮点)、溢出标志位寄存器(ASTATx/ASTATy)(STYKx/STYKy)。

Ps:①由于有两套计算处理单元，1个cycle可以做2套乘累加。

② ALU、Multiplier、Shifter在同一个cycle只能有两个在同时使用。

ADSP21469片内有5M bit RAM，4Mbit ROM(对用户不可见，可视为不存在)，片外采用的是DDR2、Flash等。片内默认以32bit存储，最小为16bit，片外全部为32bit。

|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\[`NF%FI$_V1[VH~8J}QL_UD.jpg |

图 3ADSP21469内存映射图

**SHARC internal memory supports several data types**

Short words, 16 bit (1 column )

Normal words, 32 bit (2 columns )

Long words (used as two 32 words), 64 bit (4 columns )

40 bit data words, (3 columns )

48 bit words, instructions ( 3 columns )

ADSP21469内存可以以16bit、32bit、48bit、64bit形式存储，其中默认为32bit，CPU通过读取存储地址判断其为何为存储模式，如图4所示，同一个内存空间，16bit的存储地址为32bit地址2倍，及16bit的存储地址可通过同一内存空间32bit存储地址左移一位得到，64bit的存储地址可通过32bit地址右移一位得到。

片内SRAM分为4个Block，其中Block0、Block1为1.5Mbit(192k byte)，Block2、Block3为1.0Mbit(128k byte)，各个Block地址连续，但是有些Reserved地址使各个Block可使用地址不连续。单个Block可设置成单种存储格式，也可设置成多种存储格式，但是从低地址到高地址依次为48bit、32bit、16bit(大bit在低地址，小bit在高地址位)。Block0默认存放代码。

内存访问示意图如图4所示，每个Block有一个总线对外，而数据总线DM BUS、程序总线PM BUS、IO总线IO BUS，由图4可知，当同一cycle，若数据总线DM BUS、程序总线PM BUS、IO总线IO BUS同时访问同一个Block，必然会发生冲突，需要等待，但是若将数据和程序放在不同的Block，则在同一个cycle可同时执行2条指令。

|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\252LVKP$X30NCFJ[OT_KFVX.jpg |

图4 ADSP21469存储器访问示意图

DSP core通过总线方式访问Internal Memory如图5 ADSP21469访问Internal Memory示意图所示。

|  |
| --- |
| C:\Documents and Settings\Administrator\桌面\QQ图片20131017141644.jpg |

图5 ADSP21469访问Internal Memory示意图

## SHARC SIMD 功能

**SIMD:Single Instruction Multiple Data**，指同一条指令同时在PEx/PEy两个处理单元执行，例如在SIMD模式下，执行指令F0=F1+F2，在PEx中会显式执行，而如此同时在PEy中会自动隐式执行S0=S1+S2指令。

使能SIMD模式：系统默认使用的是**SISD**(Single Instruction Single Data, **SISD**)，要想使能SIMD模式，可通过对MODE1寄存器的BIT 21(*PEYEN*)置1，命令如下：

**bit set mode1 PEYEN; // Enable SIMD (Both PEx and PEy enabled)**

**bit clr mode1 PEYEN; // Disbale SIMD (PEx enabled only)**

下面主要阐述**SIMD**模式与**SISD**模式在Register Transfers and Swaps和Memory Access两方面不同特性。

**Register Transfers and Swaps:**

|  |  |
| --- | --- |
|  |  |

**Memory Access:**

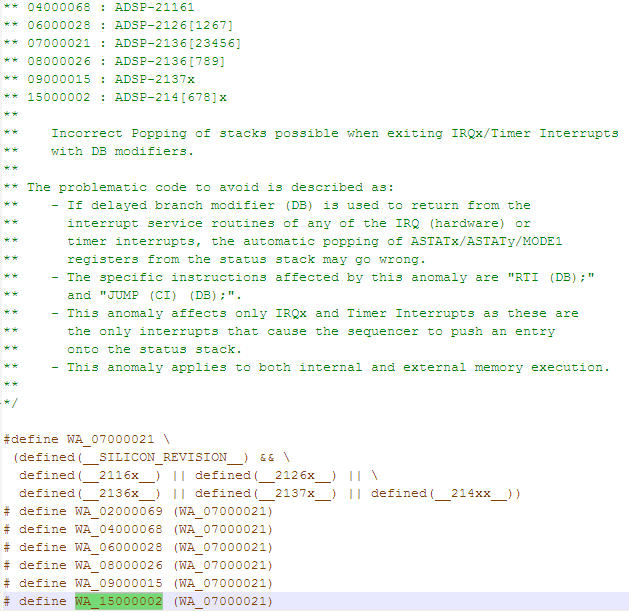
|  |
| --- |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\_Q]}_NWN5T~JD6{(SN0L`DS.jpg |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\}UD9XV`M9CY1I50(G9LZS9O.jpg |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\UGX7)IU53}Z2W7{KKZCM633.jpg |
| C:\Documents and Settings\Administrator\Application Data\Tencent\Users\877563337\QQ\WinTemp\RichOle\1C@YG0)H4)LL2H4ATX{AE]G.jpg |

# 附录：

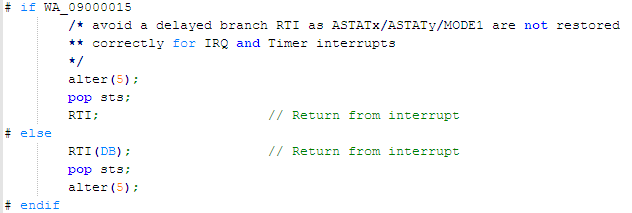
# 一.几类CPU引发的BUG及其处理方式

本章类容是基于ADI所提供的anomaly\_macros\_rtl.h文件中的说明。这几类BUG涵盖了RTI指令、IOP及寄存器操作等，容易在某些特殊情况下产生不可预计的程序执行结果，因此必须加以避免。在实际编程过程中，应该注意这几个容易引发BUG的地方，因为他们的产生原因并非是代码本身的问题造成的，而是会导致看似正确的代码执行得到预期之外的结果。

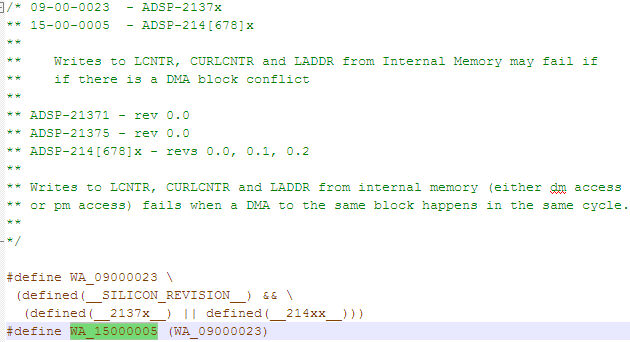
## WA\_09000015（RTI）



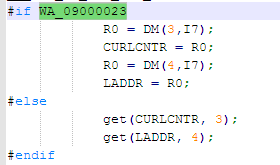
用法举例



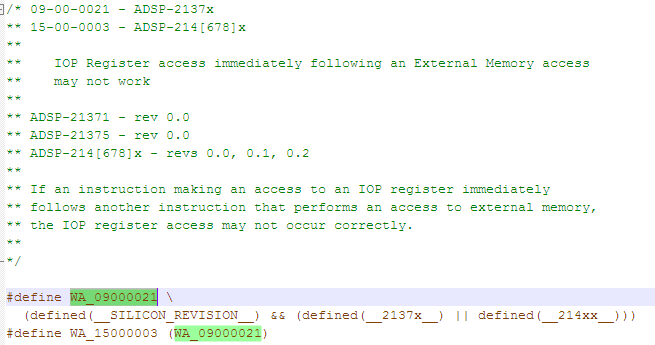
## WA\_09000023（LCNTR等）



用法举例：

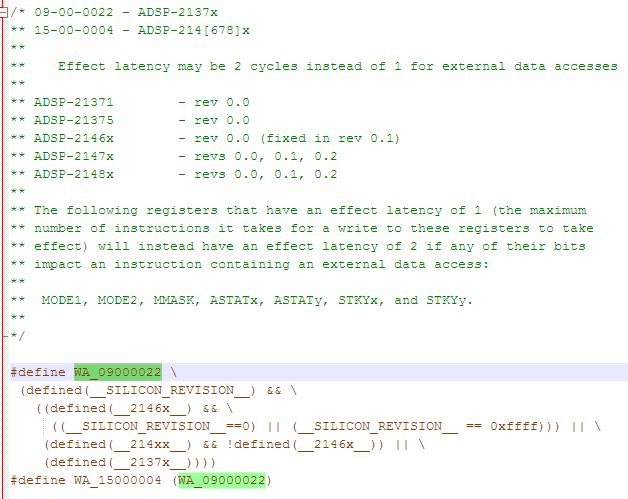


## WA\_09000021（IOP）

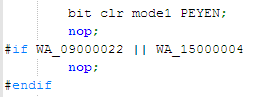


使用较少。

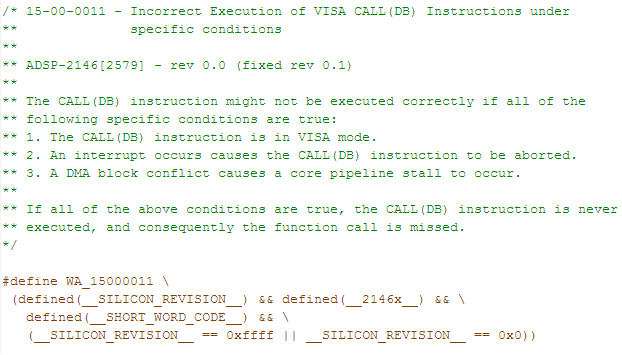
## WA\_09000022（MODE1等）



用法举例

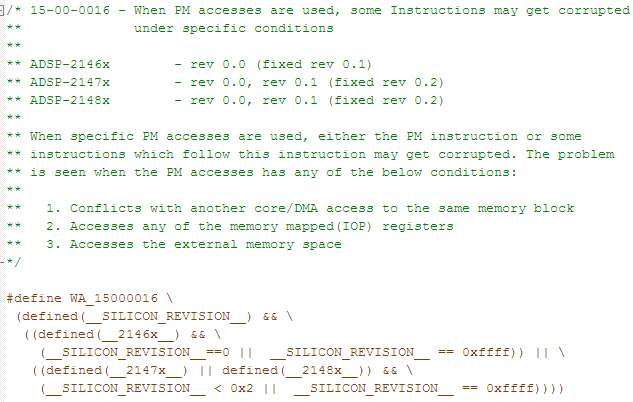


## WA\_15000011（CALL）



使用较少。

## WA\_15000016（PM）



使用较少。