源码开放学ARM

@亚嵌李明老师[1]

2012-05-10

前言

学习目标

ARM 阶段的学习，构成了嵌入式软件开发工程师知识体系中不可缺少的一个内容。在这个阶段我们努力培养学员具备以下的素质和能力：

1）掌握ARM体系结构和汇编语言。
2）能够初步学会阅读硬件原理图和芯片数据手册。
3）具备为SoC芯片常见外设如 UART，NandFlash，Timer 等编写驱动程序的能力。
4）能够完成 Bootloader 项目的程序编写和移植工作。

适合对象

本阶段对于学员学习基础的要求如下：

1）掌握C语言，熟悉指针的用法。
2）学过计算机组成原理和数字电路等课程。
3）具备一定的英文阅读能力。
4）对计算机底层的运行机制和软硬件协同工作具有浓厚的兴趣。

如何写作本书的

对于有志于参与本书编写的学员，可以通过学习以下内容来进一步了解关于如何写书的相关知识。

如何安装 GIT

* http://progit.org/book/zh/ch1-4.html

如何使用 GitHub

* http://www.worldhello.net/gotgithub/index.html

如何用 markdown 写书

* http://www.slideshare.net/larrycai/write-book-in-markdown

如何生成 pdf 版本

* http://github.com/larrycai/kaiyuanbook/blob/master/BUILD.md

简介轻量级标记语言 Markdown

* http://www.worldhello.net/gotgithub/appendix/markups.html

Github 偏爱的 Markdown

* http://github.github.com/github-flavored-markdown/

在线的 Markdown 编辑浏览器

* http://dillinger.io/

致谢

在本书的编写过程中，得到了很多热心人士，同时也是技术高手的帮助。

Peter Wang (@happycasts) — 开源电子书 LGCB 的作者，从他这儿我学会了用 github 搭建这本书的写作方法，Peter 还有很多精心录制的学习开源技术的视频，大家一定不要错过。

chunzi — Pro Git 的中文译者，正是这本书教会了我如何使用Git，启发了我想要通过协作迭代来写书，后来我发现协作本身比写作更有乐趣。虽然后来没有采用Pro Git的框架，但还是非常感谢chunzi及时回复了我的邮件，相信这本书还会帮助到更多的人。

Larry Cai (@larrycaiyu) — 上海爱立信研发中心的软件开发高级专家，从微博上加入“中文开源技术书”之后，身为管理员的larry就一直默默主动帮我解决从 markdown 到生成 pdf 格式的各种细节问题，他所维护的 mkbok 这里强烈推荐噢，以后一定会成为 GitHub 上写书必备之利器。

tonghuix — 倡导自由开源生活方式的 @爱开源未来，正是他的提醒，我最后将本书的名字定为《源码开放学ARM》，以此表示对开源社区文化的尊重和支持，等有机会我会争取在开源硬件上写一本真正开源学习的书。

所有来自这些朋友的热情帮助和技术支持，让我常有如拨云雾而见青天之顿悟，虽然你们中的大多数未有机会得以见面，但对技术传播和分享的共同热爱使我们心灵相通。在此希望能够一并致谢。

参与

你可以通过 @亚嵌李明老师 联系作者或者发邮件给 limingth AT gmail.com 告知你希望出现在书中的内容和想要解决的问题。

如果你愿意参与本书的编写，可以通过 fork 《源码开放学ARM》 作出贡献。

目录

第 1 章

开发环境搭建

1.1 硬件平台

本课程采用 广州友善之臂 的 Tiny210 开发板 作为实验开发平台。 关于这个硬件开发板的详细描述和介绍，可以参考阅读下面这个链接的内容。 http://arm9.net/tiny210.asp

图 1.1: Tiny210广州友善之臂开发板

请通过阅读上述材料之后，回答以下有关开发板硬件平台的问题：

1）开发板采用的主芯片是什么型号，基于什么 ARM 内核？
2）开发板运行程序的主频是多少？ 使用什么内存？ 内存有多大？
3）开发板上面能够运行哪几种操作系统？ 它们有什么差别？
4）什么叫 BSP ，开发板的 BSP 支持哪些外设？

1.1.1 芯片识别

```
board: tiny210
CPU: S5PV210 (封装/FBGA)
MEM: K4T1G08
```

```
FLASH: K9F2G08
NET: DM9000
AUDIO: WM8960
UART:  MAX3232
```

1.1.2 外设识别

```
reset键
key键/home/back/menu
串口
网口
USB口
SD卡
CAMERA接口
MIC/HEARPHONE
AV口
启动跳线：NAND-SDboot
可变电阻：ADC
LCD接口：外接LCD
```

1.1.3 准备工作

需要参加课程的学员提前准备好以下环境：

```
1）电源线 （5v）
2）串口线 （双母头）
3）开发板 （已经烧写了 u-boot 或者用 SD 卡可以启动到 u-boot 下）
4）超级终端 （hypertrm，115200，无硬件流控，连接开发板有输出）
5）如果是用笔记本，通常没有串口，需要自备一根 USB转RS232串口的线，并安装相应驱动。
   推荐使用 Z-Tek 力特，驱动比较好装： http://www.360buy.com/product/134961.html
```

如果开发板是刚拿到的，则一般都没有烧写 u-boot，可以自己烧写 u-boot 到开发板SD卡上，这需要准备以下条件：

```
1）SD卡（自备，2G或者4G都可以）
2）SD卡读卡器（自备，连接PC后格式化为 FAT32 分区，可以显示盘符）
烧写SD卡的步骤可以参考随开发板附带的《用户手册》，需要用到 SD-Flasher.exe 这个工具。
```

1.1.4 硬件平台的验证

在开始学习后继内容之前，通常需要对硬件平台进行以下3个方面的正确性验证：

```
1）主芯片 （通常使用 jtag 工具，能够读取到 cpu 的 ID）
2）串口输出 （通常使用 超级终端，设置好波特率和流控制，能够和开发板bootloader进行交互）
3）下载和烧写 （一般通过串口或者usb进行，下载是到 SDRAM，烧写是到 NandFlash）
```

串口连接开发板，验证bootloader命令的输入输出：

```
1）连接开发板
2）启动超级终端 hypertrm(.exe)
3）选择 COM1（pc）
4）修改波特率为 115200
5）修改流控制为 无（none）（否则影响输入）
6）连接之后，重启开发板，及时按回车键，输入 help
   （如果发现有输出但无法输入，留意 Scroll Lock 是否被误按）
```

嵌入式开发的硬件平台，是以后我们学习ARM开发课程的实验平台。目前常见的硬件开发板从 ARM7、ARM9、ARM11内核一直到 Cortex A/R/M 系列发展很快，大部分都是采用核心板+底板的结构，如何进一步了解硬件设计的原理，弄懂主芯片和外设之间的作用关系，就需要我们了解硬件设计原理图方面的知识了。

1.2 硬件原理图

通常提供硬件平台的厂家，都会随开发板光盘附带硬件原理图。硬件原理图一般都是以PDF格式提供，这和硬件设计人员通过使用 Protel/OrCAD Capture 这样的硬件原理图设计工具产生的文件不同，但PDF的文档方便查阅，易于打开，对于嵌入式软件开发工程师用于开发已经足够了。

Tiny210-core-board.pdf

Tiny210-mother-board.pdf

推荐使用 FoxReader 福昕阅读器来打开 PDF 文档

```
1. Ctrl + 放大  Ctrl - 缩小  Ctrl 滚轮也可以
2. Ctrl + F 查找
3. Ctrl + Shift + N 跳转
4. Ctrl + TAB 切换文档
```

1.2.1 原理图包含的信息

核心板的原理图和底板的原理图，通常分为两个文件存放。通过查看原理图，能够知道哪些信息?

有哪些芯片
芯片有哪些管脚
芯片之间的连接关系
芯片与外设本身之间的连接关系
得到有些芯片管脚的默认连接（接地/接电源/NC（No Connection）不连接的管脚）
电阻电容等分立元件（模拟电路）

1.2.2　核心板

核心板一般包含了最小系统，也就是主芯片SoC，内存SDRAM，闪存FLASH，复位电路Reset，调试接口JTAG，时钟CLOCK和电源Power。请在原理图上分别找到这些器件，并初步熟悉了解它们的芯片型号。

```
主芯片     SoC:  S5PV210 -> 584pin [S5PV210_UM_REV1.1.pdf]
内存       Mem: K4T1G -> 1Gb=128MB  [K4T1G164QE_rev11.pdf]
闪存       Flash:  K9F2G08 -> 2Gbx8bit=256MB [K9F2G08.pdf]
复位电路   Reset: Max811 [MAX811T.pdf]
调试电路   Jtag: TCK/TDI/TDO/TMS/TRST
时钟      CLOCK: 24Mhz (X1) XXTI/XXTO
发光二极管 LED: LED1-4
高清接口   HDMI: mini HDMI(CON8)
```

1.2.3　底板

底板一般包含了常见外设，例如通用GPIO，串口UART，网卡Net，液晶屏接口LCD，音频接口Audio等。请在原理图上找到这些器件，并初步熟悉了解它们的芯片型号。

```
CONN-AB: 30*2 * 2个 = 120pin 引出 (例如串口TxD/RxD)
CONN-C: 15*2 = 30pin 引出 (例如I2C, CAM)
Power-On/Off:  S1
NET :  DM9000AEP (Ethernet) HR91105A [DM9000.pdf]
Audio: WM8960 (DA/AD convert) [WM8960_Rev40.pdf]
Buzzer: PWM Timer (beep)
RTC: Real-Time Clock
I2C-Eeprom: MAC address (6 bytes)
Buttons: K1-K8 (XEINT16-27)
SD-CARD: Data(4pin) + (4pin)
UART: 串口(通用异步收发器) [MAX3232.pdf]
LCD :  40pin / 45pin (DATA-24pin) [H43-HSD043I9W1.pdf]
```

以上所有芯片的数据手册，都可以在开发板附带光盘中找到，也可以从 https://github.com/limingth/ARM-Resources 链接下载得到。

1.3　开发工具链

在ARM开发领域，有两大类开发工具可以选择，一类是基于 Windows 平台的 SDT，ADS，RealView MDK 系列，一类是基于 Linux 平台的 GNU Cross-Toolchain 。

考虑到从简单到复杂的学习路线，我们先介绍 Windows 平台上的工具链。一旦我们对工具链背后的开发思路比较了解之后，再来学习 Linux 上的工具就会比较容易上手。

有关 ADS 工具和 MDK 工具的介绍，可以参考阅读百度百科的介绍。

[ADS简介] (http://baike.baidu.com/view/171249.htm#sub6295819)

[MDK简介] (http://baike.baidu.com/view/1745465.htm)

总体说来，MDK 是 ADS 的升级版本，界面上做了很大改动，但后台使用的命令行工具链基本一样。以下就是以 ADS 安装为例，对命令行工具链做一个简单说明。

1.3.1 安装说明

工具下载： http://limingth.github.com/ARM-Tools
ADS1.2.zip 解压之后运行 setup.exe 安装

```
注意： 1) Full 安装，不是 Typical
2) Install Licence，在解压后的 crack\licence.dat
```

安装完成之后

```
安装目录： C:\Program Files\ARM\ADSv1_2\Bin
图形开发环境：
IDE.exe - ADS IDE
axd.exe -AXD debugger
```

1.3.2 命令行开发工具链 /(GNU tools-chain)

启动命令行方式

```
开始 -> 运行 -> cmd 命令打开一个窗口

C:>path
是否有 C:\Program Files\ARM\ADSv1_2\bin

如果是 path 问题，需要添加 路径到 path 环境变量
我的电脑 -> 鼠标右键属性 -> 高级 -> 环境变量 -> 系统变量下面添加
(注意用分号间隔；原来的不要删除掉，把 C:\Program Files\ARM\ADSv1_2\bin 添加到最后)

C:>armcc
是否有这个命令，这一点最重要，如果成功则会输出
ARM C Compiler, ADS1.2 [Build 805]

Usage:         armcc [options] file1 file2 ... filen
Main options:
xxxxx

armcc.exe -C compiler (armcpp.exe) /(gcc)
armasm.exe -ASM Assembler /(as)
armlink.exe -Linker /(ld)
fromelf.exe -Bin-Utils /(objdump/objcopy)
```

1.3.3 C 编译器

```
armcc 常用编译参数
-c 只编译，不连接
```

```
-D (定义)条件编译 (-DDEBUG)
-U (不定义)条件编译 (-DDEBUG)
-g 增加调试信息
-I 指定 include 路径(自己的)
-On 编译优化级别
-S 生成汇编
-o 指定生成文件名

思考问题： arm-linux-gcc 交叉编译器的库 和 armcc 链接的库是否一样？

用法举例：
armcc hello.c
默认会生成  __image.axf (a.out)，其中 axf 文件是 ELF 格式的可执行文件

armcc -c hello.c
默认会生成  hello.o ，此时还需要 link 之后才能生成 axf 可执行文件

特殊用法：
如果一个C程序，没有 main 函数，编译会怎么样？ （有警告warning，无错误error，能生成可执行文件axf）
```

1.3.4　asm 汇编器

```
armasm 通常只生成 .o 的目标文件

用法举例：
armasm start.s
默认会生成  start.o，此时还需要 link 之后才能生成 axf 可执行文件
```

1.3.5　obj 链接器

```
armlink 常用链接参数
-ro-base 指定可执行代码的位置（代码段执行地址）
-rw-base 数据段执行地址
-first  指定 .o 文件放在链接的最开始处
-entry  指定 axd 调试工具加载 axf 文件的入口地址（转成bin之后就丢失了）
-scatter file 指定链接脚本（.scf文件/在linux下.lds文件）

用法举例：
armlink hello.o -o hello.axf
armlink -first start.o -ro-base 0x0 -entry begin start.o main.o -o hello4.axf
```

1.3.6　二进制转换工具

```
fromelf 常用转换参数
-bin 生成bin文件，最终烧写到开发板上
-c 生成txt文本文件，反汇编文件
```

```
-d 打印数据段内容
-s 打印符号表
-t 打印字符串表

用法举例：
fromelf -bin hello.axf -o hello.bin
fromelf -c -s -d hello.axf -o hello.txt
```

1.3.7 综合应用

```
单独汇编程序的编译链接
armasm start.s
armlink start.o -o demo.axf

单独C程序的编译链接
armcc -c hello.c
armlink hello.o -o hello.axf

汇编和C程序的混合链接
armasm start.s
armcc -c main.c
armlink -first start.o -ro-base 0x0 -entry begin start.o main.o -o demo.axf
```

1.3.8 ARM Docs 开发文档

```
DDI0100E_ARM_ARM.pdf   - ARM体系结构知识，侧重于内核
ADS_CompilerGuide_D.pdf   - 编译器使用
ADS_AssemblerGuide_B.pdf  - 汇编器使用
ADS_LinkerGuide_A.pdf  - 链接器使用
ADS_DebugTargetGuide_D.pdf  - 调试器使用
```

1.4 基本开发流程

1.4.1 编写源码

ARM 汇编语言格式要点

TAB 开头

ARM 汇编语言格式要求除了 label 标号之外，其他包含伪操作和指令的行都应该以一个 TAB 开头。

AREA 名称

AREA 表示定义代码段/数据段区域的开始，后面跟的名称会进入符号表参与链接。

ENTRY 入口

表示指定汇编程序的入口，最多只能写一个，也可以不写。

END 结束

表示汇编程序的结束，必须要写。

; 注释

分号表示注释，用 // 或者 /* */ 都不是注释。

label 定义符号

用于程序跳转时的标号，必须顶格写。

import symbol

用于链接外部的符号名，例如跳转到C语言的主函数。

export symbol

用于对外输出汇编程序内部的符号名，汇编文件内部的symbol默认对外是不公开的，隐藏的，类似C程序的static修饰。

C 程序注意要点

main函数的负面影响

链接器会引入很多外部代码，带来跟踪调试的很多问题，同时造成可执行bin文件的体积明显增大。

** “main的用法” **

默认main函数真正的入口是 ” main”，因此直接用 “__main” 来定义C语言的主函数是比较便利的做法。

替换return 0

如果主程序没有无限循环，那么最后执行 return 0 的结果是不可预期的，应该用while(1); 无限循环来替代。

volatile 修饰符

编译器对 volatile 修饰的变量或者指针，都会避免优化，强制访存。这对于嵌入式寄存器的读写操作是非常必要的。

unsigned 修饰符

对于特殊功能寄存器的访问，通常不需要进行算术运算，将它们声明为 unsigned 无符号整型是通常的做法。

1.4.2 编写 Makefile

all 目标

第一个默认的目标，一般都起名为 all，make执行时不需要跟目标名，会自动执行默认目标。

clean 目标

清除中间产生的不必要的文件，例如 .o .bak 等。

宏变量

引入 PRJ，SRCS，OBJS ，增强项目的可移植性。

匹配替换技巧

$(SRC:.c=.o) 的用法，灵活替换源文件到目标文件。

依赖关系的传递

用依赖关系，对于文件数量较多，编译时间较长的项目非常有好处，可以做到增量编译。

隐含规则

%o:%c 的用法，如果需要修改编译参数，则需要重新实现该条隐含规则。

1.4.3 make 编译项目

```
make clean 清除原有临时文件
make       重新生成所有文件
```

1.4.4 测试可执行文件

```
# loadb 输入命令
超级终端菜单 -> 传送 -> 发送文件 -> 选择文件 + Kermit协议 -> 点击发送

# go 0x21000000
之后观察 LED1 灯亮的现象
```

第 2 章

芯片手册导读

2.1 内部结构框图

如何阅读芯片手册，是很多初学者需要面临和解决的问题，但大部分人都会感到难以入手不知所措。除了数据手册一般都全部采用英文书写所带来的困难之外，更多的困难还在于不知道如何去阅读相关章节，不知道哪些是真正重要的或者哪些是无关紧要的内容。

我们以 S5PV210 芯片数据手册为例，给大家讲解一下应该如何阅读芯片手册。这本手册大约有2000多页，如果能够把握住其中的内容，则阅读其他手册也应该没有障碍。

2.1.1 S5PV210 芯片数据手册目录结构

```
一、Overview
1. overview
block diagram
2. memory map
System Memory Map
SFRs (Special Function Register)
0xE0000000 - 0xFB6FFFFF (max-512M)
真实的存储容量：寄存器的个数 * 4bytes
以串口为例： 15个 * 4组 = 60个 * 50 = 3K * 4 = 12K
3. ball map
pin assignment
signal description
UART-RxD0/TxD0 CTS0/RTS0
二、System
1. GPIO
2. Clock
3. Power
4. boot sequence
三、Bus
1. AXI/AHB
四、Interrupt
1. Vectored Interrupt Controller
五、Memory
1. DRAM => DDR memeory
2. SROM => SRAM + ROM(Nor Flash)
3. OneNand
4. Nand
```

```
5. Compact Flash
六、DMA
1. DMA Controller
七、Timer
1. PWM Timer
2. System Timer
3. Watchdog Timer
4. RTC
八、Connectivity
1. UART
2. IIC-bus
3. SPI (serial peripheral interface)
4. USB Host
5. USB OTG
6. SD/MMC
九、Multimedia
1. LCD
2. CAM
3. G3D
4. CODEC
5. TVOUT
6. VIDEO
7. MIXER
8. IMAGE ROTATOR
9. JPEG
10. G2D
十、AUDIO
1. IIS
2. AC97
3. PCM
4. ADC (TS)
5. KeyPad
十一、Security
```

2.1.2　S5PV210 Block Diagram 芯片内部结构框图

```
PRODUCT OVERVIEW
Block Diagram
CPU（运算）（控制）
BUS（地址/数据/控制）
Controllers（特殊功能寄存器）
Pin Assignment Diagram 管脚定义
SIGNAL DESCRIPTIONS 信号描述
SPECIAL REGISTERS (SFRs)
SYSTEM MANAGER
System Memory Map（系统内存映射）
0x4000000 = 0x40M = 64M
System Manager Registers (10+)
SFR
Name + Address + R/W + desc. + Reset Value
Bit-Field 位域
Timing 时序图
Controller 控制器
```

```
OVERVIEW 综述
Block Diagram 框图
SPECIAL REGISTERS 特殊功能寄存器
Timing 时序图
```

2.1.3 芯片手册的一般结构

```
CORE - Cortex-A8
ALU（运算器）
Regs（通用寄存器）
MMU
CACHE
BUS
片内 - 地址线32bit
片外 - 地址线取决于BANK的大小
iRAM (SRAM)
iROM
Peripheral Controllers
Memory cont.
GPIO cont.
UART cont.
USB cont.
IIC cont.
SD cont.
SPI cont.
DMA cont.
Interrupt cont.
G3D, G2D
HDMI
CAMERA
TVOUT
IMAGE ROTATOR
JPEG
```

2.1.4 芯片手册应该怎样阅读？

```
Overview
CHIP Block Diagram, MemoryMap,
Pin Assignment, Signal Description
SFRs (how many, address range)
Controllers (how many)
System
Clock (BUS), Power
Boot Sequence（启动/引导流程）
GPIO
Device
UART
NandFlash
Memory (DDR)
```

```
DMA
Interrupt
PWM Timer
LCD + TS
AUDIO
DM9000
SD
Security
```

2.2　存储管理和地址映射

2.2.1　存储器件

对于一个SoC芯片，最重要的认识莫过于了解它所能支持的存储器件和地址空间。这对于任何一个SoC芯片而言，都应该放在精读内容的首位。

存储器件主要包括以下这些，了解和掌握它们的接口和存储特性，是做嵌入式底层开发人员所必须掌握的理论知识。

```
1）片内的 RAM
iRAM - SRAM
2）片外的 RAM
SRAM
SDRAM
DDR SDRAM
3）片内的 ROM
iROM
4）片外的 ROM
Nor Flash
Nand Flash
OneNand Flash
```

2.2.2　地址空间

地址空间主要是指32位地址线 0-4G 所对应（也可称为映射）的存储器件和访问方法（如何读写）。

```
0 地址  - 加电后运行的第一条指令
片内RAM - 可以无需初始化直接使用的存储
SFR - 特殊功能寄存器的地址范围
SDRAM  - 程序通过bootloader命令可以下载到的地址
虚地址  - 使能MMU之后的虚拟地址
```

2.2.3　Memory Map 存储映射

存储映射这个概念是本节最重要的知识，以 S5PV210 芯片为例，了解并掌握它的映射情况。

```
Boot Area: 0x0 - 0x20000000 =512M
Mirrored region depending on boot mode

DRAM0:  0x20000000 - 0x3FFFFFFF: 2^29 = 512M
DRAM1:  0x40000000 - 0x7FFFFFFF: =1G
SFR: 0xE0000000 - 0xFFFFFFFF: =512M
iROM: 0xD0000000 - 0xD0010000: =64K
iRAM:  0xD0020000 - 0xD0038000: =96K (0x18=24*0x1000)

思考问题： 根据上述地址映射，对不同地址进行的访问，会引发底层硬件产生何种响应？
*(int *)0x00000000 : it depends
*(int *)0x21000000 : DRAM
*(int *)0xE0200280 : GPIO SFR
*(int *)0xE2900000 : UART SFR
```

2.3 特殊功能寄存器

特殊功能寄存器(Special Function Register) 是在 SoC 芯片内部的一个重要组成部分，区别于 ARM 内核的 R0—R15 这样的寄存器，这些寄存器本质上和片内的 SRAM 一样，按地址访问，掉电以后值会丢失，无需初始化直接可以访问这样一些特性。

从逻辑结构上看，所有对特殊功能寄存器(以下简称 SFRs) 的访问操作的实现，都对应到不同的外设控制器上。访问指令本质上和对外部存储器的读写一样，都是通过LDR和STR汇编指令来实现的。

2.3.1 关于内核，控制器，总线和外设之间的关系

```
内核：Cortex-A8 (CORE)
寄存器 Regs (R0-R15, CPSR)
指令集 Ins (add/sub, ldr/str)
总线（访存指令）
地址概念：
cpu：32bit ldr r0, [r1]  (r1:addr)
1)SRAM:
2)SFR:
--------------------------
3)DDR:
4)Device internal memory:
总线概念：
片内总线
地址线-32bit 由 地址寄存器的字节数 决定
片外总线
地址线-29bit 由 设计cpu时，外接的单个存储器件的最大容量 决定
内存控制器
通过把 32bit 的最大访存范围，分割为若干个 bank 来进行统一访问
外设控制器概念：
外设 = 可见(接插件/关联芯片/连接线) + 不可见(外设控制器)
外设控制器 = 寄存器接口 <---- 黑匣子(类似函数库.o) ---> 外设工作的时序图
总结关系：
都是涉及到裸板驱动（无操作系统/无MMU参与）
```

本质上除了运算之外，都是变成为对地址的读写
C语言的指针，在本阶段，占一个很重要的角色。

2.3.2 特殊功能寄存器的设置

所有我们设置的SFR，都应该和外设控制器内部的工作逻辑有关。因此了解Controller的工作方式和内在逻辑，是我们能够得以正确配置这些寄存器的正确路径。

每一个 controllers 都有一批自己的寄存器，通过读写操作就可以用来进行软件编程和控制。

Register Name

全大写，未来用来宏定义的，前面部分是这个Controller的缩写，后面部分就是它的功能）
CON - control 控制
STAT - status 状态
DAT - data 数据
MOD - mode 模式
FIFO - fifo 缓冲寄存器
CFG - config 配置
CNT - counter 计数
TXH - transmit Holder 发送缓冲
RXH - receive Holder 接收缓冲
BRDIV - baud divisor 波特率分频因子

Register Address

这个地址，是在写代码的时候，所对应操作的寄存器的唯一标识。
名字只是用来帮助记忆的，不是内部标识，也不是用来给编译器的。

定义举例

```
#define PRO_ID   (*(volatile unsigned int *)0xE0000000)

volatile 关键词的作用
http://learn.akae.cn/media/index.html
http://learn.akae.cn/media/ch19s06.html

写法要点：
强制类型转换，1个括号
防止优先级结合问题，1个括号
unsigned 无符号类型，防止右移问题
int 类型对应4个字节，依据手册决定是否换为 char 类型
```

2.4 时序图

2.4.1 基本概念

时序图的基本概念

时序图是芯片与芯片之间进行数据通信所需要遵循的一种协议。通过时序图能够直观的看出，不同的芯片引脚在时间轴上的不同时刻所呈现出来的高低电平的变换，这些引脚中，有的代表地址信息，有的代表数据信息，有的代表控制信息，它们出现的先后顺序是有严格的时间参数规定的，这称为 Timing 。

芯片手册里的时序图一般都会列出其中关键的时间参数名称，在手册中都可以查到这些时间参数的具体要求。

```
min - 最小值
max - 最大值
typical - 典型值
```

[Timing Table]

第 3 章

GPIO 控制器

3.1 控制器内部结构

```
GPIO Controller
how many pin
pin number, pin name, pin mux functional
device <-> GPIO (LED1 -> GPJ2_0) 原理图
GPIO SFRs
Set mux function
控制寄存器 GPXCON
Set pin value
数据寄存器 GPXDAT
Set interrupt function
中断寄存器 ...
```

3.2 GPIO 输出引脚

General Purpose Input/Output (p92-p352)

```
1) 237 multi-functional input/output port pins
34 general port groups
2) GPIO <---> peripheral controller (signal mux)
3) GPIO Block Diagram
APB bus (addr + data) *(int *)0xE0000000 = 0x1234;
Register File (GPIO SFRs)
Mux Control (functional)
Interrupt Control (Interrupt cont.)
4) Pin Mux Description
pin name -> GPIO name
default function
```

3.3 GPIO 特殊功能寄存器

```
GPIO REGISTER DESCRIPTION
addr range: 0xE020_0000 - 0xE020_0F80
Reg1: GPA0CON: GPA0 + CON (control)
bit-field name: GPA0CON[7] -> GPA0_7
field width: [31:28]
GPA0_7 pin setting:
input: 0000
output: 0001
UART_1: 0010
INT: 1111

Reg2: GPA0DAT
bit-field: [7:0]
when input: the pin state(high-level:1 low-level:0)
when output: set bit 1, output high-level
when functional: leave this pin to peripheral controller

Reg3: GPA0PUD
Pull-up
Pull-down
00 = Pull-up/down disabled
01 = Pull-down enabled
10 = Pull-up enabled
11 = Reserved

Reg4: GPA0_INT_CON
Sets the signaling method
000 = Low level
001 = High level
010 = Falling edge triggered
011 = Rising edge triggered
100 = Both edge triggered
101 ~ 111 = Reserved

Reg5: GPA0_INT_MASK
Enables Interrupt / Masked
0 = Enables Interrupt
1 = Masked

Reg6: GPA0_INT_PEND
Interrupt occur status
0 = Not occur
1 = Occur interrupt
```

3.4 GPIO 驱动代码实现

3.4.1 汇编程序

```
TAB
AREA led
CODE, DATA
READONLY
label
instruction...
END
```

3.4.2 ARM汇编的延时函数

```
主程序中
bl delay
...

delay
ldr r0, =0x10000000
go_on
sub r0, r0, #1
cmp r0, #0
bne go_on

mov pc, lr
```

3.4.3 立即数的表示

```
mov 操作，#后面跟着的数字是立即数，会写入指令中
有效位不超过8位，同时通过循环右移偶数位得到
1 0000 0010 = 0x102
10 0000 0100 = 0x204
```

3.4.4 连接开发板

```
启动超级终端 hypertrm (.exe)
选择 COM1 (pc)
修改波特率为 115200
修改流控制为 无 (none) (影响输入)
连接之后，输入 help
如果发现有输出但无法输入，留意 Scroll Lock 是否被误按

# loadb 输入命令
超级终端菜单 -> 传送 -> 发送文件
-> 选择文件 + Kermit协议 -> 点击发送
# go 0x21000000
之后观察 LED1 灯亮啦
```

3.4.5 led.c 参考代码实现

```
// led.c
#define GPJ2CON (*(volatile unsigned int *)0xE0200280)
#define GPJ2DAT (*(volatile unsigned int *)0xE0200284)

void led_init(void)
{
// LED1-4: GPJ2_0, ... GPJ2_3
// 0001 0001 0001 0001 = output
GPJ2CON &= ~0xFFFF;
GPJ2CON |= 0x1111;

return;
}

void led_on(void)
{
// set bit 0 -> led on
GPJ2DAT &= ~0xF;

return;
}

void led_off(void)
{
// set bit 1 -> led off
GPJ2DAT |= 0xF;

return;
}
```

3.4.6 button.c 参考代码实现

```
// button.c
#define GPH2CON (*(volatile unsigned int *)0xE0200C40)
#define GPH2DAT (*(volatile unsigned int *)0xE0200C44)

void button_init(void)
{
// K1 - K4 (see mother board)
// GPH2_0 - GPH2_3
// GPH2CON[0]   [3:0]   0000 = Input
// ...
// GPH2CON[3]   [15:12]   0000 = Input
GPH2CON &= ~0xFFFF;

return;
}

int button_is_down(int which)
{
```

```
// button down -> DAT = 0
int index = which - 1;

if ((GPH2DAT & (1<<index)) == 0)
return 1;

return 0;
}
```

3.4.7 buzzer.c 参考代码实现

```
// buzzer.c
#define GPD0CON (*(volatile unsigned int *)0xE02000A0)
#define GPD0DAT (*(volatile unsigned int *)0xE02000A4)

void buzzer_init(void)
{
// buzzer GPD0CON
// [0] SET 1
GPD0CON |= 1<<0;

return;
}

void buzzer_on(void)
{
// set bit 1 -> buzzer on
GPD0DAT |= 1<<0;

return;
}

void buzzer_off(void)
{
// set bit 0 -> buzzer off
GPD0DAT &= ~(1<<0);

return;
}
```

第 4 章

CLOCK 时钟管理

4.1 时钟发生器

4.1.1 时钟源的周期换算关系

```
24Mhz 输入时钟
10^(-3) 毫秒 10^(-6) 微秒 10^(-9) 纳秒
1Khz 10^3 1Mhz 10^6 1Ghz 10^9
1Ghz -> 1纳秒
100Mhz -> 10纳秒
```

4.1.2 Clock Controller 时钟控制器 (p353-p417)

```
CMU: Clock Management Unit
总线频率
MSYS: Main  (200Mhz-100Mhz)
Cortex A8 Core
DRAM controller
DSYS: Display (166Mhz-83Mhz)
JPEG
PSYS: Peripheral (133Mhz-66Mhz)
UART
AC97
PWM Timer
GPIO

Clock Generator 时钟发生器
S5PV210 Top-Level Clocks
XXTI - 24Mhz --> (4 PLLS's input)
XrtcXTI - 32.768Khz
4 PLLs (APLL, MPLL, VPLL, EPLL)
Phase Locked Loop 锁相环（倍频）
```

4.2 时钟输出频率

4.2.1 时钟输出

```
MSYS clock domain (H->HighPerformance P->Peripheral)
AHB / APB
freq(ARMCLK)  = 1000Mhz 1000M/1
freq(HCLK_MSYS) = 200Mhz ARMCLK/5
freq(PCLK_MSYS) = 100Mhz HCLK_M/2
freq(HCLK_IMEM) = 100Mhz HCLK_M/2

DSYS clock domain
freq(HCLK_DSYS) = 166Mhz
freq(PCLK_DSYS) = 83Mhz HCLK_D/2

PSYS clock domain
freq(HCLK_PSYS) = 133Mhz
freq(PCLK_PSYS) = 66Mhz HCKL_P/2
freq(SCLK_ONENAND) = 133M/166Mhz

PLL PMS setting
```

4.3 锁相环和分频器

4.3.1 锁相环 PLL

```
PLL: Fin -> Fout 倍频 XPLL_CON
MUX: 0/1 选择器 SRC选择源 CLK_SRC
DIV: /2-16 分频器 DIV CLK_DIV

REGISTER DESCRIPTION
OM[0]: cpu pin OM[0]=0, XXTI=24M
Fin_PLL: 24Mhz
APLL: APLL_CON: e0100100: 0xa07d0301 => 1000Mhz
```

4.3.2 分频器 Divider

```
Mux_APLL: CLK_SRC0, 0xe0100200: 10001111 [0]=> 1
Mux_MSYS: CLK_SRC0, 0xe0100200: 10001111 [16]=> 0
DIV_APLL: CLK_DIV0, 0xE0100300: 14131440 [2:0]=> 0 = /1
DIV_APLL: CLK_DIV0, 0xE0100300: 14131440 [10:8]=> 100 = /5
```

4.3.3 分析 ARMCLK 的产生

```
Q1: OM[0] = 0, SRC->XXTI (24Mhz) 由硬件连线决定
Q2: APLLCON (24M->1000M) 0xE0100100 => 0xa07d0301
```

```
Fout = Fin * MDIV / (PDIV * 2 ^ SDIV-1)
= 24M * 0x7d / (3 * 2^0)
= 24M * 125 / (3*1) = 1000M
Q3: CLK_SRC0 0xE0100200 => 0x10001111
bit[0]=1  MUXPLL = 1, Fout_APLL
Q4: CLK_SRC0 0xE0100200 => 0x10001111
bit[16]=0  Fout_APLL = 1G
Q5: CLK_DIV0 Address = 0xE0100300 => 0x14131440
APLL_RATIO  [2:0]  n=0+1
ARMCLK = Fout_APLL/n = 1G/1 = 1Ghz
```

4.3.4 举例: UART 串口时钟 PCLK_PSYS 的生成过程

```
Mux_PSYS: CLK_SRC0, 0xe0100200: 10001111 [24]=>0  Fout_mpll

MPLL: MPLL_CON: 0xa29b0c01 => 667Mhz (p358)
Equation to calculate the output frequency:
FOUT = MDIV X FIN / (PDIV X 2^SDIV)
Fout = (0x29b)*24M/(12 * 2^1) = 667Mhz
Mux_MPLL: CLK_SRC0, 0xe0100200: 10001111 [4]=> 1  Fout_mpll

DIV_HCLKP: CLK_DIV0, 0xE0100300: 14131440 [27:24]=> 0100 = /5
DIV_PCLKP: CLK_DIV0, 0xE0100300: 14131440 [30:28]=> 001 = /2
```

4.4 时钟驱动代码实现

4.4.1 Clock 时钟管理知识点总结

```
时钟管理单元 CMU
MSYS Domain
MSYS (Cortex A8, DRAM)
DSYS Domain
DSYS (JPEG, IIC_HDMI)
PSYS Domain
PSYS (JTAG, NandFlash, USB, IIS, AC97, IIC, PWM Timer, RTC)

几点结论：
ARMCLK 进行分频可以得到 HCLK_MSYS, PCLK_MSYS
不同 domain 域之间，输出频率的关系 PCLK = HCLK / 2

时钟发生器 Clock Generator
锁相环 PLL (Phase-Locked Loop)
APLL, MPLL, EPLL, VPLL
倍频公式 Fout = Fin * M/(P * 2^S)
Fout_mpll / Fin_mpll

分频器 Divider
1-2-4-8-16 分频 divider
```

```
Div_out = Div_in / (DIVN + 1)
DIV_PCLKP DIV_HCLKP

二路选通器 Mux
OM 时钟源选择

输入时钟
X1: XXTI/XXTO 24Mhz
X2: XusbXTI 24Mhz
X3: XhdmiXTI 24Mhz
X4: XrtcXTI 32.768khz
24Mhz 12Mhz 常用输入频率

输出时钟
ARMCLK  (1Ghz)
HCLKM / PCLKM (200Mhz / 100Mhz)
HCLKD / PCLKD (166Mhz / 83Mhz)
HCLKP / PCLKP (133Mhz / 66Mhz)
记住 100Mhz = 10ns

Clock 时钟特殊功能寄存器
PLL_CON - APLL_CON
CLK_SRC
CLK_DIVn

经验总结
PLL Sel 决定用或者不用 PLL 锁相环的输出时钟
串口的时钟输入，采用 MPLL 的输出，但是也可以用 APLL 的输出
时钟输出的图表，可以从后往前分析
24Mhz 时钟的选择 是因为 iROM 是采用 24Mhz 时钟 (P354)

搜索芯片手册的时候，通过在图标的名字，DIV_HCKLP  DIVHCLKP , HCLKP
时钟的输出都可以通过软件来控制，输出主频越高，耗电量越高
```

4.4.2 代码举例:

```
1) 设置 ARMCLK 分频因子 从 1 到 8，把 1Ghz 调整为 128Mhz

2) 设置 DIV_PCLKP 分频因子 从 2 到 4，把 PCLK 从 66M 调整为 33M
修改超级终端把 波特率 改为 57600 来测试 PCLK 的设置是否成功
```

第 5 章

UART 控制器

5.1 串口的硬件连接 (硬件原理图)

```
COM0 接口 DB9 九针公头
pin2: RSRXD0
pin3: RSTXD0
pin5: GND
通过 TxD 发送字符（以字节为单位 5-8bits）
通过 RxD 接收字符（以字节为单位 5-8bits）

RS232 电平：-15v->+15v（+15v-逻辑0, -15v-逻辑1）
TTL 电平： 0->+5v（0-逻辑0, 5v-逻辑1）
逻辑电平的转换： MAX3232（美信芯片）

查看 MAX3232 芯片+核心板原理图可得
RSTXD0 <- XuTxD0 -- TINY1B B7 -- XuTXD0/GPA0_1
RSRXD0 -> XuRxD0 -- TINY1B B8 -- XuRXD0/GPA0_0

结论： GPA0 管理了 UART 的 Txd/Rxd 两个引脚
```

5.2 串口的管脚功能复用

参考 S5PV210芯片手册

```
GPA0 Mux Function
查看 GPA0[0] GPA0[1], 确认了 UART 的复用功能
查看 GPA0CON 寄存器，了解如何设置（0010）
GPA0CON ： 地址 0xe0200000

[FriendlyLEG-TINY210]# md 0xe0200000
e0200000: 22222222 000000ab 00005555 00000000    """"....UU......
e0200010: 00000000 00000000 00005555 00000000    ........UU......

[FriendlyLEG-TINY210]# mw 0xe0200000 0x22222202

结论：RXD 影响接收， TXD 影响发送
```

RTS和CTS对发送和接收暂时无影响（无流控制）

5.3 串口时序图

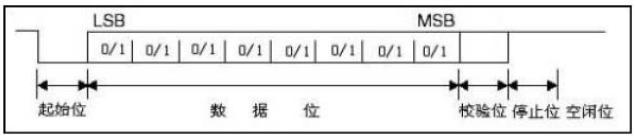

图 5.1: UART Timing

Timing:
空闲状态 high-level
起始位 start bit - 1 bit
数据位 data bit - 8bit
奇偶校验 odd/even Parity（无）
停止位 stop bit - 1 bit

5.4 串口控制器结构

5.4.1 串口控制器功能

UART Controller（p853-p882）

类似是一个函数，需要了解它的->输入，输出，如何实现
1. 输出：Timing（串行通信实现时序图）
Serial I/O Frame Timing Diagram（Normal UART） ->p860
2. 输入：SFR（串口控制器的寄存器）
REGISTER DESCRIPTION ->p864
3. 实现：Block Diagram（结构框图）
Block Diagram of UART ->p854

5.4.2 串口控制器框图

Block Diagram:

Peripheral Bus 外设总线
*(int *)SFR_ADDR = value;
Controll Unit 控制单元
Control Regs（数据位，停止位，奇偶校验位，时钟源选择，工作模式等）
Baud-Rate Gengenrator 波特率发生器
Clock Source 时钟源（PCLK=66M）

```
Transmitter 发送器
Transmit shifter 发送移位器
Transmit buffer 发送队列FIFO缓冲器
Receiver 接收器
Receiver shifter 接收移位器
Receiver buffer 接收队列FIFO缓冲器
```

5.5 串口寄存器配置

5.5.1 串口寄存器分类 SFR:

```
15 Regs Register  Address
控制类 通常是可读可写，属性 R/W 6个
ULCON0  0xE290_0000
UCON0  0xE290_0004
UFCON0  0xE290_0008
UMCON0  0xE290_000C
UBRDIV0  0xE290_0028
UDIVSLOT0  0xE290_002C

状态类 通常是只读，属性 R       4个
UTRSTAT0  0xE290_0010
UERSTAT0  0xE290_0014
UFSTAT0  0xE290_0018
UMSTAT0  0xE290_001C

数据类 通常是可读可写，属性 R/W 2个
UTXH0  0xE290_0020
URXH0  0xE290_0024

中断类 通常是可读可写，属性 R/W 3个
UINTP0  0xE290_0030
UINTSP0  0xE290_0034
UINTM0  0xE290_0038
```

5.5.2 查看 uboot 对串口寄存器的设置

```
[FriendlyLEG-TINY210]# md 0xe2900000
e2900000: 00000003 00000245 00000000 00000000    ....E...........
e2900010: 00000000 00000000 00010000 00000010    ................
e2900020: 00000000 0000000d 00000023 00000808    ........#.......
e2900030: 00000005 00000005 00000000 00000000    ................
e2900040: 00000000 00000000 00000000 00000000    ................
e2900050: 00000000 00000000 00000000 00000000    ................
e2900060: 00000000 00000000 00000000 00000000    ................
e2900070: 00000000 00000000 00000000 00000000    ................
e2900080: 00000000 00000000 00000000 00000000    ................
e2900090: 00000000 00000000 00000000 00000000    ................
```

```
ULCON0   0xE290_0000  00000003
UCON0   0xE290_0004  00000245
UFCON0   0xE290_0008  0
UMCON0   0xE290_000C  0
UBRDIV0   0xE290_0028  00000023
UDIVSLOT0   0xE290_002C  00000808

其中 FIFO control & Modem control 可以不用设置
Timing setting
ULCON0 0x3
data bit: 8bit
stop bit: 1bit
parity: none
UCON0 0x245
enable Transmit & Receive MODE: 01 01 (INT&Polling)
interrupt: disable
dma: disable

CLOCK setting
UCON0 00:  PCLK (66M)
--> 115200 bps (bit/second)
--> 波特率并不是串口控制器的工作频率，串口控制器在接收采样时，是波特率的16倍

66Mhz = 66000000hz

分频因子 = 66000000 / (115200*16) - 1
(分频因子 + 1) = PCLK/(bps*16)

UBRDIV0:  0x23 = (66000000)/(115200*16)-1= 35
UDIVSLOT0:
```

5.6　串口驱动代码实现

5.6.1　uart.c 参考代码实现

```
// uart.c
#define ULCON0   (*(volatile unsigned int *)0xE2900000)
#define UCON0   (*(volatile unsigned int *)0xE2900004)
#define UTRSTAT0   (*(volatile unsigned int *)0xE2900010)
#define UTXH0   (*(volatile unsigned char *)0xE2900020)
#define URXH0   (*(volatile unsigned char *)0xE2900024)
#define UBRDIV0  (*(volatile unsigned int *)0xE2900028)
#define UDIVSLOT0   (*(volatile unsigned int *)0xE290002C)

void uart_init(void)
{
// 66Mhz / (115200*16) - 1 = 0x23
// 66Mhz / (19200*16) - 1 = 0xD5
//UBRDIV0 = 0xD5;
return;
}
```

```
char uart_getchar(void)
{
char c;
// polling receive status: if buffer is full
//while ((UTRSTAT0 & (1<<0)) == 0)
while (!(UTRSTAT0 & (1<<0)))
;

c = URXH0;

return c;
}

void uart_putchar(char c)
{
// polling transmit status: if buffer is empty
//while ((UTRSTAT0 & (1<<2)) == 0)
while (!(UTRSTAT0 & (1<<2)))
;

UTXH0 = c;

return;
}
```

5.6.2 uart.h 参考代码实现

```
// uart.h
void uart_init(void);

char uart_getchar(void);

void uart_putchar(char c);
```

第 6 章

SDRAM 控制器

6.1 SDRAM 硬件连接

6.1.1 SDRAM 引脚描述

硬件原理图

```
1）从芯片角度
地址线 A0-A13  14根
BA0, BA1, BA2
数据线 DQ0-DQ7 * 4 = 32bit data bus (8bit * 4chips)
控制线 nCS, nRAS, nCAS

2）从处理器角度
地址线 Xm1ADDR0-Xm1ADDR13
Xm1BA0, Xm1BA1, Xm1CSn1/BA2
数据线 Xm1DATA0-Xm1DATA31
控制线 Xm1CSn0, Xm1RASn, Xm1CASn
```

6.1.2 SDRAM 内部结构

芯片手册

```
1Gbit = 128MByte

内部分 8 bank, 每个bank = 128M/8 = 16M（24根地址线）

24根地址线的信号分为行地址和列地址
行地址：14根
列地址：10根
```

6.1.3 从SoC芯片到SDRAM芯片

```
128M * 4 chips = 512M 存储容量 (512M = 2^29)

29根地址线 (A0-A28)
A28,A27,A26 : BA2,BA1,BA0
A25-A12: 行地址 14根
A11-A2: 列地址 10根
A1, A0, - GND

如何访问 0x0 地址
A31,A30,A29 : 000 -> nCS 片选无效
A28,A27,A26 : 0 00
A25-A12: 00 0000 0000 0000
A11-A2: 0000 0000 00
A1, A0: 00

如何访问 0x20000000 地址
A31,A30,A29 : 001 -> nCS 片选有效
A28,A27,A26 : 0 00
A25-A12: 00 0000 0000 0000
A11-A2: 0000 0000 00
A1, A0: 00

如何访问 0x21000000 地址
A31,A30,A29 : 001 -> nCS 片选有效
A28,A27,A26 : 0 00
A25-A12: 01 0000 0000 0000
A11-A2: 0000 0000 00
A1, A0: 00

如何访问 0x3FFFFFFF 地址
A31,A30,A29 : 001 -> nCS 片选有效
A28,A27,A26 : 1 11
A25-A12: 11 1111 1111 1111
A11-A2: 1111 1111 11
A1, A0: 11

如何访问 0x40000000 地址
A31,A30,A29 : 010 -> nCS 片选无效
A28,A27,A26 : 0 00
A25-A12: 00 0000 0000 0000
A11-A2: 0000 0000 00
A1, A0: 00
```

6.2 SDRAM 管脚功能复用

搜索芯片数据手册，发现所有管脚均无功能复用。

```
举例
Xm1ADDR[0]
Xm1DATA[0]
```

```
Xm1SCLK
Xm1RASn Xm1CASn
Xm1CSn[0]
```

6.3 SDRAM 时序图

Timing:

http://www.cnblogs.com/iqstudy/articles/2034422.html

http://www.cnblogs.com/adamite/archive/2010/05/22/1422792.html

http://blog.sina.com.cn/wangdxlove

http://blog.sina.com.cn/s/blog_5755d4b70100b3o0.html

http://blog.chinaunix.net/uid-9012903-id-3056317.html

http://blog.163.com/hanozi@126/blog/static/1865756200897105453/

6.4 SDRAM 控制器结构

http://www.doc88.com/p-47549556518.html

6.5 SDRAM 寄存器配置

```
[FriendlyLEG-TINY210]# md 0xF0000000
f0000000: 0ff02330 00202400 20e00323 00e00323    0#...$ .#.. #...
f0000010: 00110400 ff000000 79101003 00000086    ..........y....
f0000020: 00000000 00000000 ffff00ff 00000000    ................
f0000030: 00000618 2b34438a 24240000 0bdc0343    .....C4+..$$C...
f0000040: 00001db7 00000000 60000000 00000000    ..........`....
f0000050: 000005b2 00000000 00000000 00000000    ................
f0000060: 00000000 00000000 00000000 00000000    ................
f0000070: 00000000 00000000 00000000 00000000    ................
f0000080: 00000000 00000000 00000000 00000000    ................
f0000090: 00000000 00000000 00000000 00000000    ................
f00000a0: 00000000 00000000 00000000 00000000    ................
f00000b0: 00000000 00000000 00000000 00000000    ................

DDR 寄存器配置参数
data width: 32bit
row address bit: 14bit
col address bit: 10bit
memory type: DDR2
number of banks: 8banks (DDR chip)
Average Periodic Refresh Interval: 0x618
7.8us * 200Mhz = 1560 = 0x618
7.8us : 刷新周期
Timing 时间参数
...
```

6.6 SDRAM 驱动代码实现

6.6.1 课堂修改作业

1）修改 uart_init，把波特率改为 19200
2）修改 时钟发生器，把 PCLK 输出改为 33M，波特率重新计算 115200
3）实现 puts("hello, world"); 输出 "hello, world"
实现 实现 putchar_hex('a') 十六进制，输出 0x61
目的：了解 \n换行 \r回车 之间的差别

4）修改 DRAM Controller，
把 col address 的宽度改为 9bit
把 data bit 的32bit 改为 8bit
用上述 putchar_hex 输出接口，观察读取DDR内存单元的数值变化

第 7 章

NandFlash 控制器

7.1 K9F2G08 芯片

```
48-pin TSOP1 封装
NC - No Connect 不连接，留待扩展（25pin NC）
```

7.2 NandFlash 管脚功能复用

```
Signal Desc.
IO[7:0]: 无地址线，也无数据线，只有IO线（IO0-IO7）
CLE: 命令锁存使能
ALE: 地址锁存使能
nCE: 芯片使能（片选）
nRE: 读使能
nWE: 写使能
nWP: 写保护
R/nB: 就绪/忙 信号
```

7.3 Nand Flash Timing 时序

```
READ ID Timing (p31)
CLE 命令周期 写1次 90h
ALE 地址周期 写1次 00h
DAT 数据周期 读5次 id = 0x EC DA 10 95 44

READ page Timing (p23)
CLE 命令周期 写1次 00h
ALE 地址周期 写5次
CLE 命令周期 写1次 30h
忙等待周期 R/nB 高有效
数据周期 读2048次data + 64次ECC
```

7.4 NandFlash 控制器结构

7.5 NandFlash 寄存器配置

7.5.1 NandFlash 寄存器分类

```
SFR 特殊功能寄存器（部分）
控制类
NFCONF  0xB0E00000
NFCONT  0xB0E00004
NFCMMD  0xB0E00008  [7:0] 命令
NFADDR  0xB0E0000C  [7:0] 地址
数据类
NFDATA  0xB0E00010  [31:0] 数据
状态类
NFSTAT  0xB0E00028
```

7.5.2 初始化配置

```
mw 0xe0200320 0x22222222
mw 0xb0e00000 0x00006552
mw 0xb0e00004 0x00c100c5
mw 0xb0e00008 0x90
mw 0xb0e0000c 0x00
md 0xb0e00010

GPIO 功能复用设置为 NF signal
ALE: MP0_3[1]
CLE: MP0_3[0]
MP0_3CON  Address = 0xE020_0320
mw 0xe0200320 0x22222222

NFCONF 0x00001000 -> 0x00006552
NAND clock = 133M (p363-NFCON) -> 7.5ns (1 clock)
Nand Timing (k9f2g08.pdf - p13)

mw 0xb0e00000 0x00006552

NFCONT  0x00c100c6  -> 0x00c100c5
mw 0xb0e00004 0x00c100c5

NFCMMD:
mw 0xb0e00008 0x90

NFADDR:
mw 0xb0e0000c 0x00

NFDATA:
md 0xb0e00010
```

7.6 NandFlash 驱动代码实现

7.6.1 nand.c 参考代码实现

```
// nand.c
#define NFCONF  (*(volatile unsigned int *)0xB0E00000)
#define NFCONT  (*(volatile unsigned int *)0xB0E00004)
#define NFCMMD  (*(volatile unsigned char *)0xB0E00008)
#define NFADDR  (*(volatile unsigned char *)0xB0E0000C)
#define NFDATA  (*(volatile unsigned char *)0xB0E00010)
#define NFSTAT  (*(volatile unsigned int *)0xB0E00028)

#define MP0_3CON  (*(volatile unsigned int *)0xE0200320)

#define PAGE_SIZE 2048

void nand_init(void)
{
// [15:12] TACLS = 1 -> (1)  1/133Mhz = 7.5ns
// [11:8] TWRPH0 = 1 -> (1+1) 7.5ns * 2 = 15ns
// [7:4] TWRPH1 = 1 -> (1+1) 7.5ns * 2 = 15ns
NFCONF |= 1<<12 | 1<<8 | 1<<4;

// AddrCycle  [1]  1 = 5 address cycle
NFCONF |= 1<<1;

// MODE  [0]  NAND Flash controller operating mode
// 0 = Disable NAND Flash Controller
// *1 = Enable NAND Flash Controller
NFCONT |= 1<<0;

// Reg_nCE0  [1]  NAND Flash Memory nRCS[0] signal control
// *0 = Force nRCS[0] to low (Enable chip select)
// 1 = Force nRCS[0] to High (Disable chip select)
NFCONT &= ~(1<<1);

// GPIO functional mux setting
//  0010 = NF_xxx
MP0_3CON = 0x22222222;

return;
}

void nand_read_id(char id[])
{
int i;

// write read_id cmd 90h
NFCMMD = 0x90;

// write address 00h
NFADDR = 0x00;

for (i = 0; i < 5; i++)
```

```
id[i] = NFDATA;

return;
}

void nand_read_page(int addr, char buf[])
{
int i;
char tmp;

// write read_page cmd 00h
NFCMMD = 0x00;

// write 5 address
NFADDR = (addr>>0) & 0xFF;
NFADDR = (addr>>8) & 0x7;
NFADDR = (addr>>11) & 0xFF;
NFADDR = (addr>>19) & 0xFF;
NFADDR = (addr>>27) & 0x1;

// write read_page cmd 30h
NFCMMD = 0x30;

// wait for R/nB -> Ready
while ((NFSTAT & (1<<0)) == 0)
;

// read data 2048 bytes
for (i = 0; i < PAGE_SIZE; i++)
buf[i] = NFDATA;

for (i = 0; i < 64; i++)
tmp = NFDATA;

return;
}

void nand_read(int nand_addr, char * sdram_addr, int size)
{
int pages = (size - 1)/PAGE_SIZE + 1;
int i;

for (i = 0; i < pages; i++)
nand_read_page(nand_addr + i*PAGE_SIZE, sdram_addr + i*PAGE_SIZE);

return;
}
```

7.6.2 nand.h 参考代码实现

```
// nand.h
void nand_init(void);
```

```
void nand_read_id(char id[]);

void nand_read_page(int addr, char buf[]);

void nand_read(int nand_addr, char * sdram_addr, int size);
```

7.6.3 思考问题：如何访问 Flash 0地址

```
1. 软件上访问 应该是 计算出 0 地址所在的 page + block
NFCMMD = 0x00;
NFADDR = 0x0;
NFADDR = 0x0;
NFADDR = 0x0;
NFADDR = 0x0;
NFADDR = 0x0;
NFCMMD = 0x30;
wait R/nB;
read NFDATA 2048;

2. 硬件上
NFCMMD = 0x00; -> CLE 使能/ALE 禁止, IO[7:0] = 0x00;
NFADDR = 0x0; -> ALE 使能/CLE 禁止, IO[7:0] = 0x00;
NFADDR = 0x0; -> ALE 使能/CLE 禁止, IO[7:0] = 0x00;
NFADDR = 0x0; -> ALE 使能/CLE 禁止, IO[7:0] = 0x00;
NFADDR = 0x0; -> ALE 使能/CLE 禁止, IO[7:0] = 0x00;
NFCMMD = 0x00; -> CLE 使能/ALE 禁止, IO[7:0] = 0x30;
read NFSTAT; -> R/nB 线会设置 STAT 寄存器 0 位
read NFDATA; -> nRE 使能, CLE/ALE 禁止, IO[7:0] => NFDATA 中
```

第 8 章

Exception 异常处理

8.1 异常相关基本概念

8.1.1 ARM 的工作模式有几种？各是哪些？

```
7种
USR: helloworld（非特权->MSR不允许执行）
SYS: kernel        （2种非异常模式）
--------------------
SVC: reset加电     （5种异常模式）
IRQ: EINT0 key
FIQ: EINT0 key + MODE(REG)
ABT: Memory Fault
UND: execute undefine ins.
```

8.1.2 ARM 的寄存器有多少？ 各是哪些？

```
37个 = 31 通用 + 6 状态
R0-R7: 未分组（8个）
R8-R12: 分组（10个=5*2组 FIQ/~FIQ）
R13-R14: 分组（12个=2*6组 5异常+1非异常）
R13: SP（stack pointer）压栈/出栈
R14: LR（Link Register）BL/异常发生后保存PC
R15: 程序计数器（1个）
R15: PC（Program Counter）流水线 +8
-----------------------
CPSR: 程序状态寄存器 （1个）
SPSR: 备份的状态寄存器（5个-5种异常）
```

8.1.3 ARM 的异常有几种？各是哪些？

```
7种
```

```
复位异常
数据访问中止
快速中断
快速中断
普通中断
指令预取中止
软件中断： SWI
未定义指令异常： 什么样的指令是未定义？
```

8.2 异常向量表的实现

8.2.1 ARM 的异常向量表是指什么? 有什么特点？

```
异常发生后的入口地址
从0开始的32字节，7种异常都有入口，0x14保留
向量一般情况下是指地址，但是异常向量表里面存放是指令
所有ARM内核的处理器都按上述方式工作
0x0: reset
0x8: swi
0x18: IRQ
0x1C: FIQ（放在最后，那么好处是可以省一条跳转）
里面存放的是跳转指令
跳转指令可以是 B（相对，有限）/ BL(不能)
还可以是 LDR（任意跳转）
B:  0xEA000000 + offset
LDR:  0xE59ff000 + offset
```

8.3 异常处理流程

8.3.1 ARM 的软中断异常发生后，硬件做何响应？

```
硬件要做6件事情，以发生SWI异常为例：
1. 保存（PC-4）-> LR_svc（PC 是当前执行指令地址+8）
2. 保存 CPSR -> SPSR_svc
3. 修改 CPSR -> SVC mode
4. 修改 CPSR I-bit -> disable IRQ
5. 映射相应(USR ->) SVC 模式的寄存器
6. 修改 PC -> 0x8
```

8.3.2 cpu 内核跳转到 0x8 之后，软件需要做哪些工作？

PC 跳转到 0x8 之后，第一个问题是如何返回？ 包括两个返回：

```
1. 地址的返回  mov pc, lr
2. 模式的返回 movs pc, lr
```

```
mov pc, lr [0xe1a0f00e]
movs pc, lr [0xe1b0f00e]
```

直接返回没有实际意义，因此第二个问题是如何跳转到 swi_handler ？

```
b   跳转 b    swi_handler
0xEA000000 | offset offset计算公式 ( swi_handler - (0x8+8) ) / 4
offset：代表从 PC 到 目标地址 之间相差的指令数

ldr 跳转 ldr  pc, [pc, offset]
0xE59FF000 | offset offset计算公式 ( data_addr - (0x8+8) )
offset：代表从 PC 到 数据存储地址 之间相差的字节数
data_addr：代表 存储目标地址的内存单元的地址，这个地址被看成为一个数据，用ldr从内存中读出来。
```

跳转到 handler 之后， handler 需要做哪些工作?

```
swi_handler 要做以下工作：
1. 保存现场： r0-r12, r14 压栈
STMFD r13!, {r0-r12, r14}
2. 进入异常处理:
BL C_swi_handler ; 用C实现
3. 恢复现场： r0-r12, pc 出栈
A) LDMFD r13!, {r0-r12, pc}^
B) LDMFD r13!, {r0-r12, r14}
   movs pc, lr
将原来保存的 spsr 恢复给 CPSR
```

8.4 软中断异常代码实现

8.4.1 New Project

```
start.s -> add to project
main.c  -> add to project
Setting -> ARM Linker -> 1. robase 0x21000000
 2. layout start.o
 3. PostLinker fromELF

start.s
1. 切换模式到 USR = 0xD0
2. 跳转到 C

main.c
1. 在C程序中调用软中断的方法
int __swi(0x1) sys_add(int a, int b, int c);
ret = sys_add(1, 2, 3);
```

上述代码会编译生成 4 条指令
```
[0xe3a02003]   mov      r2,#3
[0xe3a01002]   mov      r1,#2
[0xe3a00001]   mov      r0,#1
[0xef000001]   swi      0x1
```

2. 在C程序中，注册 SWI 异常的处理函数
因为要跳转的地址是在 0x21000000+ 区域，
因此只能在 0x8 注册一条 LDR 跳转指令，而不是B指令
实现办法是： 0x8: 写入一条 0xE59FFxxx (xxx=>020)
0x30: 写入 handler 的地址

3. 实现一个真正的在汇编中能够返回模式和地址的 asm_swi_handler
需要修改 0x30: (int)asm_swi_handler
需要在 start.s 里面加入一个 asm_swi_handler 的函数
它的实现应该为：
```
asm_swi_handler
stmfd r13!, {r0-r12, r14}

bl C_swi_handler

ldmfd r13!, {r0-r12, r14}
movs pc, lr
```

需要 import C_swi_handler
```
export asm_swi_handler
```

4. 修改 C_swi_handler 使得它能够传递用户参数
```
int C_swi_handler(int usera, int userb, int userc)
{
return add(usera, userb, userc);
}
```

5. 修改 asm_swi_handler 使得它能够漏过返回值给用户
```
asm_swi_handler
stmfd r13!, {r1-r12, r14}

bl C_swi_handler

ldmfd r13!, {r1-r12, r14}
```

6. 最后测试时，把 0x8 处的断点去掉，
从用户看来，系统调用 sys_add 本质就是一条 swi 指令，而不是一个函数。

第 9 章

Interrupt 控制器

9.1 中断相关基本概念

9.1.1 异常和中断的概念区分

```
异常指的都是内核里面(Cortex-A8)发生的事情
例如：执行 swi 指令
中断指的都是板子上面(tiny210)发生的事情
例如：用户按下 K1 按键，
定时器Timer(不在板子上，但在芯片里)
联系在于？都有模式的切换，中断处理需要包含异常处理
异常模式包含(中断)异常模式
```

9.1.2 中断处理的相关概念

中断源

```
属于 Controller (Samsung) + Board (tiny210)
```

中断控制器 Interrupt Controller

```
属于 Controller (Samsung)
```

中断模式的响应和恢复

```
属于 Cortex-A Core (ARM)

S3C4510 - ARM7TDMI
S3C2440 - ARM920T
S3C6410 - ARM1176
S5PV210 - Cortex A8
```

9.2 中断处理流程

9.2.1 哪些事情硬件做，哪些事情软件做？

```
1. 中断触发（用户/用户设置/外部数据）
中断触发条件的初始化操作--软件有关
2. 中断响应（从当前位置跳转到 0x18）
保存原来的模式和返回的地址--硬件有关
3. 中断发生后的(老的)现场保存和恢复
压栈和出栈--软件有关
4. 中断返回
恢复原来的模式和地址--软件有关
5. 中断标志位 SFR Pending bit
pending bit 的设置--硬件有关（控制器寄存器PND）
pending bit 的清除--软件有关（控制器寄存器PND）
作用：识别中断源/调用相应的handler--软件有关(可以交给VIC实现)
6. 中断允许位 CPSR I-bit
加电之后，I-bit 的初始状态是 关闭/禁止 的（0xD3->0x53）
MSR/MRS 可以允许使能--软件有关（内核寄存器CPSR）
7. IRQ发生 -> Pending bit -> I-bit enable -> 跳转
(跳转之前，硬件会完成 disable I-bit,
因此不再响应后继任何中断，直到软件恢复CPSR)
8. 在软件恢复CPSR(重新允许中断)之前，软件需要清除之前的pending bit(VIC+ADDR寄存器=0)

内核和处理器升级的过程，就是不断把原来软件的工作交给硬件来做。
中断的优先级
中断的处理程序 ISR -> 中断的向量表
```

9.2.2 如何跳转

```
IC VIC
-----------------------------------------------------------------
b irq_handler0 VectADDR -> PC
LDR pc, =irq_handler0 VIC interface(A31-A0)
-----------------------------------------------------------------
```

9.3 中断寄存器配置

9.3.1 中断相关寄存器的设计演变

```
IC IC Vectored IC ->

ARM7(4510) ARM9(2440) ARM11(6410) & A8(210)
-------------------------------------------------------------------------------------------
CPSR I-bit CPSR I-bit CPSR I-bit
内核 VIC Port(Enable)
```

```
(Core) VIC interface(PC<->A0-A31)
-------------------------------------------------------------------------------------------
INTOFFSET       VectADDRESS(32bit->A0-A31)
Vectors(handlers)
INTPRI Priority
INTMOD INTPND (IRQ/FIQ)STATUS(PND)
中断 INTPND INTMOD SELECT(MOD) IRQ/FIQ
控制器 INTMSK INTMSK ENABLE(MSK)
(IC) SRCPND RAWINTR(SRC)
-------------------------------------------------------------------------------------------
INTMSK
INTPND(clear)
EINTCON EINTCON EINTCON
中断源 (F/R/L) (F/R/L) (F/R/L)
控制器 GPXCON GPXCON GPXCON
(GPIO) (EINT) (EINT) (EINT)
-------------------------------------------------------------------------------------------
硬件层  Key/UART/USB/Timer
```

9.3.2 S5PV210 中断相关寄存器

```
中断源 GPIO Controller
GPH2CON[0]   [3:0]   Set the pin mux function as EXT_INT
0000 = Input
0001 = Output
0010 = Reserved
0011 = KP_COL[0]
0011 ~ 1110 = Reserved
* 1111 = EXT_INT[16]

EXT_INT_2_CON[0]  [2:0]  Sets the signaling method of EXT_INT[16]
000 = Low level
001 = High level
* 010 = Falling edge triggered
011 = Rising edge triggered
100 = Both edge triggered
101 ~ 111 = Reserved

EXT_INT_2_MASK[0]  [0]
* 0 = Enables Interrupt
1 = Masked

EXT_INT_2_PEND[0]  [0]  (R)
0 = Not occur
1 = Occur interrupt

向量中断控制器 Vectored Interrupt Controller
VIC0RAWINTR  0xF200_0008 R
Specifies the Raw Interrupt Status Register
RawInterrupt  [31:0]  Shows the status of the FIQ interrupts before masking by the
VICINTENABLE and VICINTSELECT Registers:
0 = Interrupt is inactive before masking
1 = Interrupt is active before masking
```

```
VIC0INTENABLE  0xF200_0010 R/W
Specifies the Interrupt Enable Register
IntEnable  [31:0]  Enables the interrupt request lines
Write:
0 = No effect
* 1 = Enables Interrupt.

VICINTENCLEAR
IntEnable Clear
Write:
0 = No effect
* 1 = Disables Interrupt in VICINTENABLE Register.

VIC0IRQSTATUS  0xF200_0000 R
Specifies the IRQ Status Register
IRQStatus  [31:0]  Shows the status of the interrupts after masking by the
VICINTENABLE and VICINTSELECT Registers:
0 = Interrupt is inactive
1 = Interrupt is active.

VIC0INTSELECT  0xF200_000C R/W
Specifies the Interrupt Select Register
IntSelect  [31:0]  Selects interrupt type for interrupt request:
* 0 = IRQ interrupt
1 = FIQ interrupt

VIC0VECTADDR0 0xF200_0100 R/W
Specifies the Vector Address 0 Register
VICVECTADDR[0-31]  Bit  Description
VectorAddr 0-31  [31:0]  Contains ISR vector addresses.

VIC0VECTPRIORITY0  0xF200_0204 R/W
Specifies the Vector Priority 1 Register
VectPriority  [3:0]  Selects vectored interrupt priority level. You can select
any of the 16 vectored interrupt priority levels by
programming the register with the hexadecimal value of
the priority level required, from 0-15.

VIC0ADDRESS  0xF200_0F00 R/W
Specifies the Vector Address Register
VectAddr  [31:0]
Contains the address of the currently active ISR

内核
CPSR I-bit
__asm
{
mov r0, #0x53
msr CPSR_cxsf, r0
}

VIC Enable (p15)
__asm
{
mrc p15, 0, r0, c1, c0, 0
```

```
orr r0, r0, #(1<<24)
mcr p15, 0, r0, c1, c0, 0
}
```

9.4 硬件中断异常代码实现

9.4.1 实验验证结论：

```
第一阶段，观察 GPIO 控制器里面的 pending bit 设置情况

VIC0(IRQ/FIQ)STATUS 标识中断是否通过(IRQ/FIQ)
VIC0INTSELECT 选择IRQ还是FIQ
VIC0INTENABLE 使能中断通过
VIC0RAWINTR 通过 EXT_INT_2_MASK 之后的情况
---------------------
EXT_INT_2_MASK 中断 Mask bit
EXT_INT_2_PEND 中断 Pending bit
EXT_INT_2_CON 下升沿触发 Falling Edge
GPH2CON 管脚复用 EXT_INT[16]

第二阶段，如何清除 pending bit ？
写1清除 pending bit
写0无效

3种清0的写法，只有最后一种是正确的清除。
PEND |= 1<<0; (not good)
PEND = 0xFFFFFFFF; (not good)
PEND = 1<<0; (Good!)

VIC0RAWINTR 寄存器取决于 EXT_INT_2_PEND 的值，如 PEND 是 1 ，则通过 MASK 之后它也是 1；
如果 PEND 做了清除，则相应的 VIC0RAWINTR 中的位，也随之清除；无需手工清除该寄存器的位。

第三阶段，观察中断控制器中的使能Enable和状态Status标识寄存器
使能 Enable 寄存器在 RAW 和 STATUS 之间
STATUS 寄存器的标识位无需软件清除，只要清除 PENDing 位，该状态位自动清 0
SELECT 寄存器决定该中断是 IRQ 还是 FIQ，从而硬件会设置不同的 STATUS 寄存器

第四阶段，如何打开 CPSR I-bit ？
通过 汇编 MSR （SVC: 0xD3 1101->0101 0x53->CPSR)
__asm
{
mov  r0, #0x53
msr cpsr, r0
}

第五阶段，中断发生了之后怎么办？
接下来有2种处理办法：
A）简单的办法就是使用 VIC 向量中断控制器的功能
1. 跳转的地址向量要提前设置好
2. 通知 ARM11 内核，启用 VIC Port 功能
紧接着的问题是，如何在执行完 beep 之后返回主程序？
原因： beep 程序不能作为 IRQ_handler
```

真正的 IRQ_handler 应该要完成
1）保存cpu 现场 STMFD
2）清除掉 Pending bit，调用 beep
3）恢复cpu 现场 LDMFD

修改 start.s ，实现 IRQ_handler
1）IRQ 模式下的 sp 指针需要初始化
2）除了清除pending bit 之外，还需要清除 VIC0ADDRESS = 0;
3）返回地址 lr 寄存器需要 减4 即 sub lr, lr, #4
(lr-4)->PC SPSR->CPSR

B）复杂的办法就是不用 VIC ，自己实现全过程
1. 当 IRQ 异常发生的时候，cpu 跳转到 0x18
2. 背景知识：reset 0 地址被映射 map 到 iROM
0 地址 在 iROM 中 (0xD0000000)
iRAM (0xD0020000) -> 0x20000
0x18: 0xEA000018
最终在 iROM 中的程序(不可修改)会加载从 0xD0037400 地址开始的值，
作为异常发生后要跳转的地址+offset
3. (int)IRQ_handler -> 0xD0037400 + 0x18
如果是 SWI 软件中断，则在 0xD0037408 处填写swi_handler的地址

第 10 章

PWM Timer 定时器

10.1 定时器工作原理

功能：计时，中断，PWM Timer（驱动PWM信号的设备）
原理：类似于以前的 “沙漏”

```
沙漏的计时原理：沙子量，漏沙的速度，到时的反转(连续)
```

Timer：

```
沙子量： Counter的初值
漏沙速度：counter--（自动完成，并且依据Clock=PCLK=66Mhz)
反转：reload 操作（InitValue -> Counter)
装沙子：Manual update
```

真正的硬件设计是怎样的?

```
TCNTBn - 用来装沙子的量筒，可以修改
TCNTn - 真正用来做 counter--，不可修改，不可见
TCNTOn - 可以用来观察 TCNTn 的值的变化
```

10.2 定时器寄存器配置

```
Register  Address
TCFG0  0xE250_0000  65
PCLK = 66M  66000000 -> 分频
Timer Input Clock Frequency = PCLK / ( {prescaler value + 1} ) / {divider value}
{prescaler value} = 1~255
{divider value} = 1, 2, 4, 8, 16, TCLK
```

```
66M/66(65+1) = 1M = 1000000

TCFG1   0xE250_0004  0100=0x4
1M/16 = 62500

TCON   0xE250_0008
*[3] Timer 0 Auto Reload on/off
0 = One-Shot
1 = Interval Mode(Auto-Reload)
[2] Timer 0 Output Inverter on/off
0 = Inverter Off
1 = TOUT_0 Inverter-On
*[1] Timer 0 Manual Update
0 = No Operation
1 = Update TCNTB0,TCMPB0
*[0] Timer 0 Start/Stop
0 = Stop
1 = Start Timer 0

TCNTB0  0xE250_000C
val = 62500 (=1s)

TCMPB0  0xE250_0010

TCNTO0  0xE250_0014
read value -> TCNT0's value

TINT_CSTAT, R/W, Address = 0xE250_0044)
Interrupt Control and Status Register
Timer 0 Interrupt Status
[5]  Timer 0 Interrupt Status Bit.
Clears by writing ‘1’ on this bit.

Timer 0 interrupt Enable
[0]   Enables Timer 0 Interrupt.
1 = Enables
0 = Disabled
```

10.3 定时器驱动代码实现

```
// init Timer
// step 0: setup clock TCFG0+TCFG1
// step 1: TCNTBn <- init value
// setp 2: Set the manual update bit
//  and clear only manual update bit
// enable auto-reload bit
// step 3: Set the start bit
// step 4: Enable interrupt

while (1)
{
// polling TCNTO0, putint_hex()
```

```
// polling TINT_CSTAT bit[5] == 1 ?
beep();
}

// init VIC
Timer0 SRC = bit[21] belong to VIC0

Timer0 中断传递相关寄存器
VIC0(IRQ/FIQ)STATUS 标识中断是否通过(IRQ/FIQ)
VIC0INTSELECT 选择IRQ还是FIQ: bit21
VIC0INTENABLE 使能中断通过: bit21
VIC0RAWINTR 通过Timer0之后的情况 bit21
---------------------
TINT_CSTAT 中断(MSK) Enable bit [0]
TINT_CSTAT 中断(PND) Status bit [5]
TCNTB0 触发方式 1 sec interrupt
(TCFG0 + TCFG1) clock init
-GPD0CON 管脚复用 TOUT0
```