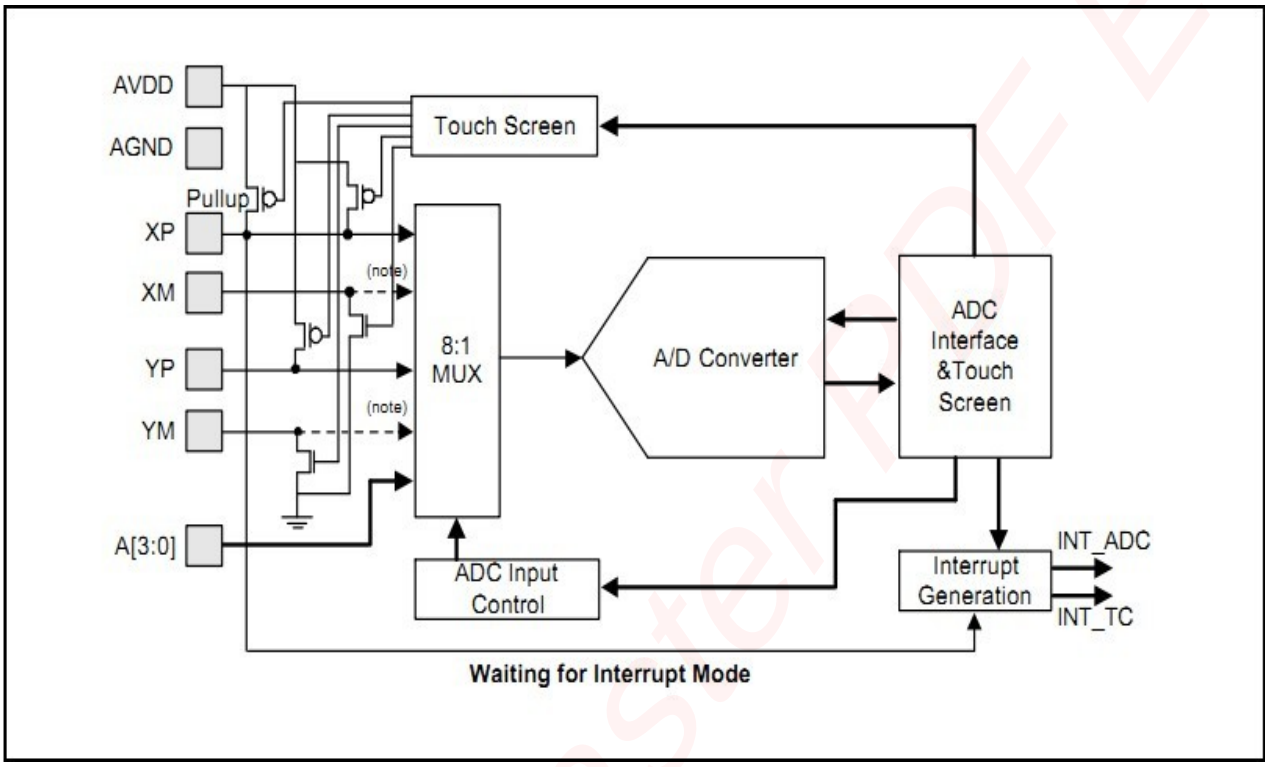
S3C2440 内部 ADC 结构图：



对于 s3c2440 来说，实现 A/D 转换比较简单，主要应用的是 ADC 控制寄存器 ADCCON 和 ADC 转换数据寄存器 ADCDAT0。寄存器 ADCDAT0 的低 10 位用于存储 A/D 转换后的数据。寄存器 ADCCON 的第 15 位用于标识 A/D 转换是否结束。第 14 位用于使能是否进行预分频，而第 6 位到第 13 位则存储的是预分频数值，因为 A/D 转换的速度不能太快，所以要通过预分频处理才可以得到正确的 A/D 转换速度，如我们想要得到 A/D 转换频率为 1MHz，则预分频的值应为 49。第 3 位到第 5 位表示的是 A/D 转换的通道选择。第 2 位可以实现 A/D 转换的待机模式。第 1 位用于是否通过读取操作来使能 A/D 转换的开始。第 0 位则是在第 1 位被清零的情况下用于开启 A/D 转换。

驱动代码：

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <mach/hardware.h>
#include <asm/io.h>
#include <linux/interrupt.h>
#include <asm/irq.h>
#include <linux/irq.h>
```

```
9.  #include <linux/miscdevice.h>
10.#include <linux/types.h>
11.#include <linux/clk.h>
12.#include <linux/errno.h>
13.#include <asm/uaccess.h>
14.#include <mach/regs-clock.h>
15.#include <plat/regs-adc.h>
16.#include <linux/kernel.h>
17.#define ADC_MINOR 100
18.#define ADC_NAME "lwp-adc"
19.struct clk *adc_clk;
20.int adc_base;
21.int adc_finish = 0;
22.static int adc_data;
23.DECLARE_WAIT_QUEUE_HEAD(adc_wait);
24.
25.static irqreturn_t adc_interrupt(int irq, void *dev_id){
26.    if(!adc_finish){
27.        adc_data = readl(adc_base + S3C2410_ADCDAT0) & 0x3ff;                //ad 转换结束会产生中断，
   此时读取 S3C2410_ADCDAT0 的 0 ~ 9 位，来获取数据
28.        adc_finish = 1;
29.        wake_up_interruptible(&adc_wait);                                    //唤醒等待其上的进程
30.    }
31.    return IRQ_HANDLED;
32.}
33.
34.int myadc_open(struct inode *inode, struct file *file){
35.    int ret;
36.    ret = request_irq(IRQ_ADC, adc_interrupt, IRQF_SHARED, ADC_NAME, 1);        //这里注册中断，
   ad 转换结束通知 cpu 有两种方式，一种靠 cpu 轮询标志位，一种靠中断
37.    if(ret){
38.        printk("IRQ %d can't get/n", IRQ_ADC);
39.        return -1;
40.    }
41.    return 0;
42.}
43.
44.int myadc_close(struct inode *inode, struct file *file){
45.    return 0;
46.        return -ENOENT;
47.}
48.
49.void start_adc(){                                    //开始 ad 转换
50.    int tmp;
51.    tmp = 0xff<<6 | 1<<14;                            //设置预分频使能，值为 255，使用通道 AIN0
52.    writel(tmp ,adc_base + S3C2410_ADCCON);
53.    tmp = readl(adc_base + S3C2410_ADCCON);
54.    tmp |= 1<<0;                                    //使能 AD 转换
```

```
55.    writel(tmp, adc_base + S3C2410_ADCCON);
56.}
57.
58.ssize_t myadc_read(struct file *filp, char __user *buff, size_t count, loff_t *offp){
59.    start_adc();                                        //要读数据开始 adc 转换
60.    wait_event_interruptible(adc_wait, adc_finish);                    //等待转换结束
61.    adc_finish = 0;
62.    copy_to_user(buff, (char *)&adc_data, sizeof(adc_data));            //将获得的数据拷贝到用户空间，
   数据会在中断程序中获得
63.    return sizeof(adc_data);
64.}
65.
66.static struct file_operations adc_ops = {
67.    .owner = THIS_MODULE,
68.    .open = myadc_open,
69.    .release = myadc_close,
70.    .read = myadc_read,
71.};
72.
73.static struct miscdevice adc_misc = {
74.    .name = ADC_NAME,
75.    .minor = ADC_MINOR,
76.    .fops = &adc_ops,
77.};
78.
79.static int __init my_adc_init(void){
80.    unsigned int ret;
81.    adc_clk = clk_get(NULL,"adc");                    //由于 ad 转换需要时钟，所以这里获取时钟
82.    if(!adc_clk){
83.        printk(KERN_ERR "fail to find adc clk resource!/n");
84.        ret = -1;
85.        goto err_clk;
86.    }
87.    clk_enable(adc_clk);                            //使能 adc 的时钟
88.    adc_base = ioremap(S3C2410_PA_ADC, 20);            //获得 ADC 控制寄存器的虚拟地址
89.    if(adc_base == 0){
90.        printk(KERN_ERR "fail to ioremap!/n");
91.        ret = -1;
92.        goto err_nomap;
93.    }
94.    ret = misc_register(&adc_misc);                    //注册这个 adc 为混杂设备
95.    if(IS_ERR(ret)){
96.        goto err_register;
97.    }
98.    return 0;
99.
```

```
100.err_register:
101.    iounmap(adc_base);
102.err_nomap:
103.    clk_disable(adc_clk);
104.    clk_put(adc_clk);
105.err_clk:
106.    return ret;
107.}
108.
109.static void __exit my_adc_exit(void){
110.    misc_deregister(&adc_misc);
111.    free_irq(IRQ_ADC, 1);
112.    iounmap(adc_base);
113.    clk_disable(adc_clk);
114.    clk_put(adc_clk);
115.}
116.
117.module_init(my_adc_init);
118.module_exit(my_adc_exit);
119.MODULE_AUTHOR("liwanpeng");
120.MODULE_LICENSE("GPL");
```

测试代码：

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <errno.h>
4.
5.  int main(int argc, char **argv)
6.  {
7.      int fd;
8.      fd = open("/dev/lwp-adc", 0);
9.      if(fd < 0)
10.     {
11.         printf("Open ADC Device Faild!/n");
12.         exit(1);
13.     }
14.     while(1)
15.     {
16.         int ret;
17.         int data;
18.         ret = read(fd, &data, sizeof(data));
19.         if(ret != sizeof(data))
20.         {
21.             if(errno != EAGAIN)
22.             {
23.                 printf("Read ADC Device Faild!/n");
24.             }
```

```
25.         continue;
26.     }
27.     else
28.     {
29.         printf("Read ADC value is: %d/n", data);
30.     }
31.   }
32.   close(fd);
33.   return 0;
34.}
```

实验效果：

```
Read ADC value is: 386
Read ADC value is: 386
Read ADC value is: 387
Read ADC value is: 387
Read ADC value is: 387
Read ADC value is: 387
Read ADC value is: 386
Read ADC value is: 387
Read ADC value is: 386
Read ADC value is: 387
Read ADC value is: 386
Read ADC value is: 386
Read ADC value is: 387
Read ADC value is: 387
Read ADC value is: 387
Read ADC value is: 387
Read ADC value is: 386
Read ADC value is: 386
Read ADC value is: 386
Read ADC value is: 387
Read ADC value is: 386
Read ADC value is: 386
Read ADC value is: 386
Read ADC value is: 387
Read ADC value is: 386
Read ADC value is: 385
Read ADC value is: 386
Read ADC value is: 387
```

嵌入式资源免费下载

总线协议：

1. 基于 PCIe 驱动程序的数据传输卡 DMA 传输
2. 基于 PCIe 总线协议的设备驱动开发
3. CANopen 协议介绍
4. 基于 PXI 总线 RS422 数据通信卡 WDM 驱动程序设计
5. FPGA 实现 PCIe 总线 DMA 设计
6. PCI Express 协议实现与验证
7. VPX 总线技术及其实现
8. 基于 Xilinx FPGA 的 PCIE 接口实现
9. 基于 PCI 总线的 GPS 授时卡设计
10. 基于 CPCI 标准的 6U 信号处理平台的设计
11. USB30 电路保护
12. USB30 协议分析与框架设计
13. USB 30 中的 CRC 校验原理及实现
14. 基于 CPLD 的 UART 设计
15. IPMI 在 VPX 系统中的应用与设计
16. 基于 CPCI 总线的 PMC 载板设计
17. 基于 VPX 总线的工件台运动控制系统研究与开发

VxWorks：

1. 基于 VxWorks 的多任务程序设计
2. 基于 VxWorks 的数据采集存储装置设计
3. Flash 文件系统分析及其在 VxWorks 中的实现
4. VxWorks 多任务编程中的异常研究
5. VxWorks 应用技巧两例
6. 一种基于 VxWorks 的飞行仿真实时管理系统
7. 在 VxWorks 系统中使用 TrueType 字库
8. 基于 FreeType 的 VxWorks 中文显示方案
9. 基于 Tilcon 的 VxWorks 简单动画开发
10. 基于 Tilcon 的某武器显控系统界面设计
11. 基于 Tilcon 的综合导航信息处理装置界面设计

12. VxWorks 的内存配置和管理
13. 基于 VxWorks 系统的 PCI 配置与应用
14. 基于 MPC8270 的 VxWorks BSP 的移植
15. Bootrom 功能改进经验谈
16. 基于 VxWorks 嵌入式系统的中文平台研究与实现

Linux：

1. Linux 程序设计第三版及源代码
2. NAND FLASH 文件系统的设计与实现
3. 多通道串行通信设备的 Linux 驱动程序实现
4. Zsh 开发指南-数组
5. 常用 GDB 命令中文速览
6. 嵌入式 C 进阶之道
7. Linux 串口编程实例
8. 基于 Yocto Project 的嵌入式应用设计
9. Android 应用的反编译
10. 基于 Android 行为的加密应用系统研究
11. 嵌入式 Linux 系统移植步步通
12. 嵌入式 CC++语言精华文章集锦
13. 基于 Linux 的高性能服务器端的设计与研究
14. S3C6410 移植 Android 内核
15. Android 开发指南中文版
16. 图解 Linux 操作系统架构设计与实现原理（第二版）

Windows CE：

1. Windows CE.NET 下 YAFFS 文件系统 NAND Flash 驱动程序设计
2. Windows CE 的 CAN 总线驱动程序设计
3. 基于 Windows CE.NET 的 ADC 驱动程序实现与应用的研究
4. 基于 Windows CE.NET 平台的串行通信实现
5. 基于 Windows CE.NET 下的 GPRS 模块的研究与开发
6. win2k 下 NTFS 分区用 ntldr 加载进 dos 源代码
7. Windows 下的 USB 设备驱动程序开发
8. WinCE 的大容量程控数据传输解决方案设计
9. WinCE6.0 安装开发详解
10. DOS 下仿 Windows 的自带计算器程序 C 源码

11. G726 局域网语音通话程序和源代码
12. WinCE 主板加载第三方驱动程序的方法
13. WinCE 下的注册表编辑程序和源代码
14. WinCE 串口通信源代码
15. WINCE 的 SD 卡程序[可实现读写的源码]
16. 基于 WinCE 的 BootLoader 研究

PowerPC：

1. Freescale MPC8536 开发板原理图
2. 基于 MPC8548E 的固件设计
3. 基于 MPC8548E 的嵌入式数据处理系统设计
4. 基于 PowerPC 嵌入式网络通信平台的实现
5. PowerPC 在车辆显控系统中的应用
6. 基于 PowerPC 的单板计算机的设计
7. 用 PowerPC860 实现 FPGA 配置

ARM：

1. 基于 DiskOnChip 2000 的驱动程序设计及应用
2. 基于 ARM 体系的 PC-104 总线设计
3. 基于 ARM 的嵌入式系统中断处理机制研究
4. 设计 ARM 的中断处理
5. 基于 ARM 的数据采集系统并行总线的驱动设计
6. S3C2410 下的 TFT LCD 驱动源码
7. STM32 SD 卡移植 FATFS 文件系统源码
8. STM32 ADC 多通道源码
9. ARM Linux 在 EP7312 上的移植
10. ARM 经典 300 问
11. 基于 S5PV210 的频谱监测设备嵌入式系统设计与实现
12. Uboot 中 start.S 源码的指令级的详尽解析

Hardware：

1. DSP 电源的典型设计
2. 高频脉冲电源设计
3. 电源的综合保护设计
4. 任意波形电源的设计
5. 高速 PCB 信号完整性分析及应用
6. DM642 高速图像采集系统的电磁干扰设计
7. 使用 COMExpress Nano 工控板实现 IP 调度设备