使用 CCS 进行 DSP 编程（三）

——实现 DMA 和 Interrupt

pacificxu

现在讨论在 CCS 进行 DSP 编程来实现 **DMA** 和 **Interrupt** 功能。假定读者对 CCS 的使用已经比较了解，并有了一定的 CCS 编程经验。如果读者还不太了解，请参阅《使用 CCS 进行 DSP 编程（一）——CCS 编程入门》、《使用 CCS 进行 DSP 编程（二）——实现 FFT》及其他 CCS 的学习文档。

下面用闻亭公司的 C6xPa 板硬件和闻亭公司的 PCI 仿真器为例，来实现 DSP 的 **DMA** 传输和硬件 **Interrupt** 功能。

首先来描述一下使用的硬件资源。闻亭公司的 C6xPa 板有两路独立的最高采样率为 40MHz 精度为 12bit 的 A/D，它与 DSP 的 EXT_INT7 相连，可以产生外部中断信号，通过 FPGA 的逻辑可以控制 A/D 的采集和采集多少数据产生一次中断，采集的数据放在 DPRAM 中（0x1400000 开始的地址空间），通过 **DMA** 传输到 DSP 芯片上的存储器中（0x80000000 开始的地址空间）。

在 C 语言环境中使用 **DMA** 和 **Interrupt** 功能，需要包含两个头文件<dma.h>和<intr.h>，同时要用到相应的运行时库文件“ csl6201.lib ”和“ dev6x.lib ”。对这两组头文件和运行时库文件，我们深入研究一下，看一看我们比较关心的函数有哪些。下一次用到这些函数时，别忘了带上相应的运行时库文件%*&^*&^喔。

在 dev6x.lib 库文件中，直接与实现 **DMA** 和 **Interrupt** 功能相关的函数有如下几个：

dma_init

dma_global_init

dma_reset

intr_reset

intr_init

intr_hook

intr_map

intr_isn

intr_get_cpu_intr

isr_jump_table

在csl6201.lib库文件中，直接与实现**DMA**功能相关的函数有如下几个：

DMA_AllocGlobalReg

DMA_GetEventId

DMA_GBL_PRIVATE

DMA_Open

DMA_Start

DMA_HCHA0

DMA_HCHA1

DMA_HCHA2

DMA_HCHA3

DMA_Wait

DMA_SetGlobalReg

DMA_ConfigA

DMA_ConfigB

DMA_Stop

DMA_AutoStart

DMA_Pause

DMA_Reset

DMA_GetGlobalReg

DMA_SetAuxCtl

DMA_Close

DMA_FreeGlobalReg

DMA_Init

DMA_GetStatus

我们只需其中的一部分便可以实现 **DMA** 和 **Interrupt** 功能。函数的具体使用说明和调用方式请参看 TI 的说明档。

下面直接看一下程序的源码，有一个整体的理解，然后再对具体的相关内容做一下说明，如果源码能够理解，就不需要再看具体说明了。为了便于读者理解，

本人对相应部分作了处理，尽量使结构清晰，突出对 **DMA** 和 **Interrupt** 功能的实现：

```
/*============================================================
/*  Copyright (C)1996-2001 Beijing Wintech Technology Co.,Ltd.
/*            By Pacificxu,           All Rights Reserved
/*============================================================

#ifndef CHIP_6201
#define CHIP_6201       1
#endif

/* includes */
#include "intr.h"
/* Interrupts Support - 'C6x peripheral support library*/
#include "dma.h"

/*DMA控制参数说明*/
DMA_CONFIG MyConfig1 = {
                        0x01000051, /* prictl */
                        0x00000000, /* secctl */
                        0x01400000, /* src */
                        0x80008000, /* dst */
                        0x00000400 /* xfrcnt */
                        };
/*函数定义*/

/*初始化中断子程序*/
void Init_int7();


 /*中断服务子程序*/
 void interrupt  Int7_ISR();

 /*调用DMA子程序*/
 void DMA_read_data1();

 /* variables definitions */

 /* DMA操作句柄*/
 static DMA_HANDLE hDma;


 main()
 {
         /*初始化中断*/
      Init_int7();

 /*通过FPGA对A/D采集进行控制，采样率40MHz，采集2k数据产生一次中断*/
      *(int *)0x3380000=0x1f;

         /*等待中断*/
        for(;;) ;

 }
```

```
/*中断服务子程序*/
void interrupt Int7_ISR()
{
        DMA_read_data1();
}

/*调用DMA子程序*/
void DMA_read_data1()
{
        hDma=DMA_Open(DMA_CHAANY,DMA_OPEN_RESET);
        DMA_ConfigA(hDma,&MyConfig1);
        DMA_Wait(hDma);
        DMA_Close(hDma);

}

/*初始化中断子程序*/
void Init_int7()
{
 /* Interrupts settings */
        intr_init();
/* it initializes the ISTP with the address of the global label
// vec_table, which is defined in intr_.asm, and resolved at link
// time. Defined in intr.c as a callable function, intr.c is a
// included with the 'C6x peripheral support library and should be
// compiled and linked with the rest of the program files
*/
        intr_map(CPU_INT7,ISN_EXT_INT7);
/* it places the indicated Interrupt Service Number
// (ISN) value in the appropriate field of the
// appropriate interrupt multiplexer register. Defined in
// intr.c as a callable function
*/
        INTR_CLR_FLAG(CPU_INT7);
/* it manually clears the selected interrupt by writing
// a 1 to the specified bit in the ICR. This is just to
// be sure that there's no unwanted/unexpected
// data in any of the bit fields of this register.
// Defined in intr.h as a macro . Even though
// this is not absolutely necessary, it is highly recommended.
*/
        intr_hook(Int7_ISR,CPU_INT7);

/* it places the function pointer indicated by the first
// parameter (a pointer to an ISR declared in C) into
// isr_jump_table[], at the location specified by the
// second parameter (ISR to invoke when servicing
// this interrupt).
*/
        INTR_ENABLE(CPU_INT_NMI);
/* it enables the non-maskable interrupt (NMI). If this
// interrupt is not enabled, the rest of the interrupts
// won't be seen/processed. Defined in intr.h as a macro.
*/
```

```
        INTR_ENABLE(CPU_INT7);
/* it enables CPU interrupt 14 by enabling its bit in the
// the interrupt enable register (IER).We have previously
// mapped this interrupt number with the cpu clock 0
// interrupt signal. Defined in intr.h as a macro.
*/
        INTR_GLOBAL_ENABLE();
/* it globally enables all maskable interrupts by setting the
// GIE bit in the control status register (CSR). If this bit is
// not enabled/set, the rest of the interrupts won't be seen
// nor processed. Defined in intr.h as a macro
*/
 }
```

接下来进行详细说明。

首先，包含头文件，

```
/* includes */
#include "intr.h"
/* Interrupts Support - 'C6x peripheral support library*/
#include "dma.h"
```

接着对DMA进行初始化赋值，

```
/*DMA控制参数说明*/
DMA_CONFIG MyConfig1 = {
                        0x01000051, /* prictl */
                        0x00000000, /* secctl */
                        0x01400000, /* src */
                        0x80008000, /* dst */
                        0x00000400 /* xfrcnt */
                        };
```

其中，各参数的含义如下：

UINT32 prictl	DMA primary control register value
UINT32 secctl	DMA secondary control register value
UINT32 src	DMA source address register value
UINT32 dst	DMA destination address register value
UINT32 xfrcnt	DMA transfer count register value

我们使用 DMA 的源地址为 0x01400000，目标地址为 0x80008000，传输数据长度为 1k*32bit，DMA 控制寄存器的值的具体含义请参看各对应 DSP 的 datasheet。

然后是对使用的函数进行引用说明，其他函数跟普通的 Visual C++函数没有太大的区别，除了以下的例外：

```
/*中断服务子程序*/
void interrupt  Int7_ISR();
```

其中，有一个关键字“interrupt”，它告诉 TI 的 C 编译器，这个函数是一个特殊

的中断服务函数，C 编译器会另眼看待。

接下来对 DMA 操作句柄定义，相应的头文件对 DMA_HANDLE 作了定义，我们这里可以直接使用：

```
/* DMA操作句柄*/
static DMA_HANDLE hDma;
```

主程序非常简单，

```
main()
{
        /*初始化中断*/
     Init_int7();

/*通过FPGA对A/D采集进行控制，采样率40MHz，采集2k数据产生一次中断*/
     *(int *)0x3380000=0x1f;

        /*等待中断*/
       for(;;) ;

}
```

实现 DMA 功能的子程序也很简单，只有四句：

```
/*调用DMA子程序*/
void DMA_read_data1()
{
        hDma=DMA_Open(DMA_CHAANY,DMA_OPEN_RESET);
        DMA_ConfigA(hDma,&MyConfig1);
        DMA_Wait(hDma);
        DMA_Close(hDma);

}
```

第一句：语法如下：

Function

```
DMA_HANDLE DMA_Open(
  int ChaNum,
  UINT32 Flags
);
```

Arguments

ChaNum	DMA channel to open: • DMA_CHAANY • DMA_CHA0 • DMA_CHA1 • DMA_CHA2 • DMA_CHA3
Flags	Open flags (logical OR of any of the following): • DMA_OPEN_RESET

Return Value

Device Handle	Handle to newly opened device

其中的 DMA_CHAANY 含义是“任意一个闲置的 DMA 通道”，并非有一个 DMA

通道叫 DMA_CHAANY，除了通道 0~3 的 4 个通道外，还有一个辅助 DMA 通道，是另做他用的。我们取得了对 DMA 操作的句柄，就可以进行初始化和使用了。

第二句：语法如下：

Function	`void DMA_ConfigA(`
	`  DMA_HANDLE hDma,`
	`  DMA_CONFIG *Config`
	`);`	

Arguments	`hDma`	Handle to DMA channel. See `DMA_Open()`
	`Config`	Pointer to an initialized configuration structure
Return Value	`None`	

这里需要提前做 `DMA_CONFIG` 的初始化，这两步也可以用一步来实现，就用到另外一个函数：

Function	`void DMA_ConfigB(`
	`  DMA_HANDLE hDma,`
	`  UINT32 prictl,`
	`  UINT32 secctl,`
	`  UINT32 src,`
	`  UINT32 dst,`
	`  UINT32 xfrcnt`
	`);`	

Arguments	`hDma`	Handle to DMA channel. See `DMA_Open()`
	`prictl`	Primary control register value
	`secctl`	Secondary control register value
	`src`	Source address register value
	`dst`	Destination address register value
	`xfrcnt`	Transfer count register value
Return Value	`none`	

它直接把各 DMA 寄存器的设置当作参数，一步到位。

第三句：语法如下：

Function	`void DMA_Wait(`
	`  DMA_HANDLE hDma`
	`);`	

Arguments	`hDma`	Handle to DMA channel. See `DMA_Open()`
Return Value	`none`	

这个函数检测 DMA 的状态位，直到 DMA 结束后才退出，读者可以在下一次使用这个 DMA 通道前使用。DSP 可以执行其他的操作，或者执行此操作等待 DMA 传输结束。

第四句：语法如下：

Function	`void DMA_Close(`
	`  DMA_HANDLE hDma`
	`);`	

Arguments	`hDma`	Handle to DMA channel, see `DMA_Open()`
Return Value	`none`	

使用完 DMA 通道后，需要对它进行关闭，释放资源以备他用。

DMA 的使用是很简单的，复杂的工作都由 DSP 硬件和 TI 的库函数来完成了。我们要做的工作是理解这些，要想使用这些函数，不可避免要知道 DMA 各控制寄存器的具体含义，除了少数“ 天才 ”可以不学而知外，最好老老实实学习 TI 的文档）。#$&^@)(*&#——多么渴望天才的出现啊 !!!

下面来看看中断功能是怎样实现的。在中断服务子程序中，调用了 DMA，完成数据的从外部双口 RAM 到 DSP 片内的传输，

```
/*中断服务子程序*/
void interrupt Int7_ISR()
{
        DMA_read_data1();
}
```

而中断服务子程序是如何与硬件中断联系起来的呢？

中断的实现主要在中断初始化子程序里，在程序中对每一步的操作都进行了详细的说明。我们一步一步来分析一下：

首先是用于中断处理的函数：

Interrupt processing functions include the following:

- intr_get_cpu_intr(isn)
- intr_hook(void(*fp)(void),cpu_intr)
- intr_init()
- intr_isn(cpu_intr)
- intr_map(cpu_intr,isn)
- intr_reset()

The value *fp is a function pointer to the user supplied ISR, cpu_intr refers to the CPU interrupt number, and isn refers to the interrupt selection number that specifies the interrupt source to map to a given CPU interrupt.

然后是用于中断处理的宏定义：

In the following list of interrupt processing macros, bit refers to the bit position on which to operate, val refers to the field value, and sel is 0 for the low interrupt selector register and non-zero for the high interrupt selector register.

- INTR_CHECK_FLAG(bit)
- INTR_CLR_FLAG(bit)
- INTR_DISABLE(bit)
- INTR_ENABLE(bit)
- INTR_EXT_POLARITY(bit,val)
- INTR_GET_ISN(intsel,sel)
- INTR_GLOBAL_DISABLE( )
- INTR_GLOBAL_ENABLE( )
- INTR_MAP_RESET( )
- INTR_SET_FLAG(bit)
- INTR_SET_MAP(intsel,val,sel)

用到的一些助记符如下：

CPU Interrupt Numbers

Mnemonic	Value
CPU_INT_RST	0
CPU_INT_NMI	1
CPU_INT_RSV1	2
CPU_INT_RSV2	3
CPU_INT4	4
CPU_INT5	5
CPU_INT6	6
CPU_INT7	7
CPU_INT8	8
CPU_INT9	9
CPU_INT10	10
CPU_INT11	11
CPU_INT12	12
CPU_INT13	13
CPU_INT14	14
CPU_INT15	15

Interrupt Selection Numbers

Mnemonic	Value
ISN_DSPINT	0
ISN_TINT0	1
ISN_TINT1	2
ISN_SD_INT	3
ISN_EXT_INT4	4
ISN_EXT_INT5	5
ISN_EXT_INT6	6
ISN_EXT_INT7	7
ISN_DMA_INT0	8
ISN_DMA_INT1	9
ISN_DMA_INT2	10
ISN_DMA_INT3	11
ISN_XINT0	12
ISN_RINT0	13
ISN_XINT1	14
ISN_RINT1	15

大家觉得上面的内容还不够丰富，请参阅 TI 的文档 spru273b.pdf。

有了上面的理解，中断的实现过程就比较清楚了：

1. 初始化中断服务表指针（ISTP）：intr_init();
2. 选择用哪一个中断：　　intr_map(CPU_INT7,ISN_EXT_INT7);
3. 清中断：　　INTR_CLR_FLAG(CPU_INT7);
4. 中断服务子程序与中断号挂钩：　　intr_hook(**Int7_ISR**,CPU_INT7);
5. 打开非屏蔽中断：　　INTR_ENABLE(CPU_INT_NMI);
6. 打开所选中断：　　INTR_ENABLE(CPU_INT7);
7. 全局中断使能：　　INTR_GLOBAL_ENABLE();

读者可能会注意到，中断处理函数都是小写的，而宏定义都是大写的，在 C 语言的语法里是要注意的，否则会出现找不到函数或者函数未定义。

如果想用其他的中断源，可以按照上面的步骤依样而行，相信不会是什么太难的事情了。每一步不是必须的，顺序也不是固定的。**Int7_ISR** 是本人举例时用的中断服务子程序名，大家可以使用任意的名字，而其中的程序也是随意根据需要编写，没有太多的限制。读者如果对 DSP 的硬件很清楚，可以直接对中断寄存器进行赋值，不需要调用这些函数与宏定义。

相信现在大家对中断与 DMA 的实现已经心中有数了，但我还要强调一下，我所讲的实现是突出软件上的实现，进行 DSP 编程需要对硬件有足够必要的了解，否则会遇到某些难以理解的问题，我在这里尽量不涉及硬件，只是希望大家仔细对 TI 有关资料认真研究，避免我的介绍产生先入为主的不良影响。例如我在 DMA 中执行了对双口 RAM 的读操作，而双口 RAM 是连接在 EMIF 上的，因此，进行读操作之前就必须对 EMIF 进行初始化操作，否则，出错是必然的，而且很难找出错误原因，切记切记。