# i.MX.6ULL 裸机开发手册

深圳百问网科技有限公司

2022-9-15

[1.0]

# 更新记录

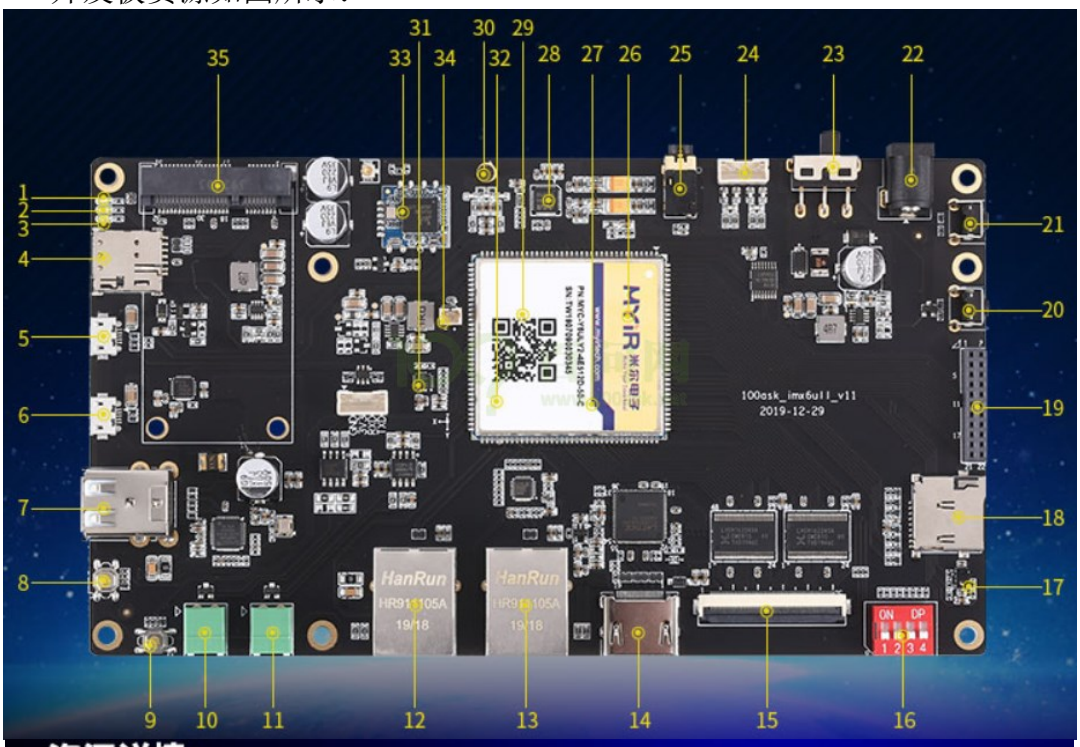
| 日期 | 更改内容 | 版本号 | 更新者 | 审核 |
| --- | --- | --- | --- | --- |
| 2022/9/15 | 初稿 | V1.0 | 百问网研发团队 | |
| | | | | |
| | | | | |

# 目录

# 第1章 硬件资源

## 1.1 板上资源

开发板资源如图所示：

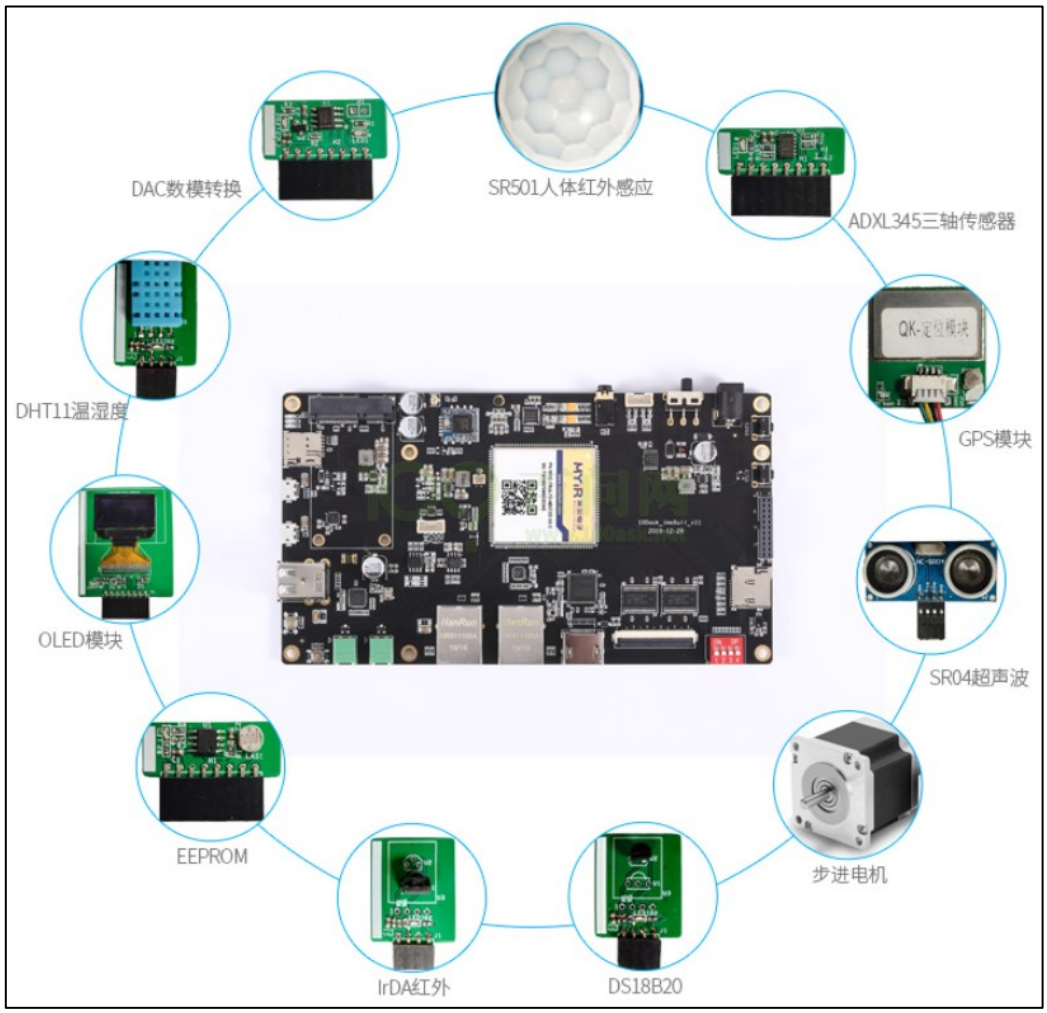



**资源详情：**

1：4G LED灯
2：用户LED灯
3：电源指示灯
4：nano SIM卡插座
5：USB OTG
6：USB 转串口
7：2路USB HOST
8：KEY 1
9：KEY2
10：RS485
11：CAN接口
12：以太网接口1(RJ45) 10M/100M
13：以太网接口2(RJ45) 10M/100M
14：板载**HDMI接口(可以接电视显示器)**
15：RGB TFT LCD接口
16：BOOT选择拨码开关
17：AP3216 三合一整合型光感测器
18：micro SD接口
19：Camera & 扩展GPIO
20：复位按键 KEY4
21：开/关机 KEY3
22：DC 6~12V电源输入
23：Power Switch
24：喇叭输出
25：4线耳麦接口
26：512MB DDR3L内存芯片
27：4GB eMMC flash
28：高性能音频编解码芯片WM8960
29：NXP IMX6ULL主控芯片（主频528MHz，具体型号请以实物为准）
30：MIC1(咪头)
31：六轴传感器（型号ICM-20608-G)
32：LAN PHY芯片
33：板载**WIFI**/蓝牙芯片(RTL8723)
34：RTC后备电池接口
35：mini PCIE 4G模块接口

## 1.2 板外模块

现有如下模块：



# 第2章 准备开发环境

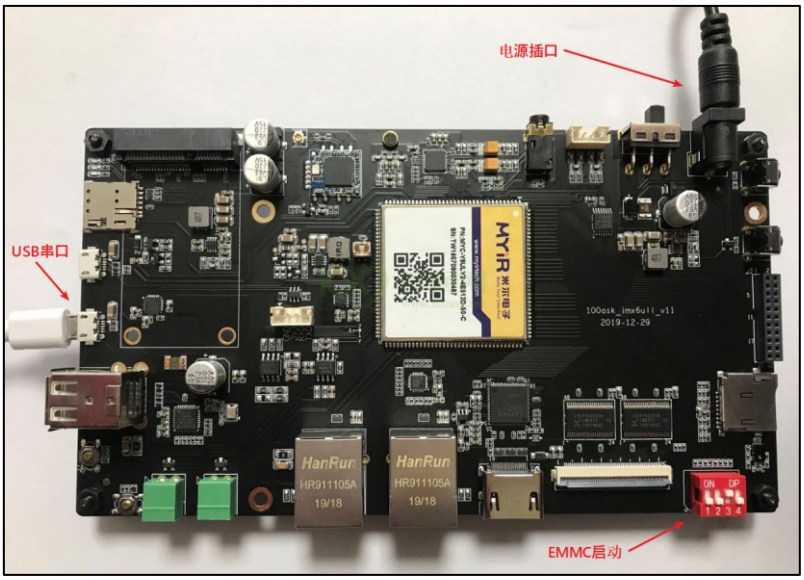
## 2.1 100ASK_IMX6ULL 开发板接线与启动

在后面的操作里，都是通过串口与板子进行“交流”。串口是串行接口的简称，是指数据一位一位地顺序传送，其特点是通信线路简单。

在电脑上安装好 MobaXterm 后，使 micro USB 数据线，连接电脑和开发板上的 6 号接口(USB 转串口)。

### 1. 连接串口线和电源线

首先如下图所示将串口线与电脑、板子连接，开发板插上电源：




其中特别需要注意的几点：

① 板子的“BOOT 选择拨码开关”要设置好，保证该启动方式里面有系统可以运行；

② 板子如图所示插上配套的电源到电源接口，电源开关暂时不用打开；

### 2. 安装 USB 串口驱动

接好 USB 串口线后，Windows 会自动安装驱动(安装可能比较慢，等一分钟左右)。打开电脑的“设备管理器”，在“端口 (COM 和 LPT)”项下，可以看到如下图中的“(COM21)”。这里的“COM21”可能与你电脑上的不一样，记住你电脑显示的数字。

如果电脑没有显示出端口号，就需要手动安装驱动，从驱动精灵官网（www.drivergenius.com）下载一个驱动精灵，安装、运行、检测，会自动安装上串口驱动。

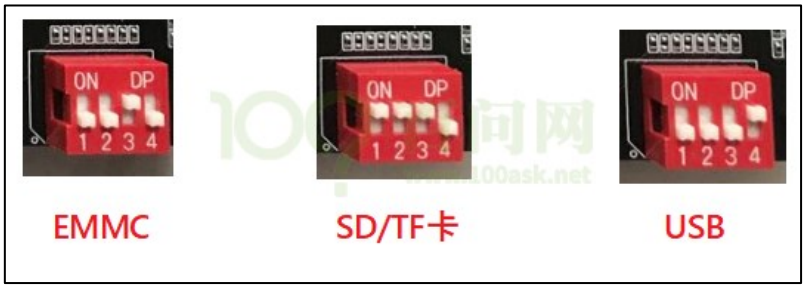
## 3．选择启动方式

板子上的红色拨码开关用来设置启动方式，支持这 3 种方式：EMMC 启动、SD 卡启动、USB 烧写。

板子背后画有一个表格，表示这 3 种方式如何设置。表格如下：

| BOOT CFG | | | | |
|---|---|---|---|---|
| BOOT | SW1(LCD_DATA5) | SW2(LCD_DATA11) | SW3(BOOT_MODE0) | SW4(BOOT_MODE1) |
| EMMC | OFF | OFF | ON | OFF |
| SD | ON | ON | ON | OFF |
| USB | X | X | OFF | ON |

这 3 种启动方式的设置示意图如下：



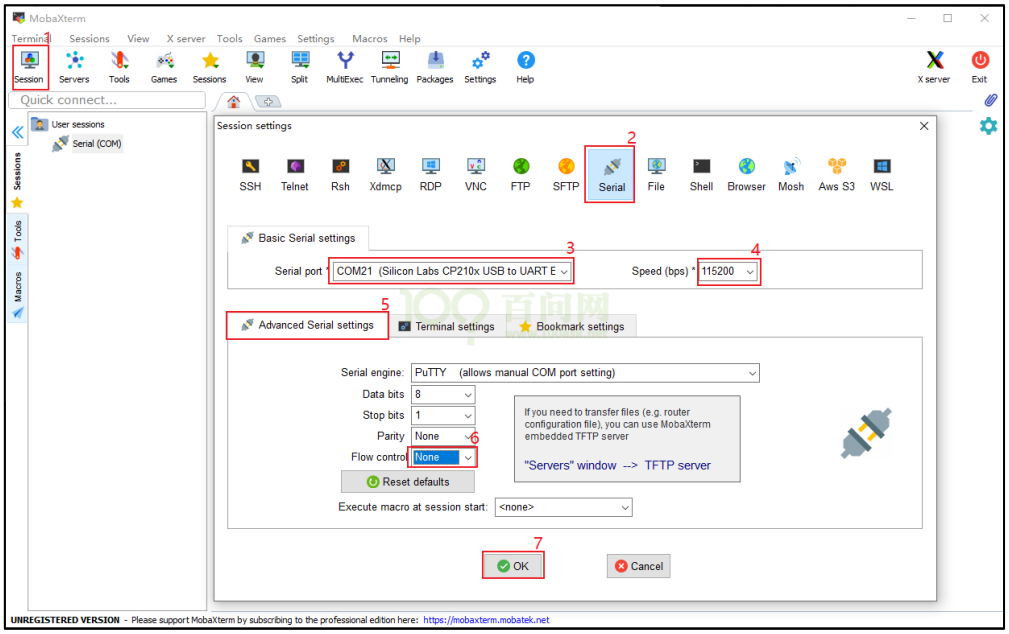

**要注意的是，设置为 USB 启动时，不能插上 SD 卡、TF 卡。**

刚出厂的板子在 EMMC 上烧写了系统，你可以设置为 EMMC 启动方式。

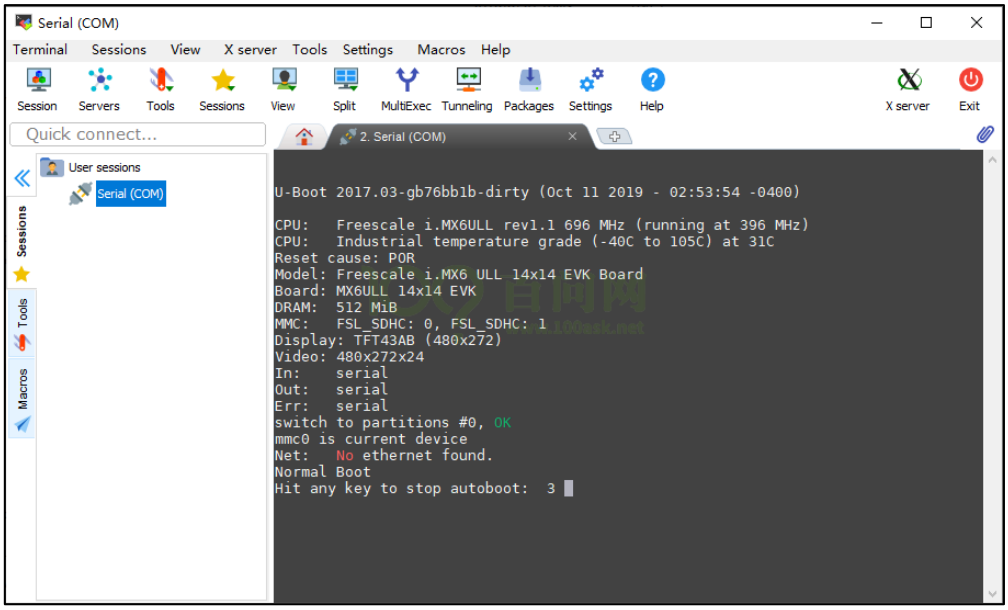
## 4．设置串口工具，启动开发板

打开 MobaXterm，点击左上角的“Session”，在弹出的界面选中“Serial”，如下图所示选择端口号（前面设备管理器显示的端口号 COM21）、波特率（Speed 115200）、流控（Flow Control: none），最后点击“OK”即可。

**注意：流控（Flow Control）一定要选择 none，否则你将无法在 MobaXterm 中向串口输入数据。**



随后显示一个黑色的窗口， 此时打开板子的电源开关，将收到板子串口发过来的数据，如下图所示：




## 2.2 安装 SDK、设置工具链

详见完全开发手册《嵌入式 Linux 应用开发完全手册 V5.1_IMX6ULL_Pro 开发板.pdf》第 3 篇第 1 章和第 2 章。

手册位于百度网盘资料 **02_100ask_imx6ull_pro_2022.08\01_学习手册**

**网盘连接：**https://pan.baidu.com/s/1HXaJC-3a2kEvlpd0wlxbiw?pwd=root

## 2.3 裸机源码

放在国内 GIT 仓库里：

https://e.coding.net/weidongshan/01_all_series_quickstart.git

使用 GIT 命令载后，在这个目录下：

**01_all_series_quickstart**

**10_裸机开发/01_100ASK_IMX6ULL 裸机程序**

里面有很多目录，比如“4_led”，它对应裸机文档的第 4 课。前 3 课没有源码，所以没有 1、2、3 开头的目录，有些章节也没有源码，就没有对应的目录。

## 2.4 配套资料和烧写工具

配套资料里含有芯片手册、原理图、烧写工具等等。放在百度网盘里，它比较大，无法放在 GIT。

网盘链接：

https://pan.baidu.com/s/1bFgKNm5616DPd8zb7zv5wQ?pwd=root

# 第3章 IMX6ULL 启动流程

## 3.1 IMX6ULL 启动方式

参考资料：

**02_100ask_imx6ull_pro_2022.08\05_芯片手册\02_Core_board(核心板)\CPUIMX6ULLRM.pdf”中《Chapter 8: System Boot》。**

### 3.1.1 芯片手册讲解

IMX6ULL 芯片内部有一个 boot ROM，上电后 boot ROM 上的程序就会运行。它会根据 BOOT_MODE[1:0]的值，以及 eFUSE 或 GPIO 的值决定后续的启动流程。

**注：eFUSE 即熔丝，只能烧写一次，一般正式发布产品时烧写最终值；平时调试时通过 GPIO 来设置开发板的启动方式。**

boot ROM 上的程序功能强大，可以从 USB 口或串口下载程序并把它烧写到 Flash 等设备上，也可以从 SD 卡或 EMMC、Flash 等设备上读出程序、运行程序。

问题来了：

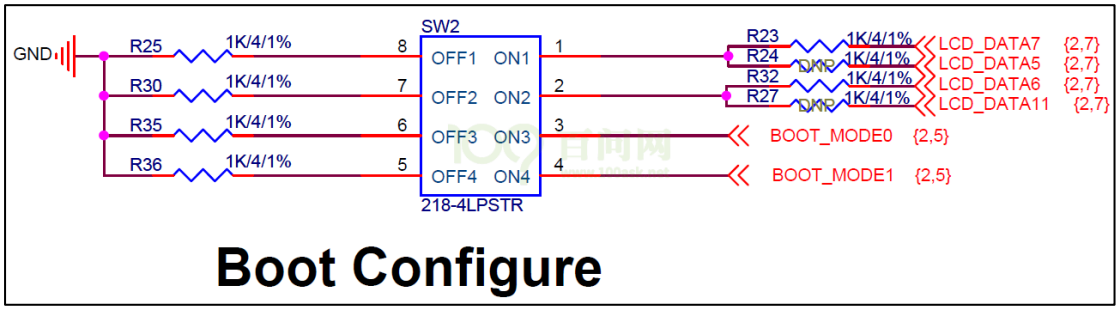
① boot ROM 是从 USB 口下载、运行程序，还是从 SD 卡等设备上读出、运行程序，谁决定？

由 BOOT_MODE[1:0]的值来决定启动方式，它们来自于 2 个引脚 BOOT_MODE1、BOOT_MODE0。这 2 个引脚在上电时是输入引脚，芯片启动后采集这 2 个引脚的值，存入 BOOT_MODE 寄存器。以后这 2 个引脚就可以用于其他功能，不会影响到 BOOT_MODE 寄存器。

BOOT_MODE[1:0]的值确定了 4 种启动模式，如下图：

| BOOT_MODE[1:0] | Boot Type | |
|---|---|---|
| 00 | Boot From Fuses | 根据eFUSE的值确定启动流程 |
| 01 | Serial Downloader | 通过USB或串口下载程序 |
| 10 | Internal Boot | 内部启动，即从SD卡、EMMC等设备启动 |
| 11 | Reserved | 保留 |

在 100ASK_IMX6ULL 中，这 2 个引脚对应的原理图如下：




00 模式在我们的开发过程中很少用到，简单介绍一下：在这种模式下，GPIO 的值被忽略。Boot ROM 会根据 eFUSE 的值来选择启动设备、设置启动设备。但是，对于刚出厂的芯片 eFUSE 值可能是错乱的、不适合你的设备的，怎么办？eFUSE 中有一个值 BT_FUSE_SEL，它的出厂值是 0，表示 eFUSE 未被烧写。boot ROM 程序发现 BT_FUSE_SEL 为 0 时，它会通过 USB 或串口来下载程序；发现 BT_FUSE_SEL 为 1 时，才会根据 eFUSE 的值选择启动设备，读出、运行该设备上的程序。

01 模式，boot ROM 程序通过 USB 或串口下载、运行程序，这个模式可以用来烧写 EMMC 等设备。我们的开发板出厂时，就是通过这个模式下载、烧写出厂程序的。

10 模式，称之为内部模式，简单地说就是从 SD 卡、EMMC 等设备启动程序。这就引入下面第 2 个问题。

② 如何选择启动设备？

00 模式下是通过 eFUSE 的值选择启动设备，我们不关心。

10 模式下既可以通过 eFUSE 的值也可以通过 GPIO 的值来选择启动设备，但是到底通过谁来决定？eFUSE 中有一个值 BT_FUSE_SEL，对，又是它。它的初始值为 0，表示 eFUSE 未被烧写。在 10 模式下，当 BT_FUSE_SEL 为 0 时就会通过 GPIO 来选择启动设备；当 BT_FUSE_SEL 为 1 时就会通过 eFUSE 来选择启动设备。

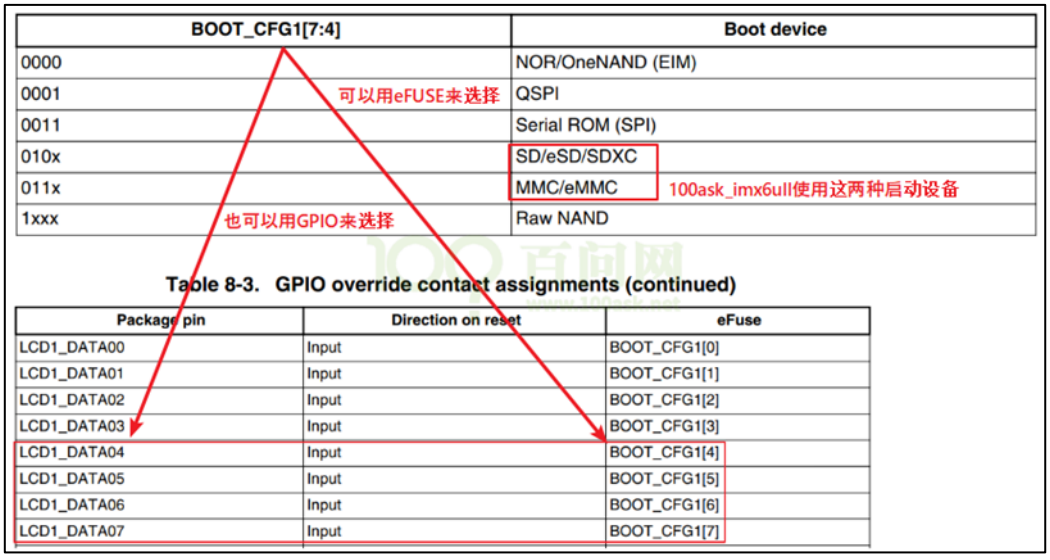
在开发阶段，我们使用 GPIO 来选择设备，这就引入下面第 3 个问题。

③ 如何通过 eFUSE 或 GPIO 选择、设置启动设备？

通过 eFUSE 或 GPIO 不仅能选择启动设备，还可以设置启动设备。

为什么还需要设置？比如 Nand Flash 参数各有不同，有些的页大小是 2048，有些是 4096。这些参数不同，boot ROM 程序读 Nand Flash 的方法就不同，我们必须把这些参数告诉 boot ROM：通过 eFUSE 或 GPIO 来标明这些参数。

首先看看要设置哪些 eFUSE 或 GPIO 来选择不同的启动设备。




| BOOT_CFG1[7:4] | Boot device |
|---|---|
| 0000 | NOR/OneNAND (EIM) |
| 0001 | QSPI |
| 0011 | Serial ROM (SPI) |
| 010x | SD/eSD/SDXC |
| 011x | MMC/eMMC |
| 1xxx | Raw NAND |

Table 8-3. GPIO override contact assignments (continued)

| Package pin | Direction on reset | eFuse |
|---|---|---|
| LCD1_DATA00 | Input | BOOT_CFG1[0] |
| LCD1_DATA01 | Input | BOOT_CFG1[1] |
| LCD1_DATA02 | Input | BOOT_CFG1[2] |
| LCD1_DATA03 | Input | BOOT_CFG1[3] |
| LCD1_DATA04 | Input | BOOT_CFG1[4] |
| LCD1_DATA05 | Input | BOOT_CFG1[5] |
| LCD1_DATA06 | Input | BOOT_CFG1[6] |
| LCD1_DATA07 | Input | BOOT_CFG1[7] |

从上图可知，既可以使用 eFUSE 也可以使用 GPIO 来选择启动设备，换句话说 GPIO 可以覆盖 eFUSE 的值。哪些 GPIO 覆盖哪些 eFUSE？这可以查看 IMX6ULL 芯片手册《Chapter 8: System Boot》里的《GPIO boot overrides》。

选择启动设备后，还需要标明一些参数。比如选择 EMMC 启动时，EMMC 接在哪一个接口，eSDHC1 还是 eSDHC2？它的速度如何？比如选择 TF 卡启动时，TF 卡接在哪一个接口，eSDHC1 还是 eSDHC2？它的速度如何？假设使用 EMMC 启动，或是 TF 卡启动，怎么设置 eFUSE 或 GPIO？这些信息可以查询 IMX6ULL 芯片手册《Chapter 5:Fusemap》。

- 当 BOOT_MODE 设置为 0b00 时，通过 eFUSE 选择启动设备，通过 eFUSE 获得设备的参数。

- 当 BOOT_MODE 设置为 0b10 时，通过 eFUSE 或 GPIO 来选择启动设备，获得设备的参数；使用 eFUSE 还是 GPIO 由 eFUSE 中的 BT_FUSE_SEL 决定，它默认是 0，表示使用 GPIO。

以 BOOT_MODE 为 0b10 为例，解析一下上图。要设置为 SD 卡、TF 卡启动，有 2 个设置方法：

a) 设置 eFUSE 的 BOOT_CFG1[7:5]为 0b010

b) 查看《3.1.3 GPIO boot overrides》确定 BOOT_CFG1[7:5]对应的 GPIO 为 LCD1_DATA07~05，把这 3 个引脚设置为 0b010。

根据 SD 卡、TF 卡的性能，可以设置 eFUSE 或 GPIO 来表示它能否提供更高的速度：

a) 设置 eFUSE 的 BOOT_CFG1[4:0]，或

b) 查看《3.1.3 GPIO boot overrides》确定 BOOT_CFG1[4:0]对应的 GPIO 为 LCD1_DATA04~00，设置这些引脚。

IMX6ULL 有两个 SD 卡、TF 卡接口，使用哪一个接口？请看下表：

c) 设置 eFUSE 的 BOOT_CFG2[4:3]可以确定使用 eSDHC1 或 eSDHC2，或

d) 查看《3.1.3 GPIO boot overrides》确定 BOOT_CFG2[4:3]对应的 GPIO 为 LCD1_DATA12~11，设置这些引脚

通过 eFUSE 或 GPIO，还可以标明启动设备的更多参数，具体细节可以参考芯片手册《Chapter 5: Fusemap》，作为硬件开发人员需要去细细研究；作为软件开发人员，实际上只需要看开发板手册知道怎么设置启动开关即可。

### 3.1.2 100ASK_IMX6ULL 启动方式选择

100ASK_IMX6ULL 开发板上的红色拨码开关用来设置启动方式、选择启动设备，支持这 3 种方式：EMMC 启动、SD 卡启动、USB 烧写。
板子背后画有一个表格，表示这 3 种方式如何设置。
表格如下：

| BOOT CFG | | | | |
|---|---|---|---|---|
| BOOT | SW1(LCD_DATA5) | SW2(LCD_DATA11) | SW3(BOOT_MODE0) | SW4(BOOT_MODE1) |
| EMMC | OFF | OFF | ON | OFF |
| SD | ON | ON | ON | OFF |
| USB | X | X | OFF | ON |

拨码开关中的 SW3、SW4 用来设置 BOOT_MODE，ON 表示 0，OFF 表示 1。
所以当 SW3、SW4 设置为 ON、OFF 时，BOOT_MODE 为 0b10，将会使用 SD 卡、TF 卡、EMMC 等设备启动。

刚出厂的开发板中 BT_FUSE_SEL 默认为 0，表示使用 GPIO 来设置参数。即使用 LCD1_DATA07~05 来选择启动设备。
100ASK_IMX6ULL 开发板只支持 SD/TF 卡、EMMC 启动，LCD1_DATA07~05 为 0b010 时选择 SD/TF 卡启动，LCD1_DATA07~05 为 0b011 时选择 EMMC 启动。这两种启动设备对应的 LCD1_DATA07~06 的值相同，都是 0b01，这在核心板上已经通过电阻设置好，我们只需要在拨码开关上设置 SW1(对应 LCD1_DATA05)就可以。

IMX6ULL 上有 2 个 EMMC Flash 接口，也复用为 2 个 SD/TF 卡接口，通过

LCD1_DATA12~11 来选择接口。0b00 对应 eSDHC1 接口，0b01 对应 eSDHC2 接口。LCD1_DATA12 的值在核心板上已经通过电阻设置好。LCD1_DATA11 的值通过拨码开关 SW2 来设置：ON 表示 0，对应 eSDHC1 接口，100ASK_IMX6ULL 的 TF 卡接口使用了 eSDHC1 接口；OFF 表示 1，对应 eSDHC2 接口，100ASK_IMX6ULL 的 EMMC 接口使用了 eSDHC2 接口。

### 3.1.3 GPIO boot overrides

IMX6ULL 中既可以通过 eFUSE 也可以通过 GPIO 来选择、设置启动设备，在手册里大部分场合只列出了 eFUSE，对应的 GPIO 需要查表：IMX6ULL 芯片手册《Chapter 8: System Boot》里的《GPIO boot overrides》。

## 3.2 IMX6ULL 启动流程

这个启动流程可以猜测出来，假设板子设置为 SD/TF 卡启动，boot ROM 程序会做什么？把程序从 SD/TF 卡读出来，运行。

从哪里读？从 SD/TF 卡读，这需要先初始化 SD/TF 卡：根据 eFUSE 或 GPIO 的设置初始化 SD/TF 卡。

读到哪里去？读到内存即 DDR 去，这需要先初始化 DDR。

除了初始化启动设备、初始化 DDR，还需要初始化什么？也许要初始化时钟，让 CPU 跑得更快一点。

总结起来就是：初始化 CPU、时钟等，初始化内存，初始化启动设备，从启动设备上把程序读入内存，运行内存的程序。

官方的启动流程如下：

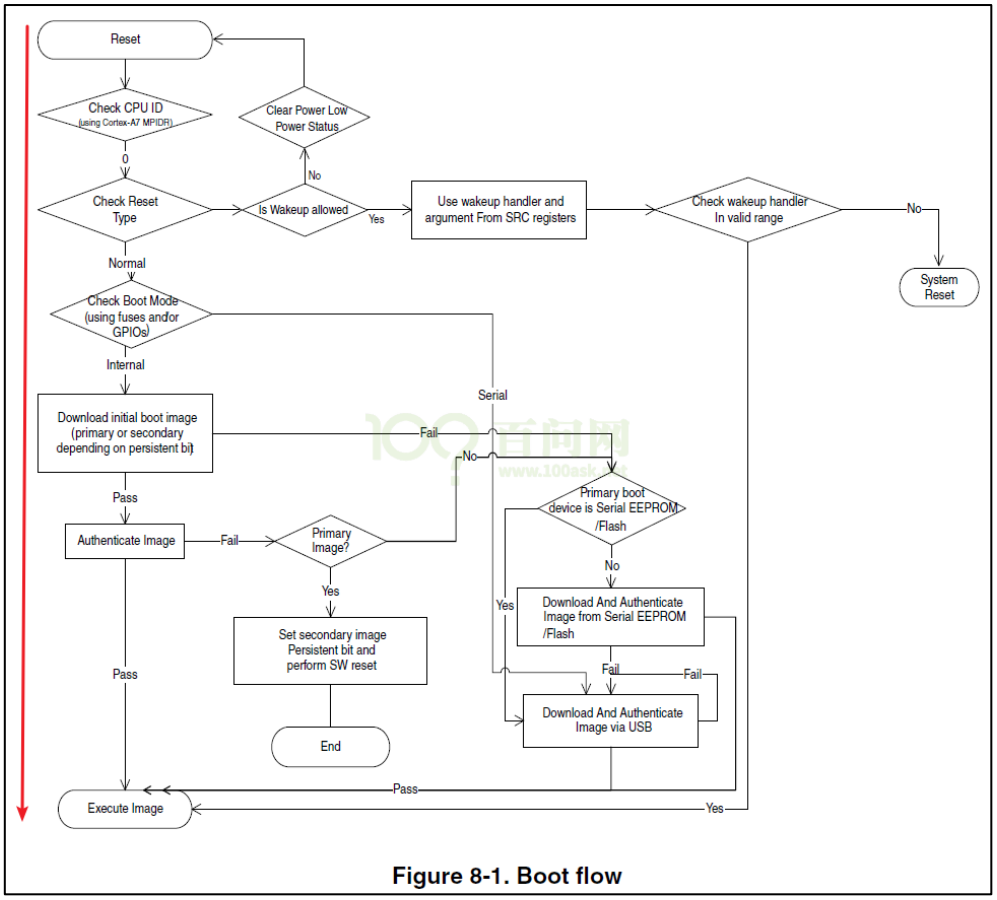



**Figure 8-1. Boot flow**

这个流程图比较粗糙，总结起来就是：

a) 检查 CPU ID

b) 检查 Reset Type，冷启动、唤醒的启动过程是不一样的

c) 检查启动模式 BOOT_MODE，检查 eFUSE 或 GPIO

d) 根据上述检查从 USB 口、UART 口或是某个启动设备下载 boot image

e) 认证 image

f) 启动

对于具体的启动设备，IMX6ULL 芯片手册《Chapter 8: System Boot》中有对应章节描述更为细致的启动流程。基本上就是对这些启动设备根据 eFUSE 或 GPIO 的设置进行初始化，尝试更高的工作频率等。在往后的学习中，如果涉及这些细节，我们再描述。

## 3.3 IMX6ULL 映像文件

假设使用 SD/TF 卡启动，卡上的程序有多大？它应该被复制到 DDR 哪里去？

### 3.3.1 格式概述

如果您有 S3C2440 或其他单片机的学习经验，可以知道程序的二进制版本，比如 lcd.bin 可以直接烧写到 Flash 上。它们是自启动的，什么意思？比如一上电，运行的是 lcd.bin 前面的代码，它会初始化内存，把自己从 Flash 上复制到内存里去执行。请记住：自己把自己复制到内存。

但是对于 IMX6ULL，烧写在 EMMC、SD/TF 卡上的程序，并不能“自己复制自己”，是“别人把它复制到内存里”。一上电首先运行的是 boot ROM 上的程序，它从 EMMC、SD/TF 卡上把程序复制进内存里。

所以：boot ROM 程序需要知道从启动设备哪个位置读程序，读多大的程序，复制到哪里去。

所以：启动设备上，不能仅仅烧写 bin 文件，需要在添加额外的信息。

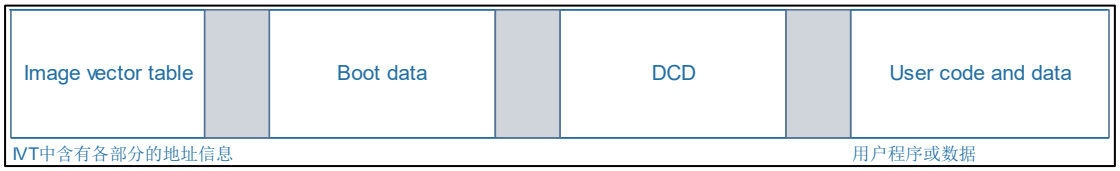
还有一个问题，IMX6ULL 的 boot ROM 程序可以把程序读到 DDR 里，那需要先初始化 DDR。每种板子接的 DDR 可能不一样，boot ROM 程序需要初始化这些不同的 DDR。boot ROM 从哪里得到这些不同的参数？

还有，IMX6ULL 支持各种启动设备，比如各种 Nor Flash。为了通用，boot ROM 程序将会使用最保守的参数，也就是最慢的时序来访问 Nor Flash。为加快启动程序，boot ROM 程序可以根据我们提供的信息初始化硬件，让它以更优的参数运行。

这些参数信息，被称为“Device Configuration Data”，设备配置数据(DCD)，这些 DCD 将会跟 bin 文件一起打包烧写在启动设备上。boot ROM 程序会从启动设备上读出 DCD 数据，根据 DCD 来写对应的寄存器以便初始化芯片。DCD 中列出的是对某些寄存器的读写操作，我们可以在 DCD 中设置 DDR 控制器的寄存器值，可以在 DCD 中使用更优的参数设置必需的硬件。这样 boot ROM 程序就会帮我们初始化 DDR 和其他硬件，然后才可以把 bin 程序读到 DDR 中并运行。

总结起来，烧写在 EMMC、SD 卡或是 TF 卡上的，除了程序本身，还有位置

信息、DCD 信息，这些内容合并成一个映像文件，如下图：




这 4 部分内容合并成为一个映像文件，烧写在 EMMC、SD 卡或 TF 卡等启动设备的某个固定地址，boot ROM 程序去这个固定地址读出映像文件。启动设备不同，固定地址不同，如下图：

**Table 8-25. Image Vector Table Offset and Initial Load Region Size**

| Boot Device Type | Image Vector Table Offset 映像文件在哪？ | Initial Load Region Size |
|---|---|---|
| NOR | 4 Kbyte = 0x1000 bytes | Entire Image Size boot ROM初始时读多少 |
| OneNAND | 256 bytes = 0x100 bytes | 1 Kbyte |
| SD/MMC/eSD/eMMC/SDXC | 1 Kbyte = 0x400 bytes | 4 Kbyte |
| SPI EEPROM | 1 Kbyte = 0x400 bytes | 4 Kbyte |

## 3.3.2 格式详解

先贴出一张图，然后再细细讲解：

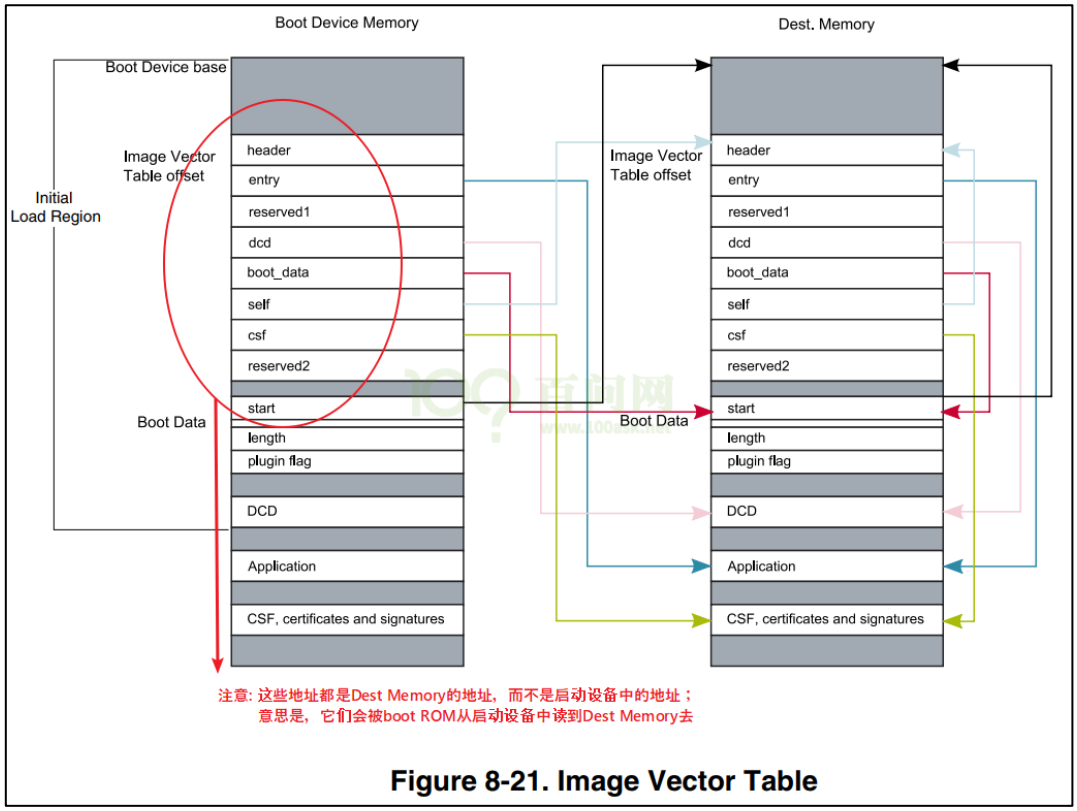



**Figure 8-21. Image Vector Table**

下面的讲解图中，列有 C 语言格式的结构体，这些结构体来源于 U-boot 的 tools 目录下的 imximage.h。对于程序员，有时候看结构体可以更快地理解映像文件的格式。

**1. Image Vector Table(IVT):**

IVT 会被放在固定的地址，IVT 中是一系列的地址，boot ROM 程序会根据这些地址来确定映像文件中其他部分在哪里。

IVT 格式如下：

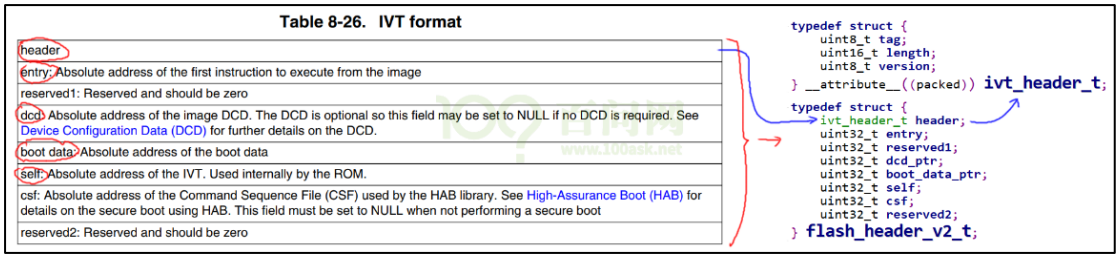


Table 8-26. IVT format

| |
|---|
| header |
| entry: Absolute address of the first instruction to execute from the image |
| reserved1: Reserved and should be zero |
| dcd: Absolute address of the image DCD. The DCD is optional so this field may be set to NULL if no DCD is required. See Device Configuration Data (DCD) for further details on the DCD. |
| boot data: Absolute address of the boot data |
| self: Absolute address of the IVT. Used internally by the ROM. |
| csf: Absolute address of the Command Sequence File (CSF) used by the HAB library. See High-Assurance Boot (HAB) for details on the secure boot using HAB. This field must be set to NULL when not performing a secure boot |
| reserved2: Reserved and should be zero |

```
typedef struct {
    uint8_t tag;
    uint16_t length;
    uint8_t version;
} __attribute__((packed)) ivt_header_t;
typedef struct {
    ivt_header_t header;
    uint32_t entry;
    uint32_t reserved1;
    uint32_t dcd_ptr;
    uint32_t boot_data_ptr;
    uint32_t self;
    uint32_t csf;
    uint32_t reserved2;
} flash_header_v2_t;
```

要注意的是上图中这 4 项：

a) header：里面有 3 项：tag、length、version。length 表示 IVT 的大小，它是 32 字节。要注意是的，它是大字节序的。

b) entry：用户程序运行时第 1 条指令的地址，就是程序的链接地址、程序被复制到内存哪里。

c) dcd：映像被复制到内存后，其中的 DCD 数据的地址。

d) boot data：映像被复制到内存后，其中的 boot data 的地址。

e) self：映像被复制到内存后，IVT 自己所在的地址。

**2.** Boot data：

映像被复制到内存后，整个映像文件(IVT 之前还有几个扇区数据，比如分区表)所在的地址。

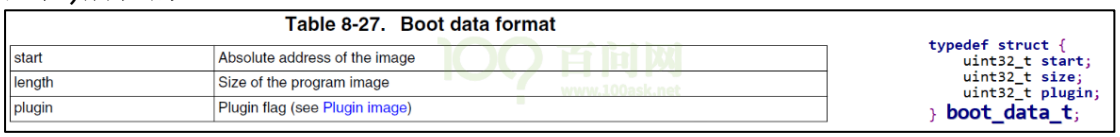


Table 8-27. Boot data format

| | |
|---|---|
| start | Absolute address of the image |
| length | Size of the program image |
| plugin | Plugin flag (see Plugin image) |

```
typedef struct {
    uint32_t start;
    uint32_t size;
    uint32_t plugin;
} boot_data_t;
```

a) start：这是映像文件在内存中的地址，以 SD/TF 卡为例：

映像文件=(1K 数据,内含分区表等信息)+IVT+BootData+DCD+用户数据(bin 文件)

注意，IVT 并不在映像文件的最前面，start 也不是 IVT 在内存中的地址，而是整个映像文件在内存中的地址：

start = IVT 在内存中的地址 - IVT offset

什么意思？假设 IVT 被保存在启动设备 TF 卡 1024 偏移地址处，IVT 被复制到内存地址 0x87000000，那么 start=0x87000000-1024。

所以 start 表示的是启动设备开头的数据，被复制到内存哪里去。

从它的含义也可以推理出：boot ROM 程序会把启动设备开头的数据，复制到内存；而不仅仅是从 IVT 开始复制。

b) length：保存在启动设备上的整个映像文件的长度，从 0 地址开始(不是从 IVT 开始)。

c) plugin：这是一个标记位，当它为 1 时表示这个映像文件是“plugin”，即插件。

boot ROM 程序可以支持有限的启动设备，如果你想双持更多的启动设备比如网络启动、CDROM 启动，就需要提供对应的驱动。这些驱动就是“plugin”，我们的教程不涉及，该标记位为 0。

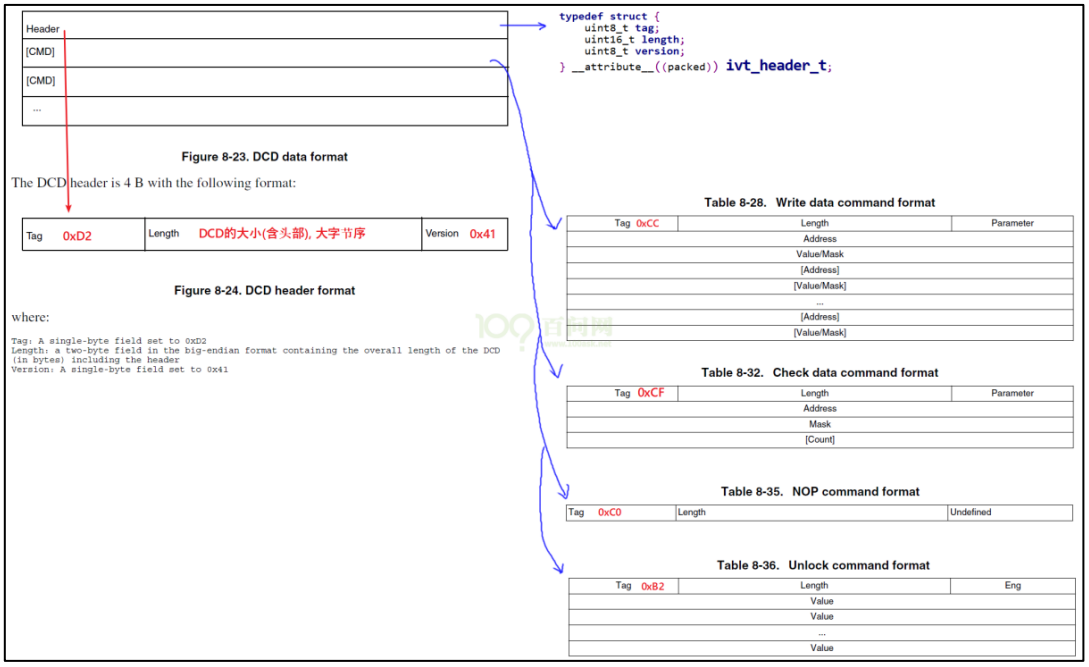
Boot data 就是用来表示映像文件应该被复制到哪里去，以前它的大小。boot ROM 程序就是根据它来把整个映像文件复制到内存去的。

## 3. DCD

DCD 的作用在前面讲解过，简单地说就是设备的配置信息，里面保存有一些寄存器值。

实际上 DCD 还可以更复杂，它支持多种命令：write data、check data、nop、unlock。我们可以通过 write data 命令写寄存器，通过 check data 命令等待寄存器就绪。

- DCD 格式如下：




DCD 以 Header 开始，里面的 TAG 为 0xD2 表示它是 DCD，里面还标明了 DCD 的大小、版本。

接下来就是各个“CMD”，你可以在一个“CMD”里操作多个寄存器，比如在一个“write data command”中，写多个寄存器。

以“write data command”为例简单介绍一下，它的格式为：

**Table 8-28. Write data command format**

| Tag 0xCC | Length | Parameter |
|---|---|---|
| Address | | |
| Value/Mask | | |
| [Address] | | |
| [Value/Mask] | | |
| ... | | |
| [Address] | | |
| [Value/Mask] | | |

上图中，TAG 为 0xCC 表示这是“write data command”；Length 表示命令的大小；Parameter 的作用稍后再说。

既然是写命令，那自然就有“地址、值”，上图中就是多个“Address、Value/Mask”。为何还有 Mask？这要结合 Parameter 来讲解：

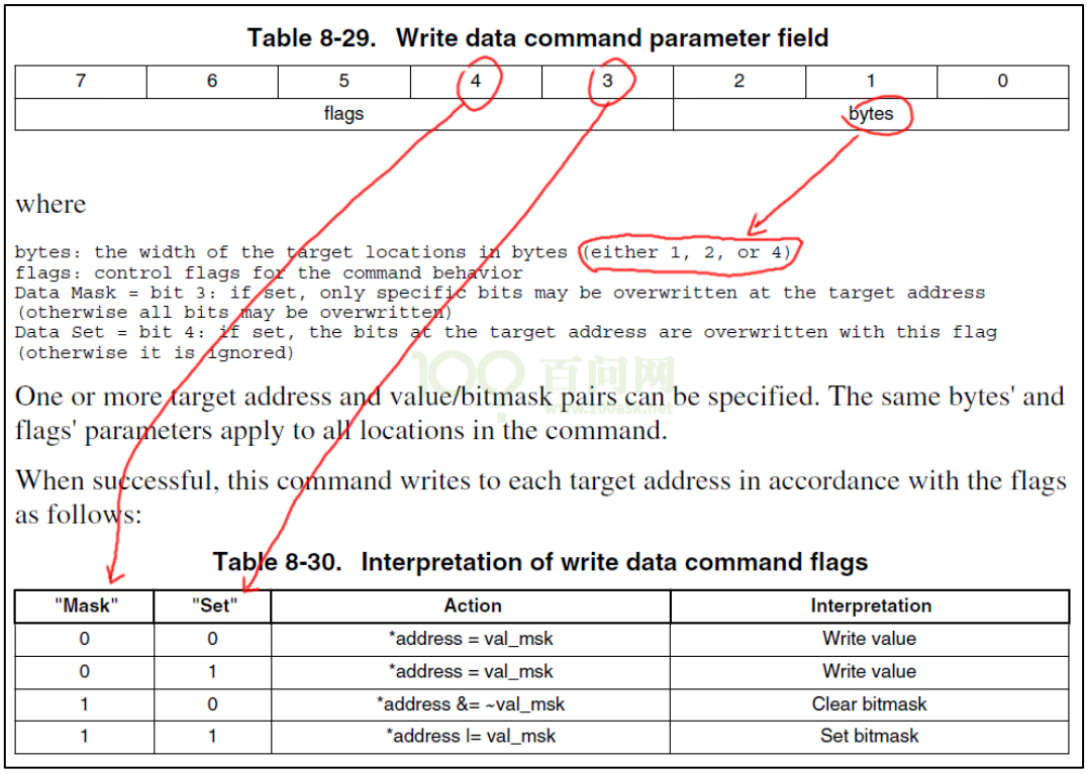


Table 8-29. Write data command parameter field

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| flags | | | | | bytes | | |

where

bytes: the width of the target locations in bytes (either 1, 2, or 4)
flags: control flags for the command behavior
Data Mask = bit 3: if set, only specific bits may be overwritten at the target address (otherwise all bits may be overwritten)
Data Set = bit 4: if set, the bits at the target address are overwritten with this flag (otherwise it is ignored)

One or more target address and value/bitmask pairs can be specified. The same bytes' and flags' parameters apply to all locations in the command.

When successful, this command writes to each target address in accordance with the flags as follows:

Table 8-30. Interpretation of write data command flags

| "Mask" | "Set" | Action | Interpretation |
|---|---|---|---|
| 0 | 0 | *address = val_msk | Write value |
| 0 | 1 | *address = val_msk | Write value |
| 1 | 0 | *address &= ~val_msk | Clear bitmask |
| 1 | 1 | *address \|= val_msk | Set bitmask |

Parameter 中 b[2:0]用来表示写操作的字节数，是以字节、半字(2 byte)，还是字(4 byte)来操作。而 b[4]、b[3]决定了是写值(write value)，清位(clear bitmask)，还是设位(set bitmask)。

对于其他命令，共格式可以参考 IMX6UL 的芯片手册，这里就不再介绍了。

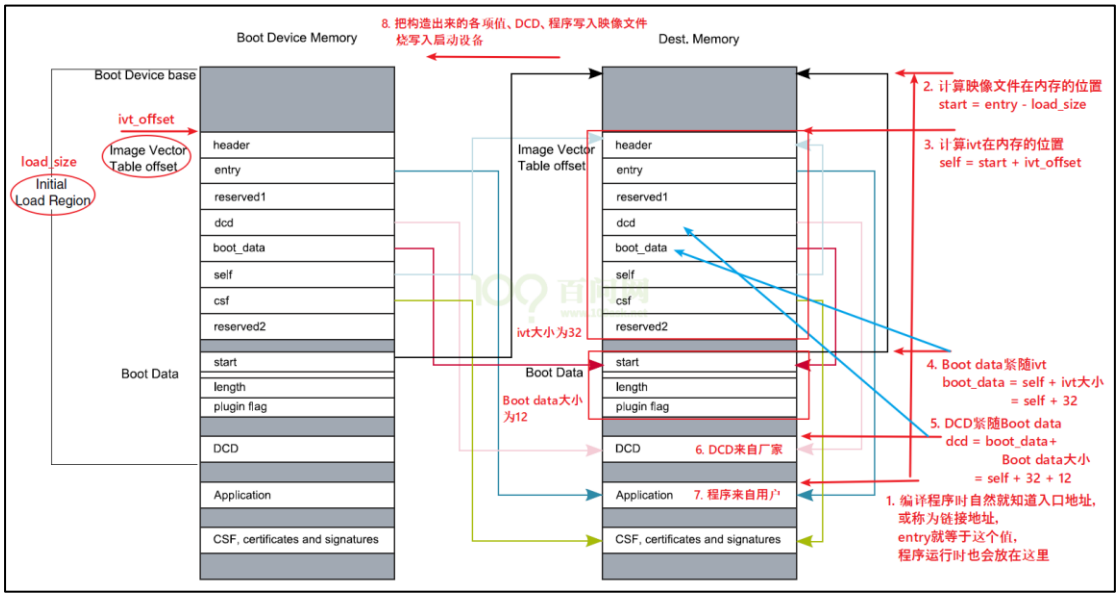
### 4. User code and data：

就是用户程序或数据，原原本本地添加到映像文件里就可以。

### 3.3.3 实例

我们制作映像文件的目的什么？把我们自己的程序烧写到启动设备，让 boot ROM 程序启动它。

所以制作映像文件的起点是：我们编写的程序。制作过程中各项值的计算方法如下图所示：




我们不需要手工去计算，一个 mkimage 命令就搞定了。上图中各步骤细说如下：

① 确定入口地址 entry：

我们的程序运行时要放在内存中哪一个位置，这是我们决定的。它被称为入口地址、链接地址。

② 确定映像文件在内存中的地址 start：

boot ROM 程序启动时，会把“Initial Load Region”读出来，“Initial load Region”里含有 IVT、Boot data、DCD。boot ROM 根据 DCD 初始化设备后，再把整个映像文件读到内存。

在启动设备上，“Initial Load Region”之后紧跟着我们的程序，反过来说就是我们程序的前面，放着“Initial Load Region”。假设“Initial Load Region”的大小为 load_size，那么在内存中“Initial Load Region”的位置 start = entry – load_size。

注意：“Initial Load Region”位于启动设备 0 位置，它的头部并不是 IVT，而是一些无用的数据(或是分区信息)。

在 IMX6ULL 中有一个表格，列出了不同启动设备对应的“Initial Load Region Size”：

**Table 8-25. Image Vector Table Offset and Initial Load Region Size**

| Boot Device Type | Image Vector Table Offset 映像文件在哪？ | Initial Load Region Size boot ROM初始时读多少 |
|---|---|---|
| NOR | 4 Kbyte = 0x1000 bytes | Entire Image Size |
| OneNAND | 256 bytes = 0x100 bytes | 1 Kbyte |
| SD/MMC/eSD/eMMC/SDXC | 1 Kbyte = 0x400 bytes | 4 Kbyte |
| SPI EEPROM | 1 Kbyte = 0x400 bytes | 4 Kbyte |

③ 确定 IVT 在内存中的地址 self：

我们知道 IVT 在启动设备上某个固定的位置，上或中的“Image Vector Table Offset”：ivt_offset。那么在内存中它的位置可以如下计算：

```
self = start + ivt_offset = entry – load_size + ivt_offset
```

④ 确定 Boot data 在内存中的地址 boot_data：

IVT 的大小是 32 字节，IVT 之后就是 Boot data，而 IVT 中的 boot_data 值表示 Boot data 在内存中的位置，计算如下：

```
boot_data = self + 32 = entry – load_size + ivt_offset + 32
```

⑤ 确定 DCD 在内存中的地址 dcd：

Boot data 的大小是 12 字节，Boot data 之后就是 DCD，而 IVT 中的 dcd 值表示 DCD 在内存中的位置，计算如下：

```
dcd = boot_data + 12 = entry – load_size + ivt_offset + 44
```

⑥ 写入 DCD 的数据：

DCD 是用初始化硬件的，特别是初始化 DDR。而 DDR 的初始化非常的复杂、专业，我们一般是使用硬件厂家提供的代码。

在后面的程序中你可以看到，我们是使用类似下面的指令来制作映象文件：

```
./tools/mkimage -n ./tools/imximage.cfg.cfgtmp -T imximage -e 0x80100000 -d led.bin led.imx
```

上述命令中的 imximage.cfg.cfgtmp 就是厂家提供的，内部截取部分贴出来：

```
IMAGE_VERSION 2
# 30 "board/freescale/mx6ullevk/imximage.cfg"
BOOT_FROM sd
# 55 "board/freescale/mx6ullevk/imximage.cfg"
DATA 4 0x020c4068 0xffffffff
DATA 4 0x020c406c 0xffffffff
DATA 4 0x020c4070 0xffffffff
DATA 4 0x020c4074 0xffffffff
DATA 4 0x020c4078 0xffffffff
DATA 4 0x020c407c 0xffffffff
DATA 4 0x020c4080 0xffffffff
```

从上图也可以看到 imximage.cfg.cfgtmp 文件中基本是对寄存器的写操作。mkimage 程序来自 u-boot，它会把 imximage.cfg.cfgtmp 中的内容转换为 DCD 数据。我们并不打算讲解 DCD 的内容，只需要了解它的大概作用：

a) 设置时钟：DDR 也需要时钟，这很好理解

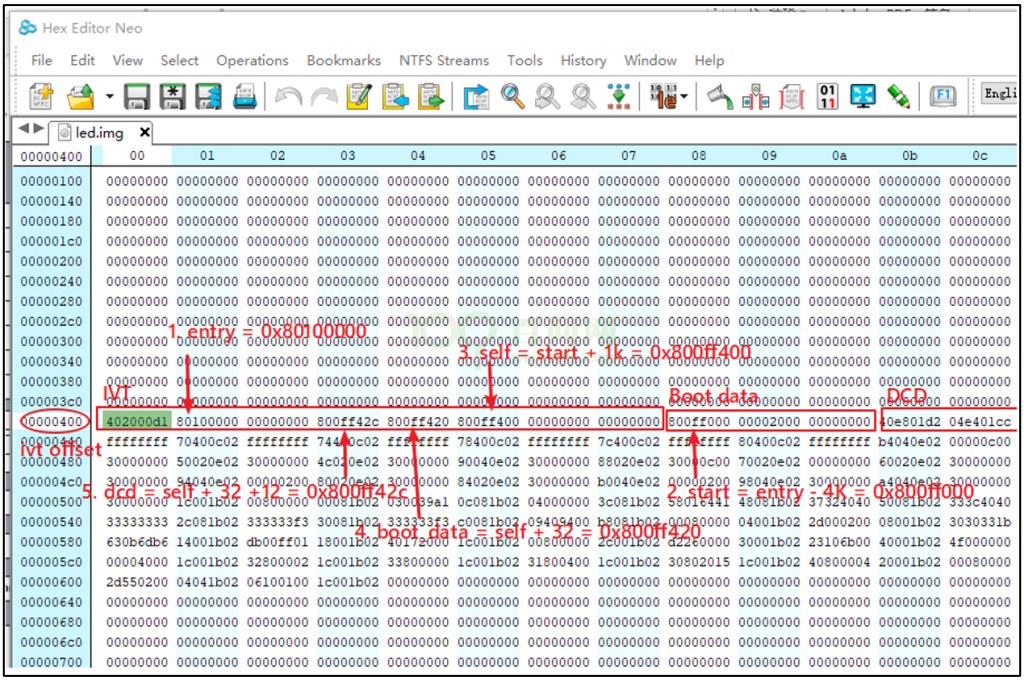
b) 设置引脚：DDR 需要很多引脚

c) 设置 DDR 控制器：Multi-mode DDR controller (MMDC)

⑦ 写入用户程序

⑧ 经过上述 7 个步骤，整个映像文件就构造出来了，可以把它烧入启动设备。

我们提供的示例程序 4_led 中有一个文件：led.img，它就是映象文件，可以直接烧入 TF 卡。用软件 Hex Editor Neo 打开 led.img，选择 doble word 方式显示，你可以自行验证一下映像文件中各个值。




## 3.4 映像文件烧写、运行

我们编译出来的映像文件有 2 类后缀：imx、img。imx 文件开头就是 IVT，可以把它烧写到 TF 卡 1024 偏移位置处；img 文件开头是 1024 字节的 0 值数据，后面才是 IVT 等，它可以直接烧写到 TF 卡 0 偏移位置处。

另外，我们还可以通过 USB 把 imx 文件直接下载到板子上，并运行。

注意：通过 USB 下载方式，可以烧写程序到 EMMC、TF 卡上，但是并非“直接烧写”。它的过程如下：

a. 通过 USB 下载 u-boot 到内存，
b. 通过 USB 下载用户程序到内存，
c. 通过 USB 发送命令运行 u-boot，
d. 用 u-boot 烧写把内存中的用户程序烧写到 EMMC、TF 卡上

我们制作的烧写工具 100ask_imx6ull_flashing_tool，可以一键实现上述过程，非常方便。

### 3.4.1 使用 USB 运行裸机程序

使用 USB 来运行裸机程序，是最简单的方法，不需要烧写。步骤如下：

① 开发板设置为 USB 启动；
② 使用 USB 线连接电脑和开发板的 OTG 口。接好线后上电。
③ 运行 100ask_imx6ull_flashing_tool：

如果不成功，请确认：

a) 开发板的启动开关是否设置为 USB 模式

b) 开发板不要插上 TF 卡

c) 开发板复位一下，确定烧写工具上的“设备已连接”的按钮变绿

d) 要下载运行的是 imx 文件，不是 bin 文件，也不是 img 文件

### 3.4.2 使用 USB 烧写裸机程序到 SD/TF

步骤如下：

① 开发板设置为 USB 启动；

② 使用 USB 线连接电脑和开发板的 OTG 口；

③ 运行 100ask_imx6ull_flashing_tool：

如果不成功，请确认：

a) 开发板的启动开关是否设置为 USB 模式

b) 开发板不要插上 TF 卡

c) 开发板复位一下，确定烧写工具上的“设备已连接”的按钮变绿，这时再插 SD/TF 卡

d) 要下载运行的是 imx 文件，不是 bin 文件，也不是 img 文件

e) 烧写成功后，开发板断电，设置为 SD/TF 启动，再重新上电观察效果

### 3.4.3 使用 USB 烧写裸机程序到 EMMC

步骤如下：

① 开发板设置为 USB 启动：

② 使用 USB 线连接电脑和开发板的 OTG 口：

③ 运行 100ask_imx6ull_flashing_tool：

如果不成功，请确认：

a) 开发板的启动开关是否设置为 USB 模式

b) 开发板不要插上 TF 卡

c) 开发板复位一下，确定烧写工具上的“设备已连接”的按钮变绿

d) 要下载运行的是 imx 文件，不是 bin 文件，也不是 img 文件

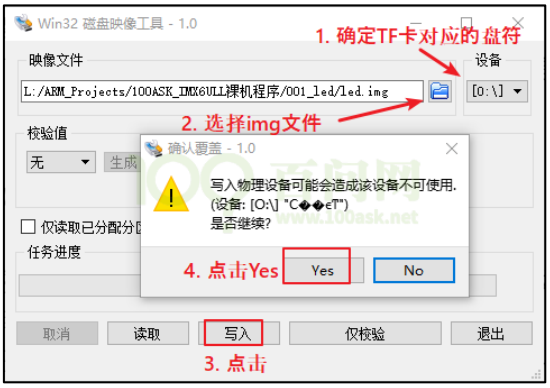
e) 烧写成功后，开发板断电，设置为 EMMC 启动，再重新上电观察效果

### 3.4.4 使用读卡器烧写裸机程序到 SD/TF 卡

这需要借助读卡器，在电脑上烧写 TF 卡。步骤如下：

a) 烧写 TF 卡：

把 TF 卡通过读卡器接到电脑上，使用 win32diskimager 把 img 文件烧写到 SD 卡上，如下图所示操作，烧写成功后会有提示：



b) 启动开发板：

把烧写好的 TF 卡插到开发板，开发板设置为 SD/TF 启动模式(如下)，上电即可。

**注意：使用 win32diskimager 烧写时，一定要选择 img 文件，不能选择 imx 文件。**

**注意：烧写、运行 led 程序时，串口是没有输出的，这只是一个点灯程序没用到串口。**

# 第4章 LED 程序

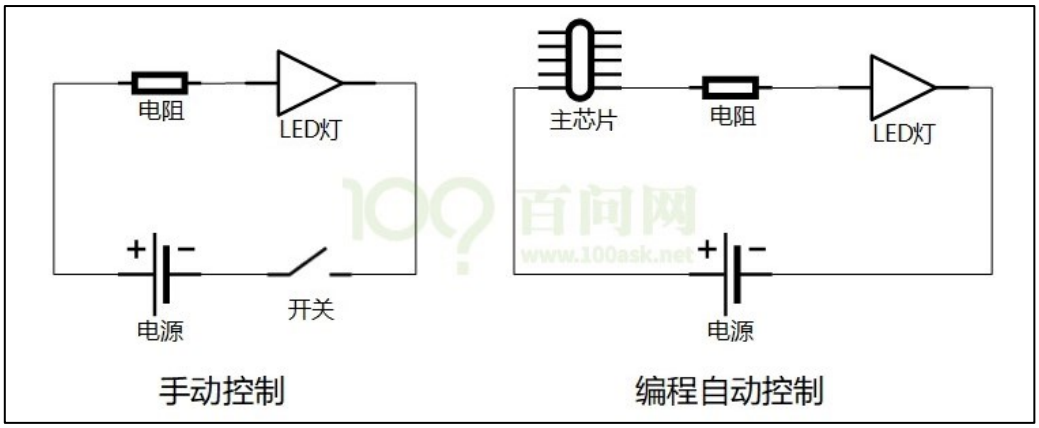
## 4.1 硬件知识_LED 原理图

当我们学习 C 语言的时候，我们会写个 Hello 程序。那当我们写 ARM 程序，也该有一个简单的程序引领我们入门，这个程序就是点亮 LED。我们怎样去点亮一个 LED 呢？分为三步：

① 看原理图，确定控制 LED 的引脚；
② 看主芯片的芯片手册，确定如何设置控制这个引脚；
③ 写程序。

LED 样子有很多种，像插脚的，贴片的。它们长得完全不一样，因此我们在原理图中将它抽象出来。

点亮 LED 需要通电源，同时为了保护 LED，加个电阻减小电流。控制 LED 灯的亮灭，可以手动开关 LED，但在电子系统中，不可能让人来控制开关，通过编程，利用芯片的引脚去控制开关。




LED 的驱动方式，常见的有四种。

① 使用引脚输出 3.3V 点亮 LED，输出 0V 熄灭 LED。
② 使用引脚拉低到 0V 点亮 LED，输出 3.3V 熄灭 LED。

有的芯片为了省电等原因，其引脚驱动能力不足，这时可以使用三极管驱动。

③ 使用引脚输出 1.2V 点亮 LED，输出 0V 熄灭 LED。
④ 使用引脚输出 0V 点亮 LED，输出 1.2V 熄灭 LED。

由此，主芯片引脚输出高电平/低电平，即可改变 LED 状态，而无需关注 GPIO 引脚输出的是 3.3V 还是 1.2V。所以简称输出 1 或 0：

- 逻辑 1-->高电平
- 逻辑 0-->低电平

## 4.2 普适的 GPIO 引脚操作方法

GPIO: General-purpose input/output，通用的输入输出口。

### 4.2.1 GPIO 模块一般结构

a) 有多组 GPIO，每组有多个 GPIO
b) 使能：电源/时钟

c) 模式(Mode)：引脚可用于 GPIO 或其他功能

d) 方向：引脚 Mode 设置为 GPIO 时，可以继续设置它是输出引脚，还是输入引脚

e) 数值：对于输出引脚，可以设置寄存器让它输出高、低电平

对于输入引脚，可以读取寄存器得到引脚的当前电平

### 4.2.2 GPIO 寄存器操作

a) 芯片手册一般有相关章节，用来介绍：power/clock

可以设置对应寄存器使能某个 GPIO 模块(Module)
有些芯片的 GPIO 是没有使能开关的，即它总是使能的

b) 一个引脚可以用于 GPIO、串口、USB 或其他功能，有对应的寄存器来选择引脚的功能

c) 对于已经设置为 GPIO 功能的引脚，有方向寄存器用来设置它的方向：输出、输入

d) 对于已经设置为 GPIO 功能的引脚，有数据寄存器用来写、读引脚电平状态

GPIO 寄存器的 2 种操作方法：原则：不能影响到其他位

**1.** 直接读写：读出、修改对应位、写入

要设置 bit n：

```
val = data_reg;
val = val | (1<<n);
data_reg = val;
```

要清除 bit n：

```
val = data_reg;
val = val & ~(1<<n);
data_reg = val;
```

**2.** set-and-clear protocol：

`set_reg, clr_reg, data_reg` 三个寄存器对应的是同一个物理寄存器，

要设置 bit n：`set_reg = (1<<n);`

要清除 bit n：`clr_reg = (1<<n);`

### 4.2.3 IMX6ULL 的 GPIO 操作方法

```
CCM: Clock Controller Module (时钟控制模块)
IOMUXC : IOMUX Controller，IO 复用控制器
GPIO: General-purpose input/output，通用的输入输出口
```

### 4.2.4 IMX6ULL 的 GPIO 模块结构

**1 有 5 组 GPIO（GPIO1～GPIO5），每组引脚最多有 32 个，但是可能实际上并没有那么多。**

- GPIO1 有 32 个引脚：GPIO1_IO0~GPIO1_IO31；
- GPIO2 有 22 个引脚：GPIO2_IO0~GPIO2_IO21；

- GPIO3 有 29 个引脚：GPIO3_IO0~GPIO3_IO28；
- GPIO4 有 29 个引脚：GPIO4_IO0~GPIO4_IO28；
- GPIO5 有 12 个引脚：GPIO5_IO0~GPIO5_IO11；

GPIO 的控制涉及 4 大模块：CCM、IOMUXC、GPIO 模块本身，框图如下：

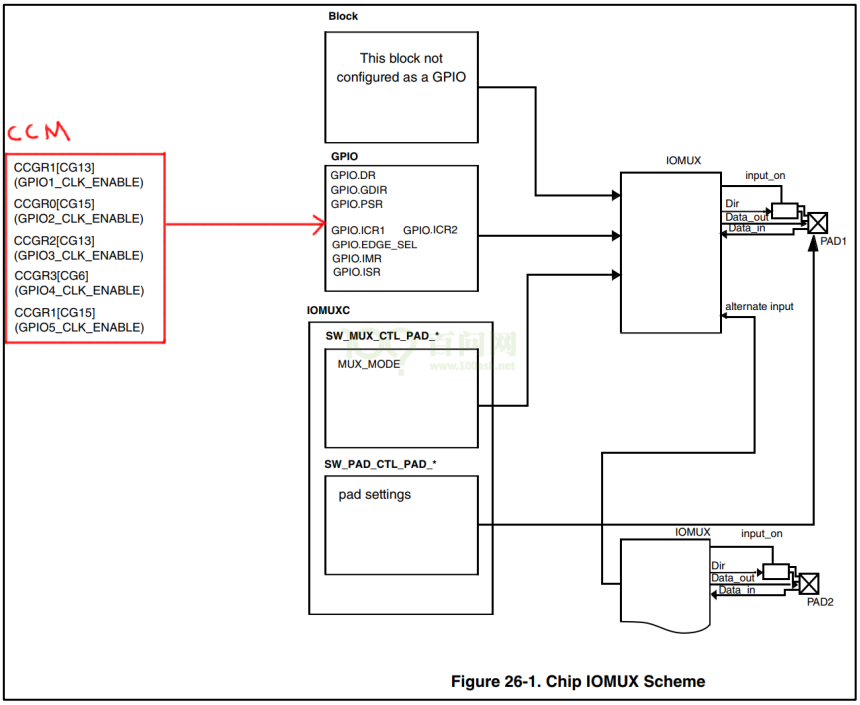



Figure 26-1. Chip IOMUX Scheme

## 4.2.5 CCM 用于设置是否向 GPIO 模块提供时钟

参考资料：网盘开发板配套资料“06_Datasheet（数据手册）/Core_board/CPU/IMX6ULLRM.pdf”芯片手册《Chapter 18: Clock Controller Module (CCM)》。

GPIOx 要用 CCM_CCGRy 寄存器中的 2 位来决定该组 GPIO 是否使能。哪组 GPIO 用哪个 CCM_CCGR 寄存器来设置，请看上图红框部分。

CCM_CCGR 寄存器中某 2 位的取值含义如下：

① 00：该 GPIO 模块全程被关闭

② 01：该 GPIO 模块在 CPU run mode 情况下是使能的；在 WAIT 或 STOP 模式下，关闭

③ 10：保留

④ 11：该 GPIO 模块全程使能

- GPIO2 时钟控制：

Address: 20C_4000h base + 68h offset = 20C_4068h

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | CG15 | | CG14 | | CG13 | | CG12 | | CG11 | | CG10 | | CG9 | | CG8 | |
| Reset | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | CG7 | | CG6 | | CG5 | | CG4 | | CG3 | | CG2 | | CG1 | | CG0 | |
| Reset | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

CCM_CCGR0 field descriptions

| Field | Description |
|---|---|
| 31–30 CG15 | gpio2_clocks (gpio2_clk_enable) |

- GPIO1、GPIO5 时钟控制：

Address: 20C_4000h base + 6Ch offset = 20C_406Ch

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | CG15 | | CG14 | | CG13 | | CG12 | | CG11 | | CG10 | | CG9 | | CG8 | |
| Reset | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | CG7 | | CG6 | | CG5 | | CG4 | | CG3 | | CG2 | | CG1 | | CG0 | |
| Reset | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**CCM_CCGR1 field descriptions**

| Field | Description |
|---|---|
| 31–30 CG15 | gpio5 clock (gpio5_clk_enable) |
| 29–28 CG14 | csu clock (csu_clk_enable) |
| 27–26 CG13 | gpio1 clock (gpio1_clk_enable) |

- GPIO3 时钟控制：

Address: 20C_4000h base + 70h offset = 20C_4070h

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | CG15 | | CG14 | | CG13 | | CG12 | | CG11 | | CG10 | | CG9 | | CG8 | |
| Reset | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | CG7 | | CG6 | | CG5 | | CG4 | | CG3 | | CG2 | | CG1 | | CG0 | |
| Reset | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**CCM_CCGR2 field descriptions**

| Field | Description |
|---|---|
| 31–30 CG15 | pxp clocks (pxp_clk_enable) |
| 29–28 CG14 | lcd clocks (lcd_clk_enable) |
| 27–26 CG13 | gpio3 clock (gpio3_clk_enable) |

- GPIO4 时钟控制：

Address: 20C_4000h base + 74h offset = 20C_4074h

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | CG15 | | CG14 | | CG13 | | CG12 | | CG11 | | CG10 | | CG9 | | CG8 | |
| Reset | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | CG7 | | CG6 | | CG5 | | CG4 | | CG3 | | CG2 | | CG1 | | CG0 | |
| Reset | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| | |
|---|---|
| 13–12 CG6 | gpio4 clock (gpio4_clk_enable) |

## 4.2.6 IOMUXC：引脚的模式(Mode、功能)

参考资料：芯片手册《Chapter 32: IOMUX Controller (IOMUXC)》。

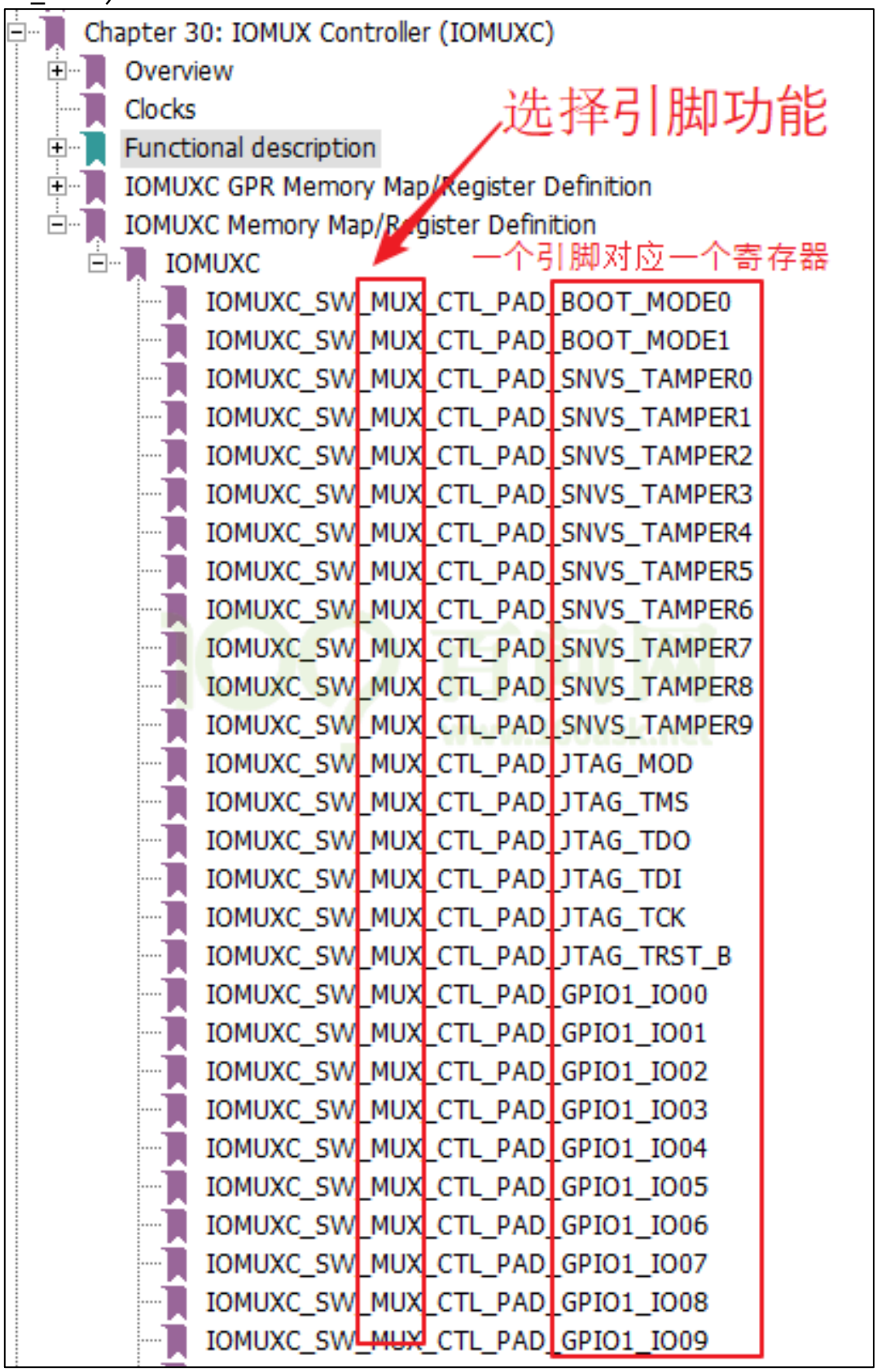
对于某个/某组引脚，IOMUXC 中有 2 个寄存器用来设置它：

① 选择功能：

- IOMUXC_SW_MUX_CTL_PAD_<PADNAME> ：Mux pad xxx，选择某个 pad 的功能

- IOMUXC_SW_MUX_CTL_GRP_<GROUP NAME>：Mux grp xxx，选择某组引脚的功能

某个引脚，或是某组预设的引脚，都有 8 个可选的模式(alternate (ALT) MUX_MODE)。



比如：

Address: 20E_0000h base + 5Ch offset = 20E_005Ch

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | Reserved | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | Reserved | | | | | | | | | | | SION | MUX_MODE | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

**IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO00 field descriptions**

| Field | Description |
|---|---|
| 31–5<br>- | This field is reserved.<br>Reserved |
| 4<br>SION | Software Input On Field. loopback回环模式，用于测试<br><br>Force the selected mux mode Input path no matter of MUX_MODE functionality.<br><br>1 **ENABLED** — Force input path of pad GPIO1_IO00<br>0 **DISABLED** — Input Path is determined by functionality |
| MUX_MODE | MUX Mode Select Field.<br><br>Select 1 of 9 iomux modes to be used for pad: GPIO1_IO00.<br><br>0000 **ALT0** — Select mux mode: ALT0 mux port: I2C2_SCL of instance: i2c2<br>0001 **ALT1** — Select mux mode: ALT1 mux port: GPT1_CAPTURE1 of instance: gpt1<br>0010 **ALT2** — Select mux mode: ALT2 mux port: ANATOP_OTG1_ID of instance: anatop<br>0011 **ALT3** — Select mux mode: ALT3 mux port: ENET1_REF_CLK1 of instance: enet1<br>0100 **ALT4** — Select mux mode: ALT4 mux port: MQS_RIGHT of instance: mqs<br>0101 **ALT5** — Select mux mode: ALT5 mux port: GPIO1_IO00 of instance: gpio1 用作GPIO时<br>0110 **ALT6** — Select mux mode: ALT6 mux port: ENET1_1588_EVENT0_IN of instance: enet1<br>0111 **ALT7** — Select mux mode: ALT7 mux port: SRC_SYSTEM_RESET of instance: src<br>1000 **ALT8** — Select mux mode: ALT8 mux port: WDOG3_WDOG_B of instance: wdog3 |

② 设置上下拉电阻等参数：

- IOMUXC_SW_PAD_CTL_PAD_<PAD_NAME>：pad pad xxx，设置某个 pad 的参数

- IOMUXC_SW_PAD_CTL_GRP_<GROUP NAME>：pad grp xxx，设置某组引脚的参数

比如：

Address: 20E_0000h base + 2E8h offset = 20E_02E8h

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R<br>W | Reserved | | | | | | | | | | | | | | | HYS |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R<br>W | PUS | | PUE | PKE | ODE | Reserved | | | SPEED | | DSE | | | Reserved | | SRE |
| Reset | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

**IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO00 field descriptions**

| Field | Description |
|---|---|
| 31–17<br>- | This field is reserved.<br>Reserved |
| 16<br>HYS | Hyst. Enable Field<br>Select one out of next values for pad: GPIO1_IO00<br>0 **HYS_0_Hysteresis_Disabled** — Hysteresis Disabled<br>1 **HYS_1_Hysteresis_Enabled** — Hysteresis Enabled<br>① 输入方式 Selectable Schmitt trigger or CMOS input mode |
| 15–14<br>PUS | Pull Up / Down Config. Field<br>Select one out of next values for pad: GPIO1_IO00<br>00 **PUS_0_100K_Ohm_Pull_Down** — 100K Ohm Pull Down<br>01 **PUS_1_47K_Ohm_Pull_Up** — 47K Ohm Pull Up<br>10 **PUS_2_100K_Ohm_Pull_Up** — 100K Ohm Pull Up<br>11 **PUS_3_22K_Ohm_Pull_Up** — 22K Ohm Pull Up<br>④ pull-up/pull-down 多少？ |
| 13<br>PUE | Pull / Keep Select Field<br>Select one out of next values for pad: GPIO1_IO00<br>0 **PUE_0_Keeper** — Keeper<br>1 **PUE_1_Pull** — Pull<br>③ 使能的是 keeper 还是 pull？ |
| 12<br>PKE | Pull / Keep Enable Field<br>② 是否使能？<br>Select one out of next values for pad: GPIO1_IO00<br>0 **PKE_0_Pull_Keeper_Disabled** — Pull/Keeper Disabled<br>1 **PKE_1_Pull_Keeper_Enabled** — Pull/Keeper Enabled<br>①②③④用于设置输入参数 |
| 11<br>ODE | Open Drain Enable Field<br>Select one out of next values for pad: GPIO1_IO00<br>ABCD 用于设置输出参数 |
| | 0 **ODE_0_Open_Drain_Disabled** — Open Drain Disabled<br>1 **ODE_1_Open_Drain_Enabled** — Open Drain Enabled<br>A. 输出方式 |
| 10–8<br>- | This field is reserved.<br>Reserved |
| 7–6<br>SPEED | Speed Field<br>Select one out of next values for pad: GPIO1_IO00<br>00 **SPEED_0_low_50MHz_** — low(50 MHz)<br>01 **SPEED_1_medium_100MHz_** — medium(100 MHz)<br>10 **SPEED_2_medium_100MHz_** — medium(100 MHz)<br>11 **SPEED_3_max_200MHz_** — max(200 MHz)<br>C. 输出速率 |
| 5–3<br>DSE | Drive Strength Field<br>Select one out of next values for pad: GPIO1_IO00<br>000 **DSE_0_output_driver_disabled_** — output driver disabled;<br>001 **DSE_1_R0_260_Ohm___3_3V__150_Ohm_1_8V__240_Ohm_for_DDR_** — R0(260 Ohm @ 3.3V, 150 Ohm@1.8V, 240 Ohm for DDR)<br>010 **DSE_2_R0_2** — R0/2<br>011 **DSE_3_R0_3** — R0/3<br>100 **DSE_4_R0_4** — R0/4<br>101 **DSE_5_R0_5** — R0/5<br>110 **DSE_6_R0_6** — R0/6<br>111 **DSE_7_R0_7** — R0/7<br>B. 驱动能力<br>D. 压摆率/转换速率 比如同是输出50M波形，还可设置高低电平的转换速率 |
| 2–1<br>- | This field is reserved.<br>Reserved |
| 0<br>SRE | Slew Rate Field<br>Select one out of next values for pad: GPIO1_IO00<br>0 **SRE_0_Slow_Slew_Rate** — Slow Slew Rate<br>1 **SRE_1_Fast_Slew_Rate** — Fast Slew Rate<br>→ 波形缓和，低干扰<br>→ 高速电路 |

### 4.2.7 GPIO 模块内部

框图如下：

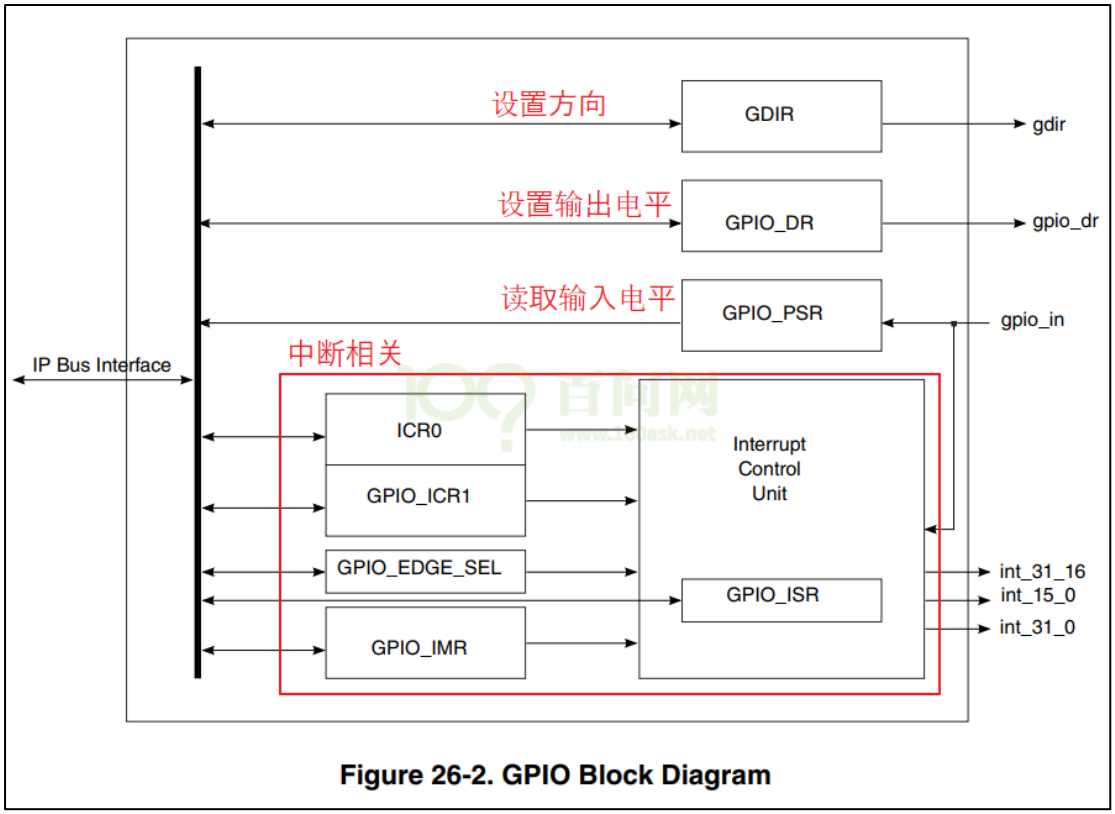



**Figure 26-2. GPIO Block Diagram**

我们暂时只需要关心 3 个寄存器：

① GPIOx_GDIR：设置引脚方向，每位对应一个引脚，1-output，0-input

Address: Base address + 4h offset 看Chapter 2: Memory Maps

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | GDIR | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**GPIOx_GDIR field descriptions**

| Field | Description |
|---|---|
| GDIR | GPIO direction bits. Bit n of this register defines the direction of the GPIO[n] signal.<br>**NOTE:** GPIO_GDIR affects only the direction of the I/O signal when the corresponding bit in the I/O MUX is configured for GPIO.<br>0 **INPUT** — GPIO is configured as input.<br>1 **OUTPUT** — GPIO is configured as output. |

② GPIOx_DR：设置输出引脚的电平，每位对应一个引脚，1-高电平，0-低电平

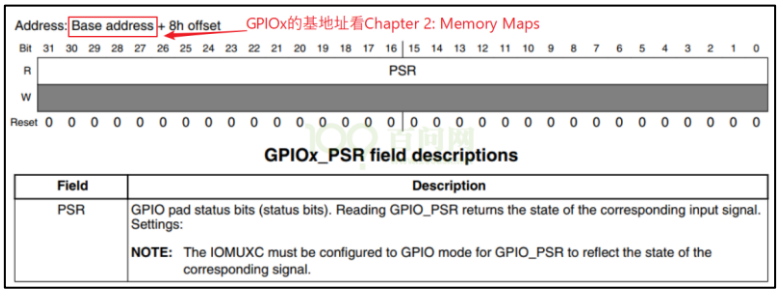
Address: Base address + 0h offset GPIOx的基地址看Chapter 2: Memory Maps

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | DR | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**GPIOx_DR field descriptions**

| Field | Description |
|---|---|
| DR | Data bits. This register defines the value of the GPIO output when the signal is configured as an output (GDIR[n]=1). Writes to this register are stored in a register. Reading GPIO_DR returns the value stored in the register if the signal is configured as an output (GDIR[n]=1), or the input signal's value if configured as an input (GDIR[n]=0).<br>**NOTE:** The I/O multiplexer must be configured to GPIO mode for the GPIO_DR value to connect with the signal. Reading the data register with the input path disabled always returns a zero value. |

③ GPIOx_PSR：读取引脚的电平，每位对应一个引脚，1-高电平，0-低电平




**GPIOx_PSR field descriptions**

| Field | Description |
|---|---|
| PSR | GPIO pad status bits (status bits). Reading GPIO_PSR returns the state of the corresponding input signal. Settings:<br>**NOTE:** The IOMUXC must be configured to GPIO mode for GPIO_PSR to reflect the state of the corresponding signal. |

## 4.2.8 怎么编程

### 1 读 GPIO

**26.4.3.1 GPIO Read Mode**

The programming sequence for reading input signals should be as follows:

1. Configure IOMUX to select GPIO mode (Via IOMUX Controller (IOMUXC) ).
2. Configure GPIO direction register to input (GPIO_GDIR[GDIR] set to 0b).
3. Read value from data register/pad status register.

A pseudocode description to read [input3:input0] values is as follows:

```
// SET INPUTS TO GPIO MODE.
write sw_mux_ctl_<input0>_<input1>_<input2>_<input3>, 32'h00000000
// SET GDIR TO INPUT.
write GDIR[31:4,input3_bit, input2_bit, input1_bit, input0_bit,] 32'hxxxxxxx0
// READ INPUT VALUE FROM DR.
read DR
// READ INPUT VALUE FROM PSR.
read PSR
```

**NOTE**

While the GPIO direction is set to input (GPIO_GDIR = 0), a read access to GPIO_DR does not return GPIO_DR data. Instead, it returns the GPIO_PSR data, which is the corresponding input signal value.

翻译一下：

① 设置 CCM_CCGRx 寄存器中某位使能对应的 GPIO 模块 // 默认是使能的，上图省略了

② 设置 IOMUX 来选择引脚用于 GPIO

③ 设置 GPIOx_GDIR 中某位为 0，把该引脚设置为输入功能

④ 读 GPIOx_DR 或 GPIOx_PSR 得到某位的值（读 GPIOx_DR 返回的是 GPIOx_PSR 的值）

### 2 写 GPIO

**26.4.3.2 GPIO Write Mode**

The programming sequence for driving output signals should be as follows:

1. Configure IOMUX to select GPIO mode (Via IOMUXC), also enable SION if need to read loopback pad value through PSR
2. Configure GPIO direction register to output (GPIO_GDIR[GDIR] set to 1b).
3. Write value to data register (GPIO_DR).

A pseudocode description to drive 4'b0101 on [output3:output0] is as follows:

```
// SET PADS TO GPIO MODE VIA IOMUX.
write sw_mux_ctl_pad_<output[0-3]>.mux_mode, <GPIO_MUX_MODE>
// Enable loopback so we can capture pad value into PSR in output mode
write sw_mux_ctl_pad_<output[0-3]>.sion, 1
// SET GDIR=1 TO OUTPUT BITS.
write GDIR[31:4,output3_bit,output2_bit, output1_bit, output0_bit,] 32'hxxxxxxxF
// WRITE OUTPUT VALUE=4'b0101 TO DR.
write DR, 32'hxxxxxxx5
// READ OUTPUT VALUE FROM PSR ONLY.
read_cmp PSR, 32'hxxxxxxx5
```

翻译一下：

① 设置 CCM_CCGRx 寄存器中某位使能对应的 GPIO 模块 // 默认是使能的，上图省略了

② 设置 IOMUX 来选择引脚用于 GPIO

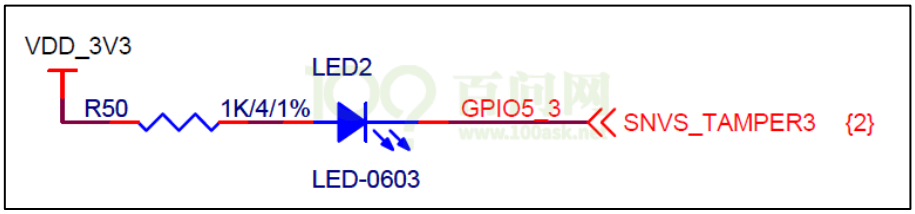
③ 设置 GPIOx_GDIR 中某位为 1，把该引脚设置为输出功能
④ 写 GPIOx_DR 某位的值

需要注意的是，你可以设置该引脚的 loopback 功能，这样就可以从 GPIOx_PSR 中读到引脚的有实电平；你从 GPIOx_DR 中读回的只是上次设置的值，它并不能反应引脚的真实电平，比如可能因为硬件故障导致该引脚跟地短路了，你通过设置 GPIOx_DR 让它输出高电平并不会起效果。

## 4.3 100ASK_IMX6ULL 的 LED 程序

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/4_led”目录下。
原理图：网盘开发板配套资料“05_Hardware（原理图)/Base_board/100ask_imx6ull_v1.1.pdf”。

### 4.3.1 看原理图确定引脚及操作方法




从上图可知，这个 LED 用到了 GPIO5_3 引脚。

在芯片手册里，这引脚的名字是：GPIO5_IO03，可以根据名字搜到对应的寄存器。

当这些引脚输出低电平时，对应的 LED 被点亮；输出高电平时，LED 熄灭。

### 4.3.2 所涉及的寄存器操作

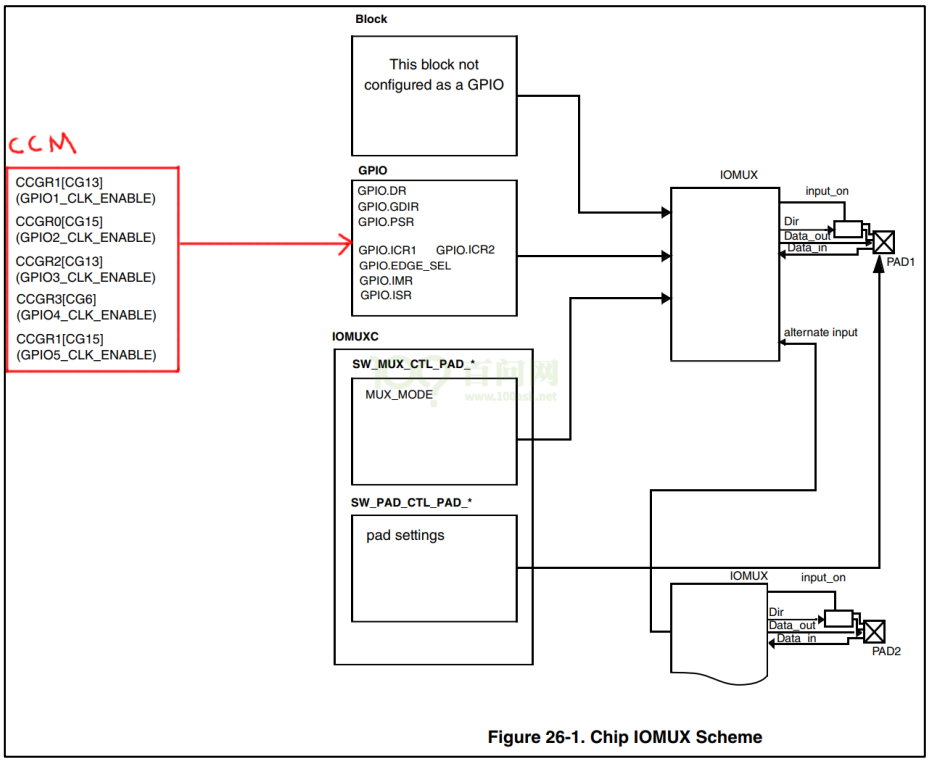



Figure 26-1. Chip IOMUX Scheme

## 第1步 使能 GPIO5

设置 b[31:30]就可以使能 GPIO5，设置为什么值呢？

注意：在 imx6ullrm.pdf 中，CCM_CCGR1 的 b[31:30]是保留位；我以前写程序时错用了 imx6ul(不是 imx6ull)的手册，导致程序中额外操作了这些保留位。不去设置 b[31:30]，GPIO5 也是默认使能的。

看下图，设置为 0b11：

| CGR value | Clock Activity Description |
|---|---|
| 00 | Clock is off during all modes. Stop enter hardware handshake is disabled. |
| 01 | Clock is on in run mode, but off in WAIT and STOP modes |
| 10 | Not applicable (Reserved). |
| 11 | Clock is on during all modes, except STOP mode. |

① 00：该 GPIO 模块全程被关闭

② 01：该 GPIO 模块在 CPU run mode 情况下是使能的；在 WAIT 或 STOP 模式下，关闭

③ 10：保留

④ 11：该 GPIO 模块全程使能

```
/* GPIO5_IO03 */
/* a. 使能 GPIO5
 * set CCM to enable GPIO5
 * CCM_CCGR1[CG15] 0x20C406C
 * bit[31:30] = 0b11
 */
```

## 第2步 设置 GPIO5_IO03 为 GPIO 模式

设置如下寄存器：

Address: 229_0000h base + 14h offset = 229_0014h

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R<br>W | Reserved | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R<br>W | Reserved | | | | | | | | | | | SION | MUX_MODE | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 field descriptions**

| Field | Description |
|---|---|
| 31–5<br>- | This field is reserved.<br>Reserved |
| 4<br>SION | Software Input On Field.<br>Force the selected mux mode Input path no matter of MUX_MODE functionality.<br>1 **ENABLED** — Force input path of pad SNVS_TAMPER3<br>0 **DISABLED** — Input Path is determined by functionality |
| MUX_MODE | Mux Mode Select Field<br>**NOTE:** ALT5 mode is only valid when TAMPER PIN is used as GPIO. This depends on FUSE setting "TAMPER_PIN_DISABLE[1:0]".<br>Following is the mux information when TAMPER PIN is used as GPIO: SNVS_TAMPER3 ==> GPIO5_03<br>101 **ALT5** — Select mux mode: ALT5 mux port, GPIO5_IO03 of instance - gpio<br>Other Reserved |

```
/* b. 设置 GPIO5_IO03 用于 GPIO
 * set IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3
 *      to configure GPIO5_IO03 as GPIO
 * IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 0x2290014
 * bit[3:0] = 0b0101 alt5
 */
```

## 第3步 设置 GPIO5_IO03 为输出引脚，设置其输出电平

寄存器地址为：

**GPIO memory map**

| Absolute address (hex) | Register name | Width (in bits) | Access | Reset value | Section/ page |
|---|---|---|---|---|---|
| 209_C000 | GPIO data register (GPIO1_DR) | 32 | R/W | 0000_0000h | 28.5.1/1358 |
| 209_C004 | GPIO direction register (GPIO1_GDIR) | 32 | R/W | 0000_0000h | 28.5.2/1359 |
| 20A_C000 | GPIO data register (GPIO5_DR) | 32 | R/W | 0000_0000h | 28.5.1/1358 |
| 20A_C004 | GPIO direction register (GPIO5_GDIR) | 32 | R/W | 0000_0000h | 28.5.2/1359 |

- 设置方向寄存器，把引脚设置为输出引脚：

Address: Base address + 4h offset

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | GDIR | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**GPIOx_GDIR field descriptions**

| Field | Description |
|---|---|
| GDIR | GPIO direction bits. Bit n of this register defines the direction of the GPIO[n] signal.<br>**NOTE:** GPIO_GDIR affects only the direction of the I/O signal when the corresponding bit in the I/O MUX is configured for GPIO.<br>0 **INPUT** — GPIO is configured as input.<br>1 **OUTPUT** — GPIO is configured as output. |

- 设置数据寄存器，设置引脚的输出电平：

Address: Base address + 0h offset

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | DR | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**GPIOx_DR field descriptions**

| Field | Description |
|---|---|
| DR | Data bits. This register defines the value of the GPIO output when the signal is configured as an output (GDIR[n]=1). Writes to this register are stored in a register. Reading GPIO_DR returns the value stored in the register if the signal is configured as an output (GDIR[n]=1), or the input signal's value if configured as an input (GDIR[n]=0).<br>**NOTE:** The I/O multiplexer must be configured to GPIO mode for the GPIO_DR value to connect with the signal. Reading the data register with the input path disabled always returns a zero value. |

```
/* c. 设置 GPIO5_IO03 作为 output 引脚
 * set GPIO5_GDIR to configure GPIO5_IO03 as output
 * GPIO5_GDIR  0x020AC000 + 0x4
 * bit[3] = 0b1
 */

/* d. 设置 GPIO5_DR 输出低电平
 * set GPIO5_DR to configure GPIO5_IO03 output 0
 * GPIO5_DR 0x020AC000 + 0
 * bit[3] = 0b0
 */

/* e. 设置 GPIO5_IO3 输出高电平
 * set GPIO5_DR to configure GPIO5_IO03 output 1
 * GPIO5_DR 0x020AC000 + 0
 * bit[3] = 0b1
 */
```

### 4.3.3 写程序

led.c 是重点，其他文件暂且不关心，比如 Makefile、imx6ull.lds，这些文件的知识在后面再介绍。

led_init 函数会初始化 LED 引脚：使能、设置为 GPIO 模式、设置为输出引脚。

值得关注的是第 13～16 行，在 C 语言中使用指针来访问寄存器，需要先设置指针的值，即它指向哪个地址——当然是寄存器的地址：

```
02 #include "led.h"
03
04 static volatile unsigned int *CCM_CCGR1                          ;
05 static volatile unsigned int *IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3;
06 static volatile unsigned int *GPIO5_GDIR                         ;
07 static volatile unsigned int *GPIO5_DR                           ;
08
09 void led_init(void)
10 {
11     unsigned int val;
12
13     CCM_CCGR1 = (volatile unsigned int *)(0x20C406C);
14   IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 = (volatile unsigned int *)(0x2290014);
15     GPIO5_GDIR  = (volatile unsigned int *)(0x020AC000 + 0x4);
16     GPIO5_DR    = (volatile unsigned int *)(0x020AC000);
17
18     /* GPIO5_IO03 */
19     /* a. 使能 GPIO5
20      * set CCM to enable GPIO5
21      * CCM_CCGR1[CG15] 0x20C406C
22      * bit[31:30] = 0b11
23      */
24     *CCM_CCGR1 |= (3<<30);
25
26     /* b. 设置 GPIO5_IO03 用于 GPIO
27      * set IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3
28      *     to configure GPIO5_IO03 as GPIO
29      * IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3  0x2290014
30      * bit[3:0] = 0b0101 alt5
31      */
32     val = *IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3;
33     val &= ~(0xf);
34     val |= (5);
35     *IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 = val;
36
37
38     /* c. 设置 GPIO5_IO03 作为 output 引脚
39      * set GPIO5_GDIR to configure GPIO5_IO03 as output
40      * GPIO5_GDIR  0x020AC000 + 0x4
41      * bit[3] = 0b1
42      */
43     *GPIO5_GDIR |= (1<<3);
44
45 }
```

- led_ctl 函数会根据参数设置 LED 引脚的输出电平：

```
47 void led_ctl(int on)
48 {
```

```
49      if (on) /* on: output 0*/
50      {
51              /* d. 设置 GPIO5_DR 输出低电平
52               * set GPIO5_DR to configure GPIO5_IO03 output 0
53               * GPIO5_DR 0x020AC000 + 0
54               * bit[3] = 0b0
55               */
56              *GPIO5_DR &= ~(1<<3);
57      }
58      else  /* off: output 1*/
59      {
60              /* e. 设置 GPIO5_IO3 输出高电平
61               * set GPIO5_DR to configure GPIO5_IO03 output 1
62               * GPIO5_DR 0x020AC000 + 0
63               * bit[3] = 0b1
64               */
65              *GPIO5_DR |= (1<<3);
66      }
67 }
```

### 4.3.4 编译程序

要编译程序，需要有交叉编译工具链。可以使用在线下载方式下载工具链，然后设置交叉编译工具链.

然后，使用 FileZilla 把 windows 下的 4_led 目录上传到 Ubuntu 中。

最后在 Ubuntu 中进入 4_led 执行 make 命令即可生成 led.imx、led.img

**注意：建议在 Ubuntu 中目录名不要含空格，使用中文目录名是可以的。**

**注意：执行 make 命令后会编译程序、制作映象文件，具体过程以后会详细介绍，这不是本章的内容。**

**注意：如果有以下错误，执行“chmod +x tools/mkimage”后再次执行 make 即可：**

```
make: execvp: ./tools/mkimage: Permission denied
```

### 4.3.5 上机实验

**注意：烧写、运行 led 程序时，串口是没有输出的，这只是一个点灯程序没用到串口。**

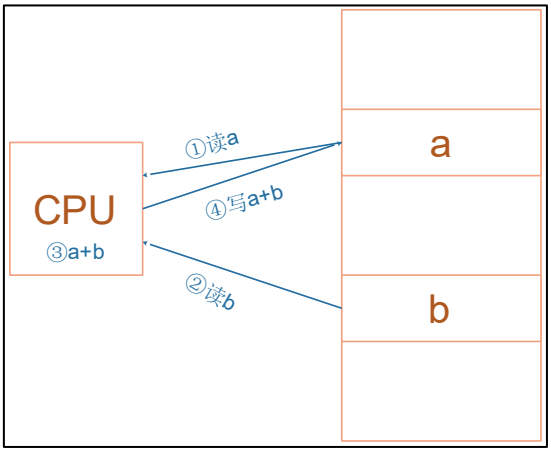
# 第5章 LED 程序涉及的编程知识

## 5.1 ARM 处理器程序运行的过程

ARM 芯片属于精简指令集计算机(RISC：Reduced Instruction Set Computing)，它所用的指令比较简单，有如下特点：

① 对内存只有读、写指令
② 对于数据的运算是在 CPU 内部实现
③ 使用 RISC 指令的 CPU 复杂度小一点，易于设计

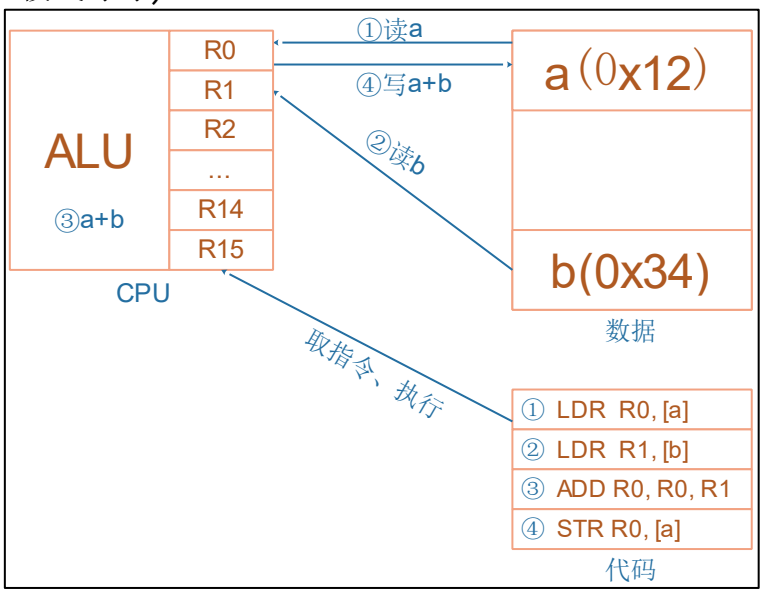
比如对于 a=a+b 这样的算式，需要经过下面 4 个步骤才可以实现：




细看这几个步骤，有些疑问：

① 读 a，那么 a 的值读出来后保存在 CPU 里面哪里？
② 读 b，那么 b 的值读出来后保存在 CPU 里面哪里？
③ a+b 的结果又保存在哪里？

我们需要深入 ARM 处理器的内部。简单概括如下，我们先忽略各种 CPU 模式(系统模式、用户模式等等)。




CPU 运行时，先去取得指令，再执行指令：

① 把内存 a 的值读入 CPU 寄存器 R0
② 把内存 b 的值读入 CPU 寄存器 R1

③ 把 R0、R1 累加，存入 R0
④ 把 R0 的值写入内存 a

现在我们知道了：CPU 内部有很多寄存器，CPU 要从外部设备上读入指令，执行指令。

## 5.2 ARM 架构的简单介绍

IMX6UL 使用 Cortex-A7 架构，本小节简单介绍一下 Cortex-A7 架构的基础知识，比如运行模式、寄存器组等。
参考资料：

- 文件名：ARMv7 编程手册(DEN0013D_cortex_a_series_PG).pdf
- 文档所在目录：网盘开发板配套资料“ 08_Reference material (ARM,NXP 参考资料)/Arm 架构参考资料.zip”
- 参考章节：《3: ARM Processor Modes and Registers》

### 5.2.1 运行模式

Cortex-A7 架构的运行模式有 9 种，分别为 User、Sys(System)、FIQ、IRQ、ABT(Abort)、SVC(Supervisor)、UND(Undef)、MON(Monitor)、Hyp 模式，如下表：

| 模式 | 描述 |
|---|---|
| User | 用户模式，非特权模式，大部分程序运行的时候就处于此模式 |
| Sys(System) | 系统模式，用于运行特权级的操作系统任务 |
| FIQ | 快速中断模式，进入 FIQ 中断异常 |
| IRQ | 一般中断模式 |
| ABT(Abort) | 数据访问终止模式，用于虚拟存储以及存储保护 |
| SVC(Supervisor) | 超级管理员模式，供操作系统使用 |
| UND(Undef) | 未定义指令终止模式 |
| MON(Monitor) | 用于安全扩展模式 |
| Hyp | 用于虚拟化扩展 |

除了 User 模式属于非特权模式，其它 8 种处理器模式都是特权模式。

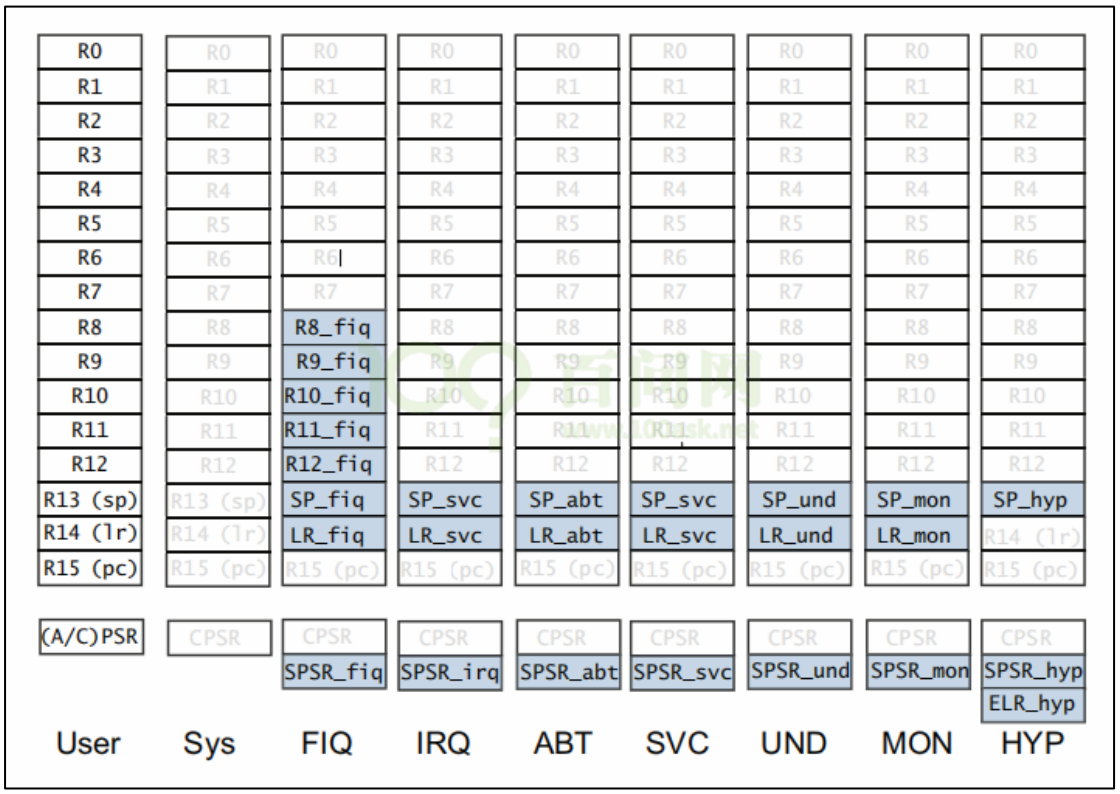
运行模式可以通过软件进行任意切换，也可以通过中断或者异常来进行切换。大多数的程序都运行在用户模式，用户模式下是不能访问系统所有资源的，有些资源是受限的，要想访问这些受限的资源就必须进行模式切换。但是用户模式是不能直接进行切换的，用户模式下需要借助异常来完成模式切换，当要切换模式的时候，应用程序可以产生异常，在异常的处理过程中完成处理器模式切换。
所谓“运行模式”，可以这样简单理解：
① 板子上电时，CPU 处于 SVC 模式，它用的是 SVC 模式下的寄存器
② 程序运行时发生了中断，CPU 进入 IRQ 模式，它用的 IRQ 模式下的寄存器
③ CPU 处理完中断，它切换回 SVC 模式，继续使用 SVC 模式下的寄存器
④ CPU 发生某种异常时，比如读取内存出错，它会进入 ABT 模式，使用 ABT 模式下的寄存器来处理错误。

在某种模式下，CPU 执行时使用的是这种模式的资源，比如使用的是这组模

式的寄存器。这样就可以免去保存上一个模式所使用的寄存器。

### 5.2.2 寄存器组

本节我们要讲的是 Cortex-A7 内核寄存器组，而不是芯片外设寄存器。上一小节我们讲了 Cortex-A7 有 9 种运行模式，每一种运行模式都有一组与之对应的寄存器组，如下图：




浅色字体是与 User 模式所共有的寄存器，浅蓝色背景是各个模式所独有的寄存器，即在所有的模式中，低寄存器组(R0~R7)是共享同一组物理寄存器的，只是一些高寄存器组在不同的模式有自己独有的寄存器，比如 FIQ 模式下 R8~R14 是独立的物理寄存器。

- 如果某个程序处于 FIQ 模式下访问寄存器 R13(SP)，那它实际访问的是寄存器 SP_fiq
- 如果某个程序处于 SVC 模式下访问寄存器 R13(SP)，那它实际访问的是寄存器 SP_svc

9 种运行模式的寄存器合计有 34 个，可以分为：

**1．未备份寄存器，即 R0~R7**

**2．备份寄存器，即 R8~R14**

**3．程序计数器 ，即 R15**

**4．程序状态寄存器**

下面一一介绍以上 4 类寄存器。

① **未备份寄存器**

未备份寄存器指的是 R0~R7，因为在所有的运行模式下 R0~R7 寄存器都是同一

个物理寄存器，在不同的模式下，R0~R7 寄存器中的数据就会被破坏，所以 R0~R7 寄存器并没有被用作特殊用途。

② 备份寄存器

备份寄存器中的 R8~R12 寄存器有两种物理寄存器，在快速中断模式下(FIQ)它们对应着 Rx_irq(x=8~12)物理寄存器，其他模式下对应着 Rx(8~12)物理寄存器。FIQ 是快速中断模式，这个中断模式要求快速执行！因为 FIQ 模式下的 R8~R12 是独立的，因此中断处理程序可以不用保存和恢复 R8~R12，从而加速中断的执行过程。

备份寄存器 R13(SP) ，也叫栈指针，有 8 个物理寄存器，其中一个是 User 和 Sys 模式共用的，剩下的 7 个分别对应 7 种不同的模式。

备份寄存器 R14(LR) ，也叫链接寄存器，有 7 个物理寄存器，其中一个是 User、Sys 和 Hyp 模式所共有的，剩下的 6 个分别对应 6 种不同的模式，主要有如下用途：

使用 R14(LR)来存放当前子程序的返回地址，如果使用 BL 或者 BLX 来调用子函数的话，R14(LR)被设置成该子函数的返回地址，在子函数中，将 R14(LR)中的值赋给 R15(PC)即可完成子函数返回，如 mov pc,lr

③ 程序计数器

程序计数器 R15(PC)，保存着当前执行指令地址值加 8 个字节

因为 ARM 处理器是三级流水线：取指->译码->执行，循环执行。比如当前正在执行第一条指令的同时也对第二条指令进行译码，第三条指令也同时被取出存放在 R15(PC)中，即 R15(PC)总是指向当前正在执行指令地址再加上 2 条指令的地址，对于 32 位的 ARM 处理器，每条指令是 4 个字节，

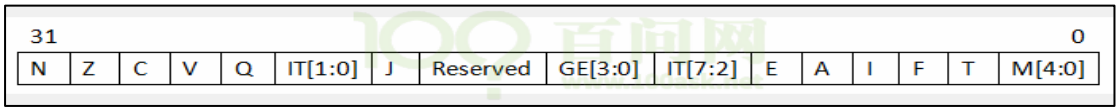
所以 R15(PC) = 当前执行指令地址 + 8 个字节

④ 程序状态寄存器

程序状态寄存器 PSR 可以分成当前程序状态寄存器 CPSR 与备份程序状态寄存器 SPSR。

所有运行模式都共用一个 CPSR 物理寄存器，因此 CPSR 可以在任何模式下被访问，该寄存器包含条件标志位、中断禁止位、当前运行模式标志等一些状态位以及一些控制位。但是所有运行模式都共用一个 CPSR 必然会导致冲突，因此除了 User 和 Sys 模式以外，其他 7 个模式都配备一个专用的物理状态寄存器，叫做 备份程序状态寄存器(SPSR)，当特定异常中断发生时，SPSR 用来保存 CPSR 的值，当异常退出以后可以用 SPSR 中保存的值来恢复 CPSR。

由于 SPSR 是 CPSR 的备份，因此 SPSR 和 CPSR 的寄存器结构相同，如下图：




- N(bit31)：当两个有符号整数运算(补码表示)时，结果用 N 表示，N=1/0 表示 负数/正数
- Z(bit30)：对于 CMP 指令，Z=1 表示进行比较的两个数大小相等
- C(bit29)：
- 在加法指令中，当结果产生了进位，则 C=1，表示无符号数运算发生上溢，其它情况下 C=0

- 在减法指令中，当运算中发生借位，则 C=0，表示无符号数运算发生下溢，其它情况下 C=1
- 对于包含移位操作的非加/减法运算指令，C 中包含最后一次溢出的位的数值
- 对于其它非加/减运算指令，C 位的值通常不受影响
- V(bit28)：对于加/减法运算指令，当操作数和运算结果表示为二进制的补码表示的带符号数时，V=1 表示符号位溢出，通常其他位不影响 V 位
- Q(bit27)：仅 ARM v5TE_J 架构支持，表示饱和状态，Q=1/0 表示累积饱和/累积不饱和
- IT[1:0](bit26:25) 和 IT[7:2](bit15:bit10)一起组成 IT[7:0]，作为 IF-THEN 指令执行状态
- J(bit24)和 T(bit5)：控制指令执行状态，表明本指令是 ARM 指令还是 Thumb 指令，如表

| J | T | 描述 |
|---|---|---|
| 0 | 0 | ARM |
| 0 | 1 | Thumb |
| 1 | 1 | ThumbEE |
| 1 | 0 | Jazelle |

- GE[3:0](bit19:16)：SIMD 指令有效，大于或等于
- E(bit9)：大小端控制位，E=1/0 表示大/小端模式
- A(bit8)：禁止异步中断位，A=1 表示禁止异步中断
- I(bit7)：I=1/0 代表 禁止/使能 IRQ
- F(bit6)：F=1/0 代表 禁止/使能 FIQ
- M[4:0]：运行模式控制位，如表
- M[4:0] 运行模式

| M[4:0] | 运行模式 |
|---|---|
| 10000 | User 模式 |
| 10001 | FIQ 模式 |
| 10010 | IRQ 模式 |
| 10011 | Supervisor(SVC)模式 |
| 10110 | Monitor(MON)模式 |
| 10111 | Abort(ABT)模式 |
| 11010 | Hyp(HYP)模式 |
| 11011 | Undef(UND)模式 |
| 11111 | System(SYS)模式 |

## 5.3 汇编与机器码、汇编指令

参考资料:文件名：armv7　arm 架构参考手册 学习 CPU 架构、内存及系统架构（DDI0406C_d_armv7ar_arm）.pdf

文档所在目录:网盘开发板配套资料“08_Reference material(ARM,NXP 参考资料)/Arm 架构参考资料.zip”

参考章节:《A5: ARM Instruction Set Encoding》

根据指令复杂度来区分，所有 CPU 可以分为 2 类：

① CISC 复杂指令集计算机，Complex Instruction Set Computer，比如 x86

② RISC 精简指令集计算机，Reduced Instruction Set Computing，比如 ARM，RISC-V

比如，对于加法运算：a = a + b，它涉及 4 个步骤的操作：读出 a，读出 b，计算 a+b，把结果写回 a。

① 使用 CISC(复杂指令集计算机，比如 x86)提供的加法指令：

只需要一条指令即可完成这 4 步操作。当然，这一个指令需要多个 CPU 周期才可以完成。

② 而 RISC 不提供“一站式”的加法指令：

需调用四条指令完成两数相加：内存 a 加载到寄存器，内存 b 加载到寄存器，两个寄存器中数相加，寄存器结果存入内存 a。

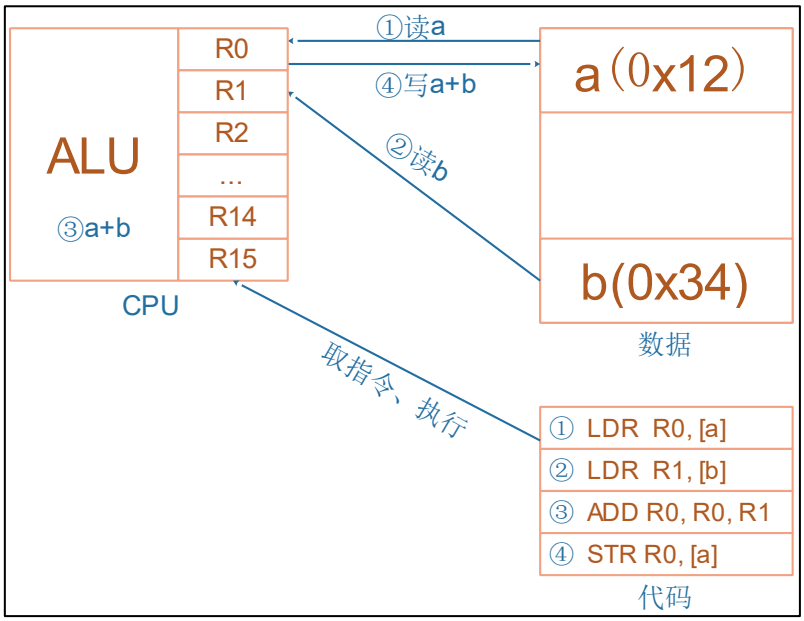
ARM 芯片属于精简指令集计算机(RISC：Reduced Instruction Set Computing)，它所用的指令比较简单，有如下特点：

① 对内存只有读、写指令

② 对于数据的运算是在 CPU 内部实现

③ 使用 RISC 指令的 CPU 复杂度小一点，易于设计

### 5.3.1 汇编与机器码




上图的例子中，数值 a 原来是保存在内存里的，执行了某条指令后，它的值被读入内存，那问题来了：

**1．什么指令，可以让 CPU 从内存里把数据读进来？**

比如：

```
mov r3, #addr_a // 把变量 a 的地址传给 CPU 寄存器 r3
ldr r0, [r3]    // 从 r3 所指的内存把数值读进 CPU 寄存器 r0
```

**2．读进来后，这个数保存在哪里？**

当然是保存在 CPU 内部了，存在某个寄存器里，上面的代码用寄存器 r0 来保存该值

**3．如何处理数据？**

CPU 执行加法指令，比如：

```
add r0, r0, r1 // 在 CPU 内部，r0=r0+r1
```

**4．最终数据怎么写入内存？**

CPU 执行指令，比如：

```
str r0, [r3]   // 将 r0 的值写入 r3 所指的内存
```

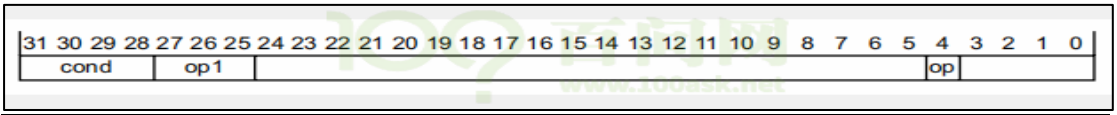
上面例子中，mov、add、ldr、str 等都是汇编指令，或者说它们是“助记符”——帮助我们记忆的。

记忆什么呢？这些指令其实是一个一个数值，我们去记这些数值有难度，所以就用 mov、add 表示不同指令对应的数值。在下面的表格中，op1 对应 bit[27:25]，不同的 op1 对应不同的指令。我们就用 mov、add 等等来表示这些值。

注意，op1 只是指令，它只占据 3 位(bit[27:25])，一条完整的指令还需要更多参数。比如“mov r1, r0”里面的 r1、r0 就是参数。这些参数也是保存在同一个 32 位数里。

这个 32 位数值就是机器码，即汇编指令是机器码的助记符。

ARM 指令机器码是有一定格式，如下：




| cond | op1 | op | 指令类型 |
|---|---|---|---|
| not 1111 | 00x | - | 数据处理和杂项指令，如 MOV |
| not 1111 | 010 | - | 加载/存储指令，如 LDR/STR |
| not 1111 | 011 | 0 | 加载/存储指令，如 LDR/STR |
| not 1111 | 011 | 1 | 媒体指令(英文：Media instructions) |
| not 1111 | 10x | - | 分支指令，如 B、BL；块数据传输指令，如 LDM/STM、POP/PUSH |
| not 1111 | 11x | - | 协处理器指令 |
| 1111 | - | - | 无条件指令，如 BL |

## 5.3.2 汇编指令

下面讲解几种常用的汇编指令。

参考资料：

- 文件名：ARMv7 编程手册(DEN0013D_cortex_a_series_PG).pdf
- 文档所在目录：网盘开发板配套资料“08_Reference material (ARM,NXP 参考资料)/Arm 架构参考资料.zip”
- 参考章节：《3: ARM Processor Modes and Registers》

汇编指令的格式，如下：

```
label:
    instruction @ comment
```

- **label，即标签，表示地址位置，可以通过 label 得到指令/数据地址**
- **instruction，即指令，表示汇编指令或伪指令**
- **@ comment，@表示后面是注释，comment 表示注释内容**

比如：

```
addnum:
    mov r0, #0 @ 将 R0 寄存器设置成 0
```

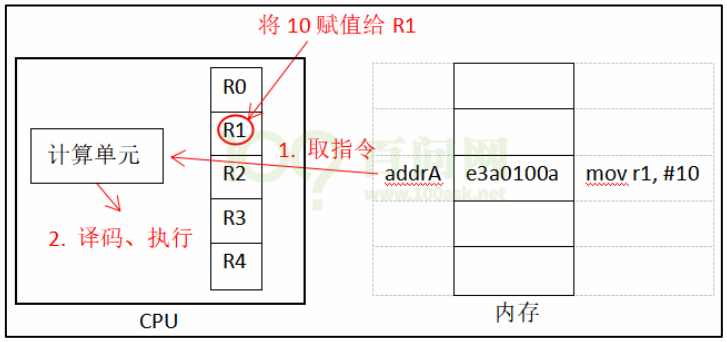
上面汇编代码中，addnum 表示标签，mov r0, #0 表示指令，@ 将 R0 寄存器设置成 0 表示 注释

常用的汇编指令一般有 mov、bl/b、add/sub、ldm/stm、push/pop 等等，下面一一介绍。

a) MOV 指令：Move register or constant，把某个寄存器的值移给另一个寄存器，或是把一个常数移入寄存器。

```
mov r1, #10  @ 将 10 赋值给寄存器 r1，即 r1=10
```

指令执行过程，如下：

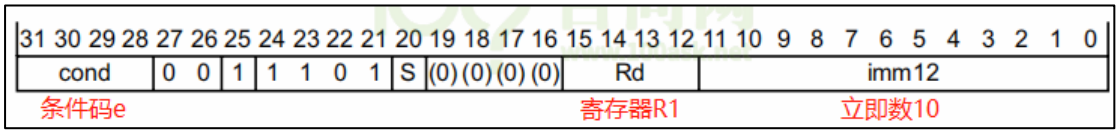



**1.** 取指：假设从内存的 addrA 地址取机器码 e3a0100a（即 mov r1, #10 指令）

**2.** 译码：原来是 MOV 指令

**3.** 执行：CPU 内部寄存器 R1 等于 10

其中，机器码 e3a0100a，MOV 指令各位的解析如下：

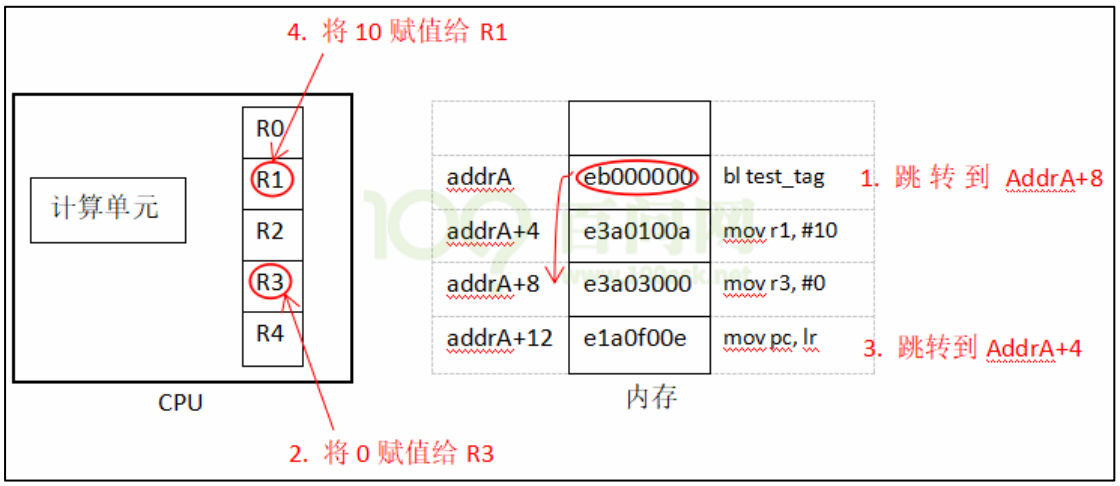


[31:28]位是条件码 0xe；[15：12]位是寄存器 R1，即 0x1；[12:0]位是立即数 10，即 0x00a

b) BL 指令：Branch with Link，跳转并把返回地址记录在 LR 寄存器里。

```
1 bl test_tag
2 mov r1, #10
3
4 test_tag:
5   mov r3, #0
6   mov pc, lr
```

第 1 行，跳转到 test_tag 标签处执行“mov r3, #0”指令，并且将下一条指令即“mov r1, #10”指令的地址存储到 LR 寄存器。
第 6 行，返回到“mov r1, #10”指令地址，并且执行“mov r1, #10”指令
指令执行过程，如下：



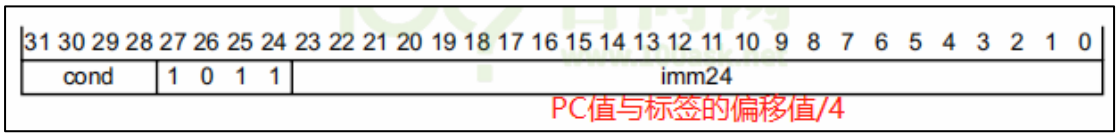

1. CPU 从内存的 addrA 地址取机器码 eb000000（即“bl test_tag”指令），执行后，PC 跳转到 test_tag 标签位置，即内存的 addrA+8 地址，从上图可知，其实 test_tag 标签的地址是“mov r3, #0”指令的地址。同时自动将内存的 addrA+4 地址存储在寄存器 LR 中。

2. CPU 从内存的 addrA+8 地址取机器码 e3a03000（即“mov r3, #0”指令），执行，CPU 内部寄存器 R3 等于 0。

3. CPU 从内存的 addrA+12 地址取机器码 e1a0f00e（即“mov pc, lr”指令），执行，PC 跳转到内存的 addrA+4 地址。

4. CPU 从内存的 addrA+4 地址取机器码 e3a0100a（即“mov r1, #10”指令），执行，CPU 内部寄存器 R1 等于 10

其中，机器码 eb000000，BL 指令各位的解析如下：




imm[23:0]是 PC 值与标签的偏移值除以 4，但是此处的偏移值是 0，为什么尼？这是因为 ARM 采用三级流水线的方法，即取指、译码、执行指令。所以当 ARM 执行 addrA 地址的“bl test_tag”指令时，但是 PC 已经指向 addrA+8 地址进行取“mov r3, #0”指令，也就是 PC 值等于当前指令地址加 8，所以要跳去执

行“addrA+8”的指令，偏移值就要设置为 0：执行这条 bl 指令后，新的 PC=PC+偏移，刚好就是“addrA+8”。

c） B 指令：Branch，跳转指令。相比于 BL 指令，它并不保存下一条指令的地址到 LR 寄存器。

```
1 b test_tag
2 mov r1, #10
3
4 test_tag:
5   mov r3, #0
```

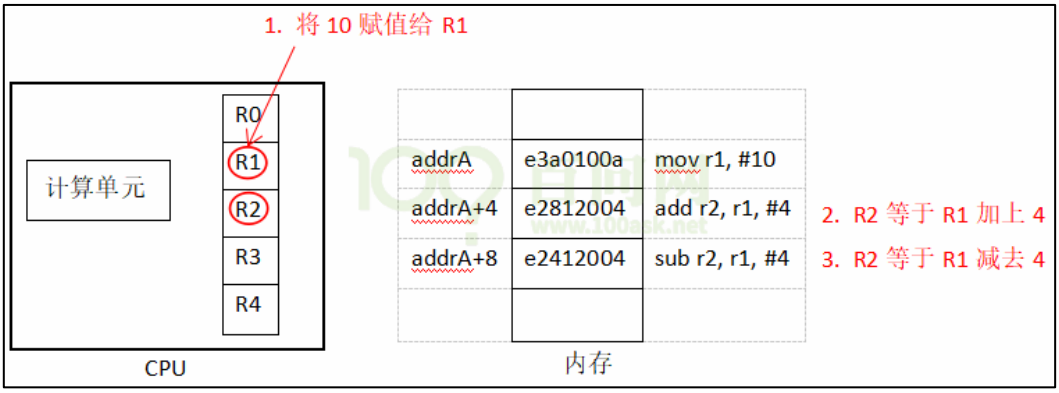
第 1 行，只是跳转到 test_tag 标签处执行“mov r3, #0”指令，没有保存返回地址。

指令 B 与指令 BL，大同小异，此处就不一一分析了，可以参数指令 BL，它们的区别：BL 指令的下一条指令的地址会被存储到 LR 寄存器，而 B 指令不会存储。

d） ADD/SUB 指令：加法、减法指令。

```
1 mov r1, #10
2 add r2, r1, #4
3 sub r2, r1, #4
```

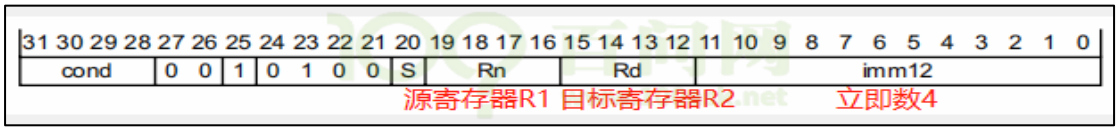
第 2 行，r2 = r1 + 4。
第 3 行，r2 = r1 - 4。
指令执行过程，如下：



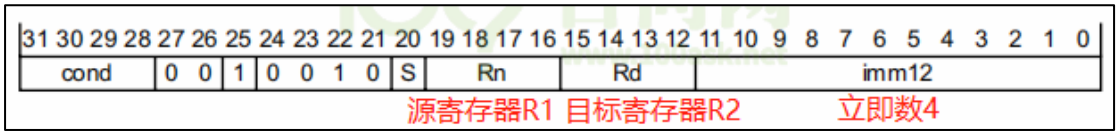

CPU 从内存的 addrA+4 地址取机器码 e2812004（即 add r2, r1, #4 指令），执行后，CPU 内部寄存器 R2 等于 14

CPU 从内存的 addrA+8 地址取机器码 e2412004（即 sub r2, r1, #4 指令），执行后，CPU 内部寄存器 R2 等于 6

其中，机器码 e2812004，ADD 指令各位的解析如下：




[19: 16]位是源寄存器 R1，即 1；[15: 12]位是目标寄存器 R2，即 2；[11: 0]位是立即数 4，即 0x004；其中，机器码 e2412004，SUB 指令各位的解析如下：

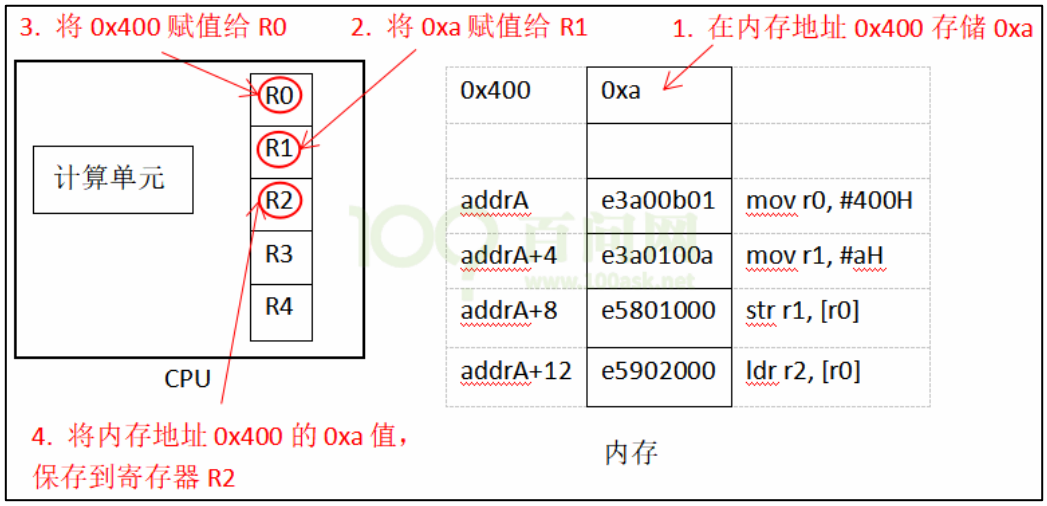


[19: 16]位是源寄存器 R1，即 1；[15: 12]位是目标寄存器 R2，即 2；[11: 0]位是立即数 4，即 0x004；

e) LDR/STR 指令：Load register from memory/Store register to memory，前者是读内存到寄存器，后者是把寄存器的值写入内存。

```
1 mov r0, #400H @ 0x400
2 mov r1, #aH   @ 0xa
3 str r1, [r0]
4 ldr r2, [r0]
```

第 3 行，将寄存器 R1 的值 0xa 存储到寄存器 R0 指向的地址 0x400
第 4 行，将寄存器 R0 指向地址 0x400 的数据赋值给寄存器 R2
指令执行过程，如下：



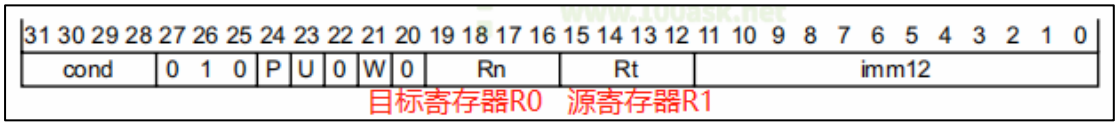

**1.** CPU 从内存的 addrA 地址取机器码 e3a00b01（即 mov r0, #400H 指令），执行后，CPU 内部寄存器 R0 等于 0x400

**2.** CPU 从内存的 addrA+4 地址取机器码 e3a0100a（即 mov r1, #aH 指令），执行后，CPU 内部寄存器 R1 等于 0xa

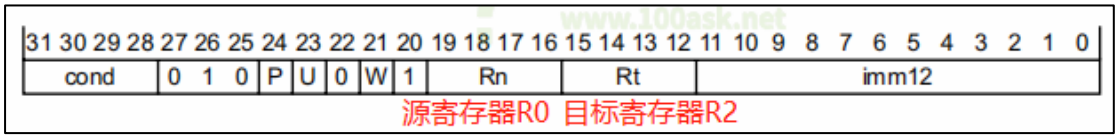
**3.** CPU 从内存的 addrA+8 地址取机器码 e5801000（即 str r1, [r0]指令），执行后，寄存器 R1 的 0xa 数据存储到寄存器 R0 指向的地址 0x400，即内存的 0x400 地址的值为 0xa

**4.** CPU 从内存的 addrA+12 地址取机器码 e5902000（即 ldr r2, [r0]指令），执行后，寄存器 R0 指向的地址 0x400 的数据存储到 CPU 内部寄存器 R2，即 CPU 内部寄存器 R2 等于 0xa

其中，机器码 e5801000，STR 指令各位的解析如下：




[19: 16]位是目标寄存器 R0，即 0；[15: 12]位是源寄存器 R1，即 1；其中，机器码 e5902000，LDR 指令各位的解析如下：



[19: 16]位是源寄存器 R0，即 0；[15: 12]位是目标寄存器 R2，即 2；

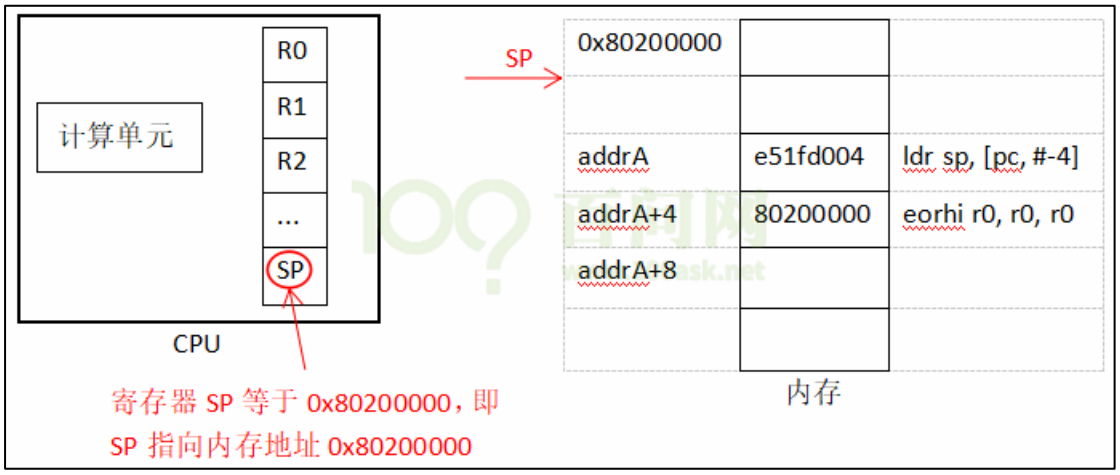
f) LDR 伪指令

所谓“伪指令”就是假的、不真实存在的指令。编译器会用真正的指令来代替它。

Ldr 指令的参数中，有“=”时，它就是伪指令，不再是上小节讲的“读内存”指令了。

```
ldr sp,=0x80200000
```

这个是一条伪指令，即实际中并不存在这个指令，它会被拆分成几个真正的 ARM 指令，实现一样的效果，将 0x80200000 赋值给寄存器 sp，即 sp=0x80200000。指令执行过程，如下：




ldr sp,=0x80200000 这条伪指令，被翻译成一条指令来执行： ldr sp, [pc, #-4]。它是一条读内存指令，显然，在“pc-4”这个地址对应的内存里存有 0x80200000 这个数值。这是编译器预先把 0x80200000 设置在这个地址的。

g) LDM/STM 指令

- **ldm，Load multiple registers，从内存里把值加载进多个寄存器。**
- **stm，Store Multiple，把多个寄存器的值存储到内存。**

格式：

```
ldm{cond} Rn{!}, reglist
stm{cond} Rn{!}, reglist
```

参数说明：

- **cond：如下所示，下面列出的前四个条件是用于数据块操作，后四个条件是用于堆栈操作：**

- IA : Increment After，先传输再增加地址
- IB : Increment Before，先增加地址再传输
- DA : Decrement After，先传输再减小地址
- DB : Decrement Before，先减小地址再传输
- FD : 满递减堆栈
- FA : 满递增堆栈

- ED ：空递减堆栈
- EA ：空递增堆栈
- Rn：基址寄存器，里面含有内存的起始地址。
- !：表示最后的地址写回到 Rn 中。
- reglist：里面列出多个寄存器，如{R1,R2,R6-R9}。

**注意：传输数据时，低号寄存器对应低地址，高号寄存器对应高地址。**

示例：

```
1 ldr r1,=0x10000000
3 ldmib r1!, {r0,r4-r6}
4 stmda r1!, {r0,r4-r6}
```

第 1 行，将起始地址 0x10000000 赋值给 r1

第 3 行，因为使用 ib，所以每次传送前地址加 4，具体操作如下(低地址的值存入低号寄存器)：

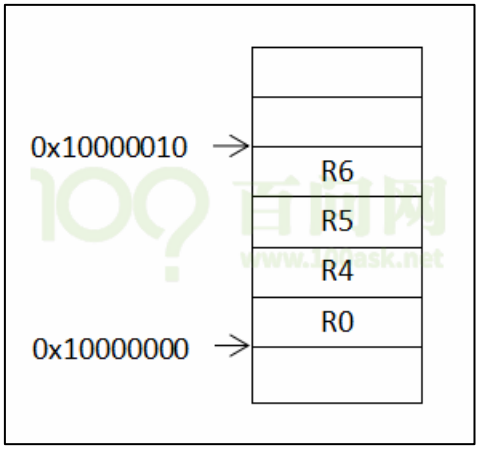
```
将 0X10000004 地址的内容赋值给 R0
将 0X10000008 地址的内容赋值给 R4
将 0X1000000C 地址的内容赋值给 R5
将 0X10000010 地址的内容赋值给 R6
```

“!”表示，最后的地址写回到 R1 中，R1=0X10000010

第 4 行，因为使用 da，所以每次传送后地址减 4，具体操作如下(低号寄存器的值存入低地址)：

```
将 R6 存储到 0X10000010 地址
将 R5 存储到 0X1000000C 地址
将 R4 存储到 0X10000008 地址
将 R0 存储到 0X10000004 地址
```

“!”表示，最后的地址写回到 R1 中，R1=0X10000000，如下图所示：




堆栈操作：满递减堆栈

```
1 ldr sp,=0x80200000
2
3 stmfd sp!, {r0-r2} @ 入栈
4 ldmfd sp!, {r0-r2} @ 出栈
```

第 1 行，将 0x80200000 赋值给 sp，作为堆栈的起始地址

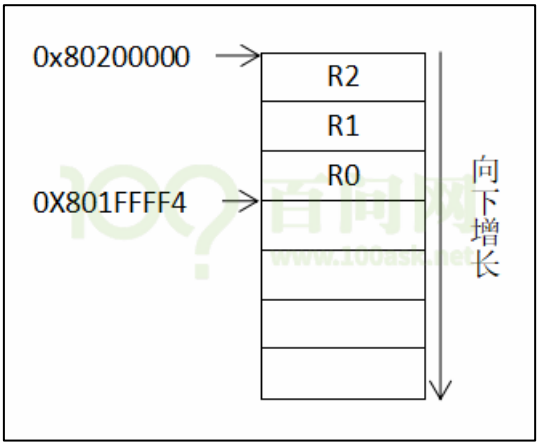
第 3 行，入栈，具体操作如下：

将 R2 存储到 0X80200000 地址
将 R1 存储到 0X801FFFFC 地址
将 R0 存储到 0X801FFFF8 地址

第 4 行，出栈，具体操作如下：

将 0X801FFFF8 地址的内容赋值给 R0
将 0X801FFFFC 地址的内容赋值给 R1
将 0X80200000 地址的内容赋值给 R2

如下图所示：




上述第 3，4 行汇编代码，就是所谓的入栈，出栈。也可以用 push，pop 指令完成入栈，出栈，如下

```
1 ldr sp,=0x80200000
2
3 push {r0-r2} @ 入栈
4 pop {r0-r2}  @ 出栈
```

## 5.4 进制

目前计算机对数据的表示方式，有十六进制、十进制、八进制与二进制。

### 5.4.1 如何理解它们的区别？

- 十六进制，逢十六进一，每一位由 0~F 组成，习惯用 0x 前缀表示或用 H 后缀表示：0xA 或 AH
- 十进制，逢十进一，每一位由 0~9 组成，无前缀或用 D 后缀表示:10 或 10D
- 八进制，逢八进一，每一位由 0~7 组成，习惯用 0(数字 0)前缀表示或用 O(字母 O)后缀表示：012 或 12O
- 二进制，逢二进一，每一位由 0~1 组成，习惯用 0b 前缀表示或用 B 后缀表示：0b1010 或 1010B

### 5.4.2 在 C 语言中怎么表示这些进制呢？

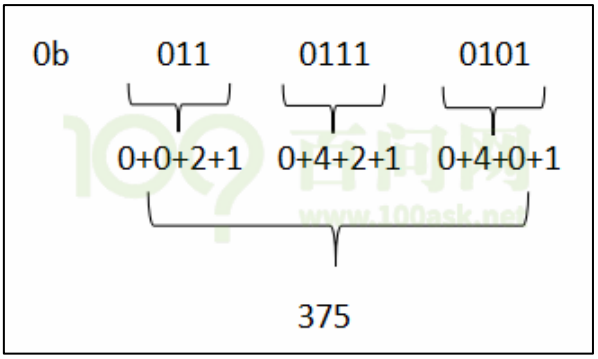
```
十六进制:int a = 0xA;   // 0x 前缀
十进制:  int a = 10;
八进制:  int a = 012;   // 0 前缀
C 语言中无法直接用“0b”这样的格式表示二进制数。
```

### 5.4.3 十六进制与二进制转换关系

在嵌入式开发中经常需要对十六进制与二进制进行转换，如何快速的转换 2/16 进制？

首先记住 **8 4 2 1** ——>二进制权重。将二进制 0b01101110101 转换成十六进制：将二进制从右到左，每四个分成一组，结果就是 0x375。

将十六进制 0xABC1 转换成二进制：将十六进制从右到左，每个分成四位：




结果就是 1010 1011 1100 0001。

## 5.5 大/小端模式与位操作

### 5.5.1 大/小端模式

- 大端模式（Big-endian），是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址中

- 小端模式（Little-endian），是指数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中

比如：0x12345678，在大/小端模式的存储位置如下：

| 内存地址 | 大端模式 | 小端模式 |
|---|---|---|
| addr+3 | 0x78 | 0x12 |
| addr+2 | 0x56 | 0x34 |
| addr+1 | 0x34 | 0x56 |
| addr | 0x12 | 0x78 |

### 5.5.2 位操作

**1 移位**

```
1 int a = 0x6; // 二进制是 0b0110
2 int b = a<<1;
3 int c = a>>1;
```

第 2 行，对 a 左移一位，从 0b0110->0b1100，即 b=0xC

第 3 行，对 a 右移一位，从 0b0110->0b0011，即 b=0x3

**2 取反**

```
1 int a = 0x6; // 二进制是 0b0110
2 int b = ~a;
```

第 2 行，对 a 按位取反，从 0b0110->0b1001，即 b=0x9

**3 位与**

只有对应的两个二进位都为 1 时，结果位才为 1

```
1 int a = 0x6; // 二进制是 0b0110
2 int b = 0x7; // 二进制是 0b0111
3
4 int c = a&b;
```

第 4 行，a&b，二进制是 0b0110，即 c=0x6

**4 位或**

只要对应的二个二进位有一个为 1 时，结果位就为 1

```
1 int a = 0x6; // 二进制是 0b0110
2 int b = 0x7; // 二进制是 0b0111
3
4 int c = a|b;
```

第 4 行，a|b，二进制是 0b0111，即 c=0x7

**5 置位**

```
1 int a = 0x6;     // 二进制是 0b0110
2
3 int a |= (1<<3);
```

第 3 行，将变量 a 的 bit3 置 1。1<<3 = 0b1000，然后 0b1000|0b0110=0b1110，即 a=0xe

**6 清位**

```
1 int a = 0x6;     // 二进制是 0b0110
2
3 int a &= ~(1<<2);
```

第 3 行，将变量 a 的 bit2 清位。~(1<<2) = 0b1011，然后 0b1011&0b0110=0b0010，即 a=0x2

## 5.6 汇编程序调用 C 程序

参考资料：网盘开发板配套资料“08_Reference material (ARM,NXP 参考资料)/ATPCS.pdf”。

在 C 程序和 ARM 汇编程序之间相互调用时必须遵守 ATPCS 规则，ATPCS 规定了一些函数间调用的基本规则。

### 5.6.1 ATPCS 规则

ATPCS 即 ARM-THUMB procedure call standard（ARM-Thumb 过程调用标准）的简称，是基于 ARM 指令集和 THUMB 指令集过程调用的规范，规定了调用函数如何传递参数，被调用函数如何获取参数，以何种方式传递函数返回值。寄存器 R0~R15 在 ATPCS 规则的使用：

- 在函数中，通过寄存器 R0~R3 来传递参数，被调用的函数在返回前无需恢复寄存器 R0~R3 的内容
- 在函数中，通过寄存器 R4~R11 来保存局部变量
- 寄存器 R12 用作函数间 scratch 寄存器
- 寄存器 R13 用作栈指针，记作 SP，在函数中寄存器 R13 不能用做其他用途，寄存器 SP 在进入函数时的值和退出函数时的值必须相等

- 寄存器 R14 用作链接寄存器，记作 LR，它用于保存函数的返回地址，如果在函数中保存了返回地址，则 R14 可用作其它的用途
- 寄存器 R15 是程序计数器，记作 PC，它不能用作其他用途

### 5.6.2 汇编程序如何向 C 程序的函数传递参数

- 当参数小于等下 4 个时，使用寄存器 R0~R3 来进行参数传递
- 当参数大于 4 个时，前四个参数按照上面方法传递，剩余参数传送到栈中，入栈的顺序与参数顺序相反，即最后一个参数先入栈

### 5.6.3 C 程序如何返回结果给汇编程序

- 结果为一个 32 位的整数时，通过寄存器 R0 返回
- 结果为一个 64 位整数时，通过 R0 和 R1 返回，依此类推.
- 结果为一个浮点数时，通过浮点运算部件的寄存器 f0，d0 或 s0 返回
- 结果为一个复合的浮点数时，通过寄存器 f0-fN 或者 d0~dN 返回
- 对于位数更多的结果，通过调用内存来传递

### 5.6.4 C 函数为何要用栈

总的来说，栈的作用就是：保存现场/上下文，传递参数

**1 保存现场/上下文**

保存现场，也叫保存上下文

现场，相当于案发现场，总有一些现场的情况，要记录下来的，否则被别人破坏掉之后，你就无法恢复现场了。而此处说的现场，就是指 CPU 运行的时候，用到了一些寄存器，比如 R0~R3，LR 等等，对于这些寄存器的值，如果你不保存而直接跳转到函数中去执行，那么很可能会被破坏了，因为函数执行需要用到这些寄存器。

因此在函数调用之前，应该将这些寄存器等现场，暂时保持起来，等调用函数执行完毕返回后，再恢复现场,这样 CPU 就可以正确的继续执行了。

保存寄存器的值，一般用的是 push 指令，将对应的某些寄存器的值，一个个放到栈中，即所谓的入栈。

然后待被调用的子函数执行完毕的时候，再调用 pop，把栈中的一个个的值，赋值给对应的入栈的寄存器，即所谓的出栈。

**2 传递参数**

当函数被调用并且参数大于 4 个时，（不包括第 4 个参数）第 4 个参数后面的参数就保存在栈中。

## 5.7 C 语言中读写寄存器

注意，我们说的这些寄存器不是 CPU 内部的寄存器，而是 CPU 之外、芯片上的某个模块里的寄存器。这些寄存器跟内存的访问方法是一样的。

每一个寄存器都有一个地址，只要找到寄存器地址，通过指针指向寄存器地址单元，通过读写指针值，就可以获得寄存器值。

首先，定义一个指针，指针类型根据寄存器大小决定，同时需要加上 volatile 关键字让编译器不要优化此指针，比如，CCM_CCGR1 寄存器值是 32 位，此处定义为 unsigned int *指针类型，寄存器地址为 0x20C406C

```
volatile unsigned int *CCM_CCGR1 = (volatile unsigned int *)(0x20C406C);
```

然后，对寄存器进行读写操作

```
val = *CCM_CCGR1;       // 读寄存器
*CCM_CCGR1 |= (3<<30);  // 写寄存器，将 CCM_CCGR1 寄存器的[31：30]位置 1
```

## 5.8 start.S 解析

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/4_led”目录下。

```
2 .text
3 .global _start
4 _start:
```

- 第 2 行，.text 表示代码段，汇编系统预定义段名，说明下面的汇编是代码段
- 第 3 行，.global 表示_start 是一个全局符号
- 第 4 行，标签_start，汇编程序的默认入口是_start，也可以在链接脚本中使用 ENTRY 来指明其它的入口点，类似 C 语言 main()函数，_start 是整个程序的入口，即程序执行的第一条指令

@ 相当于一个函数，_start 是函数名，下面汇编指令是函数内容

```
4 _start:
5
6   //设置栈
7   ldr sp,=0x80200000
8
9   bl clean_bss
10
11  bl main
12
13 halt:
14 b halt
```

- 第 7 行，将 0x80200000 赋值给寄存器 sp，即设置栈地址，因为 C 语言函数调用时，保存现场/上下文和传递参数需要用到栈
- 第 9 行，跳转到标签 clean_bss，相当于调用 clean_bss 函数，并将 bl main 指令地址存储到寄存器 lr 中
- 第 11 行，进入 C 语言的 main()函数，并将 b halt 指令地址存储到寄存器 lr 中
- 第 13 行，标签 halt
- 第 14 行，跳转到标签 halt，循环执行 b halt 指令执行，这就是一个死循环。如果 main 函数返回，就在这里死循环。

@ 相当于一个函数，clean_bss 是函数名，下面汇编指令是函数内容

```
16 clean_bss:
```

```
17  /* 清除 BSS 段 */
18  ldr r1, =__bss_start
19  ldr r2, =__bss_end
20  mov r3, #0
21 clean:      @ 下面汇编指令相当于循环体，直到 R1 与 R2 相等
22  str r3, [r1]
23  add r1, r1, #4
24  cmp r1, r2
25  bne clean
26
27  mov pc, lr @ 函数执行完毕，返回
```

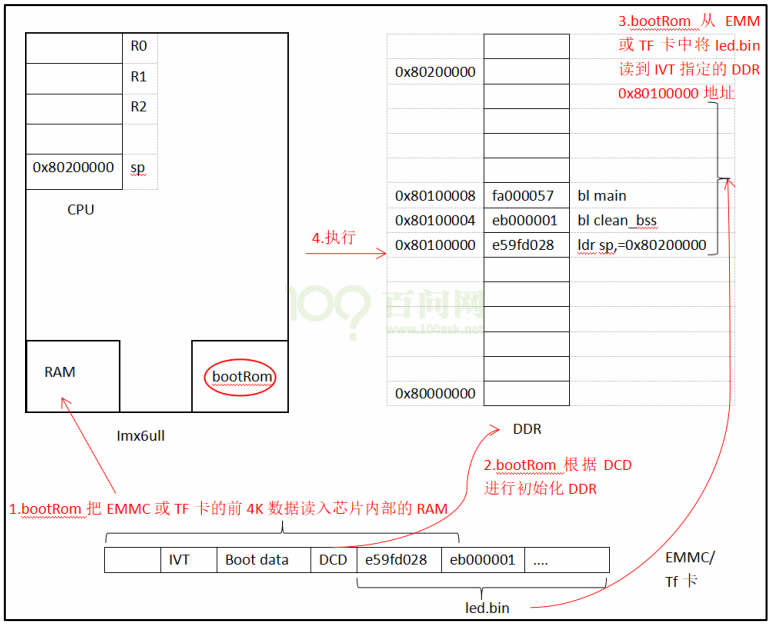
- 第 16 行，标签 clean_bss，下面汇编代码是清除 BSS 段，将 BSS 段所在内存都设置成 0
- 第 18 行，将链接脚本定义的 bss 起始地址赋值给寄存器 r1
- 第 19 行，将链接脚本定义的 bss 结束地址赋值给寄存器 r2
- 第 20 行，将 0 赋值给寄存器 r3，即 r3=0
- 第 21 行，标签 clean
- 第 22 行，将寄存器 r3 的值存储到寄存器 r1 的值对应地址中
- 第 23 行，将寄存器 r1 的值加上 4，赋值给寄存器 r1，即 r1 = r1+4
- 第 24 行，比较寄存器 r1 的值与寄存器 r2 的值
- 第 25 行，如果寄存器 r1 的值与寄存器 r2 的值不相等，跳转到标签 clean
- 第 26 行，如果寄存器 r1 的值与寄存器 r2 的值相等，就执行此行，返回到 bl main 处，继续执行。

## 5.9 根据 led.dis 分析代码的整体运行流程

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/4_led”目录下。

在分析 led.dis 文件前，我们再把 imx6ull 芯片如何将 led.bin 文件复制到内存 DDR 中过程，简单整体过一篇。

如下图，imx6ull 芯片一上电后，会先执行 bootRom 程序，此程序是芯片出厂时已经固定的程序，除了芯片原厂，咱们是无法修改的。




bootRom 有什么作用？下面一一讲解。

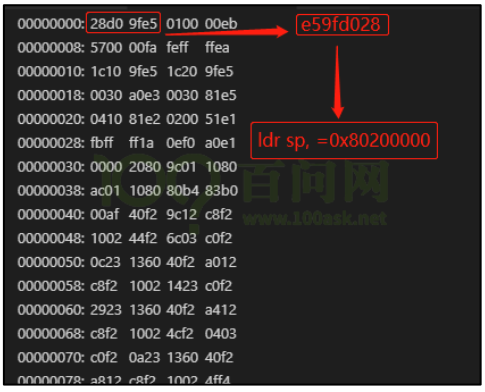
**1.** bootRom 会把 EMMC 或 TF 卡的前 4K 数据读入到芯片内部 RAM 运行

**2.** bootRom 根据 DCD 进行初始化 DDR。

**3.** bootRom 根据 IVT，从 EMMC 或 TF 卡中将 led.bin 读到 DDR 的 0x80100000 地址

**4.** 跳转到 DDR 的 0x80100000 地址执行

目前 led.bin 程序已经复制到内存中，CPU 开始从内存 0x80100000 地址开始执行机器码，每一条机器码是 32 位/4 字节，此处的机器码就是 led.bin 中的机器码，那我们能不能打开 led.bin 文件，看到里面的机器码？答案是可以的。如下图：

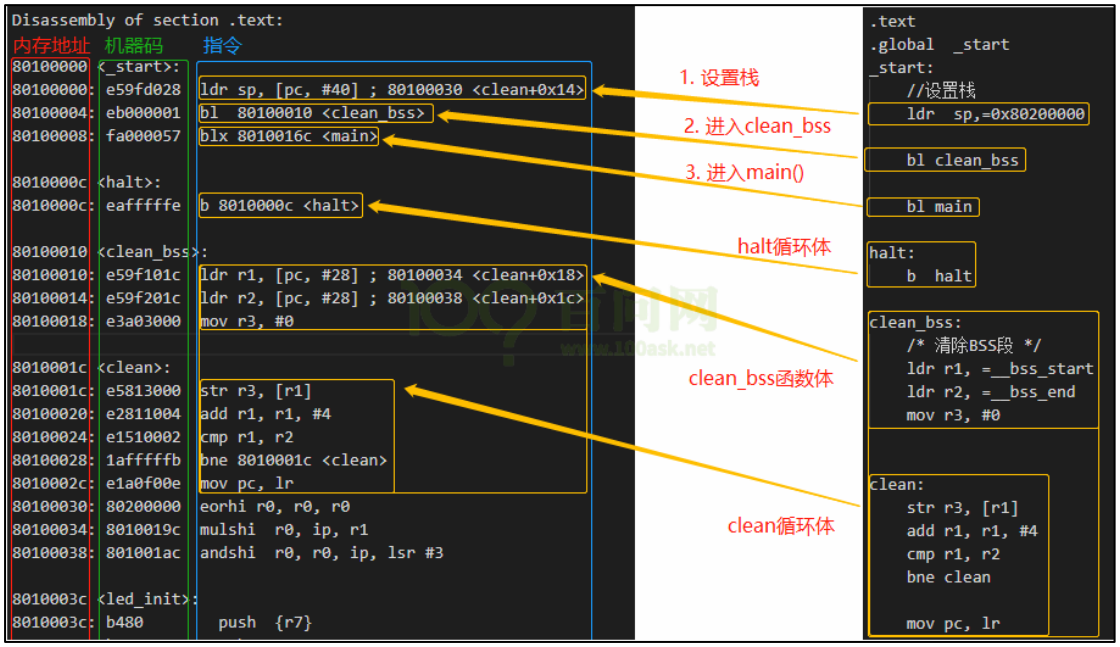


前面介绍过大/小端模式，你是否记得？如果忘记了，可以回头看一下。
此处可以看到机器码 e59fd028（指令：`ldr sp,=0x80200000`）的存储形式：
地址　机器码

```
00000000  28
00000001  d0
00000002  9f
00000003  e5
```

没错，imx6ull 的存储方式是小端模式，换一句话说，ARM 存储方式一般都是小端模式。

但是 bin 文件的机器码不方便阅读，所以我们一般会通过 arm-linux-gnueabihf-objdump 进行反汇编，得到人类容易读的 led.dis 文件。
如下图：




下面我们就来分析一下 led.dis 文件，但是在阅读此小节前，尽量把前一小节《1.8 start.S 解析》完全理解懂，不然阅读此小节，有点云里雾里。

**1 CPU 执行的第一条机器码就是内存地址 0x80100000 存储的 e59fd028 机器码**

对应的指令是“`ldr sp, [pc, #40]`”，相当于 Start.S 文件的“ldr sp,=0x80200000”指令。

执行完后，寄存器 SP 的值等于 0x80200000。

```
80100000: e59fd028 ldr sp, [pc, #40] ; 80100030 <clean+0x14>
```

**2 每执行完一条机器码，会自动执行下一个内存地址 0x80100004 存储的 eb000001 机器码**

对应的指令是“`bl 80100010`”，相当于 Start.S 文件的“bl clean_bss”指令。

```
80100004: eb000001 bl 80100010 <clean_bss>
....
80100010 <clean_bss>:
80100010: e59f101c ldr r1, [pc, #28] ; 80100034 <clean+0x18>
80100014: e59f201c ldr r2, [pc, #28] ; 80100038 <clean+0x1c>
```

**3 跳转到内存地址 0x80100010 执行 e59f101c 机器码，对应的指令是 ldr r1，[pc，#28]**

相当于 Start.S 文件的“ldr r1，=__bss_start”指令。

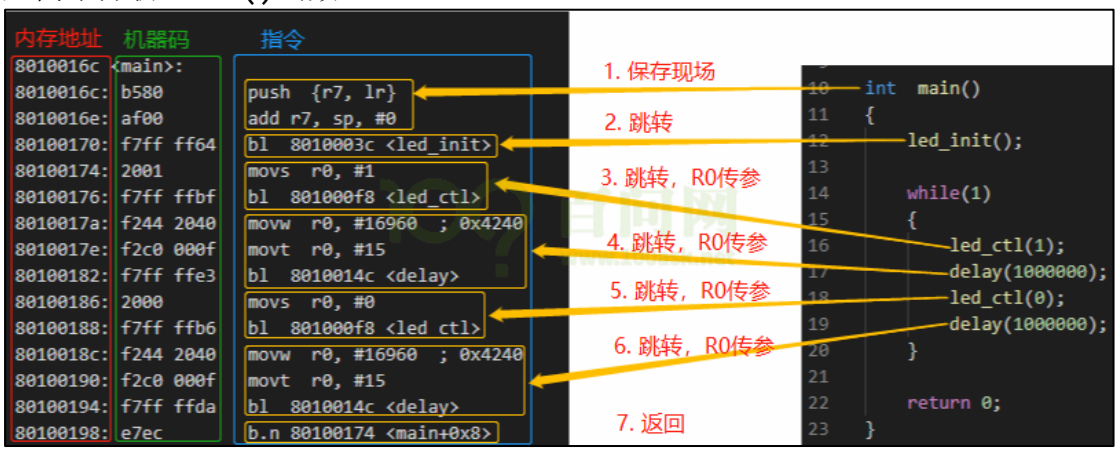
**4 此处 clean_bss 相当于一个函数，CPU 会逐条执行指令，直到执行“mov pc，lr”指令后，才返回。**

返回哪里？返回内存地址 0x80100008 处执行 fa000057 机器码，对应的指令是“blx 8010016c”。

对应 Start.S 文件的“bl main”指令。

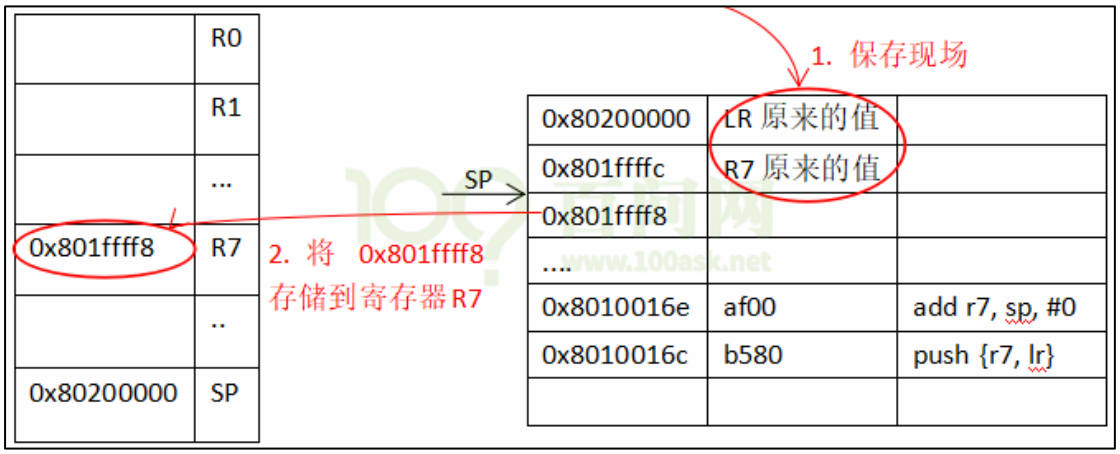
到此，CPU 跳转到 C 语言的 main()函数，继续执行。

为了让大家深入理解 C 语言函数的调用执行过程中，汇编指令如何执行，此处简单分析 main()函数




如上图所示

**1.** 进入 main()函数后，先将寄存器 R7、LR 入栈，保存现场/上下文，方便 main()函数执行完毕后返回，并且将当前栈指向的内存地址赋值给寄存器 R7，如图




**2.** 调用 led_init()函数，因为没有参数传递，所以直接调用 BL 指令进行跳转，即“bl 8010003c”指令。

**3.** 调用 led_ctl(1)函数，此处只有一个参数，通过寄存器 R0 进行传递，即“movs r0，#1”指令，然后通过 BL 指令进行跳转，即“bl 801000f8”指令，关于参数传递问题，可以参考前面《5.6 汇编程序调用 C 程序》。

**4.** 调用 delay(1000000)函数，此处只有一个参数，通过寄存器 R0 进行传

递，然后通过 BL 指令进行跳转。

**5．**调用 led_ctl(0)函数，此处只有一个参数，通过寄存器 R0 进行传递，然后通过 BL 指令进行跳转。

**6．**调用 delay(1000000)函数，此处只有一个参数，通过寄存器 R0 进行传递，然后通过 BL 指令进行跳转。

**7．**while(1)循环体到此已经结束，但是需要循环执行循环体的内容，通过 B 指令进行跳转到循环体开头，即“b.n 80100174”指令，执行内存地址 0x80100174 处的指令，也就是 led_ctl(1)函数对应的汇编指令“movs r0, #1”。

到此，进入并执行 main()函数对应的汇编指令分析已经结束，如果读者有兴趣可以分析一下，led_init()、led_ctl()与 delay()函数的汇编指令。

# 第6章 Makefile 与 GCC

## 6.1 交叉编译器

### 6.1.1 什么是交叉编译

简单地说，我们在 PC 机上编译程序时，这些程序是在 PC 机上运行的。我们想让一个程序在 ARM 板子上运行，怎么办？

ARM 板性能越来越强，可以认为 ARM 板就相当于一台 PC，当然可以在 ARM 板上安装开发工具，比如安装 ARM 版本的 GCC，这样就可以在 ARM 板上编译程序，在 ARM 板上直接运行这个程序。

但是，有些 ARM 板性能弱，或者即使它的性能很强也强不过 PC 机，所以更多时候我们是在 PC 机上开发、编译程序，再把这个程序下载到 ARM 板上去运行。

这就引入一个问题：

**1．我们使用工具比如说 gcc 编译出的程序是给 PC 机用的，这程序里的指令是 X86 指令。**

**2．那么能否使用同一套工具给 ARM 板编译程序？**

显示不行，因为 X86 的指令肯定不能在 ARM 板子上运行。所以我们需要使用另一套工具：交叉编译工具链。

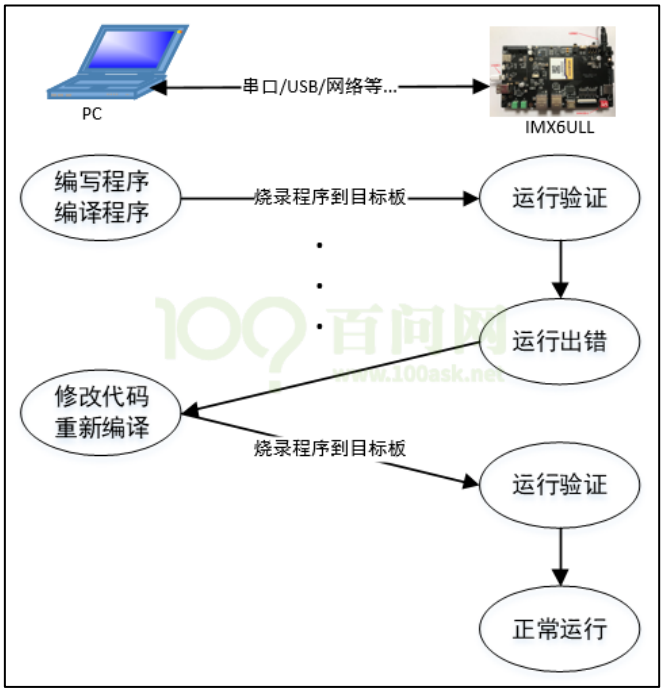
为何叫“交叉”？

首先，我们是在 PC 机上使用这套工具链来编译程序；

然后再把程序下载到 ARM 板运行；

如果程序不对，需要回到 PC 机修改程序、编译程序，再把程序下载到 ARM 板上运行、验证。如此重复。

在这个过程中，我们一会在 PC 上写程序、编译程序，一会在 ARM 板上运行、验证，中间来来回回不断重复，所以称之为“交叉”。对于所用的工具链，它是在 PC 机上给 ARM 板编译程序，称之为“交叉工具链”。



有很多种交叉工具链，举例如下：

1. Ubuntu 平台：交叉工具链有 arm-linux-gcc 编译器、arm-linux-gnueabihf-编译器。
2. Windows 平台：利用 ADS（ARM 开发环境），使用 armcc 编译器。
3. Windows 平台：利用 cygwin 环境，运行 arm-elf-gcc 编译器。

### 6.1.2 验证实例

代码:GIT 下载后在“**10_裸机开发/01_100ASK_IMX6ULL 裸机程序\6_Makefile 与 GCC\001_gcc_01**”目录下。

对于 main.c，我们采用 gcc 编译得到可执行程序，把它放在目标板上看看是否能运行起来，代码如下：

```
01  #include <stdio.h>
02
03  int main()
04  {
05      printf("100ask\n");
06      return 0;
07  }
```

在虚拟机编译运行：

```
$ gcc main.c -o 100ask
$ ./100ask
100ask
$
```

在上面的运行结果，没有任问题，然后我们将这个可执行程序放到目标板上，如下：

```
$ chmod 777 100ask
$ ./100ask
./100ask: line 1: syntax error: unexpected “(”
$
```

报错无法运行。说明为 X86 平台制作的可执行文件，不能在其他架构平台上运行。交叉编译就是为了解决这个问题。

为了方便实验，我们在 Ubuntu 中使用 gcc 来做实验，如果想使用交叉编译，参考章节《2.2 安装 SDK、设置工具链》，安装好工具链，设置好环境变量后，将所有的 gcc 替换为 arm-linux-gnueabihf-gcc 就可以完成交叉编译。

其中 gcc 是在 x86 电脑上运行，为 x86 机器编译程序。

arm-linux-gnueabihf-gcc 也是 x86 电脑上运行，为 RSIC（精简指令集）ARM 架构芯片编译程序。

## 6.2 GCC 常用选项及编译过程详解

### 6.2.1 gcc 编译过程详解

一个 C/C++文件要经过预处理(preprocessing)、编译(compilation)、汇编(assembly)和连接(linking)等 4 步才能生成可执行文件，编译流程图如下。

预处理：

C/C++源文件中，以“#”开头的命令被称为预处理命令，如包含命令“#include”、宏定义命令“#define”、条件编译命令“#if”、“#ifdef”等。预处理就是将要包含(include)的文件插入原文件中、将宏定义展开、根据条件编译命令选择要使用的代码，最后将这些东西输出到一个“.i”文件中等待进一步处理。

- **编译：对预处理后的源码进行词法和语法分析，生成目标系统的汇编代码文件，后缀名为“.s”。**
- **汇编：对汇编代码进行优化，生成目标代码文件，后缀名为“.o”。**
- **链接：解析目标代码中的外部引用，将多个目标代码文件连接为一个可执行文件。**

编译器利用这 4 个步骤中的一个或多个来处理输入文件，源文件的后缀名表示源文件所用的语言，后缀名控制着编译器的缺省动作

| 后缀名 | 语言种类 | 后期操作 |
|---|---|---|
| .c | C 源程序 | 预处理、编译、汇编 |
| .C | C++源程序 | 预处理、编译、汇编 |
| .cc | C++源程序 | 预处理、编译、汇编 |
| .cxx | C++源程序 | 预处理、编译、汇编 |
| .m | Objective-C 源程序 | 预处理、编译、汇编 |
| .i | 预处理后的 C 文件 | 编译、汇编 |
| .ii | 预处理后的 C++文件 | 编译、汇编 |
| .s | 汇编语言源程序 | 汇编 |
| .S | 汇编语言源程序 | 预处理、汇编 |
| .h | 预处理器文件 | 通常不出现在命令行上 |

其他后缀名的文件被传递给连接器(linker)，通常包括：

- **.o：目标文件(Object file，OBJ 文件)**
- **.a：归档库文件(Archive file)**

在编译过程中，除非使用了“-c”，“-S”或“-E”选项(或者编译错误阻止了完整的过程)，否则最后的步骤总是连接。在连接阶段中，所有对应于源程序的.o 文件，“-l”选项指定的库文件，无法识别的文件名(包括指定的“.o”目标文件和“.a”库文件)按命令行中的顺序传递给连接器。

## 6.2.2 gcc 命令

gcc 命令格式是：gcc [选项] 文件列表

gcc 命令用于实现 c 程序编译的全过程。文件列表参数指定了 gcc 的输入文件，选项用于定制 gcc 的行为。gcc 根据选项的规则将输入文件编译生成适当的输出文件。

gcc 的选项非常多，常用的选项，它们大致可以分为以下几类 。并且使用一个例子来描述这些选项。代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/6_Makefile 与 GCC/001_gcc_01”目录下。

代码为如下：

```
main.c:
01  #include <stdio.h>
02
03  #define HUNDRED 100
04
05  int main()
06  {
07      printf("%d ask\n",HUNDRED);
08      return 0;
09  }
```

## 6.2.3 过程控制选项

过程控制选项用于控制 gcc 的编译过程。无过程控制选项时，gcc 将默认执行全部编译过程，产生可执行代码。常用的过程控制选项有：

**1 预处理选项(-E)**

C/C++源文件中，以“#”开头的命令被称为预处理命令，如包含命令“#include”、宏定义命令“#define”、条件编译命令“#if”、“#ifdef”等。预处理就是将要包含(include)的文件插入原文件中、将宏定义展开、根据条件编译命令选择要使用的代码，最后将这些东西输出到一个“.i”文件中等待进一步处理。使用例子如下：

```
$ gcc -E main.c -o main.i
```

运行结果，生成 main.i，main.i 的内容（由于头文件展开内容过多，我将截取部分关键代码）：

```
extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 942 "/usr/include/stdio.h" 3 4
# 2 "main.c" 2
# 5 "main.c"
int main()
{
printf("%d ask\n",100);
return 0;
}
```

你会发现头文件被展开和 printf 函数中调用 HUNDRED 这个宏被展开。

**2 编译选项(-S)**

编译就是把 C/C++代码(比如上述的“.i”文件)“翻译”成汇编代码。使用例子如下：

```
$ gcc -S main.c -o main.s
```

运行结果，生成 main.s，main.s 的内容：

```
  1         .file   "main.c"
  2         .text
  3         .section        .rodata
  4 .LC0:
  5         .string "%d ask\n"
  6         .text
  7         .globl  main
  8         .type   main, @function
  9 main:
 10 .LFB0:
 11         .cfi_startproc
```

```
12        pushq   %rbp
13        .cfi_def_cfa_offset 16
14        .cfi_offset 6, -16
15        movq    %rsp, %rbp
16        .cfi_def_cfa_register 6
17        movl    $100, %esi
18        leaq    .LC0(%rip), %rdi
19        movl    $0, %eax
20        call    printf@PLT
21        movl    $0, %eax
22        popq    %rbp
23        .cfi_def_cfa 7, 8
24        ret
25        .cfi_endproc
26 .LFE0:
27        .size   main, .-main
28        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
29        .section        .note.GNU-stack,"",@progbits
```

**3 汇编选项(-c)**

汇编就是将上述的“.s”文件汇编代码翻译成符合一定格式的机器代码，在 Linux 系统上一般表现为 ELF 目标文件(OBJ 文件)

```
$ gcc -c main.c -o main.o
```

运行结果，生成 main.o(将源文件转为一定格式的机器代码)。

## 6.2.4 输出选项

输出选项用于指定 gcc 的输出特性等，常用的选项有：

**1 输出目标选项（-o filename）**

-o 选项指定生成文件的文件名为 filename。使用例子如下

```
$ gcc main.c -o main
```

运行结果，生成可执行程序 main，如下：

```
$ ls
main.c main
$ ./main
$ 100 ask
```

其中，如果无此选项时使用默认的文件名，各编译阶段有各自的默认文件名，可执行文件的默认名为 a.out。使用例子如下：

```
$ gcc main.c
```

运行结果，生成可执行文件 a.out，如下：

```
$ ls
a.out main.c
$ ./a.out
$ 100 ask
```

**2 输出所有警告选项（-Wall）**

显示所有的警告信息，而不是只显示默认类型的警告。建议使用。我们把上面的 main.c 稍微修改一下。

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/6_Makefile 与 GCC/001_gcc_02”目录下。

```
main.c:
01  #include <stdio.h>
02
03  #define HUNDRED 100
```

```
04
05  int main()
06  {
07      int a = 0;
08      printf("%d ask\n",HUNDRED);
09      return 0;
10  }
```

编译不添加-Wall 选项编译，没有任何警告信息，编译结果如下：

```
$ gcc main.c -o main.c
```

编译添加-Wall 选项编译，现实所有警告信息，编译结果如下：

```
$ gcc main.c -Wall -o main.c
main.c: In function ‘main’:
main.c:7:6: warning: unused variable ‘a’ [-Wunused-variable]
  int a=0;
      ^
```

### 6.2.5 头文件选项

头文件选项：-Idirname。

将 dirname 目录加入到头文件搜索目录列表中。当 gcc 在默认的路径中没有找到头文件时，就到本选项指定的目录中去找。

在上面的例子中创建一个 inc 目录，并在里面创建一个头文件 test.h。

然后 main.c 里面增加#include “test.h”。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/6_Makefile 与 GCC/001_gcc_03”目录下。

目录结构如下：

```
$ tree
.
├── inc
│   └── test.h
└── main.c

1 directory, 2 files
$
```

test.h 代码如下：

```
01  #ifndef __TEST_H
02  #define __TEST_H
03  /*
04      code
05  */
06  #endif
```

运行结果，这样就可以引用制定文件下的目录的头文件，如下：

```
$ gcc main.c -I inc -o main
```

如果不添加头文件选项，编译运行结果，如下：

```
$ gcc main.c -o main
    main.c:2:18: fatal error: test.h: No such file or directory
    compilation terminated.
```

会产生错误提示，无法找到 test.h 头文件。

### 6.2.6 链接库选项

（详细使用方法查看下一节：深入讲解 GCC 链接过程）

**1 添加库文件搜索目录（-Ldirname）**

将 dirname 目录加入到库文件的搜索目录列表中。

**2 加载库名选项（-lname）**

加载名为 libname.a 或 libname.so 的函数库。例如：-lm 表示链接名为 libm.so 的函数库。

**3 静态库选项（-static）使用静态库。**

注意：在命令行中，静态库必须放在目标文件之后。

比如：

```
gcc test.cpp -o test libexample.a  -static
```

### 6.2.7 代码优化选项

gcc 提供几种不同级别的代码优化方案，用“-Olevel”选项表示。level 取值可以是 0、1、2、3 和 s。

默认 0 级，即不进行优化。典型的优化选项：

① -O 或-O1：基本优化，使代码执行的更快。

② -O2：产生尽可能小和快的代码。如无特殊要求，不建议使用 O2 以上的优化。

③ -Os：生成最小的可执行文件，适合用于嵌入式软件。

### 6.2.8 调试选项及调试示例

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/6_Makefile 与 GCC/001_gcc_02”目录下。

gcc 支持数种调试选项：

| -g：产生能被 GDB 调试器使用的调试信息。 |
|---|

举个例子，首先需要在编译时加上“-g”选项，操作步骤如下：

```
$ gcc main.c -g -o main
```

### 6.2.9 GDB 调试示例：

**1 run 命令**

调试运行，使用 run 命令开始执行被调试的程序，run 命令的格式：

| run [运行参数] |
|---|

```
$ gdb -q main          <---进入调试程序
Reading symbols from output...done.
(gdb) run              <---开始执行程序
Starting program: /home/100ask/makefile/
100 ask
[Inferior 1 (process 7425) exited normally]
(gdb)
```

**2 list 命令**

列出源代码，使用 list 命令来查看源程序以及行号信息，list 命令的格式：

| list [行号] |
|---|

```
(gdb) list 1           <---列出第一行附近的源码，每次 10 行
#include <stdio.h>
```

```
#define HUNDRED 100

int main()
{
    int a = 100;

    printf("%d ask\n",HUNDRED);
    return 0;
(gdb) <Enter>            <---按 Enter 键，列出下 10 行源码
}
(gdb)
```

**3 设置断点**

① break 命令，设置断点命令，break 命令的格式： break <行号> | <函数名>

```
(gdb) break 7
Breakpoint 1 at 0x40052e: file main.c, line 7.
(gdb)
```

② info break 命令，查 看断点命令：

```
(gdb) info break
Num   Type       Disp Enb Address           What
1   breakpoint    keep y   0x000000000040052e in main at main.c:7
(gdb)
```

③ delete breakpoint 命令，删除断点命令， delete breakpoint 命令的格式： delete breakpoint <断点号>

```
(gdb) delete breakpoint 1
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

④ 跟踪运行结果

- print 命令，显示变量的值，print 命令的格式：print[/格式] <表达式>
- display 命令，设置自动现实命令，display 命令的格式： display <表达式>
- step 和 next 命令，单步执行命令，step 和 next 命令的格式：step <行号> 或 next <行号>
- continue 命令，继续执行命令。

```
(gdb) break 7
Breakpoint 1 at 0x40052e: file main.c, line 7.
(gdb) break 9
Breakpoint 2 at 0x400535: file main.c, line 9.
(gdb) run
Starting program:/home/100ask/makefile/

Breakpoint 1, main () at main.c:7
7       int a = 100;
(gdb) continue
Continuing.

Breakpoint 2, main () at main.c:9
9       printf("%d ask\n",HUNDRED);
```

```
(gdb) print a
$1 = 100
(gdb)
```

### 6.2.10 编译错误警告

在写代码的时候，其实应该养成一个好的习惯就是任何的警告错误，我们都不要错过，

编译错误是必须要解决的，否则无法生成目标文件。但是即使有编译警告，也可以生成目标文件。所以编译警告往往会被人忽略。但是有时候，编译警告提示的信息你必须去修改，否则会影响程序运行。接下来我们来简单分析一下 gcc 的编译警告如何处理，例子如下：

```
main.c
 01 #include <stdio.h>
 02 #include "hander.h"
 03
 04 int main()
 05 {
 06         float a = 0.0;
 07         int b = a;
 08         char c = 'a'
 09
 10         printf("100ask: \n",a);
 11
 12         return 0;
 13 }
```

上面文件中有三处错误：
第 2 行：包含了一个不存在的头文件。
第 8 行：语句后面没有加分号。
第 10 行：书写格式错误，变量 a 没有对应的输出格式。

我们对上面的文件进行编译，还记得上面讲的编译警告选项吗？在编译的时候加上它（-Wall），如下：

```
$ gcc main.c -Wall -o output
main.c: In function ‘main’:
main.c:2:20: fatal error: hander.h: No such file or directory
compilation terminated.
```

错误警告信息分析：在展开第二行的 hander.h 头文件的时候，产生编译错误，没有 hander.h 文件或者目录。接着我们把 hander.h 头文件去掉，再编译一次：

```
$ gcc -Wall main.c  -o output
main.c: In function ‘main’:
main.c:10:2: error: expected ‘,’ or ‘;’ before ‘printf’
  printf("100ask: \n",a);
  ^
main.c:8:7: warning: unused variable ‘c’ [-Wunused-variable]
  char c = 'a'
       ^
main.c:7:6: warning: unused variable ‘b’ [-Wunused-variable]
  int b = a;
      ^
```

错误警告信息分析：有一个错误和两个警告。一个错误是指第 10 行 prntf 之前缺少分号。两个警告是指第 7 行和第 8 行的变量没有使用。那么我继续解决错误信息和警告，将两个警告的变量删除和 printf 前添加分号，然后继续编译，如

下：

```
$ gcc -Wall main.c  -o output
main.c: In function ‘main’:
main.c:8:9: warning: too many arguments for format [-Wformat-extra-args]
  printf("100ask: \n",a);
         ^
```

错误警告信息分析：还是有警告信息，该警告指的是 printf 中的格式参数太多，也就是没有添加变量 a 的输出格式，继续解决错误信息和警告，添加变量 a 的输出格式，然后继续编译，如下：

```
$ gcc -Wall main.c  -o output
$ tree
.
├── main.c
└── output
```

最终编译成功，输出目标文件。

如果你忽略警告信息不去修改 printf 语句，程序无法输出你预期的结果。

## 6.3 深入讲解 GCC 链接过程

你会发现，可执行文件文件会比源代码大了。这是因为编译的最后一步是链接，它会解析代码中的外部应用，然后将你自己的 OBJ 文件，系统库的 OBJ 文件，库文件等等都链接起来。

我们用一个例子来说明上面描述。代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/6_Makefile 与 GCC/001_gcc_01”目录下。执行如下命令：

```
$ gcc main.c -c
$ gcc -o output main.o
$ gcc -o output_static main.o  --static
$ ls -alh
drwxrwxr-x 2 tym tym 4.0K 2月  20 07:27 .
drwxrwxr-x 6 tym tym 4.0K 2月  20 07:25 ..
-rw-rw-r-- 1 tym tym   96 2月  20 07:25 main.c
-rw-rw-r-- 1 tym tym 1.5K 2月  20 07:26 main.o
-rwxrwxr-x 1 tym tym 8.5K 2月  20 07:27 output
-rwxrwxr-x 1 tym tym 892K 2月  20 07:27 output_static
```

从上面的例子可以看出 output_static 比 output 大很多。

### 6.3.1 动态链接库和静态链接库使用例程

静态库和动态库，是根据链接时期的不同来划分。

静态库：在链接阶段，所用的库就被加进可执行程序里了。静态链接生成的可执行文件，已经内嵌了所有的库，可以独立运行。链接静态库从某种意义上来说是一种复制粘贴，被链接后库就直接嵌入可执行程序中了。如果有多个程序都用到这些库，并且都使用静态 接，那么系统里空间就有很大的浪费，而且一旦发现系统中有 bug，就必须把所有程序都重新编译、重新链接，十分麻烦。静态库是不是一无是处了呢？不是的，如果代码在其他系统上运行，且没有相应的库时，解决办法就是使用静态库。而且由于动态库是在程序运行的时候被链接，因此动态库的运行速度比较慢。

- **动态库：**

在程序执行的时候，才把所需要的库跟程序链接在一起。多个程序可以合用一份动态库，节省存储空间。如果发现 bug 或者是要升级，只要用新的库把原来的替换掉就可以了。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/6_Makefile 与 GCC/001_gcc_04”目录下。

下面我们创建三个文件 main.c，add.c，add.h，讲解静态库链接和动态库链接，如下：

```
main.c:
#include <stdio.h>
#include "add.h"

int main(int argc, char *argv[])
{
    printf("%d\n",add(10, 10));
    printf("%d\n",add(20, 20));
    return 0;
}

add.c:
#include "add.h"

int add(int a, int b)
{
    return a + b;
}

add.h:
#ifndef __ADD_H
#define __ADD_H

int add(int a, int b);

#endif
```

➢ **静态库链接**

静态库名字一般为“libxxx.a”。利用静态库编译生成的可执行文件比较大，因为整个函数库的所有数据都被整合进了可执行文件中。

优点：

- 不需要外部函数库支持。
- 加载速度快。

缺点：

- 静态库升级时，程序需要重新编译。
- 多个程序调用相同库，静态库会重复调入内存，造成内存的浪费。

静态库的制作，如下：

```
$ gcc add.c -o add.o -c
$ ar -rc libadd.a add.o
```

静态库的使用，例子如下：

```
$ gcc main.c -o output -ladd -L.
```

解析一下，“-L”用来表示库在哪里，上例中库在当前目录里(“.”表示当前目

录)，“-l”用来表示库的名字(-ladd 表示库 libadd)。
运行结果：

```
$ ./output
20
40
```

➢ **动态库链接**

动态库名字一般为“libxxx.so”，又称共享库。动态库在编译的时候没有被编译进可执行文件，所以可执行文件比较小。程序运行时，操作系统帮我们找到库，并跟程序链接起来。

**优点：多个程序可以使用同一个动态库，节省内存。**

**缺点：加载速度慢。**

动态库的制作，如下：

```
$ gcc -shared -fPIC lib.c -o libtest.so
$ sudo cp  libtest.so /usr/lib/
```

动态库的使用，如下：

```
$ gcc main.c -L. -ltest -o output
```

链接时指定库在当前目录，但是运行时用的是/usr/lib 或/lib 目录下的库。
运行结果：

```
$ ./output
20
40
```

## 6.4 Makefile 的引入及规则

### 6.4.1 为什么需要 Makefile?

在上一章节的例子中，我们都是在终端执行 gcc 命令来完成源文件的编译。感觉挺方便的，这是因为工程中的源文件只有一两个，在终端直接执行编译命令，确实快捷方便。

但是现在一些项目工程中的源文件不计其数，其按类型、功能、模块分别放在若干个目录中，如果仍然在终端输入这些命令来编译，那显然不切实际，开发效率极低。

我们需要一个工具来管理这些编译过程，这就是“make”。

make 是一个应用程序，它根据 Makefile 来做事。Makefile 负责管理整个编译流程：要编译哪些文件？怎么编译这些文件？怎么把它们链接成一个可执行程序。Makefile 定义了一系列的规则来实现这些管理。

### 6.4.2 Makefile 的引入

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/6_Makefile 与 GCC/001_Makefile_01”目录下。

Makefile 的引入是为了简化我们编译流程，提高我们的开发进度。下面我们用一个例子来说明 Makefile 如何简化我们的编译流程。我们创建一个工程内容分别 main.c，sub.c，sub.h，add.c，add.h 五个文件。sub.c 负责计算两个数减法运算，add.c 负责计算两个数加法运算，然后编译出可执行文件。其源文件内容如下：

```
main.c：
```

```
#include <stdio.h>
#include "add.h"
#include "sub.h"

int main()
{
    printf("100 ask, add:%d\n", add(10, 10));
    printf("100 ask, sub:%d\n", sub(20, 10));
    return 0;
}

add.c：
#include "add.h"

int add(int a, int b)
{
    return a + b;
}
```

```
add.h：
#ifndef __ADD_H
#define __ADD_H

int add(int a, int b);

#endif
```

```
sub.c：
#include "sub.h"

int sub(int a, int b)
{
    return a - b;
}
```

```
sub.h：
#ifndef __SUB_H
#define __SUB_H

int sub(int a, int b);

#endif
```

我们使用 gcc 对上面工程进行编译及生成可执行程序，在终端输入如下命令，如下：

```
$ gcc main.c sub.c add.c -o ouput
$ ls
add.c  add.h  main.c  output  sub.c  sub.h
$ ./output
100 ask, add:20
100 ask, sub:10
```

上面的命令是通过 gcc 编译器对 main.c sub.c add.c 这三个文件进行编译，及生成可执行程序 output，并执行可执行文件产生结果。

从上面的例子看起来比较简单，一条命令完成三个源程序的编译并产生结果，这是因为目前只有三个源文件。

如果有上千个源文件，即使你只修改了其中一个文件，你执行这样的命令时，它会把所有的源文件都编译一次。这样消耗的时间是非常恐怖的。

我们想实现：哪个文件被修改了，只编译这个被修改的文件即可，其它没有修改的文件就不需要再次重新编译了。可以使用下列命令，先单独编译文件，再最后链接，如下：

```
$ gcc -c main.c
$ gcc -c sub.c
$ gcc -c add.c
$ gcc main.o sub.o add.o -o output
```

我们将上面一条命令变成了四条，分别编译出源文件的目标文件，最后再将所有的目标文件链接成可执行文件。

当其中一个源文件的内容发生了变化，我们只需要单独编译它，然后跟其他文件重新链接成可知执行文件，不再需要重新编译其他文件。

假如我们修改了 add.c 文件，只需要重新编译生成 add.c 的目标文件，然后再将所有的.o 文件链接成可执行文件，如下：

```
$ gcc -c add.c
$ gcc main.o sub.o add.o -o output
```

这样的方式虽然可以节省时间，但是仍然存在几个问题，如下：

**1．如果源文件的数目很多，那么我们需要花费大量的时间，敲命令执行。**

**2．如果源文件的数目很多，然后修改了很多文件，后面发现忘记修改了什么。**

**3．如果头文件的内容修改，替换，更换目录，所有依赖于这个头文件的源文件全部需要重新编译。**

这些问题我们不可能一个一个去找和排查，引入 Makefile 可以解决上述问题。我们先扔出一个 Makefile，如下：

```
Makefile：
output: main.o add.o sub.o
	gcc -o output main.o add.o sub.o
main.o: main.c
	gcc -c main.c
add.o: add.c
	gcc -c add.c
sub.o: sub.c
	gcc -c sub.c

clean:
	rm *.o output
```

Makefile 编写好后只需要执行 make 命令，就可以自动帮助我们编译工程。注意，make 命令必须要在 Makefile 的当前目录执行，如下：

```
$ ls
add.c  add.h  main.c  Makefile  sub.c  sub.h
$ make
gcc -c main.c
gcc -c add.c
gcc -c sub.c
gcc -o output main.o add.o sub.o
$ ls
add.c  add.h  add.o  main.c  main.o  Makefile  output  sub.c  sub.h  sub.o
```

通过 make 命令就可以生成相对应的目标文件.o 和可执行文件。
如果我们再次使用 make 命令编译，如下，它说你的程序已经是最新的了，不需要再做什么事情：

```
$ make
make: 'output' is up to date.
```

我们可以修改一下 add.c，然后再执行 make，如下：

```
$ make
gcc -c add.c
gcc -o output main.o add.o sub.o
```

会发现，它重新编译了 add.c，并且重新生成了可执行程序。

通过上述例子，Makefile 将我们上面的三个问题都解决了，无需手工输入复杂的命令，只编译修改过的文件。在一个很庞大的工程中，只有第一次编译时间比较长，第二次编译时会大大缩短时间，节省了我们的开发周期。

下面我们来学习 Makefile 的知识。

### 6.4.3 Makefile 的规则

➢ **命名规则：**

一般来说将 Makefile 文件取名为“Makefile”或“makefile”都可以，惯例是使用首字母大写的“Makefile”。也可以使用其他名字，比如 makefile.linux，但你需要用“-f”参数指定，示例如下：

```
make -f makefile.linux
```

➢ **基本语法规则：**

目标（target）：依赖（prerequisites）
[Tab]命令（command）

- target：需要生成的目标文件
- prerequisites：生成该 target 所依赖的一些文件
- command：生成该目标需要执行的命令

三者的关系：target 依赖于 prerequisites 中的文件，其生成规则定义在 command 中。

举例，比如我们平时要编译一个文件：

```
$ gcc main.c -o main
```

换成 Makefile 的书写格式如下,,:

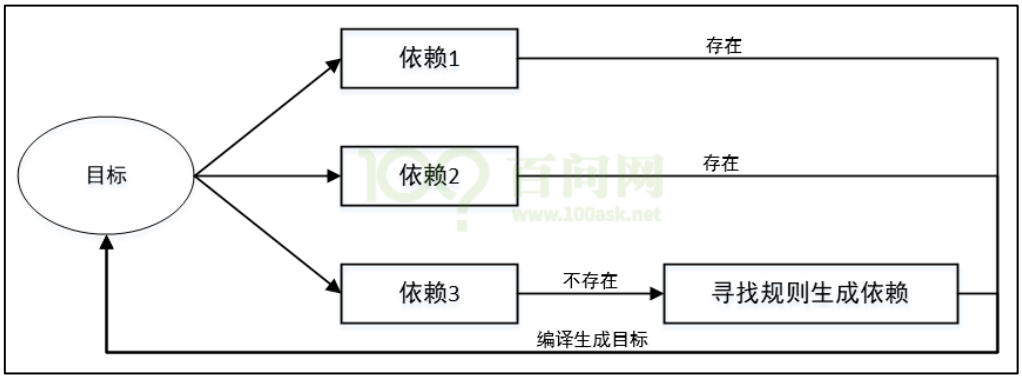
```
01  main:main.c
02      gcc main.c -o main
```

第 1 行表示 main 这个可执行程序依赖于 main.c，第 2 行表示需要用“gcc main.c -o main”这个命令来生成 main。
注意：Makefile 文件里的命令必须要使用 Tab，不能使用空格。

➢ **目标生成规则：**

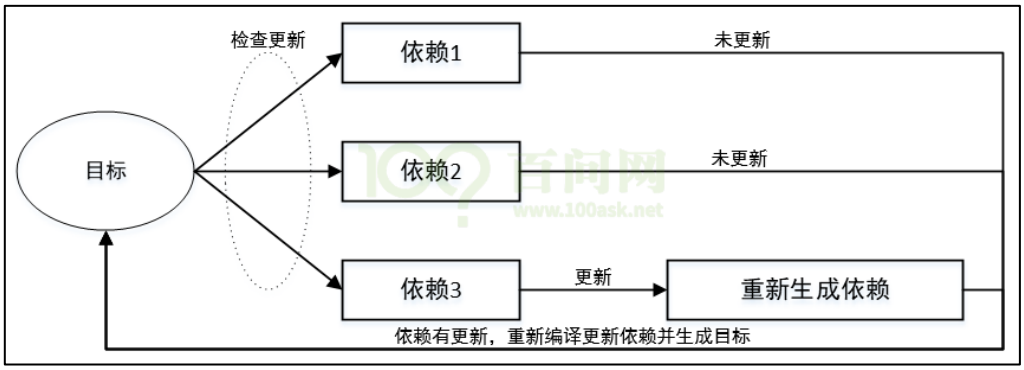
- 目标生成：
  - 检查规则中的依赖文件是否存在。
  - 若依赖文件不存在，则寻找是否有规则用来生成该依赖文件。

目标生成流程，如下：




- 目标更新：
  - 检查目标的所有依赖，任何一个依赖有更新时，就要重新生成目标。
  - 目标文件比依赖文件更新时间晚，则需要更新。

目标更新流程，如下：




我们使用上面的例子，Makefile 内容如下：

```
output: main.o add.o sub.o
	gcc -o output main.o add.o sub.o
main.o: main.c
	gcc -c main.c
add.o: add.c
	gcc -c add.c
sub.o: sub.c
	gcc -c sub.c

clean:
	rm *.o output
```

编译执行：

```
$ make
gcc -c main.c
gcc -c add.c
gcc -c sub.c
gcc -o output main.o add.o sub.o
```

make 命令会检测寻找目标的依赖是否存在，不存在，则会寻找生成依赖的命令。

① . output 依赖于“main.o add.o sub.o”，这三个文件都没有，分别去生成它们。怎么生成？继续寻找规则，发现 main.o 依赖于 main.c，于是使用“gcc -c main.c”来生成；其他两个文件类似处理。

② 现在 output 的依赖文件都有了，但是 output 文件还没有，所以使用“gcc -o output main.o add.o sub.o”来生成 output 文件。

当我们修改某一个文件时，比如之修改 add.c 文件，然后重新 make，如下：

```
$ make
gcc -c add.c
gcc -o output main.o add.o sub.o
```

make 命令还是根据 Makefile 来检查、执行：

① output 依赖于“main.o add.o sub.o”，这三个文件都有了，main.o 依赖于 main.c，main.c 没改过，所以不需要重新生成 main.o；但是 add.o 的依赖文件 add.c 被改过，需要重新生成 add.o，使用命令“gcc -c add.c”；sub.o 跟 main.o 类似，也不需要重新生成。

② 现在 output、它的三个依赖文件都有了，但是其中一个依赖文件 add.o 比 output 文件新，所以使用“gcc -o output main.o add.o sub.o”来生成 output 文件。

## 6.5 Makefile 的语法

### 6.5.1 变量的定义及取值

Makefile 也支持变量定义，变量的定义也让的我们的 Makefile 更加简化，可复用。

变量的定义：一般采用大写字母，赋值方式像 C 语言的赋值方式一样，如下：

```
DIR = ./100ask/
```

变量取值：使用括号将变量括起来再加美元符，如下：

```
FOO = $(DIR)
```

变量可让 Makefile 简化可复用，上面个的 Makefile 文件，内容如下：

```
output: main.o add.o sub.o
	gcc -o output main.o add.o sub.o
main.o: main.c
	gcc -c main.c
add.o: add.c
	gcc -c add.c
sub.o: sub.c
	gcc -c sub.c

clean:
	rm *.o output
```

我们可以将其优化，如下：

```
#Makefile 变量定义
OBJ = main.o add.o sub.o
output: $(OBJ)
	gcc -o output $(OBJ)
main.o: main.c
	gcc -c main.c
add.o: add.c
	gcc -c add.c
sub.o: sub.c
	gcc -c sub.c

clean:
	rm $(OBJ) output
```

我们分析一下上面简化过的 Makefile，第一行是注释，Makefile 的注释采用‘#’，而且不支持像 C 语言中的多行注释。第二行我们定义了变量 OBJ，并赋值字符串”main.o，add.o，sub.o“。其中第三，四，十三行，使用这个变量。这样用到用一个字符串的地方直接调用这个变量，无需重复写一大段字符串。

Makefile 除了使用‘=’进行赋值，还有其他赋值方式，比如‘:=’和‘?=’，接下来我们来对比一下这几种的区别：

- **赋值符‘=’**

我们使用一个例子来说明赋值符‘=’的用法。Makefile 内容如下：

```
01  PARA = 100
02  CURPARA = $(PARA)
03  PARA = ask
04
05  print:
06      @echo $(CURPARA)
```

分析代码：第一行定义变量 PARA，并赋值为“100”，第二行定义变量 CURPARA，并赋值引用变量 PARA，此时 CURPARA 的值和 PARA 的值是一样的，第三行，将变量 PARA 的变量修改为“ask”。第六行输出 CURPARA 的值，echo 前面增加@符号，代表只显示命令的结果，不显示命令本身。

通过命令“make print”执行 Makefile，如下：

```
$ make print
ask
```

从结果上看，变量 CURPARA 的值并不是“100”，而是 PARA 的最后一次赋值。使用赋值符“=”设置的变量，它的值在运行时才能确定，这称为“延时变量”。

其实可以理解为在 C 语言中，定义一个指针变量指向一个变量的地址。如下：

```
01  int a = 10;
02  int *b = &a;
03  a=20;
```

- **赋值符‘:=’**

我们使用一个例子来说明赋值符‘:=’的用法。Makefile 内容如下：

```
01  PARA = 100
02  CURPARA := $(PARA)
03  PARA = ask
04
05  print:
06      @echo $(CURPARA)
```

代码分析：我们见上面的 Makefile 的第二行的“=”替换成“:=”，重新编译，如下：

```
$ make print
100
$
```

从结果上看，变量 CURPARA 的值为“100”。使用“:=”设置的变量被称为“即时变量”，在赋值时就确定了它的值。

- **赋值符‘?=’**

我们用两个 Makefile 来说明赋值符“?=”的用法。如下：

```
第一个 Makefile：
PARA = 100
```

```
PARA ?= ask

print:
    @echo $(PARA)
```

编译结果：

```
$ make print
100
```

```
第二个 Makefile:
PARA ?= ask

print:
    @echo $(PARA)
```

编译结果：

```
$ make print
ask
```

上面的例子说明，使用“?=”给变量设置值时，如果这个变量之前没有被设置过，那么“?=”才会起效果；如果曾经设置过这个变量，那么“?=”不会起效果。

赋值符‘+=’

Makefile 中的变量是字符串，有时候我们需要给前面已经定义好的变量添加一些字符串进去,此时就要使用到符号“+=”，比如如下：

```
01  OBJ = main.o add.o
02  OBJ += sub.o
```

这样的结果是 OBJ 的值为:”main.o，add.o，sub.o“。说明“+=”用作与变量的追加。

## 6.5.2 系统自带变量

系统自定义了一些变量，通常都是大写，比如 CC，PWD，CLFAG 等等，有些有默认值，有些没有，比如以下几种，如下：

① CPPFLAGS：预处理器需要的选项，如：-I

② CFLAGS：编译的时候使用的参数，-Wall -g -c

③ LDFLAGS：链接库使用的选项，-L -l

其中：默认值可以被修改，比如 CC 默认值是 cc，但可以修改为 gcc：CC=gcc

使用的例子，如下：

```
01  OBJ = main.o add.o sub.o
02  output: $(OBJ)
03          gcc -o output $(OBJ)
04  main.o: main.c
05          gcc -c main.c
06  add.o: add.c
07          gcc -c add.c
08  sub.o: sub.c
09          gcc -c sub.c
10
11  clean:
12          rm $(OBJ) output
```

使用系统自带变量，如下：

```
01  CC = gcc
02  OBJ = main.o add.o sub.o
03  output: $(OBJ)
04          $(CC) -o output $(OBJ)
05  main.o: main.c
```

```
06          $(CC) -c main.c
07  add.o: add.c
08          $(CC) -c add.c
09  sub.o: sub.c
10          $(CC) -c sub.c
11
12  clean:
13          rm $(OBJ) output
```

在上面例子中，系统变量 CC 不改变默认值，也同样可以编译，修改的目的是为了明确使用 gcc 编译。

### 6.5.3 自动变量

Makefile 的语法提供一些自动变量，这些变量可以让我们更加快速地完成 Makefile 的编写，其中自动变量只能在规则中的命令使用，常用的自动变量如下：

① $@：规则中的目标
② $<：规则中的第一个依赖文件
③ $^：规则中的所有依赖文件

我们上面的例子继续完善，修改为采用自动变量的格式，如下：

```
01  CC = gcc
02  OBJ = main.o add.o sub.o
03  output: $(OBJ)
04          $(CC) -o $@ $^
05  main.o: main.c
06          $(CC) -c $<
07  add.o: add.c
08          $(CC) -c $<
09  sub.o: sub.c
10          $(CC) -c $<
11
12  clean:
13          rm $(OBJ) output
```

其中：第 4 行$^表示变量 OBJ 的值，即 main.o add.o sub.o，第四，第六，第八行的$<分别表示 main.c add.c sub.c。$@表示 output。

### 6.5.4 模式规则

模式规则实在目标及依赖中使用%来匹配对应的文件，我们依旧使用上面的例子，采用模式规则格式，如下：

```
01  CC = gcc
02  OBJ = main.o add.o sub.o
03  output: $(OBJ)
04          $(CC) -o $@ $^
05  %.o: %.c
06          $(CC) -c $<
07
08  clean:
09          rm $(OBJ) output
```

其中：第五行%.o: %.表示如下。

① main.o 由 main.c 生成
② add.o 由 add.c 生成
③ sub.o 由 sub.c 生成

### 6.5.5 伪目标

在前面的例子中，我们直接执行“make”命令，它的目的是去执行第 1 个规则，这跟执行“make output”的效果是一样的。在这里，“output”既是规则的目标，也是一个实际的文件。

而伪目标是什么呢？对于以前的例子，先执行 make 命令，然后再执行“make clean”命令，如下：

```
$make
gcc -c main.c
gcc -c add.c
gcc -c sub.c
gcc -o output main.o add.o sub.o
$make clean
rm *.o output
```

一切正常！接着我们做个手脚，在 Makefile 目录下创建一个 clean 的文件，然后依旧执行 make 和 make clean，如下：

```
$touch clean
$make
gcc -c main.c
gcc -c add.c
gcc -c sub.c
gcc -o output main.o add.o sub.o
$make clean
make: 'clean' is up to date.
```

为什么“make clean”时命令没有被执行？因为已经有名为 clean 的文件了，并且它的依赖是空的，执行规则的条件没满足。

伪目标就是为了解决这个问题，我们在 clean 前面增加“.PHONY:clean”，如下：

```
01 CC = gcc
02 OBJ = main.o add.o sub.o
03 output: $(OBJ)
04         $(CC) -o $@ $^
05 %.o: %.c
06         $(CC) -c $<
07
08 .PHONY:clean
09 clean:
10         rm $(OBJ) output
```

运行结果：

```
$make
gcc -c main.c
gcc -c add.c
gcc -c sub.c
gcc -o output main.o add.o sub.o
$make clean
rm *.o output
```

当一个目标被声明为伪目标后，make 在执行规则时就会默认它满足执行条件。这样就提高了 make 的执行效率，也不用担心由于目标和文件名重名了。

伪目标的两大好处：

① 避免只执行命令的目标和工作目录下的实际文件出现名字冲突。

② 提高执行 Makefile 时的效率

### 6.5.6 Makefile 函数

Makefile 提供了大量的函数，函数调用的格式如下：

```
$(function arguments)
```

这里`function’是函数名，`arguments’是该函数的参数。参数和函数名之间是用空格或 Tab 隔开，如果有多个参数，它们之间用逗号隔开。这些空格和逗号不是参数值的一部分。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/6_Makefile 与 GCC/001_Makefile_02”目录下。

我们经常使用的函数主要有两个(wildcard，patsubst)，先把它们单独拎出来讲讲。

创建一个文件夹 src，在里下面创建两个文件，100.c，ask.c。如下：

```
.
├── Makefile
└── src
    ├── 100.c
    └── ask.c
```

- **wildcard 函数**

用于查找指定目录下指定类型的文件，函数参数：目录+文件类型，如下：

```
$(wildcard 指定文件类型)
```

其中，指定文件类型，如果不写路径，则默认为当前目录查找，例子如下：

```
01  SRC = $(wildcard ./src/*.c)
02
03  print:
04          @echo $(SRC)
```

执行命令 make，结果如下：

```
$ make
./src/ask.c ./src/100.c
```

其中，这条规则表示：找到目录./src 下所有后缀为.c 的文件，并赋值给变量 SRC。命令执行完，SRC 变量的值：./src/ask.c ./src/100.c

- **patsubst 函数**

用于匹配替换。函数参数：原模式+目标模式+文件列表，如下：

```
$( patsubst 原模式，目标模式，文件列表)
```

其中，从文件列表中查找出符合原模式文件类型的文件，然后一一替换成目标模式。举例：将./src 目录下的.c 结尾的文件，替换成.o 文件，并赋值给 obj。如下：

```
SRC = $(wildcard ./src/*.c)
OBJ = $(patsubst %.c, %.o, $(SRC))

print:
        @echo $(OBJ)
```

执行命令 make，结果如下：

```
$ make
./src/ask.o ./src/100.o
```

其中，这条规则表示：把变量中所有后缀为.c 的文件替换为.o。 命令执行完，OBJ 变量的值：./src/ask.o ./src/100.o

字符串替换和分析函数

① $(subst from,to,text)

在文本`text’中使用`to’替换每一处`from’。比如：

```
$(subst ee,EE,feet on the street)
```

结果为‘fEEt on the strEEt’。

② $(patsubst pattern,replacement,text)，前面讲过。

寻找`text’中符合格式`pattern’的字，用`replacement’替换它们。`pattern’和`replacement’中可以使用通配符。比如：

```
$(patsubst %.c,%.o,x.c.c bar.c)
```

结果为：`x.c.o bar.o’。

③ $(strip string)

去掉前导和结尾空格，并将中间的多个空格压缩为单个空格。比如：

```
$(strip a   b c )
```

结果为`a b c’。

④ $(findstring find,in)

在字符串`in’中搜寻`find’，如果找到，则返回值是`find’，否则返回值为空。比如：

```
$(findstring a,a b c)
$(findstring a,b c)
```

将分别产生值`a’和`’(空字符串)。

⑤ $(filter pattern...,text)

返回在`text’中由空格隔开且匹配格式`pattern...’的字，去除不符合格式`pattern...’的字。比如：

```
$(filter %.c %.s,foo.c bar.c baz.s ugh.h)
```

结果为`foo.c bar.c baz.s’。

⑥ $(filter-out pattern...,text)

返回在`text’中由空格隔开且不匹配格式`pattern...’的字，去除符合格式`pattern...’的字。它是函数 filter 的反函数。比如：

```
$(filter %.c %.s,foo.c bar.c baz.s ugh.h)
```

结果为`ugh.h’。

⑦ $(sort list)

将‘list’中的字按字母顺序排序，并去掉重复的字。输出由单个空格隔开的字的列表。比如：

```
$(sort foo bar lose)
```

返回值是‘bar foo lose’。

- 文件名函数

① $(dir names...)

抽取‘names...’中每一个文件名的路径部分，文件名的路径部分包括从文件名的首字符到最后一个斜杠(含斜杠)之前的一切字符。比如：

```
$(dir src/foo.c hacks)
```

结果为‘src/ ./’。

② $(notdir names...)

抽取‘names...’中每一个文件名中除路径部分外一切字符（真正的文件名）。比如：

```
$(notdir src/foo.c hacks)
```

结果为‘foo.c hacks’。

③ $(suffix names...)

抽取‘names...’中每一个文件名的后缀。比如：

```
$(suffix src/foo.c src-1.0/bar.c hacks)
```

结果为‘.c .c’。

④ $(basename names...)

抽取‘names...’中每一个文件名中除后缀外一切字符。比如：

```
$(basename src/foo.c src-1.0/bar hacks)
```

结果为‘src/foo src-1.0/bar hacks’。

⑤ $(addsuffix suffix,names...)

参数‘names...’是一系列的文件名，文件名之间用空格隔开；suffix 是一个后缀名。将 suffix(后缀)的值附加在每一个独立文件名的后面，完成后将文件名串联起来，它们之间用单个空格隔开。比如：

```
$(addsuffix .c,foo bar)
```

结果为‘foo.c bar.c’。

⑥ $(addprefix prefix,names...)

参数‘names’是一系列的文件名，文件名之间用空格隔开；prefix 是一个前缀名。将 preffix(前缀)的值附加在每一个独立文件名的前面，完成后将文件名串联起来，它们之间用单个空格隔开。比如：

```
$(addprefix src/,foo bar)
```

结果为‘src/foo src/bar’。

⑦ $(wildcard pattern) ，前面讲过

参数‘pattern’是一个文件名格式，包含有通配符(通配符和 shell 中的用法一样)。函数 wildcard 的结果是一列和格式匹配的且真实存在的文件的名称，文件名之间用一个空格隔开。

比如若当前目录下有文件 1.c、2.c、1.h、2.h，则：

```
c_src := $(wildcard *.c)
```

结果为‘1.c 2.c’。

- **其他函数**

① $(foreach var,list,text)

前两个参数，‘var’和‘list’将首先扩展，注意最后一个参数‘text’此时不扩展；接着，‘list’扩展所得的每个字，都赋给‘var’变量；然后‘text’引用该变量进行扩展，因此‘text’每次扩展都不相同。

函数的结果是由空格隔开的‘text’在‘list’中多次扩展后，得到的新‘list’，就是说：‘text’多次扩展的字串联起来，字与字之间由空格隔开，如此就产生了函数 foreach 的返回值。

下面是一个简单的例子，将变量‘files’的值设置为 ‘dirs’中的所有目录下的所有文件的列表：

```
dirs := a b c d
files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))
```

这里‘text’是‘$(wildcard $(dir)/*)’，它的扩展过程如下：

a) 第一个赋给变量 dir 的值是`a’，扩展结果为‘$(wildcard a/*)’；

b) 第二个赋给变量 dir 的值是`b’，扩展结果为‘$(wildcard b/*)’；

c) 第三个赋给变量 dir 的值是`c’，扩展结果为‘$(wildcard c/*)’；

d) 如此继续扩展。

这个例子和下面的例有共同的结果：

```
files := $(wildcard a/* b/* c/* d/*)
```

② $(if condition,then-part[,else-part])

首先把第一个参数‘condition’的前导空格、结尾空格去掉，然后扩展。如果扩展为非空字符串，则条件‘condition’为‘真’；如果扩展为空字符串，则条件‘condition’为‘假’。

如果条件‘condition’为‘真’,那么计算第二个参数‘then-part’的值，并将该值作为整个函数 if 的值。

如果条件‘condition’为‘假’,并且第三个参数存在，则计算第三个参数‘else-part’的值，并将该值作为整个函数 if 的值；如果第三个参数不存在，函数 if 将什么也不计算，返回空值。

注意：仅能计算‘then-part’和‘else-part’二者之一，不能同时计算。这样有可能产生副作用（例如函数 shell 的调用）。

③ $(origin variable)

变量‘variable’是一个查询变量的名称，不是对该变量的引用。所以，不能采用‘$’和圆括号的格式书写该变量，当然，如果需要使用非常量的文件名，可以在文件名中使用变量引用。

函数 origin 的结果是一个字符串，该字符串变量是这样定义的：

- ‘undefined'：如果变量‘variable’从没有定义；
- ‘default'：变量‘variable’是缺省定义；
- ‘environment'：变量‘variable’作为环境变量定义，选项‘-e’没有打开；
- ‘environment override'：变量‘variable’作为环境变量定义，选项‘-e’已打开；
- ‘file' ：变量‘variable’在 Makefile 中定义；
- ‘command line'：变量‘variable’在命令行中定义；
- ‘override'：变量‘variable’在 Makefile 中用 override 指令定义；
- ‘automatic' ：变量‘variable’是自动变量

④ $(shell command arguments)

函数 shell 是 make 与外部环境的通讯工具。函数 shell 的执行结果和在控制台上执行‘command arguments’的结果相似。不过如果‘command arguments’的结果含有换行符（和回车符），则在函数 shell 的返回结果中将把它们处理为单个空格，若返回结果最后是换行符（和回车符）则被去掉。

比如当前目录下有文件 1.c、2.c、1.h、2.h，则：

```
c_src := $(shell ls *.c)
```

结果为‘1.c 2.c’。

## 6.6 Makefile 实例

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/6_Makefile 与 GCC/001_Makefile_03”目录下。

在上面的例子中，我们都是把头文件，源文件放在同一个文件里面，这样不好用于维护，所以我们将其分类，把它变得更加规范一下，把所有的头文件放在文件夹：inc，把所有的源文件放在文件夹：src。

代码目录如下：

```
$ tree
.
├── inc
│   ├── add.h
│   └── sub.h
├── Makefile
└── src
    ├── add.c
    ├── main.c
    └── sub.c
```

其中 Makefile 的内容如下：

```
01  SOURCE = $(wildcard ./src/*.c)
02  OBJECT = $(patsubst %.c, %.o, $(SOURCE))
03
04  INCLUEDS = -I ./inc
05
06  TARGET  = 100ask
07  CC      = gcc
08  CFLAGS  = -Wall -g
09
10  $(TARGET): $(OBJECT)
11          @mkdir -p output/
12          $(CC) $^ $(CFLAGES) -o output/$(TARGET)
13
14  %.o: %.c
15          $(CC) $(INCLUEDS) $(CFLAGES) -c $< -o $@
16
17  .PHONY:clean
18  clean:
19          @rm -rf $(OBJECT) output/
```

分析：

- 行 1：获取当前目录下 src 所有.c 文件，并赋值给变量 SOURCE。
- 行 2：将./src 目录下的.c 结尾的文件，替换成.o 文件，并赋值给变量 OBJECT。
- 行 4：通过-I 选项指明头文件的目录，并赋值给变量 INCLUDES。
- 行 6：最终目标文件的名字 100ask，赋值给 TARGET。
- 行 7：替换 CC 的默认之 cc，改为 gcc。
- 行 8：将显示所有的警告信息选项和 gdb 调试选项赋值给变量 CFLAGS。
- 行 11：创建目录 output，并且不再终端现实该条命令。

- 行 12：编译生成可执行程序 100ask，并将可执行程序生成到 output 目录
- 行 15：将源文件生成对应的目标文件。
- 行 17：伪目标，避免当前目录有同名的 clean 文件。
- 行 19：用与执行命令 make clean 时执行的命令，删除编译过程生成的文件。

最后编译的结果，如下：

```
$ make
gcc -I ./inc  -c src/main.c -o src/main.o
gcc -I ./inc  -c src/add.c -o src/add.o
gcc -I ./inc  -c src/sub.c -o src/sub.o
gcc src/main.o src/add.o src/sub.o  -o output/100ask
$tree
.
├── inc
│   ├── add.h
│   └── sub.h
├── Makefile
├── output
│   └── 100ask
└── src
    ├── add.c
    ├── add.o
    ├── main.c
    ├── main.o
    ├── sub.c
    └── sub.o
```

上面的 Makefile 文件算是比较完善了，不过项目开发中，代码需要不断的迭代，那么必须要有东西来记录它的变化，所以还需要对最终的可执行文件添加版本号，如下：

```
01  VERSION = 1.0.0

02  SOURCE  = $(wildcard ./src/*.c)
03  OBJECT   = $(patsubst %.c, %.o, $(SOURCE))
04
05  INCLUEDS = -I ./inc
06
07  TARGET  = 100ask
08  CC      = gcc
09  CFLAGS  = -Wall -g
10
11  $(TARGET): $(OBJECT)
12          @mkdir -p output/
13          $(CC) $^ $(CFLAGES) -o output/$(TARGET)_$(VERSION)
14
15  %.o: %.c
16          $(CC) $(INCLUEDS) $(CFLAGES) -c $< -o $@
17
18  .PHONY:clean
19  clean:
20          @rm -rf $(OBJECT) output/
```

分析：

- 行 1：将版本号赋值给变量 VERSION。
- 行 13：生成可执行文件的后缀添加版本号。

编译结果：

```
$ tree
.
├── inc
│   ├── add.h
│   └── sub.h
├── Makefile
├── output
│   └── 100ask_1.0.0
└── src
    ├── add.c
    ├── add.o
    ├── main.c
    ├── main.o
    ├── sub.c
    └── sub.o
```

# 第7章 时钟体系

时钟信号是数字时序电路的“脉搏”，电路每接收到一个周期的时钟信号，就做一个相应的动作。因此，在允许的范围内，时钟信号的快慢直接决定着电路性能的好坏。在片上系统（SOC）中，不同的模块通常需要工作在不同的时钟频率。为了满足这些需求，芯片将时钟源信号进行稳定、倍频、分频、分发以及屏蔽（gate）等操作，产生不同频率的时钟信号。这些时钟信号和它们的管理电路构成了芯片的时钟体系，驱动着各种各样的功能模块协同工作。

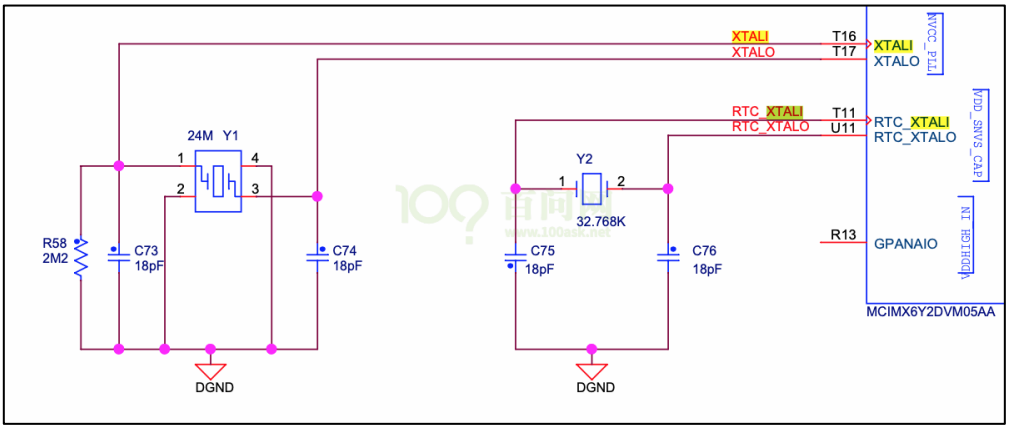
原理图：网盘开发板配套资料“05_Hardware（原理图)/Base_board/100ask_imx6ull_v1.1.pdf”。

参考资料：网盘开发板配套资料“06_Datasheet（数据手册）/Core_board/CPU/IMX6ULLRM.pdf”中《Chapter 18: Clock Controller Module (CCM)》。

## 7.1 IMX6ULL 时钟体系介绍

### 7.1.1 晶体振荡电路

时钟信号不是凭空产生的，芯片首先需有一个频率较低的源时钟信号。imx6ull 包含两个偏压放大器（biased amplifier），当外部连接合适的晶振和负载电容时，能够分别产生 24MHZ 和 32KHZ 的时钟信号。下面是开发板的时钟引脚连接原理图：




当 imx6ull 的时钟引脚 XTALI 和 XTALO 连上合适的晶振和电容时，模块 XTALOSC24M 会产生 24MHZ 的时钟信号。注意，输出频率到达 24MHZ 并不意味着模块 XTALOSC24M 已经稳定工作，仍然需要等待一段时间。一旦时钟信号稳定，可以通过设置寄存器来降低它的工作电流；但是注意，在关闭模块 XTALOSC24M 电源之前，相应的值应当被恢复，否则恢复供电时模块不能正常启动。

另外，imx6ull 还有一个内置的 24MHZ RC 振荡器，它以 RTC 时钟（32KHZ）做基准产生时钟信号。尽管能源消耗显著低于模块 XTALOSC24M，但是它的精确度存在较大误差，实际应用中应当避免使用它。

当 imx6ull 的时钟引脚 RTC_XTALI 和 RTC_XTALO 连接 32KHZ 或 32.768KHZ 的晶振时，用模块 RTC_XTAL 产生 32KHZ 的 RTC 时钟信号。除此之外，imx6ull 还包含一个内部的 32KHZ 振荡器，当时钟系统探测到 RTC 振荡器丢失时钟信号时，会自动切换到内部的 32KHZ 振荡器。但是由于该内部振荡器精度不如模块 RTC_XTAL，不能作为替代品长时间使用。

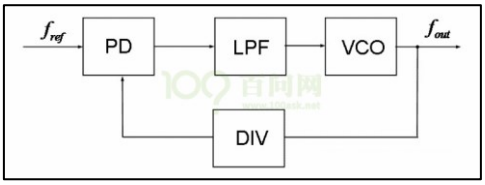
RTC 时钟信号主要用来记录时间，而 XTALOSC24M 的输出时钟信号为芯片的时钟体系提供基础的源时钟信号，是本篇讨论的重点。

除此之外，芯片本身也可以直接接收时钟信号作为源时钟信号，这需要芯片外部提供一个稳定的时钟信号源。这种方式较不常用，本文不再进行描述。

### 7.1.2 锁相环电路

XTALOSC24M 的时钟信号只有 24MHZ，远远不能满足实际需求，在芯片中需要进行稳定和倍频操作，这主要是由锁相环电路完成的。

锁相环（PLL）由鉴相器 PD、低通滤波器 LPF 和压控振荡器 VCO 组成。鉴相器（PD）用来鉴别两个输入时钟信号之间的相位差，并输出误差电压，经过低通滤波（LPF）后，形成压控振荡器（VCO）的控制。压控振荡器的输出经过分频（DIV）后反馈到鉴相器与基准信号进行比较，最终，VCO 的输出就会稳定下来。下图是锁相环工作原理示意图：




imx6ull 包含 7 个锁相环电路，它们的输入时钟信号称为源时钟信号，可通过寄存器选择，通常为 XTALOSC24M 产生的 24MHZ 时钟信号。它们的输出经过进一步选择和分频，形成不同的根时钟信号，分发到各个模块使用。这些锁相环电路及它们的分频器输出如下图所示：

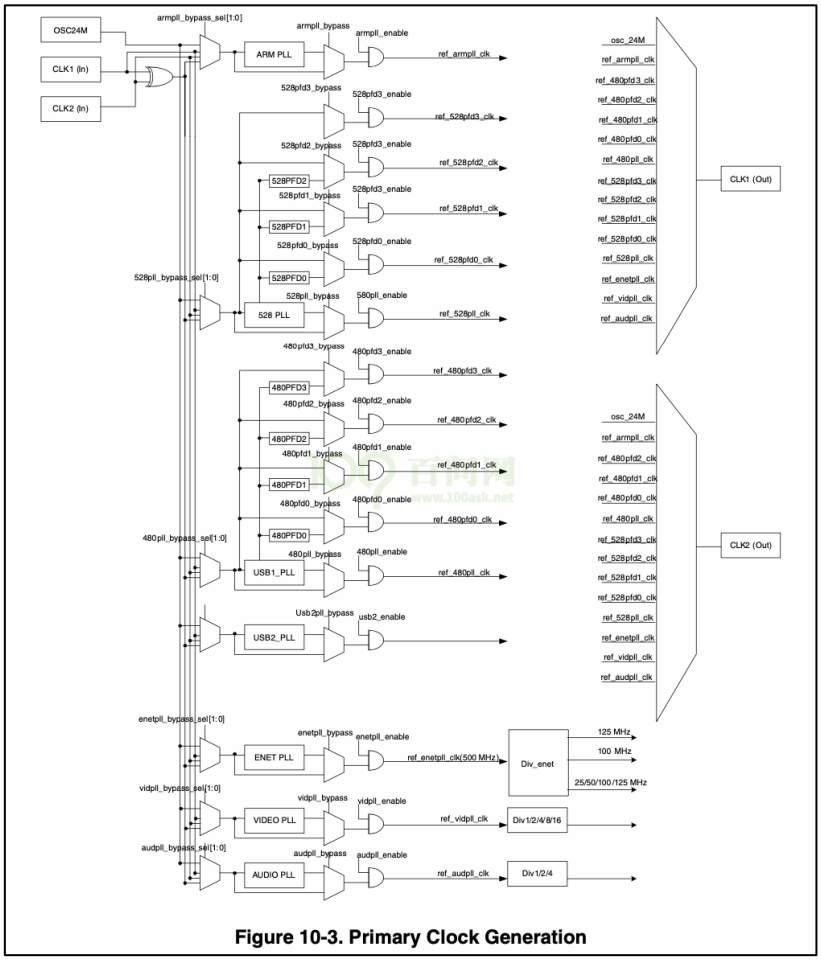



**Figure 10-3. Primary Clock Generation**

下面分别介绍这些 PLL 的功能。

**1 PLL1：**

被称为 ARM_PLL，用来驱动 ARM 核心工作。它能够倍频达到 1.3GHZ，注意这个频率超过了芯片能够工作的最大频率 1GHZ。

**2 PLL2：**

被称为 SYS_PLL 或者 528_PLL。它的倍频参数固定在 x22，在使用 XTALOSC24M 产生的 24MHZ 时钟作为参考时钟时，产生 528MHZ 的输出。除了这个主输出，SYS_PLL 还包含四个分频器，主输出和分频器的输出可用来作为根时钟。SYS_PLL 的这些输出时钟信号的频率并不需要是确定或者精确的值，可在运行时动态进行改变。通常，它们用来驱动芯片内部的系统总线，内部处理逻辑，DDR 接口以及 NAND/NOR 等等。

**3 PLL3：**

被称为 USB1_PLL，用来驱动第一个 USB 物理层实体 USBPHY1。它的倍频参数固定在 x20，在使用 24MHZ 参考时钟时产生 480MHZ 的输出。除了主输出之外，USB1_PLL 同样包含四个分频器，它们的输出用来作为需要固定频率的根时钟，比如 UART 和其它的串行接口，音频接口等。

**4 PLL4：**

被称为 AUDIO_PLL，能够进行倍频和分频操作，产生低抖动、高精度标准音频时钟信号。AUIDIO_PLL 的输出频率范围从 650MHZ 到 1300MHZ，时钟的频率分辨率要好于 1HZ。该输出时钟信号主要用来驱动串行音频接口或者作为外部音频解码器的参考时钟。另外，AUDIO_PLL 的分频器，可对 VCO 的输出时钟信号进行/1、/2 或/4 分频。

**5 PLL5：**

被称为 VIDEO_PLL。它同样具有倍频和分频功能，能够产生低抖动、高精度标准视频时钟信号。VIDEO_PLL 的输出频率范围从 650MHZ 到 1300MHZ，时钟的频率分辨率要好于 1HZ。该输出时钟主要作为显示和视频接口的时钟信号。另外，VIDEO_PLL 的分频器，可对 VCO 的输出时钟信号进行/1、/2、/4 或/8 分频。

**6 PLL6：**

被称为 ENET_PLL。它的倍频参数固定为 x20+(5/6)，在使用 24MHZ 参考时钟时产生 500MHZ 的输出。它主要用来生成：（1）50MHZ 或 25MHZ 时钟，用于外部以太网接口；（2）125MHZ 时钟，用于精简的千兆以太网接口；（3）100MHZ 时钟，用于通用功能。

**7 PLL7：**

被称为 USB2_PLL，专门用于驱动第二个 USB 物理层实体 USBPHY2。它的倍频参数固定为 x20，输出 480MHZ 的时钟信号。

上述锁相环电路都有自己专门的控制和状态寄存器，它们可独立配置为以下 3 个模式中的一种：

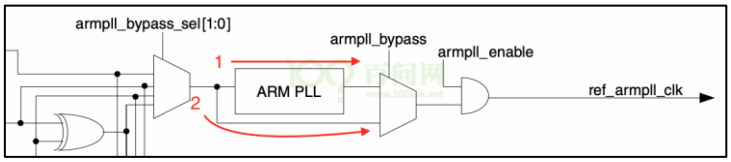
① Bypass 模式：PLL 输入的参考时钟直接传递到输出，由 BYPASS 位控制；

② 输出禁止模式：无论 bypass 时钟还是 PLL 生成的时钟均被禁止，无输出时钟信号，由 ENABLE 位控制；

③ 断电模式：PLL 中大部分电路断电，无输出时钟信号，由 POWERDOWN 位控制。

以 ARM_PLL 为例，单独截取出来说明，见下图。PLL 正常工作时，时钟信号通过路径 1 传输作为信号 ref_armpll_clk 输出；处于 Bypass 模式时，源时钟

信号不经过 PLL 放大，由路径 2 直接输出；处于输出禁止模式时，时钟信号在 armpll_enable 处被屏蔽，ref_armpll_clk 无输出；处于断电模式时，ARM_PLL 大部分电路断电，电路不工作，ref_armpll_clk 无输出信号。



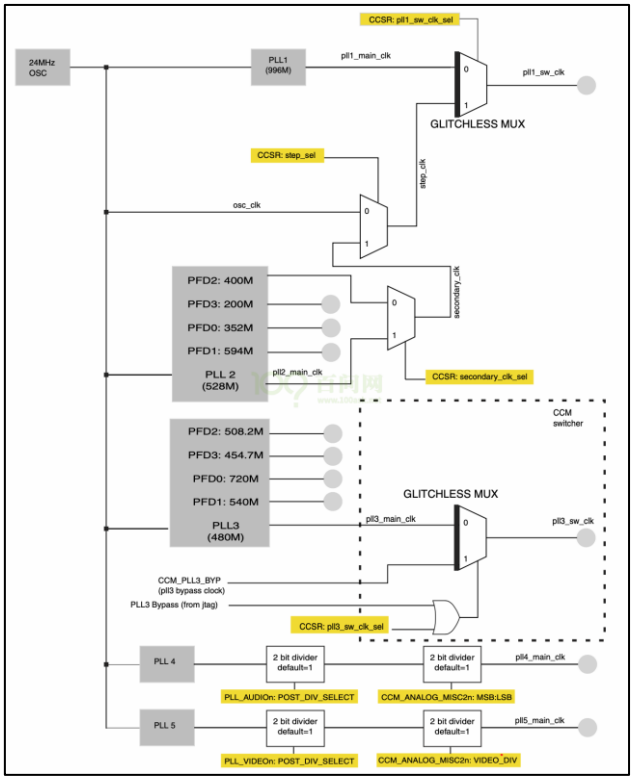

其中，两个锁相环电路 SYS_PLL 和 USB1_PLL，分别带有四个分相器（PFD）对其产生的 PLL 输出信号进行分频（每个分相器可独立设置分频参数），用来产生额外的频率输出。由于分相器完全由数字器件组成且不包含反馈回路，我们只需要改变逻辑组合就能改变分频参数的值，不影响锁相环电路的锁定状态，因此分相器能够比锁相环更快的改变输出频率。除此之外，分相器的值还能在运行时改变，不需要在改变前后关闭和开启时钟的输出。

注意，对于那些包含分相器的锁相环电路，每个分相器有自己独立的时钟屏蔽控制位。当相连的锁相环电路加电启动或者重新锁定时，分相器自动进入屏蔽状态，需要手动对每个 PFD 进行一次屏蔽和开启操作。

### 7.1.3 根时钟信号电路

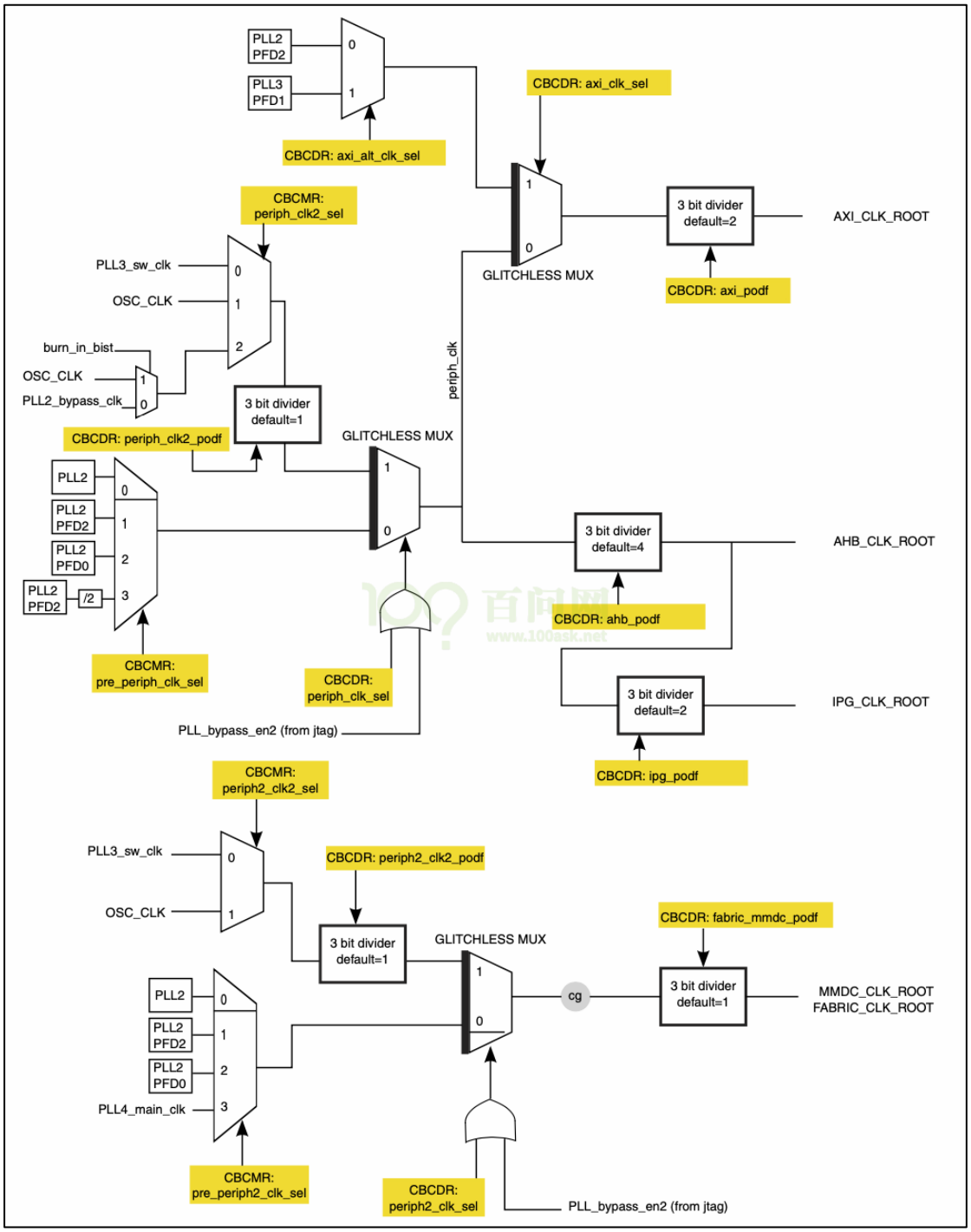
如前所述，锁相环电路的输出时钟信号并不能直接供其它模块使用。在 imx6ull 中，它们的输出时钟信号、PFD 时钟信号以及对应的 bypass 时钟信号经过选择、分频后形成根时钟信号，才会分发到各个模块。这部分电路又细分为两部分，前半部分称为时钟切换电路（switcher），后半部分称为根时钟生成电路（root generator）。

时钟切换电路主要对 PLL1 和 PLL3 的输出进行选择，被选中的信号形成 pll1_sw_clk 和 pll3_sw_clk 信号。另外，它还对 PLL4 和 PLL5 进行额外的分频操作，形成 pll4_main_clk 和 pll5_main_clk 信号。如下图所示：



后续电路不直接使用上述 PLL 的输出，而是使用 switcher 形成的这些输出信号。例如，如果我们想改变CPU的工作频率，可以先修改CCSR[pll1_sw_clk_sel]将 pll1_sw_clk 切换到 step_clk，然后修改 PLL1 的参数，等待其输出时钟信号稳定到新的频率上，再切换回 PLL1 的输出信号 pll1_main_clk。因为使用了无抖动的多路选择器（glitchless multiplexer），在切换过程中 CPU 仍正常运行，我们将在第一个编程示例中演示上述过程。

根时钟生成电路对前面的这些信号和 PLL2 的输出信号进一步筛选和分频形成根时钟信号，它们直接或者间接驱动芯片中的模块来实现各自功能。这里仅以总线相关的根时钟信号进行说明，如下图所示：




其它的根时钟信号和模块对这些根时钟信号的使用参见 CCM 和模块各自的寄存

器设置，这里不再展开叙述。本章第二个编程示例计算锁相环电路输出时钟和这些总线的根时钟的频率并打印出来，有兴趣的同学也可以参照示例代码和 imx6ull 手册计算其它的时钟信号的频率。

## 7.2 寄存器介绍

imx6ull 时钟相关的寄存器主要分布在 CCM 和 ANALOG_DIG 这两个模块，它们均连接在 AIPS-1 总线上，它们的地址范围如下所示：

**Table 2-2. AIPS-1 memory map (continued)**

| Start Address | End Address | Region | NIC Port | Size |
|---|---|---|---|---|
| 020D_8000 | 020D_BFFF | | SRC | 16 KB |
| 020D_4000 | 020D_7FFF | | EPIT2 | 16 KB |
| 020D_0000 | 020D_3FFF | | EPIT1 | 16 KB |
| 020C_C000 | 020C_FFFF | | SNVS_HP | 16 KB |
| 020C_8000 | 020C_8FFF | | ANALOG_DIG | 16 KB |
| 020C_4000 | 020C_7FFF | | CCM | 16 KB |
| 020C_0000 | 020C_3FFF | | WDOG2 | 16 KB |
| 020B_C000 | 020B_FFFF | | WDOG1 | 16 KB |
| 020B_8000 | 020B_BFFF | | KPP | 16 KB |

ANALOG_DIG 模块主要负责晶体振荡电路和锁相环电路的相关设置。它的寄存器分为两类：

① CCM_ANALOG_PLL_xxx 寄存器：

设置对应 PLL 的参数和工作状态。

② CCM_ANALOG_MISCx （x = 0-2）寄存器：

进行其它一些杂项的设置或状态显示，包括晶体振荡电路的控制参数。这些寄存器数量较多，这里不一一列出。

另外需要注意，ANALOG_DIG 与电源管理模块（PMU）共用这些 CCM_ANALOG_MISCx 寄存器，它们在 PMU 中被称为 PMU_MISCx （x = 0-2）。由于晶体振荡电路在系统启动时已初始化完毕，输出频率固定的时钟信号，在芯片运行期间通常不需要修改设置，这里不再进行说明。

而 CCM 模块控制根时钟信号的产生和分发，大部分寄存器用来对 PLL 及 PFD 产生的时钟信号进行分发和分频控制，如下表所示：

**CCM memory map**

| Absolute address (hex) | Register name | Width (in bits) | Access | Reset value | Section/ page |
|---|---|---|---|---|---|
| 20C_4000 | CCM Control Register (CCM_CCR) | 32 | R/W | 0401_167Fh | 18.6.1/660 |
| 20C_4004 | CCM Control Divider Register (CCM_CCDR) | 32 | R/W | | 18.6.2/661 |
| 20C_4008 | CCM Status Register (CCM_CSR) | 32 | R | 0000_0010h | 18.6.3/663 |
| 20C_400C | CCM Clock Switcher Register (CCM_CCSR) | 32 | R/W | 0000_0100h | 18.6.4/664 |
| 20C_4010 | CCM Arm Clock Root Register (CCM_CACRR) | 32 | R/W | 0000_0000h | 18.6.5/665 |
| 20C_4014 | CCM Bus Clock Divider Register (CCM_CBCDR) | 32 | R/W | 0001_8D00h | 18.6.6/666 |
| 20C_4018 | CCM Bus Clock Multiplexer Register (CCM_CBCMR) | 32 | R/W | 2486_0324h | 18.6.7/669 |
| 20C_401C | CCM Serial Clock Multiplexer Register 1 (CCM_CSCMR1) | 32 | R/W | 0490_0080h | 18.6.8/670 |
| 20C_4020 | CCM Serial Clock Multiplexer Register 2 (CCM_CSCMR2) | 32 | R/W | 0319_2C06h | 18.6.9/673 |
| 20C_4024 | CCM Serial Clock Divider Register 1 (CCM_CSCDR1) | 32 | R/W | 0049_0B00h | 18.6.10/ 674 |
| 20C_4028 | CCM SAI1 Clock Divider Register (CCM_CS1CDR) | 32 | R/W | 0EC1_02C1h | 18.6.11/ 676 |
| 20C_402C | CCM SAI2 Clock Divider Register (CCM_CS2CDR) | 32 | R/W | 0003_36C1h | 18.6.12/ 678 |

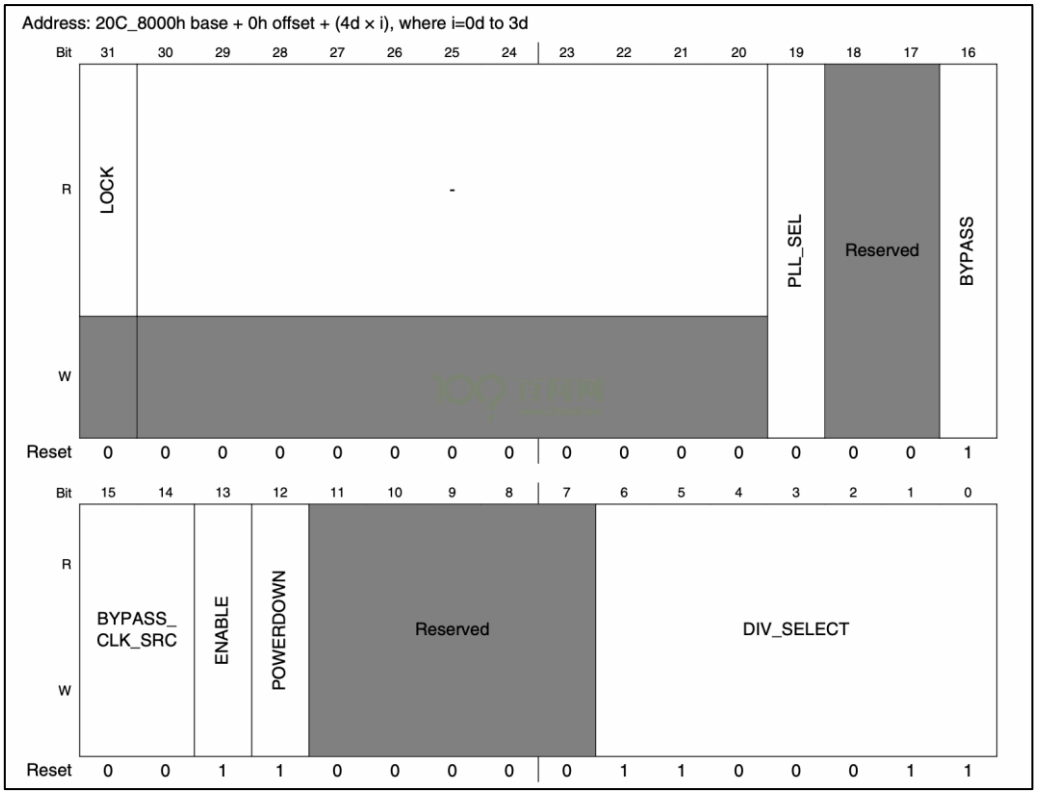
以及下面的表，注意红框部分的寄存器 CCM_CCGRx（x = 0-6），它们用来控制各个时钟信号在不同功耗模式下是否被屏蔽：

**CCM memory map (continued)**

| Absolute address (hex) | Register name | Width (in bits) | Access | Reset value | Section/ page |
|---|---|---|---|---|---|
| 20C_4030 | CCM D1 Clock Divider Register (CCM_CDCDR) | 32 | R/W | 33F7_1F92h | 18.6.13/ 680 |
| 20C_4034 | CCM HSC Clock Divider Register (CCM_CHSCCDR) | 32 | R/W | 0002_9148h | 18.6.14/ 681 |
| 20C_4038 | CCM Serial Clock Divider Register 2 (CCM_CSCDR2) | 32 | R/W | 0002_9B48h | 18.6.15/ 682 |
| 20C_403C | CCM Serial Clock Divider Register 3 (CCM_CSCDR3) | 32 | R/W | 0001_4841h | 18.6.16/ 684 |
| 20C_4048 | CCM Divider Handshake In-Process Register (CCM_CDHIPR) | 32 | R | 0000_0000h | 18.6.17/ 685 |
| 20C_4054 | CCM Low Power Control Register (CCM_CLPCR) | 32 | R/W | 0000_0079h | 18.6.18/ 688 |
| 20C_4058 | CCM Interrupt Status Register (CCM_CISR) | 32 | w1c | 0000_0000h | 18.6.19/ 690 |
| 20C_405C | CCM Interrupt Mask Register (CCM_CIMR) | 32 | R/W | FFFF_FFFFh | 18.6.20/ 693 |
| 20C_4060 | CCM Clock Output Source Register (CCM_CCOSR) | 32 | R/W | 000A_0001h | 18.6.21/ 695 |
| 20C_4064 | CCM General Purpose Register (CCM_CGPR) | 32 | R/W | 0000_FE62h | 18.6.22/ 697 |
| 20C_4068 | CCM Clock Gating Register 0 (CCM_CCGR0) | 32 | R/W | FFFF_FFFFh | 18.6.23/ 698 |
| 20C_406C | CCM Clock Gating Register 1 (CCM_CCGR1) | 32 | R/W | FFFF_FFFFh | 18.6.24/ 700 |
| 20C_4070 | CCM Clock Gating Register 2 (CCM_CCGR2) | 32 | R/W | FC3F_FFFFh | 18.6.25/ 702 |
| 20C_4074 | CCM Clock Gating Register 3 (CCM_CCGR3) | 32 | R/W | FFFF_FFFFh | 18.6.26/ 703 |
| 20C_4078 | CCM Clock Gating Register 4 (CCM_CCGR4) | 32 | R/W | FFFF_FFFFh | 18.6.27/ 704 |
| 20C_407C | CCM Clock Gating Register 5 (CCM_CCGR5) | 32 | R/W | FFFF_FFFFh | 18.6.28/ 706 |
| 20C_4080 | CCM Clock Gating Register 6 (CCM_CCGR6) | 32 | R/W | FFFF_FFFFh | 18.6.29/ 707 |
| 20C_4088 | CCM Module Enable Overide Register (CCM_CMEOR) | 32 | R/W | FFFF_FFFFh | 18.6.30/ 709 |

## 7.2.1 锁相环电路寄存器

如前所述，imx6ull 共包含 7 个锁相环电路，除 ENET_PLL 之外，其它的锁相环电路控制和状态寄存器的结构都很类似。以 ARM_PLL 为例，它的寄存器结构如下所示：

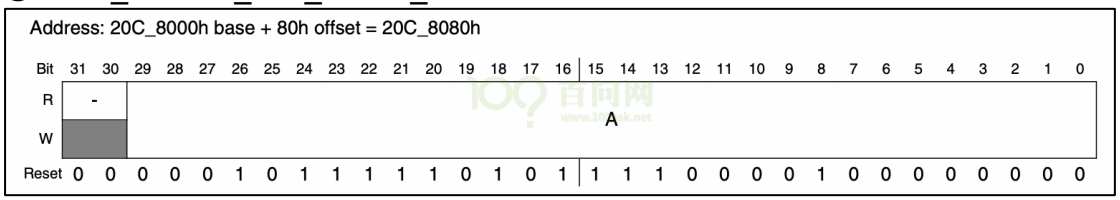



其中，控制位 BYPASS、ENABLE 和 POWERDOWN 用来控制 PLL 的工作模式（bypass 模式、输出禁止模式和断电模式）；BYPASS_CLK_SRC 选择输入时钟源，DIV_SELECT 设置频率放大倍数。正常工作时，需要设置 BYPASS 和 POWRDOWN 为 0，ENABLE 为 1。当 LOCK 值为 1 时，锁相环电路输出稳定的时钟信号。

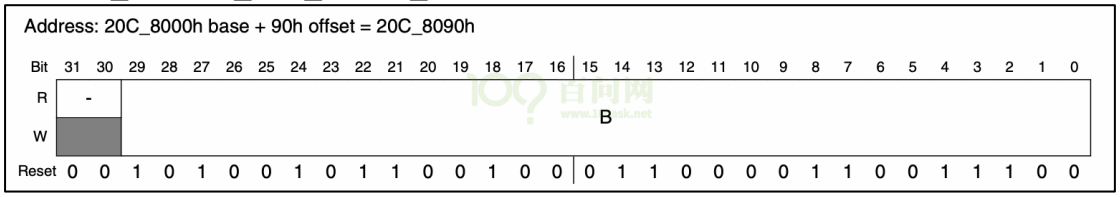
稳定工作时，ARM_PLL 的输出频率为 Fref * DIV_SELECT/2，其它 PLL 的设置方法和输出频率计算公式与 ARM_PLL 类似，但略有差别。例如，锁相环电路 USB1_PLL、USB2_PLL 和 SYS_PLL 虽然都有自己的 DIV_SELECT，它们应当被设为固定的值。

需要特别说明的是，锁相环电路 AUDIO_PLL 和 VIDEO_PLL 还增加了额外的分频参数 NUM 和 DENOM。以 AUDIO_PLL 为例，它的相应寄存器如下所示：

① CCM_ANALOG_PLL_AUDIO_NUM

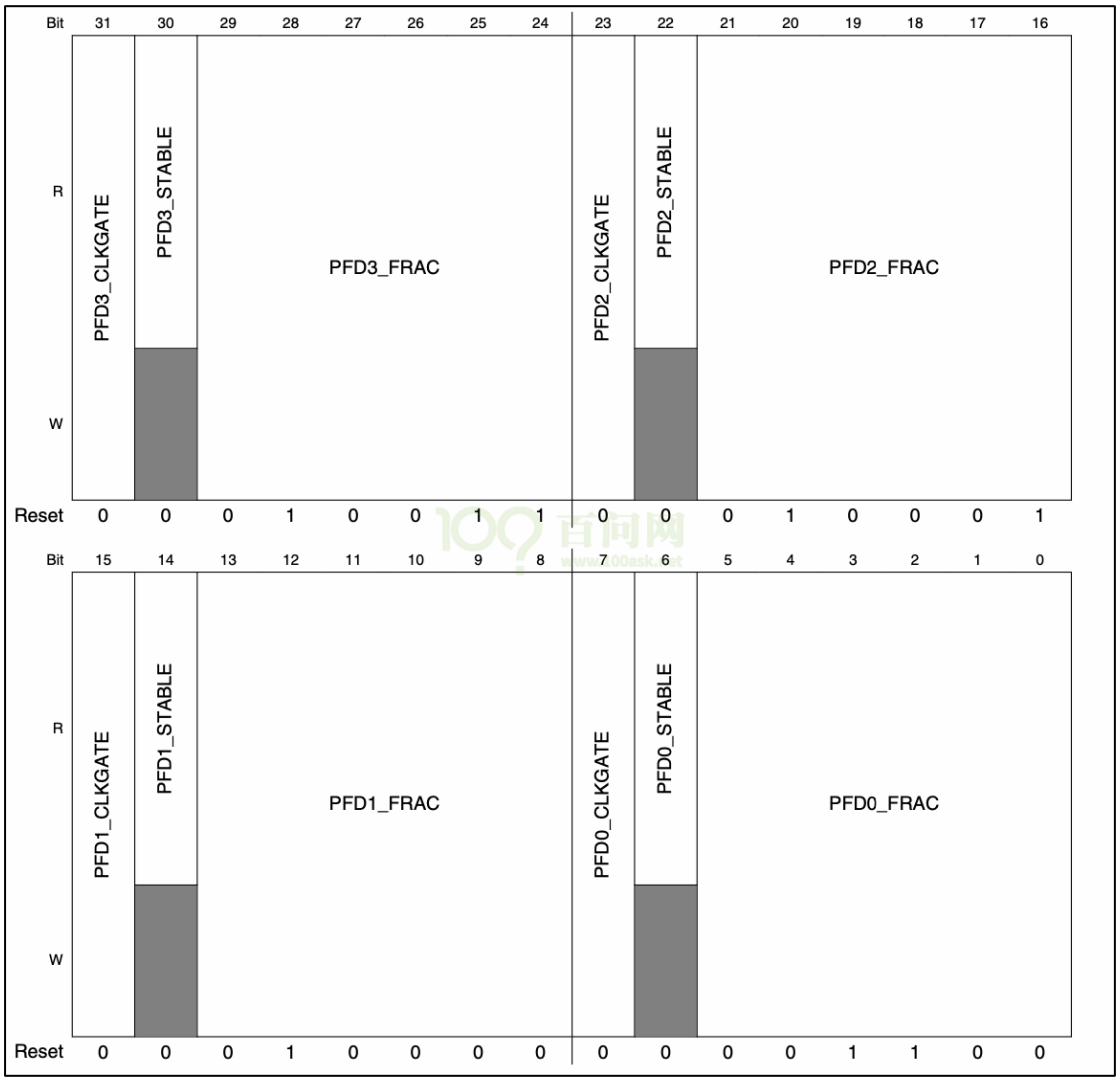



② CCM_ANALOG_PLL_AUDIO_DENOM



它们的输出频率为 Fref * (DIV_SELECT + NUM/DENOM)。除此之外，AUIDIO_PLL 和 VIDEO_PLL 还可以在时钟切换电路（switcher）中设置额外的分频参数为/1、/2、/4、/8 或/16，这些值分布在寄存器 CCM_ANALOG_PLL_AUDIO、CCM_ANALOG_PLL_VIDEO 和 CCM_ANALOG_MISC2 中。

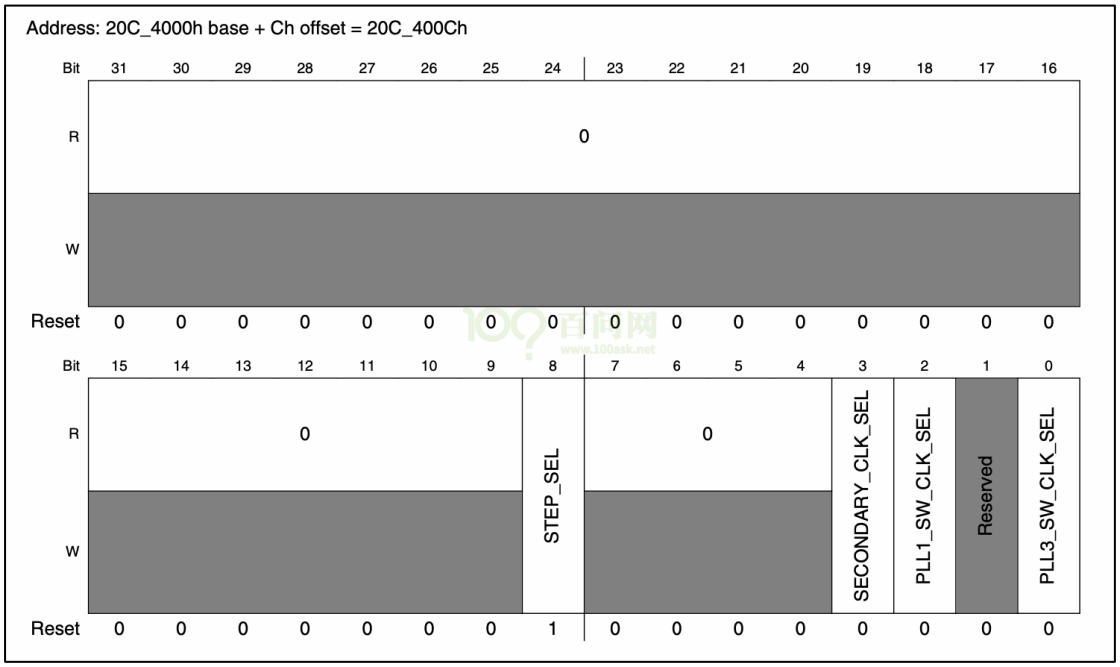
除此之外，SYS_PLL 和 USB1_PLL 还各自配有四个分相器，它们分别对 SYS_PLL 和 USB1_PLL 的输出时钟信号进行分频，分频参数分别在 CCM_ANALOG_PFD_480n 和 CCM_ANALOG_PFD_528n 中设置。这两个寄存器的结构完全一样，如下图所示：



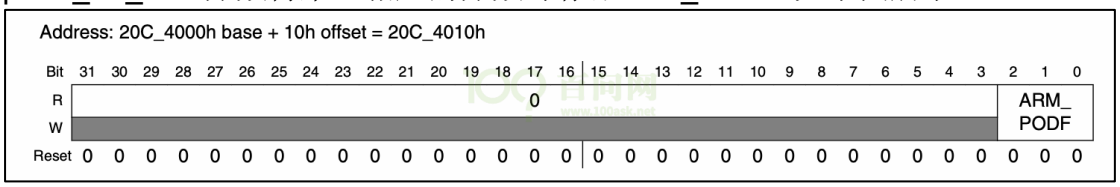

每个 PFD 的输出频率为 Fvco*18/PFD_FRAC，其中 Fvco 是相应 PLL 的输出频率，而 PFD_FRAC 的数值取值范围为 12 到 35。

## 7.2.2 根时钟控制寄存器

上述锁相环电路以及它们的 bypass 时钟信号、PFD 输出信号，经过时钟切换电路（switcher）和根时钟生成电路（root generator）处理后形成各种根时钟信号。其中，时钟切换电路寄存器 CCM_CCSR 选择时钟信号 pll1_sw_clk 和 pll3_sw_clk 的来源，如下图所示：

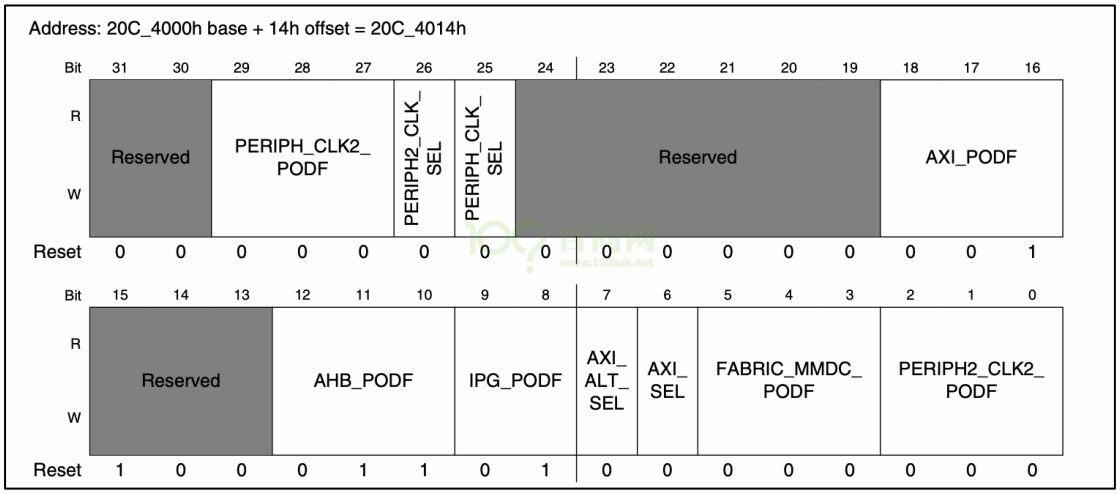



根时钟生成电路的寄存器也包含在 CCM 模块中，比如 ARM 根时钟信号由 pll1_sw_clk 分频得来，相应的分频寄存器 CCM_CACRR 如下图所示：




ARM 的时钟信号频率为 pll1_sw_clk/(ARM_PODF + 1)。

而之前提到的总线根时钟信号由寄存器 CCM_CBCDR 进行选择和分频，如下图所示：




其中各个字段的作用可参见根时钟信号电路一节中的原理图，具体设置步骤可参见后面的编程示例。其它根时钟的寄存器与其类似，详细的控制方式要参见

imx6ull 手册中 CCM 模块的寄存器描述，这里不再一一列举。

### 7.2.3 模块时钟屏蔽寄存器

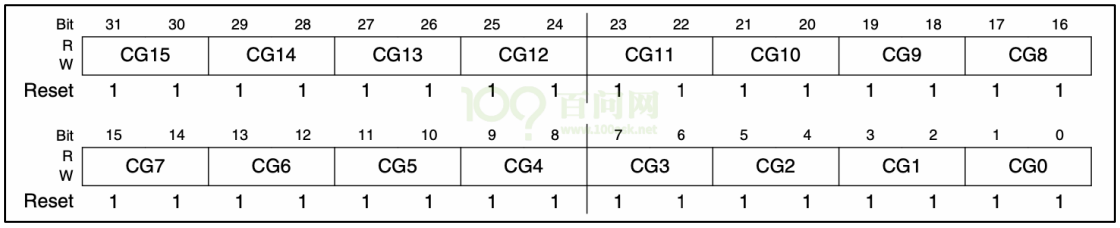
为了降低功耗，imx6ull 可以工作在以下三种模式：

① RUN 模式：CCM_CLPCR[LPM]的值为 0，CPU 正常工作，各个模块的时钟信号可以在寄存器 CCGRx 中开启和关闭。

② WAIT 模式：CCM_CLPCR[LPM]的值为 1，当 CPU 执行 WFI 指令时，开始进入 WAIT 模式。在此模式下，CPU 时钟被关闭，依据寄存器 CCGRx 中的设置，相应模块的时钟信号也会被关闭。

③ STOP 模式：CCM_CLPCR[LPM]的值为 2。STOP 模式同样重复上述 WAIT 模式的操作，并且禁用所有的 PLL。如果设置了 CCM_CLPCR[SBYOS]，该模式还将激活 cosc_pwrdown 信号，关闭晶体振荡电路的电源。

对于每个时钟信号，imx6ull 提供了在不同工作模式下是否屏蔽这些时钟信号的控制方法，这些控制位集中放在寄存器 CCGRx（x = 0-6）中，每个 CCGRx 寄存器结构如下图所示：




每两位为一个单位，控制一个时钟信号。比如，在寄存器 CCGR0 中，CG15 控制时钟信号 gpio2_clocks，CG14 控制时钟信号 uart2_clock 等等。这两位值的含义如下所示：

| CGx （x=0-15） | 时钟信号活动状态描述 |
|---|---|
| 00 | 时钟信号在三个模式中均被屏蔽。 |
| 01 | 时钟信号仅在 RUN 模式开启，在其它模式中被屏蔽。 |
| 10 | 保留 |
| 11 | 时钟信号在 RUN 和 WAIT 模式开启，在 STOP 模式中被屏蔽。 |

当用户某个模块驱动时，应当根据该模块是否需要在低功耗模式下工作来设置寄存器 CCGRx 中相应的值，以达到降低功耗的目的。另外，在正常工作模式中，用户也可以在模块空闲时将 CCGRx 中相应的值设为 0，动态调节模块的功耗。

## 7.3 编程示例

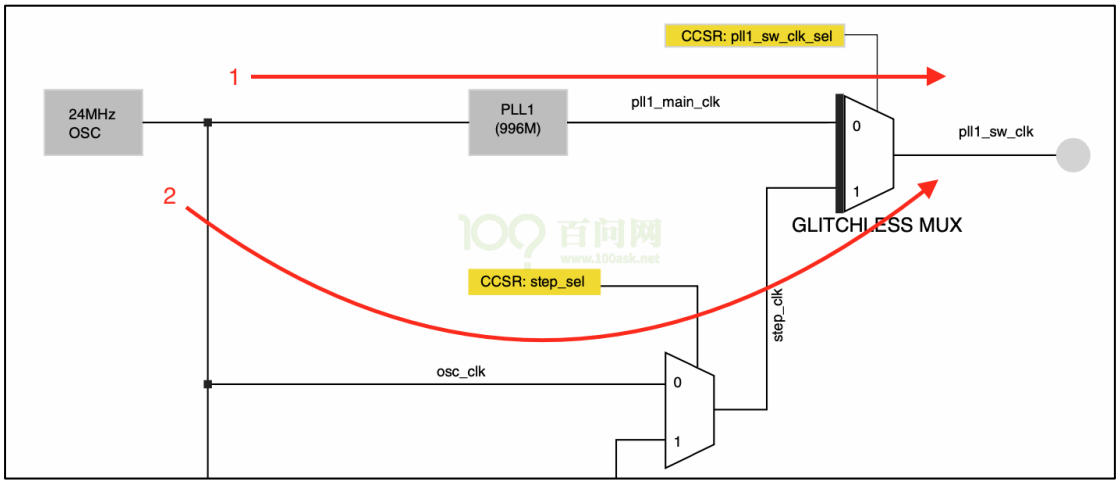
### 7.3.1 改变 CPU 工作频率

本示例动态改变 CPU 的工作频率，先将 CPU 设置工作在一个较低频率（81MHZ），然后切换至较高的工作频率（648MHZ）——即使是 528M 版本的 IMX6ULL 也可以运行，超频一点点没关系。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/7_时钟体系/fastcpu”目录下。

为验证 CPU 频率的变化，本例使用忙等待的延时方式，控制 LED 灯闪烁--随着 CPU 频率的提升，延时时间变短，LED 灯闪烁频率变快。

正常工作时，CPU 使用锁相环电路 ARM_PLL 的输出信号作为时钟源。在改变 CPU 频率之前，我们首先切换到其它时钟信号（示例中选择晶体振荡电路 XTALOSC24M 的输出），修改 ARM_PLL 设置并稳定在新的频率之后，再切换回 ARM_PLL 的输出时钟信号。这里我们再次引用时钟切换电路（switcher）的部分原理图，如下所示：




➢ **设置 PLL1_SW_CLK 的时钟路径。**

改变 CPU 频率前，pll1_sw_clk 时钟路径如图中路径 1 所示。我们首先将其切换至路径 2，待 ARM_PLL（PLL1）稳定输出后，再切换回路径 1。

怎么控制时钟路径？修改上图中用黄色标出来的寄存器值即可。对应的控制函数为 set_pll1_sw_clk(int sel_pll1)，当参数 sel_pll1 值为 0 时，选择路径 2；当参数 sel_pll1 值非 0 时，选择路径 1。

函数 set_pll1_sw_clk 定义在文件 switcher.c 中，内容如下所示：

```
4 extern struct ccm_regs *ccm;
5
/************************************************************************
 * 函数名称： set_pll1_sw_clk
 * 功能描述： 设置 PLL1_SW_CLK 的时钟路径
 * 输入参数： sel_pll1: 0-选择 XTALOSC24M 的输出，1-选择 PLL1 的输出
 * 输出参数： 无
 * 返 回 值： 无
 * 修改日期        版本号     修改人       修改内容
 * -----------------------------------------------
 * 2020/03/08     V1.0      今朝         创建
 ************************************************************************/
void sel_pll1_sw_clk(int sel_pll1)
{
    /* PLL1_SW_CLK_SEL: 0 表示 pll1_main_clk，1 表示 step_clk */
    if (sel_pll1)
        clr_bit(&ccm->ccsr, 2);     /* 选择 pll1_main_clk */
    else {
        clr_bit(&ccm->ccsr, 8);     /* step_clk 选择使用 OSC 的输出 */
        set_bit(&ccm->ccsr, 2);     /* 选择 step_clk */
    }
}
```

➢ **重新设置 ARM_PLL 的输出频率**

切换完 pll1_sw_clk 时钟路径之后，我们就可以重新设置 ARM_PLL 的输出频率，相应的设置函数 set_pll 设置指定 PLL 的倍频参数并等待其输出频率稳定。为了简化函数接口，AUDIO_PLL 和 VIDEO_PLL 的 NUM 和 DENOM 参数统一设置为 0xF，而且不支持 ENET_PLL 的设置，感兴趣的同学可以自己添加相关代码。该函数位于文件 pll.c 中，内容如下：

```
struct anadig_regs *anadig = (struct anadig_regs *)ANADIG_BASE_ADDR;

  static void wait_to_lock(u32 *pll_reg)
  {
      while (read32(pll_reg) & LOCK_MASK == 0);   /* 等待指定的 PLL 进入锁定状态 */
  }

  /************************************************************************
   * 函数名称： set_pll
   * 功能描述： 设置 PLL 的倍频参数并等待其进入锁定状态
   * 输入参数： pll: 指定 PLL 的标识，div: PLL 的倍频参数
   * 输出参数： 无
   * 返 回 值： 无
   * 修改日期        版本号     修改人       修改内容
   * -----------------------------------------------
   * 2020/03/08     V1.0      今朝        创建
   ************************************************************************/
  void set_pll(pll_e pll, u32 div)
  {
      switch (pll) {
          case ARM_PLL:
              if (div < 54 && div > 108) return;      /* ARM_PLL 的倍频参数的有效范围为
54 到 108 */
              write32(ENABLE_MASK | div, &anadig->analog_pll_arm);
              wait_to_lock(&anadig->analog_pll_arm);  /* 等待 ARM_PLL 锁定 */
              break;

          case USB1_PLL:      /* 设置 USB1_PLL 的倍频参数并等待锁定 */
              write32(ENABLE_MASK | (div&0x3), &anadig->analog_pll_usb1);
              wait_to_lock(&anadig->analog_pll_usb1);
              break;

          case USB2_PLL:      /* 设置 USB2_PLL 的倍频参数并等待锁定 */
              write32(ENABLE_MASK | (div&0x3), &anadig->analog_pll_usb2);
              wait_to_lock(&anadig->analog_pll_usb2);
              break;

          case SYS_PLL:       /* 设置 SYS_PLL 的倍频参数并等待锁定 */
              write32(ENABLE_MASK | (div&0x1), &anadig->analog_pll_sys);
              wait_to_lock(&anadig->analog_pll_sys);
              break;

          case AUDIO_PLL:
              if (div < 27 && div > 54) return;       /* AUDIO_PLL 的倍频参数的有效范围
为 27 到 54 */

              /* 简便起见，AUDIO_PLL 的分频参数 NUM 和 DENOM 都设置为 0xF */
              write32(0xF, &anadig->analog_pll_video_num);
              write32(0xF, &anadig->analog_pll_video_denom);

              write32(ENABLE_MASK | div, &anadig->analog_pll_video);
```

```
            wait_to_lock(&anadig->analog_pll_video);/* 等待 AUDIO_PLL 锁定 */
            break;

        case VIDEO_PLL:
            if (div < 27 && div > 54) return;       /* VIDEO_PLL 的倍频参数的有效范围
为 27 到 54 */

            /* 简便起见，VIDEO_PLL 的分频参数 NUM 和 DENOM 都设置为 0xF */
            write32(0xF, &anadig->analog_pll_audio_num);
            write32(0xF, &anadig->analog_pll_audio_denom);

            write32(ENABLE_MASK | div, &anadig->analog_pll_video);
            wait_to_lock(&anadig->analog_pll_video);/* 等待 VIDEO_PLL 锁定 */
            break;

        case ENET_PLL:
            /* ENET_PLL 寄存器设置方式与其它 PLL 差别较大，为了简化函数接口，这里不支持对
它的设置 */
            break;
     }
  }
```

- **设置 ARM_CLK_ROOT 的分频参数**

时钟信号 pll1_sw_clk 在成为 arm_clk_root 送往 CPU 之前，还要在根时钟生成电路（root generator）经过一次分频操作，其分频参数的设置函数为 setup_arm_podf，位于文件 clkroot.c 中，内容如下：

```
extern struct ccm_regs *ccm;
 /*************************************************************************
 * 函数名称： setup_arm_podf
 * 功能描述： 设置 ARM_CLK_ROOT 的分频参数
 * 输入参数： 取值范围为 1-8，ARM_CLK_ROOT = PLL1_SW_CLK / PODF
 * 输出参数： 无
 * 返 回 值： 无
 * 修改日期        版本号     修改人       修改内容
 * -----------------------------------------------
 * 2020/03/08     V1.0      今朝        创建
 *************************************************************************/
void setup_arm_podf(u32 podf)
{
    if (podf < 1 || podf > 8) return;   /* ARM_PODF 分频范围是 1 到 8 */
    write32(podf-1, &ccm->cacrr);
}
```

- **修改 led 闪烁函数**

最后，为了方便比较 led 的闪烁频率，我们对 led 的接口函数稍作修改，增加 led_toggle 函数，led 的状态改变一次（从亮到灭或从灭到亮），这里不再展示其代码。

在 main 函数中，我们首先初始化并点亮 led 灯，设置 CPU 频率到 81MHZ（ARM_PLL 输出为 648MHZ，分频参数为 8），亮灯和灭灯各 5 次；然后设置 CPU 频率为 648MHZ（ARM_PLL 输出为 1296MHZ，分频参数为 2），之后无限闪灯。我们可通过肉眼观测到 led 的闪烁频率明显变快。main.c 文件内容如下所示：

```
#include "regs.h"
#include "pll.h"
#include "clkroot.h"
```

```
struct ccm_regs *ccm = (struct ccm_regs *)CCM_BASE_ADDR;

#define LOOPS 1000000
static void busy_wait(void)
{
    for(u32 i = 0; i < LOOPS; i++); /* 忙等待进行延时 */
}

/* LED 灯接口函数 */
extern void led_init(void);
extern void led_toggle(void);
void led_on(void);
 /* PLL1 时钟路径及其 PODF 分频参数设置 */
extern void setup_arm_podf(u32 podf);
extern void sel_pll1_sw_clk(int sel_pll1);

void main(void)
{
    /* 首先初始化并点亮 LED 灯 */
    int blinks = 0;
    led_init();
    led_on();

    sel_pll1_sw_clk(0);     /* 将 ARM_ROOT 时钟切换至 OSC */
    setup_arm_podf(8);      /* ARM_ROOT 的分频参数设置为 8 */
    set_pll(ARM_PLL, 54);   /* 设置 ARM_PLL: 24*54/2 = 648MHZ, ARM_ROOT: 81MHZ */
    sel_pll1_sw_clk(1);     /* 将 ARM_ROOT 切换回 ARM_PLL，此时 CPU 工作频率为 81MHZ */

    /* 循环点亮/熄灭 LED 共 10 次，观察 LED 闪烁频率 */
    for (blinks = 10; blinks > 0; blinks--)
    {
        busy_wait();
        led_toggle();
    }

    sel_pll1_sw_clk(0);     /* 将 ARM_ROOT 时钟切换至 OSC */
    setup_arm_podf(2);      /* ARM_ROOT 的分频参数设置为 2 */
    set_pll(ARM_PLL, 108);  /* 设置 ARM_PLL: 24*108/2 = 1296MHZ, ARM_ROOT: 648MHZ */
    sel_pll1_sw_clk(1);     /* 将 ARM_ROOT 切换回 ARM_PLL，此时 CPU 工作频率为 648MHZ */

    /* 无限循环点亮/熄灭 LED，观察此时 LED 闪烁频率明显变快 */
    while(1)
    {
        busy_wait();
        led_toggle();
    }
}

/* 本程序的除法运算使用了 GCC 提供的函数，需要提供 raise 函数以正常编译 */
void raise(void)
{
}
```

最后说明一下，由于本程序用到了 GCC 的除法操作例程，需要添加一个空的 raise 函数，以通过编译。运行成功后可以观察到开发板用户绿色 led 灯闪烁频率由慢变快。

## 7.3.2 打印时钟信号的频率值

上个示例中我们可以通过 led 的闪烁频率观察到 CPU 的频率确实变快了。为了得到imx6ull中时钟的确切值,我们在本例程中将它们通过串口打印出来,这里我们使用了 uart 模块的打印功能,其代码参见后面的uart串口编程章节。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/7_时钟体系/showclocks”目录下。

本例程仅打印各个 PLL（不包括 ENET_PLL）及其 PFD 的输出频率，以及前面介绍的总线根时钟频率，这些频率值的计算需要分为三步进行。

- **获取 PLL 的输出频率**

首先，我们需要确定 PLL 和 PFD 的输出频率，在文件 pll.c 中添加以下代码：

```
/*****************************************************************
  * 函数名称： get_pll
  * 功能描述： 获取 PLL 的输出频率
  * 输入参数： pll: 指定 PLL 的标识
  * 输出参数： 无
  * 返 回 值： PLL 的输出频率
  * 修改日期        版本号     修改人       修改内容
  * -----------------------------------------------
  * 2020/03/08     V1.0      今朝        创建
  *****************************************************************/
 u32 get_pll(pll_e pll)
 {
   u32 div, post_div, pll_num, pll_denom;

   switch (pll) {
        case ARM_PLL:
           div = read32(&anadig->analog_pll_arm);
           if (div & BYPASS_MASK)  /* 判断 ARM_PLL 是否处于 Bypass 模式 */
               return CKIH;
           else {
               div &= 0x7F;        /* 获取 ARM_PLL 的倍频参数 */
               return (CKIH * div) >> 1;   /* ARM_PLL 的输出频率只有倍频后的一半 */
           }

       case USB1_PLL:
           div = read32(&anadig->analog_pll_usb1);
           if (div & BYPASS_MASK)  /* 判断 USB1_PLL 是否处于 Bypass 模式 */
               return CKIH;
           else {
               div = div&0x1 ? 22 : 20;    /* USB1_PLL 只有两种倍频模式，1 表示 x22，0
表示 x20 */
               return CKIH * div;
           }

     case USB2_PLL:
      div = read32(&anadig->analog_pll_usb2);
           if (div & BYPASS_MASK)  /* 判断 USB2_PLL 是否处于 Bypass 模式 */
               return CKIH;
           else {
               div = div&0x1 ? 22 : 20;    /* USB2_PLL 只有两种倍频模式，1 表示 x22，0
表示 x20 */
               return CKIH * div;
```

```
            }

        case SYS_PLL:
      div = read32(&anadig->analog_pll_sys);
            if (div & BYPASS_MASK)  /* 判断 SYS_PLL 是否处于 Bypass 模式 */
                return CKIH;
            else {
                div = div&0x1 ? 22 : 20;    /* SYS_PLL 只有两种倍频模式，1 表示 x22，0 表示 x20 */
                return CKIH * div;
            }

        case AUDIO_PLL:
      div = read32(&anadig->analog_pll_audio);
            if (!(div & ENABLE_MASK))   /* 判断 AUDIO_PLL 是否处于禁止输出模式 */
                return 0;

            if (div & BYPASS_MASK)      /* 判断 AUDIO_PLL 是否处于 Bypass 模式 */
                return CKIH;
            else {
                post_div = (div >> 19 ) & 0x3;
                if (post_div == 3)      /* reserved value */
                    return 0;
                /* AUDIO_PLL 的分频参数：0 表示除以 4，1 表示除以 2，2 表示除以 1 */
          post_div = 1 << (2 - post_div);

          pll_num = read32(&anadig->analog_pll_audio_num);
          pll_denom = read32(&anadig->analog_pll_audio_denom);

          return CKIH * (div + pll_num / pll_denom) / post_div;
            }

     case VIDEO_PLL:
      div = read32(&anadig->analog_pll_video);
            if (!(div & ENABLE_MASK))   /* 判断 VIDEO_PLL 是否处于禁止输出模式 */
                return 0;

            if (div & BYPASS_MASK)      /* 判断 VIDEO_PLL 是否处于禁止输出模式 */
                return CKIH;
            else {
                post_div = (div & 0x3) >> 19;
                if (post_div == 3)  /* reserved value */
                    return 0;
                /* VIDEO_PLL 的分频参数：0 表示除以 4，1 表示除以 2，2 表示除以 1 */
          post_div = 1 << (2 - post_div);

          pll_num = read32(&anadig->analog_pll_video_num);
          pll_denom = read32(&anadig->analog_pll_video_denom);

          return CKIH * (div + pll_num / pll_denom) / post_div;
            }

     default:
      return 0;
  }
  /* NOTREACHED */
 }
```

```
static void set_pfd(u32 *reg, pfd_e pfd, int gate, u32 frac)
{
    u32 value = read32(reg);    /* 读取指定 PLL 的 PFD 寄存器 */
    value &= ~PFD_MASK(pfd);
    if (gate) value |= PFD_GATE_MASK(pfd);  /* 设置是否屏蔽该 PFD 的输出 */
    value |= (frac<<PFD_SHIFT(pfd)) & PFD_FRAC_MASK(pfd);   /* 设置该 PFD 的分频参数
*/
    write32(value, reg);

    while(read32(reg) & PFD_STABLE_MASK(pfd));
}

/************************************************************************
 * 函数名称： set_pll_pfd
 * 功能描述： 设置 SYS_PLL 或 USB1_PLL 的 PFD 状态和分频参数
 * 输入参数： pll: 指定 PLL 的标识，pfd: 指定 PFD 的编号，gate: 是否屏蔽该 PFD 的输出，
frac: PFD 的分频参数
 * 输出参数： 无
 * 返 回 值： 无
 * 修改日期        版本号     修改人       修改内容
 * -----------------------------------------------
 * 2020/03/08     V1.0      今朝        创建
 ************************************************************************/
void set_pll_pfd(pll_e pll, pfd_e pfd, int gate, u32 frac)
{
    u32 *reg;
    if (pll = SYS_PLL)
        reg = &anadig->analog_pfd_528;
    else if (pll = USB1_PLL)
        reg = &anadig->analog_pfd_480;
    else
        /* 只有 SYS_PLL 和 USB1_PLL 支持 PFD 输出 */
        return ;

    set_pfd(reg, pfd, gate, frac);
}

/************************************************************************
 * 函数名称： get_pll_pfd
 * 功能描述： 获取 SYS_PLL 或 USB1_PLL 的 PFD 输出频率
 * 输入参数： pll: 指定 PLL 的标识，pfd: 指定 PFD 的编号
 * 输出参数： 无
 * 返 回 值： 无
 * 修改日期        版本号     修改人       修改内容
 * -----------------------------------------------
 * 2020/03/08     V1.0      今朝        创建
 ************************************************************************/
u32 get_pll_pfd(pll_e pll, pfd_e pfd)
{
 u32 div;
 u64 freq;

 switch (pll) {
       case SYS_PLL:
      div = read32(&anadig->analog_pfd_528);
      freq = (u64)get_pll(SYS_PLL);
      break;
       case USB1_PLL:
```

```
      div = read32(&anadig->analog_pfd_480);
      freq = (u64)get_pll(USB1_PLL);
      break;
     default:
           /* 只有 SYS_PLL 和 USB1_PLL 支持 PFD 输出 */
      return 0;
  }
   /* PFD 输出频率为 fPLL x 18 / N（N 是 PFD 的分频参数） */
  return (freq * 18) / PFD_FRAC_VALUE(div, pfd);
 }
```

- **获取 PLL1_SW_CLK 的时钟频率**

其次，这些时钟信号要经过时钟切换电路（switcher）的筛选，在文件 switcher.c 中增加接口获取筛选后的时钟信号频率：

```
/************************************************************************
  * 函数名称： get_pll1_sw_clk
  * 功能描述： 获取 PLL1_SW_CLK 的时钟频率
  * 输入参数： 无
  * 输出参数： 无
  * 返 回 值： PLL1_SW_CLK 的时钟频率
  * 修改日期        版本号     修改人       修改内容
  * -----------------------------------------------
  * 2020/03/08     V1.0      今朝        创建
  ************************************************************************/
 u32 get_pll1_sw_clk(void)
 {
     u32 reg = read32(&ccm->ccsr);

     if (reg & (1u<<2)) {        /* PLL1_SW_CLK_SEL：0 表示 pll1_main_clk，1 表示
step_clk */
         if (reg & (1u<<8)) {    /* STEP_SEL：1 表示 secondary_clk，0 表示 OSC */
             if (reg & (1u<<3))  /* SECONDARY_CLK_SEL：1 表示 PLL2，0 表示 PLL2 PFD2 */
                 return get_pll(SYS_PLL);
             else
                 return get_pll_pfd(SYS_PLL, PFD2);
         } else
             return CKIH;        /* OSC 输出 */
     } else
         return get_pll(ARM_PLL);
 }

 /************************************************************************
  * 函数名称： get_pll3_sw_clk
  * 功能描述： 获取 PLL3_SW_CLK 的时钟频率
  * 输入参数： 无
  * 输出参数： 无
  * 返 回 值： PLL3_SW_CLK 的时钟频率
  * 修改日期        版本号     修改人       修改内容
  * -----------------------------------------------
  * 2020/03/08     V1.0      今朝        创建
  ************************************************************************/
 u32 get_pll3_sw_clk(void)
 {
     u32 reg = read32(&ccm->ccsr);
     if (reg & 1)    /* PLL3_SW_CLK_SEL: 1 表示 pll3，0 表示 pll3_bypass（即 OSC 输出）
*/
         return get_pll(USB1_PLL);
```

```
    else
        return CKIH;            /* OSC 输出 */
}

/***************************************************************
 * 函数名称： get_pll4_main_clk
 * 功能描述： 获取 PLL4_MAIN_CLK 的时钟频率
 * 输入参数： 无
 * 输出参数： 无
 * 返 回 值： PLL4_MAIN_CLK 的时钟频率
 * 修改日期        版本号     修改人       修改内容
 * -----------------------------------------------
 * 2020/03/08     V1.0      今朝        创建
 ***************************************************************/
extern struct anadig_regs *anadig;

u32 get_pll4_main_clk(void)
{
    u32 reg, audio_div;

    reg = read32(&anadig->pmu_misc2);
    /* AUDIO_DIV_MSB(23): AUDIO_DIV_LSB(15)
     * 00：除以 1
     * 01：除以 2
     * 10：除以 1
     * 11：除以 4
     */
    audio_div = reg & (1u<<15) ? (reg & (1u<<23) ? 4 : 2) : 1;

    return get_pll(AUDIO_PLL) / audio_div;
}

/***************************************************************
 * 函数名称： get_pll5_main_clk
 * 功能描述： 获取 PLL5_MAIN_CLK 的时钟频率
 * 输入参数： 无
 * 输出参数： 无
 * 返 回 值： PLL5_MAIN_CLK 的时钟频率
 * 修改日期        版本号     修改人       修改内容
 * -----------------------------------------------
 * 2020/03/08     V1.0      今朝        创建
 ***************************************************************/
u32 get_pll5_main_clk(void)
{
    u32 reg, video_div;

    reg = read32(&anadig->pmu_misc2);
    /* AUDIO_DIV_MSB(31): AUDIO_DIV_LSB(30)
     * 00：除以 1
     * 01：除以 2
     * 10：除以 1
     * 11：除以 4
     */
    video_div = reg & (1u<<30) ? (reg & (1u<<31) ? 4 : 2) : 1;

    return get_pll(VIDEO_PLL) / video_div;
```

➢ 获取 PLL1_SW_CLK 的时钟频率

最后，根时钟生成电路（root generator）对上述信号进一步选择和分频，得到根时钟信号，在文件 clkroot.c 中添加以下代码：

```
/*****************************************************************
 * 函数名称： get_arm_clk_root
 * 功能描述： 获取 ARM_CLK_ROOT 的频率
 * 输入参数： 无
 * 输出参数： 无
 * 返 回 值： ARM_CLK_ROOT 的频率
 * 修改日期        版本号     修改人       修改内容
 * -----------------------------------------------
 * 2020/03/08     V1.0      今朝        创建
 *****************************************************************/
u32 get_arm_clk_root(void)
{
    u32 reg, freq;

    reg = read32(&ccm->cacrr);
    reg = (reg & 0x7) + 1;             /* 获取 ARM_PODF 的分频范围 */
    freq = get_pll(ARM_PLL);

    return freq / reg;
}

/*****************************************************************
 * 函数名称： get_periph_clk
 * 功能描述： 获取 PERIPH_CLK 的频率
 * 输入参数： 无
 * 输出参数： 无
 * 返 回 值： PERIPH_CLK 的频率
 * 修改日期        版本号     修改人       修改内容
 * -----------------------------------------------
 * 2020/03/08     V1.0      今朝        创建
 *****************************************************************/
static u32 get_periph_clk(void)
{
    u32 reg, per_clk2_podf = 0, freq = 0;

    reg = read32(&ccm->cbcdr);

    /* PERIPH_CLK_SEL 选择 periph_clk 的时钟源：1 表示 periph_clk2，0 表示
pre_periph_clk */
    if (reg & (1u << 25)) {         /* 选择 periph_clk2 */
        per_clk2_podf = (reg >> 27) & 0x7;    /* 获取 PERIPH_CLK2_PODF */
        reg = read32(&ccm->cbcmr);
        reg = (reg >> 12) & 0x3;    /* PERIPH_CLK2_SEL */

        /* PERIPH_CLK2_SEL：0 表示 pll3_sw_clk，1 表示 osc_clk，2 表示 pll2_bypass_clk
（即 osc_clk） */
        switch (reg) {
        case 0:
            freq = get_pll(USB1_PLL);
            break;
        case 1:
        case 2:
            freq = CKIH;
            break;
```

```
        default:
            break;
        }

        freq /= (per_clk2_podf + 1);
    } else {                        /* 选择 pre_periph_clk */
        reg = read32(&ccm->cbcmr);
        reg = (reg >> 18) & 0x3;    /* PRE_PERIPH_CLK_SEL */

        /* PRE_PERIPH_CLK_SEL：0 表示 PLL2 输出，1 表示 PLL2 PFD2 输出，2 表示 PLL2 PFD0
输出，3 表示 PLL2 PFD2 输出频率的一半 */
        switch (reg) {
        case 0:
            freq = get_pll(SYS_PLL);
            break;
        case 1:
            freq = get_pll_pfd(SYS_PLL, PFD2);
            break;
        case 2:
            freq = get_pll_pfd(SYS_PLL, PFD0);
            break;
        case 3:     /* static / 2 divider */
            freq = get_pll_pfd(SYS_PLL, PFD2) / 2;
            break;
        default:
            break;
        }
    }

    return freq;
}

/**************************************************************************
 * 函数名称： get_ahb_clk_root
 * 功能描述： 获取 AHB_CLK_ROOT 的频率
 * 输入参数： 无
 * 输出参数： 无
 * 返 回 值： AHB_CLK_ROOT 的频率
 * 修改日期        版本号     修改人       修改内容
 * -----------------------------------------------
 * 2020/03/08     V1.0      今朝        创建
 **************************************************************************/
u32 get_ahb_clk_root(void)
{
    u32 reg, ahb_podf;

    reg = read32(&ccm->cbcdr);
    ahb_podf = (reg >> 10) & 0x7;   /* 获取 AHB_PODF 的分频设置 */

    return get_periph_clk() / (ahb_podf + 1);
}

/**************************************************************************
 * 函数名称： get_ipg_clk_root
 * 功能描述： 获取 IPG_CLK_ROOT 的频率
 * 输入参数： 无
 * 输出参数： 无
 * 返 回 值： IPG_CLK_ROOT 的频率
```

```
 * 修改日期        版本号     修改人       修改内容
 * ---------------------------------------------
 * 2020/03/08     V1.0      今朝        创建
 *****************************************************************/
u32 get_ipg_clk_root(void)
{
    u32 reg, ipg_podf;

    reg = read32(&ccm->cbcdr);
    ipg_podf = (reg >> 8) & 0x3;    /* 获取 IPG_PODF 的分频设置 */

    return get_ahb_clk_root() / (ipg_podf + 1);
}

/*****************************************************************
 * 函数名称： get_axi_clk_root
 * 功能描述： 获取 AXI_CLK_ROOT 的频率
 * 输入参数： 无
 * 输出参数： 无
 * 返 回 值： AXI_CLK_ROOT 的频率
 * 修改日期        版本号     修改人       修改内容
 * ---------------------------------------------
 * 2020/03/08     V1.0      今朝        创建
 *****************************************************************/
u32 get_axi_clk_root(void)
{
    u32 root_freq, axi_podf;
    u32 reg =  read32(&ccm->cbcdr);

    axi_podf = (reg >> 16) & 0x7;   /* 获取 AXI_PODF 的分频设置 */

    if (reg & (1u << 6)) {          /* AXI_SEL: 1 表示 axi_alt_clk，0 表示 periph_clk
*/
        if (reg & (1u << 7))        /* AXI_ALT_SEL：1 表示 PLL3  PFD1 的输出，0 表示 PLL2
PFD2 的输出 */
            root_freq = get_pll_pfd(USB1_PLL, PFD1);
        else
            root_freq = get_pll_pfd(SYS_PLL, PFD2);
    } else
        root_freq = get_periph_clk();   /* periph_clk */

    return  root_freq / (axi_podf + 1);
}

/*****************************************************************
 * 函数名称： get_fabric_mmdc_clk_root
 * 功能描述： 获取 FABRIC_MMDC_CLK_ROOT 的频率
 * 输入参数： 无
 * 输出参数： 无
 * 返 回 值： FABRIC_MMDC_CLK_ROOT 的频率
 * 修改日期        版本号     修改人       修改内容
 * ---------------------------------------------
 * 2020/03/08     V1.0      今朝        创建
 *****************************************************************/
u32 get_fabric_mmdc_clk_root(void)
{
    u32 cbcmr = read32(&ccm->cbcmr);
    u32 cbcdr = read32(&ccm->cbcdr);
```

```
    u32 freq, podf, per2_clk2_podf;

    podf = (cbcdr >> 3) & 0x7;  /* 获取 FABRIC_MMDC_PODF 的分频设置 */

    if (cbcdr & (1u << 26)) {   /* PERIPH2_CLK_SEL: 1 选择 periph2_clk2，0 选择
pre_periph2_clk */
        per2_clk2_podf = cbcdr & 0x7;   /* 获取 PERIPH2_CLK2_PODF 的分频设置 */
        if (cbcmr & (1u << 20)) /* PERIPH2_CLK2_SEL：1 选择 osc_clk，0 选择
pll3_sw_clk */
            freq = CKIH;
        else
            freq = get_pll(USB1_PLL);

        freq /= (per2_clk2_podf + 1);
    } else {                    /* pre_periph2_clk */
        extern u32 get_pll4_main_clk(void);
        /* PRE_PERIPH2_CLK_SEL：0 表示 PLL2 输出，1 表示 PLL2 PFD2 输出，2 表示 PLL2 PFD0
输出，3 表示_main_clk */
        switch ((cbcmr >> 21) & 0x3) {
            case 0:
                freq = get_pll(SYS_PLL);
                break;
            case 1:
                freq = get_pll_pfd(SYS_PLL, PFD2);
                break;
            case 2:
                freq = get_pll_pfd(SYS_PLL, PFD0);
                break;
            case 3:
                freq = get_pll4_main_clk();
                break;
        }
    }

    return freq / (podf + 1);
 }
```

- 函数打印时钟值

最终，所有的时钟通过函数 show_clocks 打印，该函数定义在 main.c 中：

```
/************************************************************************
 28  * 函数名称：  show_clocks
 29  * 功能描述：  打印 PLL 输出时钟频率和总线根时钟频率
 30  * 输入参数：  无
 31  * 输出参数：  无
 32  * 返 回 值：  无
 33  * 修改日期        版本号     修改人       修改内容
 34  * -----------------------------------------------
 35  * 2020/03/08     V1.0      今朝        创建
 36  ************************************************************************/
 37 void show_clocks(void)
 38 {
 39     u32 freq;
 40
 41     printf("Show IMX6ULL Clocks: \r\n");
 42     freq = get_pll(ARM_PLL);
 43     printf("ARM_PLL     %8d MHz\r\n", freq / 1000000);
```

```
44
45      freq = get_pll(SYS_PLL);
46      printf("SYS_PLL     %8d MHz\r\n", freq / 1000000);
47      freq = get_pll_pfd(SYS_PLL, PFD0);
48      printf("|-SYS_PLL_PFD0  %8d MHz\r\n", freq / 1000000);
49      freq = get_pll_pfd(SYS_PLL, PFD1);
50      printf("|-SYS_PLL_PFD1  %8d MHz\r\n", freq / 1000000);
51      freq = get_pll_pfd(SYS_PLL, PFD2);
52      printf("|-SYS_PLL_PFD2  %8d MHz\r\n", freq / 1000000);
53      freq = get_pll_pfd(SYS_PLL, PFD3);
54      printf("|-SYS_PLL_PFD3  %8d MHz\r\n", freq / 1000000);
55
56      freq = get_pll(USB1_PLL);
57      printf("USB1_PLL    %8d MHz\r\n", freq / 1000000);
58      freq = get_pll_pfd(USB1_PLL, PFD0);
59      printf("|-USB1_PLL_PFD0 %8d MHz\r\n", freq / 1000000);
60      freq = get_pll_pfd(USB1_PLL, PFD1);
61      printf("|-USB1_PLL_PFD1 %8d MHz\r\n", freq / 1000000);
62      freq = get_pll_pfd(USB1_PLL, PFD2);
63      printf("|-USB1_PLL_PFD2 %8d MHz\r\n", freq / 1000000);
64      freq = get_pll_pfd(USB1_PLL, PFD3);
65      printf("|-USB1_PLL_PFD3 %8d MHz\r\n", freq / 1000000);
66
67      freq = get_pll(USB2_PLL);
68      printf("USB2_PLL    %8d MHz\r\n", freq / 1000000);
69      freq = get_pll(AUDIO_PLL);
70      printf("AUDIO_PLL   %8d MHz\r\n", freq / 1000000);
71      freq = get_pll(VIDEO_PLL);
72      printf("VIDEO_PLL   %8d MHz\r\n", freq / 1000000);
73
74      printf("\r\n");
75      freq = get_arm_clk_root();
76      printf("ARM_CLK_ROOT    %8d KHZ\r\n", freq / 1000);
77      freq = get_ahb_clk_root();
78      printf("AHB_CLK_ROOT    %8d KHZ\r\n", freq / 1000);
79      freq = get_ipg_clk_root();
80      printf("IPG_CLK_ROOT    %8d KHZ\r\n", freq / 1000);
81      freq = get_axi_clk_root();
82      printf("AXI_CLK_ROOT    %8d KHZ\r\n", freq / 1000);
83      freq = get_fabric_mmdc_clk_root();
84      printf("FABRIC_MMDC_CLK_ROOT    %8d KHZ\r\n", freq / 1000);
85      printf("\r\n");
86  }
```

在第二次设置完时钟后，调用 show_clocks 函数打印。我们通过串口看到 CPU 最终的频率为 648000KHZ，即 648MHZ，与程序中设置的一致。运行成功后可以观察到串口打印所有的时钟频率信息。

### 7.3.3 补充说明

第二个示例中 CPU 频率增加到 8 倍，然而 led 的闪烁频率似乎并没有变为原来的 8 倍。这是因为延时函数 busy_wait 执行过程访问了内存，内存的速度限制了程序的性能。反汇编 showclock.elf 可以看到以下代码：

函数 busy_wait 在空循环过程中访问了三次内存，造成了性能瓶颈。解决方法之一是使用嵌入式汇编重新定义该函数，消除内存访问操作，代码如下：

```
static void busy_wait(void)
{
```

```
    __asm__ __volatile__ (
           "ldr r0, =3000000\n"
           "1:\n"
           "subs r0, r0, #1\n"
           "bne 1b\n"
           :::"r0");
}
```

修改后可发现，在 CPU 频率设置为 648MHZ 之后，led 灯闪烁频率明显比修改前快很多。另一个避免直接访问内存的方法是开启 D-Cache，有兴趣的同学可以尝试一下。

另外，为了更加直观的查看时钟频率，用户还可以将时钟信号通过 CCM_CLKO1 和 CCM_CLKO2 输出，使用示波器直接观察。相关的代码在文件 clkout.c 中，默认这些代码并不执行，有条件的同学可以开启这段代码进行实验。

# 第8章 UART 串口编程

## 8.1 UART 介绍

### 8.1.1 UART 串口简介

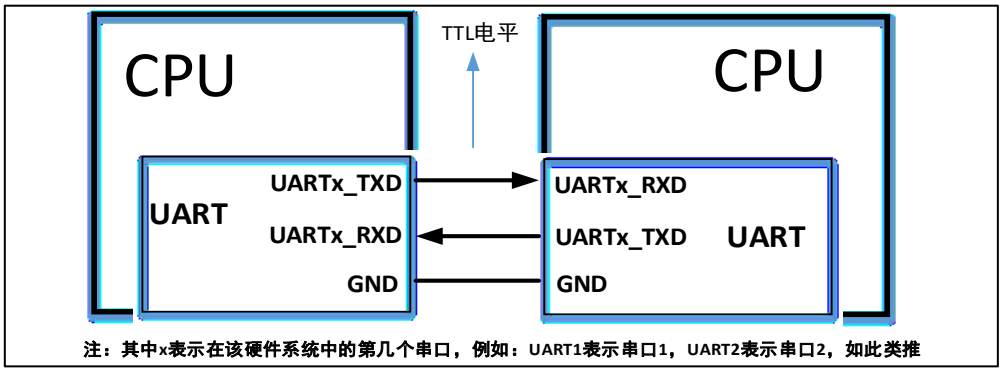
UART 全称是通用异步收发传输器（Universal Asynchronous Receiver/Transmitter)。串口顾名思义是数据串行接口，即数据的传输是一位接一位传输，属于一种串行的数据总线，属于异步通讯，同时支持全双工数据传输（全双工数据传输：允许发送数据和接收数据在同一时刻发生）。

除了 UART，另外还有一种叫 USART，全称是通用同步/异步串行接收/发送器(Universal Synchronous/Asynchronous Receiver/Transmitter)，USART 比 UART 多了同步通信功能，但是百分之 90 的工程应用中不会应用该同步功能，都是将 USART 当做 UART 使用，即采取异步串行通讯。一般开发板或者产品中都会将 UART 串口标为 serial 或 COM。

### 8.1.2 UART 硬件连接

**1 UART 串口最精简的连接是 TTL 电平三线连接**

- UARTx_TXD：用于发送数据，应连接到接收设备的 UARTx_RXD 引脚上；
- UARTx_RXD：用于接收数据，应连接到发送设备的 UARTx_TXD 引脚上；
- GND：为双方提供一个相同的参考电平。

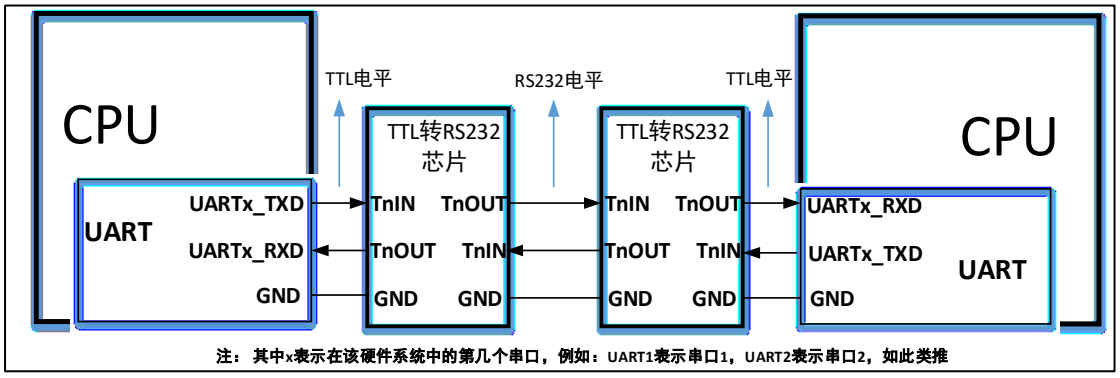


上图为 UART 串口 TTL 电平硬件连接，此时使用标准的 TTL 电平来表示数据，高电平表示 1，低电平表示 0，标准 TTL 输入高电平最小 2V，输出高电平最小 2.4V，典型值为 3.4V，输入低电平最大 0.8V，输出低电平最大 0.4V，典型值为 0.2V。直接采用 TTL 电平进行串行通讯，由于其抗干扰能力差，导致传输距离短，且容易出现数据不可靠的情况。

为提高抗干扰能力和传输距离，一般采用下面两种硬件连接方式。

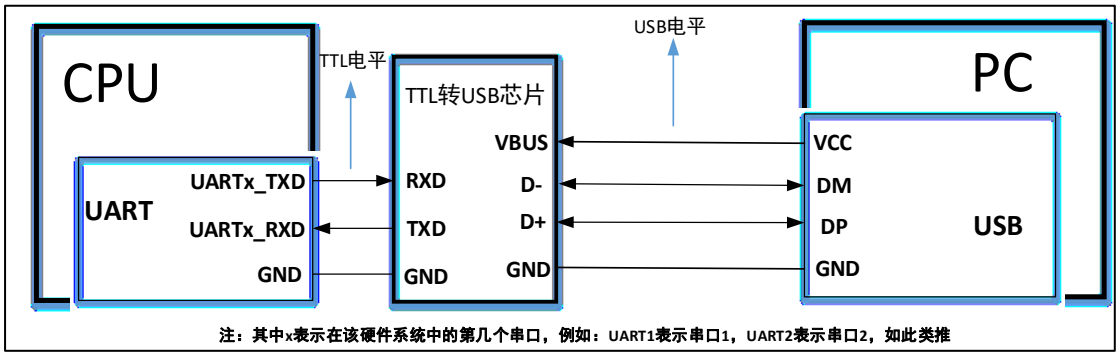
- **TTL 电平转 RS232 电平**

硬件连接如下图：




RS232 电平规定逻辑“1”的电平为-5V~-15 V，逻辑“0”的电平为+5 V～+15 V，选用该电气标准以提高抗干扰能力。常用的 TTL 转 RS232 芯片有：MAX232，SP3232 等。

**2 TTL 电平转 USB 电平**




将 TTL 电平转换为 USB 电平（D+与 D-一对差分信号采用 NRZI 编码实现通讯），提高抗干扰能力，常用的 TTL 转 USB 芯片有：PL2303，CH340， CP2102 等

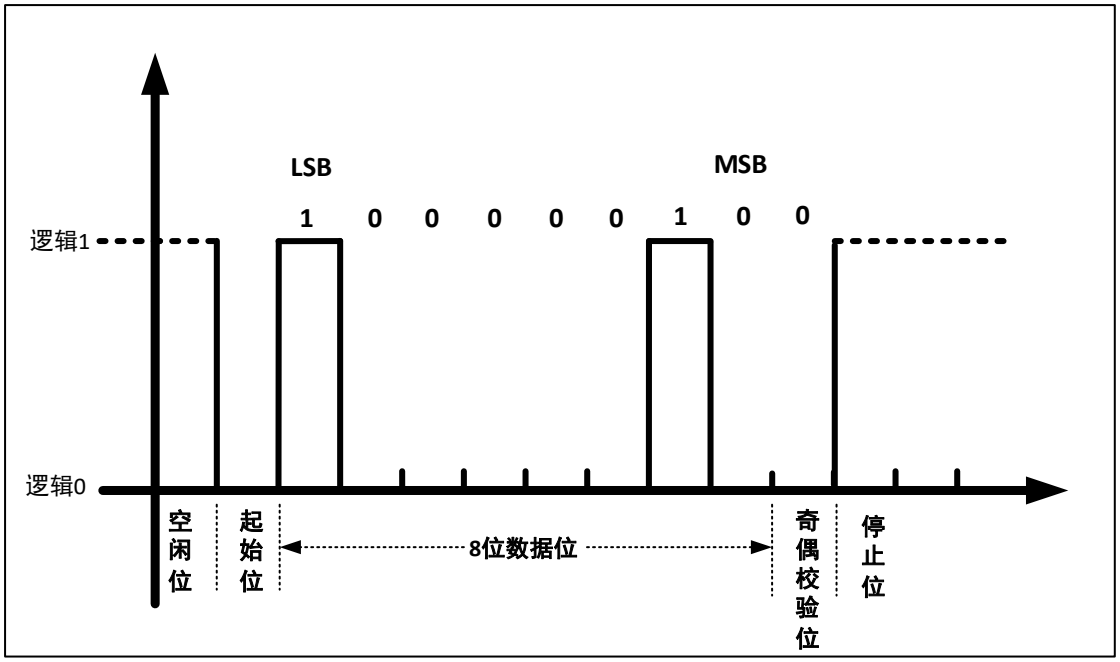
100ASK_IMX6ULL 采用的是上述方案中的“TTL 转 USB 方案”。

## 8.1.3 UART 通讯数据格式

UART 之间为何能够准确可靠的的发送和接收数据？

首先我们需要约定好传输速率（每一秒传输的数据位数，即波特率），一般选择 9600,19200,115200 等。

确定好传输速率后，我们还需要确定传输数据的格式，UART 串口通信的数据包以帧为单位，常用的帧结构为：1 位起始位+8 位数据位+1 位奇偶校验位（可选）+1 位停止位。举例说明，如下图：




上图为：1 位起始位+8 位数据位+1 位偶校验位+1 位停止位 的波形。根据查找 ASCII 码表得知’ A’字符的 ASCII 值为 41（十进制），将其转换成二进制应该为 0100 0001，小端传输，即低位（LSB）在前，高位（MSB）在后，和上图所示一致。

上图中各位的详细说明如下：

① 空闲位:平时没有数据时，数据线为高电平（逻辑 1）；

② 起始位:当需要发送数据时，UART 将改变 UARTx_TXD 的状态，变为低电平，即为上图中的起始位（逻辑 0）；

③ 数据位:可以有 5、6、7 或 8 位的数据，一般我们是按字节（8 位）传输数据，发送方一位一位的改变数据线上的状态（高电平或低电平）将它们发送出去，传输数据时先传最低位，最后传送最高位。

④ 字符’A’的 8 位二进制字符是 0100 0001，先发送最低位 bit 0，其值为 1；再发送 bit 1，其值为 0，如此继续；最后发送最高位 bit 7，其值为 0。

⑤ 奇偶校验位:如果使用了奇偶校验功能，有效数据位发送完毕后，还要发送 1 个校验位（奇偶校验位）。

⑥ 有两种校验方法：奇校验，偶校验--------数据位连同校验位中，“1”的数目等于奇数或偶数。奇偶校验只能检错，不能纠错的。而且只能检测 1 位误码，检测出有错后只能要求重发，没法纠正的。

⑦ 上图中使用的是偶较验，即 8 个数据位和 1 个校验位中，一共有偶数个“1”：2 个。

⑧ 停止位:停止位（逻辑 1）用来表示当前数据传输完毕。停止位的长度有三种：1 位，1.5 位，2 位，通常我们选择 1 位即可。

## 8.2 IMX6ULL UART 寄存器介绍

UART:Universal Asynchronous Receiver/Transmitter，通用异步收发传输器。

参考资料：网盘开发板配套资料“06_Datasheet（数据手册）/Core_board/CPU/IMX6ULLRM.pdf”中《CChapter 55: Universal

Asynchronous Receiver/Transmitter (UART)》。

### 8.2.1 IMX6ULL UART 模块简介

IMX6ULL 共 8 个独立的 UART 通道，即 8 个 UART，主要特性如下：

a) 兼容高速串口标准 TIA/EIA-232-F，高达 5Mbit/s；

b) 低速串行红外接口（IR），兼容 Ir-DA（速度高达 115.2Kbit/s）；

c) 支持 9 位或多点模式（RS-485）（自动从机地址检测）；

d) 1 或 2 位停止位；

e) 可编程奇偶校验（偶校验，奇校验，不校验）；

f) 自动波特率检测（最高支持 115.2Kbit/s）；

g) 可屏蔽中断

h) 软复位（SRST_B）

以上只是列举了部分常用的特性，如需要更加详细的了解需要查看芯片手册《Chapter 55 Universal Asynchronous Receiver/Transmitter(UART)》。

### 8.2.2 IMX6ULL UART 寄存器简介

IMX6ULL 8 路 UART 通道，每一路通道都有 17 个寄存器，掌握其中一路通道，其他通道都是照葫芦画瓢，只是基地址、IO 复用的管脚不同。

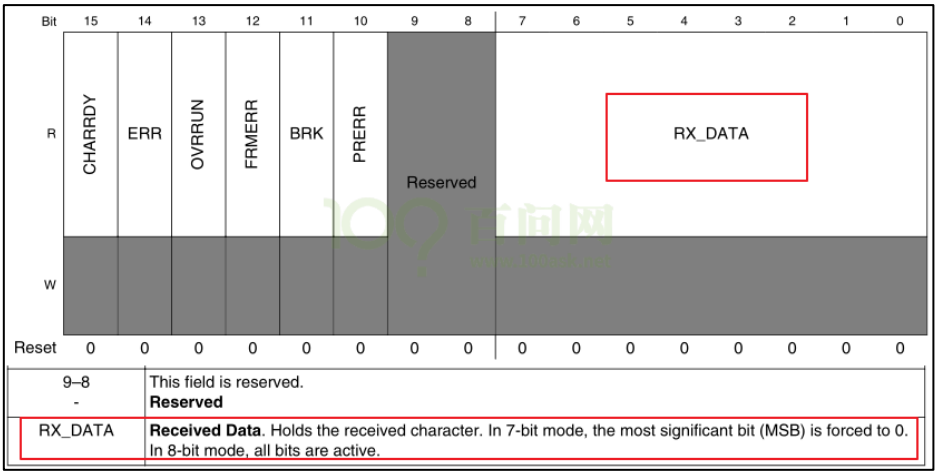
UART1 的寄存器如下图：

| | | | | | |
|---|---|---|---|---|---|
| 202_0000 | UART Receiver Register (UART1_URXD) | 32 | R | 0000_0000h | 55.15.1/ 3615 |
| 202_0040 | UART Transmitter Register (UART1_UTXD) | 32 | W | 0000_0000h | 55.15.2/ 3617 |
| 202_0080 | UART Control Register 1 (UART1_UCR1) | 32 | R/W | 0000_0000h | 55.15.3/ 3618 |
| 202_0084 | UART Control Register 2 (UART1_UCR2) | 32 | R/W | 0000_0001h | 55.15.4/ 3620 |
| 202_0088 | UART Control Register 3 (UART1_UCR3) | 32 | R/W | 0000_0700h | 55.15.5/ 3623 |
| 202_008C | UART Control Register 4 (UART1_UCR4) | 32 | R/W | 0000_8000h | 55.15.6/ 3625 |
| 202_0090 | UART FIFO Control Register (UART1_UFCR) | 32 | R/W | 0000_0801h | 55.15.7/ 3627 |
| 202_0094 | UART Status Register 1 (UART1_USR1) | 32 | R/W | 0000_2040h | 55.15.8/ 3629 |
| 202_0098 | UART Status Register 2 (UART1_USR2) | 32 | R/W | 0000_4028h | 55.15.9/ 3632 |
| 202_009C | UART Escape Character Register (UART1_UESC) | 32 | R/W | 0000_002Bh | 55.15.10/ 3634 |
| 202_00A0 | UART Escape Timer Register (UART1_UTIM) | 32 | R/W | 0000_0000h | 55.15.11/ 3635 |
| 202_00A4 | UART BRM Incremental Register (UART1_UBIR) | 32 | R/W | 0000_0000h | 55.15.12/ 3635 |
| 202_00A8 | UART BRM Modulator Register (UART1_UBMR) | 32 | R/W | 0000_0000h | 55.15.13/ 3636 |
| 202_00AC | UART Baud Rate Count Register (UART1_UBRC) | 32 | R | 0000_0004h | 55.15.14/ 3636 |
| 202_00B0 | UART One Millisecond Register (UART1_ONEMS) | 32 | R/W | 0000_0000h | 55.15.15/ 3637 |
| 202_00B4 | UART Test Register (UART1_UTS) | 32 | R/W | 0000_0060h | 55.15.16/ 3638 |
| 202_00B8 | UART RS-485 Mode Control Register (UART1_UMCR) | 32 | R/W | 0000_0000h | 55.15.17/ 3639 |

而我们本次实验使用到的只有上面红色框框 9 项。

- **UART1_URXD**

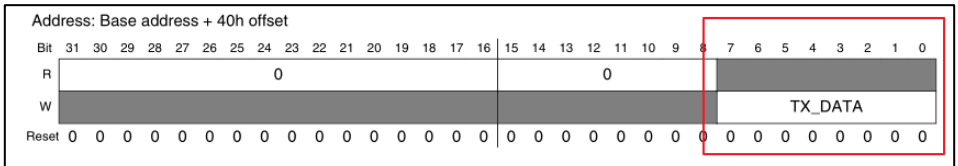
主要用于接收串口数据的寄存器，只有低八位的空间是存储接收数据的，其他是一些判断位，基本用不上；有数据时，读取这个寄存器就可以得到数据。




| | |
|---|---|
| 9–8<br>- | This field is reserved.<br>**Reserved** |
| RX_DATA | **Received Data**. Holds the received character. In 7-bit mode, the most significant bit (MSB) is forced to 0. In 8-bit mode, all bits are active. |

➢ **UART1_UTXD**

用于发送串口数据的寄存器，只有低八位的空间用于发送数据，其他位保留不使用；要发送数据时，写入这个寄存器就可以了。

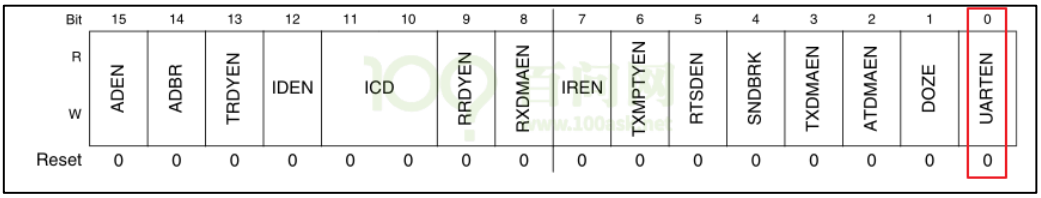



**UARTx_UTXD field descriptions**

| Field | Description |
|---|---|
| 31–16<br>Reserved | This read-only field is reserved and always has the value 0. |
| 15–8<br>Reserved | This read-only field is reserved and always has the value 0. |
| TX_DATA | **Transmit Data**. Holds the parallel transmit data inputs. In 7-bit mode, D7 is ignored. In 8-bit mode, all bits are used. Data is transmitted least significant bit (LSB) first. A new character is transmitted when the TX_DATA field is written. The TX_DATA field must be written only when the TRDY bit is high to ensure that corrupted data is not sent. |

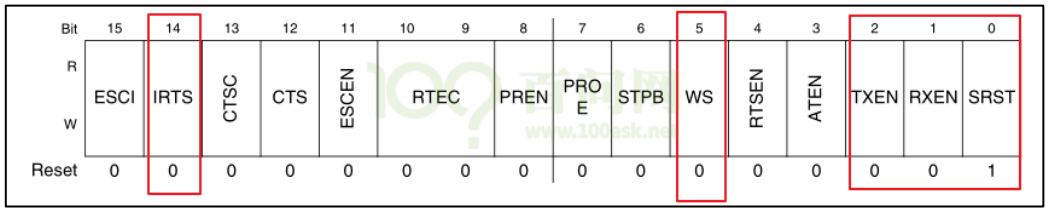
➢ **UART1_UCR1**

控制寄存器 1，用于设置串口各类功能的使能，例如自动波特率检测的使能，发送中断，串口 DMA 使能，串口使能等。而我们本章节只使用 bit0，串口使能即可。



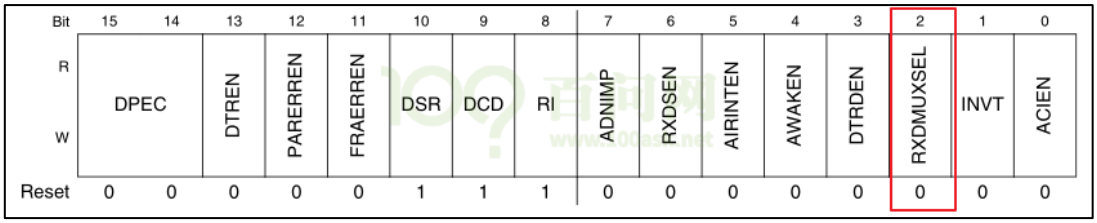

➢ **UART1_UCR2**

控制寄存器 2：主要用于设置串口的发送帧格式，帧长，是否奇偶校验，是否忽略有 RTS，软复位等，本章只使用其中的 5 位，后续 UART 编程会详细讲解。



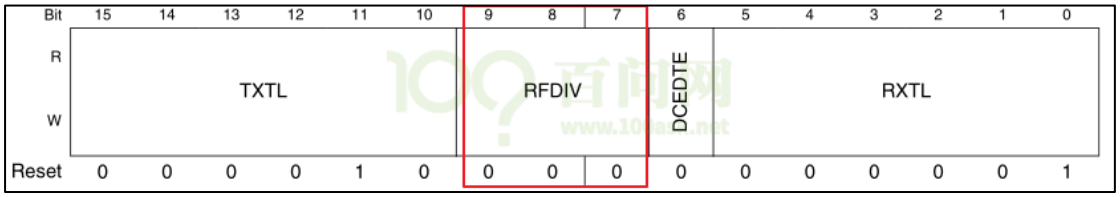
➢ **UART1_UCR3**

控制寄存器 3：我们只设置 bit2，官方要求设置，属于芯片特点。



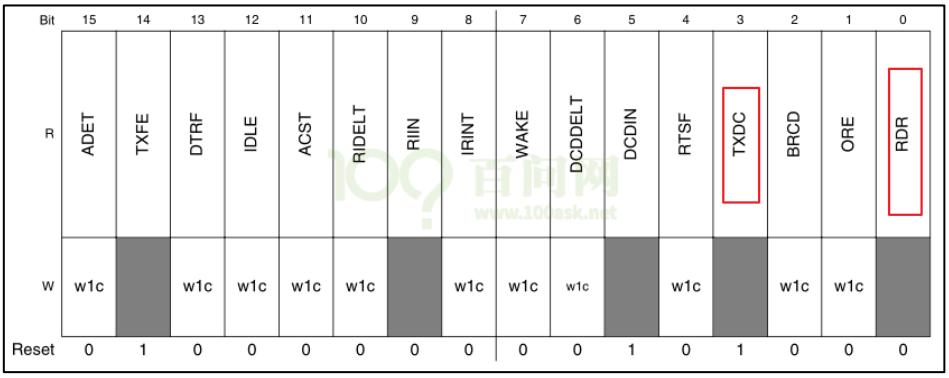

➢ **UART1_UFCR**

串口 FIFO 控制寄存器，设置发送与接收的 fifo 的大小，最大 32 字节，串口时钟分频系数等，只要把 RFDIV 此位设置为不分频，其他用默认值即可，更详细使用会在后面的 UART 编程中讲解。




➢ **UART1_USR2**

串口状态寄存器，该寄存器里面主要是一些串口的状态位，我们本章只使用了 TXDC 发送完成位与 ROR 接收数据就绪位。详细使用会在后面 UART 编程中讲解




➢ **UART1_UBIR 与 UART1_UBMR**

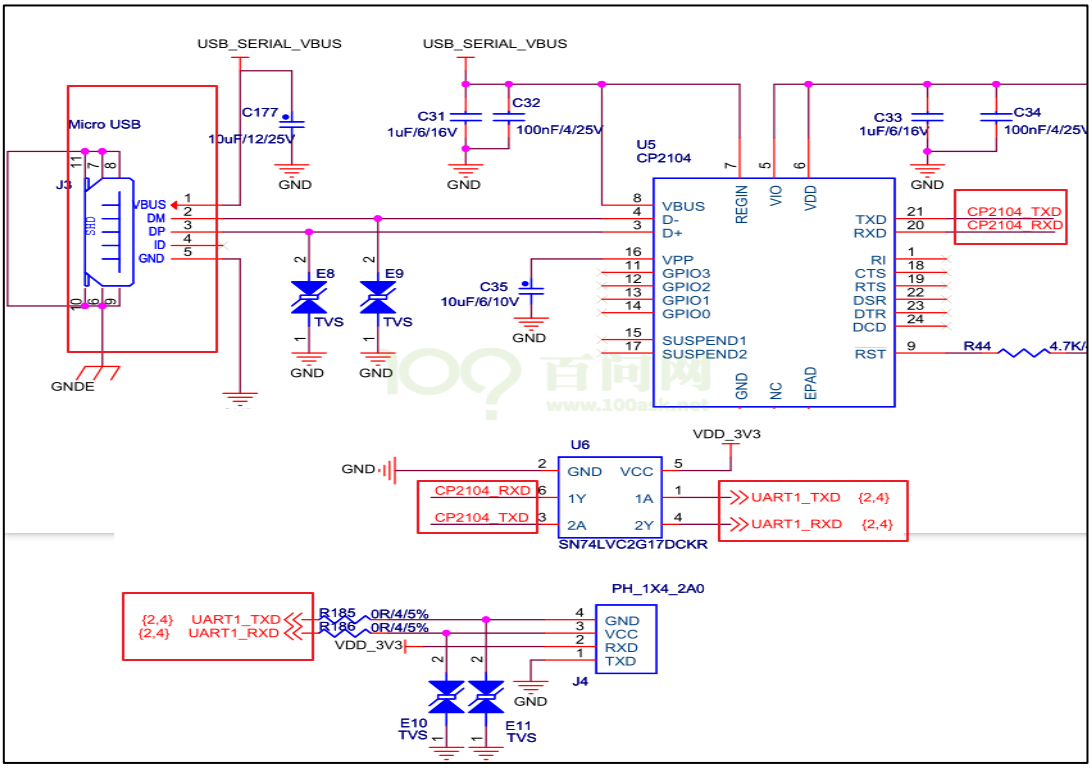
用于设置波特率即每秒可传输的位数，后面 UART 编程会详细讲解。

基本我们使用的寄存器就这些，其他都是一些功能扩展，想详细了解的可以查看芯片手册的《Chapter 55 Universal Asynchronous Receiver/Transmitter(UART)》

# 8.3 IMX6ULL UART 编程

## 8.3.1 看原理图确定 UART 引脚

原理图：网盘开发板配套资料“05_Hardware（原理图)/Base_board/100ask_imx6ull_v1.1.pdf”。




从上图可知，采用的 UART 转 USB 的方案，使用的是 UART1。查看 IMX6ULL 芯片手册《Chapter 55 Universal Asynchronous Receiver/Transmitter(UART)》中涉及关键字 UART1 的寄存器并设置它。

## 8.3.2 涉及的 UART1 寄存器配置

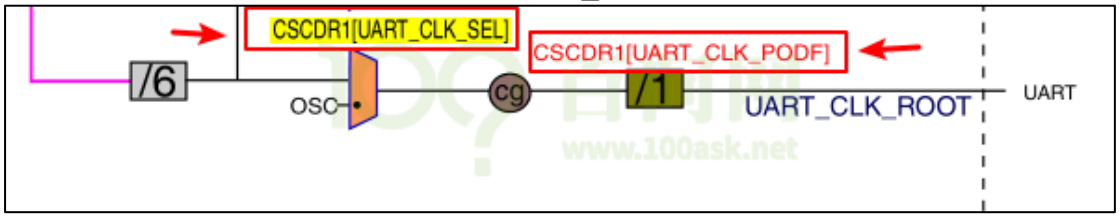
代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/8_UART 串口编程/001_uart_txd_char”目录下。

寄存器配置共分为 4 步骤，本小节中的代码都在程序文件 uart.c 中。

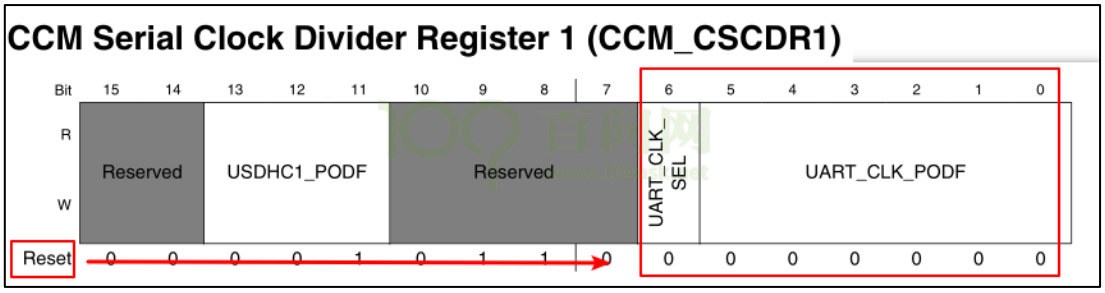
➢ **步骤 1：配置并使能 UART1 时钟**

配置并使能 UART1 时钟，参考资料：芯片手册《Chapter 18: Clock Controller Module (CCM)》。

① 配置 UART 模块的时钟（寄存器：CCM_CSCDR1）：

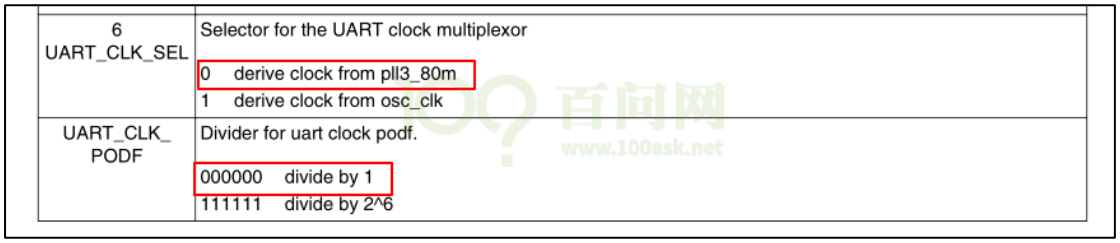



根据上图可知，我们需要设置 CCM_CSCDR1 [UART_CLK_SEL]和 CCM_CSCDR1 [UART_CLK_PODF]。



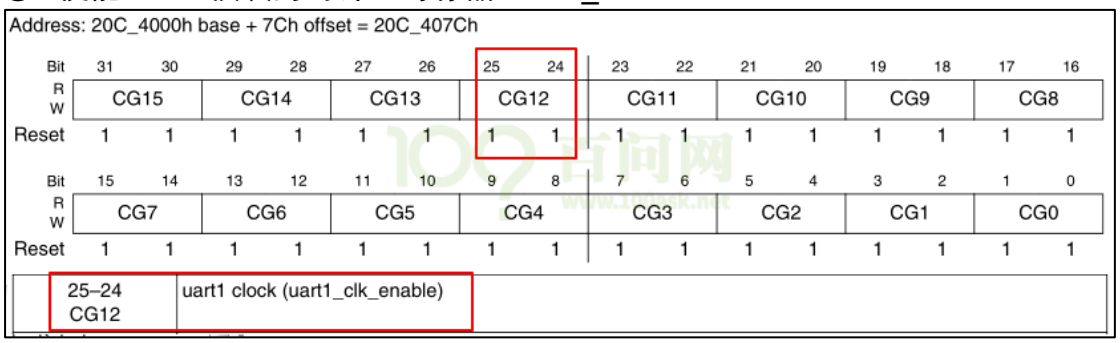

由上图 CCM_CSCDR1 寄存器，我们可以了解到 CCM_CSCDR1 [UART_CLK_SEL] 默认值为 0；CCM_CSCDR1 [UART_CLK_PODF]默认值为 0

我们一般选择 pll3_80m 作为 UART 的时钟源，UART_CLK_PODF 分频系数选 1 分频（不分频），最后得到 UART 的时钟频率为 80MHz。




正好默认值都为 0 满足我们的时钟需求，所以后续的编程实验，串口时钟这部分可以不设置，用默认值就可以了。

② 使能 UART 模块的时钟（寄存器：CCM_CCGR5）




由上图 CCM_CCGR5 寄存器，我们可以了解到 CCM_CCGR5[CG12] 的默认值为 11。参考章节《4.2.6 CCM 用于设置是否向 GPIO 模块提供时钟》我们了解到 11 表示该模块全程使能，使用默认值，无需设置。

<u>因此时钟这块我们都不需要配置，直接使用默认值即可</u>。

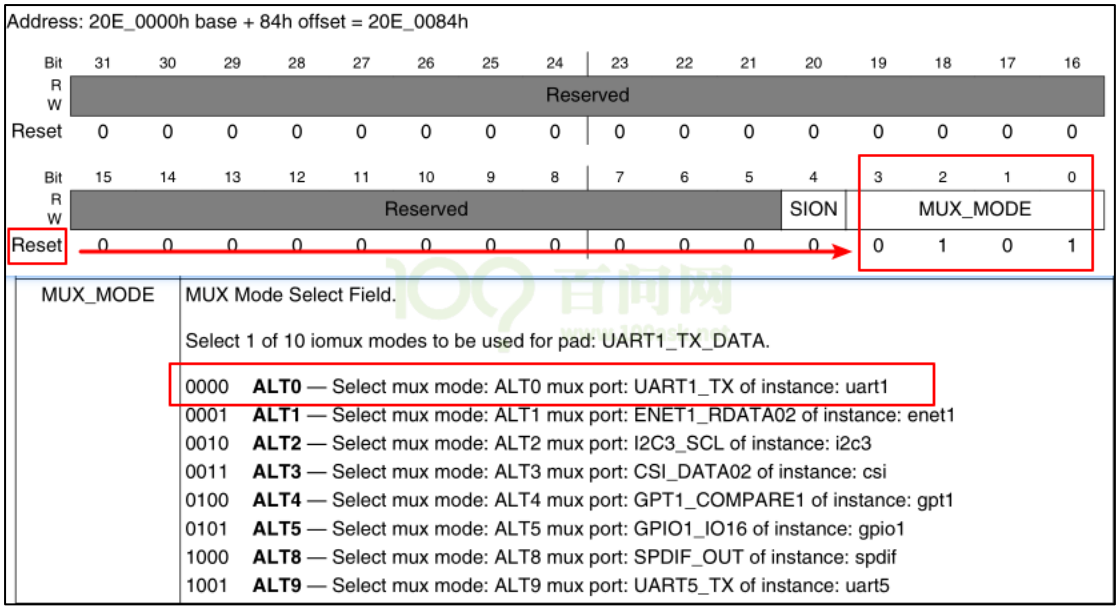
➢ **步骤 2：复用相关 GPIO 为 UART1 功能**

复用相关 GPIO 为 UART1 功能，参考资料：芯片手册《Chapter 32: IOMUX Controller (IOMUXC)》。

根据前面硬件连接讲解，我们知道串口通信精简接法需要 3 根信号线，其中 GND 已由硬连接了，所以接下来我们需要配置剩余的两个 GPIO 引脚（UART1_TXD 与 UART1_RXD）。

① 配置 UART1_TX 复用功能

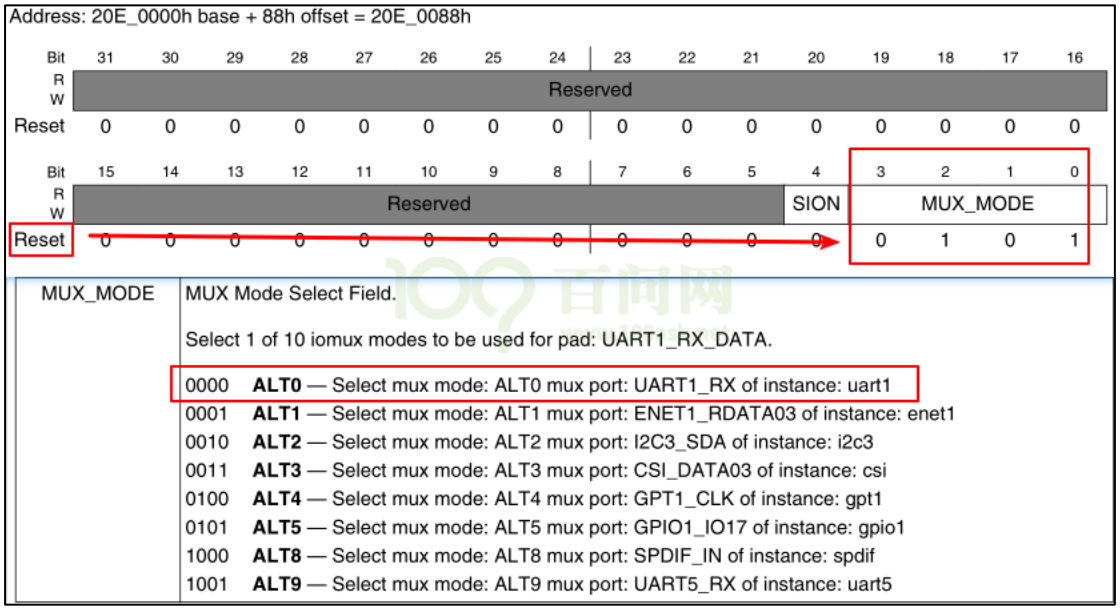
寄存器：IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA。




由上图我们得知 IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA[MUX_MODE]的默认值为 0101，因此我们需要将其改为 0，用于 UART_TX 功能。

② 配置 UART1_RX 复用功能

寄存器：IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA。



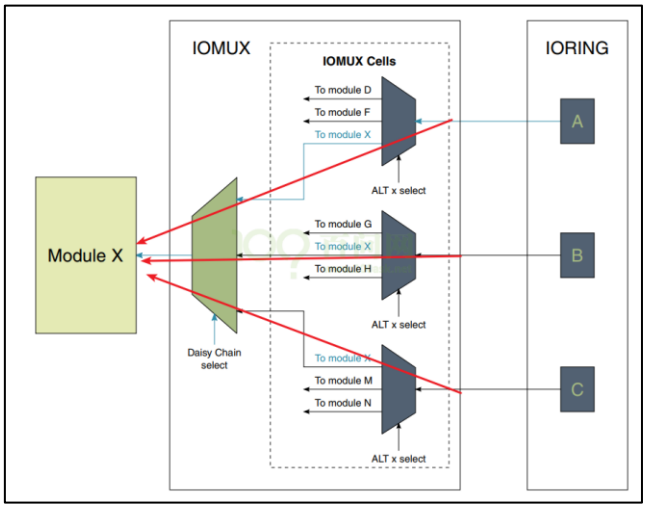

由上图我们得知 IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA[MUX_MODE]的默认值为 0101，因此我们需要将其改为 0，用于 UART_RX 功能。（程序文件：uart.c）

```
IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA        = (volatile unsigned int *)(0x20E0084);
IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA        = (volatile unsigned int *)(0x20E0088);

*IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA = 0;
*IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA = 0;
```

**着重提醒：IMX6ULL 中，可能会有多个引脚都可以驱动某个模块，还需要进一步选择。**

比如下图中：A、B、C 三个引脚都可以设置为工作于 Module X，它们都可以驱动 Module X。但是使用哪一个引脚呢？还需要设置“Daisy Chain select”，

用来选择 A、B、C 之一。




对于 UART1_RX 引脚，我们除了设置 IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA 让它工作于 ALT0 之外，还需要设置寄存器 IOMUXC_UART1_RX_DATA_SELECT_INPUT，如下图所示：

Address: 20E_0000h base + 624h offset = 20E_0624h

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | Reserved | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

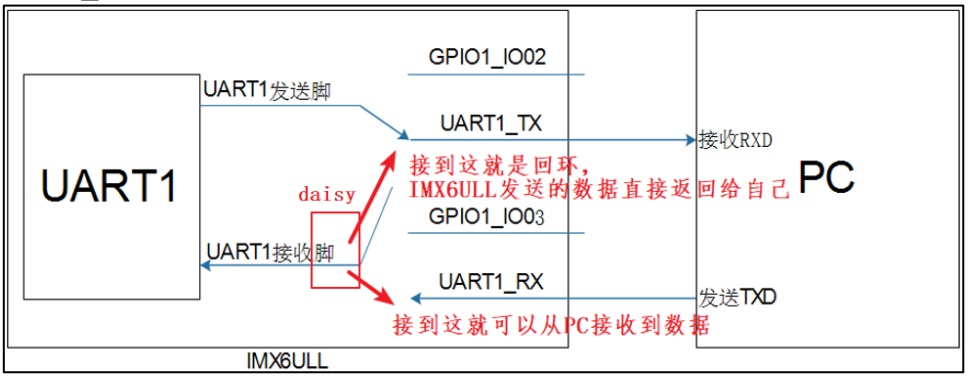
| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | Reserved | | | | | | | | | | | | | | DAISY | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**IOMUXC_UART1_RX_DATA_SELECT_INPUT field descriptions**

| Field | Description |
|---|---|
| 31–2<br>- | This field is reserved.<br>Reserved |
| DAISY | Selecting Pads Involved in Daisy Chain.<br><br>Instance: uart1, In Pin: uart_RX_DATA<br><br>00 **GPIO1_IO02_ALT8** — Selecting Pad: GPIO1_IO02 for Mode: ALT8<br>01 **GPIO1_IO03_ALT8** — Selecting Pad: GPIO1_IO03 for Mode: ALT8<br>10 **UART1_TX_DATA_ALT0** — Selecting Pad: UART1_TX_DATA for Mode: ALT0<br>11 **UART1_RX_DATA_ALT0** — Selecting Pad: UART1_RX_DATA for Mode: ALT0 |

在上图中，引脚 GPIO1_IO02 设置为 ALT8 模式时，它也可以用作 UART1 的 RX 输入引脚；引脚 UART1_RX 设置为 ALT0 模式时，它也可以用作 UART1 的 RX 输入引脚。

UART1_RX 为何可以选择那么多引脚？请看下图：




- UART1 的发送引脚有 2 个选择：GPIO1_IO02、UART1_TX，假设选择了 UART1_TX 作为发送引脚。

- UART1 的接收引脚有 2 个选择：GPIO1_IO03、UART1_RX，假设选择了 UART1_RX 作为接收引脚。

但是 UART1 的模块还可能通过 daisy 引脚再次选择：GPIO1_IO02、GPIO1_IO03、UART1_TX、UART1_RX 中的某一个。比如上图中，选择 UART1_TX 时，这就构成了回环，可以用于测试：IMX6ULL 发出去的数据直接返回给自己；选择 UART_RX 是，就可以接收外面设备比如 PC 机的数据。

在 daisy 中选择哪一个引脚连接到 UART1 模块？还需要进一步设置，代码如下(我们选择引脚 UART1_RX)：

```
IOMUXC_UART1_RX_DATA_SELECT_INPUT                = (volatile unsigned int
*)(0x20E0624);
*IOMUXC_UART1_RX_DATA_SELECT_INPUT = 3;
```

③ 配置 UART1_TX 硬件参数

寄存器：IOMUXC_SW_PAD_CTL_PAD_UART1_TX_DATA。

参考章节《4.2.6 IOMUXC：引脚的模式(Mode、功能)》，经过比对，我们使用默认值（0x10b0）即可，无需配置。

④ 配置 UART1_RX 硬件参数

寄存器：IOMUXC_SW_PAD_CTL_PAD_UART1_RX_DATA。

参考章节《4.2.6 IOMUXC：引脚的模式(Mode、功能)》， 经过比对，我们使用默认值（0x10b0）即可，无需配置。

➢ **步骤 3：设置 UART1 传输格式，波特率**

① 配置寄存器 UART1_UCR2（0x2020084）

Address: Base address + 80h offset

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R | 0 | | | | | | | | | | | | | | | |
| W | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R | ADEN | ADBR | TRDYEN | IDEN | ICD | | RRDYEN | RXDMAEN | IREN | TXMPTYEN | RTSDEN | SNDBRK | TXDMAEN | ATDMAEN | DOZE | UARTEN |
| W | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

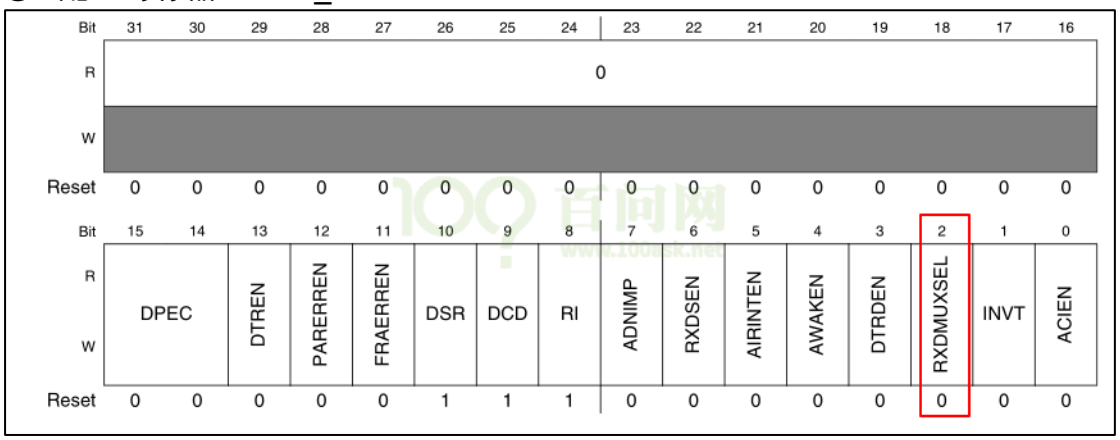
| | |
|---|---|
| 14<br>IRTS | **Ignore RTS Pin**. Forces the RTS input signal presented to the transmitter to always be asserted (set to low), effectively ignoring the external pin. When in this mode, the RTS pin serves as a general purpose input.<br><br>0 Transmit only when the RTS pin is asserted<br>1 Ignore the RTS pin |
| 5<br>WS | **Word Size**. Controls the character length. When WS is high, the transmitter and receiver are in 8-bit mode. When WS is low, they are in 7-bit mode. The transmitter ignores bit 7 and the receiver sets bit 7 to 0. WS can be changed in-between transmission (reception) of characters, however not when a transmission (reception) is in progress, in which case the length of the current character being transmitted (received) is unpredictable.<br><br>0 7-bit transmit and receive character length (not including START, STOP or PARITY bits)<br>1 8-bit transmit and receive character length (not including START, STOP or PARITY bits) |
| 4<br>RTSEN | **Request to Send Interrupt Enable.** Controls the RTS edge sensitive interrupt. When RTSEN is asserted and the programmed edge is detected on the RTS_B pin (the RTSF bit is asserted), an interrupt will be generated on the *interrupt_uart* pin. (See Table 55-5.)<br><br>0 Disable request to send interrupt<br>1 Enable request to send interrupt |
| 3<br>ATEN | **Aging Timer Enable**. This bit is used to enable the aging timer interrupt (triggered with AGTIM)<br><br>0 AGTIM interrupt disabled<br>1 AGTIM interrupt enabled |
| 2<br>TXEN | **Transmitter Enable**. Enables/Disables the transmitter. When TXEN is negated the transmitter is disabled and idle. When the UARTEN and TXEN bits are set the transmitter is enabled. If TXEN is negated in the middle of a transmission, the UART disables the transmitter immediately, and starts marking 1s. The transmitter FIFO cannot be written when this bit is cleared.<br><br>0 Disable the transmitter<br>1 Enable the transmitter |
| 1<br>RXEN | **Receiver Enable**. Enables/Disables the receiver. When the receiver is enabled, if the RXD input is already low, the receiver does not recognize BREAK characters, because it requires a valid 1-to-0 transition before it can accept any character.<br><br>0 Disable the receiver<br>1 Enable the receiver |
| 0<br>SRST | **Software Reset**. Once the software writes 0 to SRST_B, the software reset remains active for 4 *module_clock* cycles before the hardware deasserts SRST_B. The software can only write 0 to SRST_B. Writing 1 to SRST_B is ignored.<br><br>0 Reset the transmit and receive state machines, all FIFOs and register USR1, USR2, UBIR, UBMR, UBRC , URXD, UTXD and UTS[6-3].<br>1 No reset |

设置 UART1 传输格式

```
UART1->UCR2 |= (1<<14) |(1<<5) |(1<<2)|(1<<1);
```

- [14]: 1：忽略 RTS 引脚
- [8] : 0：关闭奇偶校验　　默认为 0，无需设置
- [6] : 0：停止位 1 位　　默认为 0，无需设置
- [5] : 1：数据长度 8 位
- [2] : 1：发送数据使能
- [1] : 1：接收数据使能

② 配置寄存器 UART1_UCR3（0x2020088）




| | |
|---|---|
| 2<br>RXDMUXSEL | RXD Muxed Input Selected. Selects proper input pins for serial and Infrared input signal.<br>**NOTE:** In this chip, UARTs are used in MUXED mode, so that this bit should always be set. |

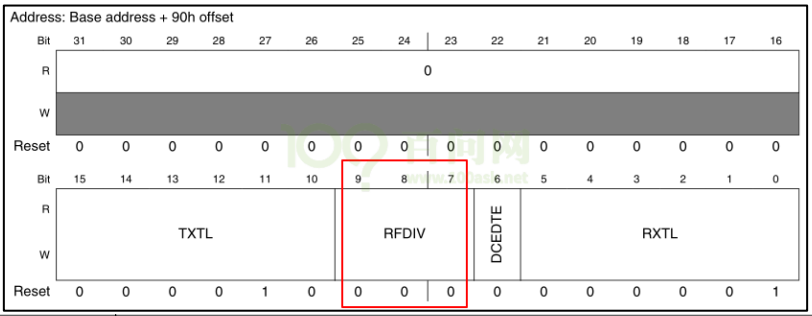
根据官方文档表示 [RXDMUXSEL]需要设置为 1。

[2]: 1:IMX6ULL 的 UART 用了这个 MUXED 模型，因此这一位需要置位为 1

设置串口模型

```
UART1->UCR3 |= (1<<2);
```

③ 寄存器 UART1_UFCR（0x2020090）




| | |
|---|---|
| 9–7<br>RFDIV | Reference Frequency Divider. Controls the divide ratio for the reference clock. The input clock is *module_clock*. The output from the divider is *ref_clk* which is used by BRM to create the 16x baud rate oversampling clock (*brm_clk*).<br><br>000 Divide input clock by 6<br>001 Divide input clock by 5<br>010 Divide input clock by 4<br>011 Divide input clock by 3<br>100 Divide input clock by 2<br>101 Divide input clock by 1<br>110 Divide input clock by 7<br>111 Reserved |

UART1_UFCR[9-7]：UART 的时钟源分频系数

我们选择 101（即十进制的 5），表示不分频。

```
UART1->UFCR = 5 << 7;        /* Uart 的时钟 clk：80MHz */
```

④ 寄存器 UART1_UBIR（0x20200A4），UART1_UBMR（0x20200A8）通过设置 UART1_UBIR 与 UART1_UBMR，最终确定波特率。

Address: Base address + A4h offset

| Bit | 31–16 | 15–0 |
|---|---|---|
| R | 0 | INC |
| W | | |
| Reset | 0 | 0 |

**UARTx_UBIR field descriptions**

| Field | Description |
|---|---|
| 31–16 Reserved | This read-only field is reserved and always has the value 0. |
| INC | Incremental Numerator. Holds the numerator value minus one of the BRM ratio (see Binary Rate Multiplier (BRM)). The UBIR register MUST be updated before the UBMR register for the baud rate to be updated correctly. If only one register is written to by software, the BRM will ignore this data until the other register is written to by software. Updating this field using byte accesses is not recommended and is undefined. |

Address: Base address + A8h offset

| Bit | 31–16 | 15–0 |
|---|---|---|
| R | 0 | MOD |
| W | | |
| Reset | 0 | 0 |

**UARTx_UBMR field descriptions**

| Field | Description |
|---|---|
| 31–16 Reserved | This read-only field is reserved and always has the value 0. |
| MOD | **Modulator Denominator.** Holds the value of the denominator minus one of the BRM ratio (see Binary Rate Multiplier (BRM)). The UBIR register MUST be updated before the UBMR register for the baud rate to be updated correctly. If only one register is written to by software, the BRM will ignore this data until the other register is written to by software. Updating this register using byte accesses is not recommended and undefined. |

IMX6ULL 波特率计算公式：

$$BaudRate = \frac{Ref\ Freq}{\left(16 \times \frac{UBMR + 1}{UBIR + 1}\right)}$$

**Figure 55-10. Frequency and Baud Rate for UBIR and UBMR**

a) 设置 115200 的波特率即 BaudRate = 115200;

b) UART1 的时钟频率前面内容已确定 80Mhz 即 Ref Freq = 80000000;

c) IMX6ULL 波特率计算公式得 115200 = 80000000 /(16*(UBMR + 1)/(UBIR+1));

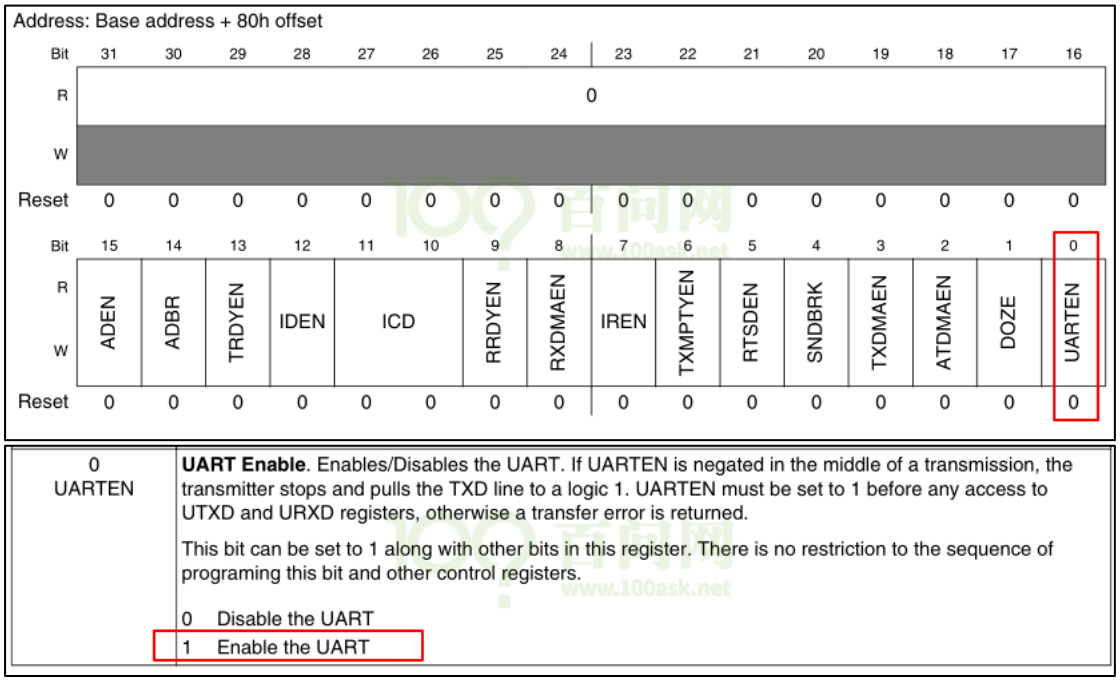
d) 选取一组满足上式的参数：UBMR、UBIR 即可;

e) UART1_UBIR = 71 ; UART1_UBMR = 3124

```
UART1->UBIR = 71;
UART1->UBMR = 3124;
```

➢ **步骤 4：使能 UART1**

使能 UART1，UART1_UCR1(0x2020080)寄存器如下：




配置 UART1_UCR1[0]：1 表示使能 UART， 0 表示关闭 UART。

```
Base->UCR1 |= (1 << 0);       /*使能当前串口*/
```

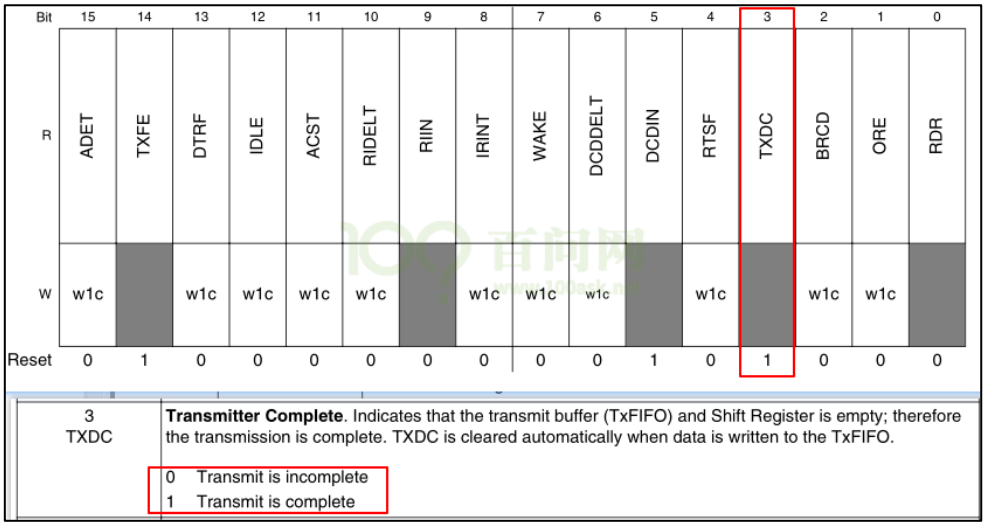
## 8.3.3 实现串口发送功能

➢ **步骤 1： 编写 UART1 发送单字节函数**

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/8_UART 串口编程/001_uart_txd_char”目录下。

写这个函数之前，我们需要了解什么时候才能发，什么时候不能发。

只有当上一个数据发完的时候，我们才能继续发送，因此需要用到 UART1_USR2 寄存器中表示 UART1 发送状态的只读状态位[TXDC]。




UART1_USR2[3] ： 0 表示发送未完成 ， 1 表示发送已完成

程序文件：uart.c

```
void PutChar(int c)
{
    while (!((UART1->USR2) & (1<<3))); /*等待上个字节发送完毕*/
```

```
    UART1->UTXD = (unsigned char)c;
}
```

➢ **步骤 2：编写用于测试的 main 函数**

程序文件：main.c

```
#include "uart.h"

int  main()
{
    unsigned char cTestData = 'A'; /*用于测试发送的数据*/
    Uart_Init()   ;

    while(1)
    {
        PutChar(cTestData);
    }

    return 0;
}
```

注明：整个完整工程代码目录在裸机 Git 仓库 NoosProgramProject/(8_UART 串口编程/001_uart_txd_char)文件夹下。

➢ **步骤 3：参考章节《4.3.4 编译程序》编译程序**

➢ **步骤 4：参考章节《3.4 映像文件烧写、运行》烧写、运行程序**

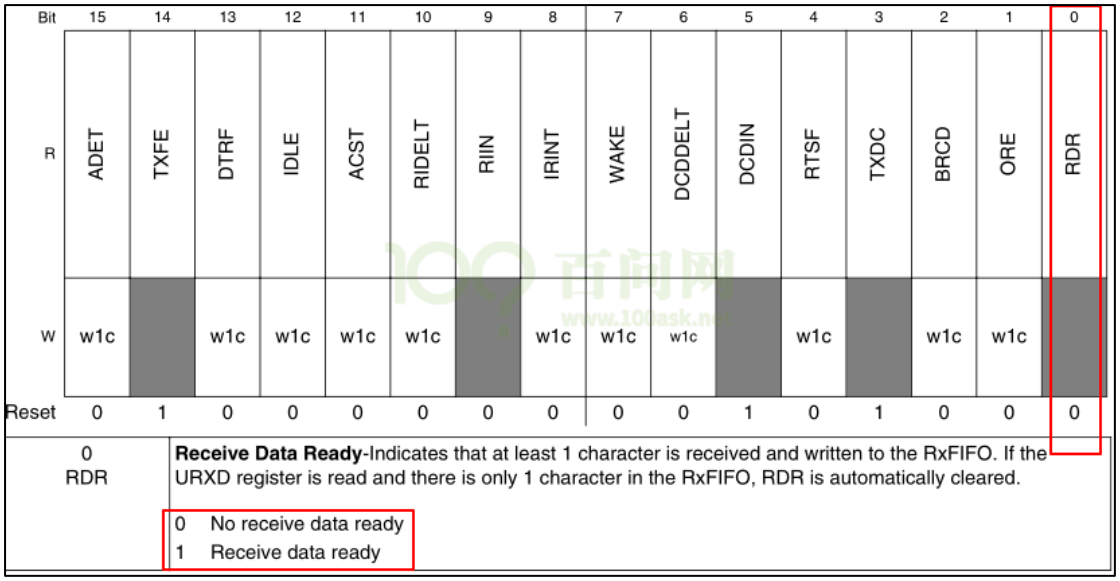
如果实验结果如上图电脑串口将会不断收到 ‘A’，表明实验成功。

## 8.3.4 实现串口接收功能

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/8_UART 串口编程/002_uart_txd_char”目录下。

➢ **步骤 1：编写 UART1 接收单字节函数**

编写 UART1 接收单字节函数，接收单字节时，我们也需要去判定 UART1_USR2 寄存器中的只读状态位[ROR]。




UART1_USR2[0] ： 0 表示没有接收数据就绪， 1 表示接收数据准备就绪。

程序文件：uart.c

```
unsigned char GetChar(void)
{
    while (!(UART1->USR2 & (1<<0)));  /*等待接收数据*/
    return (unsigned char)UART1->URXD;
}
```

## 第4步 步骤 2：编写用于测试 main 函数

编写用于测试 main 函数，我们让串口接收到的数据，交给上一小节中所写的发送函数中，从而实现串口回显。

程序文件：main.c

```
#include "uart.h"

int  main()
{
    unsigned char cTestData ;       /*用于测试发送的数据*/
    Uart_Init()   ;

    while(1)
    {
        cTestData = GetChar() ;                 /*等待从串口获取数据*/
        PutChar(cTestData)    ;                 /*从串口发送数据*/
    }

    return 0;
}
```

接着我们参照参考《4.3.4 编译程序》与参考《3.4 映像文件烧写、运行》，上机验证


在键盘中输入的数据，会在终端回显出来表明实验成功，例如：如上图，在键盘上输入 100ask.6ull，终端会完整回显出来。

## 8.3.5 完善回显功能

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/8_UART 串口编程/003_uart_better_char”目录下。

我们会发现上一小节所写的函数中有点问题，当我们按下回车的时候不会换行到行首，而是直接到行首，所以我们稍微修改 main 函数完善它。

程序文件：main.c

```
while(1)
    {
        cTestData = GetChar() ;                 /*等待从串口获取数据*/

            if (cTestData == '\r')          /*添加回车换行\n\r*/
            {
                PutChar('\n');
            }

            if (cTestData == '\n')
            {
                PutChar('\r');
            }

        PutChar(cTestData)    ;                 /*从串口发送数据*/
    }
```

在获得数据的时候，判定回车符‘\r’和换行符‘\n’即可，重新烧写上机实验，按下回车键就能回车换行

## 8.3.6 实现串口发送字符串功能

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/8_UART 串口编程/004_uart_str”目录下。

- **步骤 1：实现打印字符串函数**

实现打印字符串函数，在发送单字节的基础上，加上判断语句，实现连续打印字符。

程序文件：my_printf.c

```
void PutStr(const char *s)
{
    while (*s)
    {
        PutChar(*s);
        s++;
    }
}
```

- **步骤 2：在 main 函数中添加打印字符串函数的调用**

在 main 函数中添加打印字符串函数的调用。

程序文件：main.c

```
PutStr("Hello, world!\n\r");  /*发送字符串*/
```

- **步骤 3：参考章节《4.3.4 编译程序》编译程序**
- **步骤 4：参考章节《3.4 映像文件烧写、运行》烧写、运行程序**

终端将打印出 Hello，world!

```
Hello, world!
```

## 8.4 移植 printf

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/8_UART 串口编程/005_myprintf_test”目录下。

### 8.4.1 在上一个程序的基础上移植

上一个程序是 004_uart_str，下面按顺序修改。

① 在 uart.c 中加入 raise 函数，用于防止编译失败。

程序文件：uart.c

```
119 int raise(int signum)/* raise 函数，防止编译报错 */
120 {
121     return 0;
122 }
```

② 修改 Makefile

my_printf.c 中用到除法的求模运算，需要提供除法库。一般的交叉工具链里都提示有基本的数学运算，它们位于 libgcc.a 中。我们需要把 libgcc.a 也链接进程序里，需要修改 Makefile。

注意：链接指令中，每个“-L”表示库在哪里，即它的目录；“-l”表示哪个库，即库的名称，-lgcc 表示会链接“libgcc.a”库。

对 Makefile 作如下修改：

a) 增加`$(CC) -nostdlib -g -c -o my_printf.o my_printf.c`

b) 在`$(LD) -T imx6ull.lds -g start.o uart.o main.o my_printf.o -o my_printf.elf`后添加`-lgcc -L<libgcc.a`的路径>

```
例如：$(LD) -T imx6ull.lds -g start.o uart.o main.o my_printf.o -o my_printf.elf -
lgcc
-L/home/book/100ask_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86_64_arm-
linux-gnueabihf/lib/gcc/arm-linux-gnueabihf/6.2.1
```

### 8.4.2 变参数函数移植

函数参数列表包括了字符串（format）和变参（…）组合而成，在 vc6.0 的头文件 stdarg.h 中找到 `typedef char *  va_list;`

```
#define _INTSIZEOF(n)   ( (sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1) )
#define va_start(ap,v)  ( ap = (va_list)&v + _INTSIZEOF(v) )
#define va_arg(ap,t)    ( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )
#define va_end(ap)      ( ap = (va_list)0 )
```

① _INTSIZEOF(n) 用于获取其中一个变参类型占用的空间长度

② va_start(ap,v) 令 ap 指向第一个变参地址

③ va_arg(ap,t) 取出一个变参，同时指针指向下一个变参

④ va_end(ap) 将指针指向 NULL，防止野指针

移植以上的代码，编写一个属于自己的 printf

参考 int printf(const char *format, ...)库函数，实现 my_printf。

程序文件：my_printf.c

```
    int printf(const char *fmt, ...)
    {
```

```
    va_list ap;

    va_start(ap, fmt);
    my_vprintf(fmt, ap);
    va_end(ap);
    return 0;
}
```

### 8.4.3 编写 my_vprintf(fmt, ap)

参考 int vprintf(const char *format, va_list ap)实现 my_vprintf
程序文件：my_printf.c

```
static int my_vprintf(const char *fmt, va_list ap)
{
    char lead=' ';
    int  maxwidth=0;

     for(; *fmt != '\0'; fmt++)
     {
        if (*fmt != '%') {
            outc(*fmt);
            continue;
        }

         lead=' ';
         maxwidth=0;

        //format : %08d, %8d,%d,%u,%x,%f,%c,%s
           fmt++;
        if(*fmt == '0'){
            lead = '0';
            fmt++;
        }


        while(*fmt >= '0' && *fmt <= '9'){
            maxwidth *=10;
        maxwidth += (*fmt - '0');
        fmt++;
    }

    switch (*fmt) {
    case 'd': out_num(va_arg(ap, int),         10,lead,maxwidth); break;
   case 'o': out_num(va_arg(ap, unsigned int),  8,lead,maxwidth); break;

    case 'u': out_num(va_arg(ap, unsigned int), 10,lead,maxwidth); break;
    case 'x': out_num(va_arg(ap, unsigned int), 16,lead,maxwidth); break;
    case 'c': outc(va_arg(ap, int   )); break;
   case 's': outs(va_arg(ap, char *)); break;

   default:
        outc(*fmt);
       break;
        }
    }
    return 0;
}
```

### 8.4.4 编写 out_c，outs 与 out_num 函数

➢ 利用之前我们实现的单字节打印函数 **void PutChar(int c)**实现 **out_c，outs** 与 **out_num** 函数 **outc** 用于格式化输出中的**%c** 的输出。

程序文件：my_printf.c

```
static int outc(int c)
{
    PutChar(c);
    return 0;
}
```

➢ **outs** 用于格式化输出中的**%s** 的输出。

程序文件：my_printf.c

```
static int outs (const char *s)
{
    while (*s != '\0')
        PutChar(*s++);
    return 0;
}
```

➢ **out_num** 用于格式化输出中的**%d，%o，%u，%x** 的输出。

程序文件：my_printf.c

```
    static int out_num(long n, int base,char lead,int maxwidth)
    {
        unsigned long m=0;
        char buf[MAX_NUMBER_BYTES], *s = buf + sizeof(buf);
        int count=0,i=0;


        *--s = '\0';

        if (n < 0){
            m = -n;
        }
        else{
            m = n;
        }

        do{
            *--s = hex_tab[m%base];
            count++;
        }while ((m /= base) != 0);

        if( maxwidth && count < maxwidth){
            for (i=maxwidth - count; i; i--)
                *--s = lead;
    }

        if (n < 0)
            *--s = '-';

        return outs(s);
```

## 8.4.5 编写 my_printf_test 的测试函数

程序文件：my_printf.c

```
int my_printf_test(void)
{
    printf("This is www.100ask.org   my_printf test\n\r") ;
    printf("test char          =%c,%c\n\r", 'A','a') ;
    printf("test decimal number =%d\n\r",    123456) ;
    printf("test decimal number =%d\n\r",    -123456) ;
    printf("test hex     number =0x%x\n\r",  0x55aa55aa) ;
    printf("test string        =%s\n\r",    "www.100ask.org") ;
    printf("num=%08d\n\r",   12345);
    printf("num=%8d\n\r",    12345);
    printf("num=0x%08x\n\r", 0x12345);
    printf("num=0x%8x\n\r",  0x12345);
    printf("num=0x%02x\n\r", 0x1);
    printf("num=0x%2x\n\r",  0x1);

    printf("num=%05d\n\r", 0x1);
    printf("num=%5d\n\r",  0x1);

    return 0;
}
```

## 8.4.6 编写 main 测试程序

在 main 函数中只需要调用 8.3.5 中实现的 my_printf_test 即可。

程序文件：main.c

```
#include "my_printf.h"
#include "uart.h"
int  main()
{
Uart_Init();
my_printf_test();
return 0;
}
```

- **步骤 1：参考章节《4.3.4 编译程序》编译程序**
- **步骤 2：参考章节《3.4 映像文件烧写、运行》烧写、运行程序**

最后编译程序烧写至开发板，观察终端输出信息。

```
This is www.100ask.org   my_printf test
test char          =A,a
test decimal number =123456
test decimal number =-123456
test hex     number =0x55aa55aa
test string        =www.100ask.org
num=00012345
num=   12345
num=0x00012345
num=0x   12345
num=0x01
num=0x 1
num=00001
num=    1
```

串口终端如果打印以上信息，证明实验成功！

# 第9章 重定位

## 9.1 段的概念

段是程序的组成元素。将整个程序分成一个一个段，并且给每个段起一个名字，然后在链接时就可以用这个名字来指示这些段，使得这些段排布在合适的位置。
程序的段包括

- 代码段(.text)：存放代码指令
- 只读数据段(.rodata)：存放有初始值并且 const 修饰的全局类变量（全局变量或 static 修饰的局部变量）
- 数据段(.data)：存放有初始值的全局类变量
- 零初始化段(.bss)：存放没有初始值或初始值为 0 的全局类变量
- 注释段(.comment)：存放注释

注意：

- bss 段和注释段不保存在 bin/elf 文件中
- 注释段里面的机器码是用来表示文字的

下面将通过一个实例来直观地感受程序中的段。
代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/9_重定位/001_segment”目录下。

➢ **步骤 1：在主函数文件中创建不同属性的全局变量**

程序文件：main.c

```
05 char g_charA = 'A';            //存储在 .data 段
06 const char g_charB = 'B';      //存储在 .rodata 段
07 const char g_charC;         //存储在 .bss 段
08 int g_intA = 0;             //存储在 .bss 段
09 int g_intB;                 //存储在 .bss 段
```

➢ **步骤 2：创建链接脚本**

本节会用到链接脚本，可以先不管其中内容，它的用法在后续《章节 9-1.2 链接脚本分析》中有详细说明。
链接脚本：imx6ull.lds

```
SECTIONS {
    . = 0x80100000;

    . = ALIGN(4);
    .text      :
    {
      *(.text)
    }

    . = ALIGN(4);
    .rodata : { *(.rodata) }
```

```
    . = ALIGN(4);
    .data : { *(.data) }

    . = ALIGN(4);
    __bss_start = .;
    .bss : { *(.bss) *(.COMMON) }
    __bss_end = .;
}
```

➢ **步骤 3：在 Makefile 文件中指明使用链接脚本 imx6ull.lds 控制链接过程**

```
#使用链接脚本
$(LD) -T imx6ull.lds -g start.o uart.o main.o my_printf.o -o relocate.elf -lgcc -
L/home/book/100ask_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-
gnueabihf/lib/gcc/arm-linux-gnueabihf/6.2.1
```

➢ **步骤 4：参考章节《4.3.4 编译程序》编译程序并查看反汇编文件 relocate.dis**

打开反汇编文件发现

- 在反汇编文件中程序的地址从 0x80100000 开始
- 整个程序被分为不同的段，每个段以 Disassembly of section …作为开始
- 段落之间的地址是连续的，并且从低地址到高地址，段依次为：代码段、只读数据段、数据段、bss 段、注释段（注意 bss 段和注释段不包含在 elf/bin 文件中）

反汇编文件：relocate.dis

```
relocate.elf:     file format elf32-littlearm


Disassembly of section .text: //代码段

80100000 <_start>:
80100000:    e59fd028     ldr sp, [pc, #40]    ; 80100030 <clean+0x14>
80100004:    eb000001     bl  80100010 <clean_bss>
80100008:    fb000070     blx 801001d2 <main>

……(省略)

Disassembly of section .rodata:   //只读数据段

8010086c <g_charB>:
8010086c:    00000042     andeq    r0, r0, r2, asr #32

……(省略)

Disassembly of section .data: //数据段

8010098c <g_charA>:
8010098c:    00000041     andeq    r0, r0, r1, asr #32

80100990 <hex_tab>:
80100990:    33323130     teqcc    r2, #48, 2
```

```
80100994:    37363534             ; <UNDEFINED> instruction: 0x37363534
80100998:    62613938     rsbvs   r3, r1, #56, 18  ; 0xe0000
8010099c:    66656463     strbtvs r6, [r5], -r3, ror #8

Disassembly of section .bss:  //bss 段，不保存在.bin 文件中

801009a0 <__bss_start>:
801009a0:    00000000     andeq   r0, r0, r0

801009a4 <IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA>:
801009a4:    00000000     andeq   r0, r0, r0

801009a8 <g_intA>:
801009a8:    00000000     andeq   r0, r0, r0

801009ac <g_intB>:
801009ac:    00000000     andeq   r0, r0, r0

801009b0 <g_charC>:
    ...

……(省略)

Disassembly of section .comment:      //comment 段，不保存在.bin 文件中

……(省略)
```

## 9.2 链接脚本解析

顾名思义，链接脚本控制程序的链接过程，它规定如何把输入文件内的段放入输出文件，并控制输出文件内的各部分在程序地址空间内的布局。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/9_重定位/02_clean_bss”目录下。

为了在链接时使用链接脚本，需要在 Makefile 用-T filename.lds 指定。否则在编译时将使用默认的链接脚本。

```
#使用链接脚本 imx6ull.lds
$(LD) -T imx6ull.lds -g start.o uart.o main.o my_printf.o -o relocate.elf -lgcc -
L/home/book/100ask_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-
gnueabihf/lib/gcc/arm-linux-gnueabihf/6.2.1
```

需要注意，对于结构较为简单的程序，也可以使用默认的链接脚本，并手动指定不同段在输出文件中的位置。

```
#将所有程序的.text 段放在一起，起始地址设置为 0x80100000
#将所有程序的.data 段放在一起，起始地址设置为 0x80102000
$(LD) -Ttext 0x80100000 -Tdata 0x80102000 -g start.o uart.o main.o my_printf.o -o
relocate.elf -lgcc -L/home/book/100ask_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-
2016.11-x86_64_arm-linux-gnueabihf/lib/gcc/arm-linux-gnueabihf/6.2.1
```

默认的链接脚本无法进行一些段的复杂操作，所以下面的程序中我们一律使用链接脚本。

### 9.2.1 链接脚本语法

本章节中所有的知识都来源于 GNU 官方文档：
http://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_mono/ld.html

链接脚本的结构为

```
SECTIONS {
...
secname start BLOCK(align) (NOLOAD) : AT ( ldadr )
  { contents } >region :phdr =fill
...
}
```

- secname：段的名称
- start：段的运行地址（runtime addr），也称为重定位地址（relocation addr）
- AT ( ldadr )：ldadr 是段的加载地址（load addr）；AT 是链接脚本函数，用于将该段的加载地址设定为 ldadr；如果不添加这个选项，默认的加载地址等于运行地址。
- 其他的链接脚本函数我们之后用到了再讲，想进一步了解可以参考上面的官方文档
- { contents }：{ }用来表示段的起始结束；content 为该段包含的内容，可以由用户自己指定。
- BLOCK(align) (NOLOAD)，>region :phdr =fill：很少用到不深入讲解

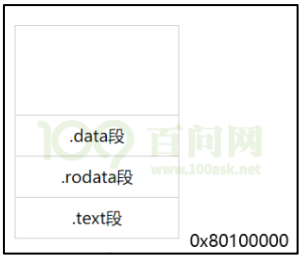
依照上述的结构我们来分析本章节中前面用到的链接脚本 imx6ull.lds。

### 9.2.2 解析链接脚本

链接脚本：imx6ull.lds

```
01 SECTIONS {
02     . = 0x80100000;             //设定链接地址为 0x80100000
03
04     . = ALIGN(4);           //将当前地址以 4 字节为标准对齐
05     .text      :            //创建段，其名称为 .text
06     {                       //.text 包含的内容为所有链接文件的数据段
07       *(.text)              // *：表示所有文件
08     }
09
10     . = ALIGN(4);           //将当前地址以 4 字节为标准对齐
11     .rodata : { *(.rodata) }        //.rodata 存放在.text 之后,包含所有链接文件的只读
数据段
12
13     . = ALIGN(4);
14     .data : { *(.data) }            //.data 存放在.rodata 之后，包含所有链接文件的只
读数据段
15
16     . = ALIGN(4);
17     __bss_start = .;            //将当前地址的值存储为变量__bss_start
18     .bss : { *(.bss) *(.COMMON) }       //.bss 存放在.data 段之后，包含所有文件的 bss
段和注释段
19     __bss_end = .;          //将当前地址的值存储为变量__bss_end
20 }
```

根据上述链接脚本的配置，.bin 文件中的数据结构如下图所示：




上面我们写的链接脚本称为一体式链接脚本，与之相对的是分体式链接脚本，区别在于代码段(.text)和数据段(.data)的存放位置是否是分开的。

例如现在的一体式链接脚本的代码段后面依次就是只读数据段、数据段、bss 段，都是连续在一起的。 分体式链接脚本则是代码段、只读数据段，中间间隔很远之后才是数据段、bss 段。

分体式链接脚本实例：

```
SECTIONS {
    . = 0x80100000;             //设置链接地址为 0x80100000，这也是.text 段的起始地址

    . = ALIGN(4);
    .text      :
    {
      *(.text)
    }

    . = ALIGN(4);
    .rodata : { *(.rodata) }            //假设 rodata 段的结束地址为 0x8010xxxx

    . = ALIGN(4);
.data 0x80800000 : { *(.data) }         //指定 data 段的起始地址为 0x80200000,和 rodata 段之间有较大间隔

……（省略）
```

之后的代码更多的采用一体式链接脚本，原因如下：

1. 分体式链接脚本适合单片机，因为单片机自带有 flash，不需要将代码复制到内存占用空间。而我们的嵌入式系统内存非常大，没必要节省这点空间，并且有些嵌入式系统没有可以直接运行代码的 Flash，就需要从存储设备如 Nand Flash 或者 SD 卡复制整个代码到内存；
2. JTAG 等调试器一般只支持一体式链接脚本；

## 9.2.3 清除 bss 段

之前提到过 bin 文件中并不会保存 bss 段的值，因为这些值都是 0，保存这些值没有意义并会使得 bin 文件臃肿。

当程序运行涉及到 bss 段上的数据时，CPU 会从 bss 段对应的内存地址去读取对应的值，为了确保从这段内存地址上读取到的 bss 段数值为 0，在程序运行前需要将这一段内存地址上的数据清零，即清除 bss 段。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/9_重定位/02_clean_bss”目录下。

- **步骤 1：修改汇编文件**

我们在汇编文件中实现清除 bss 段，具体思路就是将 bss 段对应的地址读

取，并将地址上的数据依次清零。

汇编文件：start.S

```
01
02 .text
03 .global  _start
04
05 _start:
06
07      /* 设置栈 */
08      ldr  sp,=0x80200000
09
10      /* 清除 bss 段 */
11      bl clean_bss
12
13      /* 跳转到主函数 */
14      bl main
15
16 halt:
17      b  halt
18
19 clean_bss:
20      ldr r1, =__bss_start        //将链接脚本变量__bss_start 变量保存于 r1
21      ldr r2, =__bss_end          //将链接脚本变量__bss_end 变量保存于 r2
22      mov r3, #0
23 clean:
24      strb r3, [r1]           //将当前地址下的数据清零
25      add r1, r1, #1          //将 r1 内存储的地址+1
26      cmp r1, r2              //相等：清零操作结束；否则继续执行 clean 函数清零 bss 段
27      bne clean
28
29      mov pc, lr
```

- **步骤 2：在主函数汇中添加测试代码**

主函数中打印存放在 bss 段内数据的值。

程序文件：main.c

```
37 int main (void)
38 {
39      Uart_Init();    //初始化 uart 串口
40
41      printf("g_intA = 0x%08x\n\r", g_intA);  //打印 g_intA 的值
42      printf("g_intB = 0x%08x\n\r", g_intB);  //打印 g_intB 的值
43
44      return 0;
45 }
```

- **步骤 3：参考章节《4.3.4 编译程序》编译程序**

- **步骤 4：参考章节《3.4 映像文件烧写、运行》烧写、运行程序**

最终在终端中输出结果如下，保存在 bss 段中的变量 g_intA， g_intB 的值都为 0，表明清除 bss 段成功。

```
g_intA = 0x00000000
g_intB = 0x00000000
```

# 9.3 重定位的引入

## 9.3.1 什么是重定位

接触过 S3C2440 的朋友应该很熟悉，在程序运行之前我们需要手动将.bin 文件上的全部代码从 Nor Flash 或 Nand Flash 拷贝到 SDRAM 上。对于 imx6ull 来说，这部分拷贝代码的操作由 Boot Rom 自动完成，板子上电后 Boot Rom 会将映像文件从启动设备(TF 卡、eMMC)自动拷贝到 DDR3 内存上。上述拷贝代码的过程就是重定位。

那么 Boot Rom 应该将映像文件拷贝到内存的哪个位置呢？这部分内容已经在章节《3-1.2 IMX6ULL 启动流程》中详细讨论过了。简而言之 100ask_imx6ull 的映像文件包含多个部分，其中.bin 文件的起始地址由地址 entry 决定，需要在 Makefile 中手动配置。

```
./tools/mkimage -n ./tools/imximage.cfg.cfgtmp -T imximage -e 0x80100000 -d
relocate.bin relocate.imx
```

按照上述的配置，整个映像文件被自动重定位到 DDR3 内存上，其中.bin 文件的起始地址为 0x80100000。重定位结束后，CPU 会从这个地址读取第一条指令开始执行程序。

## 9.3.2 汇编重定位 data 段

下面我们将通过一个实例来说明为什么要重定位 data 段以及如何通过汇编重定位 data 段。

在 002_clean_bss 代码的基础上，在主函数中添加测试代码，不断地打印 data 段中的数据 g_charA。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/9_重定位/003_without_relocation”目录下。

程序文件：main.c

```
37 int main (void)
38 {
39     Uart_Init();    //初始化 uart 串口
40
41     printf("\n\r");
42     /* 在串口上输出 g_charA */
43     while (1)
44     {
45             PutChar(g_charA);
46             g_charA++;
47             delay(1000000);
48     }
49
50     return 0;
51 }
```

- **步骤 1：参考章节《4.3.4 编译程序》编译程序**
- **步骤 2：参考章节《3.4 映像文件烧写、运行》烧写、运行程序**

最终在终端上成功打印字符 g_charA 的值。

在程序运行时，CPU 需要不断地访问 DDR3 内存来获取 g_charA 的值，访问 DDR3 会花费大量的时间，那么如何提升访问的效率呢？

答：在程序运行先前将 data 段的数据重定位到 imx6ull 的片内 RAM 上，因为 CPU 访问片内 RAM 的速度远快于访问 DDR3 的速度。

下面我们将通过汇编重定位 data 段。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/9_重定位/004_manual_relocate_data”目录下。

➢ **步骤 1：参考芯片手册确定片内 RAM 的位置**

● 参考资料：芯片手册《Chapter 2: Memory Maps》

● 参考芯片手册得到片内 RAM 的地址为：0x900000 ~ 0x91FFFF。所以我们将.data 段重定位后的地址设置为 0x900000。

**Table 2-1. System memory map**

| Start address | End address | Size | DDR3内存地址 Description | |
|---|---|---|---|---|
| 8000_0000 | FFFF_FFFF | 2048 MB | MMDC—x16 DDR Controller. | 100ask_im6ull所使用的DDR3内存大小为512MB 对应的地址为：0x80000000 - 0xA0000000 |
| 7000_0000 | 7FFF_FFFF | 256 MB | Reserved | |
| 6000_0000 | 6FFF_FFFF | 256 MB | QSPI1 Memory | |
| 5800_0000 | 5FFF_FFFF | 128 MB | EIM Aliased | |
| 5000_0000 | 57FF_FFFF | 128 MB | EIM (NOR/SRAM) | |
| 0098_0000 | 009B_FFFF | 256 KB | Reserved | |
| 0092_0000 | 0097_FFFF | 384 KB | OCRAM aliased | |
| 0090_0000 | 0091_FFFF | 128 KB | OCRAM 128 KB | 128KB的片内RAM |
| 008F_8000 | 008F_FFFF | 32 KB | Reserved | |
| 007F_8000 | 008F_7FFF | 1 MB | Reserved | |
| 0010_0000 | 0010_7FFF | 32 KB | Reserved | |
| 0001_8000 | 000F_FFFF | 928 KB | Reserved | |
| 0001_7000 | 0001_7FFF | 4 KB | Boot ROM—Protected 4 KB area | |
| 0000_0000 | 0001_6FFF | 92 KB | Boot ROM (ROMCP) | |

➢ **步骤 2：修改链接脚本**

创建一个变量用来存储.data 段的起始加载地址。

```
. = ALIGN(4);
.rodata : { *(.rodata) }

. = ALIGN(4);

data_load_addr = .;              //将当前地址存储在变量中（大概的值为 0x8010xxxx）
```

将.data 段的运行地址(runtime address)设定为 0x900000。加载地址由变量 data_load_addr 确定。这样设置后，在.bin 文件中“.data”段仍旧存储在“.rodata”段之后。但在程序运行时，CPU 会从 0x900000 开始的空间内读取“.data 段”的值。

```
.data 0x900000 : AT(data_load_addr)
```

下面我们将重定位后.data 段的起始地址存储在变量 data_start，重定位后的.data 段的结束地址存储在变量 data_end，这两个变量将供汇编文件调用。

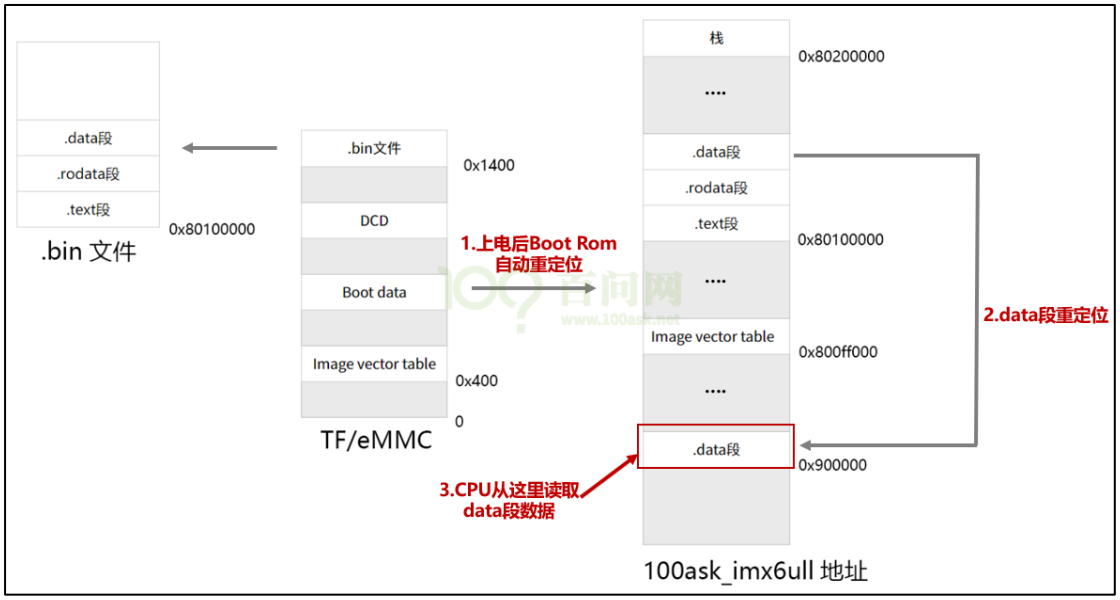
```
{
  data_start = . ;        //addr = 0x900000
  *(.data)
  data_end = . ;      //addr = 0x900000+SIZEOF(.data)
}
```

修改后的链接脚本如下所示。

链接脚本 imx6ull.lds

```
SECTIONS {
    . = 0x80100000;

    . = ALIGN(4);
    .text      :
    {
      *(.text)
    }

    . = ALIGN(4);
    .rodata : { *(.rodata) }

    . = ALIGN(4);

    data_load_addr = .;
    .data 0x900000 : AT(data_load_addr)
    {
      data_start = . ;
      *(.data)
      data_end = . ;
    }

    . = ALIGN(4);
    __bss_start = .;
    .bss : { *(.bss) *(.COMMON) }
    __bss_end = .;
}
```

通过上述操作，CPU 虽然会去片内 RAM 中读取.data 段数据，但实际上片内 RAM 并没有准备好.data 段的数据，如下图所示。下面我们将通过汇编将 DDR3 内存上的.data 段数据重定位到片内 RAM 上。




➢ **步骤 3：修改汇编文件重定位.data 段**

设置完栈后直接跳转到 copy_data 函数重定位 data 段。

汇编文件：start.S

```
    /* 设置栈 */
    ldr  sp,=0x80200000
```

```
    /* 重定位 data 段 */
    bl copy_data

    /* 清除 bss 段 */
    bl clean_bss
```

实现 copy_data 函数
汇编文件：start.S

```
copy_data:
    /* 重定位 data 段 */
    ldr r1, =data_load_addr /* data 段的加载地址，从链接脚本中得到，0x8010xxxx */
    ldr r2, =data_start     /* data 段重定位地址，从链接脚本中得到，0x900000 */
    ldr r3, =data_end       /* data 段结束地址，从链接脚本中得到,0x90xxxx */
cpy:
    ldr r4, [r1]        /* 从 r1 读到 r4 */
    str r4, [r2]        /* r4 存放到 r2 */
    add r1, r1, #4      /* r1+1 */
    add r2, r2, #4      /* r2+1 */
    cmp r2, r3      /* r2 r3 比较 */
    bne cpy         /* 如果不等则继续拷贝 */

    mov pc, lr      /* 跳转回调用 copy_data 函数之前的地址 */
```

- **步骤 3：参考章节《4.3.4 编译程序》编译程序**
- **步骤 4：参考章节《3.4 映像文件烧写、运行》烧写、运行程序**
  分别烧写这 2 个程序，对比一下效果：

● 代码 1：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/9_重定位/003_without_relocation”目录下。

● 代码 2：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/9_重定位/004_manual_relocate_data”目录下。

可以发现把 data 段重定位到片内 RAM 后，程序在终端上打印字符的速度明显变快。

## 9.4 C 函数重定位 data 段和清除 bss 段

到目前为止我们已经通过汇编实现了重定位 data 段和清除 bss 段。为了让汇编程序更加简洁，这一节中我们将通过 C 语言实现重定位 data 段和清除 bss 段。

### 9.4.1 通过汇编传递链接脚本变量

这一小节中我们将通过汇编文件获得链接脚本中的变量，再将这些变量传递给 C 函数。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/9_重定位/005_relocate_data_with_c”目录下。

- **步骤 1：修改汇编文件**

打开 start.S 将之前的汇编函数 copy_data，clean_bss 删除，改为直接调用 C 函数。在调用对应的 C 函数之前，需要通过寄存器 r0~r4 将 C 函数的参数准备好。

汇编文件：start.S

```
.text
.global  _start

_start:

    /* 设置栈 */
    ldr  sp,=0x80200000

    /* 重定位 data 段 */
    ldr r0, =data_load_addr /* data 段的加载地址 (0x8010....) */
    ldr r1, =data_start     /* data 段重定位地址，0x900000 */
    ldr r2, =data_end       /* data 段结束地址(重定位后地址 0x90....) */
    sub r2, r2, r1      /* r2 的值为 data 段的长度 */

    bl copy_data        /* 跳转到函数 copy_data 并将 r1,r2,r3 作为函数参数传入 */

    /* 清除 bss 段 */
    ldr r0, =__bss_start
    ldr r1, =__bss_end

    bl clean_bss        /* 跳转到函数 clean_bss 并将 r0，r1 作为函数参数传入*/

    /* 跳转到主函数 */
    bl main

halt:
    b  halt
```

➢ 步骤 2：创建程序文件 init.c 实现 copy_data，clean_bss 函数

程序文件：init.c

```
/* 从汇编得到参数 src，dest，len 的值 */
void copy_data (volatile unsigned int *src, volatile unsigned int *dest, unsigned
int len)
{
    unsigned int i = 0;

    while (i < len)
    {
        *dest++ = *src++;
        i += 4;
    }
}

/* 从汇编得到参数 start，end 的值 */
void clean_bss (volatile unsigned int *start, volatile unsigned int *end)
{
    while (start <= end)
    {
        *start++ = 0;
    }
}
```

需要注意的是，上述两个函数的参数都是从汇编文件传入

- 对于 copy_data 函数来说，参数 src，dest，len 分别对应汇编文件中 r0，r1，r2 的值

- 对于 clean_bss 函数来说，参数 start，end 分别对应汇编文件中 r0，r1 的值

➢ **步骤 3：修改 Makefile**

修改 Makefile 文件，编译 init.c 并链接 init.o。

文件：Makefile

```
08 relocate.img : start.S  uart.c main.c my_printf.c init.c
09      $(CC) -nostdlib -g -c -o start.o start.S
10      $(CC) -nostdlib -g -c -o uart.o uart.c
11      $(CC) -nostdlib -g -c -o main.o main.c
12      $(CC) -nostdlib -g -c -o my_printf.o my_printf.c
13      $(CC) -nostdlib -g -c -o init.o init.c
14
15      $(LD) -T imx6ull.lds -g start.o uart.o main.o my_printf.o init.o -o
relocate.elf -lgcc -L/home/book/100ask_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-
2016.11-x86_64_arm-linux-gnueabihf/lib/gcc/arm-linux-gnueabihf/6.2.1
```

➢ **步骤 3：参考章节《4.3.4 编译程序》编译程序**

➢ **步骤 4：参考章节《3.4 映像文件烧写、运行》烧写、运行程序**

程序成功运行，在终端成功输出字符串.

## 9.4.2 C 函数直接调取链接脚本变量

上一节中 C 函数需要通过汇编文件传入参数，在这一节我们将进一步改进 C 函数，使得 C 函数跳过汇编文件，直接从链接脚本中调用所需变量。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/9_重定位/006_relocate_data_with_c_modified”目录下。

➢ **步骤 1：修改汇编文件，改为直接调用 C 函数**

汇编文件：start.S

```
.text
.global  _start

_start:

    /* 设置栈 */
    ldr  sp,=0x80200000

    /* 重定位 data 段 */
    bl copy_data

    /* 清除 bss 段 */
    bl clean_bss

    /* 跳转到主函数 */
    bl main

halt:
    b  halt
```

➢ **步骤 2：修改 init.c 通过函数来获取参数**

程序文件：init.c

```
12 void copy_data (void)
```

```
13 {
14     /* 从链接脚本中直接获得参数 data_load_addr, data_start, data_end */
15     extern int data_load_addr, data_start, data_end;
16
17     volatile unsigned int *dest = (volatile unsigned int *)&data_start;
18     volatile unsigned int *end = (volatile unsigned int *)&data_end;
19     volatile unsigned int *src = (volatile unsigned int *)&data_load_addr;
20
21     /* 重定位数据 */
22     while (dest < end)
23     {
24             *dest++ = *src++;
25     }
26 }

38 void clean_bss(void)
39 {
40     /* 从 lds 文件中获得 __bss_start, __bss_end */
41     extern int __bss_end, __bss_start;
42
43     volatile unsigned int *start = (volatile unsigned int *)&__bss_start;
44     volatile unsigned int *end = (volatile unsigned int *)&__bss_end;
45
46     while (start <= end)
47     {
48             *start++ = 0;
49     }
50 }
```

- **步骤 3：参考章节《4.3.4 编译程序》编译程序**
- **步骤 4：参考章节《3.4 映像文件烧写、运行》烧写、运行程序**

程序成功运行，在终端成功输出字符串。

## 9.4.3 总结:如何在 C 函数中使用链接脚本变量

结合上面的例子，我们来总结一下如何在 C 函数中使用链接脚本中定义的变量

a) 在 C 函数中声明该变量为外部变量，用 extern 修饰，例如： extern int _start;
b) 使用取址符号(&)得到该变量的值，例如：int *p = &_start;//p 的值为 lds 文件中_start 的值

为什么在汇编文件中可以直接使用链接脚本中的变量，而在 C 函数中需要加上取址符号呢？

原因：C 函数中定义一个全局变量 int g_i = 10;，程序中必然有 4 字节的空间留出来给这个变量 g_i，然而链接脚本中的变量并不像全局变量一样都保存在.bin 文件中。如果我们在 C 程序中只用到链接脚本变量 a1, a2, a3，那么程序中并不保存这 3 个变量。

但是这些变量的符号，都会保存在 symbol_table 符号表中，如下图所示：


从上图中我们注意到：

- 对于全局变量，symbol table 里面存储的是变量的地址；可以通过&g_i 得到变量的地址 addr。

- 对于链接脚本变量，symbol table 里面存储的是变量的值；为了取出这个值，C 代码要通过&a1。

# 9.5 重定位全部代码

## 9.5.1 C 函数实现重定位全部代码

对于 imx6ull，它的 boot ROM 功能强大，会帮我们把程序重定位到 DDR3 内存上。但对于一些采用其他芯片的板子，这一部分的操作可能需要我们手动去完成。例如 S3C2440 上电后，因为硬件的限制，.bin 文件的前 4k 程序需要将整个程序重定位到大小能够执行整个程序的 SDRAM 上。

为了练习代码重定位所需知识，在这一节中我们将重定位整个.bin 文件到片内 RAM 上。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/9_重定位/007_relocate_all_with_c”目录下。

需要注意，虽然将全部代码重定位到片内 RAM 上可以加快命令的执行、数据的读取写入，但是这样的做法并不适合体积较大的程序，因为片内 RAM 只有 128KB 空间。

➢ **步骤 1：修改链接脚本**

① 修改链接地址为 0x900000

② 删除与.data 段相关的链接脚本变量。

③ 添加变量_load_addr 并将它的值设置为 Makefile 中 entry 地址的值，供 C 函数调用。

链接脚本：imx6ull.lds

```
01 SECTIONS {
02     _load_addr = 0x80100000;
03
```

```
04     . = 0x900000;
05
06     . = ALIGN(4);
07     .text      :
08     {
09       *(.text)
10     }
11
12     . = ALIGN(4);
13     .rodata : { *(.rodata) }
14
15     . = ALIGN(4);
16     .data : { *(.data) }
17
18     . = ALIGN(4);
19     __bss_start = .;
20     .bss : { *(.bss) *(.COMMON) }
21     __bss_end = .;
22 }
```

- **步骤 2：修改 init.c**

重定位全部代码和重定位.data 段原理相同。在这里只需要修改 copy_data 函数中调用的外部变量。

程序文件：init.c

```
12 void copy_data (void)
13 {
14     /* 从链接脚本中获得参数 _start, __bss_start, */
15     extern int _load_addr, _start, __bss_start;
16
17     volatile unsigned int *dest = (volatile unsigned int *)&_start;
//_start = 0x900000
18     volatile unsigned int *end = (volatile unsigned int *)&__bss_start;
//__bss_start = 0x9xxxxx
19     volatile unsigned int *src = (volatile unsigned int *)&_load_addr;
//_load_addr = 0x80100000
20
21     /* 重定位数据 */
22     while (dest < end)
23     {
24             *dest++ = *src++;
25     }
26 }
```

- **步骤 3：修改汇编文件**

重定位之后，需要使用绝对跳转命令 ldr pc, = xxx，跳转到重定位后的地址。

汇编文件：start.S

```
16     /* 跳转到主函数 */
17     // bl main            /* 相对跳转，程序仍在 DDR3 内存中执行 */
18     ldr pc, =main     /* 绝对跳转，程序在片内 RAM 中执行 */
```

- **步骤 3：参考章节《4.3.4 编译程序》编译程序**

- **步骤 4：参考章节《3.4 映像文件烧写、运行》烧写、运行程序**

程序成功运行，在终端成功输出字符串。

## 9.5.2 位置无关码

查看上述程序的反汇编发现，在重定位函数 copy_data 执行之前，已经涉及到了片内 RAM 上的地址，但此时片内 RAM 上并没有任何程序，那为什么程序还能正常运行呢？

反汇编文件：relocate.dis

```
07 00900000 <_start>:
08   900000:    e59fd00c  ldr      sp, [pc, #12]   ; 900014 <halt+0x4>
09   900004:    fa00016f   blx     9005c8 <copy_data>
10   900008:    fb000180  blx      900612 <clean_bss>
11   90000c:    e59ff004  ldr      pc, [pc, #4]    ; 900018 <halt+0x8>
12
13 00900010 <halt>:
14   900010: eafffffe     b    900010 <halt>
15   900014: 80200000     eorhi    r0, r0, r0
16   900018: 009001b3             ; <UNDEFINED> instruction: 0x009001b3
……
009001b2 <main>:
```

dis 文件中左边的 90000xx 是链接地址，表示程序运行“应该位于这里”。但是实际上，我们一上电，boot ROM 把程序放到 0x80100000 去了。所以一开始运行这些指令时，它们是位于 DDR 里的。

第 9 行的 blx 命令，并不是跳到 0x9005c8。这要根据当前的 PC 值来计算，在 dis 里写成 9005c8，这只是表示“如果程序从 0x900000 开始运行的话，第 9 行就会跳到 0x9005c8”。现在程序被 boot ROM 复制到 0x80100000，从 0x80100000 开始运行，我们需要根据机器码来计算出实际跳转的地址。

blx 是相对跳转指令，要跳到“pc + offset”这个地址去。程序从 0x8010000 运行，运行到第 9 行时，如下计算新地址：

```
PC=当前地址+8=0x8010004+8=0x801000C
offset=机器码“fa00016f”里的 bit[23:0]*4=0x16f*4=0x5BC
新 PC=PC + offset = 0x80105C8
```

在 0x80105C8 这个位置，确实存有 copy_data 函数，所以：即使程序并不在链接地址 0x900000 上，它也可以运行。因为 blx 是相对跳转指令，它用的不是链接地址，它是“位置无关”的。使用“位置无关码”写出的代码，它可以在任何位置上运行，不一定要在“链接地址”上运行。

下面我们来分析一下实际板子上电后，程序是如何执行的

**1．程序被 boot ROM 重定位到 0x80100000，并从这个地址开始执行第一条指令，此时 pc = 0x80100000 + 8 = 0x80100008。**

**2．执行到第 2 条指令“fa00016f”时，根据上述算法，它跳到地址 0x80105C8 去执行 copy_data 函数**

**3．在执行完 copy_data 和 clean_bss 函数后，片内 RAM 0x900000 上已经有程序了。**

**4．执行绝对跳转命令“ldr pc, =main”，它是一条伪指令，真实指令是“ldr pc, [pc, #4] ; 900018 <halt+0x8>”：**

从 dis 文件里很容易看出，执行完这条指令后，pc 等于 dis 文件中“900018”上的值“009001b3”，所以程序跳到片内 RAM 去执行 main 函数了。

注意：在 dis 文件中，main 函数的链接地址是 0x009001b2，往 pc 寄存器里赋值 0x009001b3 时，bit0 为 1，表示 main 函数的代码是用 Thumb 指令写的。

那么我们应该如何写位置无关码呢？

答：使用相对跳转命令 b 或 bl，并注意

- 重定位之前，不可使用绝对地址

a) 不可访问全局类变量（全局变量或 static 修饰的局部变量）

b) 不可访问有初始值的数组（初始值放在 rodata 里，需要绝对地址来访问）

- 重定位之后，使用 ldr pc = xxx，跳转到绝对地址（runtime address）

# 第10章 异常与中断

参考资料：网盘开发板配套资料“08_Reference material (ARM,NXP 参考资料)/Arm 架构参考资料.zip”里：

- armv7 ar 架构参考手册 学习 CPU 架构、内存及系统架构（DDI0406C_d_armv7ar_arm）.pdf
- ARM® Generic Interrupt Controller Architecture Specification Architecture version 2.0

## 10.1 异常与中断的引入

### 10.1.1 妈妈怎么知道孩子醒了

妈妈怎么知道卧室里小孩醒了？

① 时时进房间看一下： 简单，但是累

② 进去房间陪小孩一起睡觉，小孩醒了会吵醒她：不累，但是妈妈干不了活了

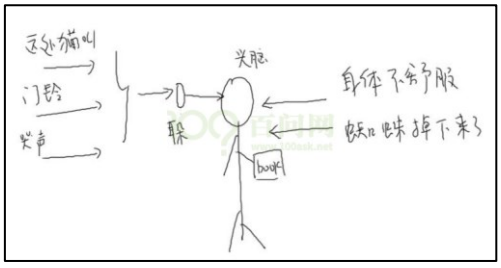
③ 妈妈要干很多活，但是可以陪小孩睡一会，定个闹钟：要浪费点时间，但是可以继续干活。妈妈要么是被小孩吵醒，要么是被闹钟吵醒。

④ 妈妈在客厅干活，小孩醒了他会自己走出房门告诉妈妈：妈妈、小孩互不耽误。

后面的 3 种方式，都需要“小孩来中断妈妈”：中断她的睡眠、中断她的工作。

实际上，能“中断”妈妈的事情可多了：

- 远处的猫叫：这可以被忽略
- 门铃、小孩哭声：妈妈的应对措施不一样
- 身体不舒服：那要赶紧休息
- 有蜘蛛掉下来了：赶紧跑啊，救命



妈妈当前正在看书，被“中断”后她会怎么做？流程如下：

- 妈妈正在看书
- 发生了各种声音
  - 可忽略的远处猫叫
  - 快递员按门铃
  - 卧室中小孩哭了
- 妈妈怎么办？

a) 先在书中放入书签，合上书(保存现场)

b) 去处理

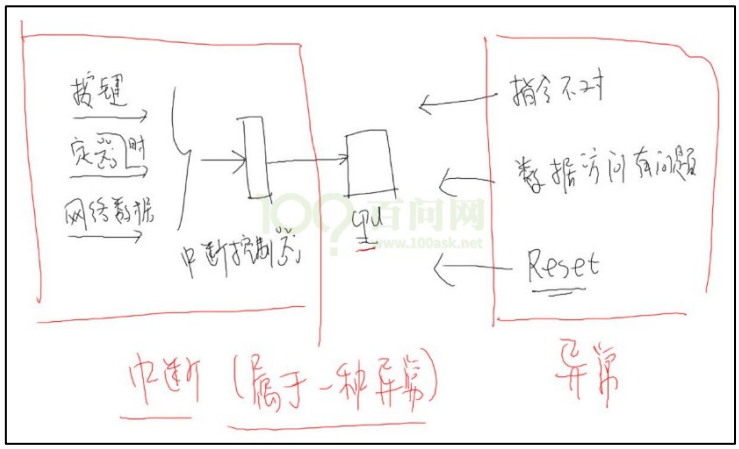
① 对于不同的情况，处理方法不同：

② 对于门铃：开门取快递

③ 对于哭声：照顾小孩

c) 回来继续看书(恢复现场)

## 10.1.2 嵌入系统中也有类似的情况




CPU 在运行的过程中，也会被各种“异常”打断。这些“异常”有：

① 指令未定义

② 指令、数据访问有问题

③ SWI(软中断)

④ 快中断

⑤ 中断

中断也属于一种“异常”，导致中断发生的情况有很多，比如：

① 按键

② 定时器

③ ADC 转换完成

④ UART 发送完数据、收到数据

⑤ 等等

这些众多的“中断源”，汇集到“中断控制器”，由“中断控制器”选择优先级最

高的中断并通知 CPU。

## 10.2 异常与中断的处理流程

arm 对异常(中断)处理过程：

① 初始化：

a) 设置中断源，让它可以产生中断

b) 设置中断控制器(可以屏蔽某个中断，优先级)

c) 设置 CPU 总开关(使能中断)

② 执行其他程序：正常程序

③ 产生中断：比如按下按键--->中断控制器--->CPU

④ CPU 每执行完一条指令都会检查有无中断/异常产生

⑤ CPU 发现有中断/异常产生，开始处理。

对于不同的异常，跳去不同的地址执行程序。

这地址上，只是一条跳转指令，跳去执行某个函数(地址)，这个就是异常向量。③④⑤都是硬件做的。⑥ 这些函数做什么事情？软件做的：

a) 保存现场(各种寄存器)

b) 处理异常(中断):分辨中断源，再调用不同的处理函数

c) 恢复现场

## 10.3 怎么保存现场：栈

中断中断，中断谁？中断当前正在运行的进程、线程。

进程、线程是什么？内核如何切换进程、线程、中断？

要理解这些概念，必须理解栈的作用。

### 10.3.1 ARM 处理器程序运行的过程

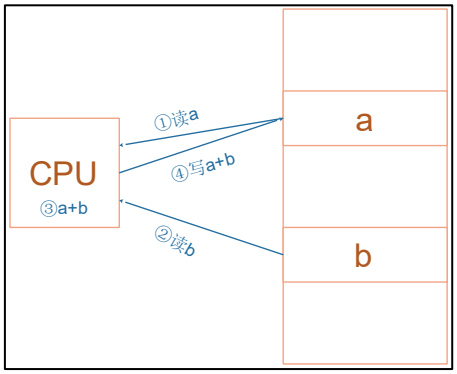
ARM 芯片属于精简指令集计算机(RISC：Reduced Instruction Set Computing)，它所用的指令比较简单，有如下特点：

① 对内存只有读、写指令

② 对于数据的运算是在 CPU 内部实现

③ 使用 RISC 指令的 CPU 复杂度小一点，易于设计

比如对于 a=a+b 这样的算式，需要经过下面 4 个步骤才可以实现：



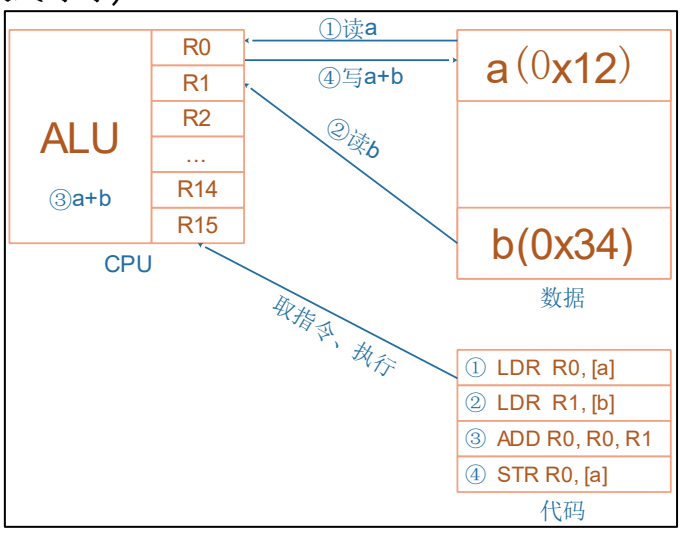
细看这几个步骤，有些疑问：

① 读 a，那么 a 的值读出来后保存在 CPU 里面哪里？

② 读 b，那么 b 的值读出来后保存在 CPU 里面哪里？

③ a+b 的结果又保存在哪里？

我们需要深入 ARM 处理器的内部。简单概括如下，我们先忽略各种 CPU 模式(系统模式、用户模式等等)。




CPU 运行时，先去取得指令，再执行指令：

① 把内存 a 的值读入 CPU 寄存器 R0

② 把内存 b 的值读入 CPU 寄存器 R1

③ 把 R0、R1 累加，存入 R0

④ 把 R0 的值写入内存 a

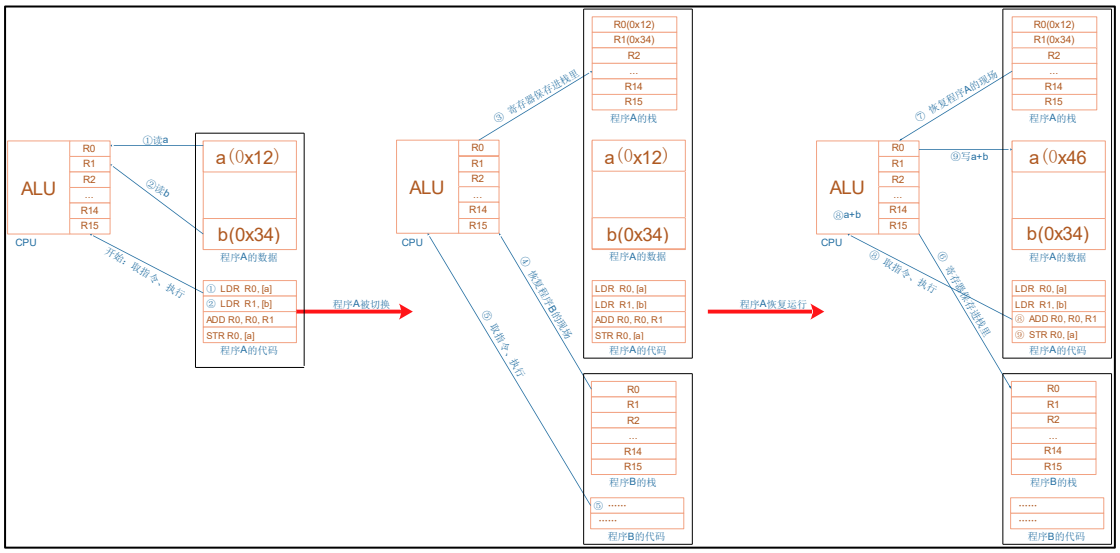
## 10.3.2 程序被中断时，怎么保存现场

从上图可知，CPU 内部的寄存器很重要，如果要暂停一个程序，中断一个程序，就需要把这些寄存器的值保存下来：这就称为保存现场。

保存在哪里？内存，这块内存就称之为栈。

程序要继续执行，就先从栈中恢复那些 CPU 内部寄存器的值。

这个场景并不局限于中断，下图可以概括程序 A、B 的切换过程，其他情况是类似的：



a） 函数调用：

在函数 A 里调用函数 B，实际就是中断函数 A 的执行。
那么需要把函数 A 调用 B 之前瞬间的 CPU 寄存器的值，保存到栈里；
再去执行函数 B；
函数 B 返回之后，就从栈中恢复函数 A 对应的 CPU 寄存器值，继续执行。

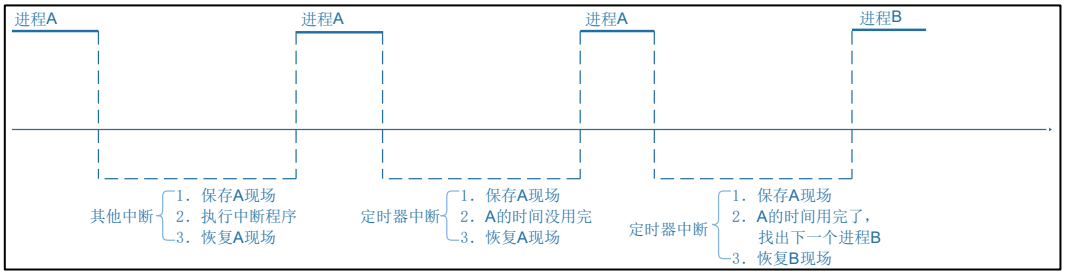
b） 中断处理

进程 A 正在执行，这时候发生了中断。
CPU 强制跳到中断异常向量地址去执行，
这时就需要保存进程 A 被中断瞬间的 CPU 寄存器值，
可以保存在进程 A 的内核态栈，也可以保存在进程 A 的内核结构体中。
中断处理完毕，要继续运行进程 A 之前，恢复这些值。

c） 进程切换

在所谓的多任务操作系统中，我们以为多个程序是同时运行的。如果我们能感知微秒、纳秒级的事件，可以发现操作系统时让这些程序依次执行一小段时间，进程 A 的时间用完了，就切换到进程 B。

怎么切换？切换过程是发生在内核态里的，跟中断的处理类似。
进程 A 的被切换瞬间的 CPU 寄存器值保存在某个地方；
恢复进程 B 之前保存的 CPU 寄存器值，这样就可以运行进程 B 了。
所以，在中断处理的过程中，伴存着进程的保存现场、恢复现场。
进程的调度也是使用栈来保存、恢复现场：




## 10.4 ARM 处理器模式和寄存器

ARM processor modes before ARMv6

| Mode | Function | Privilege |
|---|---|---|
| User (USR) | Mode in which most programs and applications run | Unprivileged |
| FIQ | Entered on an FIQ interrupt exception | Privileged |
| IRQ | Entered on an IRQ interrupt exception | |
| Supervisor (SVC) | Entered on reset or when a Supervisor Call instruction (SVC) is executed | |
| Abort (ABT) | Entered on a memory access exception | |
| Undef (UND) | Entered when an undefined instruction executed | |
| System (SYS) | Mode in which the OS runs, sharing the register view with User mode | |

ARM 体系结构是一种基于模式的体系结构。在引入安全扩展之前，它具有 7 种处理器模式，如上表所示。有六个特权模式和一个非特权用户模式。特权是执行用户（非特权）模式无法完成的某些任务的能力。在用户模式下，对影响整个系统配置的操作存在一些限制，例如，MMU 的配置和缓存操作。

模式与异常事件相关，可以这样简单理解：

① 板子上电时，CPU 处于 SVC 模式，它用的是 SVC 模式下的寄存器

② 程序运行时发生了中断，CPU 进入 IRQ 模式，它用的 IRQ 模式下的寄存器

③ CPU 处理完中断，它切换回 SVC 模式，继续使用 SVC 模式下的寄存器

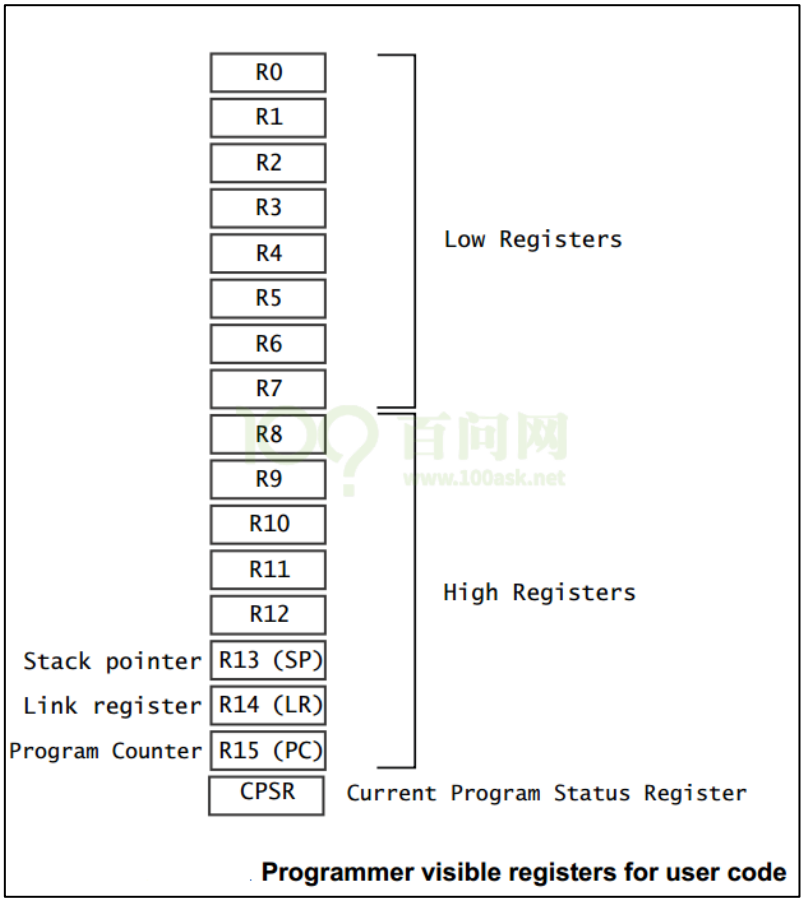
④ CPU 发生某种异常时，比如读取内存出错，它会进入 ABT 模式，使用 ABT 模式下的寄存器来处理错误。

在某种模式下，CPU 执行时使用的是这种模式的资源，比如使用的是这组模式的寄存器。这样就可以免去保存上一个模式所使用的寄存器。

当前的处理器模式和执行状态包含在当前程序状态寄存器（CPSR）中。可以通过异常或者在特权软件下显式地修改 CPSR 来进入到不同处理器模式。

引入安全扩展之后，情况比较复杂，我们先不介绍。我们也用不到安全扩展。

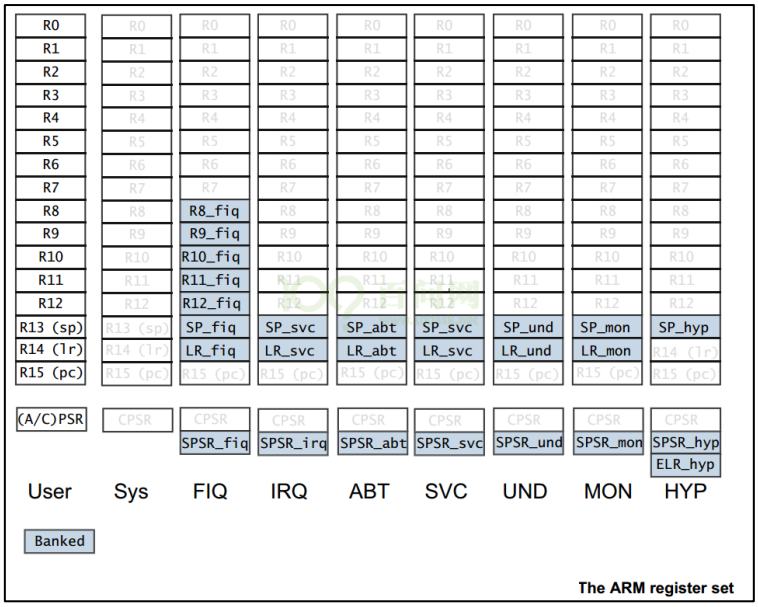
## 10.4.1 寄存器




ARM 体系结构提供了十六个 32 位通用寄存器（R0-R15）供软件使用。其中的 15 个（R0-R14）可用于通用数据存储，而 R15 是程序计数器，其值随处理器执行指令而改变。显式地写入 R15 可以更改程序流程。软件还可以访问 CPSR 和 SPSR。SPSR 中保存的上一个运行模式的 CPSR 的副本。

同一个寄存器可能在不同模式下对应物理上不同的位置。只有特定模式下才能访问到这些位置的寄存器。

有些寄存器，不同的工作模式下有自己的副本，当切换到另一个工作模式时，那个工作模式的寄存器副本将被使用，这些寄存器被称为备份寄存器。备份寄存器在物理上使用不同的存储，通常仅在特定模式下才可以访问它们。下图中带阴影标记的寄存器都是备份寄存器。




在所有模式下，“低位寄存器”和 R15 共享相同的物理存储位置。图 3-5 显示了某些模式下的某些“高位寄存器”被备份。例如，FIQ 模式下使用 R8-R12 备份寄存器，也就是说，FIQ 模式下对其的访问将转到另一个物理存储位置。对于除用户和系统模式以外的所有模式，R13 和 SPSR 都是备份寄存器。

对于备份寄存器，软件通常不会指定要访问哪个模式下的寄存器，这是由当前运行的模式隐含的。例如，在用户模式下使用 R13 时实际上将访问 R13_usr，而在 SVC 模式下将访问 R13_svc。

- R13（在所有模式下）是堆栈指针，但是当堆栈操作不需要时，它可以用作通用寄存器。

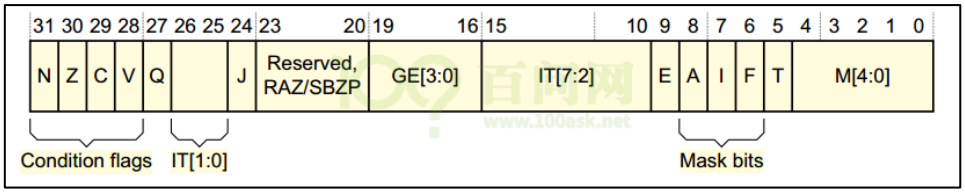
- R14（链接寄存器）保存 BL 分支指令的下一条指令的地址。当它不支持子程序的返回时，它也可以用作通用寄存器。R14_svc，R14_irq，R14_fiq，R14_abt 和 R14_und 同样用于在发生中断和异常时，或者执行转移和链接指令时，备份 R15 的返回值。

- R15 是程序计数器并保存当前程序地址（实际上，在 ARM 状态下，它始终指向当前指令之后的八个字节，而在 Thumb 状态下，它始终指向当前指令之后的四个字节，这是原始 ARM1 的三级流水线的遗留特性）。在 ARM 状态下读取 R15 时，位[1：0]为零，位[31：2]包含 PC 值。在 Thumb 状态下，位[0]始终读为零。

- R0-R14 的复位值是不定的。在使用堆栈之前，必须通过引导代码初始化 SP（堆栈指针）。因为每种模式下的 R13 即 SP 寄存器都有自己的实体，所以你用到哪种模式，就需要单独为该模式设置 SP。ARM 体系结构过程调用标准（AAPCS）或 ARM 嵌入式 ABI（AEABI）指定了软件应如何使用通用寄存器，以便在不同的工具链或编程语言之间进行互操作。

## 10.4.2 状态寄存器

程序状态寄存器（CPSR，current programmer status register）包含处理器的状态和一些控制标记位。




➢ **条件标记，bits[31:28]。**

根据指令的执行结果设置，这些标记位是：

- N，bit[31]负数标记位
- Z，bit[30]零标记位
- C，bit[29]进位标记位
- V，bit[28]溢出标记位

这些条件标记位可以在任何模式下读写。

- GE[3:0],bit[19:16]一些 SIMD 指令使用
- IT[7:2]，bit[15:10]Thumb2 指令集的 If-then 条件指令使用
- J，bit[24] 处理器是否处于 Jazelle 状态，和 T，bit[5]一起决定执行的指令集
- E，bit[9] 大小端状态位，0 表示小端，1 表示大端。
- Mask bits，bits[8:6]
- A，bit[8] 异步中止禁止位 Asynchronous abort mask bit.
- I，bit[7] IRQ 禁止位
- F，bit[6] FIQ 禁止位
- Q，bit[27]为 1 的话，表明执行一些指令时出现饱和或者溢出，一般与 DSP 有关。
- T，bit[5] Thumb 指令位，和 J，bit[24]位决定了处理器的指令集，ARM，Thumb，Jazzelle 或者 ThumbEE
- M[4:0]，bits[4:0]工作模式位，决定了处理器当前处于的工作模式。

处理器可以使用直接写入 CPSR 模式位来实现模式之间切换。更常见的是，处理器会由于异常事件而自动切换模式。在用户模式下，无法改变处理器模式的 PSR 位[4：0]来切换模式和 A，I 和 F 位来使能或者禁止异步中止、IRQ 和 FIQ。

## 10.4.3 协处理器 CP15

协处理器，顾名思义它也是一个处理器，不过它是为了“协助”主处理器而存在的。ARM 架构里有 16 个协处理器，CP0～CP15。编程时涉及 CP15 比较多。

CP15 是系统控制协处理器，可控制处理器核的许多功能。它可以包含 16 个 32 位主寄存器。对 CP15 的访问受权限控制，并且在用户模式下并非所有寄存器都可用。CP15 寄存器访问指令指定所需的主寄存器，指令中的其他字段用于更精确地指定更多参数。CP15 中的 16 个主要寄存器的名称为 c0 至 c15，但通常

使用名称来引用。例如，CP15 系统控制寄存器称为 CP15.SCTLR。

通过从一个通用寄存器（Rt）读取或写入位于 CP15 内的一组寄存器（CRn）中，可以控制系统架构的某些功能。该指令的 Op1，Op2 和 CRm 字段也可以用于选择寄存器或操作。

格式如下所示：

(1) 从 CP15 寄存器读值到 ARM 寄存器：

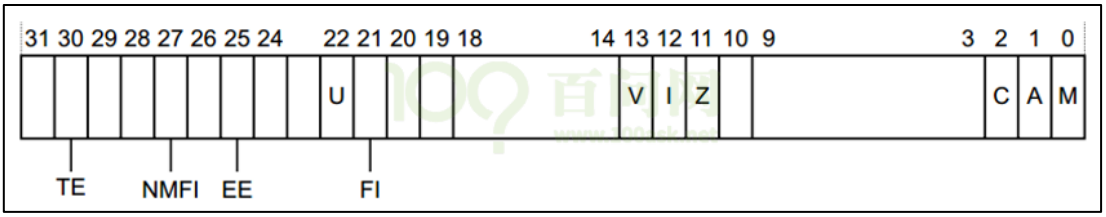
```
MRC p15, Op1, Rt, CRn, CRm, Op2 ; read a CP15 register into an ARM register
```

(2) 从 ARM 寄存器写值到 CP15 寄存器：

```
MCR p15, Op1, Rt, CRn, CRm, Op2 ; write a CP15 register from an ARM register
```

➢ **System control register (SCTLR)**

SCTLR 是通过 CP15 访问的寄存器，它控制存储器，系统功能，并提供反映处理器核中实现的功能的状态信息。系统控制寄存器只能从 PL1 或更高特权等级访问。




各个位的描述如下：

- TE －Thumb 异常使能。这可以控制异常进入 ARM 状态，还是 Thumb 状态。
- NMFI －是否支持不可屏蔽的 FIQ（NMFI）支持。
- EE =异常字节序。这定义了在异常时的字节序，的 CPSR.E 位的值。
- U －使用对齐模型。
- FI － FIQ 配置启用。
- V －该位选择异常向量表的基地址。
- I －指令缓存使能位。
- Z －分支预测使能位。
- C －缓存使能位。
- A －对齐检查使能位。
- M －启用 MMU。

引导代码序列的一部分通常将是设置 CP15：SCTLR 系统控制寄存器中的 Z 位，以启用分支预测功能。操作的代码如下：

```
MRC p15, 0, r0, c1, c0, 0 ; Read System Control Register configuration data
ORR r0, r0, #(1 << 2) ; Set C bit
ORR r0, r0, #(1 << 12) ; Set I bit
ORR r0, r0, #(1 << 11) ; Set Z bit
```

```
MCR p15, 0, r0, c1, c0, 0 ; Write System Control Register configuration data
```

## 10.5 异常处理

异常（exception）就是发生了意外的情况，它会中止处理器正常的执行流程。发生异常时，处理器要去执行对应的程序（称为异常处理程序）。异常有很多种，每种都有自己的处理程序，中断是一种异常。处理完异常后，要恢复发生异常之前的、被打断的操作。

CPU 每执行完一条指令，都会检查一下是否发生了某个异常，若是则中断当前执行流程，转去处理异常。

其他体系结构可能会将 ARM 所谓的异常称为陷阱（traps）或中断（interrupts），但是，在 ARM 体系结构中，这些术语保留用于特定类型的异常。

所有微处理器都必须响应外部异步事件，例如按下按钮或时钟超时。处理器核可以响应此类事件的速度可能是系统设计中的关键问题，称为中断等待时间（interrupt latency）。

很多单片机程序是写一个 main 主循环，在里面不断去查询是否发生了某些事，然后处理这些事。

实际上，在许多嵌入式系统中，并没有这样的 main 主循环，系统的所有功能都由中断代码来驱动。发生了某个中断，处理器就去执行对应的函数。复杂的系统有许多中断源，它们具有不同的优先级，并且支持中断嵌套，其中较高优先级的中断可以中断较低优先级的中断。

在正常程序执行中，程序计数器在地址空间中递增，程序中的分支指令会修改执行流程，例如，函数调用，循环和条件代码。当发生异常时，此预定的执行顺序将中断，并暂时切换到异常处理程序以处理该异常。

除了响应外部中断外，还有许多其他因素可能导致处理器核发生异常，包括外部（例如，复位），来自内存系统的异常终止以及内部（例如 MMU 生成的异常终止或通过 SVC 指令进行的 OS 调用）异常。处理异常会导致 CPU 核在模式之间切换并将某些寄存器复制到其他寄存器中。

### 10.5.1 异常的类型

ARMv7-A 和 ARMv7-R 体系结构支持多种处理器模式，称为 FIQ，IRQ，Supervisor，中止，未定义和系统的六种特权模式以及非特权的用户模式。在特权模式下可以修改 CPSR 寄存器来改变当前的处理器模式，或者在发生异常时自动改变。

非特权用户模式不能直接影响处理器核的异常行为，但是可以使用 SVC 异常以请求特权服务。这是用户应用程序请求操作系统来完成任务的方式，比如 Linux 的用户程序可以执行“SVC”指令触发异常，从而进入内核。

发生异常时，内核将保存当前状态和返回地址，进入特定模式，并可能禁用硬件中断。异常程序一般从某个固定的地址开始执行，这个固定的地址被称为“异常向量”。新的处理器允许把异常向量设在任何地方，但是要把它的地址写入某些寄存器，这样 CPU 会去这些寄存器获得地址并跳转执行。

存在以下异常类型：

（1）中断

ARMv7-A 内核提供两种中断，称为 IRQ 和 FIQ。

FIQ 的优先级高于 IRQ，并且 FIQ 模式下可用的更多的备份寄存器，因此 FIQ 具有一些潜在的速度优势。高优先级保证了它能尽快被处理，有更多备份寄存器可以节省将寄存器保存到堆栈时耗费的时钟周期。

FIQ、IRQ 异常通常都与处理器核上的输入引脚相关联：外部硬件会触发一条中断请求线，通过中断控制器去中断处理器。

FIQ 和 IRQ 都是发给处理器核的物理信号，如果处理器的 CPSR 寄存器中 FIQ 和 IRQ 处于打开状态，它将处理相应的异常。

几乎在所有系统上，通过使用中断控制器连接各种中断源。中断控制器对中断进行仲裁并确定优先级，然后依次提供串行化的单个信号，然后将其连接到内处理器核核的 FIQ 或 IRQ 引脚。

你不知道 IRQ 和 FIQ 中断何时发生，跟处理器当前执行的软件无关，因此将它们分类为异步异常。

（2）中止(ABT)

中止可以在指令预取失败（预取中止）或数据访问失败（数据中止）时生成。它们可以来自外部存储器系统，在存储器访问时给出错误响应（可能表明指定的地址不对应于系统中的实际存储器）。另外，中止可以由内核的内存管理单元（MMU）生成。操作系统可以使用 MMU 中止来为应用程序动态分配内存。

预取一条指令时，可以在指令流水线中中将其标记为已中止。仅当内核尝试执行它时，才导致预取中止异常。异常发生在指令执行之前。如果标记为中止的指令到达指令流水线的执行阶段之前刷新了指令流水线，则不会发生中止异常。数据中止异常发生在加载或存储指令执行时，并且是在尝试读取或写入数据之后发生的。

如果中止是由于指令流的执行或尝试执行而产生的，则中止被描述为同步的，并且返回地址将提供导致该中止的指令的详细信息。

异步的中止不是由执行指令生成，异步中止的返回地址可能不提供导致中止的原因的信息。

ARMv7 体系结构分为精确的和不精确的异步中止。MMU 产生的中止总是同步的。ARMv7 体系结构不需要外部中止的类型是同步的。例如，在一个特定的实现上，页表翻译时报告的外部异常中止被认为是精确的，但这并不是所有处理器核都需要的。对于精确的异步中止，中止处理程序可以确定是哪条指令导致了中止，并且在该指令之后没有执行其他指令。这与不精确的异步异常中止相反，异步异常中止是外部存储器系统报告有关无法识别的访问的错误时的结果。在这种情况下，中止处理程序无法确定是哪条指令导致了问题，或者在产生中止的指令之后是否还会执行其他指令。

例如，如果缓冲写入从外部存储系统接收到错误响应，则执行存储指令后很可能执行了其他指令。这意味着中止处理程序无法修复此问题并返回到应用程序。它所能做的就是杀死导致问题的应用程序。因此，设备探测需要特殊的处理，因为从外部报告的对不存在区域的读取中止将产生不精确的同步中止，即使将此类存储器标记为“strong odered”或“设备”。

异步中止的检测由 CPSR A 位控制。如果将 A 位置 1，CPU 核将识别出外部存储系统的异步异常中止，但不会产生中止异常。取而代之的是，内核将中止挂起状态挂起，直到清除 A 位时才采取异常处理为止。内核代码将使用屏障指令来确保针对正确的应用程序识别未处理的异步中止。如果由于不精确的中止而不得不终止线程，则该线程必须是正确的线程。

（3）复位

所有处理器核都有复位输入，并且在复位后将立即执行复位异常。它是最高优先级的异常，无法屏蔽。上电后，此异常用于在处理器核上执行代码以对其进行初始化。

（4）生成异常的指令

某些指令的执行会产生异常。通常执行以下指令，以便从更高特权级别的软件中请求服务：

- Supervisor Call（SVC）指令使用户模式程序可以请求操作系统服务。
- 如果实施了虚拟化扩展，则可以使用 Hypervisor 调用（HVC）指令，使虚拟机可以请求 Hypervisor 服务。
- 如果实施了安全扩展，则可以使用（SMC）指令，使普通环境可以请求安全环境服务。

任何试图执行处理器核无法识别的指令都会产生未定义的异常。

发生异常时，CPU 核将执行与该异常对应的处理程序。异常处理程序在内存中的存储位置称为异常向量。在 ARM 体系结构中，异常向量存储在称为异常向量表的表中。因此，用于特定异常的向量可以位于异常向量表起始位置的固定偏移处。该向量表的基地址由特权软件在系统寄存器中指定，以便处理器核可以在发生异常时找到相应的处理程序。

可以用 ARM 或 Thumb 代码编写异常处理程序。CP15 SCTLR.TE 位用于指定异常处理程序将使用 ARM 还是 Thumb 指令集。处理异常时，必须保留处理器核先前的模式，状态和寄存器，以便可以在处理异常后恢复原来程序的执行。

## 10.5.2 异常优先级

当异常同时发生时，将依次处理每个异常，然后返回原来执行的应用程序。所有异常不可能同时发生。例如，未定义指令（Undef）和 supervisor call（SVC）异常是互斥的，因为它们都是由执行指令触发的。

注意：ARM 体系结构未定义何时采用异步异常。因此，异步异常相对于其他异常（同步和异步）的优先级由芯片厂家决定。

所有异常均禁用 IRQ，只有 FIQ 和复位禁用 FIQ。这是由处理器核自动设置 CPSR I（IRQ）和 F（FIQ）位来完成的。意思是说，发生某个异常时，处理器 CPSR 的 IRQ 位就被禁止了，在异常处理期间按键等中断是不会被处理的。这也意味着发生了某个 IRQ 中断后，处理器 CPSR 的 IRQ 位就被禁止，除非你的代码中再次去开启 CPSR 的 IRQ 位，否则中断无法嵌套。

可能同时产生多个异常，但是某些组合是互斥的。预取中止将一条指令标记为无效，因此不能与未定义的指令或 SVC 同时发生（当然，SVC 指令也不能是未定义的指令）。这些指令不会导致任何内存访问，因此不会导致数据中止。该体系结构未定义何时必须采取异步异常，FIQ，IRQ 或异步异常中止，但是采用 IRQ 或数据异常中止不会禁用 FIQ 异常这一事实意味着 FIQ 执行将优先于 IRQ 或异常中止异常。

发生异常时，处理器跳转去固定的地址执行代码，每个异常都有对应的固定地址。默认情况下，这些地址位为：从 0x00 到 0x1C，向量表基地址为 0。向量表可以从 0x0 移到 0xFFFF0000。

对于带有 Security Extensions 的内核，情况更加复杂，本教程不涉及。

下表总结了各种状态下异常的行为，我们只需要关心第 1、2、3 列。

| Normal Vector offset | High vector address | Non-secure | Secure | Hypervisor[a] | Monitor |
|---|---|---|---|---|---|
| 0x0 | 0xFFFF0000 | Not used | Reset | Reset | Not used |
| 0x4 | 0xFFFF0004 | UNDEFINED instruction | UNDEFINED instruction | UNDEFINED instruction from Hyp mode. | Not used |
| 0x8 | 0xFFFF0008 | Supervisor Call | Supervisor Call | Secure Monitor Call | Secure Monitor Call |
| 0xC | 0xFFFF000C | Prefetch Abort | Prefetch Abort | Prefetch Abort from Hyp mode. | Prefetch Abort |
| 0x10 | 0xFFFF0010 | Data Abort | Data Abort | Data Abort from Hyp mode, | Data Abort |
| 0x14 | 0xFFFF0014 | Not used | Not used | Hyp mode entry | Not used |
| 0x18 | 0xFFFF0018 | IRQ interrupt | IRQ interrupt | IRQ interrupt | IRQ interrupt |
| 0x1C | 0xFFFF001C | FIQ interrupt | FIQ interrupt | FIQ interrupt | FIQ interrupt |

进入异常时，CPSR 的 I 和 F 设置如下表所示：

| Exception | Mode | CPSR interrupt mask |
|---|---|---|
| Reset | Supervisor | F = 1<br>I = 1 |
| UNDEFINED instruction | Undef | I = 1 |
| Supervisor Call | Supervisor | I = 1 |
| Prefetch Abort | Abort | I = 1 |
| Data Abort | Abort | I = 1 |
| Not used | HYP | - |
| IRQ interrupt | IRQ | I = 1 |
| FIQ interrupt | FIQ | F = 1<br>I = 1 |

### 10.5.3 向量表

“向量”里存有一条指令，用来处理某个异常。触发异常时 ARM 核会跳转到某个“向量”，执行其中的指令。这些“向量”汇集在一起，把它们称为“向量表”，它位于内存中的特定位置。默认向量基址为 0x00000000，但大多数 ARM 核允许将向量基址移至 0xFFFF0000（或 HIVECS）。所有 Cortex-A 系列处理器都允许这样做，这是 Linux 内核选择的默认地址。实现安全扩展的内核还可以使用 CP15 向量基地址寄存器为安全状态和非安全状态分别设置向量基地址。

每个异常只能在向量表中放置一条指令（尽管从理论上讲，可以使用两条 16 位 Thumb 指令）。因此，向量表条目几乎总是包含以下两种形式的分支之一。

① B <label>

这将执行 PC 相对跳转。它可以跳转到当前指令的前后 32MB。

② LDR PC, [PC, #offset]

这将从地址相对于异常指令 offset 偏移量的值加载到 PC。这样就可以将异常处理程序放置在 32 位内存地址空间内的任意地址处（但相对于 B 指令，要多花一些额外的指令周期）。

Non-secure 状态下 SCTLR.V 位决定了向量表的基地址，如果 V==0 的话，Non-secure VBAR 保存了异常向量基地址；如果 v==1 的话，异常向量基地址为 0xFFFF0000。

### 10.5.4 FIQ and IRQ

FIQ 很少用到，它只能用于某一个中断：系统中有很多中断，如果某个中断的处理需要有严格的时间保证，你可以把它设置为 FIQ。

而 IRQ 用于系统中的所有其他中断。

对 FIQ 的设计是精心考虑的，它是向量表中的最后一项。一般的异常向量里跳转指令，但是 FIQ 向量位于最后一项，处理程序可以直接放置在向量入口位置，并从该地址开始顺序运行。这避免了分支指令和任何相关的延迟，从而加快了 FIQ 响应时间。相对于其他模式，FIQ 模式下可用的备份寄存器数量比较多，从而避免要将寄存器的值保存到栈上，提高了执行速度。

Linux 通常不使用 FIQ，某些运行 Linux 的系统仍可以使用 FIQ，但是由于 Linux 内核从不禁用 FIQ，因此它们比系统中的其他任何事物都具有优先权，因此需要格外小心。

### 10.5.5 返回指令

处理异常后，链接寄存器（LR）用于为存储返回地址。下表提供了包含此调整的异常的返回指令。

| Exception | Adjustment | Return instruction | Instruction returned to |
|---|---|---|---|
| SVC | 0 | MOVS PC, R14 | Next instruction |
| Undef | 0 | MOVS PC, R14 | Next instruction |
| Prefetch Abort | -4 | SUBS PC, R14, #4 | Aborting instruction |
| Data abort | -8 | SUBS PC, R14, #8 | Aborting instruction if precise |
| FIQ | -4 | SUBS PC, R14, #4 | Next instruction |
| IRQ | -4 | SUBS PC, R14, #4 | Next instruction |

### 10.5.6 异常处理

发生异常时，ARM 内核会自动执行以下操作：

① 将 CPSR 复制到 SPSR_<mode>，比如发生 IRQ 异常时，当前 CPSR 就会被保存到 SPSR_IRQ 里。

② 将返回地址存储在新模式的链接寄存器（LR）中。

③ 将 CPSR 模式位修改为与异常类型相关联的模式。

- 其他 CPSR 模式位设置由 CP15 系统控制寄存器的值确定。
- T 位设置为 CP15 TE 位给定的值。
- J 位被清除，E 位（字节序）被设置为 EE（异常字节序）位的值。

这意味着，设置好了 CP15 后，异常处理程序的状态(ARM 或 Thumb)、字节序(小端或大端)就确定了。无论 CPU 核在异常之前处于何种状态，都不会影响到异常处理程序的状态。

④ 将 PC 设置为指向异常向量表中的相关指令。

在新模式下，CPU 核将访问与该模式关联的寄存器。

异常处理程序软件几乎总是需要在进入异常处理程序时立即将寄存器保存到堆栈中。FIQ 模式具有更多的备份寄存器，因此可以编写不使用堆栈的简单处理程序。

在 ARMv6 及更高版本的 ARM 体系结构中，提供了一种特殊的汇编语言指令来帮助保存必要的寄存器，称为 SRS（store return state 存储返回状态）。该指令将 LR 和 SPSR 压入任何模式的堆栈，所使用的堆栈由指令操作数指定。

- **从异常处理程序返回**

要从异常处理程序返回，必须进行两个单独的操作：

① **从保存的 SPSR 中恢复 CPSR。**

② **将返回地址写入 PC 寄存器。**

在 ARM 体系结构中，这可以通过使用两类指令从异常中返回：

a) **RFE 指令：**

它将链接寄存器和 SPSR 从当前模式堆栈弹出。

b) **其他会设置 PC 寄存器指令，并让该指令带有 S 后缀：**

例如“SUBS PC，LR，#offset”，注意：“S”表示从 SPSR 中恢复 CPSR。

如果异常处理程序入口，使用堆栈来保存现场，则它可以使用带有“^”限定符的加载指令返回。例如，异常处理程序可以使用以下命令在一条指令中返回：

```
LDMFD sp!, {pc} ^
LDMFD sp!, {R0-R12, pc} ^
```

在此示例中，^限定符表示 SPSR 同时复制到 CPSR。

注意：不能使用 16 位 Thumb 指令从异常中返回，因为这些指令无法还原 CPSR。

## 10.5.7 中止(ABT)处理程序

“中止异常”的处理程序代码在系统之间可能有很大差异。

在许多嵌入式系统中，中止异常表示意外错误，处理程序将记录所有诊断信息，报告错误并让应用程序（或系统）退出。

在使用 MMU 支持虚拟内存的系统中，中止处理程序可以将所需的虚拟页加载到物理内存中。比如一个程序很大，没必要在一开始执行时就把它全部读入内存。等程序执行时发现内存中没数据时，就会触发中止异常，在异常的处理函数中再读入更多的数据。

CP15 寄存器提供了导致中止异常的存储器访问地址（故障地址寄存器 Fault Address Register）和中止的原因（故障状态寄存器 Fault Status Register）。原因可能是缺少访问权限，外部中止或地址转换错误。此外，链接寄存器（进行了 –8 或 –4 调整，取决于中止是由指令获取还是数据访问引起的），给出了导致中止异常的指令的地址。通过检查这些寄存器、最后执行的指令，以及系统中可能的其他内容（例如转换表条目），中止异常的处理程序就可以知道应该采取什么处理措施了。

## 10.5.8 未定义的指令处理

简单地说，就是 CPU 或协处理器不认识这条指令，执行这样的指令时就会产生“未定义指令异常”。

如果 CPU 核尝试使用操作码执行一条指令（在 ARM 体系结构规范中描述为 UNDEFINED），或者执行了协处理器指令但没有协处理器将其识别为可以执行的

指令，则会导致未定义的指令异常。

在某些系统中，代码可能包含用于协处理器（例如 VFP 协处理器）的指令，但是系统中不存在相应的 VFP 硬件。另外，VFP 硬件有可能无法处理特定指令，而是想调用软件来对其进行仿真。或者，可能会禁用 VFP 硬件，采用异常处理，以便可以启用它，然后重新执行指令。

使用未定义的指令，可以实现一些仿真器。比如在你的芯片中，它并未支持某条硬件除法指令，但是你还可以在代码中使用它。当 CPU 执行这条指令时会发生异常，在异常处理函数中，你用软件来实现该指令的功能。

对于不是特别设置的未定义指令，在异常处理函数中不能处理它时，通常做法是记录适当的调试信息，并杀死对应的应用程序。

在某些情况下，未定义指令异常的另一个用途是实现用户断点：调试器去修改代码，替换断点位置的指令为一条未定义指令。

### 10.5.9 SVC 异常处理

简单地说就是执行 SVC 这条汇编指令时就会触发这个异常，CPU 就会跳转过执行 SVC 异常向量的代码。

supervisor call（SVC）通常在用户模式下使用，这使得用户模式的代码能够访问 OS 功能。例如，如果用户代码想要访问系统的特权部分（例如执行文件 I / O），则通常将使用 SVC 指令执行此操作。在 Linux 中对文件的 open/read/write 等 APP 层的系统函数，它的本质都是执行 SVC 指令，从而进入 Linux 内核中预设的 SVC 异常处理函数，在内核里操作文件。

可以使用寄存器或者操作码中某个字段将参数传递给 SVC 处理程序。

发生异常时，异常处理程序可能必须确定内核是处于 ARM 还是 Thumb 状态。

特别是 SVC 处理程序，可能必须读取指令集状态。这是通过检查 SPSR T 位完成的。该位设置为 Thumb 状态，清除为 ARM 状态。

ARM 和 Thumb 指令集都具有 SVC 指令。从 Thumb 状态调用 SVC 时，必须考虑以下因素：

- 指令地址位于 LR-2，而不是 LR-4；
- 指令本身是 16 位的，因此需要半字加载；
- SVC 编号为 8 位而不是 ARM 状态下的 24 位。

注意：ARM9 等比较老的芯片里，这个异常是 SWI 异常，对应的指令是 SWI。

10.5.10 代码分析

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/10_异常与中断/001_exception_undef”目录下。

通过在代码段里里插入一个未定义指令（0xeeadc0de），从而产生未定义指令异常。在未定义异常指令异常的处理函数里，调用 printException 函数，打印出当前的 CPSR 值，和产生异常的原因。

在复位 Reset_Handler 里要分别设置好 SVC 模式和 und 模式的栈，这样我们就可以在各自的模式里调用 C 代码。

通过如下指令，设置好异常向量的基地址：

```
mcr p15, 0, r0, c12, c0, 0
```

在异常向量表里，通过如下指令跳转到 Undefined_Handler 标签处

```
ldr pc, =Undefined_Handler
```

在 Undefined_Handler 里将 r0-r12 和 lr 保存在 und 模式的栈上，然后调用 printException 打印当前的 CPSR 值，并打印一个字符串。最后将 r0-r12 从栈上恢复，lr 从栈上弹出到 PC，并同时将 SPSR 恢复到 CPSR，从而返回去执行出现未定义异常指令的下一条指令。

代码如下 008_exception_undef\start.S：

```
.text
.global  _start, _vector_table
_start:
_vector_table:
    ldr pc, =Reset_Handler             /* Reset                   */
    ldr pc, =Undefined_Handler         /* Undefined instructions */
    //b Reset_Handler
    //b Undefined_Handler
    b halt//b SVC_Handler//ldr    pc, =SVC_Handler          /* Supervisor Call
*/
    b halt//ldr  pc, =PrefAbort_Handler         /* Prefetch abort          */
    b halt//ldr  pc, =DataAbort_Handler         /* Data abort              */
    .word    0                                /* RESERVED            */
    b halt//ldr  pc, =IRQ_Handler           /* IRQ interrupt       */
    b halt//ldr  pc, =FIQ_Handler           /* FIQ interrupt       */

.align 2
Undefined_Handler:
    /* 执行到这里之前:
     * 1. lr_und 保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_und 保存有被中断模式的 CPSR
     * 3. CPSR 中的 M4-M0 被设置为 11011，进入到 und 模式
     * 4. 跳到 0x4 的地方执行程序
     */

    /* 在 und 异常处理函数中有可能会修改 r0-r12，所以先保存 */
    /* lr 是异常处理完后的返回地址，也要保存 */
    stmdb sp!, {r0-r12, lr}

    /* 保存现场 */
    /* 处理 und 异常 */
    mrs r0, cpsr
    ldr r1, =und_string
    bl printException

    /* 恢复现场 */
    ldmia sp!, {r0-r12, pc}^  /* ^会把 spsr 的值恢复到 cpsr 里 */

und_string:
    .string "undefined instruction exception"

.align 2
Reset_Handler:
    /* Reset SCTlr Settings */
    mrc p15, 0, r0, c1, c0, 0   /* read SCTRL, Read CP15 System Control register */
    bic r0,  r0, #(0x1 << 13)/* Clear V bit 13 to use normal exception vectors */
    bic r0,  r0, #(0x1 << 12)/* Clear I bit 12 to disable I Cache */
    bic r0,  r0, #(0x1 <<  2)/* Clear C bit  2 to disable D Cache */
    bic r0,  r0, #(0x1 << 2)  /* Clear A bit  1 to disable strict alignment */
    bic r0,  r0, #(0x1 << 11)/* Clear Z bit 11 to disable branch prediction */
    bic r0,  r0, #0x1             /* Clear M bit  0 to disable MMU */
```

```
    mcr  p15, 0, r0, c1, c0, 0/* write SCTRL, CP15 System Control register   */

    cps     #0x1B               /* Enter undef mode               */
    ldr     sp, =0x80300000     /* Set up undef mode stack      */

    cps     #0x13               /* Enter Supervisor mode         */
    ldr     sp, =0x80200000     /* Set up Supervisor Mode stack  */

    ldr r0, =_vector_table
    mcr p15, 0, r0, c12, c0, 0  /* set VBAR, Vector Base Address Register*/
    //mrc p15, 0, r0, c12, c0, 0  //read VBAR

    bl clean_bss

    bl system_init

und_code:
    .word 0xeeadc0de  /* undefine instruction */
    //.word 0xFFFFFFFF

    bl main

halt:
    b  halt

clean_bss:
    /* 清除 BSS 段 */
    ldr r1, =__bss_start
    ldr r2, =__bss_end
    mov r3, #0
clean:
    cmp r1, r2
    strlt r3, [r1]
    add r1, r1, #4
    blt clean

    mov pc, lr
```

插入的未定义指令代码如下(008_exception_undef\start.S):

```
und_code:
.word 0xeeadc0de  /* undefine instruction */
```

printException 函数代码如下（008_exception_undef \main.c）:

```
void printException(unsigned int cpsr, char *str)
{
printf("Exception! cpsr is 0x%x\r\n", cpsr);
printf("%s\r\n", str);
}
```

- 参考章节《4.3.4 编译程序》编译程序
- 参考章节《3.4 映像文件烧写、运行》烧写、运行程序
- 此时观察串口打印

# 10.6 SVC 异常模示程序示例

## 10.6.1 代码分析

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/10_异常与中断/002_exception_swi”目录下。

然后通过执行“swi 123”指令，触发 SVC 异常，程序跳转到异常向量表偏移 0x8 的地方执行，在异常向量表里通过如下指令跳转到 SVC_Handler 标签处执行。

```
ldr pc, =SVC_Handler
```

在 SVC_Handler 里将 r0-r12 和 lr 保存在 SVC 模式的栈上，然后将 lr 的值移动到 R4，调用 printException 函数打印出当前的 CPSR 值，和产生异常的原因。将 R4 减去 4，赋值给 R0，这是 swi 指令所在的地址，然后调用 printSWIVal 函数打印出 swi 指令的参数。最后将 r0-r12 从栈上恢复，lr 从栈上弹出到 PC，并同时将 SPSR 恢复到 CPSR，从而返回去执行 swi 指令的下一条指令。代码如下（002_exception_swi\start.S）：

```
.text
.global  _start, _vector_table
_start:
_vector_table:
    ldr pc, =Reset_Handler            /* Reset                    */
    ldr pc, =Undefined_Handler        /* Undefined instructions */
    ldr pc, =SVC_Handler          /* Supervisor Call     */
    b halt//ldr  pc, =PrefAbort_Handler        /* Prefetch abort          */
    b halt//ldr  pc, =DataAbort_Handler        /* Data abort              */
    .word    0                              /* RESERVED            */
    b halt//ldr  pc, =IRQ_Handler          /* IRQ interrupt       */
    b halt//ldr  pc, =FIQ_Handler          /* FIQ interrupt       */

.align 2
Undefined_Handler:
    /* 执行到这里之前:
     * 1. lr_und 保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_und 保存有被中断模式的 CPSR
     * 3. CPSR 中的 M4-M0 被设置为 11011，进入到 und 模式
     * 4. 跳到 0x4 的地方执行程序
     */

    /* 在 und 异常处理函数中有可能会修改 r0-r12，所以先保存 */
    /* lr 是异常处理完后的返回地址，也要保存 */
    stmdb sp!, {r0-r12, lr}

    /* 保存现场 */
    /* 处理 und 异常 */
    mrs r0, cpsr
    ldr r1, =und_string
    bl printException

    /* 恢复现场 */
    ldmia sp!, {r0-r12, pc}^  /* ^会把 spsr 的值恢复到 cpsr 里 */

und_string:
    .string "undefined instruction exception"
```

```
.align 2
SVC_Handler:
    /* 执行到这里之前:
     * 1. lr_svc 保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_svc 保存有被中断模式的 CPSR
     * 3. CPSR 中的 M4-M0 被设置为 10011，进入到 svc 模式
     * 4. 跳到 0x08 的地方执行程序
     */

    /* 保存现场 */
    /* 在 swi 异常处理函数中有可能会修改 r0-r12，所以先保存 */
    /* lr 是异常处理完后的返回地址，也要保存 */
    stmdb sp!, {r0-r12, lr}

    mov r4, lr

    /* 处理 swi 异常 */
    mrs r0, cpsr
    ldr r1, =swi_string
    bl printException

    sub r0, r4, #4
    bl printSWIVal

    /* 恢复现场 */
    ldmia sp!, {r0-r12, pc}^  /* ^会把 spsr 的值恢复到 cpsr 里 */

swi_string:
    .string "swi exception"

.align 2
Reset_Handler:
    /* Reset SCTlr Settings */
    mrc p15, 0, r0, c1, c0, 0/* read SCTRL, Read CP15 System Control register */
    bic r0,  r0, #(0x1 << 13)/* Clear V bit 13 to use normal exception vectors */
    bic r0,  r0, #(0x1 << 12)/* Clear I bit 12 to disable I Cache */
    bic r0,  r0, #(0x1 <<  2)/* Clear C bit  2 to disable D Cache */
    bic r0,  r0, #(0x1 << 2) /* Clear A bit  1 to disable strict alignment */
    bic r0,  r0, #(0x1 << 11)/* Clear Z bit 11 to disable branch prediction */
    bic r0,  r0, #0x1            /* Clear M bit  0 to disable MMU */
    mcr p15, 0, r0, c1, c0, 0/* write SCTRL, CP15 System Control register */

    cps     #0x1B               /* Enter undef mode              */
    ldr     sp, =0x80300000     /* Set up undef mode stack     */

    cps     #0x13               /* Enter Supervisor mode        */
    ldr     sp, =0x80200000     /* Set up Supervisor Mode stack  */

    ldr r0, =_vector_table
    mcr p15, 0, r0, c12, c0, 0  /* set VBAR, Vector Base Address Register*/
    //mrc p15, 0, r0, c12, c0, 0  //read VBAR

    bl clean_bss

    bl system_init

und_code:
```

```
    .word 0xdeadc0de  /* undefine instruction */
    //.word 0xFFFFFFFF

swi_code:
    swi 0x123  /* 执行此命令，触发 SWI 异常，进入 0x8 执行 */

    bl main

halt:
    b  halt

clean_bss:
    /* 清除 BSS 段 */
    ldr r1, =__bss_start
    ldr r2, =__bss_end
    mov r3, #0
clean:
    cmp r1, r2
    strlt r3, [r1]
    add r1, r1, #4
    blt clean

    mov pc, lr
```

printException 函数与 und 异常模式代码里相同，printSWIVal 函数打印出 swi 的参数，代码如下（002_exception_swi\main.c）：

```
void printSWIVal(unsigned int *pSWI)
{
printf("SWI val = 0x%x\r\n", *pSWI & ~0xff000000);
}
```

注意：在上述代码中我们使用“swi”指令，实际上你可以在 except.dis 中看到它被翻译为 SVC 指令。在老的芯片比如 S3C2440 中我们使用 SWI 指令，在 IMX6ULL 中可以使用 SVC 代替它。

- **参考章节《4.3.4 编译程序》编译程序**
- **参考章节《3.4 映像文件烧写、运行》烧写、运行程序**
- **此时观察串口打印**

```
hello world
Exception! cpsr is 0x800001db
undefined instruction exception
Exception! cpsr is 0x800001d3
swi exception
SWI val = 0x123
```

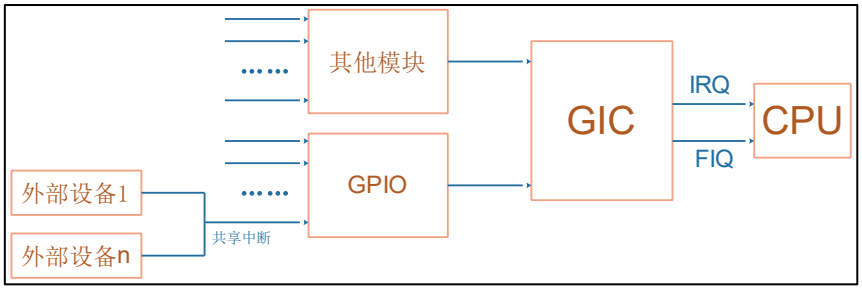
## 10.7 中断处理

对于 S3C2440 等较旧的 ARM 体系结构版本，芯片厂家在设计外部中断控制器时具有很大的自由度，在中断的数量、类型、与中断控制器模块接口等时方面都可以自由发挥。

通用中断控制器 v2（GIC）架构提供了更为严格的规范，不同厂商的中断控制器之间具有更高的一致性。这使中断处理程序代码更易于移植。

### 10.7.1 外部中断请求

ARM 核有两个外部中断请求 FIQ 和 IRQ，如下图所示：




各个不同的芯片实现，里面都有中断控制器。中断控制器接受来自各种外部源的中断请求并将它们映射为 FIQ 或 IRQ，从而导致 ARM 核发生异常。

通常，只有当相应的 CPSR 禁止位（分别为 F 和 I 位）清零并且相应的输入为有效时，才可以产生中断异常。

CPS 指令提供了一种简单的机制来启用或禁用由 CPSR A，I 和 F 位（分别为异步中止，IRQ 和 FIQ）控制的异常。

CPS IE 或 CPS ID 将分别启用或禁用异常。使用字母 A，I 和 F 中的一个或多个指定要启用或禁用的异常。省略了相应字母的异常将不会被修改。

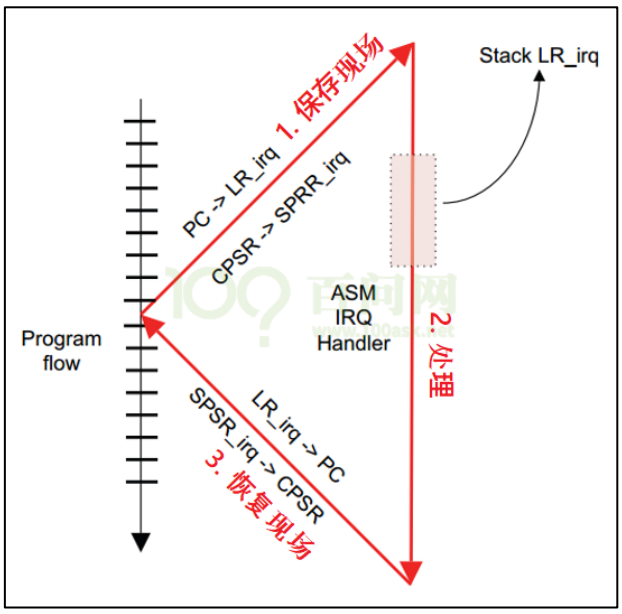
在 Cortex-A 系列处理器中，可以配置 CPU 核，以使 FIQ 不能被软件屏蔽。这被称为不可屏蔽 FIQ，并由 CPU 核复位时采样的硬件配置的输入信号控制。发生 FIQ 异常后，它们仍将自动被屏蔽。

### 10.7.2 分配中断

中断控制器接受来自各种源的中断， 并决定谁能优先被处理(仲裁)。控制器通常包含多个寄存器，通过这些寄存器，我们能够屏蔽或使能各个中断源，为各个中断源分配优先级；当发生中断时，可以确定中断源。

中断控制器可以由芯片厂家自己设计(比如比较老的芯片 S3C2440)，也可以使用 ARM 通用中断控制器（GIC）架构。

### 10.7.3 简单的中断处理




上图代表了最简单的中断处理程序。发生中断时，将禁用其他同类中断，直

到稍后显式启用。我们只能在第一个中断请求完成时才能处理其他中断，并且在此期间没有更高优先级或更紧急的中断需要处理。在这种情况下是不可重入的中断处理程序。
处理中断所采取的步骤如下：

**1．外部硬件引发 IRQ 异常**
ARM 核自动执行几个步骤：
① 当前模式下 PC 的内容存储在 LR_IRQ 中；
② CPSR 寄存器被复制到 SPSR_IRQ；
③ CPSR 内容被更新，设置模式位为 IRQ 模式，并且将 I 位设置为屏蔽其他 IRQ；
④ PC 被设置为向量表中的 IRQ 入口。

**2．执行向量表中 IRQ 入口处（中断异常的分支）的指令；**

**3．中断处理程序保存被中断程序的上下文：它将被该中断处理程序损坏的所有寄存器压入堆栈。**

**4．中断处理程序确定中断源，然后调用响应的处理程序。**

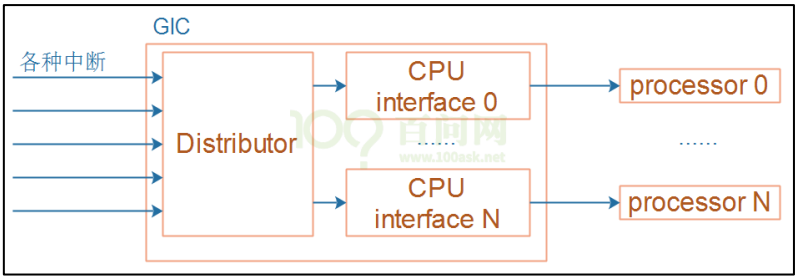
**5．恢复现场：**
通过将 SPSR_IRQ 复制到 CPSR，并还原先前保存的上下文，准备将 CPU 核切换到先前的执行状态，最后从 LR_IRQ 恢复 PC。
相同的顺序也适用于 FIQ 中断。

### 10.7.4 嵌套中断处理

嵌套中断处理是软件可以在完成对当前中断的处理之前，接受另一个中断。这可以将中断进行优先级分级，降低高优先级事件的响应延迟，代价是增加了软件的复杂性。值得注意的是，嵌套中断处理是由软件来实现。
支持中断嵌套的处理函数，被称为“可重入中断处理程序”。在 Linux 系统中，不支持中断嵌套。对于裸机程序，我们也不需要花费精力在这上面。在实时性要求很高的 RTOS 中，也许对中断嵌套有所要求。但是可以通过其他办法习实现，比如把某个中断单独设置在 FIQ。

## 10.8 通用中断控制器（GIC，Generic Interrupt Controller）

ARM 体系结构定义了通用中断控制器（GIC），该控制器包括一组用于管理单核或多核系统中的中断的硬件资源。GIC 提供了内存映射寄存器，可用于管理中断源和行为，以及（在多核系统中）用于将中断路由到各个 CPU 核。它使软件能够屏蔽，启用和禁用来自各个中断源的中断，以（在硬件中）对各个中断源进行优先级排序和生成软件触发中断。它还提供对 TrustZone 安全性扩展的支持。GIC 接受系统级别中断的产生，并可以发信号通知给它所连接的每个内核，从而有可能导致 IRQ 或 FIQ 异常发生。
从软件角度来看，GIC 具有两个主要功能模块，简单画图如下：




① 分发器(Distributor)

系统中的所有中断源都连接到该单元。可以通过仲裁单元的寄存器来控制各个中断源的属性，例如优先级、状态、安全性、路由信息和使能状态。

分发器把中断输出到“CPU 接口单元”，后者决定将哪个中断转发给 CPU 核。

② CPU 接口单元（CPU Interface）

CPU 核通过控制器的 CPU 接口单元接收中断。CPU 接口单元寄存器用于屏蔽，识别和控制转发到 CPU 核的中断的状态。系统中的每个 CPU 核心都有一个单独的 CPU 接口。

中断在软件中由一个称为中断 ID 的数字标识。中断 ID 唯一对应于一个中断源。软件可以使用中断 ID 来识别中断源并调用相应的处理程序来处理中断。呈现给软件的中断 ID 由系统设计确定，一般在 SOC 的数据手册有记录。

➢ 中断可以有多种不同的类型：

① 软件触发中断（SGI，Software Generated Interrupt）

这是由软件通过写入专用仲裁单元的寄存器即软件触发中断寄存器（ICDSGIR）显式生成的。它最常用于 CPU 核间通信。SGI 既可以发给所有的核，也可以发送给系统中选定的一组核心。中断号 0-15 保留用于 SGI 的中断号。用于通信的确切中断号由软件决定。

② 私有外设中断（PPI，Private Peripheral Interrupt）

这是由单个 CPU 核私有的外设生成的。PPI 的中断号为 16-31。它们标识 CPU 核私有的中断源，并且独立于另一个内核上的相同中断源，比如，每个核的计时器。

③ 共享外设中断（SPI，Shared Peripheral Interrupt）

这是由外设生成的，中断控制器可以将其路由到多个核。中断号为 32-1020。SPI 用于从整个系统可访问的各种外围设备发出中断信号。

中断可以是边沿触发的（在中断控制器检测到相关输入的上升沿时认为中断触发，并且一直保持到清除为止）或电平触发（仅在中断控制器的相关输入为高时触发）。

➢ 中断可以处于多种不同状态：

① 非活动状态（Inactive）－这意味着该中断未触发。

② 挂起（Pending）－这意味着中断源已被触发，但正在等待 CPU 核处理。待处理的中断要通过转发到 CPU 接口单元，然后再由 CPU 接口单元转发到内核。

③ 活动（Active）－描述了一个已被内核接收并正在处理的中断。

④ 活动和挂起（Active and pending）－描述了一种情况，其中 CPU 核正在为中断服务，而 GIC 又收到来自同一源的中断。

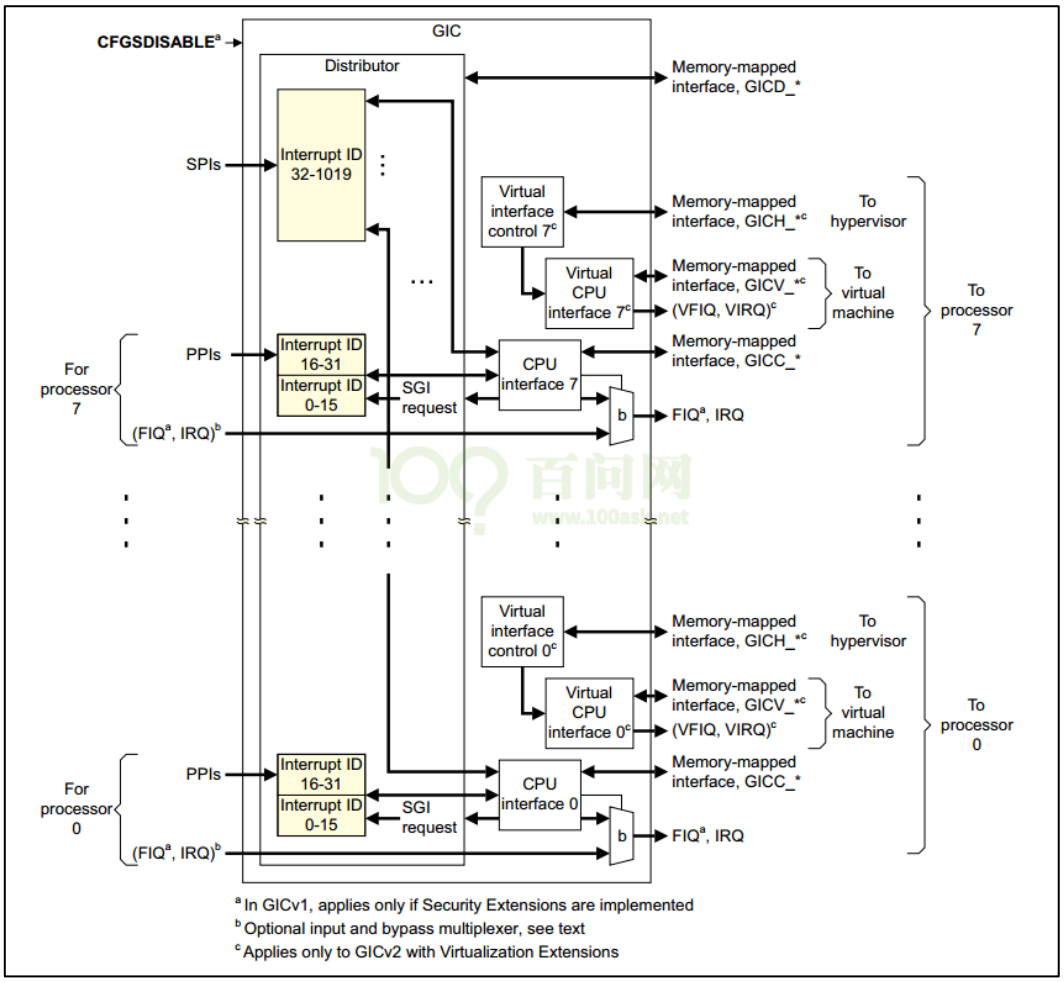
中断的优先级和可接收中断的核都在分发器(distributor)中配置。外设发给分发器的中断将标记为 pending 状态（或 Active and Pending 状态，如

触发时果状态是 active）。distributor 确定可以传递给 CPU 核的优先级最高的 pending 中断，并将其转发给内核的 CPU interface。通过 CPU interface，该中断又向 CPU 核发出信号，此时 CPU 核将触发 FIQ 或 IRQ 异常。

作为响应，CPU 核执行异常处理程序。异常处理程序必须从 CPU interface 寄存器查询中断 ID，并开始为中断源提供服务。完成后，处理程序必须写入 CPU interface 寄存器以报告处理结束。然后 CPU interface 准备转发 distributor 发给它的下一个中断。

在处理中断时，中断的状态开始为 pending，active，结束时变成 inactive。中断状态保存在 distributor 寄存器中。

下图是 GIC 控制器的逻辑结构：




## 10.8.1 配置

GIC 作为内存映射的外围设备，被软件访问。所有内核都可以访问公共的 distributor 单元，但是 CPU interface 是备份的，也就是说，每个 CPU 核都使用相同的地址来访问其专用 CPU 接口。一个 CPU 核不可能访问另一个 CPU 核的 CPU 接口。

Distributor 拥有许多寄存器，可以通过它们配置各个中断的属性。这些可配置属性是：

- 中断优先级：Distributor 使用它来确定接下来将哪个中断转发到 CPU 接口。

- 中断配置：这确定中断是对电平触发还是边沿触发。
- 中断目标：这确定了可以将中断发给哪些 CPU 核。
- 中断启用或禁用状态：只有 Distributor 中启用的那些中断变为挂起状态时，才有资格转发。
- 中断安全性：确定将中断分配给 Secure 还是 Normal world 软件。
- 中断状态。

Distributor 还提供优先级屏蔽，可防止低于某个优先级的中断发送给 CPU 核。

每个 CPU 核上的 CPU interface，专注于控制和处理发送给该 CPU 核的中断。

## 10.8.2 初始化

Distributor 和 CPU interface 在复位时均被禁用。复位后，必须初始化 GIC，才能将中断传递给 CPU 核。

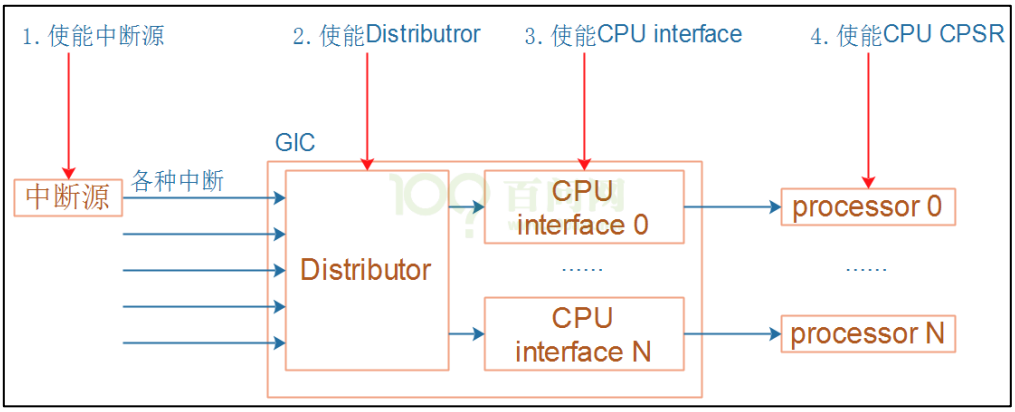
在 Distributor 中，软件必须配置优先级、目标核、安全性并启用单个中断；随后必须通过其控制寄存器使能。

对于每个 CPU interface，软件必须对优先级和抢占设置进行编程。每个 CPU 接口模块本身必须通过其控制寄存器使能。

在 CPU 核可以处理中断之前，软件会通过在向量表中设置有效的中断向量并清除 CPSR 中的中断屏蔽位来让 CPU 核可以接收中断。

可以通过禁用 Distributor 单元来禁用系统中的整个中断机制；可以通过禁用单个 CPU 的 CPU 接口模块或者在 CPSR 中设置屏蔽位来禁止向单个 CPU 核的中断传递。也可以在 Distributor 中禁用（或启用）单个中断。

为了使某个中断可以触发 CPU 核，必须将各个中断，Distributor 和 CPU interface 全部使能，并将 CPSR 中断屏蔽位清零，如下图：




## 10.8.3 GIC 中断处理

当 CPU 核接收到中断时，它会跳转到中断向量表执行。

顶层中断处理程序读取 CPU 接口模块的 Interrupt Acknowledge Register，以获取中断 ID。除了返回中断 ID 之外，读取操作还会使该中断在 Distributor 中标记为 active 状态。一旦知道了中断 ID（标识中断源），顶层处理程序现在就可以分派特定于设备的处理程序来处理中断。

当特定于设备的处理程序完成执行时，顶级处理程序将相同的中断 ID 写入

CPU interface 模块中的 End of Interrupt register 中断结束寄存器，指示中断处理结束。除了把当前中断移除 active 状态之外，这将使最终中断状态变为 inactive 或 pending（如果状态为 inactive and pending），这将使 CPU interface 能够将更多待处理 pending 的中断转发给 CPU 核。这样就结束了单个中断的处理。

同一 CPU 核上可能有多个中断等待服务，但是 CPU interface 一次只能发出一个中断信号。顶层中断处理程序重复上述顺序，直到读取特殊的中断 ID 值 1023，表明该内核不再有任何待处理的中断。这个特殊的中断 ID 被称为伪中断 ID（spurious interrupt ID）。

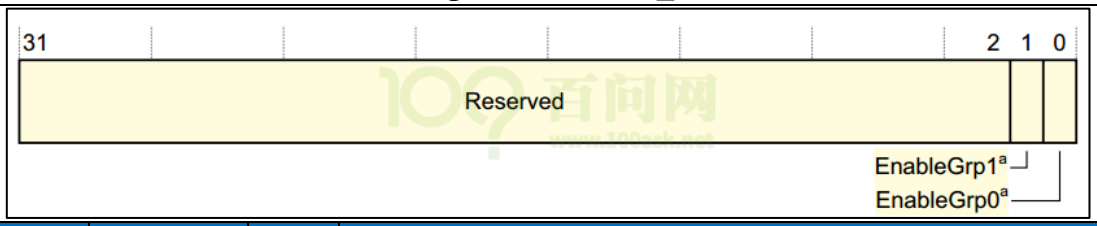
伪中断 ID 是保留值，不能分配给系统中的任何设备。

## 10.9 中断控制器寄存器

GIC 分为两部分：Distributor 和 CPU interface，它们的寄存器都有相应的前缀:“GICD_”、“GICC_”。这些寄存器都是映射为内存接口(memery map)，CPU 可以直接读写。

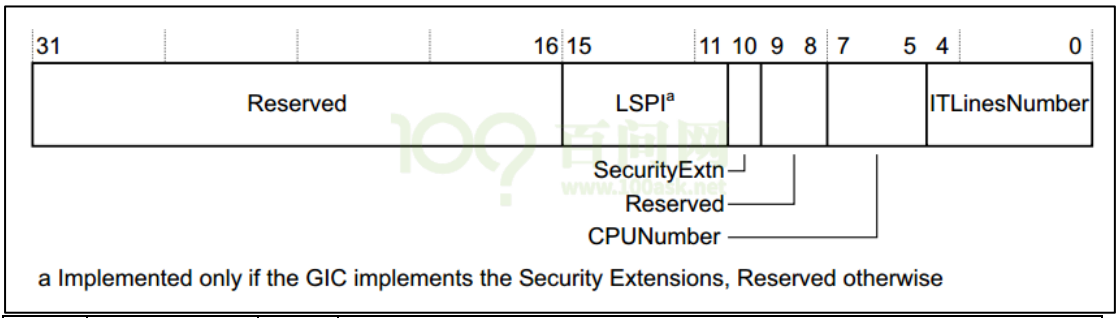
### 10.9.1 Distributor 寄存器描述

Distributor Control Register, GICD_CTLR




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| 1 | EnableGrp1 | R/W | 用于将 pending Group 1 中断从 Distributor 转发到 CPU interfaces<br>0：group 1 中断不转发<br>1：根据优先级规则转发 Group 1 中断 |
| 0 | EnableGrp0 | R/W | 用于将 pending Group 0 中断从 Distributor 转发到 CPU interfaces<br>0：group 0 中断不转发<br>1：根据优先级规则转发 Group 0 中断 |

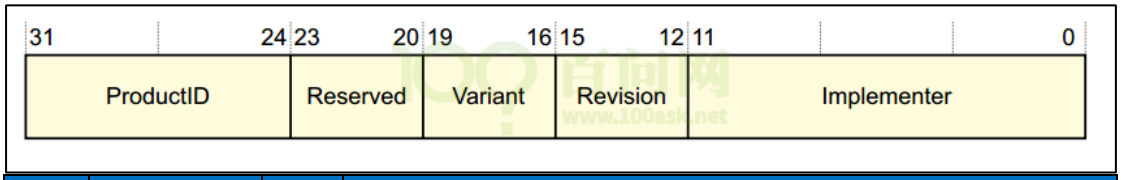
### 10.9.2 Interrupt Controller Type Register, GICD_TYPER




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|

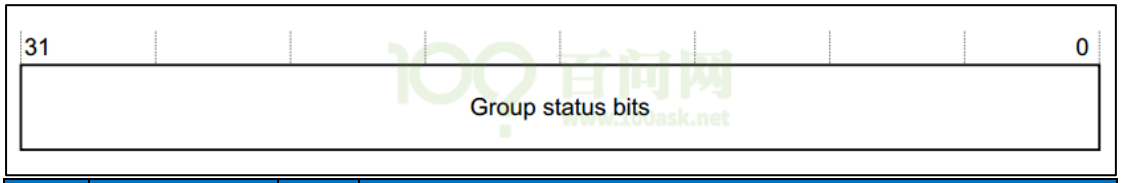
| | | | |
|---|---|---|---|
| 15:11 | LSPI | R | 如果 GIC 实现了安全扩展，则此字段的值是已实现的可锁定 SPI 的最大数量，范围为 0（0b00000）到 31（0b11111）。<br>如果此字段为 0b00000，则 GIC 不会实现配置锁定。<br>如果 GIC 没有实现安全扩展，则保留该字段。 |
| 10 | SecurityExtn | R | 表示 GIC 是否实施安全扩展：<br>0 未实施安全扩展；<br>1 实施了安全扩展 |
| 7:5 | CPUNumber | R | 表示已实现的 CPU interfaces 的数量。<br>已实现的 CPU interfaces 数量比该字段的值大 1。<br>例如，如果此字段为 0b011，则有四个 CPU interfaces。 |
| 4:0 | ITLinesNumber | R | 表示 GIC 支持的最大中断数。<br>如果 ITLinesNumber = N，则最大中断数为 32*(N+1)。<br>中断 ID 的范围是 0 到（ID 的数量– 1）。<br>例如：0b00011 最多 128 条中断线，中断 ID 0-127。<br>中断的最大数量为 1020（0b11111）。<br>无论此字段定义的中断 ID 的范围如何，都将中断 ID 1020-1023 保留用于特殊目的 |

### 10.9.3 Distributor Implementer Identification Register, GICD_IIDR




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| 31:24 | ProductID | R | 产品标识 ID |
| 23:20 | 保留 | | |
| 19:16 | Variant | R | 通常是产品的主要版本号 |
| 15:12 | Revision | R | 通常此字段用于区分产品的次版本号 |
| 11:0 | Implementer | R | 含有实现这个 GIC 的公司的 JEP106 代码；<br>[11:8]：JEP106 continuation code，对于 ARM 实现，此字段为 0x4；<br>[7]：始终为 0；<br>[6:0]：实现者的 JEP106code，对于 ARM 实现，此字段为 0x3B |

### 10.9.4 Interrupt Group Registers, GICD_IGROUPRn




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| 31:0 | Group status bits | R/W | 组状态位，对于每个位：<br>0：相应的中断为 Group 0；<br>1：相应的中断为 Group 1。 |

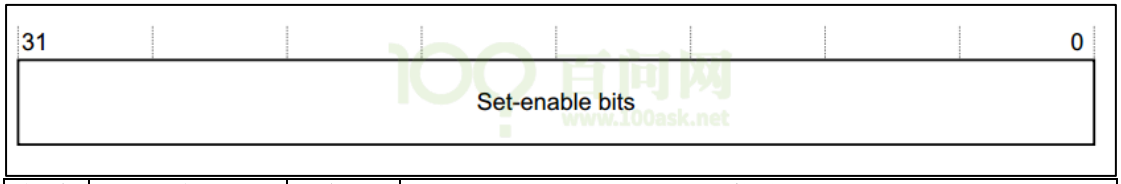
对于一个中断，如何设置它的 Group ？首先找到对应的 GICD_IGROUPRn 寄

存器，即 n 是多少？还要确定使用这个寄存器里哪一位。

对于 interrtups ID m，如下计算：

```
n = m DIV 32，GICD_IGROUPRn 里的 n 就确定了；
GICD_IGROUPRn 在 GIC 内部的偏移地址是多少？0x080+(4*n)
使用 GICD_IPRIORITYRn 中哪一位来表示 interrtups ID m？
bit = m mod 32。
```

## 10.9.5 Interrupt Set-Enable Registers, GICD_ISENABLERn

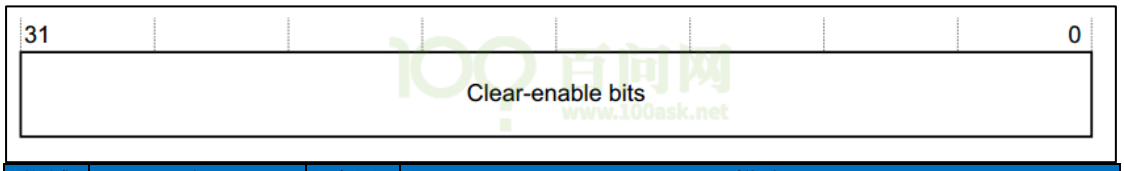



| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| 31:0 | Set-enable bits | R/W | 对于 SPI 和 PPI 类型的中断，每一位控制对应中断的转发行为：从 Distributor 转发到 CPU interface：<br>读：<br>0：表示当前是禁止转发的；<br>1：表示当前是使能转发的；<br>写：<br>0：无效<br>1：使能转发 |

对于一个中断，如何找到 GICD_ISENABLERn 并确定相应的位？

```
对于 interrtups ID m，如下计算：
n = m DIV 32，GICD_ISENABLERn 里的 n 就确定了；
GICD_ISENABLERn 在 GIC 内部的偏移地址是多少？0x100+(4*n)
使用 GICD_ISENABLERn 中哪一位来表示 interrtups ID m？
bit = m mod 32。
```

## 10.9.6 Interrupt Clear-Enable Registers, GICD_ICENABLERn




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| 31:0 | Clear-enable bits | R/W | 对于 SPI 和 PPI 类型的中断，每一位控制对应中断的转发行为：从 Distributor 转发到 CPU interface：<br>读：<br>0：表示当前是禁止转发的；<br>1：表示当前是使能转发的；<br>写：<br>0：无效<br>1：禁止转发 |

对于一个中断，如何找到 GICD_ICENABLERn 并确定相应的位？

```
对于 interrtups ID m，如下计算：
n = m DIV 32，GICD_ICENABLERn 里的 n 就确定了；
GICD_ICENABLERn 在 GIC 内部的偏移地址是多少？0x180+(4*n)
使用 GICD_ICENABLERn 中哪一位来表示 interrtups ID m？
bit = m mod 32。
```

## 10.9.7 Interrupt Set-Active Registers, GICD_ISACTIVERn


| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| 31:0 | Set-active bits | R/W | 读：<br>0：表示相应中断不是 active 状态；<br>1：表示相应中断是 active 状态；<br>写：<br>0：无效<br>1：把相应中断设置为 active 状态，如果中断已处于 Active 状态，则写入无效 |

对于一个中断，如何找到 GICD_ISACTIVERn 并确定相应的位？

**对于 interrtups ID m，如下计算：**
**n = m DIV 32，GICD_ISACTIVERn 里的 n 就确定了；**
**GICD_ISACTIVERn 在 GIC 内部的偏移地址是多少？0x300+(4*n)**
**使用 GICD_ISACTIVERn 中哪一位来表示 interrtups ID m？**
**bit = m mod 32。**

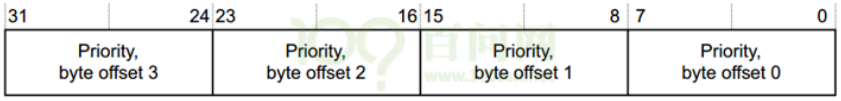
## 10.9.8 Interrupt Clear-Active Registers, GICD_ICACTIVERn


| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| 31:0 | Clear-active bits | R/W | 读：<br>0：表示相应中断不是 active 状态；<br>1：表示相应中断是 active 状态；<br>写：<br>0：无效<br>1：把相应中断设置为 deactive 状态，如果中断已处于 dective 状态，则写入无效 |

对于一个中断，如何找到 GICD_ICACTIVERn 并确定相应的位？

**对于 interrtups ID m，如下计算：**
**n = m DIV 32，GICD_ICACTIVERn 里的 n 就确定了；**
**GICD_ICACTIVERn 在 GIC 内部的偏移地址是多少？0x380+(4*n)**
**使用 GICD_ICACTIVERn 中哪一位来表示 interrtups ID m？**
**bit = m mod 32。**

## 10.9.9 Interrupt Priority Registers, GICD_IPRIORITYRn




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| 31:24 | Priority, byte offset 3 | R/W | 对于每一个中断，都有对应的 8 位数据用来描述：它的优先级。<br>每个优先级字段都对应一个优先级值，值越小，相应中断的优先级越高 |
| 23:16 | Priority, byte offset 2 | R/W | |
| 15:8 | Priority, byte offset 1 | R/W | |
| 7:0 | Priority, byte offset 0 | R/W | |

对于一个中断，如何设置它的优先级(Priority)，首先找到对应的 GICD_IPRIORITYRn 寄存器，即 n 是多少？还要确定使用这个寄存器里哪一个字节。

```
对于 interrtups ID m，如下计算：
n = m DIV 4，GICD_IPRIORITYRn 里的 n 就确定了；
GICD_IPRIORITYRn 在 GIC 内部的偏移地址是多少？0x400+(4*n)
使用 GICD_IPRIORITYRn 中 4 个字节中的哪一个来表示 interrtups ID m 的优先级？
byte offset = m mod 4。
byte offset 0 对应寄存器里的[7:0]；
byte offset 1 对应寄存器里的[15:8]；
byte offset 2 对应寄存器里的[23:16]；
byte offset 3 对应寄存器里的[31:24]。
```

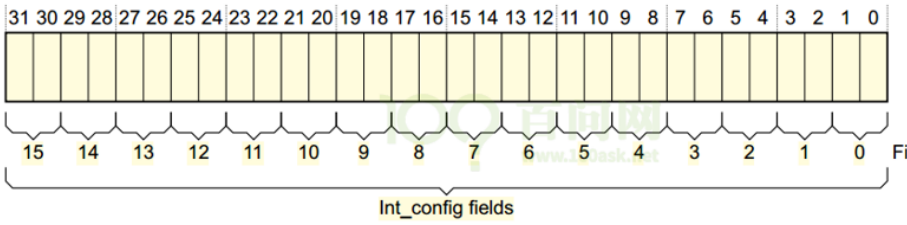
## 10.9.10 Interrupt Processor Targets Registers, GICD_ITARGETSRn


| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| 31:24 | CPU targets, byte offset 3 | R/W | 对于每一个中断，都有对应的 8 位数据用来描述：这个中断可以发给哪些 CPU。<br>处理器编号从 0 开始，8 位数里每个位均指代相应的处理器。<br>例如，值 0x3 表示将中断发送到处理器 0 和 1。<br>当读取 GICD_ITARGETSR0～GICD_ITARGETSR7 时，读取里面任意字节，返回的都是执行这个读操作的 CPU 的编号。 |
| 23:16 | CPU targets, byte offset 2 | R/W | |
| 15:8 | CPU targets, byte offset 1 | R/W | |
| 7:0 | CPU targets, byte offset 0 | R/W | |

对于一个中断，如何设置它的目标 CPU？优先级(Priority)，首先找到对应的 GICD_ITARGETSRn 寄存器，即 n 是多少？还要确定使用这个寄存器里哪一个字节。

```
对于 interrtups ID m，如下计算：
n = m DIV 4，GICD_ITARGETSRn 里的 n 就确定了；
GICD_ITARGETSRn 在 GIC 内部的偏移地址是多少？0x800+(4*n)
使用 GICD_ITARGETSRn 中 4 个字节中的哪一个来表示 interrtups ID m 的目标 CPU？
byte offset = m mod 4。
byte offset 0 对应寄存器里的[7:0]；
byte offset 1 对应寄存器里的[15:8]；
byte offset 2 对应寄存器里的[23:16]；
byte offset 3 对应寄存器里的[31:24]。
```

## 10.9.11 Interrupt Configuration Registers, GICD_ICFGRn



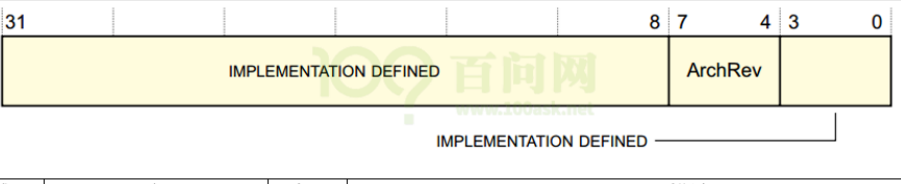

See the bit assignment table for more information about the properties of each Int_config[1:0] field.

| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [2F+1:2F] | Int_config, field F | R/W | 对于每一个中断，都有对应的 2 位数据用来描述：它的边沿触发，还是电平触发。<br>对于 Int_config[1]，即高位[2F+1]，含义为：<br>0：相应的中断是电平触发；<br>1：相应的中断是边沿触发。<br><br>对于 Int_config[0]，即低位[2F]，是保留位。 |

对于一个中断，如何找到 GICD_ICFGRn 并确定相应的位域 F？

**对于 interrtups ID m，如下计算：**
**n = m DIV 16，GICD_ICFGRn 里的 n 就确定了；**
**GICD_ICACTIVERn 在 GIC 内部的偏移地址是多少？0xC00+(4*n)**
**F = m mod 16。**

## 10.9.12 Identification registers: Peripheral ID2 Register, ICPIDR2




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:0] | - | R/W | 由实现定义 |
| [7:4] | ArchRev | R | 该字段的值取决于 GIC 架构版本：<br>0x1：GICv1；<br>0x2：GICv2。 |
| [3:0] | - | R/W | 由实现定义 |

## 10.9.13 CPU interface 寄存器描述

CPU Interface Control Register, GICC_CTLR

此寄存器用来控制 CPU interface 传给 CPU 的中断信号。对于不同版本的 GIC，这个寄存器里各个位的含义大有不同。以 GICv2 为例，有如下 2 种格式：

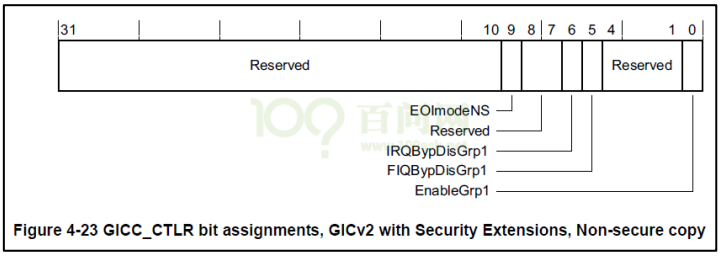



Figure 4-23 GICC_CTLR bit assignments, GICv2 with Security Extensions, Non-secure copy

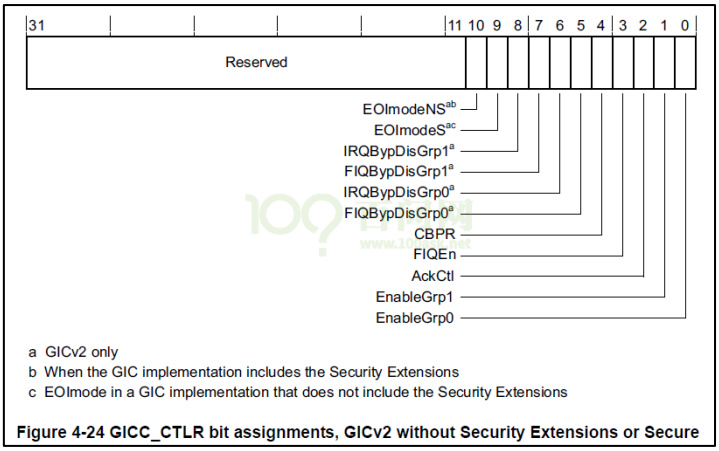



**Figure 4-24 GICC_CTLR bit assignments, GICv2 without Security Extensions or Secure**

以“GIC2 with Security Extensions, Non-secure copy”为例，GICC_CTLR 中各个位的定义如下：

| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:10] | - | | 保留 |
| [9] | EOImodeNS | R/W | 控制对 GICC_EOIR 和 GICC_DIR 寄存器的非安全访问：<br>0：GICC_EOIR 具有降低优先级和 deactivate 中断的功能；<br>对 GICC_DIR 的访问是未定义的。<br>1：GICC_EOIR 仅具有降低优先级功能；<br>GICC_DIR 寄存器具有 deactivate 中断功能。 |
| [8:7] | - | | 保留 |
| [6] | IRQBypDisGrp1 | R/W | 当 CPU interface 的 IRQ 信号被禁用时，该位控制是否向处理器发送 bypass IRQ 信号：<br>0：将 bypass IRQ 信号发送给处理器；<br>1：将 bypass IRQ 信号不发送到处理器。 |
| [5] | FIQBypDisGrp1 | R/W | 当 CPU interface 的 FIQ 信号被禁用时，该位控制是否向处理器发送 bypass FIQ 信号：<br>0：将 bypass FIQ 信号发送给处理器；<br>1：旁路 FIQ 信号不发送到处理器 |
| [4:1] | - | | 保留 |
| [0] | - | R/W | 使能 CPU interface 向连接的处理器发出的组 1 中断的信号:<br>0：禁用中断信号<br>1：使能中断信号 |

## 10.9.14 Interrupt Priority Mask Register, GICC_PMR

提供优先级过滤功能，优先级高于某值的中断，才会发送给 CPU。


| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:8] | - | | 保留 |
| [7:0] | - | R/W | 优先级高于这个值的中断，才会发送给 CPU |

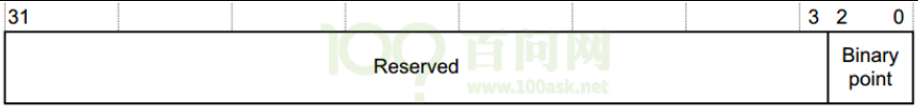
[7:0]共 8 位，可以表示 256 个优先级。但是某些芯片里的 GIC 支持的优先级少于 256 个，则某些位为 RAZ / WI，如下所示：

- 如果有 128 个级别，则寄存器中 bit[0] = 0b0，即使用[7:1]来表示优先级；

- 如果有 64 个级别，则寄存器中 bit[1:0] = 0b00，即使用[7:2]来表示优先级；
- 如果有 32 个级别，则寄存器中 bit[2:0] = 0b000，即使用[7:3]来表示优先级；
- 如果有 16 个级别，则寄存器中 bit[3:0] = 0b0000，即使用[7:4]来表示优先级；
- 注意：imx6ull 最多为 32 个级别

### 10.9.15 Binary Point Register, GICC_BPR

此寄存器用来把 8 位的优先级字段拆分为组优先级和子优先级，组优先级用来决定中断抢占。




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:3] | - | | 保留 |
| [2:0] | Binary point | R/W | 此字段的值控制如何将 8bit 中断优先级字段拆分为组优先级和子优先级，组优先级用来决定中断抢占。<br>更多信息还得看看 GIC 手册。 |

### 10.9.16 Interrupt Acknowledge Register, GICC_IAR

CPU 读此寄存器，获得当前中断的 interrtup ID。


| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:13] | - | | 保留 |
| [12:10] | CPUID | R | 对于 SGI 类中断，它表示谁发出了中断。例如，值为 3 表示该请求是通过对 CPU interface 3 上的 GICD_SGIR 的写操作生成的。 |
| [9:0] | Interrupt ID | R | 中断 ID |

### 10.9.17 Interrupt Register, GICC_EOIR

写此寄存器，表示某中断已经处理完毕。GICC_IAR 的值表示当前在处理的中断，把 GICC_IAR 的值写入 GICC_EOIR 就表示中断处理完了。


| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:13] | - | | 保留 |
| [12:10] | CPUID | W | 对于 SGI 类中断，它的值跟 GICD_IAR.CPUID 的相同。 |
| [9:0] | EOIINTID | W | 中断 ID，它的值跟 GICD_IAR 里的中断 ID 相同 |

# 第11章 GPIO 中断

## 11.1 GPIO 中断介绍(通用的概念)

假设你现在正在写作业，突然电话响起，你需要停下写作业接电话，挂电话后继续写作业。突然有人按门铃，你需要先去开门，然后继续回来写作业。

电话和门铃打断了写作业，能中断写作业的事情有很多，比如身体不舒服，口渴等。被打断后怎么做？身体不舒服就停下休息一会，身体好了继续写作业。口渴就停下喝水，喝完水继续写作业。

如果你写作业中途去接听了电话，这时突然门铃响了，这时要优先处理其中一件事：继续打电话呢，还是让打电话的人稍等一会让你去开门。这就存在一个中断优先级的问题。

当有事件产生时，处理事件之前：

① 我们需要记住现在作业写到第几页了，比如放个书签；

② 再去取处理事件：电话铃响了需要到放电话的地方去，门铃响了需要到门口去，口渴需要到放饮水机地方去，也就是说，不同的突发事件需要到不同的地方去处理；

③ 处理完突发事件后，继续写作业。

嵌入式系统中也有类似的情况，CPU 在运行过程中，也会被各种异常打断。这些异常有

① 指令未定义

② 指令、数据访问有问题

③ SWI（软中断）

④ 快中断

⑤ 中断

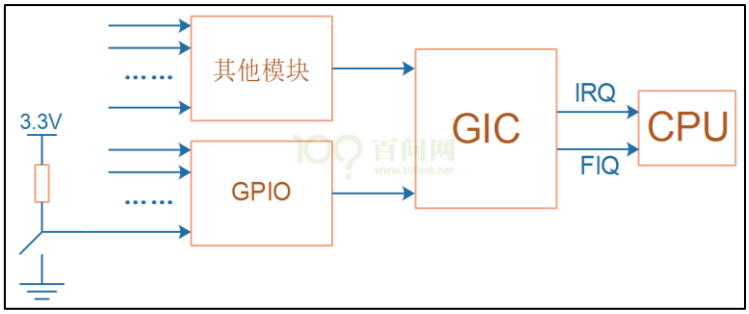
中断也属于一种异常，导致中断发生的中断源有很多，比如：

- 按键
- 定时器
- ADC 转换完成
- UART 发生完数据、接收数据等等

这些众多的中断源，汇集于中断管理器，由中断管理器选择优先级最高的中断并通知 CPU。CPU 会根据中断的类型到跳转到不同的地址处理中断。发生中断后，CPU 并不是随便跳到一个地址处理中断，而是根据异常向量表，跳转到对应的地址处理中断。

### 11.1.1 GPIO 中断

GPIO 中断，指由 GPIO 模块产生的中断，有边沿触发中断或者电平翻转中断。GPIO 模块能检测到引脚上电平的变化，并向中断控制器(GIC)发出中断信号，GIC 再向 CPU 发出中断信号。框架如下图：




CPU 在每执行完了条指令时会检查是否发生了中断，若是则会跳转到中断处理地址进行中断处理。为了避免破坏主任务数据，CPU 会处理保存当前相关寄存器（保存现场）并进入中断服务函数，执行完中断服务函数后，CPU 会恢复相关寄存器（恢复现场），回到主任务继续执行程序。

程序发生 GPIO 中断后会根据异常向量表强制跳转到 0x18（IRQ 中断地址）。如下图：

**Vector tables**

| Offset | Hyp[a] | Monitor[b] | Secure | Non-secure |
|---|---|---|---|---|
| 0x00 | Not used | Not used | Reset | Not used |
| 0x04 | Undefined Instruction, from Hyp mode | Not used | Undefined Instruction | Undefined Instruction |
| 0x08 | Hypervisor Call, from Hyp mode | Secure Monitor Call | Supervisor Call | Supervisor Call |
| 0x0C | Prefetch Abort, from Hyp mode | Prefetch Abort | Prefetch Abort | Prefetch Abort |
| 0x10 | Data Abort, from Hyp mode | Data Abort | Data Abort | Data Abort |
| 0x14 | Hyp Trap, or Hyp mode entry[c] | Not used | Not used | Not used |
| 0x18 | IRQ interrupt | IRQ interrupt | IRQ interrupt | IRQ interrupt |
| 0x1C | FIQ interrupt | FIQ interrupt | FIQ interrupt | FIQ interrupt |

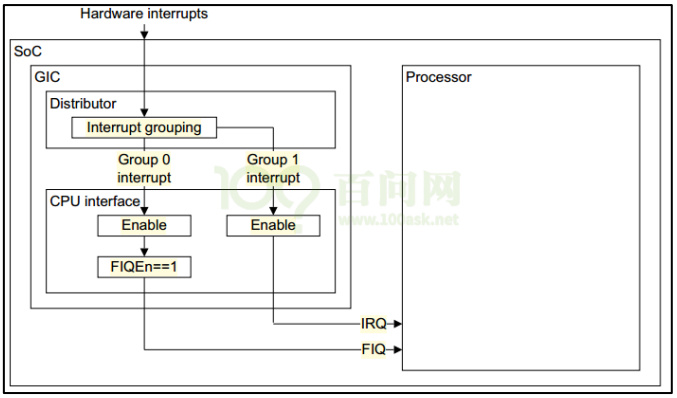
a. Non-secure state only. Implemented only if the implementation includes the Virtualization Extensions.
b. Secure state only. Implemented only if the implementation includes the Security Extensions.
c. See *Use of offset 0x14 in the Hyp vector table* on page B1-1168.

异常向量表并不总是从 0 地址开始，IMX6ULL 可以设置 vector base 寄存器，指定向量表在其他位置，比如设置 vector base 为 0x80000000，指定为 DDR 的某个地址。但是表中的各个异常向量的偏移地址，是固定的：复位向量偏移地址是 0，中断是 0x18。

本次实验使用 GPIO 中断方式实现按键控制 LED 亮灭，并通过串口把中断 ID 打印出来。

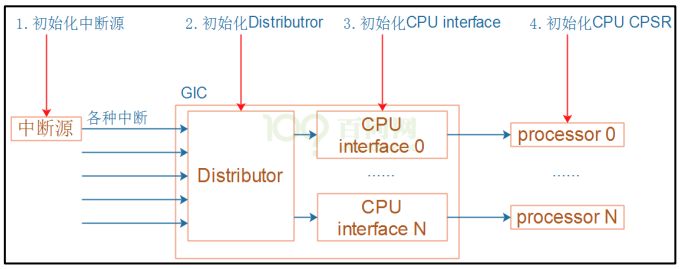
## 11.1.2 IMX6ULL GIC 中断控制器功能概述

IMX6ULL 是 Cortex-A7 内核，采用 GIC V2（Generic Interrupt Controller）中断控制器。在这里只简单的介绍一下 GIC，具体可以参考上一课或是 ARM 文档。

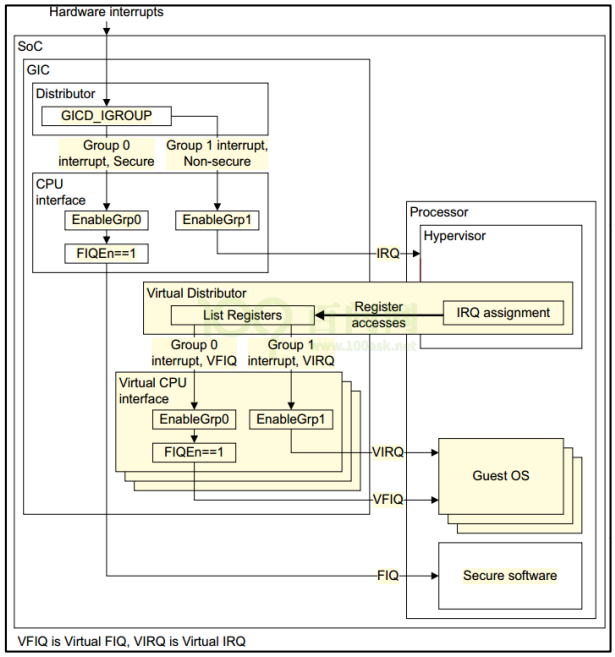



如上图所示，硬件中断信号发送到 GIC（Generic Interrupt Controller），GIC 产生一个 FIQ 或 IRQ 信号给 CPU。GPIO 模块、UART 模块均能产生硬件中断。

在初始化中断时，要初始化这 4 部分：产生中断的源头(GPIO 模块或 UART 模块等)、GIC(内部有 Distributor 或 CPU interface)、CPU 本身(设置 CPSR 寄存器)，如下图：

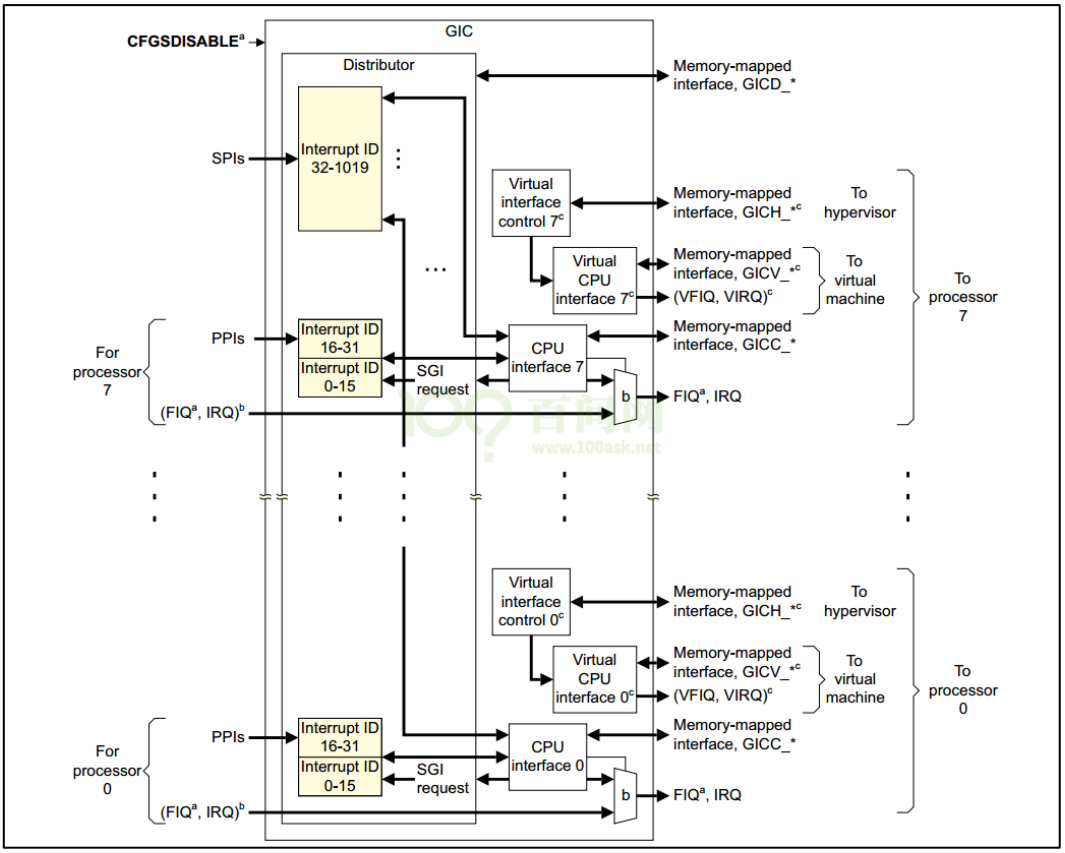



GIC 的主要作用可以归结为接受硬件中断信号，并进行简单的处理，按照一定的设置策略，分给对应的 CPU 处理。如下图：



GIC 可以通过四个信号向 CPU 核汇报中断情况：VIRQ（虚拟快速 IRQ）、VFIQ（虚拟快速 FIQ）、IRQ、FIQ。VIRQ、VFIQ 是针对虚拟化，剩下就是 IRQ 和 FIQ。GPIO 中断属于 IRQ 中断，所以在本次实验中 GIC 上报 IRQ 信号给 ARM 内核。

接下来看一下 GIC 内部过程，如下图：




中断源分为 SPI（Shared Peripheral Interrupt）、PPI（Private Peripheral Interrupt）、SGI request（Software-generated Interrupt）。外部中断都属于 SPI 中断源。

GIC 控制器包括分发器（Distributor）和 CPU 接口端（CPU interface）。

分发器（Distributor）主要完成对整个中断控制器使能，设置中断优先级，设置中断触发方式，决定每个中断信号发送到哪一个具体的 CPU 上执行。

CPU 接口端（CPU interface）主要完成使能和发送一个具体的中断信号到特定的 CPU 上，确认中断已被 CPU 接受、处理以及处理完成，设置 CPU 能接受中断的优先级以及基于级别的中断抢占。

中断信号先到达分发器，分发器根据该中断所设定的 CPU，把中断发送到 CPU 对应的 CPU interface 上；在 CPU interface 里判断该中断的优先级是否足够高，能否抢断或打断当前的中端处理，如果可以，CPU interface 就会发送一个物理的 signa 到 CPU 的 IRQ 线上；CPU 接收到中断信号，转到中断处理模式进行处理。

### 11.1.3 IMX6ULL GIC 中断寄存器

GIC 寄存器分为 Distributor register 和 CPU interface register。寄存器数目较多，这里介绍本次实验中需要我们设置的寄存器。

Distributor 的寄存器名字中有“GICD_”前缀，CPU interface 的寄存器名字中有“GICC_”前缀。

➢ **GICC_IAR 寄存器**

GICC_IAR 寄存器属于 CPU interface register，作用是：保存中断 ID，读取 GICC_IAR 寄存器可以获得中断 ID，这个过程可以当作对中断的确认。


| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:13] | - | | 保留 |
| [12:10] | CPUID | R | 对于 SGI 类中断，它表示谁发出了中断。例如，值为 3 表示该请求是通过对 CPU interface 3 上的 GICD_SGIR 的写操作生成的。 |
| [9:0] | Interrupt ID | R | 中断 ID |

➢ **GICC_EOIR 寄存器**

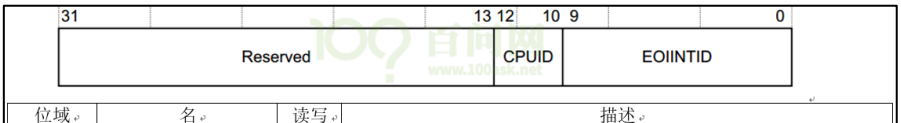
**End of Interrupt Register, GICC_EOIR**

The GICC_EOIR characteristics are:

**Purpose** A processor writes to this register to inform the CPU interface either:

- that it has completed the processing of the specified interrupt
- in a GICv2 implementation, when the appropriate GICC_CTLR.EOImode bit is set to 1, to indicate that the interface should perform priority drop for the specified interrupt.

写此寄存器，表示某中断已经处理完毕。GICC_IAR 的值表示当前在处理的中断，把 GICC_IAR 的值写入 GICC_EOIR 就表示中断处理完了。




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:13] | - | | 保留 |
| [12:10] | CPUID | W | 对于 SGI 类中断，它的值跟 GICD_IAR.CPUID 的相同。 |
| [9:0] | EOIINTID | W | 中断 ID，它的值跟 GICD_IAR 里的中断 ID 相同 |

## 11.1.4 CP15 协处理器

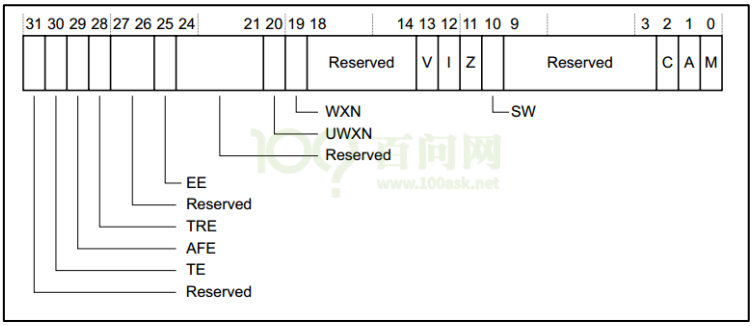
### 1 CP15 协处理器介绍

在基于 ARM 的嵌入式系统中，存储系统通常是协处理器 CP15 完成的。ARM 处理器使用协处理器指令 MCR 和 MRC 来读写寄存器，控制 cache、MMU、配置时钟（在 bootloader 时钟初始化时会用到）等。CP15 包含 16 个 32 位寄存器，编号为 0~15。

在本次实验中，需要设置的寄存器有：SCTLR（System Control Register）寄存器，VBAR（Vector Base Address）寄存器。

➢ **SCTLR（System Control Register）寄存器**

设置 SCTLR 寄存器可以控制 cache、MMU 等。



| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [13] | V | R/W | 向量位，用来设置向量表的基地址。<br>0：Low exception vectors，基地址为 0；如果有 VBAR 寄存器，则使用它来指定向量表基地址；<br>1：High exception vectors，向量表基地址为 0xFFFF0000 |
| [12] | I | R/W | 指令 cache 使能位<br>0：指令 cache 禁止，这是默认值<br>1：指令 cache 使能 |
| [11] | Z | RAO/WI | 分支预测使能位，当 MMU 使能时该位自动使能 |
| [2] | C | R/W | 数据 cache 使能位<br>0：数据 cache 禁止，这是默认值<br>1：数据 cache 使能 |
| [1] | A | R/W | 字节对齐设置位，<br>0：地址对齐检查禁止，这是默认值<br>1：地址对齐检查使能 |
| [0] | M | R/W | MMU 使能位，<br>0：MMU 禁止，这是默认值；<br>1：MMU 使能 |

可以使用以下指令读写 SCTLR 寄存器：

```
MRC p15, 0, <Rt>, c1, c0, 0 ;把 SCTLR 寄存器的值读到 ARM 寄存器 Rt 中。
MRC p15, 0, <Rt>, c1, c0, 0 ;把 ARM 寄存器 Rt 的值写入 SCTLR 寄存器。
```

### ➢ VBAR（Vector Base Address）寄存器

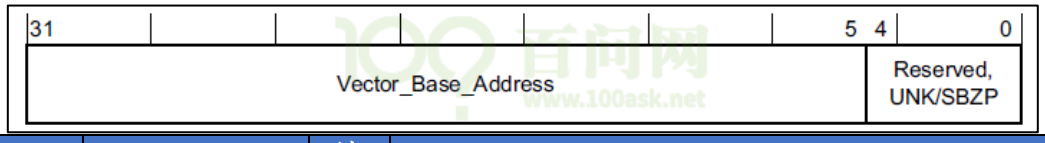
通过 VBAR 寄存器，可以设置异常向量表的映射地址。如果不把异常向量表的映射地址告诉 CPU，在发生异常时，CPU 就找不到异常向量表，就无法处理异常。

**VBAR, Vector Base Address Register, Security Extensions**

The VBAR characteristics are:

**Purpose** When high exception vectors are not selected, the VBAR holds the exception base address for exceptions that are not taken to Monitor mode or to Hyp mode, see *Exception vectors and the exception base address* on page B1-1165.

This register is part of the Security Extensions registers functional group.



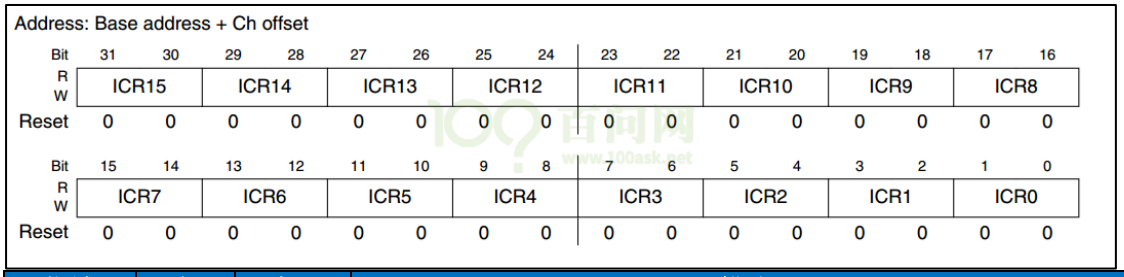

| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:5] | Vector_Base_Address | R/W | 向量表基地址的 b[31:5]，这也意味着这个地址的低 5 位全为 0 |

```
MRC p15, 0, <Rt>, c12, c0, 0 ;把 VBAR 寄存器的值读到 ARM 寄存器 Rt 中。
MRC p15, 0, <Rt>, c12, c0, 0 ;把 ARM 寄存器 Rt 的值写入 VBAR 寄存器。
```

# 11.2 IMX6ULL 的 GPIO 中断寄存器介绍

## 11.2.1 GPIO interrupt configuration register1 (GPIOx_ICR1)

GPIO 中断配置寄存器 1，用来配置 GPIO 中断 1～15 的触发类型。

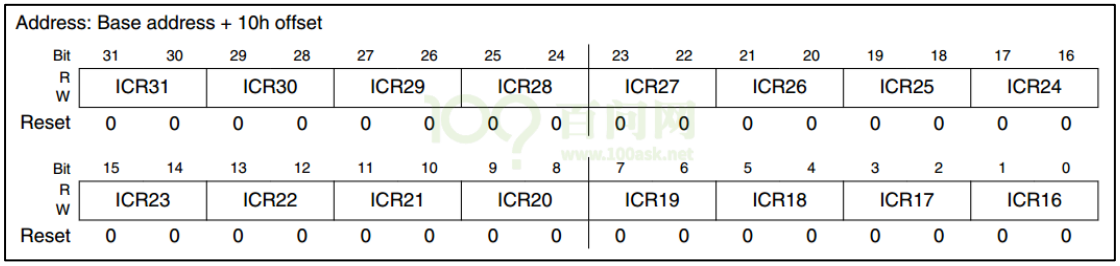



| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [2n+1:2n] | ICRn | R/W | 用来设置 GPIO 中断的触发类型，<br>00：低电平触发；<br>01：高电平触发；<br>10：上升沿触发；<br>11：下降沿触发 |

ICR0~ICR15 对应 GPIO interrupt 0-15。

## 11.2.2 GPIO interrupt configuration register2 (GPIOx_ICR2)

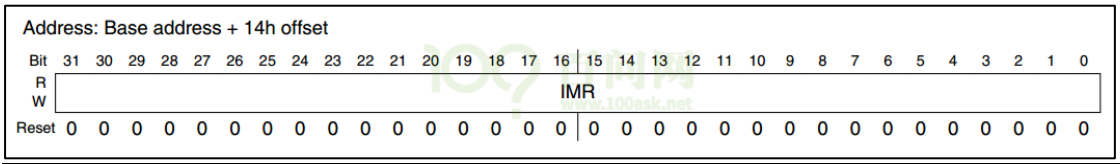
GPIO 中断配置寄存器 2，用来配置 GPIO 中断 16～31 的触发类型。




与 GPIOx_ICR1 类似，ICR15~ICR31 对应 GPIO interrupt 16-31。

## 11.2.3 GPIO interrupt mask register (GPIOx_IMR)

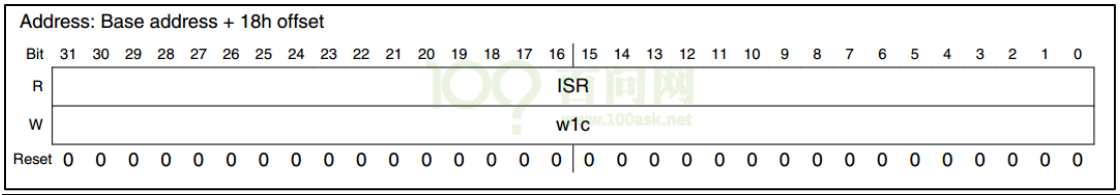
GPIO 中断屏蔽寄存器，用来屏蔽或使能某个 GPIO 中断。




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [n] | IMR | R/W | 每一位对应一个 GPIO 中断，<br>0：中断被屏蔽<br>1：中断使能，未被屏蔽 |

## 11.2.4 GPIO interrupt status register (GPIOx_ISR)

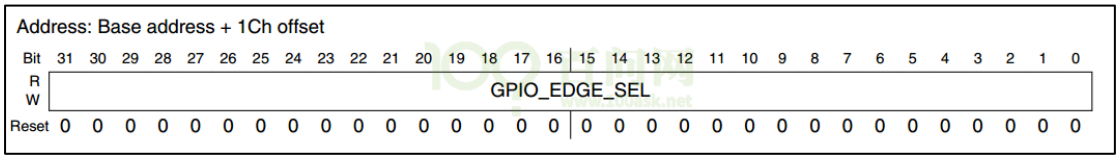
GPIO 中断状态寄存器，表示某个 GPIO 中断是否发生了。




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [n] | ISR | R/W | 每一位对应一个 GPIO 中断，跟 GPIO_IMR 无关，就是说即使屏蔽了某个中断，还是可以在本寄存器中观察它的状态。<br>读：<br>0：中断未发生；<br>1：中断已发生。<br>写：某位写入 1 时，清零该位。 |

### 11.2.5 GPIO edge select register (GPIOx_EDGE_SEL)

GPIO 中断边沿选择寄存器，它可以用来覆盖 GPIOx_ICR1/2 中的配置值。




每一位对应一个 GPIO 中断，一旦设置了 GPIO_EDGE_SEL[n]时，GPIO 会忽略 ICR [n]设置，GPIO interrupt n 的触发类型就是双边沿触发。

## 11.3 按键中断程序编程示例

100ASK_IMX6ULL 有 2 个按键，本节程序将设置它们的中断处理函数，在按键被按下或松开时进行打印；另外，还可以用 KEY1 来操作 LED。

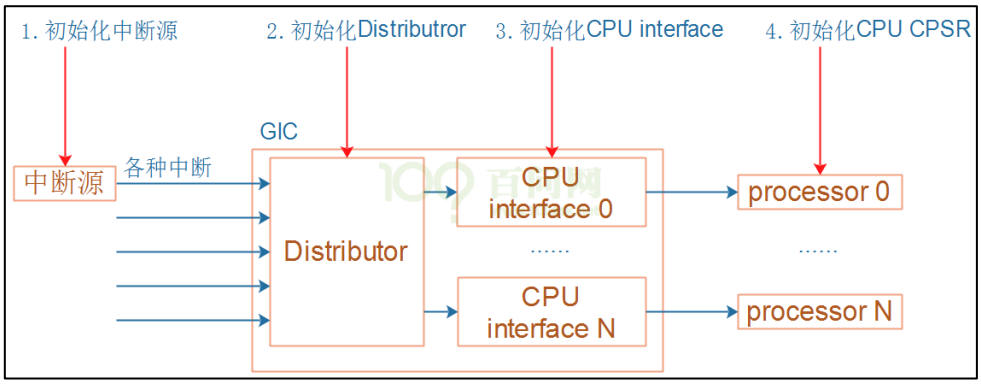
程序的总体流程是：

① 在中断向量中，保存现场，调用处理函数，恢复现场；

② 初始化：为 KEY1、KEY2 设置处理函数；初使化 GPIO 模块、初始化 GIC；

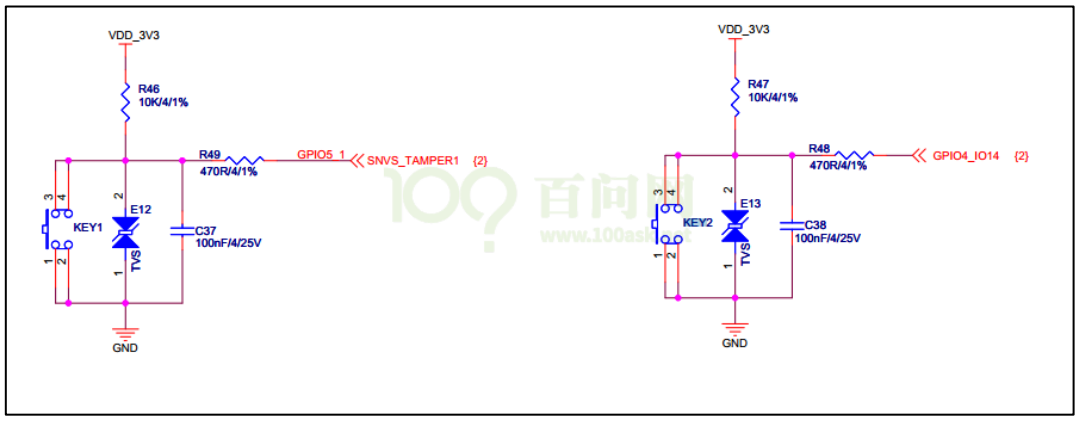
③ 准备好一切之后，使能中断。

阅读代码时，建议按照下图来理解：




代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/11_GPIO 中断/001_exception”目录下。

### 11.3.1 管脚设置和查询中断号




从上面的电路图可见 KEY1 接在 GPIO5_1（SNVS_TAMPER1 pad，ALT5）上，KEY4 接在 GPIO4_14（NAND_CE1_B pad，ALT5）上。

程序中使用 IOMUXC_SetPinMux 函数设置这两个引脚为 GPIO 模式。

如何获取这两个 GPIO 的中断号呢？查阅数据手册的《chapter3，CORTEX A7interrupts》章节，这两个 GPIO 的中断号如下表所示。对应到 GIC 的 SPI 中断号需要在此编号基础上加上 32，所以 KEY1 对应的 GIC interrupt ID 为（74 + 32 = 106），KEY2 对应的 GIC interrupt ID 为（72 + 32 = 104）。

| | | | |
|---|---|---|---|
| 72 | gpio4 | - | Combined interrupt indication for GPIO4 signal 0 throughout 15 |
| 73 | gpio4 | - | Combined interrupt indication for GPIO4 signal 16 throughout 31 |
| 74 | gpio5 | - | Combined interrupt indication for GPIO5 signal 0 throughout 15 |

- 当发生 GIC 104 号中断时，表示发生了 GPIO4 中 interrupt 0～15，需要进一步细分出是 GPIO4 里的哪一个中断。

- 当发生 GIC 106 号中断时，表示发生了 GPIO5 中 interrupt 0～15，需要进一步细分出是 GPIO5 里的哪一个中断。

### 11.3.2 GIC 控制器基地址的获取方法

直接查数据手册 Table 2-1. System memory map，可以知道 gic 的基地址是 0xA0000，如下图：

| | | | |
|---|---|---|---|
| 00A0_0000 | 00A0_7FFF | 32 KB | ARM Peripherals: GIC400 Only visible to ARM core(s) |

对于 GIC 基地址，还可以通过 CP15 查询，下面指令将 GIC 的基地址读到 r0 寄存器：

```
mrc p15, 4, r0, c15, c0, 0
```

### 11.3.3 GIC 的初始化

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/11_GPIO 中断/001_exception/gic.c”。

gic_init 函数实现了如下功能：

① 通过 CP15 获取 GIC 的基地址，

② 读取 GICD_TYPER 寄存器获得中断的数目，

③ 往 GICD_ ICENABLERn 寄存器写入 0xFFFFFFFF 禁用所有的 SGI，PPI 和 SPI；
④ 通过 GICC_PMR 设置优先级等级，设置为 0xF8；
⑤ 将 GICC_BPR 设置为 2，这允许各个优先级进行抢占；
⑥ 最后使能 group0 的 distributor 和 CPU interface。

```
void gic_init(void)
{
    u32 i, irq_num;

    GIC_Type *gic = get_gic_base();

    /* the maximum number of interrupt IDs that the GIC supports */
    irq_num = (gic->D_TYPER & 0x1F) + 1;

    /* On POR, all SPI is in group 0, level-sensitive and using 1-N model */

    /* Disable all PPI, SGI and SPI */
    for (i = 0; i < irq_num; i++)
      gic->D_ICENABLER[i] = 0xFFFFFFFFUL;

    /* The priority mask level for the CPU interface. If the priority of an
     * interrupt is higher than the value indicated by this field,
     * the interface signals the interrupt to the processor.
     */
    gic->C_PMR = (0xFFUL << (8 - 5)) & 0xFFUL;

    /* No subpriority, all priority level allows preemption */
    gic->C_BPR = 7 - 5;

    /* Enables the forwarding of pending interrupts from the Distributor to the CPU
interfaces.
     * Enable group0 distribution
     */
    gic->D_CTLR = 1UL;

    /* Enables the signaling of interrupts by the CPU interface to the connected
processor
     * Enable group0 signaling
     */
    gic->C_CTLR = 1UL;
}
```

## 11.3.4 中断异常处理汇编部分

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/11_GPIO 中断/001_exception/start.S”。

start.S 中对于中断的处理，概括如下：

- 在异常向量表偏移为 0x18 的地方使用“ldr pc, =IRQ_Handler”跳转；
- IRQ_Handler 标号的处理可以简单分为：保存现场，执行 C 函数，恢复现场：

在 IRQ_Handler 标号，处理器处于中断模式，“lr_irq - 4”就是被中断的、尚未执行的指令的地址，我们将 r0-r12 和“lr-4”都保存在栈上。

然后调用 C 函数 handle_irq_c 来处理中断。

C 函数返回来后，执行“ldmia sp!, {r0-r12, pc}^”，这条指令做的事情可多了：

- 把保存在栈上的值恢复到 r0-r12，把之前保存的“lr_irq - 4”恢复到 PC
- 同时，把 SPSR 中保存的被中断状态的 CPSR，恢复到 CPSR(指令后的“^”号表示这个操作)

这样，被中断的程序就继续运行了。start.S 代码如下：

```
.text
.global  _start, _vector_table
_start:
_vector_table:
    ldr pc, =Reset_Handler            /* Reset                   */
    ldr pc, =Undefined_Handler        /* Undefined instructions */
    ldr pc, =SVC_Handler          /* Supervisor Call     */
    b halt//ldr  pc, =PrefAbort_Handler        /* Prefetch abort          */
    b halt//ldr  pc, =DataAbort_Handler        /* Data abort              */
    .word    0                           /* RESERVED            */
    ldr pc, =IRQ_Handler          /* IRQ interrupt       */
    b halt//ldr  pc, =FIQ_Handler          /* FIQ interrupt       */
……
.align 2
IRQ_Handler:
    /* 执行到这里之前:
     * 1. lr_irq 保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_irq 保存有被中断模式的 CPSR
     * 3. CPSR 中的 M4-M0 被设置为 10010，进入到 irq 模式
     * 4. 跳到 0x18 的地方执行程序
     */

    /* 保存现场 */
    /* 在 irq 异常处理函数中有可能会修改 r0-r12，所以先保存 */
    /* lr-4 是异常处理完后的返回地址，也要保存 */
    sub lr, lr, #4
    stmdb sp!, {r0-r12, lr}

    /* 处理 irq 异常 */
    bl handle_irq_c

    /* 恢复现场 */
    ldmia sp!, {r0-r12, pc}^  /* ^会把 spsr_irq 的值恢复到 cpsr 里 */
.align 2
Reset_Handler:
    /* Reset SCTlr Settings */
    mrc p15, 0, r0, c1, c0, 0  /* read SCTRL, Read CP15 System Control register */
    bic r0,  r0, #(0x1 << 13)  /* Clear V bit 13 to use normal exception vectors */
    bic r0,  r0, #(0x1 << 12)  /* Clear I bit 12 to disable I Cache */
    bic r0,  r0, #(0x1 <<  2)  /* Clear C bit  2 to disable D Cache */
    bic r0,  r0, #(0x1 << 2)   /* Clear A bit  1 to disable strict alignment*/
    bic r0,  r0, #(0x1 << 11)  /* Clear Z bit 11 to disable branch prediction */
    bic r0,  r0, #0x1              /* Clear M bit  0 to disable MMU */
    mcr p15, 0, r0, c1, c0, 0  /* write SCTRL, CP15 System Control register */

   cps     #0x1B              /* Enter undef mode             */
```

```
    ldr     sp, =0x80300000     /* Set up undef mode stack      */

    cps     #0x12               /* Enter irq mode               */
    ldr     sp, =0x80400000     /* Set up irq mode stack      */

    cps     #0x13               /* Enter Supervisor mode         */
    ldr     sp, =0x80200000     /* Set up Supervisor Mode stack  */

    ldr r0, =_vector_table
    mcr p15, 0, r0, c12, c0, 0  /* set VBAR, Vector Base Address Register*/
    //mrc p15, 0, r0, c12, c0, 0  //read VBAR

    bl clean_bss

    bl system_init
    cpsie    i                      /* Unmask interrupts           */

    bl main

halt:
    b  halt


clean_bss:
    /* 清除 BSS 段 */
    ldr r1, =__bss_start
    ldr r2, =__bss_end
    mov r3, #0
clean:
    cmp r1, r2
    strlt r3, [r1]
    add r1, r1, #4
    blt clean

    mov pc, lr
```

**注意：执行 Reset_Handler 时，CPU 处于 IRQ 模式，用的是 IRQ 模式下的栈，需要先在 Reset_Handler 里设置好 IRQ 模式的栈，这样在中断模式里才可以使用栈，才能调用 C 函数。**

**注意：在 Reset_Handler 里调用“cpsie i”打开中断，这是把 CPSR 中的 I 位清零。**

**注意：在 Reset_Handler 里使用如下两条指令设置异常向量的基地址**

```
ldr r0, =_vector_table
mcr p15, 0, r0, c12, c0, 0  /* set VBAR, Vector Base Address Register*/
```

### 11.3.5 中断异常处理 C 函数部分

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/11_GPIO 中断/001_exception/gic.c”。

handle_irq_c 函数功能简述如下：

① 获取到 gic 的基地址；

② 读取 GICC_IAR 获得中断号；

③ 根据中断号调用对应中断号的 irq_handler 函数，该函数是用户通过 request_irq 注册的中断处理函数，④ 然后往 GICC_EOIR 写入中断号清除掉中断。

gic.代码如下：

```
void handle_irq_c(void)
{
    int nr;

    GIC_Type *gic = get_gic_base();
    /* The processor reads GICC_IAR to obtain the interrupt ID of the
     * signaled interrupt. This read acts as an acknowledge for the interrupt
     */
    nr = gic-> C_IAR;
    printf("irq %d is happened\r\n", nr);

    irq_table[nr].irq_handler(nr, irq_table[nr].param);

    /* write GICC_EOIR inform the CPU interface that it has completed
     * the processing of the specified interrupt
     */
    gic->C_EOIR = nr;
}
```

谁调用 reqeust_irq 设置了 irq_table[nr].irq_handler？请看下一节。

11.3.6 GPIO 中断初始化和安装中断处理程序

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/11_GPIO 中断/001_exception/main.c”。

先看看初始化函数 key_irq_init，功能如下(以 KEY1 为例)：

① 对于 KEY1，对应的引脚是 GPIO5_01，通过 EDGE_SEL 设置成双边沿触发；

② 设置 IMR 使能中断；

③ 为了防止误触发，先将 ISR 对应位写 1 清除掉中断；

④ 调用 request_irq 注册对应中断的中断处理函数，就是设置 irq_table 数组中某一项，设置函数指针：

对于 GPIO5_01，处理函数是 key_gpio5_handle_irq；对于 GPIO4_14，处理函数是 key_gpio4_handle_irq。初始化代码如下：

```
void key_irq_init(void)
{
    /* if set detects any edge on the corresponding input signal*/
    GPIO5->EDGE_SEL |= (1 << 1);
    /* if set 1, unmasked, Interrupt n is enabled */
    GPIO5->IMR |= (1 << 1);
    /* clear interrupt first to avoid unexpected event */
    GPIO5->ISR |= (1 << 1);

    GPIO4->EDGE_SEL |= (1 << 14);
    GPIO4->IMR |= (1 << 14);
    GPIO4->ISR |= (1 << 14);

    request_irq(GPIO5_Combined_0_15_IRQn, (irq_handler_t)key_gpio5_handle_irq,
NULL);
    request_irq(GPIO4_Combined_0_15_IRQn, (irq_handler_t)key_gpio4_handle_irq,
NULL);
}
```

还是以 KEY1 为例讲解处理函数 key_gpio5_handle_irq，它的功能如下：

① 读取 GPIO_DR 寄存器，根据 GPIO5_01 的状态打印信息、操作 LED

② 在 GPIO 模块内部清除中断

代码如下：

```
void key_gpio5_handle_irq(void)
{
    /* read GPIO5_DR to get GPIO5_IO01 status*/
    if((GPIO5->DR >> 1 ) & 0x1) {
        printf("key 1 is release\r\n");
        /* led off, set GPIO5_DR to configure GPIO5_IO03 output 1 */
        GPIO5->DR |= (1<<3); //led on
    } else {
        printf("key 1 is press\r\n");
        /* led on, set GPIO5_DR to configure GPIO5_IO03 output 0 */
        GPIO5->DR &= ~(1<<3); //led off
    }
    /* write 1 to clear GPIO5_IO03 interrput status*/
    GPIO5->ISR |= (1 << 1);
}
void key_gpio4_handle_irq(void)
{
    /* read GPIO4_DR to get GPIO4_IO014 status*/
    if((GPIO4->DR >> 14 ) & 0x1)
        printf("key 2 is release\r\n");
    else
        printf("key 2 is press\r\n");
    /* write 1 to clear GPIO4_IO014 interrput status*/
    GPIO4->ISR |= (1 << 14);
}
```

## 11.3.6 特定中断号的中断使能和禁止

设置好一切之后，就是使能中断了。对于 GIC，程序里使用 gic_enable_irq，它的功能为：

- 根据中断号找到对应的 GICD_ISENABLERn 寄存器；
- 往相应位中写入 1，即可使能中断。

要关闭中断时，操作是类似的，函数是 gic_disable_irq，通过往 GICD_ICENABLERn 对应的位写入 1 来禁止中断。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/11_GPIO 中断/001_exception/gic.c”。

```
void gic_enable_irq(IRQn_Type nr)
{
    GIC_Type *gic = get_gic_base();

    /* The GICD_ISENABLERs provide a Set-enable bit for each interrupt supported by
the GIC.
     * Writing 1 to a Set-enable bit enables forwarding of the corresponding
interrupt from the
     * Distributor to the CPU interfaces. Reading a bit identifies whether the
interrupt is enabled.
     */
    gic->D_ISENABLER[nr >> 5] = (uint32_t)(1UL << (nr & 0x1FUL));

}
```

```
void gic_disable_irq(IRQn_Type nr)
{
    GIC_Type *gic = get_gic_base();

    /* The GICD_ICENABLERs provide a Clear-enable bit for each interrupt supported
by the
     * GIC. Writing 1 to a Clear-enable bit disables forwarding of the
corresponding interrupt from
    * the Distributor to the CPU interfaces. Reading a bit identifies whether the
interrupt is enabled.
     */
    gic->D_ICENABLER[nr >> 5] = (uint32_t)(1UL << (nr & 0x1FUL));
}
```

### 11.3.7 修改 CPSR 使能中断

在 start.S 中，可以看到如下代码，它把 CP15 中 SCTRL 的值读出后，把 I bit 清零，再写入。这就是在 CPU 核中使能 IRQ 中断。代码如下：

```
Reset_Handler:
    /* Reset SCTRL Settings */
    mrc p15, 0, r0, c1, c0, 0  /* read SCTRL, Read CP15 System Control register */
    bic r0,  r0, #(0x1 << 13)  /* Clear V bit 13 to use normal exception vectors */
    bic r0,  r0, #(0x1 << 12)  /* Clear I bit 12 to disable I Cache */
    bic r0,  r0, #(0x1 <<  2)  /* Clear C bit  2 to disable D Cache */
    bic r0,  r0, #(0x1 << 2)   /* Clear A bit  1 to disable strict alignment*/
    bic r0,  r0, #(0x1 << 11)  /* Clear Z bit 11 to disable branch prediction */
    bic r0,  r0, #0x1              /* Clear M bit  0 to disable MMU */
    mcr p15, 0, r0, c1, c0, 0  /* write SCTRL, CP15 System Control register */
```

### 11.3.8 主函数调用

main 函数调用 system_init 进行系统初始化，system_init 做了这些事情：

① 调用 system_init_irq_table 初始化中断跳转表；
② 调用 key_irq_init 初始化按键中断：配置 GPIO、注册中断处理函数；
③ 调用 gic_init 初始化 GIC 控制器；
④ 最后通过 gic_enable_irq 使能中断。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/11_GPIO 中断/001_exception/main.c”。

```
void system_init()
{
    init_pins();
    led_gpio_init();
    led_ctl(0);//turn off led
    boot_clk_gate_init();
    boot_clk_init();
    uart1_init();
    puts("hello world\r\n");
    system_init_irq_table();
    key_irq_init();
    gic_init();
    gic_enable_irq(GPIO5_Combined_0_15_IRQn);
    gic_enable_irq(GPIO4_Combined_0_15_IRQn);
}
```

### 11.3.9 参考章节《4.3.4 编译程序》编译程序

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/11_GPIO 中断/001_exception”目录下。

### 11.3.10 参考章节《3.4 映像文件烧写、运行》烧写、运行程序

此时观察串口打印

按下 KEY1，绿灯点亮，松开，绿灯熄灭，同时串口会打印按下松开的信息。按下或者松开 KEY2，串口会打印出 KEY2 按下松开的信息。串口打印如下所示：

```
hello world
irq 106 is happened
key 1 is press
irq 106 is happened
key 1 is release
irq 104 is happened
key 2 is press
irq 104 is happened
key 2 is release
```

# 第12章 GTP 定时器和 EPIT 定时器编程

参考资料：网盘开发板配套资料“06_Datasheet（数据手册）/Core_board/CPU/IMX6ULLRM.pdf”：

- 《Chapter30 : General Purpose Timer (GPT)》。
- 《Chapter24 : Enhanced Periodic Interrupt Timer (EPIT)》。

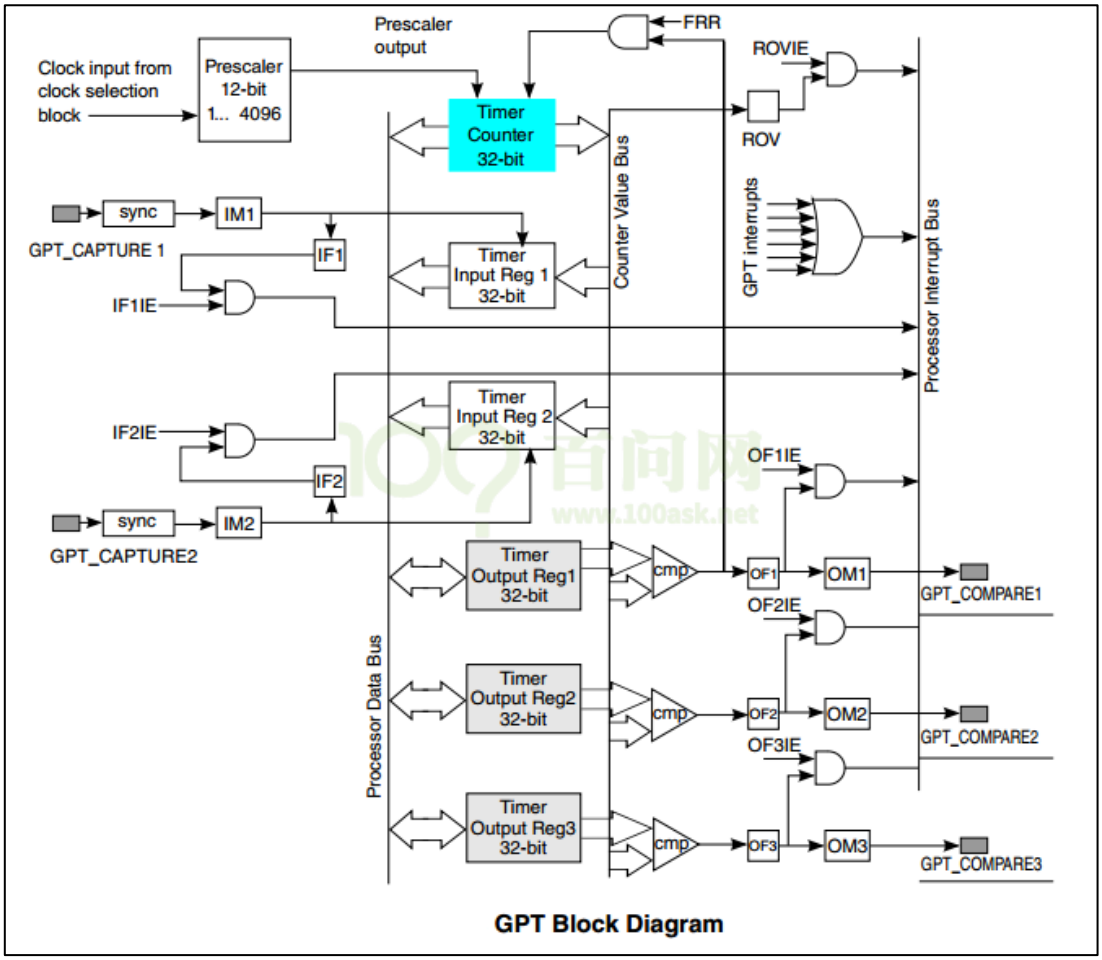
IMX6ULL 中有 2 种定时器：General Purpose Timer (GPT)——通用目的定时器，Enhanced Periodic Interrupt Timer (EPIT)——加强版周期性中断定时器。IMX6ULL 中 GPT、EPIT 定时器各有 2 个。本章将一一讲解。

## 12.1 GPT 定时器介绍

GPT 的英文是 General Purpose Timer，即通用目的定时器。IMX6ULL 有 2 个 GPT 定时器。

GPT 具有 32 位递增计数器。可以将外部引脚上的事件捕获到 GPT 寄存器中，这些输入事件可以设置为上升沿或下降沿触发，甚至双边沿触发。当定时器达到设定的值时，可以让定时器的输出引脚输出指定电平，或是产生中断。GPT 具有一个 12 位预分频器，它可以对多个时钟源的时钟进行分频。

GPT 框图如下：




GPT Block Diagram

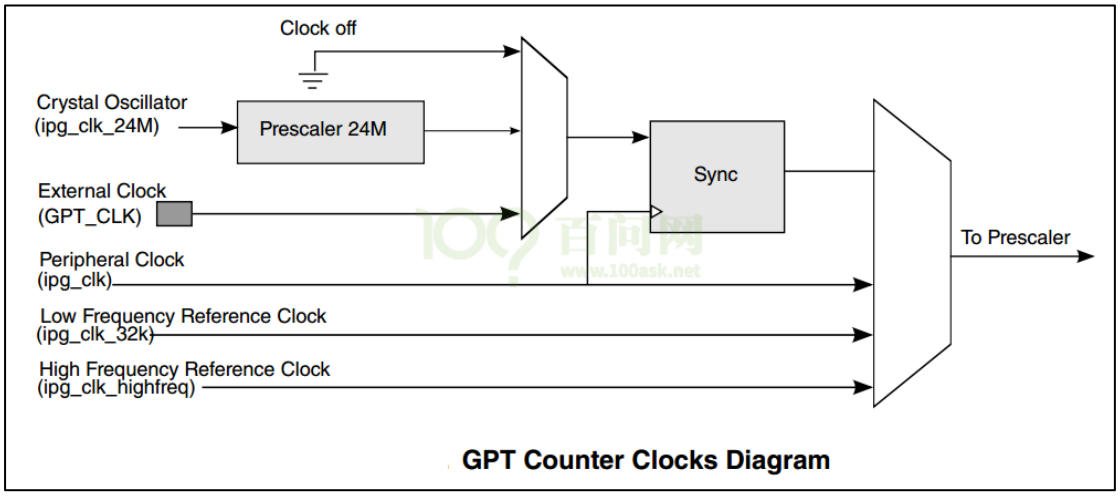
特性：

- 一个带有时钟源选择的 32 位递增计数器，时钟源包括外部时钟。
- 两个具有可编程触发沿的输入捕捉通道。
- 具有可编程输出模式的三个输出比较通道，还有一个“force compare”功能。
- 可以通过编程，让它在低功耗（low power）和调试（debug）模式下运行。
- 可以在捕获（capture），比较（compare）和翻转（rollover）事件时产生中断。
- 两种计数模式：重新启动（restart）或自由运行（free-run）模式。

### 12.1.1 时钟源选择

| Clock name | Clock Root | Description |
|---|---|---|
| ipg_clk | ipg_clk_root | Peripheral clock |
| ipg_clk_32k | ckil_sync_clk_root | Low-frequency reference clock (32 kHz) |
| ipg_clk_highfreq | perclk_clk_root | High-frequency reference clock |
| ipg_clk_s | ipg_clk_root | Peripheral access clock |

上表是 GPT 模块会使用到的时钟，ipg_clk 提供了 Peripheral 的时钟，ipg_clk_32k 提供低频参考时钟，ipg_clk_highfreq 提供了高频参考时钟，ipg_clk_s 提供了访问控制器寄存器所需的时钟。

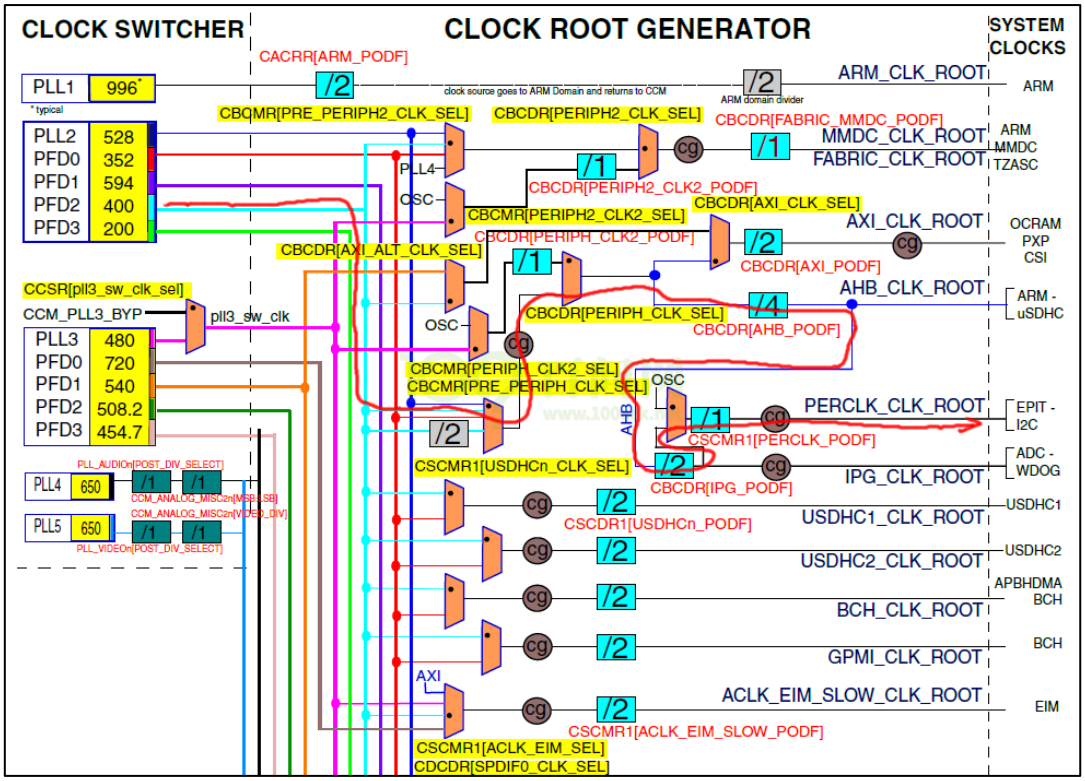


GPT Counter Clocks Diagram

从上图可以看出，可以从 4 个时钟源中选择输入到预分频器的时钟，分别为：
① 高频参考时钟（ipg_clk_highfreq），
② 低频参考时钟（ipg_clk_32k），
③ 外围时钟（ipg_clk），
④ 外部时钟（GPT_CLK）或者晶体振荡器时钟（ipg_clk_24M）
由于外部时钟（GPT_CLK）或者晶体振荡器时钟（ipg_clk_24M）只能选择一个。

本章暂时不关注 lower power mode 的内容，实验里将 ipg_clk 作为预分频器的时钟源。

ipg_clk 怎么设置呢？需要查看时钟树，下图中的红色箭头就是时钟路线：




根据 IMX6ULLRM.pdf 第 18 章的内容，可知 PPL2 又被称作 System PLL、SYS PLL。除 PLL2 的输出外，它还有 4 个分相器的输出：PFD0～3。

从上图可知，我们选用 SYS PLL PFD2 作为时钟源，经过 CBCDR[AHB_PODF](实验里设置为 2，对应分频 3)分频，得到 AHB_CLK_ROOT。

AHB_CLK_ROOT 再经过 CBCDR[IPG_PODF](实验里设置为 1，对应分频 2)分频，得到 IPG_CLK_ROOT。

PERCLK_CLK_ROOT 有两个来源：OSC(晶振)、IPG_CLK_ROOT。本实验中选择 IPG_CLK_ROOT，它经过 CSCMR1[PERCLK_PODF](实验里设置为 0，对应分频值 1)分频，就得到 PERCLK_CLK_ROOT。

本实验中 SYS PLL PFD2 是 396M，所以 PERCLK_CLK_ROOT = 396M / 3 / 2 = 66M。

## 12.1.2 时钟源选择的操作流程

GPT_CR 寄存器中的 CLKSRC 字段用于选择时钟源。要修改 CLKSRC 字段，需要先禁用 GPT(EN = 0)，这是理所当然的：GPT 正在运行着你不能去修改它的时钟源。

更改 GPT 时钟源时，应该按照以下顺序编写程序：

① 通过在 GPT_CR 寄存器中设置 EN = 0 来禁用 GPT；
② 禁用 GPT 中断寄存器（GPT_IR）；
③ 将输出模式配置为未连接/断开连接：往 GPT_CR 中的 OM1，OM2，OM3 写 0；
④ 禁用输入捕获模式：往 GPT_CR 的 IM1 和 IM2 中写入零；
⑤ 在 GPT_CR 寄存器中将时钟源 CLKSRC 更改为所需的值；
⑥ 将 GPT_CR 寄存器中的 SWR 位置 1；
⑦ 清除 GPT 状态寄存器（GPT_SR）（该寄存器是往相应位写 1 清 0）；

⑧ 在 GPT_CR 寄存器中设置 ENMOD = 1，以使 GPT 计数器为 0x00000000；
⑨ 在 GPT_CR 寄存器中启用 GPT（EN = 1）；
⑩ 启用 GPT 中断寄存器（GPT_IR）。

### 12.1.3 GPT 的计数模式

GPT 有两种计数模式：重新启动计数模式(Restart mode)、自由运行模式(free-run mode)。

① 重新启动计数模式(Restart mode)：

在重启模式下（可通过 GPT 控制寄存器 GPT_CR 选择），当计数器达到比较值时，计数器将复位并从 0x00000000 重新开始计数。

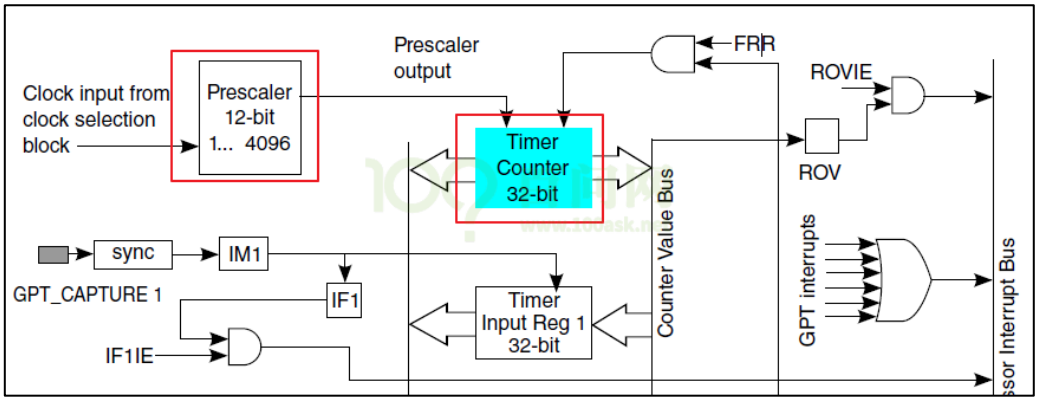
重新启动功能仅与比较通道 1 相关联。对通道 1 的比较寄存器的任何写操作都将复位 GPT 计数器。这样做是为了避免在进行计数时将比较值从较高的值更改为较低的值时可能丢失比较事件。对于其他两个比较通道，当发生比较事件时，计数器不会复位。

② 自由运行模式(free-run mode)：

在自由运行模式下，当所有三个通道发生比较事件时，计数器不会复位；而是，计数器继续计数直到 0xffffffff，然后翻转（变为 0x00000000）。

### 12.1.4 GPT 的操作

通用定时器（GPT）具有一个计数器（GPT_CNT），该计数器是 32 位递增计数器，在由软件启用该计数器后（EN = 1）开始计数，下文称之为主计数器。还有一个预分频器计数器，是用来给 GPT 提供分频时钟的。




① GPT 的禁止与使能

当 GPT(EN = 0)被禁止时，则主计数器和预分频器计数器将冻结其当前计数值。

当 EN 位置 1 且计数器再次使能时，ENMOD 位决定这两个计数器是否重置为 0：

•如果将 ENMOD 位置 1：则当启用 GPT（EN = 1）时，主计数器和预分频器计数器值将重置为 0。

•如果将 ENMOD 位设置为 0：则当启用 GPT（EN = 1）时，主计数器和预分频器计数器将从其冻结值继续计数。

② 低功耗模式(Low Power Down)

当 GPT 进入低功耗模式（STOP / WAIT）时，主计数器和预分频器计数器将冻结在其当前计数值。

当 GPT 退出低功耗模式时，无论 ENMOD 位值如何，主计数器和预分频器计

数器都将从其冻结值开始计数。

处理器可以随时读取 GPT_CNT，并且两个输入捕获通道都使用相同的计数器（GPT_CNT）。

③ 硬件复位将所有 GPT 寄存器复位为各自的复位值

除输出比较寄存器（OCR1，OCR2，OCR3）以外的所有寄存器的值均为 0x0，比较寄存器复位为 0xFFFF_FFFF。

④ ④ 软件复位（GPT_CR 控制寄存器中的 SWR 位）将复位所有寄存器位

EN，ENMOD，STOPEN，WAITEN 和 DBGEN，这些位的状态不受软件复位的影响。

**注意：禁用 GPT 时可以进行软件复位操作。**

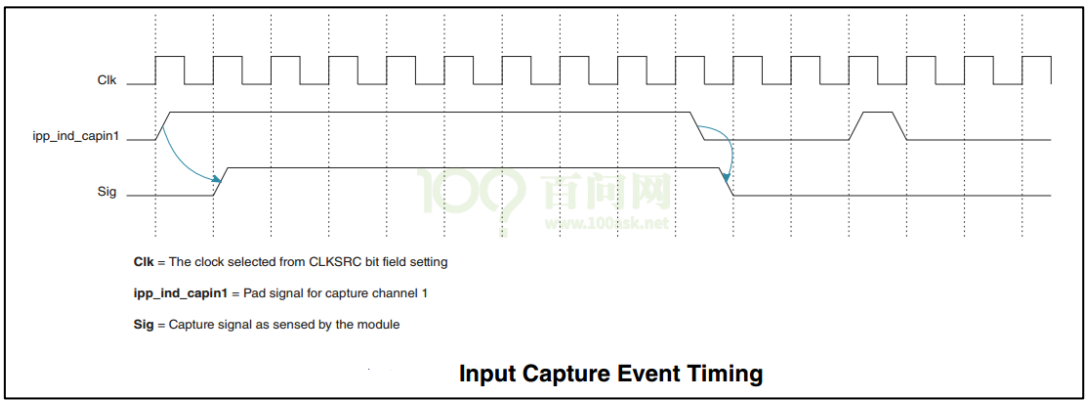
## 12.1.5 GPT 的输入捕获

GPT 有两个输入捕获通道，每个输入捕获通道都有：一个专用的捕获引脚，捕获寄存器和输入边沿检测/选择逻辑。每个输入捕获功能都有一个状态标记位，并且可以向处理器发出中断服务请求。

我们先设置好捕获引脚的边沿检测/选择逻辑，当该引脚上发生指定的边沿转换时，GPT_CNT 的内容被捕捉到相应的捕捉寄存器中，并设置适当的中断状态标志。如果该中断被使能了，它就可以产生中断。

有哪些边沿检测/选择逻辑？上升沿，下降沿，双边沿，或者禁用捕获。

捕获事件与计数器的时钟同步。假设当前正在记录一个事件，如果紧接着立刻发生另一个事件，那么第 2 个事件可能会丢失。当前事件正在记录时，至少一个时钟周期之后发生的事件，才能保证不丢失。可以随时读取输入捕获寄存器，而不会影响它们的值。具体时序图如下所示：




## 12.1.6 GPT 的输出比较

GPT 中有三个输出比较通道，它们都使用同一个计数器（GPT_CNT），输入捕获通道也是使用这个计数器。

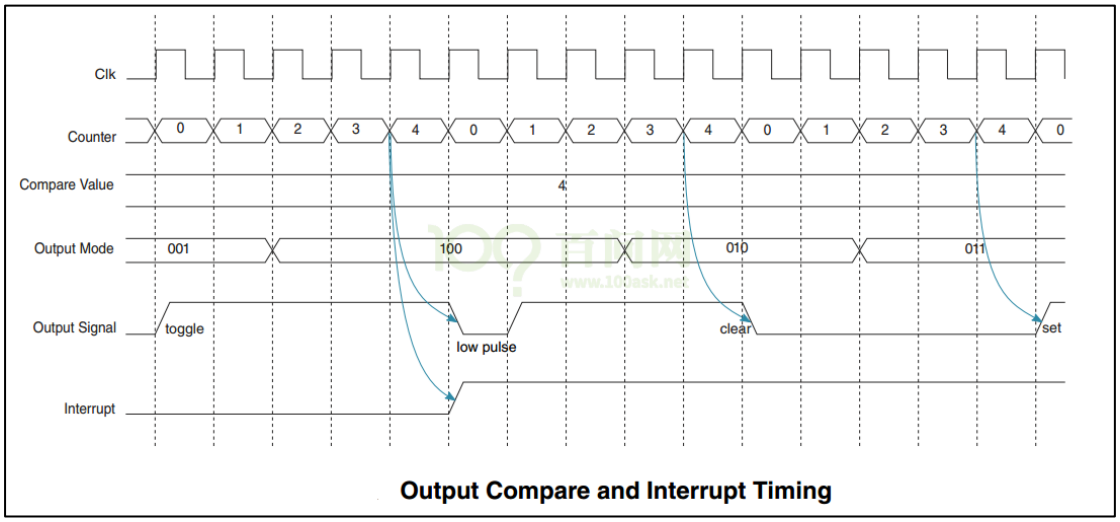
软件上先设置输出比较寄存器，当GPT_CNT 的值与输出比较寄存器匹配时，GPT 就会设置“输出比较状态标志”(output compare status flag)为 1，并产生中断。

并且，根据设置的不同，输出比较定时器对应的引脚将被置位（set）、清除（clear）、翻转（toggle）或者不受影响，或输出一个低脉冲(脉冲持续时间为定时器的时钟源的周期)。

还有一个“强制比较（forced-compare）”功能，一旦设置，就会马上产生比较事件；不管当前计数器值是否等于比较值。强制比较的产生的事件，跟正常

的输出事件相同，只是它不会设置状态标记位并且不会产生中断。一旦设置 force-compare 位，该事件会即刻产生，这个位是自动清除的，读的话一直零。

下图是输出比较时的时序图：




Output Compare and Interrupt Timing

## 12.1.7 GPT 的中断

GPT 可以产生 6 种不同的中断：

① 翻转中断(Rollover Interrupt)

当 GPT 计数器达到 0xffffffff，然后重新设置为 0x00000000 并继续计数时，将产生翻转中断。

翻转中断通过 GPT_IR 寄存器中的 ROVIE 位来使能。相关的状态位是 GPT_SR 寄存器中的 ROV 位。

② 输入捕获中断 1、2

捕获事件发生后，相应的输入捕获通道会产生一个中断。

“捕获事件中断”通过 IF2IE 和 IF1IE 位（在 GPT_IR 寄存器中）使能；相应的状态位是 IF2 和 IF1（在 GPT_SR 寄存器中）。

发生捕获事件时会产生中断，但是该中断无论是否被处理，都不会影响下次捕获事件的发生。当再次发生捕获事件时，无论之前的中断是否被处理了，捕获寄存器都会被再次更新。

③ 输出比较中断 1、2、3

当比较事件发生后，相应的输出比较通道会产生一个中断。

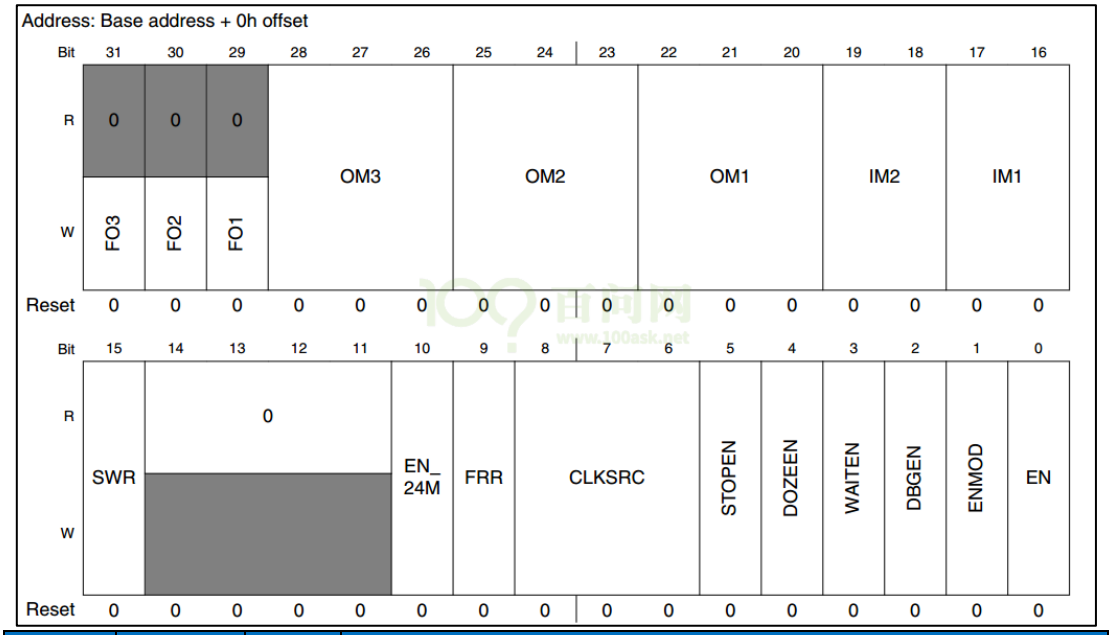
“比较事件中断”由 OF3IE，OF2IE 和 OF1IE 位（在 GPT_IR 寄存器中）使能；相应的状态位是 OF3，OF2 和 OF1（在 GPT_SR 寄存器中）。

“强制比较（Force compare）”不会产生中断。

# 12.2 GPT 寄存器介绍

## 12.2.1 GPT Control Register (GPTx_CR)

GPT 控制寄存器，用来配置 GPT。

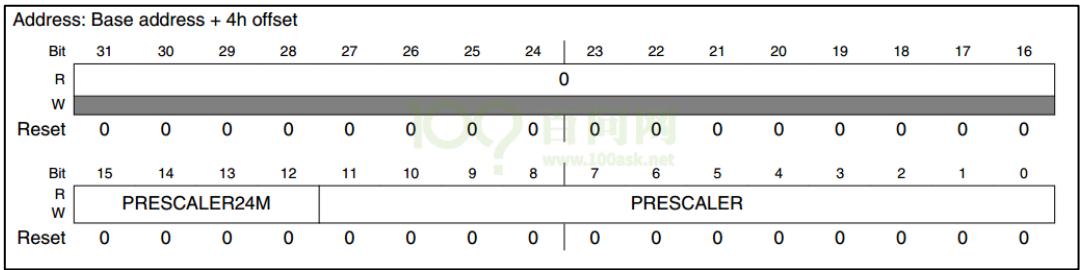



| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:29] | FO3,2,1 | R/W | 对应 3 个输出通道，用于设置强制比较（Force compare），<br>写 0：没影响<br>写 1：强制比较，会导致相应的输出引脚状态发生变化，但是 OFn 不受影响 |
| [28:26] | OM3 | R/W | 用来设置输出通道 3 的工作模式，<br>000：输出引脚跟 GPT 断开，即输出引脚不受影响；<br>001：输出引脚翻转；<br>010：输出引脚清 0；<br>011：输出引脚置位；<br>1xx：输出引脚产生一个低脉冲 |
| [25:23] | OM2 | R/W | 用来设置输出通道 2 的工作模式，跟 OM3 类似 |
| [22:20] | OM1 | R/W | 用来设置输出通道 1 的工作模式，跟 OM3 类似 |
| [19:18] | IM2 | R/W | 用来设置输入通道 2 的工作模式，<br>00：捕获功能关闭；<br>01：捕获上升沿；<br>10：捕获下降沿；<br>11：同时捕获上升和下降沿 |
| [15] | SWR | R/W | 软件复位，会自动清零，<br>a. GPT 在复位状态时，该位自动置 1<br>b. 复位结束时，该位自动清 0<br>c. 设置该位为 1 时，会把所有寄存器设置为它们的默认值，GTPx_CR 中这些位不受影响：EN、ENMOD、STOPEN、WAITEN、DBGEN |
| [10] | EN_24M | R/W | 是否使用 24M 晶振作为 GPT 时钟，<br>0：不使用<br>1：使用<br>硬件复位时，该位被设置为 0；软件复位不影响该位 |

| | | | |
|---|---|---|---|
| [9] | FRR | R/W | 用来选择“Restart”模式或“Free-Run”模式，<br>0：Restart 模式(比较事件发生后，计数值清 0，重新计数)<br>1：Free-Run 模式(比较事件不影响计数值，计数值到达 0xFFFFFFFF 才清 0) |
| [8:6] | CLKSRC | R/W | 时钟源选择，<br>000：时钟源断开；<br>001：Peripheral Clock (ipg_clk)；<br>010：High Frequency Reference Clock (ipg_clk_highfreq)；<br>011：External Clock；<br>100：Low Frequency Reference Clock (ipg_clk_32k)；<br>101：Crystal oscillator as Reference Clock (ipg_clk_24M) |
| [5] | STOPEN | R/W | stop mode 时 GPT 是否使能，<br>0：在 stop mode 下，GPT 禁止<br>1：在 stop mode 下，GPT 仍然使能 |
| [4] | DOZEEN | R/W | Doze mode 时 GPT 是否使能，<br>0：在 doze mode 下，GPT 禁止<br>1：在 doze mode 下，GPT 仍然使能 |
| [3] | WAITEN | R/W | Wait mode 时 GPT 是否使能，<br>0：在 wait mode 下，GPT 禁止<br>1：在 wait mode 下，GPT 仍然使能 |
| [2] | DBGEN | R/W | Debug mode 时 GPT 是否使能，<br>0：在 debug mode 下，GPT 禁止<br>1：在 debug mode 下，GPT 仍然使能 |
| [1] | ENMOD | R/W | 当 EPIT 重新使能后，主计数器和预分频器计数器从什么值开始计数，<br>0：从上次关闭时的计数值继续计数；<br>1：主计数器、预分频计数器都从 0 开始计数 |
| [0] | EN | R/W | GPT 使能位，<br>0：GPT 禁止；<br>1：GPT 使能 |

## 12.2.2 GPT Prescaler Register (GPTx_PR)

GPT 预分频寄存器，用来决定 GPT 时钟的分频系数。

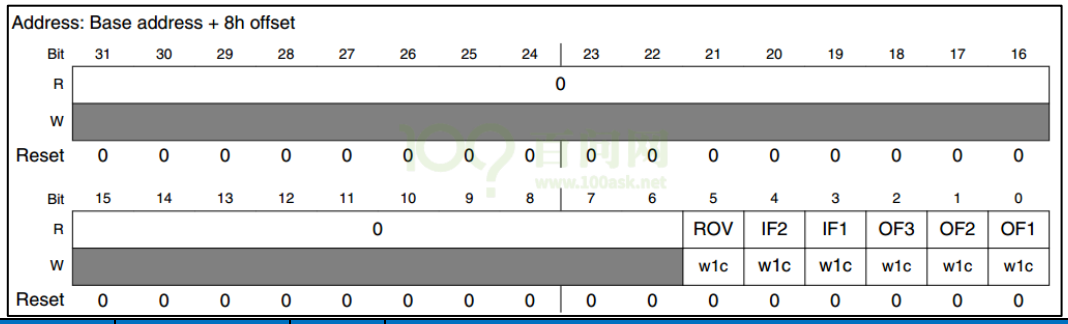



| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [15:12] | PRESCALER24M | R/W | 时钟源在 24M 晶振时，GPT 时钟的分频系数 |

| | | | |
|---|---|---|---|
| [11:0] | PRESCALER | R/W | 选择其它时钟源时，GPT 时钟的分频系数 |

### 12.2.3 GPT Status Register (GPTx_SR)

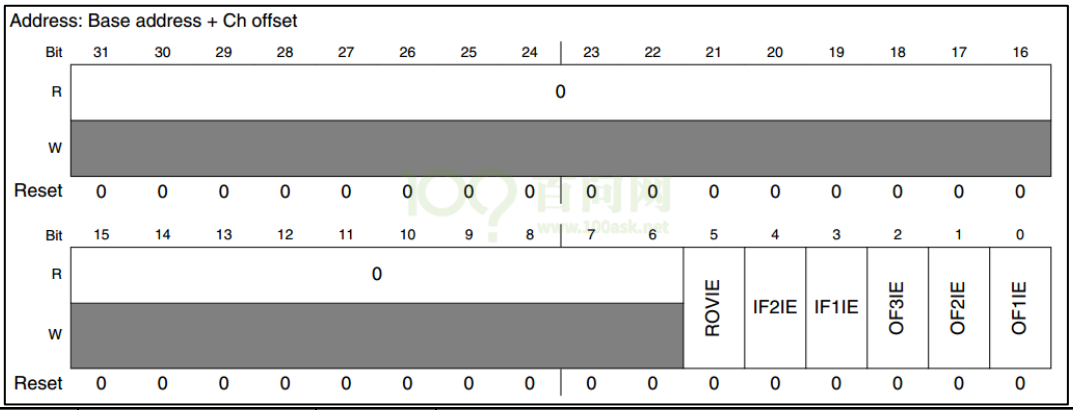
GPT 状态寄存器，用来显示计数值是否翻转、输入事件、输出事件是否发生了。




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [5] | ROV | R/W | 计数器的值是否到达了 0xFFFFFFFF(Rollover)，<br>0：Rollover 未发生；<br>1：Rollover 已发生 |
| [4:3] | IF2、IF1 | R/W | 输入捕获通道 2、1 的事件是否已经发生，<br>0：未发生；<br>1：已发生 |
| [2:0] | OF3、OF2、OF1 | R/W | 输出比较通道 3、2、1 的事件是否已经发生，<br>0：未发生；<br>1：已发生 |

### 12.2.4 GPT Interrupt Register (GPTx_IR)

GPT 中断寄存器，用来设置翻转，输入和输出通道事件的中断使能位，与状态寄存器的位对应。

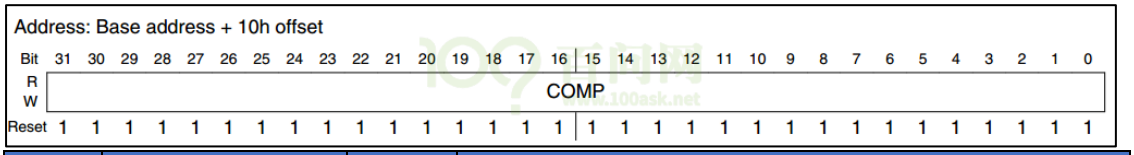



| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [5] | ROVIE | R/W | Rollover 中断使能位(计数值达到 0xFFFFFFFF)，<br>0：Rollover 中断禁止；<br>1：Rollover 中断使能 |
| [4:3] | IF2IE、IF1IE | R/W | 输入捕获通道 2、1 的中断使能位，<br>0：中断禁止；<br>1：中断使能 |

| | | | |
|---|---|---|---|
| [2:0] | OF3IE、OF2IE、OF1IE | R/W | 输出比较通道 3、2、1 的中断使能位，<br>0：中断禁止；<br>1：中断使能 |

### 12.2.5 GPT Output Compare Register 1~3 (GPTx_OCR1~3)

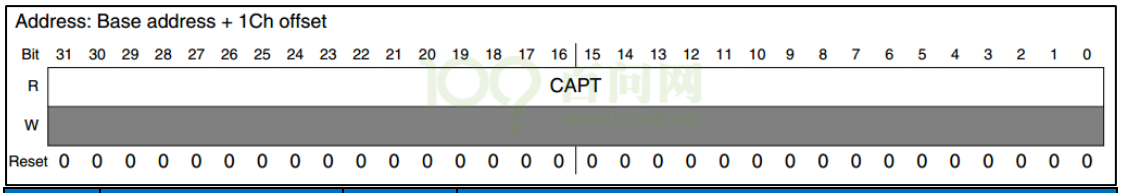
GPT 输出比较寄存器，有 GPTx_OCR1-3 共 3 个输出比较寄存器。
当计数器达到输出比较寄存器的值时，将在相应通道上产生事件。
当 GPT 使用 Restart mode 时，写入比较寄存器的同时会复位 GPT 计数器。
写寄存器的值在一个时钟周期后生效，读寄存器的值会立即返回。




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:0] | COMP | R/W | 比较值 |

### 12.2.6 GPT Input Capture Register 1~2 (GPTx_ICR1~2)

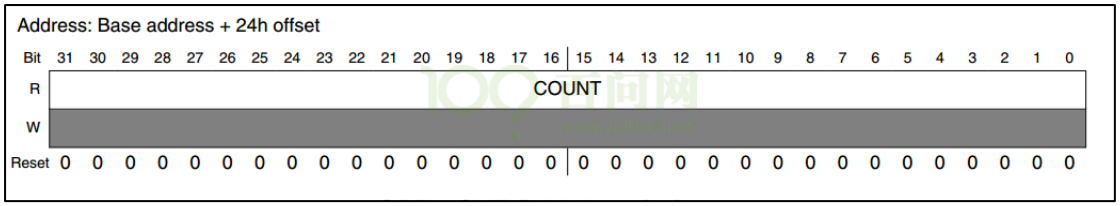
GPT 输入捕获寄存器，有 GPTx_ICR1-2 共两个输入捕获寄存器。它们是只读寄存器，用于保存相应输入捕获通道中，发生捕获事件发生时计数器的值。




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:0] | CAPT | R/W | 发生捕获事件时，GPT 计数器的值 |

### 12.2.7 GPT Counter Register (GPTx_CNT)

GPT 计数器寄存器，主计数值，它是只读寄存器；读取 GPT 计数器的值，不影响计数过程。




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:0] | COUNT | R/W | GPT 计数器的值 |

## 12.3 GPT 查询方式延时代码详解与测试

### 12.3.1 代码分析

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/001_timer_gpt_poll/gpt.c”。

首先看看 gpt_poll_init 函数，它用于初始化 GPT 定时器：

① 首先对 GPT 进行软件复位，
② 设置 gpt 为 restart 模式，

③ 时钟源选择 Peripheral Clock (ipg_clk)为 66M，预分频值设置为 0（即预分频值为 1）。

gpt_poll_init 函数如下：

```
void gpt_poll_init(GPT_Type *base)
{
    /* bit15 SWR, Software reset*/
    base->CR |= (1 << 15);
    /* Wait reset finished. */
    while((base->CR >> 15) & 0x1) {
    }

    /*
     *bit10: Enable 24 MHz clock input from crystal
     *bit9: 0 restart mode, 1 free-run mode:set 0
     *bit8-6: Clock Source select :001 Peripheral Clock (ipg_clk)
     *bit5: GPT Stop Mode enable
     *bit3: GPT Wait Mode enable.
     *bit1: GPT Enable Mode
     */
    base->CR = (1 << 6) | (1 << 5) | (1 << 3) | (1 << 1);

    /*
     *bit15-bit12:PRESCALER24M
     *bit11-0:PRESCALER
     */
    base->PR = 0;
}
```

再看看 gpt_poll_restart 函数，用它来设置输出比较寄存器(想延时多少 us？)，并启动 GPT 定时器：

① 根据传输入的延时时间值，设置输出比较寄存器；

② 为了防止状态寄存器已经设置，先清除一下状态寄存器对应的位；

③ 再使能中断寄存器对应的位，

④ 最后使能 GPT 计数器；

⑤ 等待 compare flag 被 GPT 置位，置位后表示延时时间已到；

⑥ 关闭 GPT 计数器，禁止中断寄存器对应的位，清除掉状态位。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/001_timer_gpt_poll/gpt.c”。

```
void gpt_poll_restart(GPT_Type *base, enum gpt_comp_channel chan, unsigned int us)
{
    base->OCR[chan] =  USEC_TO_COUNT(us);
    /* write 1 to clear int status to avoid unexpected compare event*/
    base->SR |= (1 << chan);
    /* enable interrupt*/
    base->IR |= (1 << chan);
    /* gpt enable*/
    base->CR |= (1 << 0);
    /*wait for compare flag set*/
    while(!((base->SR >> chan) & 0x1))
        ;
    /* gpt disable*/
    base->CR &= ~(1 << 0);
    /* disable interrupt*/
    base->IR &= ~(1 << chan);
```

```
    /* write 1 to clear int status*/
    base->SR |= (1 << chan);
}
```

在主函数中，使用 gpt_poll_init 初始化 GPT1，不断调用 gpt_poll_restart(GPT1, OUT_COMP1, 1000000);延时 1s，并且依次点亮和关闭绿灯。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/001_timer_gpt_poll/main.c”。

```
gpt_poll_init(GPT1);
    while(1) {
        gpt_poll_restart(GPT1, OUT_COMP1, 1000000);
        GPIO5->DR &= ~(1<<3);//led on
        printf("led is on\r\n");
        gpt_poll_restart(GPT1, OUT_COMP1, 1000000);
        GPIO5->DR |= (1<<3);//led off
        printf("led is off\r\n");
    }
```

### 12.3.2 参考章节《4.3.4 编译程序》编译程序

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/001_timer_gpt_poll/”目录下。

### 12.3.3 参考章节《3.4 映像文件烧写、运行》烧写、运行程序

此时观察串口信息

绿灯不断闪烁，每隔 1s 翻转状态。串口输出如下：


## 12.4 GPT 中断方式延时代码详解与测试

### 12.4.1 GPT1 中断号的确定

查询数据手册 chapter 3 的 Table3-1 中断号表，gpt1 如下所示：

| | | | |
|---|---|---|---|
| 55 | gpt1 | - | OR of GPT1 Rollover interrupt line, Input Capture 1 & 2 lines, Output Compare 1,2 &3 Interrupt lines |

GIC 中断号需要再这个序号基础上加上 32，所以 gpt1 的 gic 中断号为 55+32=87。

### 12.4.2 代码分析

- **通过 gpt_init 函数初始化 gpt。**

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/002_timer_gpt_int”目录下。

先看看 gpt_init 函数，它的功能如下：

① 先对 GPT 进行软件复位；

② 然后设置 gpt 为 restart 模式；

③ 时钟源选择 Peripheral Clock (ipg_clk)为 66M，预分频值设置为 0（即预分频值为 1）；

④ 根据延时值设置设置输出比较寄存器。

```
gpt.c：
/* assume use ipc clk which is 66MHz, 1us against to 66 count */
#define USEC_TO_COUNT(us) (us * 66 - 1)

void gpt_init(GPT_Type *base, enum gpt_comp_channel chan, int us)
{
    /* bit15 SWR, Software reset*/
    base->CR |= (1 << 15);
    /* Wait reset finished. */
    while((base->CR >> 15) & 0x1) {
    }

    /*
     *bit10: Enable 24 MHz clock input from crystal
     *bit9: 0 restart mode, 1 free-run mode:set 0
     *bit8-6: Clock Source select :001 Peripheral Clock (ipg_clk)
     *bit5: GPT Stop Mode enable
     *bit3: GPT Wait Mode enable.
     *bit1: GPT Enable Mode
     */
    base->CR = (1 << 6) | (1 << 5) | (1 << 3) | (1 << 1);

    /*
     *bit15-bit12:PRESCALER24M
     *bit11-0:PRESCALER
     */
    base->PR = 0;

    /* GPTx_OCR1  bit31-0: Compare Value
     * When the counter value equals the COMP bit field value,
     * a compare event is generated on Output Compare Channel 1.
     */
    base->OCR[chan] = USEC_TO_COUNT(us);
}
```

- **gpt 中断使能函数**

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/002_timer_gpt_int/gpt.c”。

```
void gpt_enable_interrupt(GPT_Type *base, enum gpt_interrupt_bit bit, int on)
{
    if (on)
        base->IR |= (1 << bit);
    else
        base->IR &= ~(1 << bit);
}
```

- **启动 GPT 函数**

设置控制寄存器的运行位，bit[0]。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/002_timer_gpt_int/gpt.c”。

```
void gpt_run(GPT_Type *base, int on)
{
    /* bit0: GPT Enable */
    if (on)
        base->CR |= (1 << 0);
    else
        base->CR &= ~(1 << 0);
}
```

- 中断处理函数

中断处理函数里首先清除 GPT1 状态寄存器对应的位，然后每发生一次中断翻转绿灯的状态。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/002_timer_gpt_int/main.c”。

```
void GPT1_COMP1_handle_irq(void)
{
    static int on = 1;

    printf("GPT1 comp0 interrupt happened\r\n");
    /*
     * bit0: OF1 Output Compare 1 Flag
     * write 1 clear it */
    GPT1->SR |= 1;

    /* read GPIO5_DR to get GPIO5_IO01 status*/
    if(on) {
        /* led off, set GPIO5_DR to configure GPIO5_IO03 output 1 */
        GPIO5->DR |= (1<<3); //led off
        on = 0;
    } else {
        /* led on, set GPIO5_DR to configure GPIO5_IO03 output 0 */
        GPIO5->DR &= ~(1<<3); //led on
        on = 1;
    }
}
```

- 主函数的设置

主函数做了这些事情：

① 首先调用 gpt_init 初始化 GPT1 计数器：延时时间设置为 1s；

② 然后调用 request_irq 注册中断处理函数；

③ 接下来使能中断：先在 GIC 中使能 GPT1 中断，再在 GPT1 中使能输出通道 1 的中断；

④ 最后启动 GPT1 计数器。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/002_timer_gpt_int/main.c”。

```
gpt_init(GPT1, OUT_COMP1, 1000000);// set 1s
request_irq(GPT1_IRQn, (irq_handler_t)GPT1_COMP1_handle_irq, NULL);
gic_enable_irq(GPT1_IRQn);
gpt_enable_interrupt(GPT1, IR_OF1IE, 1);
gpt_run(GPT1, 1);
```

### 12.4.3 参考章节《4.3.4 编译程序》编译程序

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/002_timer_gpt_int”目录下。

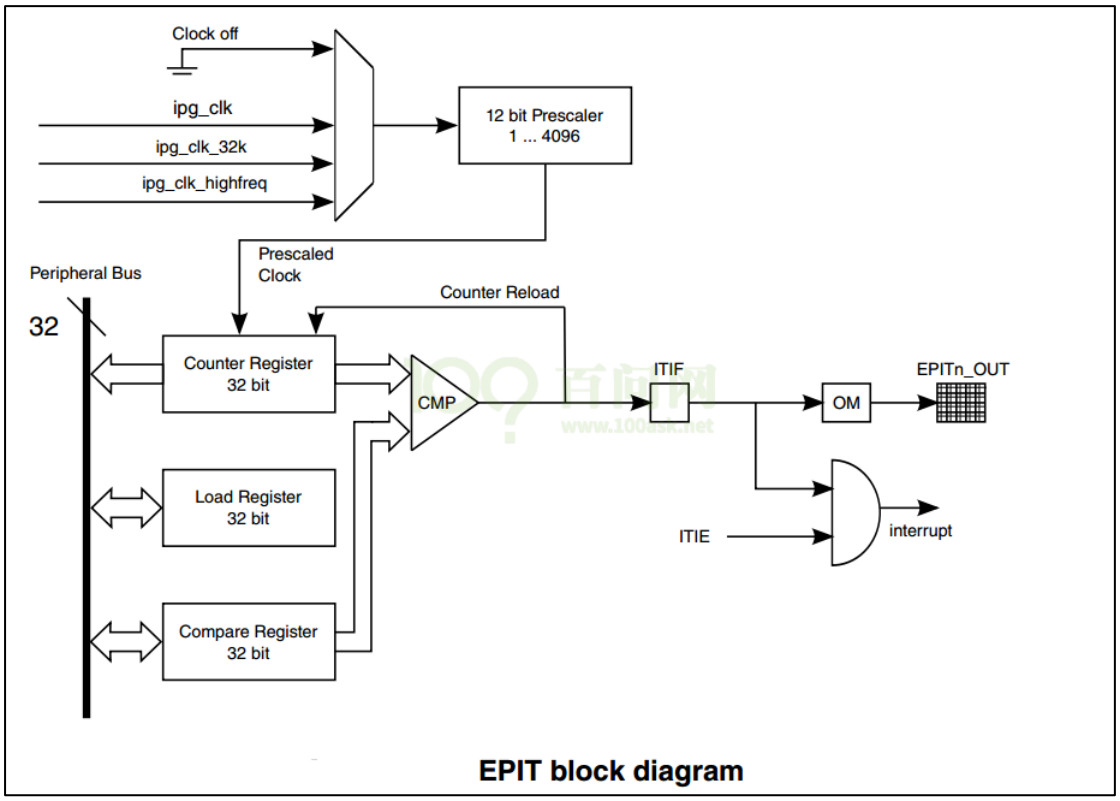
### 12.4.4 参考章节《3.4 映像文件烧写、运行》烧写、运行程序

此时观察串口及开发板 led 灯现象
绿灯不断闪烁，每隔 1s 翻转状态
串口输出：

```
hello world
irq 87 is happened
GPT1 comp0 interrupt happened
irq 87 is happened
GPT1 comp0 interrupt happened
irq 87 is happened
GPT1 comp0 interrupt happened
irq 87 is happened
GPT1 comp0 interrupt happened
irq 87 is happened
GPT1 comp0 interrupt happened
```

## 12.5 EPIT 定时器介绍

EPIT 是一个 32 位的计时器，能够在处理器很少干预的情况下以固定的时间间隔提供精确的中断。软件使能后，EPIT 就开始计数。IMX6ULL 有 2 个 EPIT 定时器。框图如下：



### 12.5.1 EPIT 特性

EPIT 具有以下主要特性：

- •可选择时钟源的 32 位递减计数器
- •12 位预分频器，用于对输入时钟进行分频
- •可即时编程的计数器值
- •可以设置在低功耗和调试模式下，计数器仍然运行
- •计数器达到比较值时产生中断

时钟源的选择与 GPT 类似，后续实验时钟源也选择 ipg_clk 作为时钟源，ipg_clk 设置的频率为 66M。

### 12.5.2 操作模式

EPIT 可以设置为 set-and-forget 或 free-running 模式，设置 EPIT_CR[RLD]选择所需的模式。

① set-and-forget 模式

要选择这种操作模式，将控制寄存器（EPIT_CR）中的 RLD 位置 1。

在这种模式下，计数器从加载寄存器（EPIT_LR）获取初始值，你不能直接写入初始值。每当计数器达到零时，EPIT_LR 中的值就会加载到计数器中，计数器再次将此值减到零。

要想立刻设置 EPIT 的计数值，不必等到它减小到 0 后再加载 EPIT_LR。可以设置 EPIT 计数器覆盖使能位（EPIT_CR [IOVW]），并将所需的初始化值写入 EPIT_LR。

② free-runnning 模式

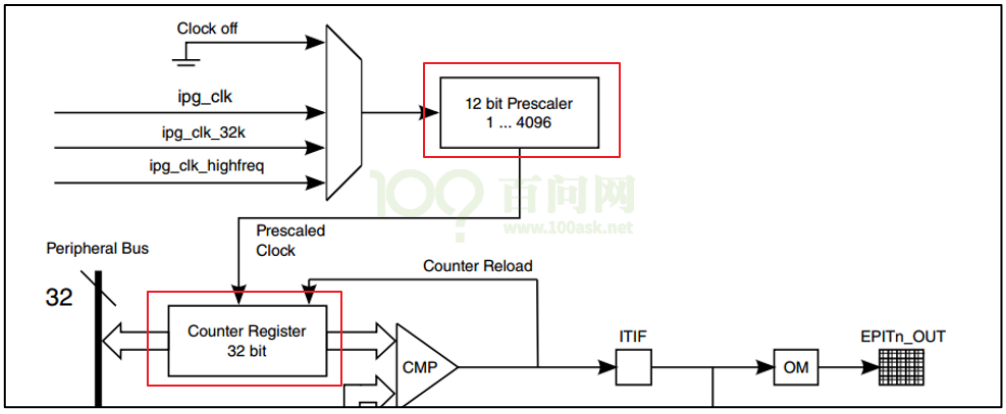
选择此操作模式的话，需要清除 RLD 位。

在这种模式下，计数器递减到 0000 0000h 时，会翻转到 FFFF FFFFh。不会从加载寄存器重新加载新值。翻转后，计数器继续递减计数。

要直接初始化计数器，需要设置 EPIT 计数器覆盖使能位（EPIT_CR[IOVW]），并将所需的初始化值写入 EPIT_LR。

### 12.5.3 操作过程

EPIT 具有一个 32 位递减计数器，在由软件启用该计数器后（EN = 1）开始计数，下文称之为主计数器。还有一个预分频器计数器，是用来给 EPIT 提供分频时钟的。如下图：



主计数器的起始值从 EPITx_LR 寄存器中加载，程序可以随时把新值写入该寄存器。比较寄存器中的值用来确定中断发生的时间。

当禁用 EPIT（EN = 0）时，主计数器和预分频器计数器会将其计数冻结为当前计数值。当重新启用 EPIT（EN = 1）时，ENMOD 位（可读可写位）决定这 2 个计数器的值：

① 如果设置了 ENMOD：则主计数器从加载寄存器加载值（如果 RLD = 1）或者 FFFF FFFFh（如果 RLD = 0）；预分频计数器复位（000h）。

② 如果清除了 ENMOD：则主计数器和预分频器计数器均从其冻结值重新开始计数。

如果将 EPIT 设置为在低功耗模式（STOP / WAIT）下被禁用，则当 EPIT 进入低功耗模式时，主计数器和预分频器计数器都冻结在其当前计数值。当 EPIT 退出低功耗模式时，无论 ENMOD 位如何，主计数器和预分频器计数器都将从其冻结值开始计数。

硬件复位会将所有 EPIT 寄存器复位为各自的复位值。

软件复位的效果跟硬件复位相同，但是无法影响控制寄存器中的这些位：EN、ENMOD、STOPEN 和 WAITEN 位，这些位的状态不受软件复位的影响。即使禁用了 EPIT，也可以进行软件复位操作。

### 12.5.4 比较事件

EPIT 启动后，EPIT_CNR 中的值递减，当它等于 EPIT_CMPR 的值时，就产生比较事件：设置比较状态标志，产生中断(如果控制寄存器中的 OCIEN 位是 1 的话)。

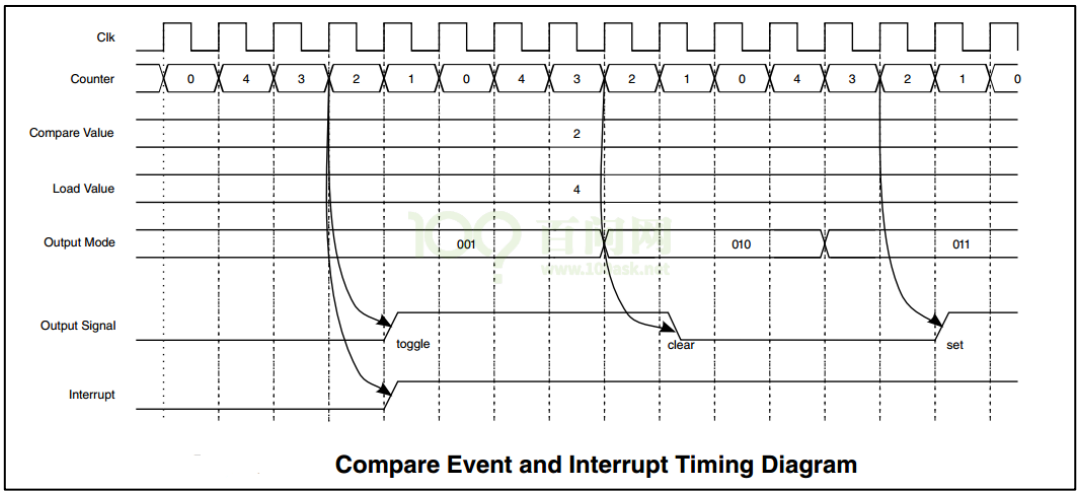
可以设置控制寄存器中输出模式（OM）位，当发生比较事件时，对应的输出引脚如何动作：置位（set）、清除（clear）、翻转（toggle）或者不受影响。先讲个概念：翻转(rollover)，就是当计数器值达到 0x0000_0000 时，它需要加载新值，这就叫 rollover。

如果需要在翻转时产生中断，则比较寄存器的取值要小心设置：

① 在 set-and-forget 模式下，应该取 EPITx_LR 的值；

② 在 free-running 模式下，应该取 0xFFFF_FFFF。

下图显示了比较事件和中断的时序：




Compare Event and Interrupt Timing Diagram

计数器值覆盖：可以在任何时候将 EPIT 计数器值设置为所需要的值，操作方法为：

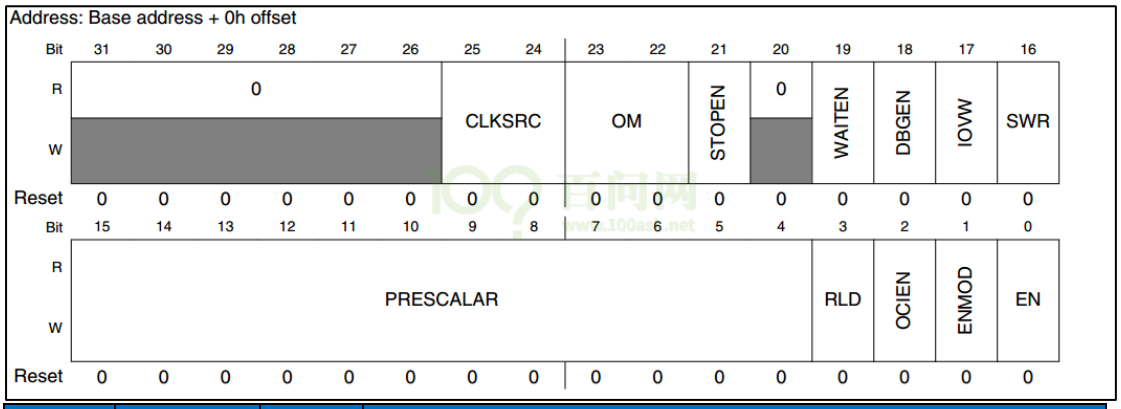
① 设置控制寄存器中的 IOVW 位，即 EPIT_CR[IOVW]；

② 然后将值写入到加载寄存器，即 EPIT_LR；
如果 EPIT 正在运行，则计数器将以新值继续计数。

## 12.6 EPIT 寄存器介绍

### 12.6.1 Control register (EPITx_CR)

EPIT 控制寄存器，用来配置 EPIT。

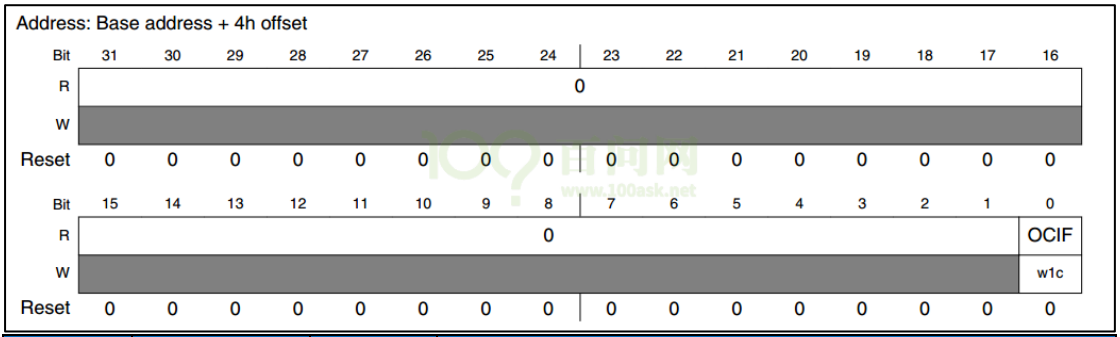



| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [25:24] | CLKSRC | R/W | 时钟源选择，<br>00：时钟源断开；<br>01：Peripheral clk，即 ipg_clk；<br>10：Hign-frequency，即 ipg_clk_highfreq；<br>11：low-frequency，即 ipg_clk_32k |
| [23:22] | OM | R/W | 用来设置输出通道模式，<br>00：输出引脚跟 EPIT 断开，即输出引脚不受影响；<br>01：输出引脚翻转；<br>10：输出引脚清 0；<br>11：输出引脚置位 |
| [21] | STOPEN | R/W | stop mode 时 EPIT 是否使能，<br>0：在 stop mode 下，EPIT 禁止<br>1：在 stop mode 下，EPIT 仍然使能 |
| [19] | WAITEN | R/W | Wait mode 时 EPIT 是否使能，<br>0：在 wait mode 下，EPIT 禁止<br>1：在 wait mode 下，EPIT 仍然使能 |
| [18] | DBGEN | R/W | Debug mode 时 EPIT 是否使能，<br>0：在 debug mode 下，EPIT 禁止<br>1：在 debug mode 下，EPIT 仍然使能 |
| [17] | IOVW | | EPIT 计数器覆盖使能位，<br>0：写入 EPIT_LR 的值不会覆盖 EPIT 计数器；<br>1：写入 EPIT_LR 的值会马上覆盖 EPIT 计数器的值 |
| [16] | SWR | R/W | 软件复位，这位会自动清零，<br>a. EPIT 在复位状态时，该位自动置 1<br>b. 复位结束时，该位自动清 0<br>c. 设置该位为 1 时，会把所有寄存器设置为它们的默认值，EPITx_CR 中这些位不受影响：EN、ENMOD、STOPEN、WAITEN、DBGEN |

| | | | |
|---|---|---|---|
| [15:4] | PRESCALER | R/W | EPIT 时钟的分频系数，<br>0x000：除以 1；<br>0x001：除以 2；<br>……<br>0xFFF：除以 4096 |
| [3] | RLD | R/W | 计数器模式(计数器加载模式)，<br>0：free-running mode，计数器到达 0 时，变为 0xFFFFFFFF；<br>1：set-and-forget mode，计数器到达 0 时，加载 EPITx_LR 的值 |
| [2] | OCIEN | R/W | 输出比较中断使能，<br>0：比较事件发生时，中断禁止；<br>1：比较事件发生时，中断使能 |
| [1] | ENMOD | R/W | 当 EPIT 重新使能后，主计数值从什么值开始计数：<br>0：从上次关闭时的计数值继续计数；<br>1：如果 RLD 为 1，从加载计数器开始计数；<br>如果 RLD 为 0，从 0xFFFF_FFFF 开始计数；<br>EPIT 重新使能时，预分频计数器总是从 0 开始计数。 |
| [0] | EN | R/W | EPIT 使能位，<br>0：EPIT 禁止；<br>1：EPIT 使能 |

## 12.6.2 Status register (EPITx_SR)

EPIT 状态寄存器，只有一个状态位，用来表示输出比较事件是否发生，写 1 清除掉该位。

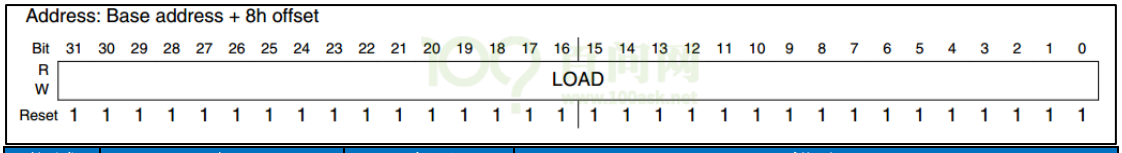



| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [0] | OCIF | R/W | Output compare interrupt flag，<br>0：比较事件未发生；<br>1：比较事件已发生 |

## 12.6.3 Load register (EPITx_LR)

EPIT 加载寄存器，如果 EPIT_CR 的 RLD 置位的话，当 EPIT 计数器计数到 0 时，会将 EPIT_LR 值加载到计数器中。

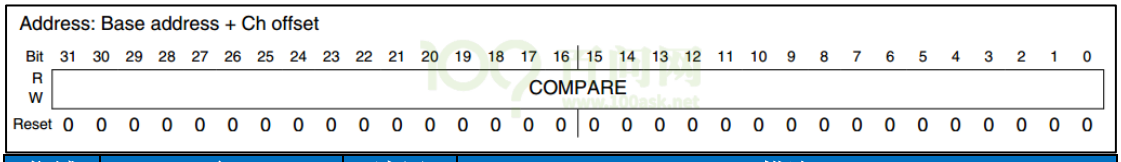
如果设置了 IOVW 位，往该寄存器写值会同时覆盖掉计数器的值。这个覆盖特性与 RLD 是否设置无关。




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:0] | LOAD | R/W | 加载值 |

### 12.6.4 Compare register (EPITx_CMPR)

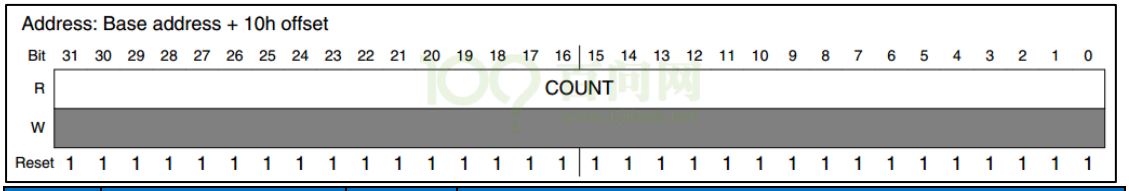
EPIT 比较寄存器，当主计数器的值等于比较寄存器时，产生比较事件。




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:0] | COMPARE | R/W | 比较值 |

### 12.6.5 Counter register (EPITx_CNR)

EPIT 计数器，主计数值，它是只读寄存器；读取 EPIT 计数器的值，不影响计数过程。如果控制寄存器的 IOVW 位设置的话，当往加载寄存器 EPIT_IR 写值时会覆盖当前的计数值。




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:0] | COUNT | R/W | EPIT 计数器的值 |

## 12.7 EPIT 查询方式延时代码详解

### 12.7.1 代码分析

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/003_timer_epit_poll”目录下。

程序的思路是使用 epit_poll_init 函数进行初始化，再用 epit_poll_restart 函数设置延时值。

➢ **epit_poll_init 函数**

首先看看 epit_poll_init 函数，它用于初始化 EPIT 定时器：

① 首先对 EPIT 进行软件复位，
② 设置 EPIT 为 set-and-forget 模式，
③ 设置在各种模式下，EPIT 仍然运行，
④ 时钟源选择 Peripheral Clock (ipg_clk)为 66M，预分频值设置为 0（即预分频值为 1），
⑤ 使能 overwrite，先不设置 EPITx_LR 寄存器；
⑥ 设置比较寄存器 EPITx_CMPR 为 0，当计数值减小到 0 时触发中断，
⑦ 最后使能 OCIEN，打开输出比较中断。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/003_timer_epit_poll/epit.c”。

```
void epit_poll_init(EPIT_Type *base)
{
    base->CR = 0;

    /* software reset
     * bit16
     */
    base->CR |= (1 << 16);
    /* wait for software reset self clear*/
    while((base->CR) & (1 << 16))
        ;

    /*
     * EPIT_CR
     * bit21 stopen; bit19 waiten; bit18 debugen
     * bit17 overwrite enable; bit3 reload
     * bit2 compare interrupt enable; bit1 enable mode
     */
    base->CR |= (1 << 21) | (1 << 19) | (1 << 17) | (1 << 3) | (1 << 1);

    /*
     * EPIT_CR
     * bit25-24: 00 off, 01 peripheral clock(ipg clk), 10 high, 11 low
     * bit15-4: prescaler value, divide by n+1
     */
    base->CR &= ~((0x3 << 24) | (0xFFF << 4));
    base->CR |= (1 << 24);

    /* EPIT_CMPR: compare register */
    base->CMPR = 0;
    /* EPIT_LR: load register , assue use ipc clk 66MHz*/
    //base->LR = USEC_TO_COUNT(us);

    /*  EPIT_CR bit2 OCIEN compare interrupt enable */
    base->CR |= (1 << 2);
}
```

- **epit_poll_restart 函数**

epit_poll_restart 函数用来设置 EPITx_LR(想延时多少 us？)，并启动定时器：

① 根据传输入的延时时间值，设置 EPITx_LR 寄存器；
② 为了防止状态寄存器已经设置，先清除一下状态寄存器对应的位；
③ 使能 GPT 计数器；
④ 循环判断状态寄存器 OCIF 位，等待比较事件发生；
⑤ 清除状态位 OCIF。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/003_timer_epit_poll/epit.c”。

```
void epit_poll_restart(EPIT_Type *base, unsigned int us)
{
    epit_run(base, 0);
    /* EPIT_LR: load register , assue use ipc clk 66MHz*/
    base->LR = USEC_TO_COUNT(us);
```

```
    /* write 1 clear it, avoid it happened before */
    EPIT1->SR |= (1 << 0);
    epit_run(base, 1);
    /* wait compare event happened*/
    while(!(EPIT1->SR & 0x1))
        ;
    /* write 1 clear it */
    EPIT1->SR |= (1 << 0);
}
```

- **epit_run 函数**

epit_run 函数用来使能和关闭 EPIT。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/003_timer_epit_poll/epit.c”。

```
void epit_run(EPIT_Type *base, int on)
{
    /*  EPIT_CR bit0 EN */
    if (on)
        base->CR |= (1 << 0);
    else
        base->CR &= ~(1 << 0);
}
```

- **主函数**

使用 epit_poll_init 初始化 EPIT1，不断调用 epit_poll_restart(EPIT1, 1000000)延时 1s，并且依次点亮和关闭绿灯。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/003_timer_epit_poll/main.c”。

```
epit_poll_init(EPIT1);
while(1) {
    epit_poll_restart(EPIT1, 1000000);
    GPIO5->DR &= ~(1<<3); //led on
    printf("led is on\r\n");
    epit_poll_restart(EPIT1, 1000000);
    GPIO5->DR |= (1<<3); //led off
    printf("led is off\r\n");
}
```

### 12.7.2 参考章节《4.3.4 编译程序》编译程序

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/003_timer_epit_poll”目录下。

### 12.7.3 参考章节《3.4 映像文件烧写、运行》烧写、运行程序

此时观察串口及开发板 led 灯现象：绿灯不断闪烁，每隔 1s 翻转状态；串口输出不断打印”led is on”和“led is off”。

## 12.8 EPIT 中断实现延时代码详解

### 12.8.1 EPIT1 中断号的确定

查询数据手册 chapter 3 的 Table3-1 中断号表，EPIT1 如下所示：

| | | | |
|---|---|---|---|
| 56 | epit1 | - | EPIT1 output compare interrupt |

GIC 中断号需要再这个序号基础上加上 32，所以 EPIT1 的 gic 中断号为 56+32=88。

## 12.8.2 代码分析

本程序的思路是根据延时时间设置 EPITx_LR 寄存器，并设置 EPITx_CMPR 为 0，当计数器递减到 0 时产生比较事件，触发中断。

- **初始化 EPIT1**

首先看看 epit_poll_init 函数，它用于初始化 EPIT 定时器：

① 首先对 EPIT 进行软件复位，
② 设置 EPIT 为 set-and-forget 模式，
③ 设置在各种模式下，EPIT 仍然运行，
④ 时钟源选择 Peripheral Clock (ipg_clk)为 66M，预分频值设置为 0（即预分频值为 1），
⑤ 使能 overwrite，先不设置 EPITx_LR 寄存器；
⑥ 设置比较寄存器 EPITx_CMPR 为 0，当计数值减小到 0 时触发中断，
⑦ 根据延时时间，设置 EPITx_LR 寄存器。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/004_timer_epit_int”目录下。

```
epit_init 代码在 epit.c 中：
/* assume use ipc clk which is 66MHz, 1us against to 66 count */
#define USEC_TO_COUNT(us) (us * 66 - 1)

void epit_init(EPIT_Type *base, unsigned int us)
{
    base->CR = 0;

    /* software reset
     * bit16
     */
    base->CR |= (1 << 16);
    /* wait for software reset self clear*/
    while((base->CR) & (1 << 16))
        ;

    /*
     * EPIT_CR
     * bit21 stopen; bit19 waiten; bit18 debugen
     * bit17 overwrite enable; bit3 reload
     * bit2 compare interrupt enable; bit1 enable mode
     */
    base->CR |= (1 << 21) | (1 << 19) | (1 << 3) | (1 << 1);

    /*
     * EPIT_CR
     * bit25-24: 00 off, 01 peripheral clock(ipg clk), 10 high, 11 low
     * bit15-4: prescaler value, divide by n+1
     */
    base->CR &= ~((0x3 << 24) | (0xFFF << 4));
    base->CR |= (1 << 24);

    /* EPIT_CMPR: compare register */
    base->CMPR = 0;
```

```
    /* EPIT_LR: load register , assue use ipc clk 66MHz*/
    base->LR = USEC_TO_COUNT(us);
}
```

- **打开或者关闭比较中断**

epit_enable_interrupt 函数用来打开或者关闭比较中断。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/004_timer_epit_int/epit.c”。

```
void epit_enable_interrupt(EPIT_Type *base, int on)
{
    /*  EPIT_CR bit2 OCIEN compare interrupt enable */
    if (on)
        base->CR |= (1 << 2);
    else
        base->CR &= ~(1 << 2);
}
```

- **EPIT 运行使能函数**

启动或停止 EPIT 定时器。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/004_timer_epit_int/epit.c”。

```
void epit_run(EPIT_Type *base, int on)
{
    /*  EPIT_CR bit0 EN */
    if (on)
        base->CR |= (1 << 0);
    else
        base->CR &= ~(1 << 0);
}
```

- **中断处理函数**

EPIT1_handle_irq 是中断处理函数，它首先清除中断状态位，然后翻转绿灯的状态。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/004_timer_epit_int/main.c”。

```
void EPIT1_handle_irq(void)
{
    static int on = 1;

    printf("EPIT1 interrupt happened\r\n");
    /* write 1 clear it */
     EPIT1->SR |= (1 << 0);

    /* read GPIO5_DR to get GPIO5_IO01 status*/
    if(on) {
        /* led off, set GPIO5_DR to configure GPIO5_IO03 output 1 */
        GPIO5->DR |= (1<<3); //led on
        on = 0;
    } else {
        /* led on, set GPIO5_DR to configure GPIO5_IO03 output 0 */
        GPIO5->DR &= ~(1<<3); //led off
        on = 1;
    }
}
```

➢ 主函数

主函数做了这些事情：

① 首先调用 epit_init(EPIT1, 1000000)设置延时时间为 1s，并初始化 EPIT；

② 然后调用 request_irq 注册中断处理函数，设置中断处理函数为 EPIT1_handle_irq；

③ 接下来使能中断：先在 GIC 中使能 EPIT1 中断，再在 EPIT1 中使能比较中断；

④ 最后调用 epit_run(EPIT1, 1)使能 EPIT 开始计数。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/004_timer_epit_int/main.c”。

```
epit_init(EPIT1, 1000000);// set 1s
request_irq(EPIT1_IRQn, (irq_handler_t)EPIT1_handle_irq, NULL);
gic_enable_irq(EPIT1_IRQn);
epit_enable_interrupt(EPIT1, 1);
epit_run(EPIT1, 1);
```

## 12.8.3 参考章节《4.3.4 编译程序》编译程序

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/12_定时器编程/004_timer_epit_int”目录下。

## 12.8.4 参考章节《3.4 映像文件烧写、运行》烧写、运行程序

此时观察串口及开发板 led 灯现象

绿灯不断闪烁，每隔 1s 翻转状态

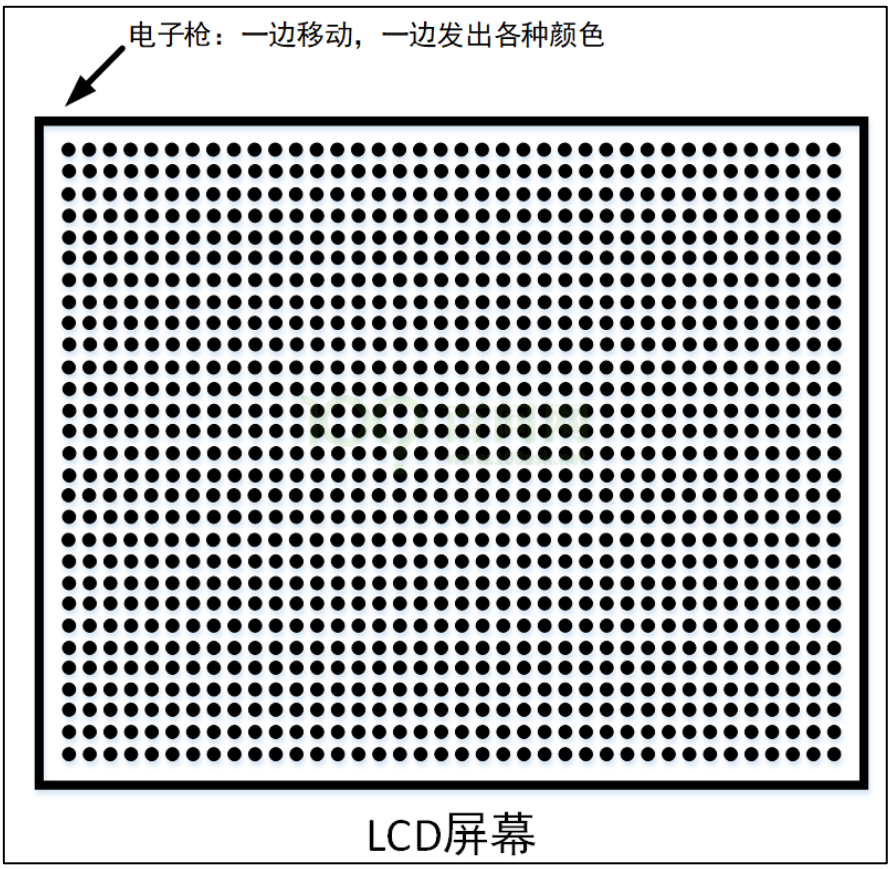
串口输出如下：

```
hello world
irq 88 is happened
EPIT1 interrupt happened
irq 88 is happened
EPIT1 interrupt happened
irq 88 is happened
EPIT1 interrupt happened
irq 88 is happened
EPIT1 interrupt happened
```

# 第13章 LCD 编程

## 13.1 LCD 硬件原理

### 13.1.1 LCD 硬件工作原理简介




假设上图是一个 LCD 屏幕，屏幕中一个一个密密麻麻的黑点称之为像素点，每一行有若干个点，试想下有一个电子枪，电子枪位于某一个像素点的背后，然后向这个像素发射红，绿，蓝三种原色，这三种颜色按不同的比例组合成任意一种颜色。电子枪在像素点的背后，一边移动一边发出各种颜色的光，电子枪从左往右移动，到右边边缘之后就跳到下一行的行首，继续从左往右移动，如此往复，一直移动到屏幕右下角的像素点，最后就跳回原点。

- **问题 1：电子枪如何移动？**

答：有一条像素时钟信号线（DCLK），连接屏幕，每来一个像素时钟信号（DCLK），电子枪就移动一个像素。

- **问题 2：电子枪打出的颜色该如何确定？**

答：有三组红，绿，蓝信号线（RGB），连接屏幕，由这三组信号线（RGB）传递颜色

- **问题 3：电子枪移动到 LCD 屏幕右边边缘时，如何得知需要跳到下一行的行首？**

答：有一条水平同步信号线（HSYNC），连接屏幕，当接收到水平同步信号（HSYNC），电子枪就跳到下一行的最左边

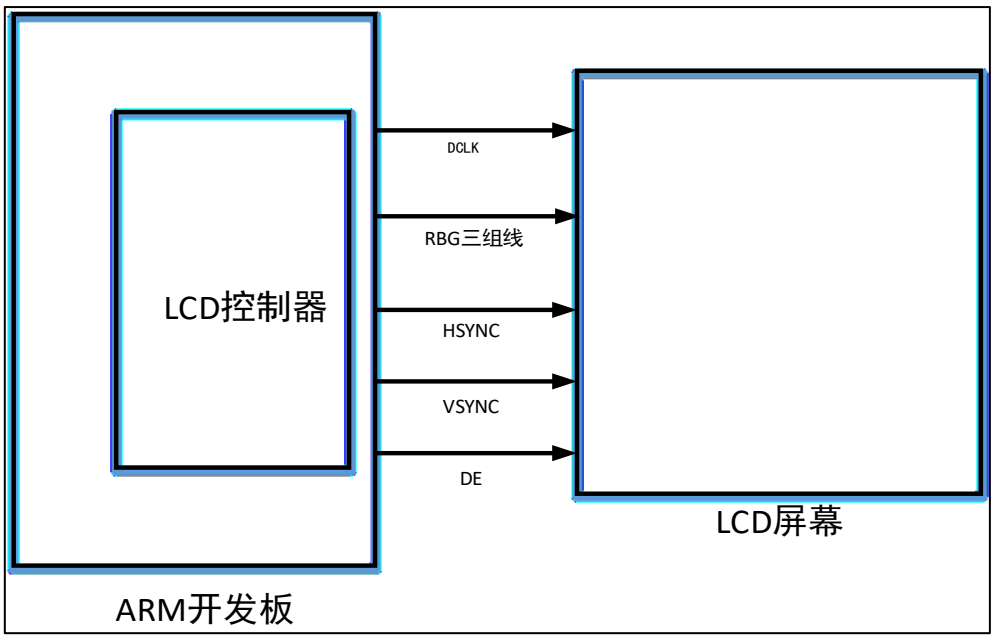
➢ **问题 4：电子枪如何得知需要跳到原点？**

答：有一条垂直同步信号线（VSYNC），连接屏幕，当接收到垂直同步信号线（VSYNC），电子枪就由屏幕右下脚跳到左上角（原点）

➢ **问题 5：电子枪如何得知三组信号线（RGB）确定的颜色就是它是需要的呢？**

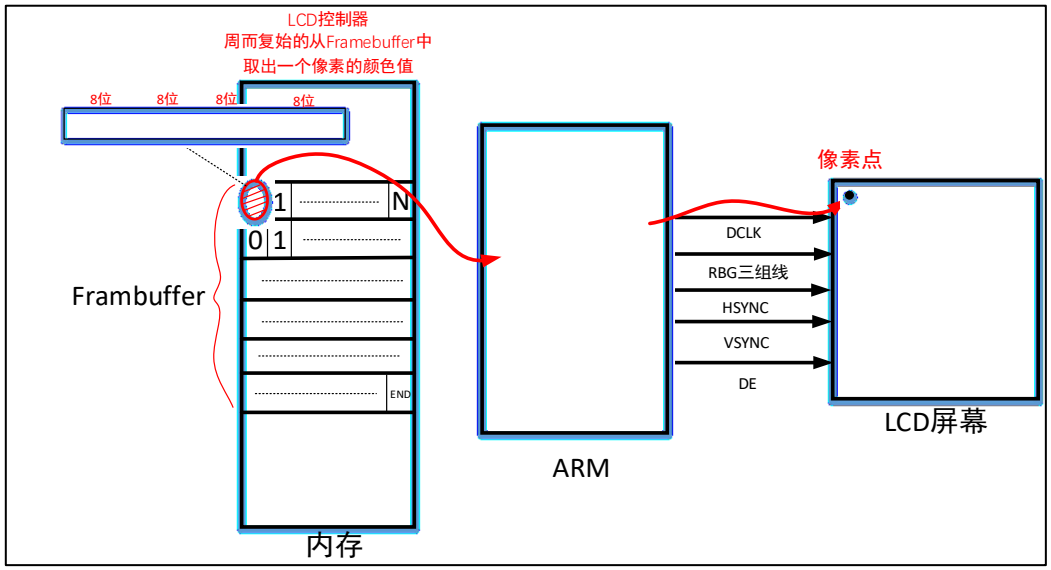
答：有一条 RGB 数据使能信号线（DE），连接屏幕，当接收到数据使能信号线（DE），电子枪就知道这时由这三组信号线（RGB）确定的颜色是有效的，可以发射到该像素点。

下图是开发板，LCD 控制器，LCD 屏幕的框图




之前提到的像素时钟（DCLK）， 三组红，绿，蓝信号线（RGB），水平同步信号线（HSYNC），垂直同步信号线（VSYNC），RGB 数据使能信号线（DE）都是从 LCD 控制器发出的，只要开发板支持 LCD 显示，它肯定就会有一个 LCD 控制器。

➢ **问题 6：RGB 三组信号线上的数据从何而来？**




上图是 RGB 数据来源框图，内存中划出一部分区域，这块区域被称为

Framebuffer。在 Framebuffer 里面我们会构造好每一个颜色所对应的像素。Framebuffer 中的值会被 LCD 控制器读出来，通过 RGB 三组线传给电子枪，电子枪再把它转换成红绿蓝三种颜色打到屏幕上。在屏幕上的每一个像素，在 Frambuffer 里面都有一个对应存储空间，里面存有屏幕上对应像素的颜色。

LCD 控制器会周而复始的从 Framebuffer 中取出一个个像素的颜色值，发给电子枪，同时需要和 DCLK，VSYNC，HSYNC，DE 这些信号配合好。

### 13.1.2 RGB 接口的 LCD 硬件连接信号

本次实验编程的屏幕属于 RGB 接口的显示屏，RGB 接口的显示屏至少具备以下信号：

① 像素时钟信号(DCLK):用于同步 LCD 上的 DE，VS，HS，RGB 信号线。

② RGB 数据信号（R[0:7] ,G[0:7],B[0:7]）

三组信号线组成，分别代表 R(红色)，G(绿色)，B（蓝色），这三组信号中的每一组都会有 8 根信号，三组共同组成 24 根线来控制颜色数据。

有些 LCD 只需要 16 位颜色(RGB565)，可以只使用 8 条红色数据线中的高 5 位，其余 3 位悬空；只使用 8 条绿色数据线中的高 6 位，其余 2 位悬空；只使用 8 条蓝色数据线中的高 5 位，其余 3 位悬空。

③ RGB 数据使能信号（DE）

RGB 接口的 LCD 有两种驱动模式：DE 模式和 HV 模式。

在 HV 模式下，需要用到 HS 与 VS 来控制刷新。比如对于分辨率为 1024x600RGB 的 LCD，LCD 控制器发出 HS 信号后，就会发出 1024 个 DCLK，在每个 DCLK 上传输像素数据；当发出 600 个 HS 信号后，就会发出一个 VS 信号。

在 DE 模式下，需要用到 DE 信号来控制刷新，比如对于分辨率为 1024x600RGB 的 LCD，LCD 控制器发出 DE 信号后，就要发出 1024 个 DCLK，在每个 DCLK 上传输像素数据；当发出 600 个 DE 信号，刷新完一帧数据后，又从第 1 行开始扫描。

编写 LCD 程序时，一般都会兼容两种模式，所以程序中会设置这 3 个信号：数据使能信号（DE）,垂直同步信号（HS）,水平同步信号（VS）。

④ 水平同步信号

电路中常用 HS 或 HSYNC 表示，详细说明下一小节会说明。

⑤ 垂直同步信号（帧同步或场同步）

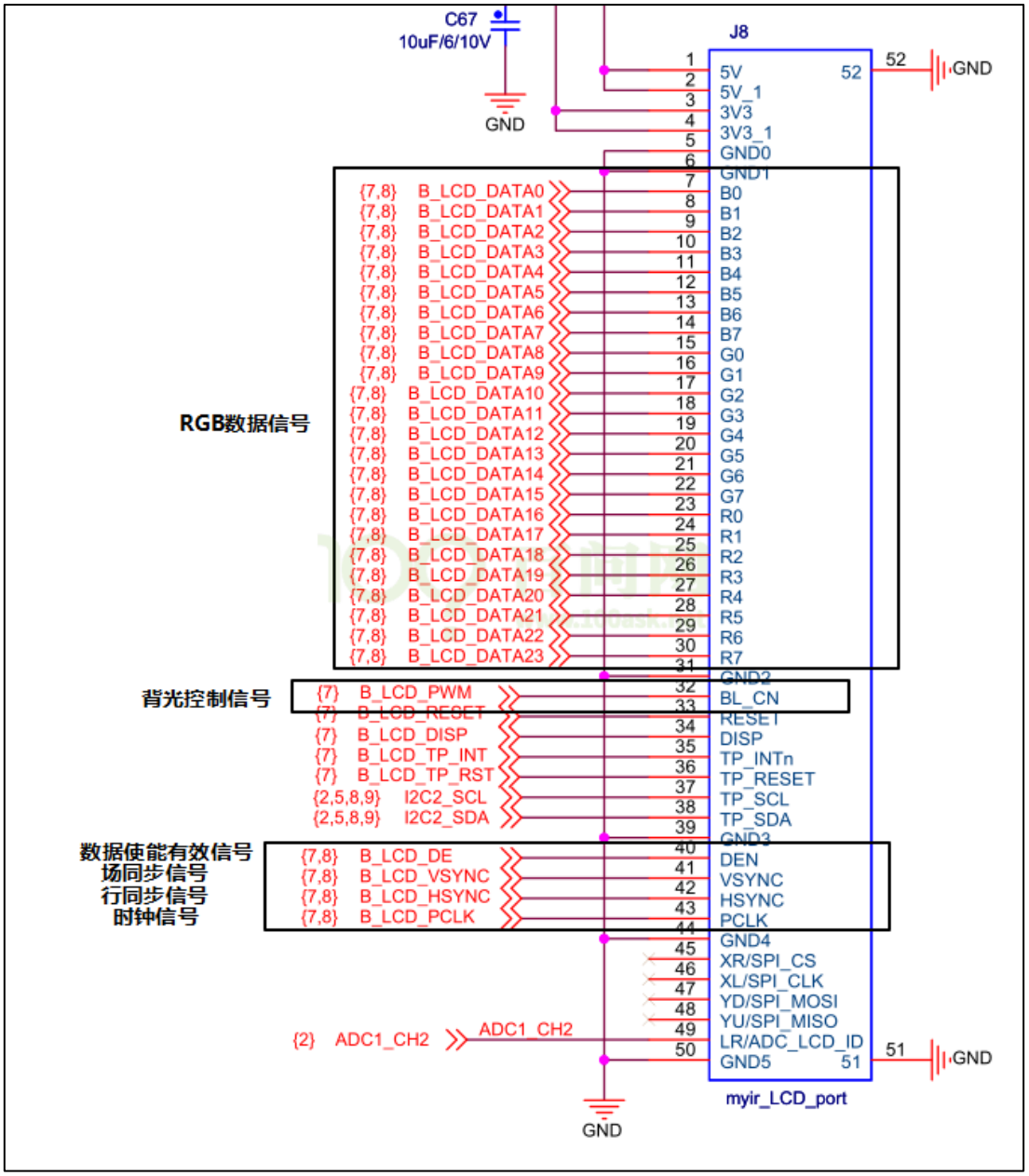
⑥ 电路中常用 VS 或 VSYNC 表示，详细说明下一小节会说明。

⑦ LCD 背光电源控制信号

所谓背光，就是在 LCD 显示屏的背部有很多的灯珠，用它们来照亮屏幕。

背光电路可以用 GPIO 引脚控制，输出高低电平表示亮灭，只有亮灭两种状态；也可以用 PWM 引脚来控制，它可以输出占空比不同的方波，根据占空比来调节 LCD 的亮度，更加精细。

100ASK_IMX6ULL 开发板的 LCD 接口定义，就包含了上面所述的几种信号类型，如下图：




## 13.1.3 TFT 材质液晶屏接口简介(7 寸 1024600TN-RGB)

参考资料：网盘开发板配套资料“06_Datasheet（数据手册）/Extend_modules/7 寸 LCD 模块.zip”，解压后里面的“7.0-13SPEC(7 寸 1024600TN-RGB).pdf”。

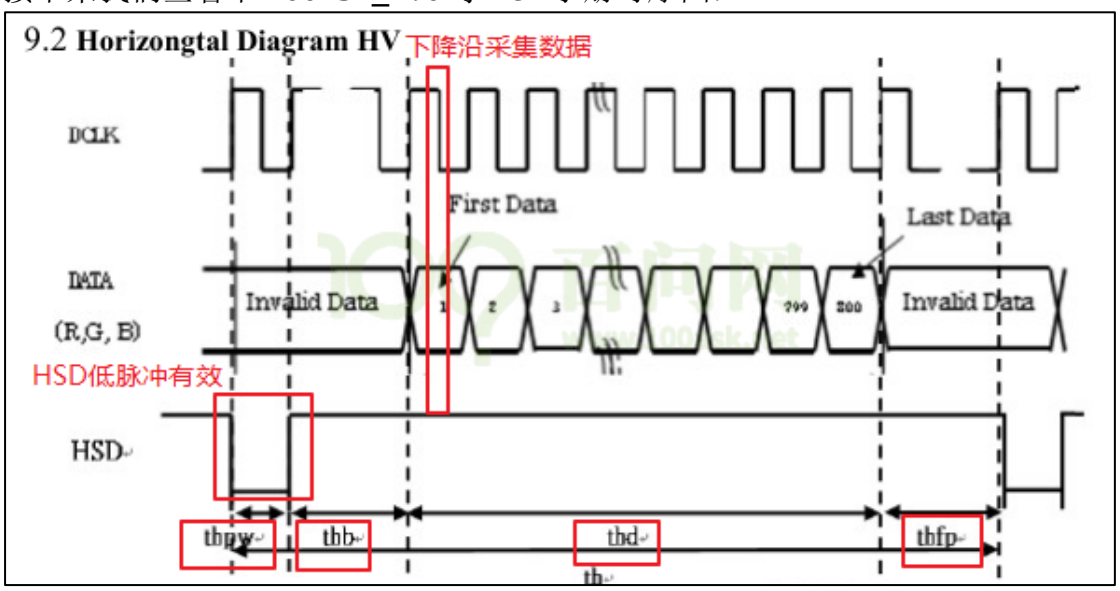
嵌入式设备中一般都采用 TFT 材质的液晶屏，如遇到别的材质的屏幕，操作方法也是相似的，可能稍微有些差异，针对差异去做修改即可，7 寸 1024600TN-RGB 液晶屏幕接口引脚如下图，一些关键的引脚做了注释。

| | | | |
|---|---|---|---|
| LED+ | 1 | P | Backlight LED Cathode input pin (+) 背光 |
| LED+ | 2 | P | |
| LED- | 3 | P | Backlight LED Anode input pin (-) |
| LED- | 4 | P | |
| GND | 5 | P | Ground |
| VCOM | 6 | P | Common voltage |
| VDD | 7 | P | Power supply for digital block (+3.3V) |
| MODE | 8 | I | DE/SYNC mode select DE模式或HV模式 |
| DE | 9 | I | Data Enable 数据使能 |
| VSYNC | 10 | I | Vertical Synchronize Singal 场同步信号 |
| HSYNC | 11 | I | Horizontal Synchronize Singal 行同步信号 |
| B7 | 12 | I | Blue Data |
| B6 | 13 | I | Blue Data |
| B5 | 14 | I | Blue Data |
| B4 | 15 | I | Blue Data |
| B3 | 16 | I | Blue Data |
| B2 | 17 | I | Blue Data |
| B1 | 18 | I | Blue Data |
| B0 | 19 | I | Blue Data |
| G7 | 20 | I | Green Data |
| G6 | 22 | I | Green Data |
| G5 | 22 | I | Green Data RGB数据 |
| G4 | 23 | I | Green Data |
| G3 | 24 | I | Green Data |
| G2 | 25 | I | Green Data |
| G1 | 26 | I | Green Data |
| G0 | 27 | I | Green Data |
| R7 | 28 | I | Red Data |
| R6 | 29 | I | Red Data |
| R5 | 30 | I | Red Data |
| R4 | 31 | I | Red Data |
| R3 | 32 | I | Red Data |
| R2 | 33 | I | Red Data |
| R1 | 34 | I | Red Data |
| R0 | 35 | I | Red Data |
| GND | 36 | P | Ground |
| CLK | 37 | I | Pixel Clock 像素时钟 |
| GND | 38 | P | Ground |
| SHLR | 39 | I | Left / right selection 电子枪扫描方向 |
| UPDN | 40 | I | Up/down selection |
| VGH | 41 | P | Gate ON Voltage 偏置电压 |
| VGL | 42 | P | Gate OFF Voltage |
| VDDA | 43 | P | Power supply for analog block |
| RESET | 44 | I | Global reset pin. |
| NC | 45 | - | No Connection |
| VCOM | 46 | P | Common Voltage |
| DITHER | 47 | I | Dithering function<br>DITHER = "1", Enable internal dithering function<br>DITHER = "0", will bypass R0/R1、G0/RG、B0/B1 |
| GND | 48 | P | Ground |
| NC | 49 | - | No Connection |
| NC | 50 | - | No Connection |

## 13.1.4 LCD 关键特性

➢ 行时序

接下来我们查看下 100ASK_7.0 寸 LCD 手册时序图：



如上图所示，从最小的像素开始分析，电子枪每次在 CLK 下降沿采集数据，即从数据线上得到数据，并发射到显示屏上，然后移动到下一个位置。DATA 数据线上的数据来源就是前面介绍的 FrameBuffer。就这样从一行的最左边，一直移动到一行的最右边，完成了一行的显示。

电子枪当打完一行的最后一个数据后，就会收到 Hsync 行同步信号，如上图可知该 LCD 的 Hsync 有效脉冲为低脉冲，根据时序图，一个行周期可以大致分为四部分组成：

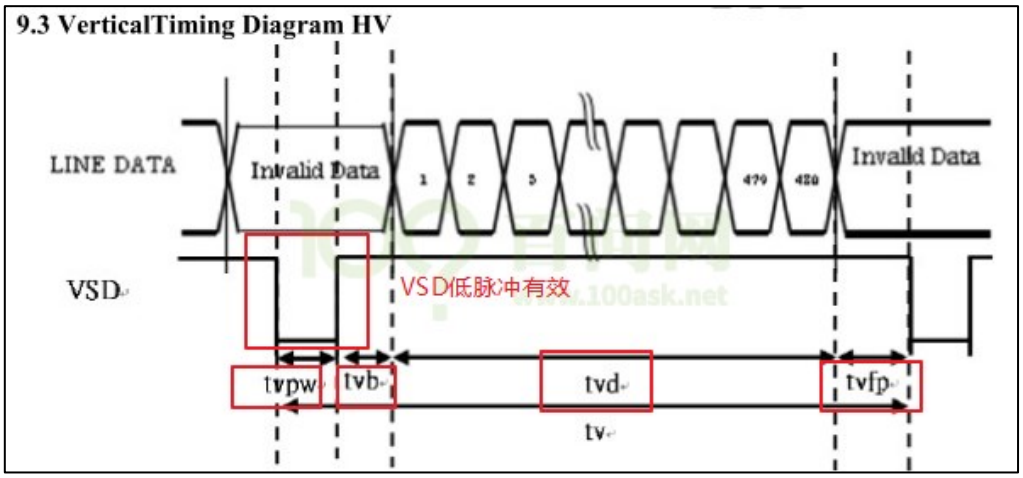
① thp：horizontal pulse：Hsync 信号的脉冲，thpw 称为脉冲宽度，这个时间不能太短，太短电子枪可能识别不到。

② thb：horizontal back porch

③ thf：horizontal front porch：Hsync 信号发出后，电子枪从最右端移动到最左端，这是需要时间的：这个移动的时间就是 thb。从 Hsync 结束到 DE 开始的区间被称为行扫描的后肩(back porch)，从 DE 结束到 Hsync 开始称为前肩(front porch)。

④ thd：horizontal data：thd 为数据有效区，假设一行的有效像素个数为 x。

➢ 列时序




同理，电子枪被 Hsync 驱动，一行一行地从上面移动到最下面时，它会收到一个 Vsync 信号(上图中标为 VSD)：你应该回到原点去了。如上图可知该 LCD 的 VSD 有效脉冲为低脉冲，然后就让电子枪移动回最上边。VSD 中的 tvpw 是脉冲宽度，tvb 是移动时间，tvfp 表示显示完最下一行像素，再过多久 VSD 才来，tvd 为数据有效区，tv 为打完一帧所需要的时间。假设一共有 y 行，则 LCD 的分辨率就是 x*y。

电子枪打完一帧的最后一个数据后，就会收到 Vsync 行同步信号。根据时序图，一帧的周期可以大致分为四部分组成：

① tvp：vertical pulse：Vsync 信号的脉冲，tvpw 称为脉冲宽度，这个时间不能太短，太短电子枪可能识别不到。

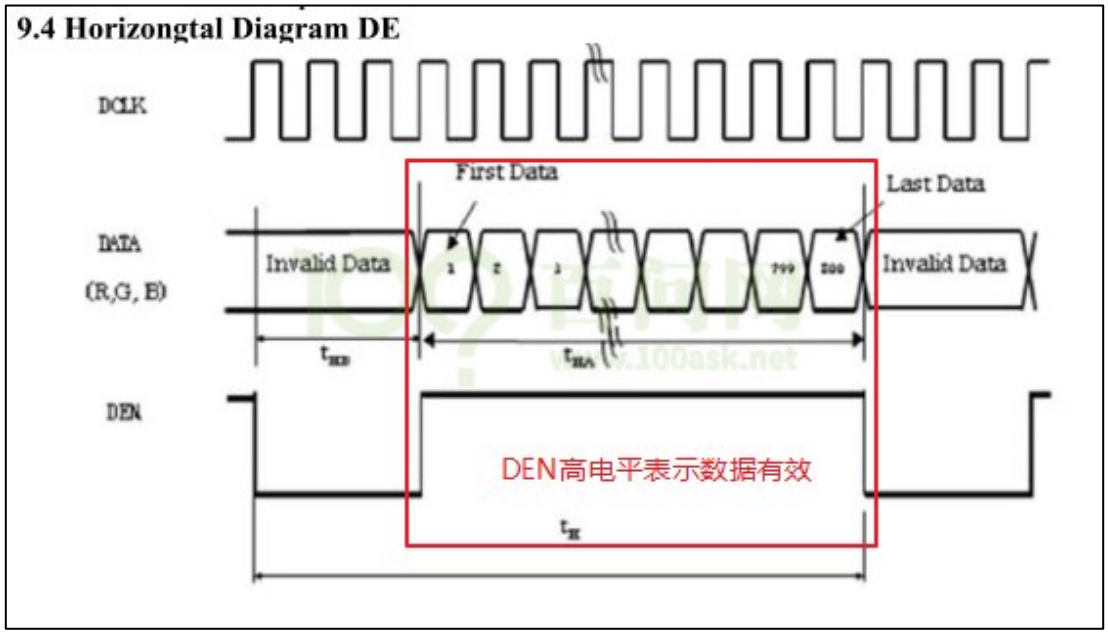
② tvb：vertical back porch

③ tvf：vertical front porch：Vsync 信号发出后，电子枪从最底端移动到最上端，这是需要时间的：这个移动的时间就是 tvb。在电子枪移动的过程中，Hsync 信号会一直有，但是 RGB 上没有有效的数据。假设一帧信号里，含有有效数据的第 1 个 Hsync 信号发生时间称为 T1；一帧里，最后一行有效数

据的下一个 Hsync 信号发生的时间称为 T2。从 Vsync 结束到 T1 的这段时间，被称为列扫描的后肩(back porch)，从 T2 到 Vsync 开始称为前肩(front porch)。

④ tvd：vertical data：tvd 为数据有效区，假设一帧数据里的有效行数为 y。：LCD 的分辨率是：x*y。

- **DE 模式**




RGB 数据有效信号（DEN），高电平表示数据有效。

根据以上信息大致了解几个关键信号的时序和极性，后面章节会详细介绍。

| ITEM | SYMBOL | MIN. | TYP. | MAX. | UNIT | Note |
|---|---|---|---|---|---|---|
| Dot Clock | 1/tCLK | 45 | 51.2 | 57 | MHz | |

再根据上图，我们就可以确定像素时钟是 51.2Mhz。

- **RGB 数据的存放形式**

前面的 LCD 硬件接口图里用到了 24 条数据线：R0-R7、G0-G7、B0-B7，每个像素的颜色占据 3*8=24 位。硬件上 LCD 的数据引脚连接是固定的，但是我们使用的时候，在软件上可以支持不同的像素格式，比如 ARGB888，ARGB555，RGB565 等等。也就是说虽然硬件上用了 24 条数据线，但是软件上我们可以提供 24 位数据，也可以只提供 16 位数据。当只提供 16 位数据时，硬件上 24 位数据线中会有 8 条数据线上没有数据。比如对于 RGB565 格式，R0R1R2、G0G1、B0B1B2 这 8 条数据线上是不传输数据的，恒为 0。

本实验支持 ARGB888 和 ARGB555。

ARGB888：每个像素就占据 32 位数据，其中最高字节 A 表示灰度透明度其余 RGB 数据 8+8+8=24BPP，因为硬件上根本没接透明度的数据线，所以透明度的数据没用上。

ARGB555：每个像素就占据 16 位数据，其中最高位 A 表示灰度透明度其余 RGB 数据 5+5+5=15BPP，但是 LCD 本身没有透明度的数据线，所以透明度数据没用上。

## 13.2 IMX6ULL LCD 控制器操作及寄存器

参考资料：网盘开发板配套资料“06_Datasheet（数据手册）/Core_board/CPU/IMX6ULLRM.pdf”：《Chapter 34: Enhanced LCD Interface (eLCDIF)》。

### 13.2.1 LCD 控制器模块介绍

IMX6ULL 的 LCD 控制器名称为 eLCDIF(Enhanced LCD Interface，增强型 LCD 接口)，主要特性如下：

① 支持 MPU 模式：有些显示屏自带显存，只需要把命令、数据发送给显示屏即可；

② 支持 DOTCLK 模式：RGB 接口，本实验就是此模式；

③ VSYNC 模式：针对高速数据传输（行场信号）；

④ 支持 ITU-R BT.656 接口，可以把 4:2:2 YcbCr 格式的数据转换为模拟电视信号；

⑤ 8/16/18/24/32 bit 的 bpp 数据都支持，取决于 IO 的复用设置及寄存器配置；

⑥ MPU 模式，VSYNC 模式，DOTCLK 模式，都可以配置时序参数。

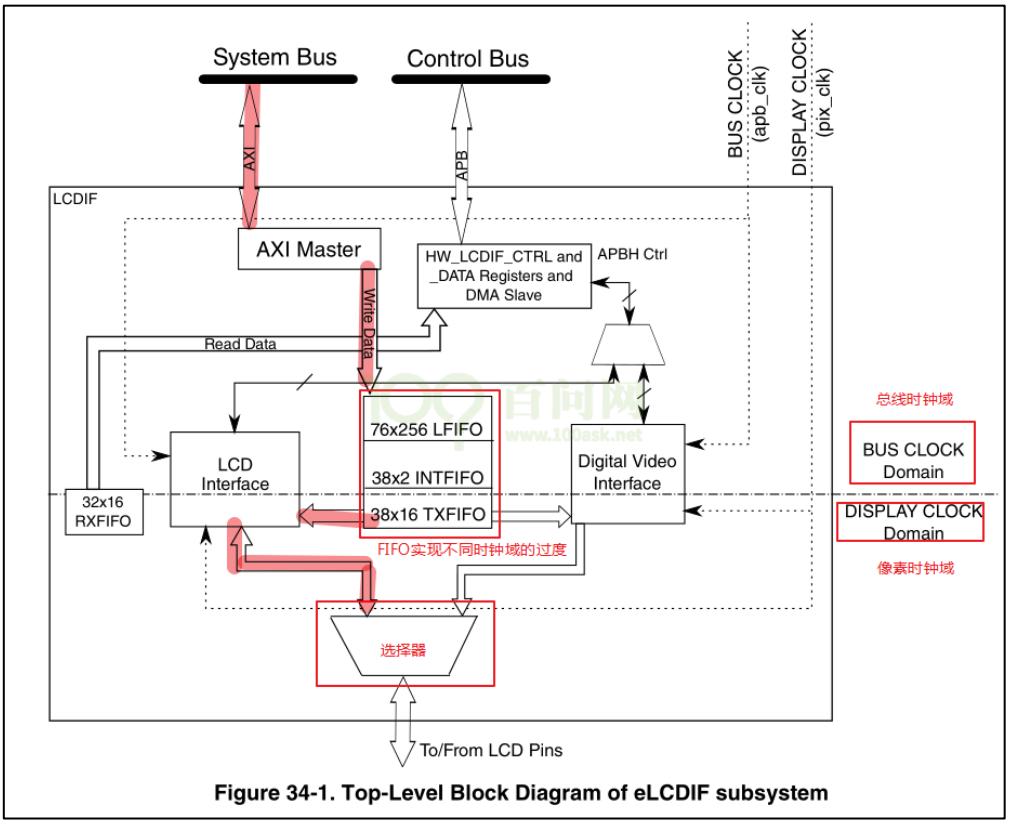



Figure 34-1. Top-Level Block Diagram of eLCDIF subsystem

上图是 IMX6ULL 的 LCD 控制器框图。

我们在内存中划出一块内存，称之为显存，软件把数据写入显存。

设置好 LCD 控制器之后，它会通过 AXI 总线协议从显存把 RGB 数据读入 FIFO，再到达 LCD 接口(LCD Interface)。本实验中，LCD 接口上连着 TFT LCD。

LCD 控制器有两个时钟域：外设总线时钟域，LCD 像素时钟域。前者是用来让 LCD 控制器正常工作，后者是用来控制电子枪移动。

上图的 Read_Data 操作，在 MPU 模式下才用到；我们采用的是 DCLK 模式，

因此不予考虑。

更详细的内容可以查看 IMX6ull 芯片手册《Chapter 34 Enhanced LCD Interface (eLCDIF)》。

### 13.2.2 LCD 控制器寄存器简介

查看任何芯片的 LCD 控制器寄存器时，记住几个要点：

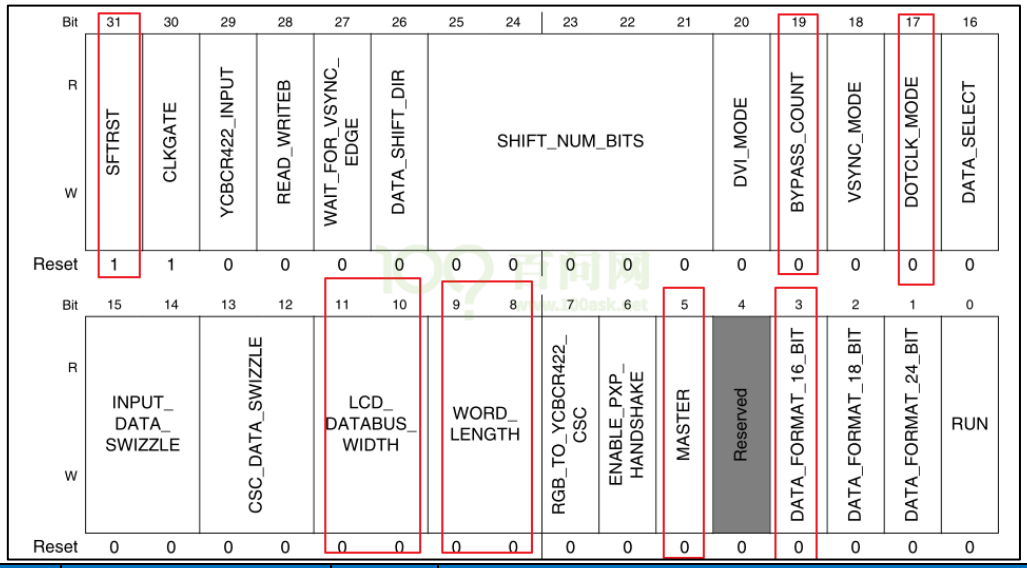
- 怎么把 LCD 的信息告诉 LCD 控制器：即分辨率、行列时序、像素时钟等；
- 怎么把显存地址、像素格式告诉 LCD 控制器。

**LCDIF memory map**

| Absolute address (hex) | Register name | Width (in bits) | Access | Reset value | Section/ page |
|---|---|---|---|---|---|
| 21C_8000 | eLCDIF General Control Register (LCDIF_CTRL) | 32 | R/W | C000_0000h | 34.6.1/2168 |
| 21C_8010 | eLCDIF General Control1 Register (LCDIF_CTRL1) | 32 | R/W | 000F_0000h | 34.6.2/2171 |
| 21C_8030 | eLCDIF Horizontal and Vertical Valid Data Count Register (LCDIF_TRANSFER_COUNT) | 32 | R/W | 0001_0000h | 34.6.4/2176 |
| 21C_8040 | LCD Interface Current Buffer Address Register (LCDIF_CUR_BUF) | 32 | R/W | 0000_0000h | 34.6.5/2176 |
| 21C_8050 | LCD Interface Next Buffer Address Register (LCDIF_NEXT_BUF) | 32 | R/W | 0000_0000h | 34.6.6/2177 |
| 21C_8070 | eLCDIF VSYNC Mode and Dotclk Mode Control Register0 (LCDIF_VDCTRL0) | 32 | R/W | 0000_0000h | 34.6.8/2178 |
| 21C_8080 | eLCDIF VSYNC Mode and Dotclk Mode Control Register1 (LCDIF_VDCTRL1) | 32 | R/W | 0000_0000h | 34.6.9/2180 |
| 21C_8090 | LCDIF VSYNC Mode and Dotclk Mode Control Register2 (LCDIF_VDCTRL2) | 32 | R/W | 0000_0000h | 34.6.10/ 2180 |
| 21C_80A0 | eLCDIF VSYNC Mode and Dotclk Mode Control Register3 (LCDIF_VDCTRL3) | 32 | R/W | 0000_0000h | 34.6.11/ 2181 |
| 21C_80B0 | eLCDIF VSYNC Mode and Dotclk Mode Control Register4 (LCDIF_VDCTRL4) | 32 | R/W | 0000_0000h | 34.6.12/ 2182 |
| 21C_80C0 | Digital Video Interface Control0 Register (LCDIF_DVICTRL0) | 32 | R/W | 0000_0000h | 34.6.13/ 2183 |
| 21C_80D0 | Digital Video Interface Control1 Register (LCDIF_DVICTRL1) | 32 | R/W | 0000_0000h | 34.6.14/ 2183 |
| 21C_80E0 | Digital Video Interface Control2 Register (LCDIF_DVICTRL2) | 32 | R/W | 0000_0000h | 34.6.15/ 2184 |
| 21C_80F0 | Digital Video Interface Control3 Register (LCDIF_DVICTRL3) | 32 | R/W | 0000_0000h | 34.6.16/ 2185 |
| 21C_8100 | Digital Video Interface Control4 Register (LCDIF_DVICTRL4) | 32 | R/W | 0000_0000h | 34.6.17/ 2186 |

上图是我们将要使用到的寄存器，下面逐个讲解这些寄存器，在后续的 LCD 控制编程实验会用到。

## ➢ LCDIF_CTRL 寄存器

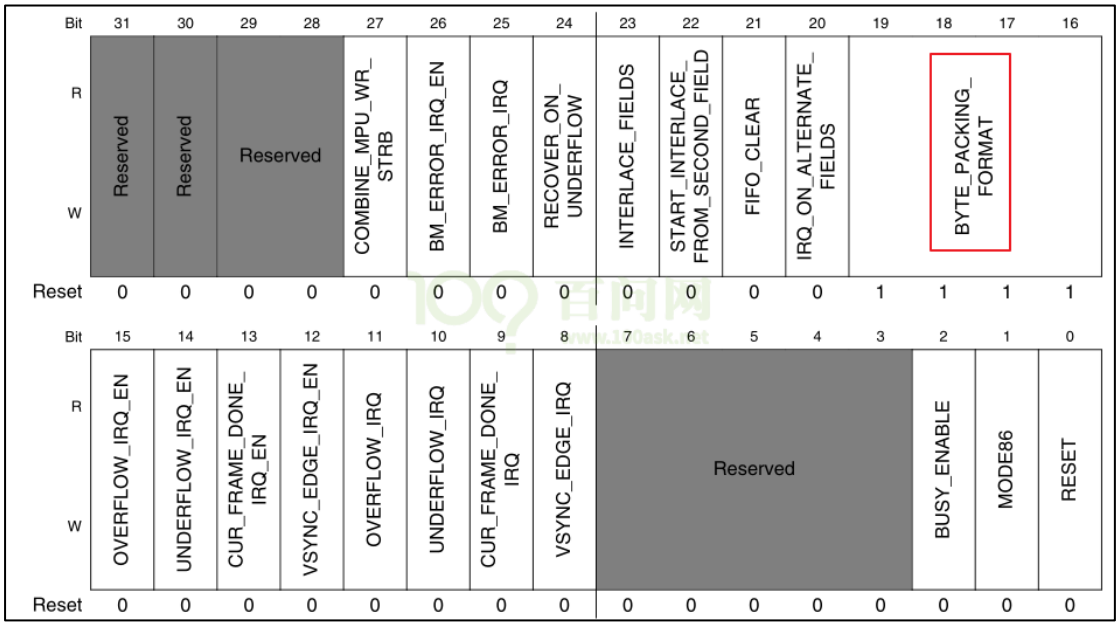



| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31] | SFTRST | R/W | 软件复位，正常工作时应设为 0；如果设为 1，它会复位整个 LCD 控制器 |
| [30] | CLKGATE | R/W | 时钟开关，<br>0：正常工作时要设置为 0；<br>1：关闭 LCD 控制器时钟 |
| [29] | YCBCR422_INPUT | R/W | 使用 RGB 接口时，设置为 0；其他接口我们暂时不关心 |
| [28] | READ_WRITEB | R/W | 使用 RGB 接口时，设置为 0；其他接口我们暂时不关心 |
| [27] | WAIT_FOR_VSYNC_EDGE | R/W | 在 VSYNC 模式时，设置为 1；我们不关心 |
| [26] | DATA_SHIFT_DIR | R/W | 在 DVI 模式下才需要设置，我们不关心 |
| [25:21] | SHIFT_NUM_BITS | R/W | 在 DVI 模式下才需要设置，我们不关心 |
| [20] | DVI_MODE | R/W | 设置为 1 时，使用 DVI 模式，就是 ITU-R BT.656 数字接口 |
| [19] | BYPASS_COUNT | R/W | DOTCLK 和 DVI 模式下需要设置为 1;MPU、VSYNC 模式时设为 0 |
| [18] | VSYNC_MODE | R/W | 使用 VSYNC 模式时，设置为 1 |
| [17] | DOTCLK_MODE | R/W | 使用 DOTCLK 模式时，设置为 1；本实验用的就是这个模式 |
| [16] | DATA_SELECT | R/W | MPU 模式下才用到，我们不关心 |

| | | | |
|---|---|---|---|
| [15:14] | INPUT_DATA_SWIZZLE | R/W | 显存中像素颜色的数据转给 LCD 控制器时，字节位置是否交换：<br>0x0：NO_SWAP，不交换；<br>0x0：LITTLE_ENDIAN，小字节序，跟 NO_SWAP 一样；<br>0x1：BIG_ENDIAN_SWAP，字节 0、3 交换；字节 1、2 交换；<br>0x1：SWAP_ALL_BYTES，字节 0、3 交换；字节 1、2 交换；<br>0x2：HWD_SWAP，半字交换，即 0x12345678 转为 0x56781234<br>0x3：HWD_BYTE_SWAP，在每个半字内部放换字节，<br>即 0x12345678 转换为 0x34127856 |
| [13:12] | CSC_DATA_SWIZZLE | R/W | 显存中的数据被传入 LCD 控制器内部并被转换为 24BPP 后，在它被转给 LCD 接口之前，字节位置是否交换：<br>0x0：NO_SWAP，不交换；<br>0x0：LITTLE_ENDIAN，小字节序，跟 NO_SWAP 一样；<br>0x1：BIG_ENDIAN_SWAP，字节 0、3 交换；字节 1、2 交换；<br>0x1：SWAP_ALL_BYTES，字节 0、3 交换；字节 1、2 交换；<br>0x2：HWD_SWAP，半字交换，即 0x12345678 转为 0x56781234<br>0x3：HWD_BYTE_SWAP，在每个半字内部放换字节，<br>即 0x12345678 转换为 0x34127856 |
| [11:10] | LCD_DATABUS_WIDTH | R/W | LCD 数据总线宽度，就是对外输出的 LCD 数据的位宽，<br>0x0：16 位；<br>0x1：8 位；<br>0x2：18 位；<br>0x3：24 位 |
| [9:8] | WORD_LENGTH | R/W | 输入的数据格式，即显存中每个像素占多少位，<br>0x0：16 位；<br>0x1：8 位；<br>0x2：18 位；<br>0x3：24 位 |
| [7] | RGB_TO_YCBCR422_CSC | R/W | 设置为 1 时，使能颜色空间转换：RGB 转为 YCbCr |

| | | | |
|---|---|---|---|
| [6] | ENABLE_PXP_HANDSHAKE | R/W | 当 LCDIF_MASTER 设置为 1 时，再设置这位，<br>则 LCD 控制器跟 PXP 之间的握手机制被关闭(我们不关心) |
| [5] | MASTER | R/W | 设置为 1 时，LCD 控制器成为 bus master |
| [4] | RSRVD0 | R/W | 保留 |
| [3] | DATA_FORMAT_16_BIT | R/W | WORD_LENGTH 为 0 时，表示一个像素用 16 位，此位作用如下：<br>0：数据格式为 ARGB555；<br>1：数据格式为 RGB565 |
| [2] | DATA_FORMAT_18_BIT | R/W | WORD_LENGTH 为 2 时，表示一个像素用 18 位，RGB 数据还是保存在 32 位数据里，此位作用如下：<br>0：低 18 位用来表示 RGB666，高 14 位无效<br>1：高 18 位用来表示 RGB666，低 14 位无效 |
| [1] | DATA_FORMAT_24_BIT | R/W | WORD_LENGTH 为 3 时，表示一个像素用 24 位，此位作用如下：<br>0:所有的 24 位数据都有效,格式为 RGB888<br>1：转给 LCD 控制器的数据是 24 位的，但只用到其中的 18 位，<br>每个字节用来表示一个原色，每字节中高 2 位无效 |
| [0] | RUN | R/W | 使能 LCD 控制器，开始传输数据 |

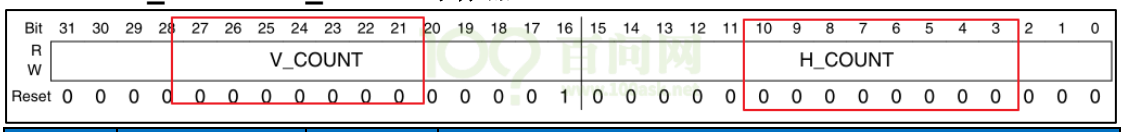
➢ **LCDIF_CTRL1 寄存器**




本实验中使用 TFT LCD，LCD 控制器使用 DOTCLK 模式。本寄存器中其他用不到的位，就不介绍了。

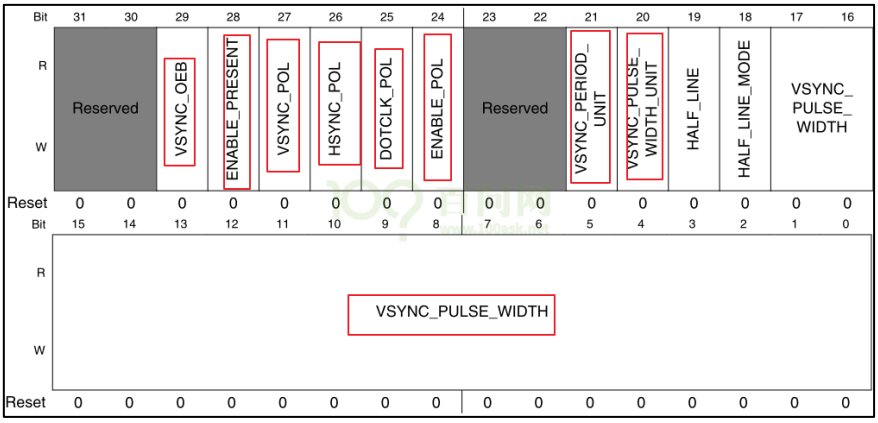
| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [19:16] | BYTE_PACKING_FORMAT | R/W | 用来表示一个 32 位的 word 中，哪些字节是有效的，即哪些字节是用来表示颜色的。<br>bit16、17、18、19 分别对应 byte0、1、2、3；某位为 1，就表示对应的字节有效。<br>默认值是 0xf，表示 32 位的 word 中，所有字节都有效。<br>对于 8bpp，可以忽略本设置，所有的字节都是有效的；<br>对于 16bpp，bit[1:0]、bit[3:2]分别对应一个字节，组合中的 2 位都为 1 时，对应的字节才有效；<br>对于 24bpp，0x7 表示 32 位数据中只用到 3 个字节，这称为“24 bit unpacked format”，即 ARGB，其中的 A 字节被丢弃 |
| [0] | RESET | R/W | 用来复位了接的 LCD，<br>0：LCD_RESET 引脚输出低电平；<br>1：LCD_RESET 引脚输出高电平 |

- **LCDIF_TRANSFER_COUNT 寄存器**




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:16] | V_COUNT | R/W | 一帧中，有多少行有效数据 |
| [15:0] | H_COUNT | R/W | 一行中，有多少个像素 |

- **LCDIF_VDCTRL0 寄存器**

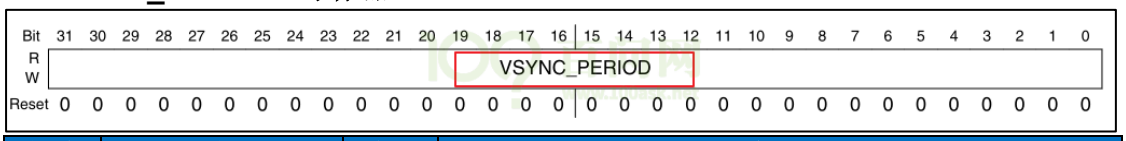



本寄存器用来设置 Vsync 信号相关的时序，及极性。

| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [29] | VSYNC_OEB | R/W | 用来控制 VSYNC 信号，对于 DOTCLK 模式，设为 0，<br>0：VSYNC 是输出引脚，用 LCD 控制器产生；<br>1：VSYNC 是输入引脚 |
| [28] | ENABLE_PRESENT | R/W | 在 DOTCLK 模式下，硬件是否会产生数据使能信号 ENALBE：<br>0：不产生；<br>1：产生 |
| [27] | VSYNC_POL | R/W | 用来决定 VSYNC 脉冲的极性，<br>0：低脉冲；<br>1：高脉冲 |
| [26] | HSYNC_POL | R/W | 用来决定 HSYNC 脉冲的极性，<br>0：低脉冲；<br>1：高脉冲 |

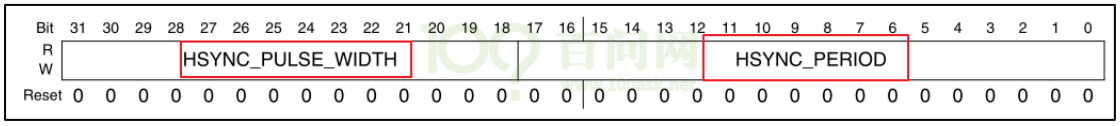
| | | | |
|---|---|---|---|
| [25] | DOTCLK_POL | R/W | 用来决定 DOTCLK 的极性，<br>0：LCD 控制器在 DOTCLK 下降沿发送数据，LCD 在上升沿捕获数据；<br>1：反过来 |
| [24] | ENABLE_POL | R/W | 用来决定 ENABLE 信号的极性，<br>0：数据有效期间，ENABLE 信号为低；<br>1：反过来 |
| [21] | VSYNC_PERIOD_UNIT | R/W | 用来决定 VSYNC_PERIOD 的单位，<br>0：单位是像素时钟(pix_clk)，这在 VSYNC 模式下使用；<br>1：单位是“整行”，这在 DOTCLK 模式下使用 |
| [20] | VSYNC_PULSE_WIDTH_UNIT | R/W | 用来决定 VSYNC_PULSE_WIDTH 的单位，<br>0：单位是像素时钟(pix_clk)；<br>1：单位是“整行” |
| [19] | HALF_LINE | R/W | VSYNC 周期是否周加上半行的时间，<br>0：VSYNC 周期=VSYNC_PERIOD；<br>1：VSYNC 周期=VSYNC_PERIOD+HORIZONTAL_PERIOD/2 |
| [18] | HALF_LINE_MODE | R/W | 0：第 1 帧将在一行的中间结束，第 2 帧在一行的中间开始；<br>1：所有帧结束前都加上半行时间，这样所有帧都会起始于“行的开头” |
| [17:0] | VSYNC_PULSE_WIDTH | R/W | VSYNC 脉冲的宽度 |

- **LCDIF_VDCTRL1 寄存器**




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [29] | VSYNC_PERIOD | R/W | 两个垂直同步信号之间的间隔，即垂直方向同步信号的总周期；<br>单位由 VSYNC_PERIOD_UNIT 决定 |

- **LCDIF_VDCTRL2 寄存器**

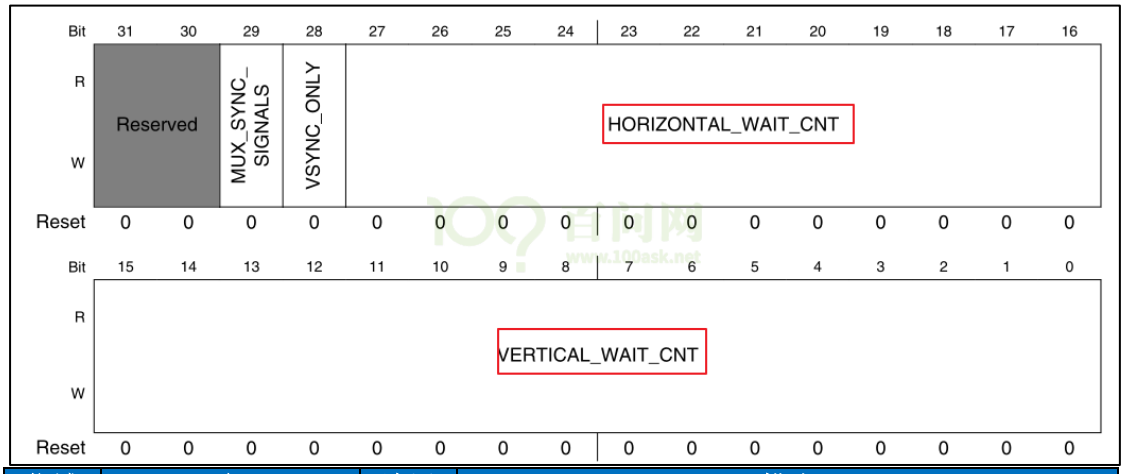



HSYNC_PULSE_WIDTH：水平同步信号脉冲宽度；

HSYNC_PERIOD：两个水平同步信号之间的总数，即水平方向同步信号的总周期

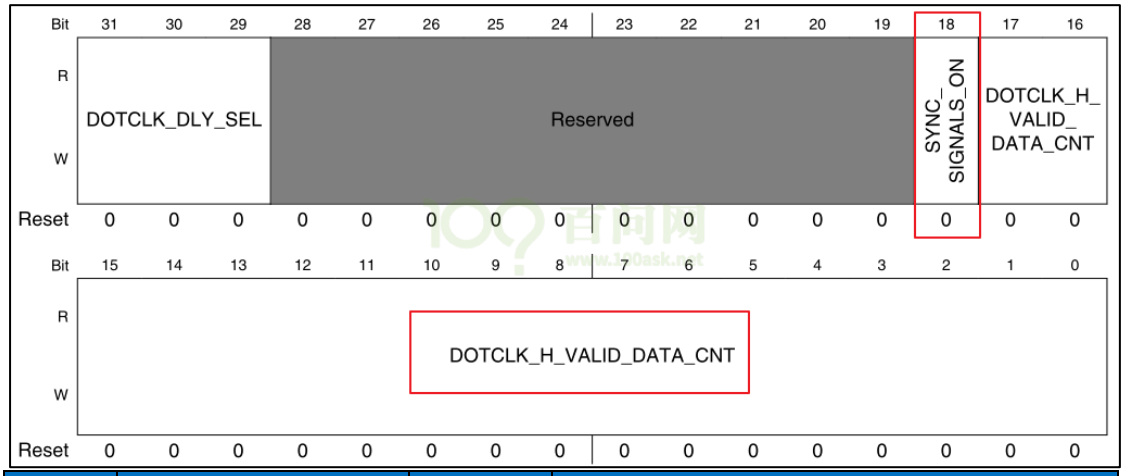
| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:18] | HSYNC_PULSE_WIDTH | R/W | HSYNC 脉冲的宽度(单位：pix_clk) |
| [17:0] | HSYNC_PERIOD | R/W | 整行的宽度，即两个 HYSNC 信号之间的宽度(单位：pix_clk) |

➢ **LCDIF_VDCTRL3 寄存器**



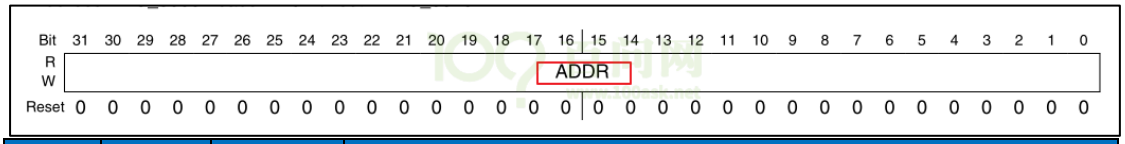

| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [29] | MUX_SYNC_SIGNALS | R/W | 用不着 |
| [28] | VSYNC_ONLY | R/W | 0：DOTCLK 模式时必须设置为 0；<br>1：VSYNC 模式时必须设置为 1 |
| [27:16] | HORIZONTAL_WAIT_CNT | R/W | 水平方向上的等待像素个数，等于 thp+thb |
| [15:0] | VERTICAL_WAIT_CNT | R/W | 垂直方向上的等待行数，等于 tvp+tvb |

➢ **LCDIF_VDCTRL4 寄存器**



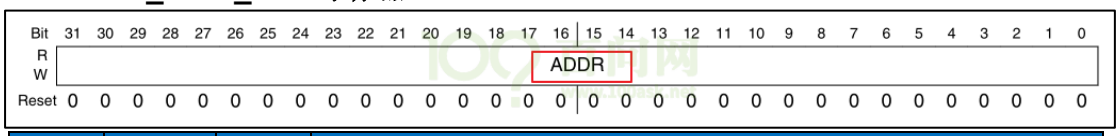

| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:29] | DOTCLK_DLY_SEL | R/W | 在 LCD 控制器内部的 DOTCLK 输出到 LCD_DOTCK 引脚时，延时多久：<br>0：2ns；<br>1：4ns；<br>2：6ns；<br>3：8ns；<br>其他值保留 |
| [18] | SYNC_SIGNALS_ON | R/W | DOTCLK 模式下必须设为 1 |
| [17:0] | DOTCLK_H_VALID_DATA_CNT | R/W | 水平方向上的有效像素个数(pix_clk)，即分辨率的 y |

➢ **LCDIF_CUR_BUF 寄存器**




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:0] | ADDR | R/W | LCD 控制器正在传输的当前帧在显存中的地址 |

➢ **LCDIF_NEXT_BUF 寄存器**




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:0] | ADDR | R/W | 下一帧在显存中的地址 |

LCD 控制器传输完当前帧后，会把 LCDIF_NEXT_BUF 寄存器的值复制到 LCDIF_CUR_BUF 寄存器。

## 13.3 编程_框架与准备

本文档虽然讲的是 IMX6ULL，但是也可以观看 S3C2440 的视频。本节文档对应的视频是 JZ2440 裸机视频中的《第 003 节_LCD 编程_框架与准备》，它是需要购买的，但是看本文档也完全没问题。

### 13.3.1 功能目的

我们最终的目的是在 LCD 显示屏上画线、画圆和写字。我们会提供一个测试函数来提供菜单，通过菜单来选择画线、画圆或写字。这些操作的核心是画点，在画点的基础上实现画线、画圆、写字。

在实现画点函数之前，要配置 LCD 控制器让它能驱动 LCD。

### 13.3.2 编程框架

一个好的框架，容易扩展。在实现画点之前想两个问题：

① 很多 LCD 的参数各不相同，它们都可以接到你用的开发板，你的程序是否容易支持不同参数的 LCD？

② 我们工作中用到很多开发板，你的程序是否容易支持不同的开发板？

为了让程序更加好扩展，我们需要掌握“面向对象编程”的概念，比如：

- 对于 LCD：

  LCD 型号虽然多，但是里面参数大家都得有，只是值不一样。

  所以可以抽像出一个 LCD 结构体，里面有时序参数、操作函数。每种 LCD 提供它自己的 LCD 结构体。

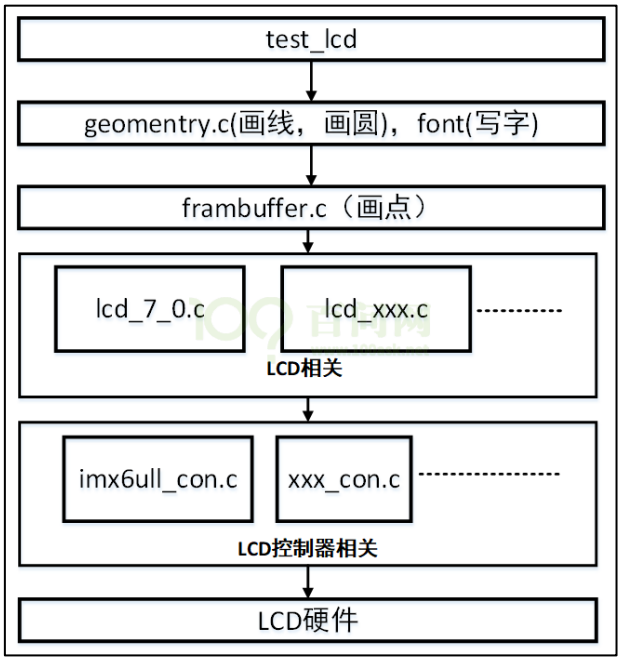
- 对于开发板，它有自己的 LCD 控制器：

  对于不同的 LCD 控制器，它们的寄存器各不相同，但是要实现的功能都是类似的：根据 LCD 的参数来设置 LCD 控制器，分配显存，启动 LCD 控制器。

  所以可以抽像出一个 LCD 控制器结构体，里面有设置参数的函数及各种操作函数。每种开发板提供它自己的 LCD 控制器结构体。

  在 C++里，这些结构体可以用类来实现；在 C 语言里，我们用结构体。

下图是本章程序的框架，尽可能的“高内聚低耦合”，即某个结构体尽量自己把一件事做好，跟其他结构体的联系尽可能少。本章程序分为 4 层：画线画圆和写字，画点，LCD 层，LCD 控制器，如下图：




对于 100ASK_IMX6ULL 开发板，通过 imx6ull_con.c 来操作 LCD 控制器。如果你有其它开发板 XXX，那么可以提供对应的 LCD 控制器程序 xxx_con.c。对于 100ASK_IMX6ULL 开发板所用的 7 寸 LCD，通过 lcd_7_0.c 来提供 LCD 的参数；如果你有其他型号的 LCD，可以提供对应的 LCD 程序 lcd_xxx.c。

## 13.4 编程_抽象出重要结构体

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/15_LCD 编程/01_simple_test”目录下：lcd_manager.h、lcd_controller_manager.h。

本文档虽然讲的是 IMX6ULL，但是也可以观看 S3C2440 的视频。本节文档对应的视频是 JZ2440 裸机视频中的《第 004 节_LCD 编程_抽象出重要结构体_P》，它是需要购买的，但是看本文档也完全没问题。

### 13.4.1 抽象出 LCD 屏幕的结构体

建立一个 lcd_manager.h，将 LCD 共有的参数（引脚的极性、时序、数据的格式 bpp、分辨率等）使用面向对象的思维方式，将这些封装成结构体 lcd_params，代码放在 lcd_manager.h 中：

```
4   enum {
5       NORMAL = 0,
6       INVERT = 1,
7   };
8
9   /* NORMAL : 正常
10  * INVERT : 取反
11  */
12  typedef struct pins_polarity {
13      int de;    /* normal: 高电平使能数据 */
```

```
14      int vclk;  /* normal: 在下降沿获取数据 */
15      int hsync; /* normal:高脉冲 */
16      int vsync; /* normal:高脉冲  */
17  }pins_polarity, *p_pins_polarity;
18
19  typedef struct time_sequence {
20      /* 垂直方向 */
21      int tvp; /*  vysnc 脉冲宽度 */
22      int tvb; /*上边黑框 , Vertical Back porch */
23      int tvf; /*下边黑框, Vertical Front porch */
24
25      /* 水平方向 */
26      int thp; /* hsync 脉冲宽度 */
27      int thb; /* 左边黑框 ,Horizontal Back porch */
28      int thf; /* 右边黑框,Horizontal Front porch */
29
30      int vclk;
31  }time_sequence, *p_time_sequence;
32
33
34  typedef struct lcd_params {
35
36      char *name;
37
38      /*引脚极性参数*/
39      pins_polarity pins_pol;
40
41      /*时序参数*/
42      time_sequence time_seq;
43
44      /*分辨率*/
45      int xres;
46      int yres;
47      int bpp;
48
49      /*显存*/
50      unsigned int fb_base;
51
52  }lcd_params, *p_lcd_params;
```

以后就可以使用 lcd_params 结构体来表示一个 LCD。

我们还会提供一个 lcd_manager.c，用来管理 lcd_params 结构体。

以后要使用某款 LCD 时，按如下步骤：

① 构造一个 lcd_params 结构体来表示这款 LCD，填充参数；

② 然后通过 register_lcd 函数注册这个结构体，它会存入一个结构体数组中；

③ 数组中可能有多款 LCD，通过 select_lcd 选择你要用的 LCD。

## 13.4.2 抽象出 LCD 控制器的结构体

将 LCD 控制器共有的函数（初始化函数，使能函数等）抽象出来，封装成 lcd_controller 结构体，代码在 lcd_controller_manager.h 中：

```
7   typedef struct lcd_controller{
8       char* name;
9       void (*init)(p_lcd_params plcdparams);
10      void(*enable)(void);
```

```
11      void(*disable)(void);
12  }lcd_controller, *p_lcd_controller;
```

以后就可以使用 lcd_controller 结构体来表示一个 LCD 控制器。

我们还会提供一个 lcd_controller_manager.c，用来管理 lcd_controller 结构体。

要使用某款开发板的 LCD 控制器时，按如下步骤：

① 构告一个 lcd_controller 结构体，填充这个 LCD 控制器的各类操作函数；

② 然后通过 register_lcd_controller 函数注册这个控制器，它会存入一个结构体数组中；

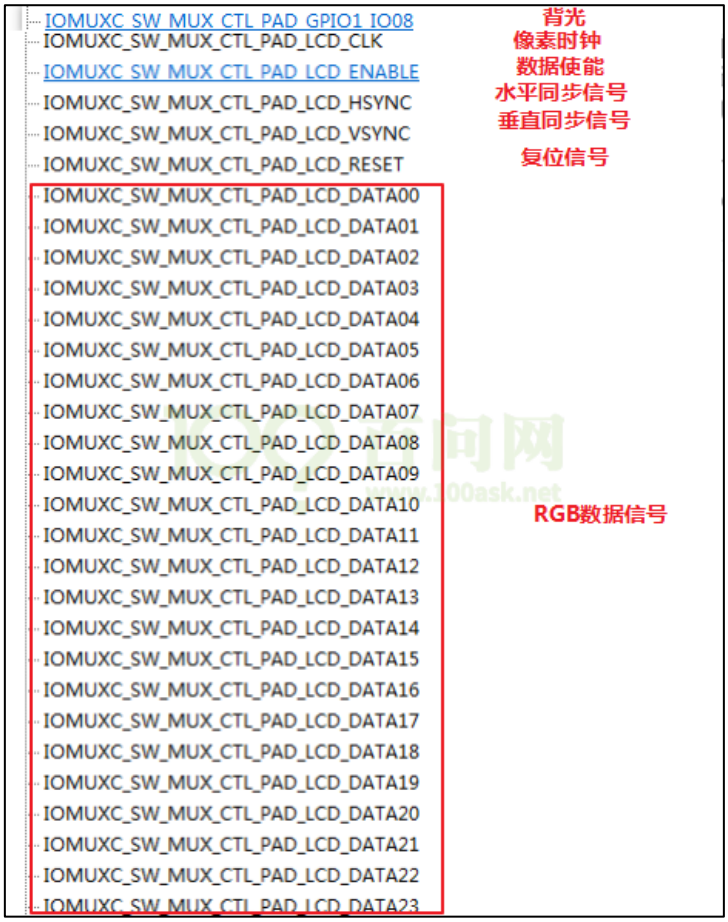
③ 数组中有可能有多款 LCD 控制器，通过 select_lcd_controller 函数选择你要的 LCD 控制器。

## 13.5 编程_LCD 控制器

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/15_LCD 编程/01_simple_test”目录下：imx6ull_con.c。

本文档虽然讲的是 IMX6ULL，但是也可以观看 S3C2440 的视频。本节文档对应的视频是 JZ2440 裸机视频中的《第 005 节_LCD 编程_LCD 控制器_P》，它是需要购买的，但是看本文档也完全没问题。

### 13.5.1 LCD 控制器相关引脚复用配置




根据前面硬件接口章节 15.1.3 和上图，我们知道要设置这 30 个引脚。

设置引脚按两步走：
① 设置引脚复用功能：让它们用于 LCD；
② 设置引脚的硬件属性：比如上下拉电阻什么的，对于 LCD 引脚基本不用设置。

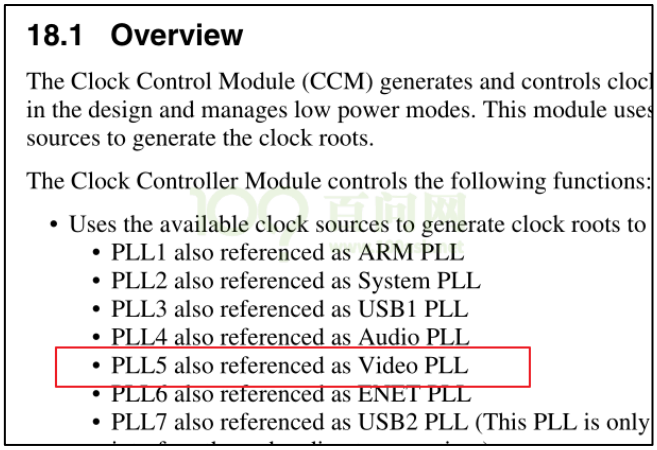
## 13.5.2 LCD 控制器像素时钟配置

在 LCD 手册中，可以看到这样的图：

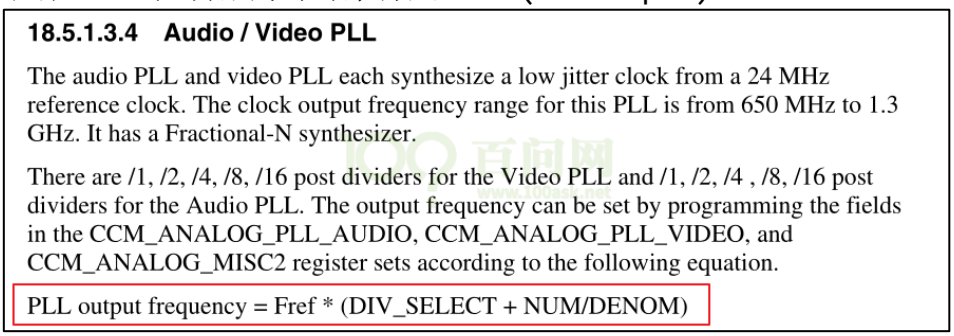
| ITEM | SYMBOL | MIN. | TYP. | MAX. | UNIT | Note |
|---|---|---|---|---|---|---|
| Dot Clock | 1/tCLK | 45 | 51.2 | 57 | MHz | |

所以需要设置像素时钟为 51.2Mhz，本节内容参考 IMX6ULL 芯片手册的《Chapter 18 Clock Controller Module (CCM)》。

➢ 确定 PLL



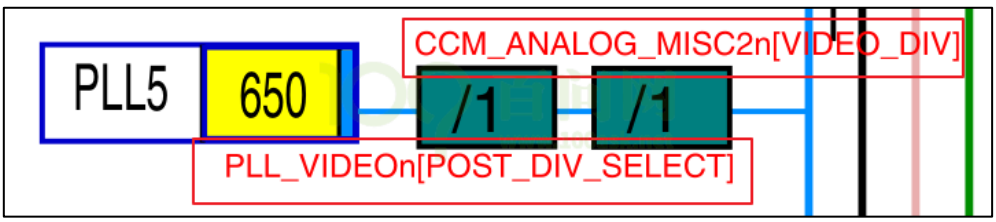

由上图可知 LCD 控制器的时钟来源是 PLL5(video pll)。




根据上图可得知 VIDEO PLL 的频率计算公式，为了方便计算，我们不需要后面的小数运算，即：

```
Video PLL output frequency(PLL5)= Fref * (DIV_SELECT + 0)。
```

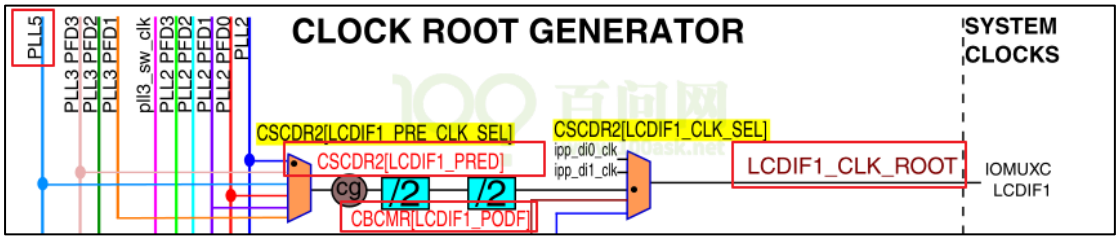
➢ 确定 PLL 后的分频系数




根据上图可知 PLL5 出来后经过两级分频，即：

```
PLL5_MAIN_CLK = PLL5 / POST_DIV_SELECT / VIDEO_DIV
```

➢ PLL 分频后进入 LCDIF 控制器前的分频系数




根据上图可知 PLL5 分频后，到达 LCDIF 控制器前也有两级分频：

```
LCDIF1_CLK_ROOT = PLL5_MAIN_CLK /LCDIF1_PRED / LCDIF1_PODF,
```

根据上面三个内容，我们可以采用以下这个配置来达到像素时钟 51.2Mhz：

```
DIV_SELECT = 32, NUM = 0, DENOM = 0;
POST_DIV_SELECT = 1, VIDEO_DIV = 1;
LCDIF1_PRED = 3, LCDIF1_PODF = 5
```

代入时钟公式得 24*(32+0)/1/1/3/5 ≈ 51.2Mhz。

下节开始编程。

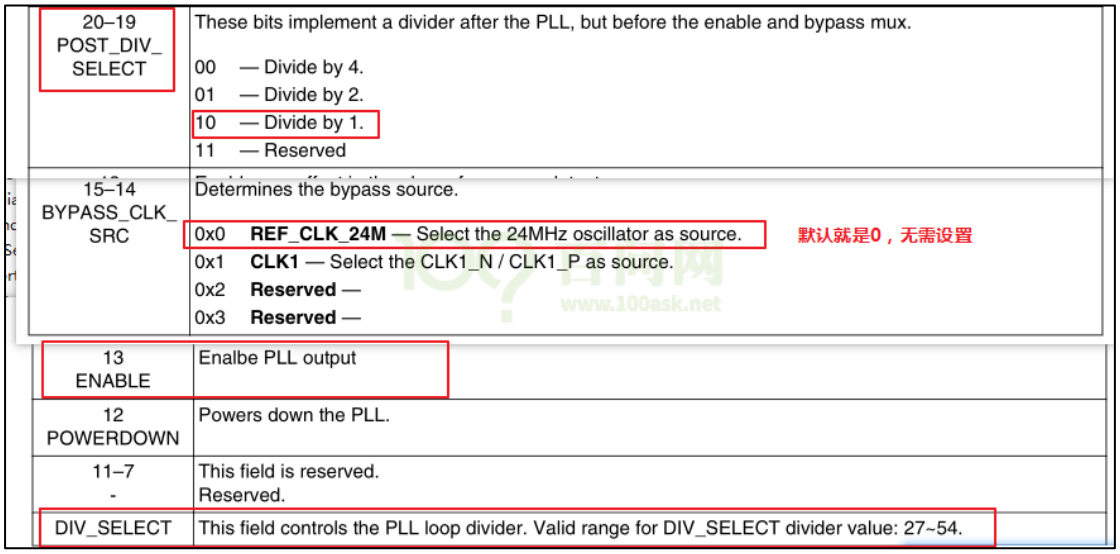
## 13.5.3 LCD 控制器时钟编程

➢ 取消小数分配器

```
79    CCM_ANALOG->PLL_VIDEO_NUM   = 0;
80    CCM_ANALOG->PLL_VIDEO_DENOM = 0;
```

清零表示取消小数分配器。

➢ 设置 CCM_ANALOG_PLL_VIDEOn 寄存器

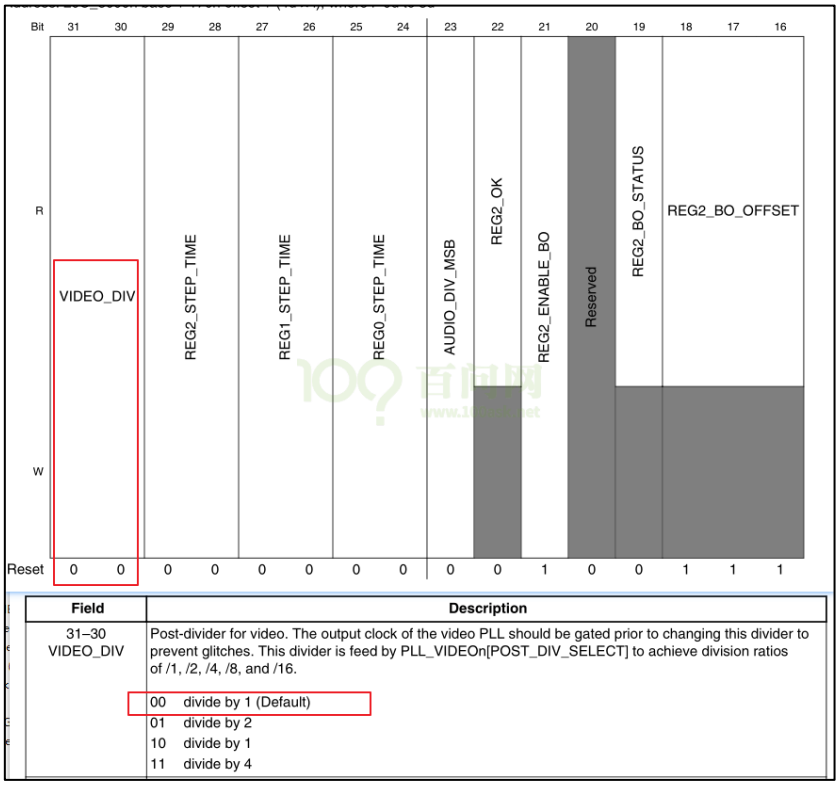


| | |
|---|---|
| 20–19 POST_DIV_SELECT | These bits implement a divider after the PLL, but before the enable and bypass mux.<br>00 — Divide by 4.<br>01 — Divide by 2.<br>10 — Divide by 1.<br>11 — Reserved |
| 15–14 BYPASS_CLK_SRC | Determines the bypass source.<br>0x0 **REF_CLK_24M** — Select the 24MHz oscillator as source. 默认就是0，无需设置<br>0x1 **CLK1** — Select the CLK1_N / CLK1_P as source.<br>0x2 **Reserved** —<br>0x3 **Reserved** — |
| 13 ENABLE | Enalbe PLL output |
| 12 POWERDOWN | Powers down the PLL. |
| 11–7 - | This field is reserved.<br>Reserved. |
| DIV_SELECT | This field controls the PLL loop divider. Valid range for DIV_SELECT divider value: 27~54. |

```
93   CCM_ANALOG->PLL_VIDEO =  (2 << 19) | (1 << 13) | (32<< 0);
```

设置 PLL5 使能，倍频为 32 倍，1 分频（即不分频），选择外部 24M 晶振为时钟源，至此 PLL5 分频后为 24 * 32 / 1 。

- **设置 CCM_ANALOG_MISC2n**



| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R | VIDEO_DIV | | REG2_STEP_TIME | | REG1_STEP_TIME | | REG0_STEP_TIME | | AUDIO_DIV_MSB | REG2_OK | REG2_ENABLE_BO | Reserved | REG2_BO_STATUS | REG2_BO_OFFSET | | |
| W | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |

| Field | Description |
|---|---|
| 31–30 VIDEO_DIV | Post-divider for video. The output clock of the video PLL should be gated prior to changing this divider to prevent glitches. This divider is feed by PLL_VIDEOn[POST_DIV_SELECT] to achieve division ratios of /1, /2, /4, /8, and /16.<br>00 divide by 1 (Default)<br>01 divide by 2<br>10 divide by 1<br>11 divide by 4 |

默认就为 1 分频，所以无需设置，至此 PLL5 分频后为 24 * 32 / 1 / 1 = 768Mhz。

- **设置 CCM_CSCDR2**

| | |
|---|---|
| 17–15 LCDIF1_PRE_CLK_SEL | Selector for lcdif1 root clock pre-multiplexer<br>000 derive clock from PLL2<br>001 derive clock from PLL3 PFD3<br>010 derive clock from PLL5<br>011 derive clock from PLL2 PFD0<br>100 derive clock from PLL2 PFD1<br>101 derive clock from PLL3 PFD1<br>110-111 Reserved |
| 14–12 LCDIF1_PRED | Pre-divider for lcdif1 clock.<br>**NOTE:** Divider should be updated when output clock is gated.<br>000 divide by 1<br>001 divide by 2<br>010 divide by 3<br>011 divide by 4<br>100 divide by 5<br>101 divide by 6<br>110 divide by 7<br>111 divide by 8 |
| 11–9 LCDIF1_CLK_SEL | Selector for LCDIF1 root clock multiplexer<br>000 derive clock from divided pre-muxed LCDIF1 clock<br>001 derive clock from ipp_di0_clk<br>010 derive clock from ipp_di1_clk<br>011 derive clock from ldb_di0_clk<br>100 derive clock from ldb_di1_clk<br>101-111 Reserved |

设置选中对应的时钟源，预分频系数为 3，得 768 / 3 = 256Mhz。代码如下：

```
    CCM->CSCDR2 &= ~(7 << 15);
     CCM->CSCDR2 |=  (2 << 15);

     CCM->CSCDR2 &= ~(7 << 12);
     CCM->CSCDR2 |=  (2 << 12);

     CCM->CSCDR2 &= ~(7 << 9);
```

- 设置 CCM_CBCMR

| | |
|---|---|
| 25–23 LCDIF1_PODF | Post-divider for LCDIF1 clock.<br>000 divide by 1<br>001 divide by 2<br>010 divide by 3<br>011 divide by 4<br>100 divide by 5 |

```
        CCM->CBCMR &= ~(7 << 23);
        CCM->CBCMR |=    4 << 23; /*[25:23] :4 : 表示 5 分频*/
```

设置预分频系数为 5，最终得到 51.2Mhz。

- 重新同步时钟

```
/* 重新设置时钟后，需要软复位 LCD 控制器，让 LCD 控制器像素时钟同步*/
     LCDIF->CTRL  = 1<<31;

  /*软复位需要花费好几个时钟周期，这里需要一些时间等待*/
     delay(100);

  /*同步像素时钟结束*/
    LCDIF->CTRL  = 0<<31; /* 取消复位 */
```

## 13.5.4 LCD 控制器像素格式配置

设置 LCDIF_CTRLn 寄存器

| | |
|---|---|
| 19 BYPASS_COUNT | When this bit is 0, it means that eLCDIF will stop the block operation and turn off the RUN bit after the amount of data indicated by the LCDIF_TRANSFER_COUNT register has been transferred out. When this bit is set to 1, the block will continue normal operation indefinitely until it is told to stop. This bit must be 0 in MPU and VSYNC modes, and must be 1 in DOTCLK and DVI modes of operation. |
| 18 VSYNC_MODE | Setting this bit to 1 will make the eLCDIF hardware go into VSYNC mode. WAIT_FOR_VSYNC_EDGE can be used only if this bit is set. If VSYNC signal is required to be an output from the block, SYNC_SIGNALS_ON bit in LCDIF_VDCTRL4 register must be set. DOCLK模式必须设置为1 |
| 17 DOTCLK_MODE | Set this bit to 1 to make the hardware go into the DOTCLK mode, i.e. VSYNC/HSYNC/DOTCLK/ENABLE interface mode. ENABLE is optional, selected by the ENABLE_PRESENT bit. Toggle this bit from 1 to 0 to make the hardware go out of DOTCLK mode after completing all data transfer and deasserting the RUN bit. |
| 16 DATA_SELECT | Command Mode polarity bit. This bit should only be changed when RUN is 0.<br>0x0 **CMD_MODE** — Command Mode. LCD_RS signal is Low.<br>0x1 **DATA_MODE** — Data Mode. LCD_RS signal is High. |
| 15–14 INPUT_DATA_SWIZZLE | This field specifies how to swap the bytes fetched by the bus master interface. The swizzle function is independent of the WORD_LENGTH bit. The supported swizzle configurations are:<br>0x0 **NO_SWAP** — No byte swapping.(Little endian) 数据不交换<br>0x0 **LITTLE_ENDIAN** — Little Endian byte ordering (same as NO_SWAP).<br>0x1 **BIG_ENDIAN_SWAP** — Big Endian swap (swap bytes 0,3 and 1,2).<br>0x1 **SWAP_ALL_BYTES** — Swizzle all bytes, swap bytes 0,3 and 1,2 (aka Big Endian).<br>0x2 **HWD_SWAP** — Swap half-words.<br>0x3 **HWD_BYTE_SWAP** — Swap bytes within each half-word. |
| 13–12 CSC_DATA_SWIZZLE | This field specifies how to swap the bytes after the data has been converted into an internal representation of 24 bits per pixel and before it is transmitted over the LCD interface bus. The data is always transmitted with the least significant byte/hword (half word) first after the swizzle takes place. So, INPUT_DATA_SWIZZLE takes place first on the incoming data, and then CSC_DATA_SWIZZLE is applied. The swizzle function is independent of the WORD_LENGTH or the LCD_DATABUS_WIDTH fields. If RGB_TO_YCRCB422_CSC bit is set, the swizzle occurs on the Y, Cb, Cr values. The supported swizzle configurations are:<br>CSC数据不交换<br>0x0 **NO_SWAP** — No byte swapping.(Little endian) |

| | |
|---|---|
| 11–10 LCD_DATABUS_WIDTH | LCD Data bus transfer width.<br>0x0 **16_BIT** — 16-bit data bus mode.<br>0x1 **8_BIT** — 8-bit data bus mode.<br>0x2 **18_BIT** — 18-bit data bus mode.<br>0x3 **24_BIT** — 24-bit data bus mode.<br>硬件RGB带宽24位 |
| 9–8 WORD_LENGTH | Input data format.<br>0x0 **16_BIT** — Input data is 16 bits per pixel.<br>0x1 **8_BIT** — Input data is 8 bits wide.<br>0x2 **18_BIT** — Input data is 18 bits per pixel.<br>0x3 **24_BIT** — Input data is 24 bits per pixel.<br>程序支持16位和24位的像素<br>根据LCD传上来的bpp确定 |
| 5 MASTER | Set this bit to make the eLCDIF act as a bus master.<br>设置为主机模式 |
| 3 DATA_FORMAT_16_BIT | When this bit is 1 and WORD_LENGTH = 0, it implies that the 16-bit data is in ARGB555 format. When this bit is 0 and WORD_LENGTH = 0, it implies that the 16-bit data is in RGB565 format. When WORD_LENGTH is not 0, this bit does not care. 当LCD设置为16bit时需要设置该位，ARGB555 |
| 1 DATA_FORMAT_24_BIT | Used only when WORD_LENGTH = 3, i.e. 24-bit. Note that this applies to both packed and unpacked 24-bit data.<br>24位数据均有效<br>0x0 **ALL_24_BITS_VALID** — Data input to the block is in 24 bpp format, such that all RGB 888 data is contained in 24 bits.<br>0x1 **DROP_UPPER_2_BITS_PER_BYTE** — Data input to the block is actually RGB 18 bpp, but there is 1 color per byte, hence the upper 2 bits in each byte do not contain any useful data, and should be dropped. |

代码如下：

```
LCDIF->CTRL |= (1 << 19) | (1 << 17) |(3 << 10) | (bpp_mode << 8) | (1 << 5) ;

        /* [3]当 bpp 为 16 时，数据格式为 ARGB555*/
        if(plcdparams->bpp == 16)
        {
            LCDIF->CTRL |= 1<<3;
        }
```

- **设置 LCDIF_CTRL1n 寄存器**

| | |
|---|---|
| 19–16 BYTE_PACKING_FORMAT | This bitfield is used to show which data bytes in a 32-bit word are valid. Default value 0xf indicates that all bytes are valid. For 8-bit transfers, any combination in this bitfield will mean valid data is present in the corresponding bytes. In the 16-bit mode, a 16-bit half-word is valid only if adjacent bits [1:0] or [3:2] or both are 1. A value of 0x0 will mean that none of the bytes are valid and should not be used. For example, set the bit field value to 0x7 if the display data is arranged in the 24-bit unpacked format (A-R-G-B where A value does not have be transmitted). When input data is in YCbCr 4:2:2 format (YCBCR422_INPUT is 1), H_COUNT should be the number of pixels that should be fetched by the block and the BYTE_PACKING_FORMAT should be 0xF.(Note - YCBCR422_INPUT = 1 implies 2 pixels per 32 bits). |

```
if(plcdparams->bpp == 24 || plcdparams->bpp == 32)
    {
            LCDIF->CTRL1 &= ~(0xf << 16);
            LCDIF->CTRL1 |=  (0x7 << 16);
    }
```

表示 ARGB 传输格式模式下，传输 24 位无压缩数据，A 通道不用传输），当我们选用 16bpp 即 ARGB555 时，不需要设置此位。

## 13.5.5 LCD 控制器时序配置及极性配置

➢ 设置 LCDIF_TRANSFER_COUNT 寄存器

| Field | Description |
|---|---|
| 31–16 V_COUNT | Number of horizontal lines per frame which contain valid data. In DOTCLK mode, V_COUNT should be the same as the number of active horizontal lines in a progressive frame. In DVI mode, V_COUNT should be the number of active horizontal lines per frame, and not per field. 垂直方向上的像素个数 |
| H_COUNT | Total valid data (pixels) in each horizontal line. The data size is given by the WORD_LENGTH. When input data is in YCbCr 4:2:2 format (YCBCR422_INPUT is 1), H_COUNT should be the number of 32-bit words that should be fetched by the block and the BYTE_PACKING_FORMAT should be 0xF. In 24-bit packed format (WORD_LENGTH=0x3, BYTE_PACKING_FORMAT=0xF), the H_COUNT must be a multiple of 4 pixels. In 16-bit packed format (WORD_LENGTH=0x0, BYTE_PACKING_FORMAT=0xF), the H_COUNT must be a multiple of 2 pixels. 水平方向上的像素个数 |

```
LCDIF->TRANSFER_COUNT  = (plcdparams->yres << 16) | (plcdparams->xres << 0);
```

➢ 设置 LCDIF_VDCTRL0n 寄存器

| | |
|---|---|
| 29 VSYNC_OEB | 0 means the VSYNC signal is an output, 1 means it is an input. Should be set to 0 in the DOTCLK mode.<br>0x0 **VSYNC_OUTPUT** — The VSYNC pin is in the output mode and the VSYNC signal has to be generated by the eLCDIF block. 垂直同步信号为输出信号<br>0x1 **VSYNC_INPUT** — The VSYNC pin is in the input mode and the LCD controller sends the VSYNC signal to the block. |
| 28 ENABLE_PRESENT | Setting this bit to 1 will make the hardware generate the ENABLE signal in the DOTCLK mode, thereby making it the true RGB interface along with the remaining three signals VSYNC, HSYNC and DOTCLK. DOCLK模式下，硬件自动产生使能信号 |
| 27 VSYNC_POL | Default 0 active low during VSYNC_PULSE_WIDTH time and will be high during the rest of the VSYNC period. Set it to 1 to invert the polarity. |
| 26 HSYNC_POL | Default 0 active low during HSYNC_PULSE_WIDTH time and will be high during the rest of the HSYNC period. Set it to 1 to invert the polarity. 根据LCD资源文件决定 |
| 25 DOTCLK_POL | Default is data launched at negative edge of DOTCLK and captured at positive edge. Set it to 1 to invert the polarity. Set it to 0 in DVI mode. |
| 24 ENABLE_POL | Default 0 active low during valid data transfer on each horizontal line. |
| 23–22 RSRVD1 | This field is reserved.<br>Reserved bits. Write as 0. |
| 21 VSYNC_PERIOD_UNIT | Default 0 for counting VSYNC_PERIOD in terms of DISPLAY CLOCK (pix_clk) cycles. Set it to 1 to count in terms of complete horizontal lines. DISPLAY CLOCK (pix_clk) cycles should be used in the VSYNC mode, while horizontal line should be used in the DOTCLK mode. DOTCLK模式需设置为1 |
| 20 VSYNC_PULSE_WIDTH_UNIT | Default 0 for counting VSYNC_PULSE_WIDTH in terms of DISPLAY CLOCK (pix_clk) cycles. Set it to 1 to count in terms of complete horizontal lines. DOTCLK模式需设置为1 |
| 19 HALF_LINE | Setting this bit to 1 will make the total VSYNC period equal to the VSYNC_PERIOD field plus half the HORIZONTAL_PERIOD field (i.e. VSYNC_PERIOD field plus half horizontal line), otherwise it is just VSYNC_PERIOD. Should be only used in the DOTCLK mode, not in the VSYNC interface mode. |
| 18 HALF_LINE_MODE | When this bit is 0, the first field (VSYNC period) will end in half a horizontal line and the second field will begin with half a horizontal line. When this bit is 1, all fields will end with half a horizontal line, and none will begin with half a horizontal line. 垂直同步信号的宽度 |
| VSYNC_PULSE_WIDTH | Number of units for which VSYNC signal is active. For the DOTCLK mode, the unit is determined by the VSYNC_PULSE_WIDTH_UNIT. If the VSYNC_PULSE_WIDTH_UNIT is 0 for DOTCLK mode, VSYNC_PULSE_WIDTH must be less than HSYNC_PERIOD. For the VSYNC interface mode, it should |

```
LCDIF->VDCTRL0 = (1 << 28)|( plcdparams->pins_pol.vsync << 27)
                          |( plcdparams->pins_pol.hsync << 26)
                          |( plcdparams->pins_pol.vclk << 25)
                          |(plcdparams->pins_pol.de << 24)
                          |(1 << 21)|(1 << 20)|( plcdparams->time_seq.tvp << 0);
```

➢ 配置 LCDIF_VDCTRL1 寄存器

LCDIF_VDCTRL1 field descriptions

| Field | Description |
|---|---|
| VSYNC_PERIOD | Total number of units between two positive or two negative edges of the VSYNC signal. If HALF_LINE is set, it is implicitly calculated to be VSYNC_PERIOD plus half HSYNC_PERIOD. |

```
LCDIF->VDCTRL1 = plcdparams->time_seq.tvb + plcdparams->time_seq.tvp +
plcdparams->yres + plcdparams->time_seq.tvf;
```

设置垂直方向的总周期 = 上黑框 tvb+垂直同步脉冲 tvp+垂直有效高度 yres+下黑框 tvf。

- **配置 LCDIF_VDCTRL2 寄存器**

LCDIF_VDCTRL2 field descriptions

| Field | Description |
|---|---|
| 31–18 HSYNC_PULSE_WIDTH | Number of DISPLAY CLOCK (pix_clk) cycles for which HSYNC signal is active. |
| HSYNC_PERIOD | Total number of DISPLAY CLOCK (pix_clk) cycles between two positive or two negative edges of the HSYNC signal. |

```
LCDIF->VDCTRL2 = (plcdparams->time_seq.thp << 18) | (plcdparams->time_seq.thb + plcdparams->time_seq.thp + plcdparams->xres + plcdparams->time_seq.thf);
```

LCDIF_VDCTRL2 寄存器中：

[18:31]：表示水平同步信号脉冲宽度；
[17: 0]：表示水平方向总周期。

水平方向的总周期 = 左黑框 thb+水平同步脉冲 thp+水平有效高度 xres+右黑框 thf。

- **配置 LCDIF_VDCTRL3 寄存器**

LCDIF_VDCTRL3 field descriptions

| Field | Description |
|---|---|
| 31–30 RSRVD0 | This field is reserved.<br>Reserved bits, write as 0. |
| 29 MUX_SYNC_SIGNALS | When this bit is set, the eLCDIF block will internally mux HSYNC with LCD_D14, DOTCLK with LCD_D13 and ENABLE with LCD_D12, otherwise these signals will go out on separate pins. This feature can be used to maintain backward compatible with 37xx. |
| 28 VSYNC_ONLY | This bit must be set to 1 in the VSYNC mode of operation, and 0 in the DOTCLK mode of operation. |
| 27–16 HORIZONTAL_WAIT_CNT | In the DOTCLK mode, wait for this number of clocks from falling edge (or rising if HSYNC_POL is 1) of HSYNC signal to account for horizontal back porch plus the number of DOTCLKs before the moving picture information begins. |
| VERTICAL_WAIT_CNT | In the VSYNC interface mode, wait for this number of DISPLAY CLOCK (pix_clk) cycles from the falling VSYNC edge (or rising if VSYNC_POL is 1) before starting LCD transactions and is applicable only if WAIT_FOR_VSYNC_EDGE is set. Minimum is CMD_SETUP+5. In the DOTCLK mode, it accounts for the vertical back porch lines plus the number of horizontal lines before the moving picture begins. The unit for this parameter is inherently the same as the VSYNC_PERIOD_UNIT. |

```
LCDIF->VDCTRL3 = ((plcdparams->time_seq.thb + plcdparams->time_seq.thp) << 16) | (plcdparams->time_seq.tvb + plcdparams->time_seq.tvp);
```

设置 ELCDIF 的 VDCTRL3 寄存器：

[27:16] ：水平方向上的等待时钟数 = thb + thp；
[15:0] ：垂直方向上的等待时钟数 = tvb + tvp。

- **设置 LCDIF_VDCTRL4 寄存器**

LCDIF_VDCTRL4 field descriptions

| Field | Description |
|---|---|
| 31–29 DOTCLK_DLY_SEL | This bitfield selects the amount of time by which the DOTCLK signal should be delayed before coming out of the LCD_DOTCK pin. 0 = 2ns; 1=4ns;2=6ns;3=8ns. Remaining values are reserved. |
| 28–19 RSRVD0 | This field is reserved.<br>Reserved bits, write as 0. |
| 18 SYNC_SIGNALS_ON | Set this field to 1 if the LCD controller requires that the VSYNC or VSYNC/HSYNC/DOTCLK control signals should be active at least one frame before the data transfers actually start and remain active at least one frame after the data transfers end. The hardware does not count the number of frames automatically. Rather, the VSYNC edge interrupt can be monitored by software to count the number of frames that have occurred after this bit is set and then the RUN bit can be set to start the data transactions. This bit must always be set in the DOTCLK mode of operation, and it must be set in the VSYNC mode of operation when VSYNC signal is an output. |
| DOTCLK_H_VALID_DATA_CNT | Total number of DISPLAY CLOCK (pix_clk) cycles on each horizontal line that carry valid data in DOTCLK mode. |

```
LCDIF->VDCTRL4 = (1<<18) | (plcdparams->xres);
```

设置 ELCDIF 的 VDCTRL4 寄存器：

[18] ：使用 VSHYNC、HSYNC、DOTCLK 模式此为置 1；
[17:0]：水平方向的宽度。

## 13.5.6 设置显存

**LCDIF_CUR_BUF field descriptions**

| Field | Description |
|---|---|
| ADDR | Address of the current frame being transmitted by eLCDIF. |

**LCDIF_NEXT_BUF field descriptions**

| Field | Description |
|---|---|
| ADDR | Address of the next frame that will be transmitted by eLCDIF. |

```
LCDIF->CUR_BUF  =  plcdparams->fb_base;
LCDIF->NEXT_BUF =  plcdparams->fb_base;
```

- CUR_BUF ： 当前显存地址；
- NEXT_BUF ： 下一帧显存地址；

方便运算，都设置为同一个显存地址。当 LCD 控制器发送完当前帧后，它会把 LCDIF_NEXT_BUF 寄存器的值复制进 LCDIF_CUR_BUF 中。

# 13.6 编程_LCD 设置

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/15_LCD 编程/01_simple_test”目录下：lcd_7_0.c。

本文档虽然讲的是 IMX6ULL，但是也可以观看 S3C2440 的视频。本节文档对应的视频是 JZ2440 裸机视频中的《第 006 节_LCD 编程_LCD 设置_P》，它是需要购买的，但是看本文档也完全没问题。

## 13.6.1 添加 LCD 屏幕名称

```
.name = "lcd_7.0",
```

先给我们本次实验的 LCD 屏幕参数一个名称，可以根据名称去选择我们需要的 LCD 屏幕参数。

## 13.6.2 极性设置

LCDIF_VDCTRL0n 寄存器极性设置位

| | |
|---|---|
| 27 VSYNC_POL | Default 0 active low during VSYNC_PULSE_WIDTH time and will be high during the rest of the VSYNC period. Set it to 1 to invert the polarity. |
| 26 HSYNC_POL | Default 0 active low during HSYNC_PULSE_WIDTH time and will be high during the rest of the HSYNC period. Set it to 1 to invert the polarity. |
| 25 DOTCLK_POL | Default is data launched at negative edge of DOTCLK and captured at positive edge. Set it to 1 to invert the polarity. Set it to 0 in DVI mode. |
| 24 ENABLE_POL | Default 0 active low during valid data transfer on each horizontal line. |

- VSYNC_POL 与 HSYNC_POL：0 表示低电平有效，1 表示高电平有效
- DOTCLK_POL：0 表示上升沿有效，1 表示下降沿有效

- ENABLE_POL：0 表示低电平有效，1 表示高电平有效

接着根据 15－1.4 关键特性章节中，我们知道所用的 LCD 参数为：

a） 像素时钟 DCLK 是下降沿时获取数据；

b） 水平同步信号 HSD 和垂直同步信号 VSD 都是低电平有效；

c） 数据使能信号 DEN 是高电平表示数据有效

所以，我们只要在 lcd_7_0.c 文件中设定对应的极性即可：
① 数据使能信号 de 设置为 1；
② 像素时钟 vclk 极性设置为 1；
③ 水平同步信号 HSD 和垂直同步信号 VSD 都设置为 0。
代码如下：

```
enum {
    NORMAL = 0,
    INVERT = 1,
};
```

根据枚举内容，填入对应的参数即可

```
.pins_pol = {
    .de    = INVERT, /* normal: 低电平表示使能输出 */
    .vclk  = INVERT, /* normal: 在上升降沿获取数据*/
    .hsync = NORMAL,    /* normal: 低脉冲*/
    .vsync = NORMAL, /* normal: 低脉冲*/
},
```

## 13.6.3 时序设置

下图来自 LCD 的芯片手册，从中可以知道它的时序：

| | ITEM | SYMBOL | MIN. | TYP. | MAX. | UNIT | Note |
|---|---|---|---|---|---|---|---|
| SYNC MODE | Horizontal Total Time | TH | 1324 | 1344 | 1364 | tCLK | |
| | Horizontal Pulse Width | Thpw | | 20 | - | tCLK | thb + thpw =160DCLK is fixed |
| | Horizontal Back Porch | Thb | | 140 | - | tCLK | |
| | Horizontal Front Porch | Thfp | 140 | 160 | 180 | tCLK | |
| | Horizontal Effective Time | THA | | 1024 | | tCLK | |
| | Vertical Total Time | TV | 625 | 635 | 645 | tH | |
| | Vertical Pulse Width | Tvpw | | 3 | - | th | tvpw + tvb =23th is fixed |
| | Vertical Back Porch | Tvb | - | 20 | - | th | |
| | Vertical Front Porch | Tvfp | 2 | 12 | 22 | th | |
| | Vertical Valid | Tvd | | 600 | | th | |

观察上图中红色框框内容，在 lcd_7_0.c 中定义的 lcd_7_0_params 结构体中填入相应的值：
① Thpw：结构体中的 tvb = 20；
② Thb：结构体中的 thb = 140；
③ Thfp：结构体中的 thf = 160；
④ THA：结构体中的 tvp = 1024；
⑤ Tvpw：结构体中的 tvp = 3；
⑥ Tvb：结构体中的 thp = 20；
⑦ Tvfp：结构体中的 tvf = 12；
⑧ Tvd：结构体中的 yres = 600。
代码如下：

```
    .time_seq = {
    /* 垂直方向 */
```

```
    .tvp=    3, /* vysnc 脉冲宽度 */
    .tvb=    20,  /* 上边黑框, Vertical Back porch */
    .tvf=    12,  /* 下边黑框, Vertical Front porch */

    /* 水平方向 */
    .thp=    20, /* hsync 脉冲宽度 */
    .thb=    140,  /* 左边黑框, Horizontal Back porch */
    .thf=    160,  /* 右边黑框, Horizontal Front porch */

    .vclk=   51.2,  /* MHz */
},
.xres = 1024,
.yres = 600
```

## 13.6.4 显存地址设置

### 2.2 ARM Platform Memory Map

The chip memory map has been provided in the following tables.

**Table 2-1. System memory map**

| Start address | End address | Size | Description |
|---|---|---|---|
| 8000_0000 | FFFF_FFFF | 2048 MB | MMDC—x16 DDR Controller. |
| 7000_0000 | 7FFF_FFFF | 256 MB | Reserved |
| 6000_0000 | 6FFF_FFFF | 256 MB | QSPI1 Memory |
| 5800_0000 | 5FFF_FFFF | 128 MB | EIM Aliased |
| 5000_0000 | 57FF_FFFF | 128 MB | EIM (NOR/SRAM) |
| 1000_0000 | 4FFF_FFFF | 1024 MB | Reserved |
| 0E00_0000 | 0FFF_FFFF | 32 MB | Reserved |
| 0C00_0000 | 0DFF_FFFF | 32 MB | QSPI1 Rx Buffer |
| 0900_0000 | 0BFF_FFFF | 48 MB | Reserved |
| 0800_0000 | 08FF_FFFF | 16 MB | Reserved |
| 02C0_0000 | 07FF_FFFF | 84 MB | Reserved |
| 0230_0000 | 02BF_FFFF | 9 MB | Reserved |
| 0220_0000 | 022F_FFFF | 1 MB | Table 2-4 AIPS-3. See IP listing on the separate map. |
| 0210_0000 | 021F_FFFF | 1 MB | Table 2-3 AIPS-2. See the IP listing on the separate map. |
| 0200_0000 | 020F_FFFF | 1 MB | Table 2-2 AIPS-1. See the IP listing on the separate map. |
| 0181_0000 | 01FF_FFFF | 8128 KB | Reserved |
| 0180_C000 | 0180_FFFF | 16 KB | Reserved |

根据上图 imx_6ull 可以挂接最大 2GB 的内存，可实际我们使用的 100ask_imx_6ull 开发板只挂接了 512M 的内存，因此我们可选的内存地址范围是 0x80000000~0xa0000000。

裸机实验的链接地址是 0x80100000，裸机实验程序不大，往大里算也不超过 9M，那么 0x80100000 ～ 0x80100000+0x900000 留给程序，0x80100000+0x900000 = 0x81000000。

没使用的内存区域为 0x81000000~0xa0000000，显存大小为 1024*600*4=2457600≈2.4MB。

所以，在 0x81000000~0xa0000000 范围内，任意 2.4MB 的内存都可以用作显存。

本次实验选取的显存是 0x99000000。

# 13.7 编程_简单测试

简单测试编程实现 LCD 全屏顺序显示红，绿，蓝三种颜色。

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/15_LCD 编程/01_simple_test”目录下，你可以编译运行它。

本文档虽然讲的是 IMX6ULL，但是也可以观看 S3C2440 的视频。本节文档对应的视频是 JZ2440 裸机视频中的《第 007 节_LCD 编程_简单测试_P》，它是需要购买的，但是看本文档也完全没问题。

## 13.7.1 初始化 LCD

使用 lcd_manager.c 和 lcd_controller_manager.c 中的管理函数，添加 LCD 屏幕参数和 LCD 控制器到各自的结构体数组中，代码如下：

```
/*添加 LCD 屏幕参数*/
    lcd_7_0_add();

    /*添加 LCD 控制器*/
    lcd_contoller_add();
```

选择 LCD 控制器，选择 LCD：

```
    /*选择 Imx6ull 的 LCD 控制器*/
    select_lcd_controller("Imx6ull");

    /*选择 LCD 屏幕参数*/
    select_lcd("lcd_7.0");
```

最后通过 lcd_controller_init 函数，把选择的 LCD 参数传入到 LCD 控制器中，初始化 LCD 控制器：

```
lcd_controller_init(g_p_lcd_selected);
```

## 13.7.2 使能 LCD

g_p_lcd_controller_selected 变量就是被选中 LCD 控制器，调用里面的 enable 函数即可使能 LCD：

```
    void lcd_controller_enable(void)
    {
        if (g_p_lcd_controller_selected)
        {
            g_p_lcd_controller_selected->enable();
        }
    }
```

对于 IMX6ULL，上述函数最终调用到 Imx6ull_lcd_controller_enable，如下：

```
static void Imx6ull_lcd_controller_enable(void)
{
    LCDIF->CTRL |= 1<<0; /* 使能 6ULL 的 LCD 控制器 */
}
```

## 13.7.3 获取 LCD 参数

在执行简单测试之前，还需要获得 LCD 的关键参数：分辨率和 BPP：

```
74  void get_lcd_params(unsigned int *fb_base, int *xres, int *yres, int *bpp)
75  {
76      *fb_base = g_p_lcd_selected->fb_base;
```

```
77      *xres = g_p_lcd_selected->xres;
78      *yres = g_p_lcd_selected->yres;
79      *bpp = g_p_lcd_selected->bpp;
80  }
```

### 13.7.4 往 framebuffer 中写数据

对 16bpp 或 32bpp，我们分别处理。

对于 16bpp，LCD 控制器中设定的颜色格式是 ARGB555，也就是 ARRRRRGGGGGBBBBB，红绿蓝各占 5 位，最高一位为灰度。

以下代码用来处理 16bpp 的情况：

```
23 /* red */
24  p = (unsigned short *)fb_base;
25      for (x = 0; x < xres; x++)
26          for (y = 0; y < yres; y++)
27              *p++ = 0x7c00;
28
29      /* green */
30      p = (unsigned short *)fb_base;
31      for (x = 0; x < xres; x++)
32          for (y = 0; y < yres; y++)
33              *p++ = 0x3E0;
34
35      /* blue */
36      p = (unsigned short *)fb_base;
37      for (x = 0; x < xres; x++)
38          for (y = 0; y < yres; y++)
39              *p++ = 0x1f;
```

对于 32bpp，LCD 控制器中设定的颜色格式是 ARGB888，也就是 AAAAAAAARRRRRRRRGGGGGGGGBBBBBBBB，红绿蓝各占 8 位，最高字节表示灰度。

以下代码用来处理 32bpp 的情况：

```
47 /* red */
48      p2 = (unsigned int *)fb_base;
49          for (x = 0; x < xres; x++)
50          for (y = 0; y < yres; y++)
51          *p2++ = 0xff0000;
52
53      /*green*/
54      p2 = (unsigned int *)fb_base;
55      for (x = 0; x < xres; x++)
56          for (y = 0; y < yres; y++)
57              *p2++ = 0x00ff00;
58
59      /*blue*/
60      p2 = (unsigned int *)fb_base;
61      for (x = 0; x < xres; x++)
62          for (y = 0; y < yres; y++)
63              *p2++ = 0x0000ff;
```

## 13.8 编程_画点线圆

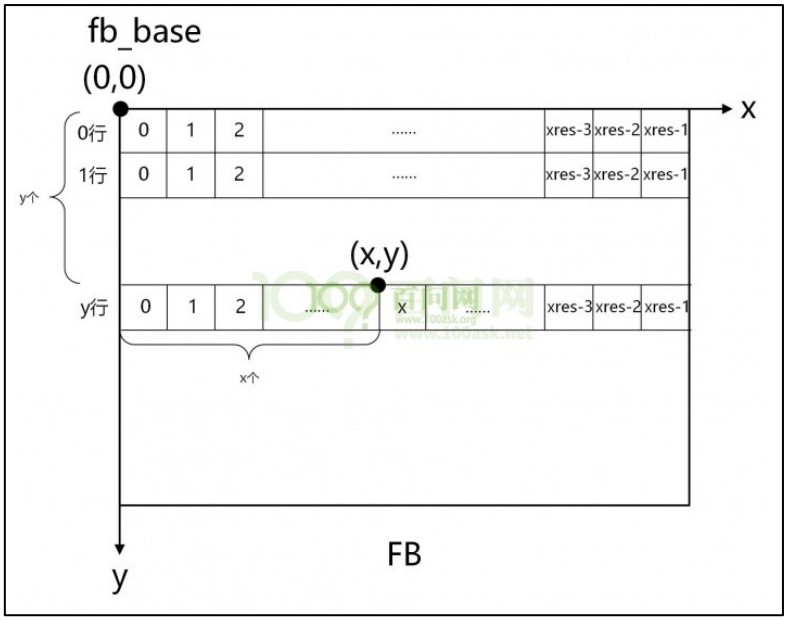
代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/15_LCD 编程/02_dot_line_circle”目录下，你可以编译运行它。

本文档虽然讲的是 IMX6ULL，但是也可以观看 S3C2440 的视频。本节文档

对应的视频是 JZ2440 裸机视频中的《第 008 节_LCD 编程_画点线圆_P》，它是需要购买的，但是看本文档也完全没问题。

## 13.8.1 实现画点

先计算点(x,y)在显存中的位置，如下图所示：




可以得出其计算公式：(x，y)像素起始地址=fb_base+(xres*bpp/8)*y + x*bpp/8。

然后新建立一个 frambuffer.c 文件，在里面实现画点函数。

需要获得 LCD 的参数，我们编写一个 fb_get_lcd_params 函数，它调用 get_lcd_params 来获取 LCD 参数的函数，代码如下：

```
21  void fb_get_lcd_params(void)
22  {
23      get_lcd_params(&fb_base, &xres, &yres, &bpp);
24  }
```

为了方便，我们画点时传入的颜色用 32bpp 来表示，格式为：0x00RRGGB。
如果 LCD 是 32bpp，像素(x,y)在显存的位置上，可以直接填入颜色值。

如果 LCD 是 16bpp，需要把 32 位颜色数据变成 16 位颜色数据。我们使用的颜色格式为 ARGB555，先将 32 位中的 RGB 分离出来，再通过移位指令构造出 RGB555 的颜色。

```
29  int r = (rgb >> 16)& 0xff;
30  int g = (rgb >> 8) & 0xff;
31  int b = rgb & 0xff;
32
33  /* argb555 */
34  r = r >> 3;
35  g = g >> 3;
36  b = b >> 3;
37  return ((r<<10) | (g<<5) | (b));
```

最后，实现画点函数。它需要的参数为：x 坐标，y 坐标，颜色。代码如下：

```
56  void fb_put_pixel(int x, int y, unsigned int color)
57  {
58      unsigned short *pw;  /* 16bpp */
59      unsigned int   *pdw; /* 32bpp */
```

```
60
61      unsigned int pixel_base = fb_base + (xres * bpp / 8) * y + x * bpp / 8;
62
63      switch (bpp)
64      {
65          case 16:
66              pw = (unsigned short *) pixel_base;
67              *pw = convert32bppto16bpp(color);
68          break;
69          case 32:
70              pdw = (unsigned int *) pixel_base;
71              *pdw = color;
72              break;
73      }
74  }
```

## 13.8.2 实现画线

画线的具体原理不是我们的主要内容，我们直接百度“C 语言 LCD 画线”可以得到相关的实现代码，比如这篇博客：http://blog.csdn.net/p1126500468/article/details/50428613

新建一个 geometry.c，复制博客中代码，替换里面的描点显示函数即可

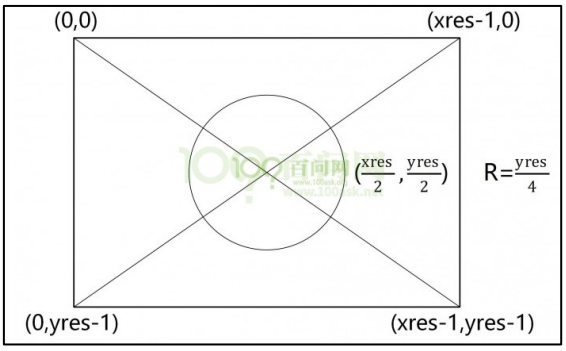
最后在主函数测试程序里，加上画线的测试代码：

```
69  fb_get_lcd_params();
70
71  delay(100000);
72
73  draw_line(0, 0, xres - 1, 0, 0xff0000);
74  delay(100000);
75  draw_line(xres - 1, 0, xres - 1, yres - 1, 0xffff00);
76  delay(100000);
77  draw_line(0, yres - 1, xres - 1, yres - 1, 0xff00aa);
78  delay(100000);
79  draw_line(0, 0, 0, yres - 1, 0xff00ef);
80  delay(100000);
81  draw_line(0, 0, xres - 1, yres - 1, 0xff4500);
82  delay(100000);
83  draw_line(xres - 1, 0, 0, yres - 1, 0xff0780);
```

## 13.8.3 实现画圆

画圆的具体原理不是我们的主要内容，我们直接百度“C 语言 LCD 画圆”可以得到相关的实现代码，比如这篇博客：http://blog.csdn.net/p1126500468/article/details/50428613

在 geometry.c 中添加博客中代码，替换里面的描点显示函数即可，代码如下：



调用画圆函数时，只需提供圆心的坐标，圆的半径，圆的颜色。
最后在主函数测试程序里，加上画圆画线的测试代码：

```
86  draw_circle(xres/2, yres/2, yres/4, 0xff00);
```

## 13.9 编程_显示文字

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/15_LCD 编程/03_font_test”目录下，你可以编译运行它。

本文档虽然讲的是 IMX6ULL，但是也可以观看 S3C2440 的视频。本节文档对应的视频是 JZ2440 裸机视频中的《第 009 节_LCD 编程_显示文字_P》，它是需要购买的，但是看本文档也完全没问题。

### 13.9.1 获取 LCD 参数

新建一个 font.c，由于新建文件中没有 LCD 参数，因此也需要获取 LCD 参数，代码如下：

```
20  void font_init(void)
21  {
22      get_lcd_params(&fb_base, &xres, &yres, &bpp);
23  }
```

### 13.9.2 编写单个字符显示函数

文字也是由点构成的，它是一个个点组成的点阵。可以参考 Linux 内核源码，从中得到字符的点阵，打开内核文件 font_8x16.c，可以看到里面的 A 字符内容如下：

```
    /* 65 0x41 'A' */
    0x00, /* 00000000 */
    0x00, /* 00000000 */
    0x10, /* 00010000 */
    0x38, /* 00111000 */
    0x6c, /* 01101100 */
    0xc6, /* 11000110 */
    0xc6, /* 11000110 */
    0xfe, /* 11111110 */
    0xc6, /* 11000110 */
    0xc6, /* 11000110 */
    0xc6, /* 11000110 */
    0xc6, /* 11000110 */
    0x00, /* 00000000 */
    0x00, /* 00000000 */
    0x00, /* 00000000 */
    0x00, /* 00000000 */
```

根据这些数据，在一个 8*16 的区域里，将为 1 的点显示出来，为 0 的则不显示，最终将呈现一个字母“A”。

显示字母，就是根据字母的点阵在 LCD 上显示像素。点阵中每一位对应一个像素，位值为 0 时显示一种颜色，位值为 1 时显示另一种颜色。

需要的步骤如下：

① 根据带显示的字符的 ascii 码在 fontdata_8x16 中得到点阵数据；
② 根据点阵来设置对应象素的颜色；
③ 根据点阵的某位决定是否描颜色。

代码如下：

```
35  void fb_print_char(int x, int y, char c, unsigned int color)
36  {
37      int i, j;
38
39      /* 根据 c 的 ascii 码在 fontdata_8x16 中得到点阵数据 */
40      const unsigned char *dots = &fontdata_8x16[c * 16];
41
42      unsigned char data;
43      int bit;
44
45      /* 根据点阵来设置对应象素的颜色 */
46      for (j = y; j < y+16; j++)
47      {
48          data = *dots++;
49          bit = 7;
50          for (i = x; i < x+8; i++)
51
52              /* 根据点阵的某位决定是否描颜色 */
53              if (data & (1<<bit))
54                  fb_put_pixel(i, j, color);
55              bit--;
56          }
57      }
58  }
```

在 font_8x16.c 里面，每个字符占据 16 位，因此想要根据 ascii 码找到对应的点阵数据，需要对应的乘 16，再取地址，得到该字符的首地址。

然后再根据每个点阵数据每位是否为 1，来调用描点函数 fb_put_pixel()。这样，依次显示 16 个点阵数据，最终在 LCD 上显示出字符。

## 13.9.3 编写实现字符串显示函数

显示字符串，那么就需要在每显示完一个字符后，x 方向加 8 个像素，同时考虑是否超出屏幕（xres）显示范围进行换行处理（y+16）。

代码如下：

```
71  void fb_print_string(int x, int y, char* str, unsigned int color)
72  {
73      int i = 0, j;
74
75      while (str[i])
76      {
77          if (str[i] == '\n')
78              y = y+16;
79          else if (str[i] == '\r')
80              x = 0;
81
82          else
83          {
84              fb_print_char(x, y, str[i], color);
85              x = x+8;
86              if (x >= xres) /* 换行 */
87              {
88                  x = 0;
89                  y = y+16;
90              }
```

```
91          }
92          i++;
93      }
94  }
```

最后在在主函数里，如下调用：

```
98  fb_print_string(10, 10, "www.100ask.net\n\r100ask.taobao.com", 0xff00);
```

# 第14章 I2C 编程

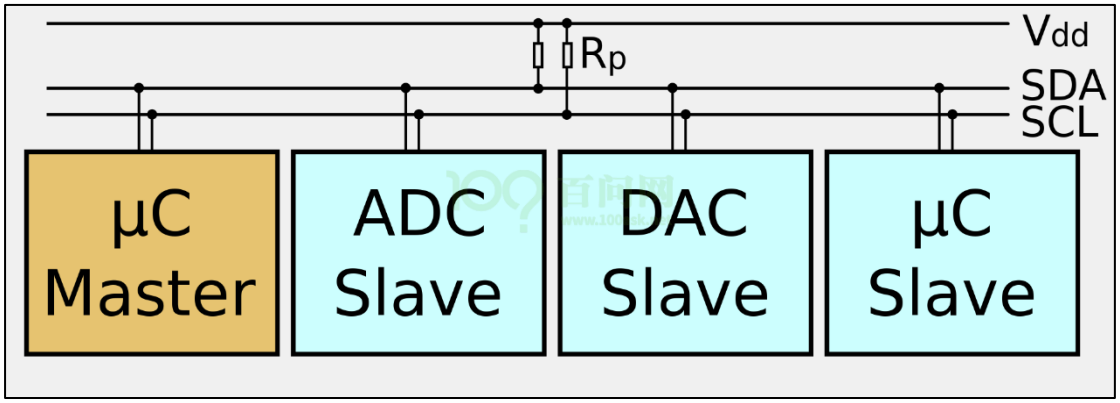
I2C（Inter-Integrated Circuit BUS）是 I2C BUS 简称，中文为集成电路总线，是目前应用最广泛的总线之一。和 IMX6ULL 有些相关的是，刚好该总线是 NXP 前身的 PHILIPS 设计。

## 14.1 I2C 协议

### 14.1.1 概述

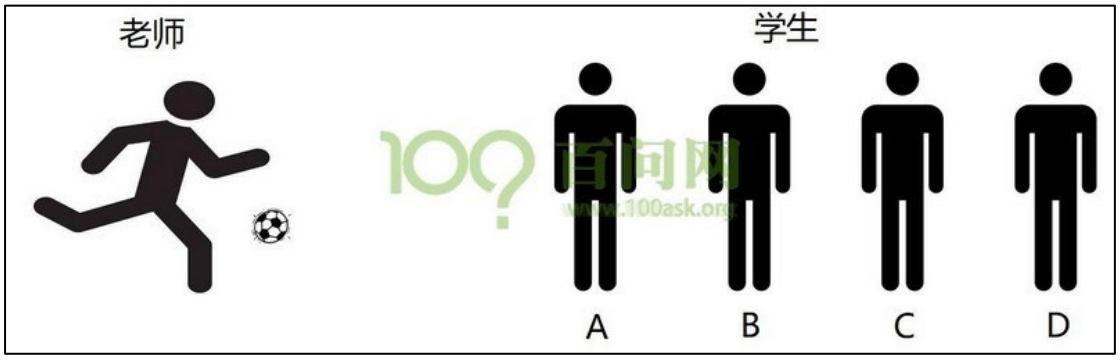
I2C 是一种串行通信总线，使用多主从架构，最初设计目的为了让主板、嵌入式系统或手机用来连接低速周边设备。多用于小数据量的场合，有传输距离短，任意时刻只能有一个主机等特性。严格意义上讲，I2C 应该是软硬件结合体，所以我们将分物理层和协议层来介绍该总线。

I2C 总线结构如下图：




传输数据时，我们需要发数据，从主设备发送到从设备上去；也需要把数据从从设备传送到主设备上去，数据涉及到双向传输。

对于 I2C 通信的过程，下面使用一个形象的生活例子进行类比。




体育老师：可以把球发给学生，也可以把球从学生中接过来。

① 发球：

a) 老师说：注意了(start)；

b) 老师对 A 学生说，我要球发给你(A 就是地址)；

c) 老师就把球发出去了（传输）；

d) A 收到球之后，应该告诉老师一声（回应）；

e) 老师说下课（停止）。

② 接球：

a) 老师说注意了(start)；

b) 老师说：B 把球发给我(B 是地址)；

c) B 就把球发给老师（传输）；

d) 老师收到球之后，给 B 说一声，表示收到球了（回应）；

e) 老师说下课（停止）。

我们就使用这个简单的例子，来解释一下 IIC 的传输协议：
① 老师说注意了，表示开始信号(start)
② 老师告诉某个学生，表示发送地址(address)
③ 老师发球/接球，表示数据的传输
④ 老师/学生收到球，回应表示：回应信号(ACK)
⑤ 老师说下课，表示 IIC 传输接受(P)

### 14.1.2 物理层

- **特性 1：半双工（非全双工）**

I2C 总线中只使用两条线路：SDA、SCL。

① SDA(串行数据线)：

主芯片通过一根 SDA 线既可以把数据发给从设备，也可以从 SDA 上读取数据。在 I2C 设备内部有两个引脚（发送引脚/接受引脚），它们都连接到外部的 SDA 线上。

② SCL(串行时钟线)：

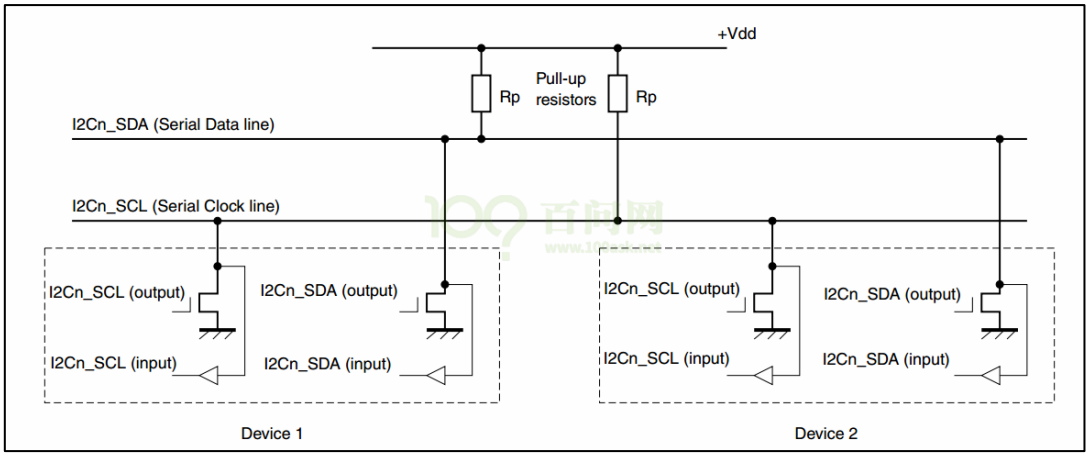
I2C 主设备发出时钟，从设备接收时钟。

SDA 和 SCL 引脚的内部电路结构一致，引脚的输出驱动与输入缓冲连在一起。其中输出为漏极开路的场效应管、输入缓冲为一只高输入阻抗的同相器。这样结构有如下特性：

- 由于 SDA、SCL 为漏极开路结构，借助于外部的上拉电阻实现了信号的“线与”逻辑；

- 引脚在输出信号的同时还作用输入信号供内部进行检测，当输出与输入不一致时，就表示有问题发生了。这为 “时钟同步”和“总线仲裁”提供硬件基础。

SDA 和 CLK 连接线上连有两个上拉电阻，当总线空闲时，两根线均为高电平。连到总线上的任一器件输出的低电平，都将使总线的信号变低。
物理层连接如下图所示：




➢ **特性 2：地址和角色可配置**

每个连接到总线的器件都可以通过唯一的地址和其它器件通信，主机/从机角色和地址可配置，主机可以作为主机发送器和主机接收器。

➢ **特性 3：多主机**

IIC 是真正的多主机总线，I2C 设备可以在通讯过程转变成主机。如果两个或更多的主机同时请求总线，可以通过冲突检测和仲裁防止总线数据被破坏。

➢ **特性 4：传输速率**

传输速率在标准模式下可以达到 100kb/s，快速模式下可以达到 400kb/s。

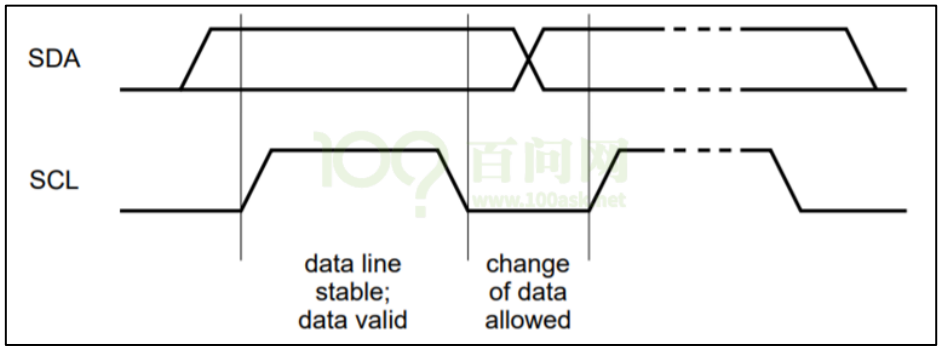
➢ **特性 5：负载和距离**

节点的最大数量受限于地址空间以及总线电容决定，另外总电容也限制了实际通信距离只有几米。

## 14.1.3 协议层

➢ **数据有效性**

I2C 协议的数据有效性是靠时钟来保证的，在时钟的高电平周期内，SDA 线上的数据必须保持稳定。数据线仅可以在时钟 SCL 为低电平时改变。

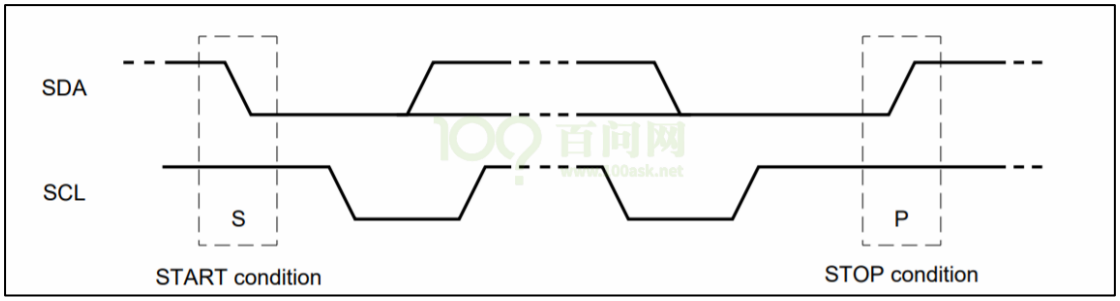



➢ **起始和结束条件**

- 起始条件：当 SCL 为高电平的时候，SDA 线上由高到低的跳变被定义为起始条件。

- 结束条件：当 SCL 为高电平的时候，SDA 线上由低到高的跳变被定义为停止条件。

要注意起始和终止信号都是由主机发出的，连接到 I2C 总线上的器件，若具有 I2C 总线的硬件接口，则很容易检测到起始和终止信号。



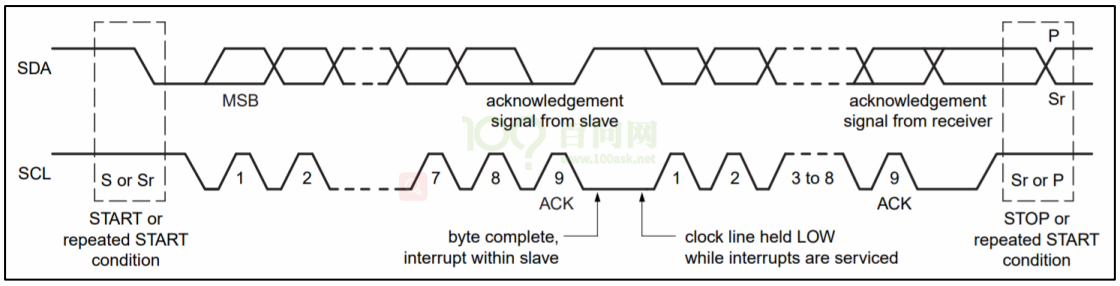

总线在起始条件之后，视为忙状态，在停止条件之后被视为空闲状态。

- **应答**

每当主机向从机发送完一个字节的数据，主机总是需要等待从机给出一个应答信号，以确认从机是否成功接收到了数据，从机应答主机所需要的时钟仍是主机提供的，应答出现在每一次主机完成 8 个数据位传输后紧跟着的时钟周期，低电平 0 表示应答，1 表示非应答。

- **数据帧格式**

SDA 线上每个字节必须是 8 位长，在每个传输(transfer)中所传输字节数没有限制，每个字节后面必须跟一个 ACK。8 位数据中，先传输最高有效位（MSB）传输。



## 14.2 IMX6ULL 的 I2C 控制器操作与寄存器介绍

参考资料：网盘开发板配套资料“06_Datasheet（数据手册）/Core_board/CPU/IMX6ULLRM.pdf”：《Chapter 31: I2C Controller (I2C)》。

IMX6ULL 的 I2C 提供了标准 I2C 主设备、从设备的功能，本章把 IMX6ULL 用作 I2C 主设备。

掌握 IMX6ULL I2C 控制器的使用，重点要熟悉 IMX6ULL I2C 的寄存器的操作。

重点介绍寄存器之前，我们先来看一下 IMX6ULL 的 I2C 控制器的框图，对 IMX6ULL I2C 的结构和特性有一个初步认识。

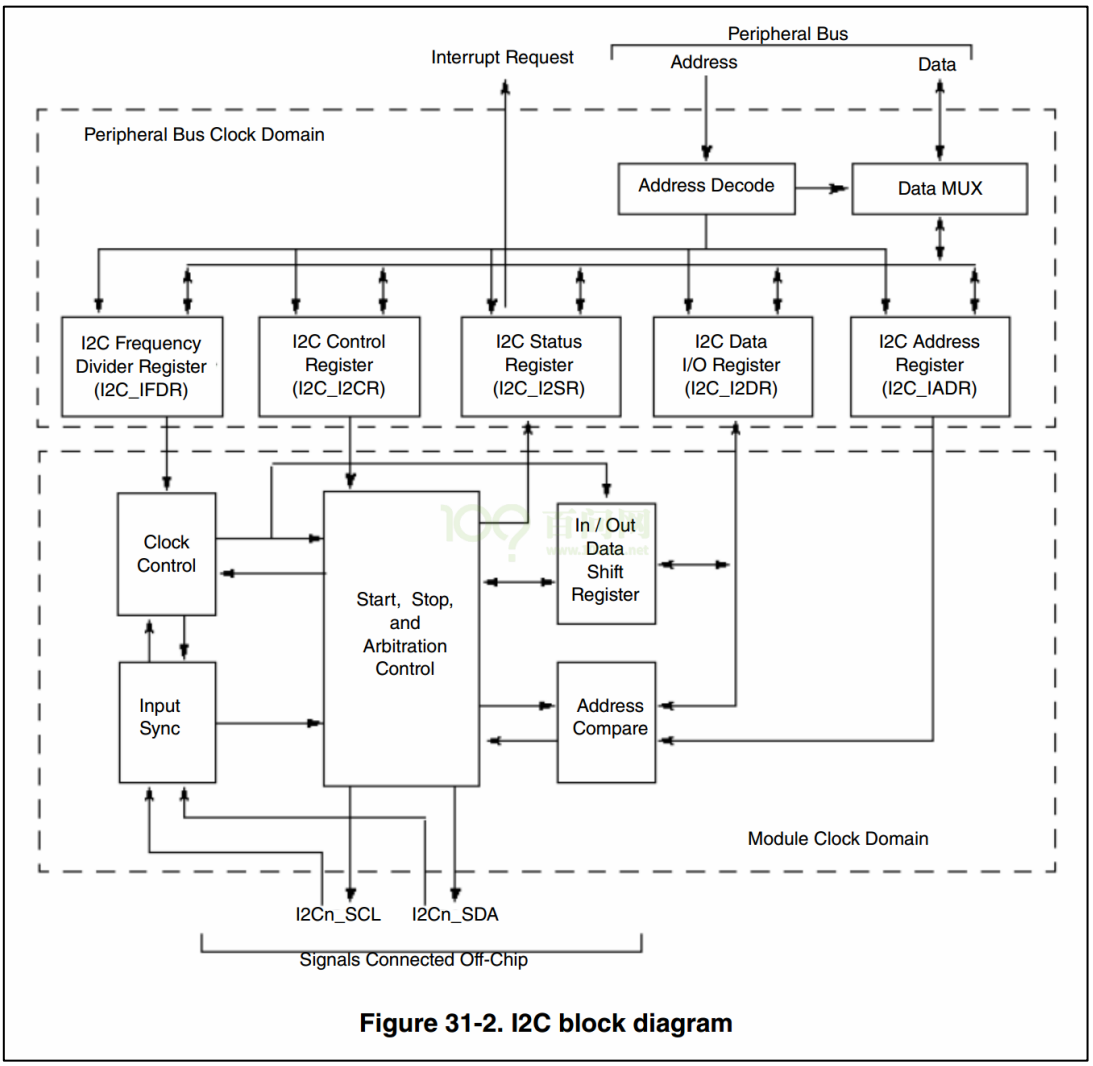



**Figure 31-2. I2C block diagram**

IMX6ULL 的 I2C 控制器有如下额外特性：

① 多主机运行。
② 64 种不同的串行时钟频率之一的软件可编程性。
③ 软件可选择的应答位。
④ 中断驱动，逐字节数据传输。
⑤ 仲裁丢失中断与自动模式切换从主到从。
⑥ 启动和停止信号生成/检测。
⑦ 重复启动信号生成。

⑧ 应答位生成和检测。
⑨ 总线忙检测。

另外的 IMX6ULL 的 I2C 也支持两种模式：标准模式和快速模式，标准模式下 I2C 数据传输速率最高是 100Kbits/s，在快速模式下数据传输速率最高为 400Kbits/s。

通过上面的介绍对 IMX6ULL 的 I2C 控制器有了整体认识，下面结合 I2C 框图对重点寄存器进行介绍。

## 14.2.1 I2C Memory Map

IMX6ULL 中有 4 个 I2C 控制器，每个 I2C 控制器有 5 个 16-bit 的寄存器。

注意：寄存器在偏移量 0x0002/0x0006/0x000A/0x000E 作为保留位。

可以看到 I2C1 的入口地址为 21A_0000，这个我们重点关注，后面做实验，编程会使用到。

**I2C memory map**

| Absolute address (hex) | Register name | Width (in bits) | Access | Reset value | Section/ page |
|---|---|---|---|---|---|
| 21A_0000 | I2C Address Register (I2C1_IADR) | 16 | R/W | 0000h | 31.7.1/1463 |
| 21A_0004 | I2C Frequency Divider Register (I2C1_IFDR) | 16 | R/W | 0000h | 31.7.2/1463 |
| 21A_0008 | I2C Control Register (I2C1_I2CR) | 16 | R/W | 0000h | 31.7.3/1465 |
| 21A_000C | I2C Status Register (I2C1_I2SR) | 16 | R/W | 0081h | 31.7.4/1466 |
| 21A_0010 | I2C Data I/O Register (I2C1_I2DR) | 16 | R/W | 0000h | 31.7.5/1468 |
| 21A_4000 | I2C Address Register (I2C2_IADR) | 16 | R/W | 0000h | 31.7.1/1463 |
| 21A_4004 | I2C Frequency Divider Register (I2C2_IFDR) | 16 | R/W | 0000h | 31.7.2/1463 |
| 21A_4008 | I2C Control Register (I2C2_I2CR) | 16 | R/W | 0000h | 31.7.3/1465 |
| 21A_400C | I2C Status Register (I2C2_I2SR) | 16 | R/W | 0081h | 31.7.4/1466 |
| 21A_4010 | I2C Data I/O Register (I2C2_I2DR) | 16 | R/W | 0000h | 31.7.5/1468 |
| 21A_8000 | I2C Address Register (I2C3_IADR) | 16 | R/W | 0000h | 31.7.1/1463 |
| 21A_8004 | I2C Frequency Divider Register (I2C3_IFDR) | 16 | R/W | 0000h | 31.7.2/1463 |
| 21A_8008 | I2C Control Register (I2C3_I2CR) | 16 | R/W | 0000h | 31.7.3/1465 |
| 21A_800C | I2C Status Register (I2C3_I2SR) | 16 | R/W | 0081h | 31.7.4/1466 |
| 21A_8010 | I2C Data I/O Register (I2C3_I2DR) | 16 | R/W | 0000h | 31.7.5/1468 |
| 21F_8000 | I2C Address Register (I2C4_IADR) | 16 | R/W | 0000h | 31.7.1/1463 |
| 21F_8004 | I2C Frequency Divider Register (I2C4_IFDR) | 16 | R/W | 0000h | 31.7.2/1463 |
| 21F_8008 | I2C Control Register (I2C4_I2CR) | 16 | R/W | 0000h | 31.7.3/1465 |
| 21F_800C | I2C Status Register (I2C4_I2SR) | 16 | R/W | 0081h | 31.7.4/1466 |
| 21F_8010 | I2C Data I/O Register (I2C4_I2DR) | 16 | R/W | 0000h | 31.7.5/1468 |

## 14.2.2 寄存器介绍

我们先把这些寄存器的用途做个介绍，后面在编程框架那里会结合位图来讲解。下面这部分也可以先跳过，回头当表来查。

➢ **I2C 地址寄存器(I2Cx_IADR)**

Address: Base address + 0h offset

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Read | 0 | | | | | | | | ADR | | | | | | | 0 |
| Write | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [7:1] | ADR | R/W | 作为 I2C 从设备时的地址；使用软件复位时，这个寄存器不受影响 |

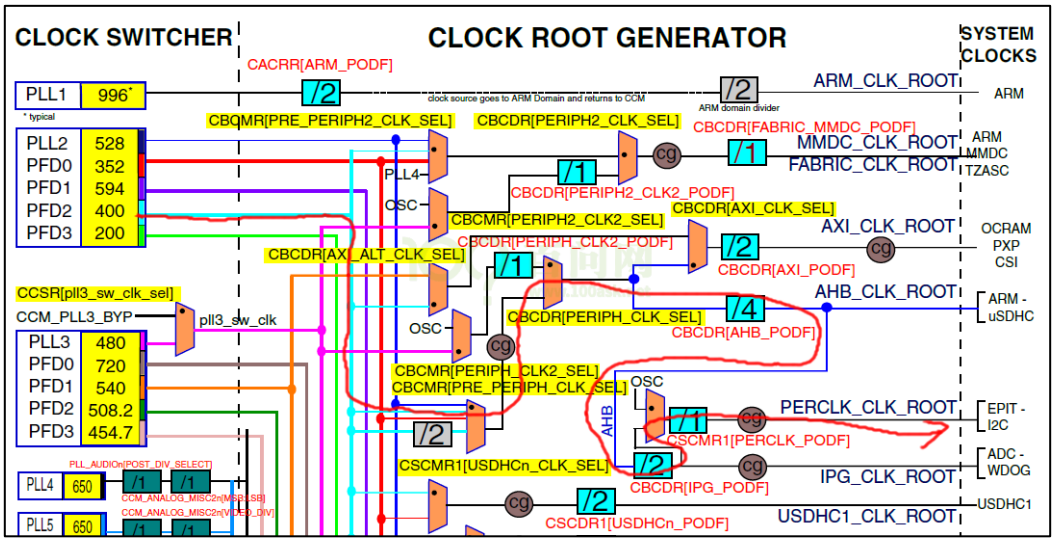
## ➢ I2C 分频寄存器(I2Cx_IFDR)

Address: Base address + 4h offset

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Read | 0 | | | | | | | | IC | | | | | | | |
| Write | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [5:0] | IC | R/W | I2C clock rate，用来设置 SCL 时钟，IC 值对应的分频系数要查表。<br>在数据传输过程中，IC 值不能改变；但是在发出 S 信号之前，可以改变 IC 值。 |

I2C 的时钟树如下图红色箭头所示：




I2C 的时钟源来源于 PLL2 PFD2，根据时钟树计算出 PER_CLK_ROOT：

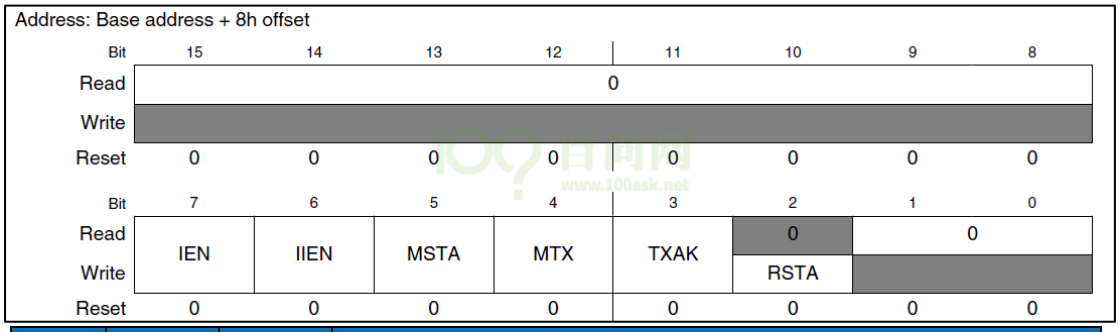
```
PLL2 = 528 MHz
PLL2_PFD2 = 528 * 18 / 24 = 396 MHz
IPG_CLK_ROOT = (PLL2_PFD2 / ahb_podf )/ ipg_podf = (396 MHz/4)/2 = 49.5 MHz
PER_CLK_ROOT = IPG_CLK_ROOT/perclk_podf = 49.5  MHz/1 = 49.5 MHz
```

要设置 I2C 的波特率为 100K， I2C 分频值 = 49500000/100000 = 495 参考 Table 31-3. I2C_IFDR Register Field Values ，表中 0x37 对应的 512 最接近，所以 I2CX_IFDR[IC]取值为 0x37：

**Table 31-3. I2C_IFDR Register Field Values**

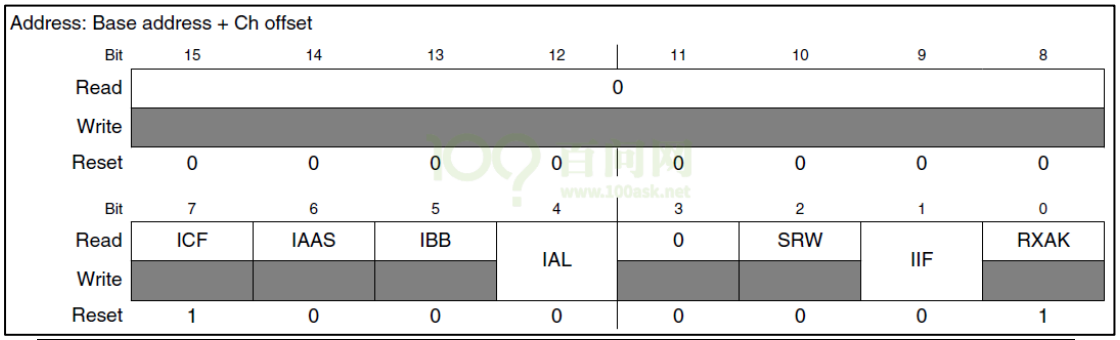
| IC | Divider | IC | Divider | IC | Divider | IC | Divider |
|---|---|---|---|---|---|---|---|
| 0x00 | 30 | 0x10 | 288 | 0x20 | 22 | 0x30 | 160 |
| 0x01 | 32 | 0x11 | 320 | 0x21 | 24 | 0x31 | 192 |
| 0x02 | 36 | 0x12 | 384 | 0x22 | 26 | 0x32 | 224 |
| 0x03 | 42 | 0x13 | 480 | 0x23 | 28 | 0x33 | 256 |
| 0x04 | 48 | 0x14 | 576 | 0x24 | 32 | 0x34 | 320 |
| 0x05 | 52 | 0x15 | 640 | 0x25 | 36 | 0x35 | 384 |
| 0x06 | 60 | 0x16 | 768 | 0x26 | 40 | 0x36 | 448 |
| 0x07 | 72 | 0x17 | 960 | 0x27 | 44 | 0x37 | 512 |
| 0x08 | 80 | 0x18 | 1152 | 0x28 | 48 | 0x38 | 640 |
| 0x09 | 88 | 0x19 | 1280 | 0x29 | 56 | 0x39 | 768 |
| 0x0A | 104 | 0x1A | 1536 | 0x2A | 64 | 0x3A | 896 |
| 0x0B | 128 | 0x1B | 1920 | 0x2B | 72 | 0x3B | 1024 |
| 0x0C | 144 | 0x1C | 2304 | 0x2C | 80 | 0x3C | 1280 |
| 0x0D | 160 | 0x1D | 2560 | 0x2D | 96 | 0x3D | 1536 |
| 0x0E | 192 | 0x1E | 3072 | 0x2E | 112 | 0x3E | 1792 |
| 0x0F | 240 | 0x1F | 3840 | 0x2F | 128 | 0x3F | 2048 |

## ➢ I2C 控制寄存器(I2Cx_I2CR)



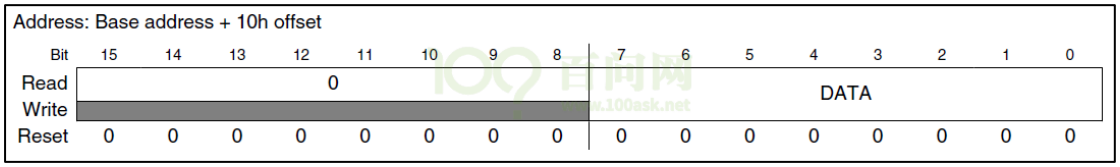

| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [7] | IEN | R/W | I2C 使能，<br>0：I2C 控制器被禁止，但是还可以访问它的寄存器；<br>1：I2C 控制器使能，要想让本寄存器中其他位起效，此位必须先置 1 |
| [6] | IIEN | R/W | I2C 中断使能，<br>0：I2C 中断禁止，但是中断状态位(I2C_I2SR[IIF])还是可以使用的；<br>1：I2C 中断使能，发生中断时，中断状态位(I2C_I2SR[IIF])也会被设置 |
| [5] | MSTA | R/W | 主从模式选择，<br>0：从设备模式，MSTA 从 1 变 0 时，会发出 STOP 信号，并变为从设备模式；<br>1：主机模式，MSTA 从 0 变 1 时，会发即 STAT 信号，并变为主机模式。<br>注意 1：I2C 主设备失去总线时，硬件会清除此位，但是不会发出 STOP 信号<br>注意 2：要修改此位时，要先提供 I2C 控制器时钟<br>注意 3：软件清除此位时，会发出 STOP 信号；如果失去总线，硬件会清除此位 |
| [4] | MTX | R/W | 发送/接收模式，<br>0：接收模式，<br>1：发送模式。<br>作为主机时，应该根据数据传输的方向设置 MTX 位，当然，发送 I2C 设备地址时 MTX 总是 1。 |
| [3] | TXAK | R/W | 发送响应使能，当 I2C 设备处于接收状态时，此位才有效，<br>0：在第 9 个时钟，发送响应信号，即把 SDA 拉低；<br>1：在第 9 个时钟，不发送响应信号 |
| [2] | RSTA | R/W | Repeat start，读该位时总得到 0，<br>写：<br>0：不发送 repeat start 信号；<br>1：发送 repeat start 信号 |

## ➢ I2C 状态寄存器(I2Cx_I2SR)




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [7] | ICF | R/W | Data transferring bit，数据正在传输标志，<br>0：正在传输；<br>1：传输结束，当第 9 个时钟发出后，硬件置位 |
| [6] | IAAS | R/W | I2C address as a slave bit，I2C 控制器作用从设备时，通过此位来判断是否有人来寻址它：<br>0：没人找我；<br>1：有别的 I2C 主机在找我了 |
| [5] | IBB | R/W | I2C bus busy bit，用来显示总线状态，<br>0：总线空闲，发现 STOP 信号时，IBB 就被清 0；<br>1：总线忙，发现 START 信号时，IBB 就被置 1 |
| [4] | IAL | R/W | Arbitration lost，此位由硬件设置，软件要写入 0 来清零，<br>0：没有丢失总线，<br>1：总线丢失了 |
| [2] | SRW | R/W | Slave read/write，作为从设备时，主机发送设备地址时会带有读写位，读写位的内容就写入 SRW 中，<br>0：slave receive，主设备要写数据；<br>1：slave transmit，主设备要读数据 |
| [1] | IIF | | I2C 中断状态，写 0 清中断，<br>0：没有 I2C 中断发生；<br>1：I2C 中断发生了，如果 IIEN=1 则会产生中断，有这些中断：<br>一个字节传输完毕；<br>作为从设备，接收到了一个吻合的地址；<br>总线丢失。 |
| [0] | RXAK | | Received acknowledge，是否接收到回应信号，<br>0：收到了：在一个字节传完后第 9 个时钟周期，收到了回应信号；<br>1：没收到 |

## ➢ I2C 数据寄存器(I2Cx_I2DR)

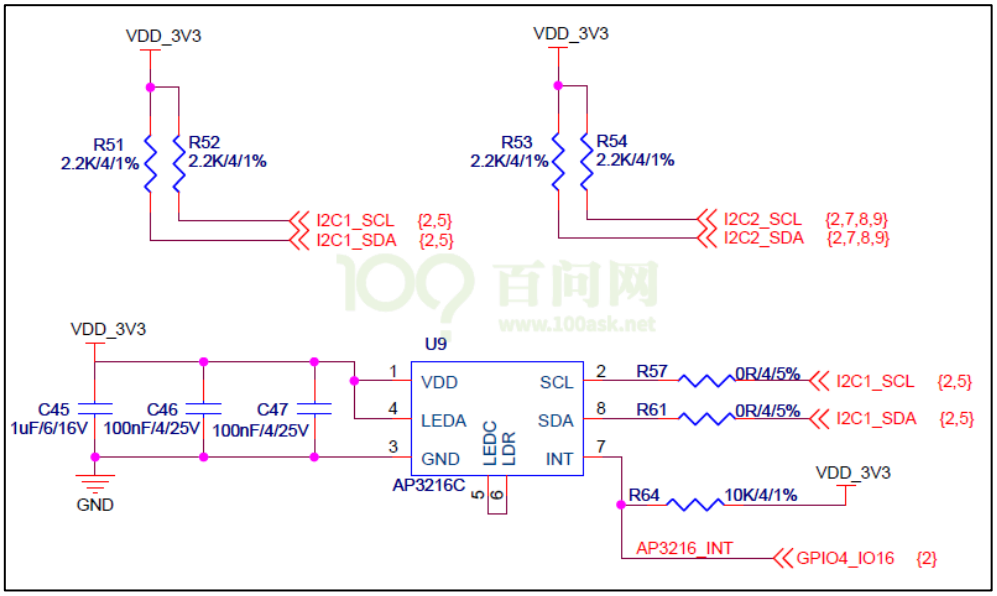


| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [7:0] | DATA | R/W | Data Byte。<br>对于接收：存有接收到的最新数据，软件把数据读出来；<br>对于发送：存放下一个即将要发送的数据，这是软件写进去的。 |

## 14.3 AP3216C 操作方法

参考资料：网盘开发板配套资料“06_Datasheet（数据手册）/Base_board/100ask_imx6ull 底板_规格书/AP3216C.pdf”。

100ASK_IMX6ULL 中集成有 AP3216C 芯片，它接的 I2C1，原理图如下：




### 14.3.1 AP3216C 简介

AP3216C 芯片集成了光强传感器（ALS: Ambient Light Sensor），接近传感器（PS: Proximity Sensor），还有一个红外 LED（IR LED）。

这个芯片设计的用途是给手机之类的使用，比如：返回当前环境光强以便调整屏幕亮度；用户接听电话时，将手机放置在耳边后，自动关闭屏幕避免用户误触碰。

该芯片通过 I2C 接口作为 slave 与主控制器相连，支持中断。

### 14.3.2 AP3216C 特性

① 接口：I2C
② 速率：FS mode 可达 400kbit/s
③ 模式：ALS/PS+IR/ALS+PS+IR/PD/ALS once/SW Reset/PS+IR once/ALS+PS+IR once。
④ 内置温度补偿电路。
⑤ 工作温度：-30℃ to +80℃
⑥ ALS(Ambient Light Photo Sensor)，光强传感器：

a) (0-65536)16 位有效的线性输出

b) 4 量程动态范围可选择

c) 防闪烁（50/60 HZ）

d) 高敏感性@黑玻璃

e) 窗口损失补偿

⑦ PS(Proximity Sensor)，接近传感器：

a) (0-1023)10 位有效线性输出

b) 4 增益

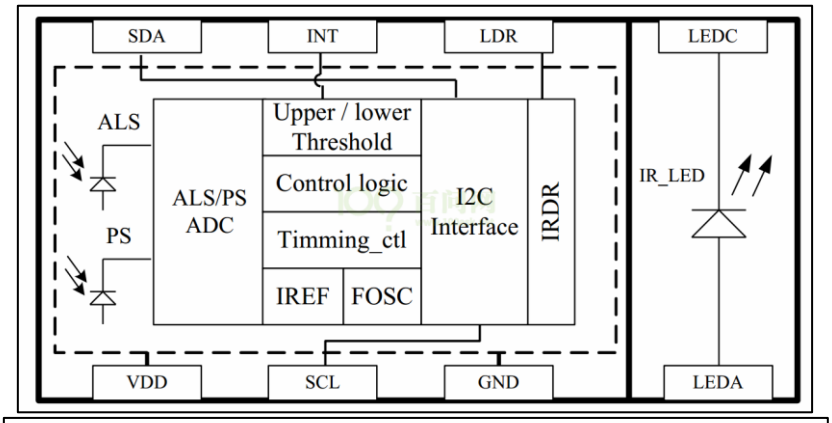
c) 高环境光抑制

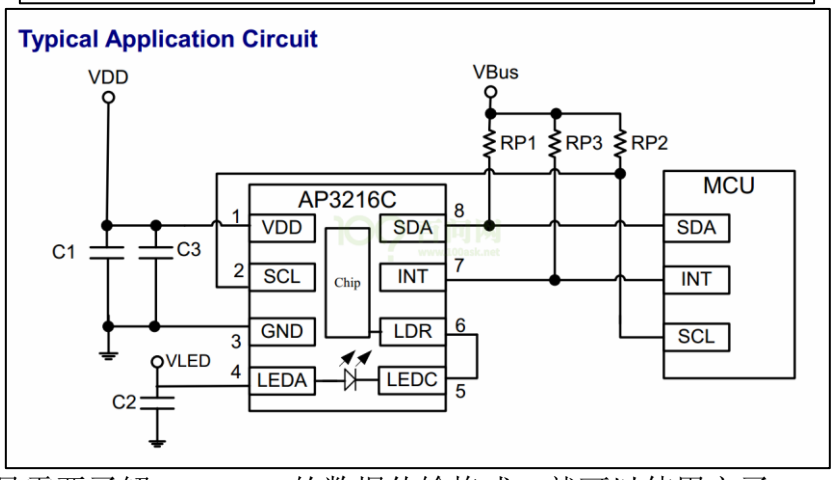
d) 串扰补偿

⑧ 封装大小：4.1mm x 2.4mm x 1.35mm

### 14.3.3 AP3216C 结构图

下面是 AP3216C 的结构图以及典型电路应用图：




我们只需要了解 AP3216C 的数据传输格式，就可以使用它了。

### 14.3.4 AP3216C 寄存器

要访问 AP3216C，首先要知道它的设备地址，然后理清楚它的内部寄存器。

I2C 主机与某个 I2C 设备通信时，发出的第 1 个字节中高 7 位是 I2C 设备的地址，第 8 位是“读/写”位。

AP3216C 的设备地址是 0x1E。从 AP3216C 的芯片手册中，可以看到下表，里面列出了所有的寄存器：

**System Register Descriptions**

| ADDR (Hex) | REGISTER NAME | Bits | REGISTER COMMAND | FUNCTIONS/DESCRIPTION |
|---|---|---|---|---|
| 0x00 | System Configuration (Default : 0x00) | 2:0 | System Mode (Default : 000) | 000: Power down (Default) |
| | | | | 001: ALS function active |
| | | | | 010: PS+IR function active |
| | | | | 011: ALS and PS+IR functions active |
| | | | | 100: SW reset |
| | | | | 101: ALS function once |
| | | | | 110: PS+IR function once |
| | | | | 111: ALS and PS+IR functions once |
| 0x01 | INT Status | 1 | PS Int (Read only) (Default : 0) | 0: Interrupt is cleared or not triggered yet<br>1: Interrupt is triggered<br>**Note1** |
| | | 0 | ALS Int (Read only) (Default : 0) | 0: Interrupt is cleared or not triggered yet<br>1: Interrupt is triggered<br>**Note1** |
| 0x02 | INT Clear Manner | 0 | Clear Manner (Default : 0) | 0: INT is automatically cleared by reading data registers<br>1: Software clear after writing 1 into address 0x01 each bit |
| 0x0A | IR Data Low | 7 | IR overflow (Read only) | 0: Valid IR and PS data<br>1: Invalid IR and PS data |
| | | 1:0 | (Read only) | IR lower byte of ADC output |
| 0x0B | IR Data High | 7:0 | (Read only) | IR higher byte of ADC output |
| 0x0C | ALS Data Low | 7:0 | (Read only) | ALS lower byte of ADC output |
| 0x0D | ALS Data High | 7:0 | (Read only) | ALS higher byte of ADC output |
| 0x0E | PS Data Low | 7 | Object detect (Read only) | 0: The object leaving<br>1: The object closed |
| | | 6 | IR overflow (Read only) | 0: Valid IR, PS data and object detected<br>1: Invalid IR, PS data and object detected |
| | | 3:0 | (Read only) | PS lower byte of ADC output |
| 0x0F | PS Data High | 7 | Object detect (Read only) | 0: The object leaving<br>1: The object closed |
| | | 6 | IR overflow (Read only) | 0: Valid IR, PS data and object detected<br>1: Invalid IR, PS data and object detected |
| | | 5:0 | (Read only) | PS higher byte of ADC output |

## 14.3.5 AP3216C 寄存器读写方法

I2C 中线上既有 AP3216C，也可能有其他芯片，所以要读写 AP3216C，有 2 个问题要解决：

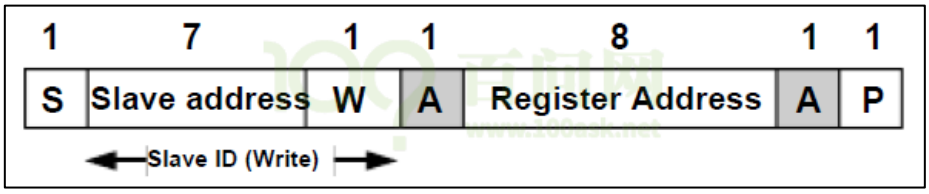
① 怎么访问到 AP3216C？发出设备地址 0x1E。

② 怎么访问某个寄存器？发出寄存器地址。

根据上述 2 点，看懂 AP3216C 芯片手册中的操作时序图，并不难。

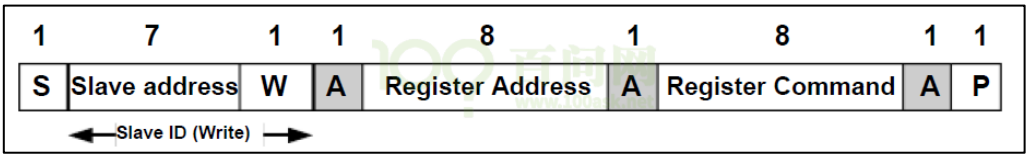
写 AP3216C 的寄存器时，有 2 类方法：

① 只需要发起写操作，不用传输数据：




从上图中，只需要发送设备地址、寄存器地址就可以了。

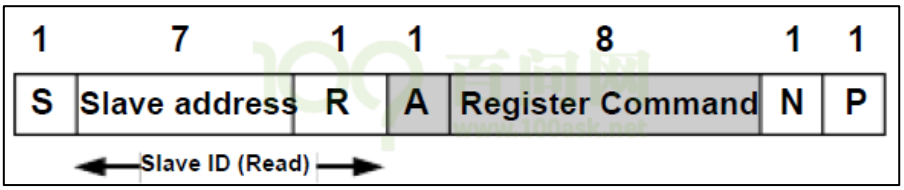
② 需要提供寄存器的数据：




在 AP3216C 中，寄存器的数据被称为“Register Command”。

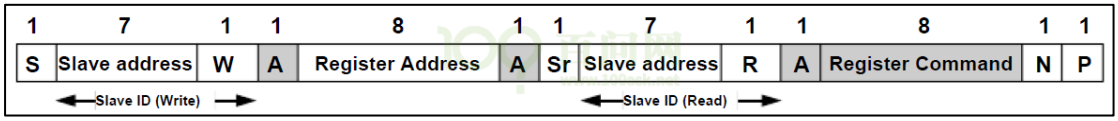
读 AP3216C 的寄存器时，也有 2 类方法：

① 直接读，不发寄存器地址：




那么它返回的是哪个寄存器的值？可能是上一次读过的寄存器，也可能是上一次读过的寄存器的下一个寄存器，有待测试。

② 需要提供寄存器地址：




需要发起 2 次操作：

a) 第一次是写操作，发出设备地址、寄存器地址；

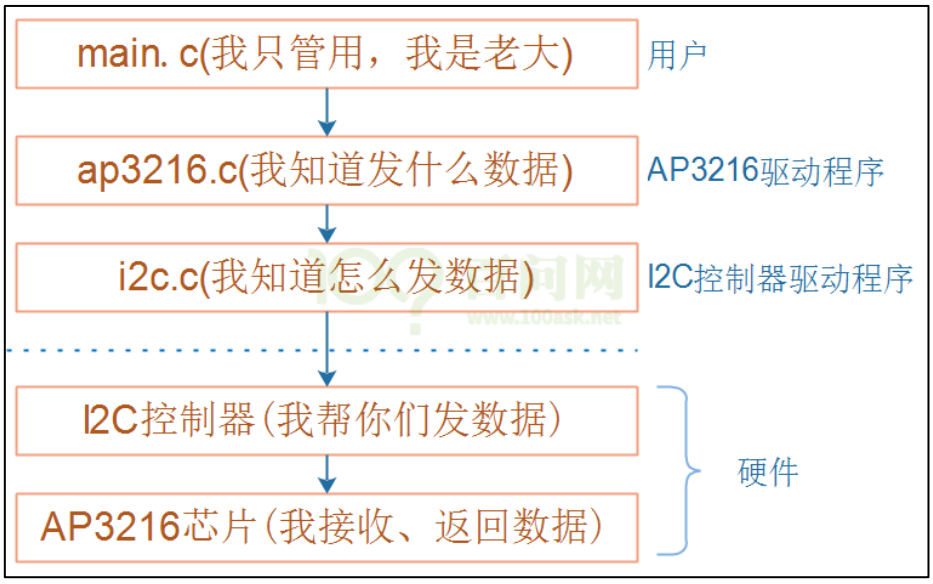
b) 第二次是读操作，发出设备地址，返回数据。

各个寄存器的值有什么含义，在后面写程序时两解释；你也可以看芯片手册。

## 14.4 程序框架

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/16_I2C 编程/001_example_i2c_ap3216c_led_show”目录下。

所涉及的程序有 3 个：main.c、ap3216.c、i2c.c，它们的作用如下图所示：




其中 i2c.c 最复杂，跟 IMX6ULL 密切相关。

## 14.5 I2C 控制器编程

对 I2C 的操作分为两层：I2C 控制器、I2C 设备。我们要读写 AP3216C 等 I2C 设备时，是通过 I2C 控制器发出 I2C 信号，所以：

① **要有 I2C 控制器的代码：**

它实现 I2C 控制器的初始化、I2C 数据的读、写。

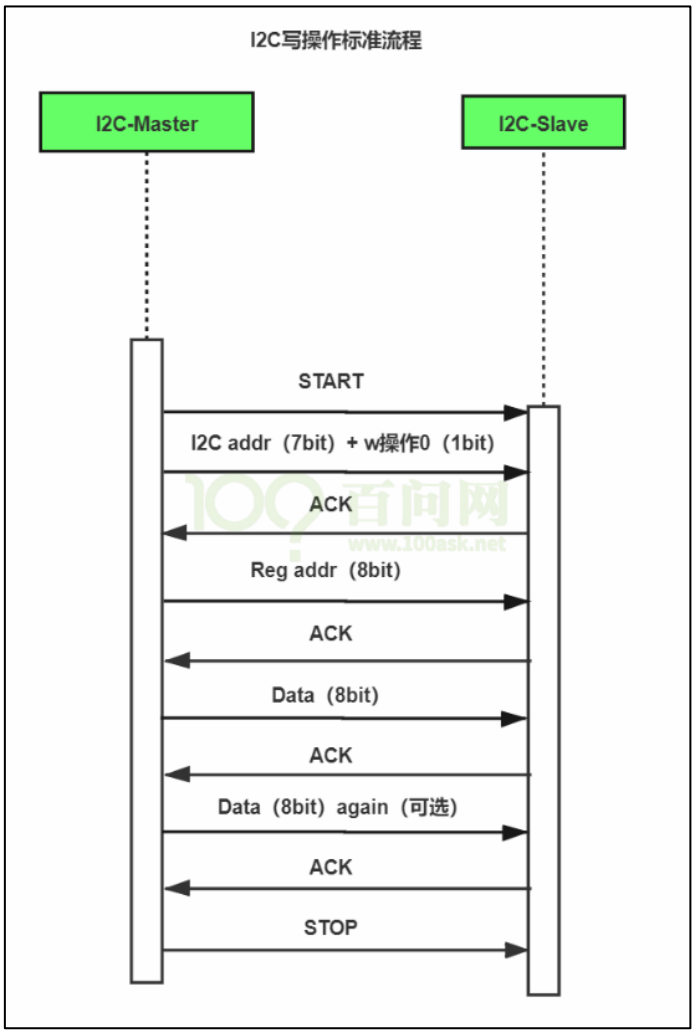
② **I2C 设备相关的代码，比如 AP3216C：**

不同的 I2C 设备，它的数据含义不同。比如有 AP3216C、AT24C02 两个芯片，虽然它们都可以通过底层的 I2C 控制器代码来访问，但是发出的数据个数不同、含义不同，应该有对应的代码：ap3216.c、at24c02.c。

本节讲解 I2C 控制器的程序，代码在“16_I2C 编程\001_example_i2c_ap3216c_led_show\i2c.c”源文件中。

### 14.5.1 I2C 读写标准流程

➢ **写寄存器的标准流程如下图：**




a) Master 发起 START

b) Master 发送 I2C addr（7bit）和 w 操作 0（1bit），等待 ACK

c) Slave 发送 ACK

d) Master 发送 reg addr（8bit），等待 ACK

e) Slave 发送 ACK

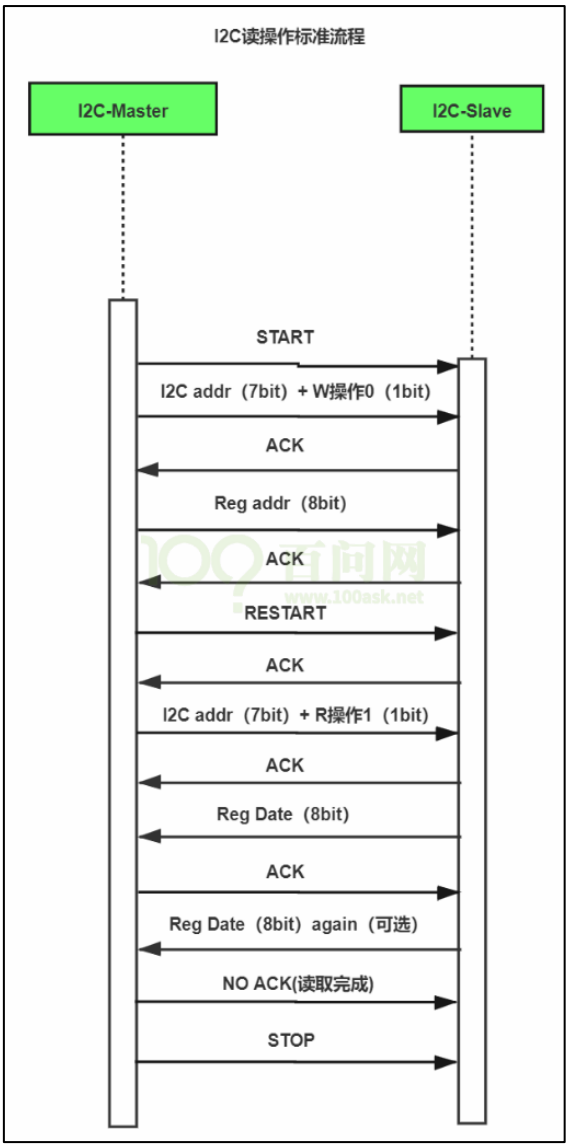
f) Master 发送 data（8bit），即要写入寄存器中的数据，等待 ACK

g) Slave 发送 ACK

h) 第 6 步和第 7 步可以重复多次，即顺序写多个寄存器

i) Master 发起 STOP

- 读寄存器的标准流程如下图：




a) Master 发送 I2C addr（7bit）和 w 操作 1（1bit），等待 ACK

b) Slave 发送 ACK

c) Master 发送 reg addr（8bit），等待 ACK

d) Slave 发送 ACK

e) Master 发起 RESTART

f) Master 发送 I2C addr（7bit）和 r 操作 1（1bit），等待 ACK

g) Slave 发送 ACK

h) Slave 发送 data（8bit），即寄存器里的值

i) Master 发送 ACK

j) 第 8 步和第 9 步可以重复多次，即顺序读多个寄存器

k) Master 发送 NO ACK 表示读取完成，从机也不用发送 ACK

l) Master 发送 STOP

## 14.5.2 重要的结构体

IMX6ULL 中有 4 个 I2C 控制器，它们的基地址各有不同。100ASK_IMX6ULL 开发板上 AP3216C 接到了 I2C1，所以先确定 I2C1 的基地址(注意：I2C 寄存器是 16 位的)：

```
30 #define I2C1_BASE_ADDR                                  (0x21A0000u)
31 /*  I2C1 Base address */
32 #define I2C1                                        ((I2C_REGISTERS *)I2C1_BASE_ADDR)
33 #define I2C2_BASE_ADDR                                  (0x21A4000u)
34 /*  I2C1 Base address */
35 #define I2C2                                        ((I2C_REGISTERS *)I2C2_BASE_ADDR)
```

然后定义如下结构体，以后使用时，定义一个 I2C_REGISTERS 结构体指针，指向 I2C1 的基地址即可：

```
39 typedef struct tagRegisters{
40   volatile uint16_t IADR;             /*I2C Address Register, offset: 0x0 */
41   uint8_t ReservedIADR[2];
42   volatile uint16_t IFDR;             /*I2C Frequency Divider Register, offset:
0x4 */
43    uint8_t ReservedIFDR[2];
44   volatile uint16_t I2CR;             /*I2C Control Register, offset: 0x8 */
45    uint8_t ReservedI2CR[2];
46   volatile uint16_t I2SR;             /*I2C Status Register, offset: 0xC */
47    uint8_t ReservedI2SR[2];
48   volatile uint16_t I2DR;            /*I2C Data I/O Register, offset: 0x10 */
49 } I2C_REGISTERS;
I2C 对外操作可以分为读、写两类，用一个 enum 变量区分一下：
50 typedef enum enI2C_OPCODE
51 {
52    I2C_WRITE = 0,           /* 主机向从机写数据 */
53    I2C_READ  = 1,             /* 主机从从机读数据 */
54    I2C_DONOTHING_BULL
55 } I2C_OPCODE;
```

最后，我们用一个结构体来描述一个 I2C 传输：哪个设备、哪个寄存器、是读还是写、读多少、写多少、数据保存在哪里，等等信息，结构体如下：

```
60 typedef struct tagI2cTransfer
61 {
62    uint8_t  ucSlaveAddress;         /* 7 位从机地址 */
63    uint32_t ulOpcode  ;            /* 操作码*/
64    uint32_t ulSubAddress;          /* 目标寄存器地址 */
65    uint8_t  ulSubAddressLen;        /* 寄存器地址长度 */
66    volatile uint32_t ulLenth;          /* 数据长度 */
67    uint8_t *volatile pbuf;          /* 数据*/
```

```
68 } I2C_TRANSFER;
```

有了这些寄存器的基本信息、操作方式、传输结构，下面来实现具体的函数和功能。

## 14.5.3 i2c_init

i2c_init 用来初始化 I2C 控制器，需要传入某个 I2C 控制器的结构体指针(就是寄存器基地址)。函数原型在 i2c.h 中：

```
75 /*************************************************************************
76  * 函数名称： i2c_init
77  * 功能描述： 初始化 i2c
78  * 输入参数： 1.I2C 的基地址
79  * 输出参数： 无
80  * 返 回 值： 无
81  * 修改日期        版本号     修改人       修改内容
82  * ----------------------------------------------
83  * 2020/02/22            V1.0          小火山           创建
84  *************************************************************************/
85 void i2c_init(I2C_REGISTERS *I2C_BASE);
```

我们不使用中断，初始化代码非常简单，只需要设置分频系数。在设置之前先禁止 I2C，设置好后再使能。代码在 i2c.c 中，如下：

```
03 void i2c_init(I2C_REGISTERS *I2C_BASE)
04 {
05     /*I2C_I2CR 是控制寄存器,
06      * 可以: 使能 I2C,使能中断, 选择主从模式.
07      */
08
09     /* 配置 I2C 控制器步骤: 关闭 I2C,配置,打开 I2C */
10
11     /* 设置 SCL 时钟为 100K
12      * I2C 的时钟源来源于 IPG_CLK_ROOT=49.5Mhz
13       *      PLL2 = 528 MHz
14       *      PLL2_PFD2 = 528 *18 /24 = 396 MHz
15       *      IPG_CLK_ROOT = (PLL2_PFD2 / ahb_podf )/ ipg_podf = (396 MHz/4)/2 =
49.5Mhz
16       *
17       *      PER_CLK_ROOT = IPG_CLK_ROOT/perclk_podf = 49.5 MHz/1 = 49.5 MHz
18       * 设置 I2C 的波特率为 100K， 因此当分频值=49500000/100000=495
19       * 参考 Table 31-3. I2C_IFDR Register Field Values 表中 0x37 对应的 512 最接近
20       * 即寄存器 IFDR 的 IC 位设置为 0X37
21       */
22      I2C_BASE->I2CR &= ~(1 << 7);
23      I2C_BASE->IFDR = 0x37;
24      I2C_BASE->I2CR |= (1<<7);
25 }
```

I2CR、IFDR 寄存器的介绍可以看前面的章节。

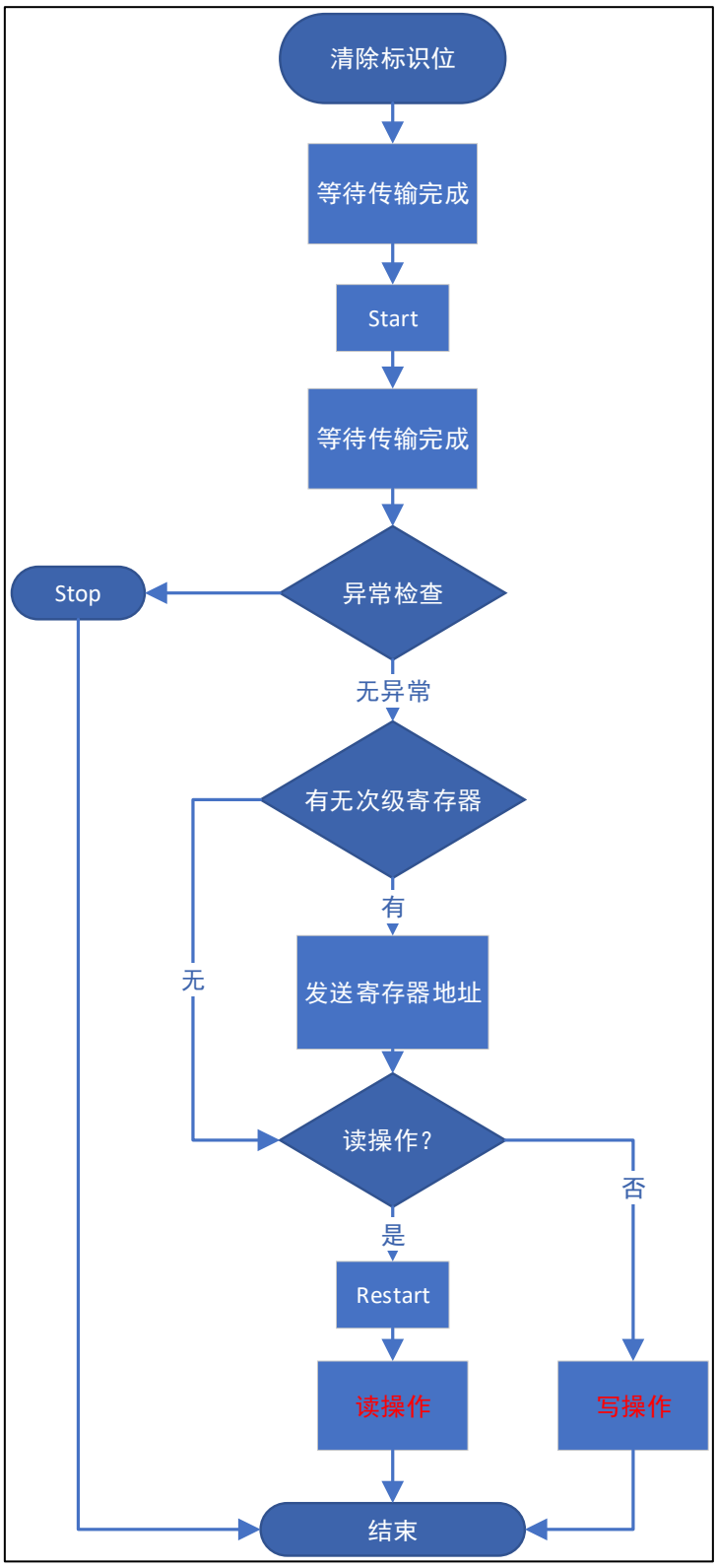
## 14.5.4 i2c_transfer

函数原型在 i2c.h 中，如下：

```
162 /*************************************************************************
163  * 函数名称： i2c_transfer
164  * 功能描述： i2c 传输，包括读写功能
165  * 输入参数： 1.I2C 的基地址 2.传输结构体
166  * 输出参数： 无
167  * 返 回 值： 无
```

```
168  * 修改日期        版本号      修改人              修改内容
169  * ---------------------------------------------
170  * 2020/02/22          V1.0          小火山           创建
171  ***************************************************************************/
172 uint8_t i2c_transfer(I2C_REGISTERS *I2C_BASE, I2C_TRANSFER *transfer);
```

第 1 参数表示用哪个 I2C 控制器，第 2 个参数用来描述一个传输。下面是函数流程图：



下面分阶段介绍代码，在 i2c.c 中。

代码的第 187 行用来等待当前 I2C 传输结束，也许根本没人正在使用 I2C，为了保险还是等一下吧：

```
177 uint8_t i2c_transfer(I2C_REGISTERS *I2C_BASE, I2C_TRANSFER *transfer)
178 {
179     uint32_t ulRet = 0;
180     uint32_t ulOpcode = transfer->ulOpcode;
181 
182     /*开始前准备工作，清除标志位
183      *bit-4 IAL 仲裁位，bit-1 IIF 中断标志位
184      */
185     I2C_BASE->I2SR &= ~((1 << 1) | (1 << 4));
186     /* 等待传输完成 */
187     while(!((I2C_BASE->I2SR >> 7) & 0X1)){};
```

对于读操作，你首先需要发出设备地址，所以第 1 次肯定是写操作。第 182～195 行就是做这样的判断：

```
189     /* 如果要读某个寄存区,寄存器地址要先"写"给从设备
190      * 所以方向要"先写","后读"
191      */
192     if ((transfer->ulSubAddressLen > 0) && (transfer->ulOpcode == I2C_READ))
193     {
194         ulOpcode = I2C_WRITE;
195     }
```

然后调用 i2c_start 发起 I2C 传输，它会发出 START 信号，接着发出设备地址，在第 8 个数据时钟周期根据 ulOpcode 设置 SDA 的值(写操作，SDA 为 0；读操作，SDA 为 1)：

```
196     ulRet = i2c_start(I2C_BASE, transfer->ucSlaveAddress, ulOpcode);
197 
198     if (ulRet)
199     {
200             return ulRet;
201     }
202 
203     /* 等待传输完成: 中断状态为会被置 1 */
204     while(!(I2C_BASE->I2SR & (1 << 1))){};
205 
206     /* 检查是否出错 */
207     ulRet = i2c_check(I2C_BASE, I2C_BASE->I2SR);
208 
209     if (ulRet)
210     {
211         i2c_stop(I2C_BASE);                         /* 发送停止信号 */
212             return ulRet;
213     }
```

要访问某个 I2C 设备时，一般要发出设备地址、寄存器地址。

本程序中，寄存器地址保存在 ulSubAddress 中，长度用 ulSubAddressLen 来表示。下列代码是用来发送寄存器地址的：

```
215     /*如果 ulSubAddressLen 不为 0，表示要发送寄存器地址*/
216     if (transfer->ulSubAddressLen)
217     {
218             do
219             {
220                     /* 清除中断状态位 */
221         I2C_BASE->I2SR &= ~(1 << 1);
```

```
222
223         /* 调整长度，也许寄存器地址有多个字节，本程序最多支持 4 字节 */
224          transfer->ulSubAddressLen--;
225
226         I2C_BASE->I2DR = ((transfer->ulSubAddress) >> (8 *
transfer->ulSubAddressLen));
227
228          while(!(I2C_BASE->I2SR & (1 << 1)));    /* 等待传输完成: 中断状态位被置 1
*/
229
230            /* 检查是否出错 */
231            ulRet = i2c_check(I2C_BASE, I2C_BASE->I2SR);
232            if(ulRet)
233            {
234                   i2c_stop(I2C_BASE);          /* 出错:发送停止信号 */
235                   return ulRet;
236            }
237            }
238            while ((transfer->ulSubAddressLen > 0) && (ulRet == I2C_OK));
```

第 226 行：往 I2DR 寄存器中写值，就可以通过 I2C 总线发送一个数据。

第 238 行：不断循环，直到发送完寄存器地址。

下面是一些状态清除操作：

```
240            if (I2C_READ == transfer->ulOpcode)
241            {
242            I2C_BASE->I2SR &= ~(1 << 1);                    /* 清除中断状态位 */
243            i2c_restart(I2C_BASE, transfer->ucSlaveAddress, I2C_READ); /* 发送重
复开始信号和从机地址 */
244             while(!(I2C_BASE->I2SR & (1 << 1))){}; /* 等待传输完成: 中断状态位被置
1 */
245
246            /* 检查是否出错 */
247            ulRet = i2c_check(I2C_BASE, I2C_BASE->I2SR);
248
249            if(ulRet)
250            {
251                ulRet = I2C_ADDRNAK;
252                i2c_stop(I2C_BASE);                 /* 出错:发送停止信号 */
253                return ulRet;
254            }
255
256
257            }
258
259    }
```

终于到数据传输部分了，可能是发送数据，也可能是读数据。

下面是发送数据的代码，调用 i2c_write 函数：

```
260    /* 发送数据 */
261    if ((I2C_WRITE == transfer->ulOpcode) && (transfer->ulLenth > 0))
262    {
263            i2c_write(I2C_BASE, transfer->pbuf, transfer->ulLenth);
264    }

下面是发送数据的代码，调用 i2c_read 函数：
266    /* 读取数据 */
267    if ((I2C_READ == transfer->ulOpcode) && (transfer->ulLenth > 0))
268    {
```

```
269             i2c_read(I2C_BASE, transfer->pbuf, transfer->ulLenth);
270     }
```

程序中经常用下面的代码来等待数据传输完毕(传输一个字节):

```
while(!(I2C_BASE->I2SR & (1 << 1))){}; /* 等待传输完成: 中断状态位被置 1 */
```

i2c_transfer 函数中用到了 i2c_start、i2c_check、i2c_write、i2c_read 等函数。下面一一介绍。

### 14.5.5 i2c_start

需要先判断 I2C 是否忙，然后设置发送模式，最后将 slave 地址通过数据寄存器发出去。

代码如下:

```
46 uint8_t i2c_start(I2C_REGISTERS *I2C_BASE, uint8_t ucSlaveAddr, uint32_t
ulOpcode)
47 {
48
49      if(I2C_BASE->I2SR & (1 << 5))                 /* I2C 忙 */
50              return 1;
51
52      /*
53          * 设置控制寄存器 I2CR
54          * bit[5]: 1 主模式(master)
55          * bit[4]: 1 发送(transmit)
56       */
57      I2C_BASE->I2CR |=  (1 << 5) | (1 << 4);
58
59      /*
60          * 设置数据寄存器 I2DR
61          * bit[7:0] : 要发送的数据,
62          * START 信号后第一个数据是从设备地址
63       */
64      I2C_BASE->I2DR = ((uint32_t)ucSlaveAddr << 1) | ((I2C_READ == ulOpcode)? 1 :
0);
65      return 0;
66
67 }
```

### 14.5.6 i2c_check

i2c_check 函数用来检查 I2C 传输是否有错误产生，它的第 2 个参数是读取 I2SR 寄存器得到的状态值。函数里先判断是否发生错误，清除错误，返回错误值。

代码如下:

```
27 uint8_t i2c_check(I2C_REGISTERS *I2C_BASE, uint32_t status)
28 {
29      /* 检查是否发生仲裁丢失错误(arbitration lost) */
30      if(status & (1<<4))
31      {
32              I2C_BASE->I2SR &= ~(1<<4);      /* 清除仲裁丢失错误位          */
33
34              I2C_BASE->I2CR &= ~(1 << 7);    /* 复位 I2C: 先关闭 I2C          */
35              I2C_BASE->I2CR |= (1 << 7);     /* 再打开 I2C                  */
36              return I2C_ARBITRATIONLOST;
37      }
```

```
38      else if(status & (1 << 0))      /* 检查 NAK */
39      {
40              return I2C_NAK;         /* 返回 NAK(无应答) */
41      }
42      return I2C_OK;
43
44 }
```

## 14.5.7 i2c_write

当 I2C 主机发出设备地址、寄存器地址后，调用 i2c_write 函数用来向从设备发出一个或多个数据。

函数的核心在于第 123 行，从缓冲区中得到一个数据，写入 I2DR 寄存器：这就可以启动 I2C 传输了。之后就是调用 i2c_check 来判断结果。

代码如下：

```
116 void i2c_write(I2C_REGISTERS *I2C_BASE, const uint8_t *pbuf, uint32_t len)
117 {
118     /* 等待数据寄存器就绪,可以再次发送数据 */
119     while(!(I2C_BASE->I2SR & (1 << 7)));
120
121     I2C_BASE->I2SR &= ~(1 << 1);      /* 清除 IICIF */
122     I2C_BASE->I2CR |= 1 << 4;            /* 发送数据(transmit) */
123     while(len--)
124     {
125             I2C_BASE->I2DR = *pbuf++;           /* 将 buf 中的数据写入到数据寄存器
I2DR */
126
127         while(!(I2C_BASE->I2SR & (1 << 1)));/*等待传输完成,完成或失败,中断状态位被
置 1*/
128             I2C_BASE->I2SR &= ~(1 << 1);                  /* 清除中断状态位 */
129
130             /* 检查有无错误 */
131             if(i2c_check(I2C_BASE, I2C_BASE->I2SR))
132                     break;
133     }
134
135     I2C_BASE->I2SR &= ~(1 << 1);     /* 清除中断状态位 */
136     i2c_stop(I2C_BASE);             /* 发送停止信号 */
137
138 }
```

## 14.5.8 i2c_read

读函数具体流程如下，值得注意两个点：

① 假读：

这是为了启动读操作，有兴趣的人可以查看官方解释：https://community.nxp.com/thread/378405。代码在第 155 行。

② 只读一个字节时：

需要发送一个 NACK 信号，即将 I2CR bit3 TXAK 置 1，代码在第 153 行。

其他寄存器和操作我们在上面已经都讲解过，全部代码如下：

```
140 void i2c_read(I2C_REGISTERS *I2C_BASE, uint8_t *pbuf, uint32_t len)
141 {
142     volatile uint8_t dummy = 0;
143     dummy++;        /* 防止编译警告 */
144
```

```
145     /* 等待数据寄存器就绪 */
146     while(!(I2C_BASE->I2SR & (1 << 7)));
147
148     I2C_BASE->I2SR &= ~(1 << 1);                    /* 清除 IICIF */
149     I2C_BASE->I2CR &= ~((1 << 4) | (1 << 3));      /* 接收数据: Receive,TXAK */
150
151     /* 如果只接收一个字节数据的话发送 NACK 信号 */
152     if(len == 1)
153         I2C_BASE->I2CR |= (1 << 3);
154
155     dummy = I2C_BASE->I2DR; /* 假读 */
157
158     while(len--)
159     {
160             while(!(I2C_BASE->I2SR & (1 << 1)));    /* 等待传输完成 */
161             I2C_BASE->I2SR &= ~(1 << 1);                    /* 清除标志位 */
162
163             if(len == 0)
164         {
165             i2c_stop(I2C_BASE);                    /* 发送停止信号 */
166         }
167
168         if(len == 1)
169         {
170             I2C_BASE->I2CR |= (1 << 3);
171         }
172             *pbuf++ = I2C_BASE->I2DR;
173     }
174
175 }
```

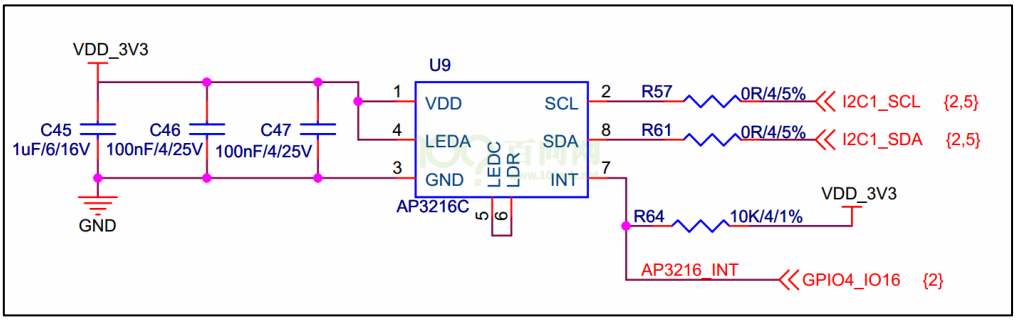
## 14.6 AP3216C 编程

本节讲解 AP3216C 的程序，它是调用 I2C 控制器的代码实现 I2C 的数据传输。

AP3216C 程序在“16_I2C 编程\001_example_i2c_ap3216c_led_show\ap3216c.c”文件中。

### 14.6.1 AP3216C 初始化 IO

参考 100ASK_IMX6ULL 原理图如下：




AP3216C 接口简单，仅将 PIN2 和 PIN8 分别连接到 I2C1_SCL 和 I2C1_SDA。需要将这 2 个引脚设置为 I2C 功能，代码如下：

```
08 static void ap3216c_init_io()
09 {
```

```
10      /*I2C1 复用 UART4  SCL-TXD SDA-RXD*/
11      IOMUXC_SetPinMux(IOMUXC_UART4_TX_DATA_I2C1_SCL, 1);
12      IOMUXC_SetPinMux(IOMUXC_UART4_RX_DATA_I2C1_SDA, 1);
13      IOMUXC_SetPinConfig(IOMUXC_UART4_TX_DATA_I2C1_SCL, 0x70B0);
14      IOMUXC_SetPinConfig(IOMUXC_UART4_RX_DATA_I2C1_SDA, 0X70B0);
15 }
```

## 14.6.2 初始化 AP3216C

通过 AP3216C 的系统寄存器表格可知，0X00 这个寄存器是模式控制寄存器，用来设置 AP3216C 的工作模式。

初始化流程如下：

① 复位（设置 0X00 寄存器为 0X04）；

② 设置工作模式（如 0X03,开启 ALS+PS+IR,其他模式请参考 AP3216C 系统寄存器表格）；

③ 设置中断(我们没用中断)。

代码如下：

```
38      i2c_init(I2C1);
39
40      /*初始化/复位 AP3216C           */
41      i2c_write_one_byte(AP3216C_ADDR, AP3216C_SYSTEMCONFIG, AP3216C_INIT, I2C1);
42      /*手册上写至少等待 10ms*/
43      delay(10000);
44      /*开始转换*/
45      i2c_write_one_byte(AP3216C_ADDR, AP3216C_SYSTEMCONFIG, AP3216C_START_ALL,
I2C1);
37      ret = i2c_write_one_byte(AP3216C_ADDR, AP3216C_SYSTEMCONG, 0X3);/* 开启 ALS、
PS+IR*/
```

i2c_write_one_byte 和 i2c_read_one_byte 就是对 i2c_transfer 的包装，详细可以参考 i2c.c 中具体实现。

### 14.6.3 AP3216C 数据读取

重点介绍一下读数据操作，函数原型如下：

```
50 void ap3216c_read_ir(uint16_t *ir)
71 void ap3216c_read_als(uint16_t *als)
84 void ap3216c_read_ps(uint16_t *ps)
```

它们分别读出红外值、光强、距离值，存入 ir、als、ps 指针。
这 3 个值所在的寄存器，地址相邻，每个值占据 2 个寄存器：

| | | | | |
|---|---|---|---|---|
| 0x0A | IR Data Low | 7 | IR overflow<br>(Read only) | 0: Valid IR and PS data<br>1: Invalid IR and PS data |
| | | 1:0 | (Read only) | IR lower byte of ADC output |
| 0x0B | IR Data High | 7:0 | (Read only) | IR higher byte of ADC output |
| 0x0C | ALS Data Low | 7:0 | (Read only) | ALS lower byte of ADC output |
| 0x0D | ALS Data High | 7:0 | (Read only) | ALS higher byte of ADC output |
| 0x0E | PS Data Low | 7 | Object detect<br>(Read only) | 0: The object leaving<br>1: The object closed |
| | | 6 | IR overflow<br>(Read only) | 0: Valid IR, PS data and object detected<br>1: Invalid IR, PS data and object detected |
| | | 3:0 | (Read only) | PS lower byte of ADC output |
| 0x0F | PS Data High | 7 | Object detect<br>(Read only) | 0: The object leaving<br>1: The object closed |
| | | 6 | IR overflow<br>(Read only) | 0: Valid IR, PS data and object detected<br>1: Invalid IR, PS data and object detected |
| | | 5:0 | (Read only) | PS higher byte of ADC output |

以红外数值为例，可以通过 I2C 函数把 0x0A、0x0B 这 2 个寄存器的值读出来，再合并：

```
50 void ap3216c_read_ir(uint16_t *ir)
51 {
52     uint8_t ucIrLow;
53     uint8_t ucIrHigh;
54     uint8_t i;
55
56      ucIrLow = i2c_read_one_byte(AP3216C_ADDR, AP3216C_IRDATALOW, I2C1);
57      ucIrHigh = i2c_read_one_byte(AP3216C_ADDR, AP3216C_IRDATAHIGH, I2C1);
58
59     if(ucIrLow & 0x80)
60      {
61             *ir = 0;
62             return ;
63      }
64      else
65      {
66             *ir = ((uint16_t)ucIrHigh << 2) | (ucIrLow & 0X03);
67      }
68      return;
69 }
```

## 14.7 AP3216C 上机实验

通过下面的两个例程总结本章对于 AP3216C 通过 I2C 总线将数据发送给 IMX6ULL 并使用不同输出方法显示。

第 1 个程序通过 LED 来显示效果：调整光强和接近距离可以控制 LED 亮灭。代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/16_I2C 编程/001_example_i2c_ap3216c_led_show”目录下。
main 函数代码如下：

```
int  main()
{
    ……
    led_init();    // 或 Uart_Init() ;
    ret = ap3216c_init();
    ……
    while(!ret)
    {
        /*环境光强度(ALS)、接近距离(PS)和红外线强度(IR)*/
        ap3216c_read_data(&ir,&ps,&als);
        /*调整光强和接近距离可以控制 LED 亮灭*/
        if (als>100 || ps <200)
        {
            led_ctl(1);
        }
        else
        {
            led_ctl(0);
        }
    }

    return 0;
}
```

第 2 个程序把 AP3216C 的红外值、光强度值、接近距离通过串口打印出来。
代码在“16_I2C 编程/002_example_i2c_ap3216c_printf_show”目录下，跟第 1 个程序只有 main.c 稍有不同。

## 14.7.1 AP3216C 实验一

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/16_I2C 编程/001_example_i2c_ap3216c_led_show”目录下。
编译、运行程序。AP3216C 在红色启动开关和 SD 卡之间，可以用手电筒照射，或是用手指靠近它，观察实验现象。

## 14.7.2 AP3216C 实验二

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/16_I2C 编程/002_example_i2c_ap3216c_printf_show”目录下。
编译、运行程序，打开串口观察输出信息。

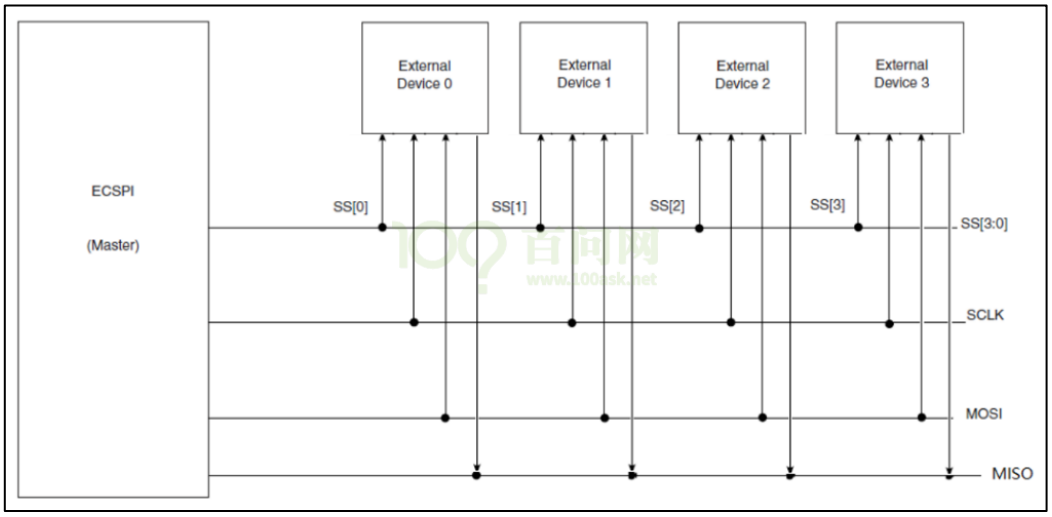
# 第15章 SPI 编程

## 15.1 SPI 接口简介

SPI（Serial Peripheral Interface）接口是全双工的同步串行通讯总线，支持通过多个不同的片选信号来连接多个外设。SPI 接口通常由四根线组成，分别是提供时钟的 SCLK，提供数据输出的 MOSI(Master Out Slave In)，提供数据输入的 MISO(Master In Slave Out)和提供片选信号的 CS。同一时刻只能有一个 SPI 设备处于工作状态，即多个 CS 信号中某时间只能有一个有效。为了适配不同的外设 ，SPI 支持通过寄存器来配置片选信号和时钟信号的极性和相位。（imx6ull 支持 ecspi，即增强配置型 spi，这里为了与其他兼容，统一用 spi 来称呼）。

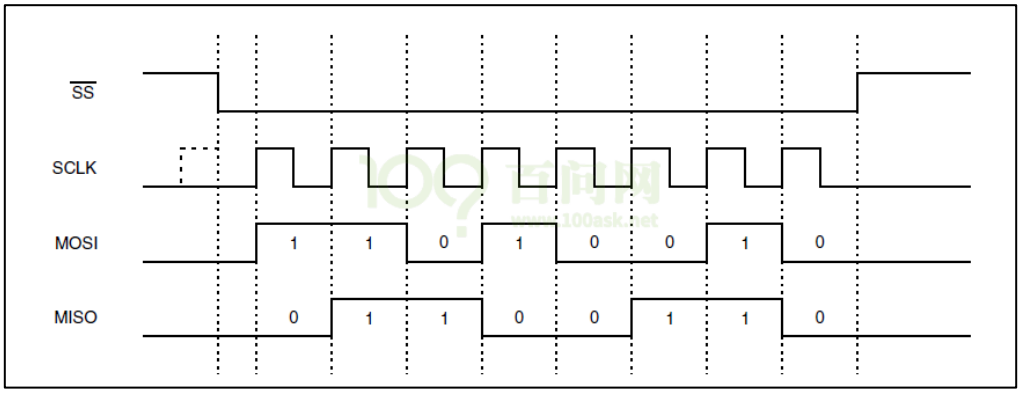
### 15.1.1 SPI 硬件连接

SPI 支持 slave 和 master 两种模式，作为 APU 来说，多数情况下是作为 master 来使用的。在 master 模式下，通过不同的片选引脚 ss[n](n=0,1,2,3)来连接多个不同的设备。下图为 MASTER 模式下的 SPI 单线通讯模式框图：




### 15.1.2 SPI 通讯数据格式

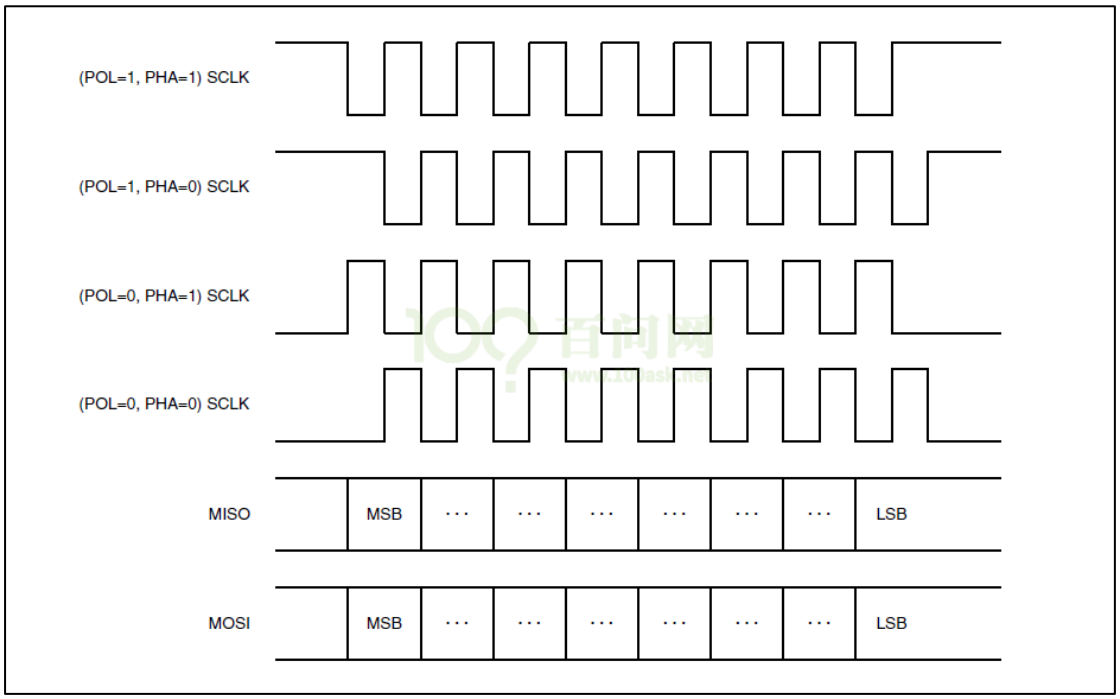
在 master 模式下，ss、sclk 和 mosi 作为信号输出接口，MISO 作为信号输入接口。通过 SS 片选信号使能外部 SPI 设备，SCLK 同步数据传输。MOSI 和 MISO 信号在 SCLK 上升沿变化，在下降沿锁存数据。SPI 的具体通讯格式如下图所示（默认高位在前，低位在后），输出数据为 0xD2，输入数据为 0x66。



SPI 支持不同的 SPI 时钟和 CS 片选相位和极性设置，通过设置 POL 和 PHA 值的不同来设置相位和极性。POL:表示 SPICLK 的初始电平，0 为电平，1 为高电平 ；CHA:表示相位，即第一个还是第二个时钟沿采样数据，0 为第一个时钟沿，1 为第二个时钟沿。具体如下表所示：

| POL | PHA | 模式 | 含义 |
|---|---|---|---|
| 0 | 0 | 0 | 初始电平为低电平，在第一个时钟沿采样数据 |
| 0 | 1 | 1 | 初始电平为低电平，在第二个时钟沿采样数据 |
| 1 | 0 | 2 | 初始电平为高电平，在第一个时钟沿采样数据 |
| 1 | 1 | 3 | 初始电平为高电平，在第二个时钟沿采样数据 |

实际时钟和相位关系如下图所示，我们常用的是模式 0 和模式 3，因为它们都是在上升沿采样数据，不用去在乎时钟的初始电平是什么，只要在上升沿采集数据就行。POL 和 PHA 怎么选择？通常去参考外接的模块的芯片手册。




## 15.2 IMX6ULL 的 SPI 控制器操作与寄存器介绍

参考资料：网盘开发板配套资料“06_Datasheet（数据手册）/Core_board/CPU/IMX6ULLRM.pdf ”：《 Chapter 20: Enhanced Configurable SPI (ECSPI)》。

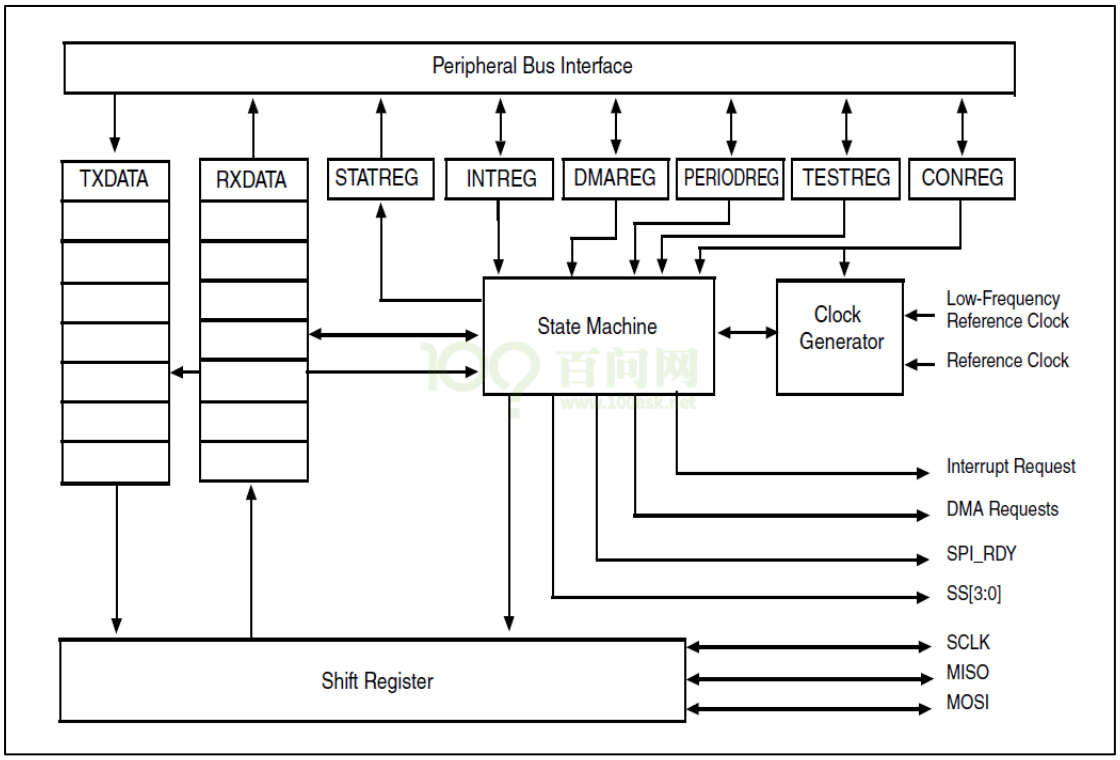
### 15.2.1 SPI 控制器介绍

IMX6ULL 的 SPI 控制器 ECSPI(Enhanced Configurable Serial Peripheral Interface) 为全双工同步四线串行通讯模块，有 4 路独立的控制器。主要特性如下：

- 全双工同步串行接口
- 可配置为 MASTER 或 SLAVE 模式

- 支持多达四个片选信号来连接不同的外设，使用 GPIO 作为片选的话数量不限
- 传输数据长度不限
- 包含一个 64X32 的接收 FIFO 和 64X32 的发送 FIFO
- 片选信号和时钟信号的极性及相位可配置
- 支持 DMA（Direct Memory Acess）
- 最大运行频率依赖于参考时钟频率

SPI 控制器内部包含位移寄存器、接收和发送 FIFO、控制寄存器等相关寄存器。SPI 控制器时钟来源于外部，有低频和参考时钟源两个来源。通过时钟发生器提供给状态机进行 SPI 控制器的状态控制，控制器内部框图如下图所示。

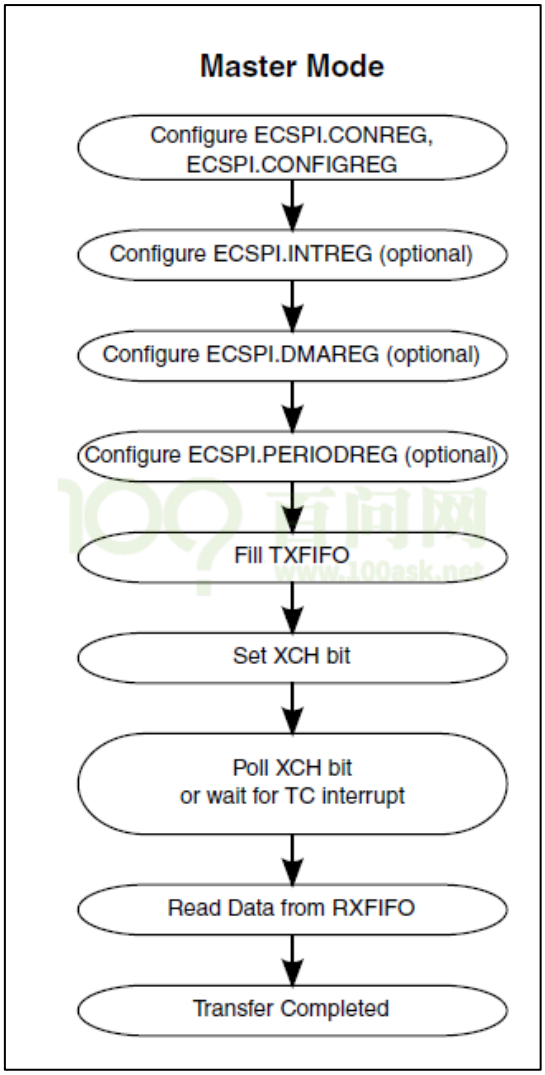



## 15.2.2 SPI 控制器初始化流程

SPI 控制器在使用时，需要首先进行初始化。具体的初始化流程如下：

① 清除 CONFREG 寄存器的 EN 位来复位模块；
② 使能 SPI 时钟，具体在 CCM 模块中进行设置；
③ 配置控制寄存器，并使能 ECSPI_CONFREG 寄存器的 EN 位来让模块工作；
④ 配置 SPI 引脚，具体在 IOMUX 中设置；
⑤ 根据外部 SPI 设备规格信息来配置 SPI 寄存器。

在 MASTER 模式下，对 SPI 的操作可以参考下图所示的流程进行设置



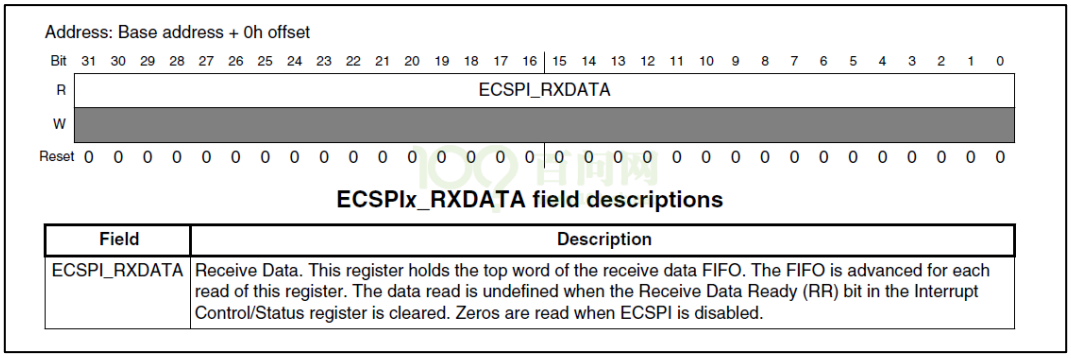

## 15.2.3 SPI 控制器寄存器介绍

IMX6ULL 有 4 路 SPI 控制器，掌握其中一路 SPI 控制器即可。其他几路 SPI 控制器都类似，只需要修改寄存器基址和 IO 复用管脚即可。这里以 SPI3 为例进行说明,SPI3 的相关寄存器地址如下图所示，其中红框内的寄存器为我们本次实验使用到的寄存器。

| 201_0000 | Receive Data Register (ECSPI3_RXDATA) | 32 | R | 0000_0000h | 20.7.1/806 |
|---|---|---|---|---|---|
| 201_0004 | Transmit Data Register (ECSPI3_TXDATA) | 32 | W | 0000_0000h | 20.7.2/807 |
| 201_0008 | Control Register (ECSPI3_CONREG) | 32 | R/W | 0000_0000h | 20.7.3/807 |
| 201_000C | Config Register (ECSPI3_CONFIGREG) | 32 | R/W | 0000_0000h | 20.7.4/810 |
| 201_0010 | Interrupt Control Register (ECSPI3_INTREG) | 32 | R/W | 0000_0000h | 20.7.5/812 |
| 201_0014 | DMA Control Register (ECSPI3_DMAREG) | 32 | R/W | 0000_0000h | 20.7.6/813 |
| 201_0018 | Status Register (ECSPI3_STATREG) | 32 | R/W | 0000_0003h | 20.7.7/815 |
| 201_001C | Sample Period Control Register (ECSPI3_PERIODREG) | 32 | R/W | 0000_0000h | 20.7.8/816 |
| **Absolute address (hex)** | **Register name** | **Width (in bits)** | **Access** | **Reset value** | **Section/ page** |
| 201_0020 | Test Control Register (ECSPI3_TESTREG) | 32 | R/W | 0000_0000h | 20.7.9/818 |
| 201_0040 | Message Data Register (ECSPI3_MSGDATA) | 32 | W | 0000_0000h | 20.7.10/ 819 |

### ➢ ECSPIx_RXDATA

该寄存器用于保存数据传输中从外部设备接收到的外部数据，是只读寄存器。该寄存器与接收数据 FIFO 的顶端数据相同，只允许四个字节的读操作。当模块被禁止的时候，读取到的是 0。

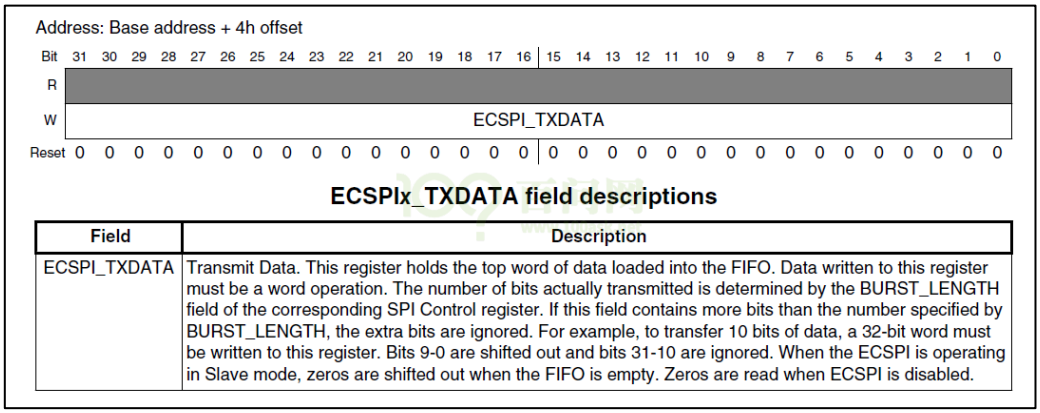



**ECSPIx_RXDATA field descriptions**

| Field | Description |
|---|---|
| ECSPI_RXDATA | Receive Data. This register holds the top word of the receive data FIFO. The FIFO is advanced for each read of this register. The data read is undefined when the Receive Data Ready (RR) bit in the Interrupt Control/Status register is cleared. Zeros are read when ECSPI is disabled. |

### ➢ ECSPIX_TXDATA

该寄存器用于保存数据传输中发送给外部设备的数据，是只写寄存器。该寄存器与发送数据 FIFO 有关，只允许四个字节的写操作。当模块被禁止的时候，写入的数据将被忽略。

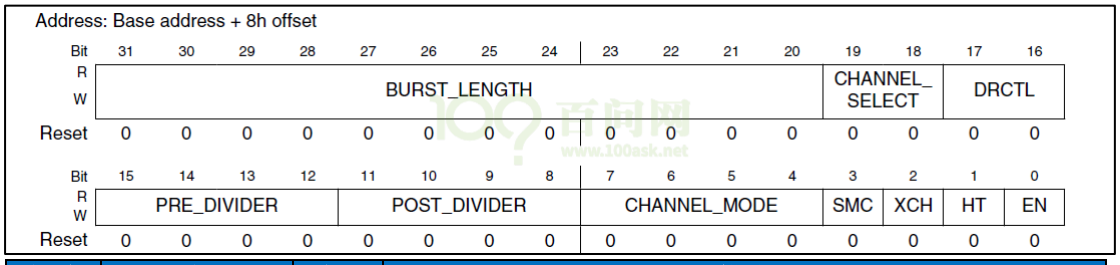



**ECSPIx_TXDATA field descriptions**

| Field | Description |
|---|---|
| ECSPI_TXDATA | Transmit Data. This register holds the top word of data loaded into the FIFO. Data written to this register must be a word operation. The number of bits actually transmitted is determined by the BURST_LENGTH field of the corresponding SPI Control register. If this field contains more bits than the number specified by BURST_LENGTH, the extra bits are ignored. For example, to transfer 10 bits of data, a 32-bit word must be written to this register. Bits 9-0 are shifted out and bits 31-10 are ignored. When the ECSPI is operating in Slave mode, zeros are shifted out when the FIFO is empty. Zeros are read when ECSPI is disabled. |

### ➢ ECSPIX_CONREG

该寄存器用于设置 SPI 控制器的操作模式，包括时钟频率，相关控制方式和数据传输长度等。




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31:20] | BURST_LENGTH | R/W | Burst Length，突发传输长度。SPI 移位寄存器中保存有要传输的数据，它是 32 位的。我们只想传输一个字节即 8 位时，怎么办？要传输 n 位数据时，可以设置此位域为(n-1)，<br>0x000：传输 32 位里的 1 位(LSB，最低位)；<br>0x001：传输 32 位里的 2 位(LSB)；<br>..... |

| | | | |
|---|---|---|---|
| | | | 0x01F：传输 32 位；<br>……<br>0xFFF：传输 4096 位，即 512 字节，即 128 个 word |
| [19:18] | CHANNEL_SELECT | R/W | SPI 控制器中有 4 个片选信号：SS0、SS1、SS2、SS3，可以通过此位域让使用某个片选信号，让它输出 0：<br>00：SS0；<br>01：SS1；<br>10：SS2；<br>11：SS3 |
| [17:16] | DRCTL | R/W | SPI Data Read Control，是否使用 SPI_RDY 信号，<br>00：不用；<br>01：在 SPI_RDY 的下降沿，开始 SPI 传输(边沿触发)；<br>10：在 SPI_RDY 变为低电平时，开始 SPI 传输(电平触发)；<br>11：保留 |
| [15:12] | PRE_DIVIDER | R/W | SPI Pre Divider，SPI 控制器有 2 阶段的提时钟分频,此位域用于控制第 1 阶段:pre-divider，<br>0000：除以 1；<br>0001：除以 2；<br>……<br>1111：除以 16 |
| [11:8] | POST_DIVIDER | R/W | SPI Post Divider，SPI 控制器有 2 阶段的提时钟分频，此位域用于控制第 2 阶段：post-divider，<br>0000：除以(2^0)，即 1；<br>0001：除以(2^1)，即 2；<br>……<br>1110：除以(2^14)；<br>1111：除以(2^15) |
| [7:4] | CHANNEL_MODE | R/W | IMX6ULL 有 4 个 SPI 控制器，每个 SPI 控制器里有 4 个通道(channel)，它们的模式可以分别设置；<br>CHANNEL_MODE[3]对应 SPI 通道 3；<br>CHANNEL_MODE[2]对应 SPI 通道 2；<br>CHANNEL_MODE[1]对应 SPI 通道 1；<br>CHANNEL_MODE[0]对应 SPI 通道 0；<br>取值含义如下：<br>0：Slave 模式；<br>1：Master 模式 |

| | | | |
|---|---|---|---|
| [3] | SMC | R/W | Start Mode Control，此位仅对 master 模式的通道有用，<br>用来决定何时启动 SPI 传输，<br>0：使用 XCH 位(SPI Exchange Bit)来控制，设置 XCH 位时启动传输，<br>这要配合 SS_CTL 位来使用；<br>1：写 TXFIFO 时就启动 SPI 传输 |
| [2] | XCH | R/W | SPI Exchange Bit，此位仅对 master 模式的通道有用，<br>当 SMC 为 0 时，<br>写 XCH 为 1 时，启动 SPI 传输；<br>在传输过程中 XCH 一直保持为 1，<br>当 FIFO 和移位寄存器中的数据都发送完毕时，硬件会把 XCH 清 0 |
| [1] | HT | R/W | Hardware Trigger Enable，硬件触发使能，仅用于 master 模式，<br>0：禁止硬件触发模式；<br>1：使能硬件触发模式 |
| [0] | EN | R/W | SPI Block Enable Control，用来使能 SPI 控制器，<br>在写其他寄存器之前，必须设置此位为 1，<br>0：禁止 SPI 控制器；<br>1：使能 SPI 控制器 |

➢ **ECSPIX_CONFIGREG**

该寄存器用于配置 SPI 中的 4 个通道，包括：操作模式、时钟的相位和极性、片选信号，HT 模式(IMX6ULL 不 HT，即硬件触发模式)。

Address: Base address + Ch offset

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R<br>W | Reserved | | | HT_LENGTH | | | | | SCLK_CTL | | | | DATA_CTL | | | | SS_POL | | | | SS_CTL | | | | SCLK_POL | | | | SCLK_PHA | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

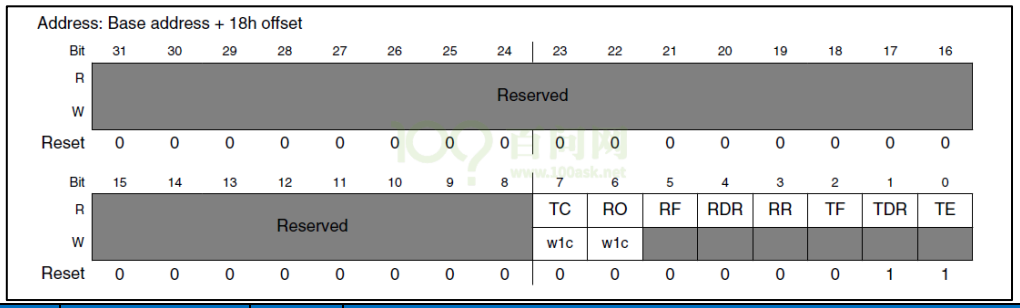
| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [28:24] | HT_LENGTH | R/W | IMX6ULL 用不到 |
| [23:20] | SCLK_CTL | R/W | 使用某个 SPI 通道时，非激活状态下 SCLK 的电平，由此位控制。<br>SCLK_CTL[3]对于 SPI 通道 3；<br>SCLK_CTL[2]对于 SPI 通道 2；<br>SCLK_CTL[1]对于 SPI 通道 1；<br>SCLK_CTL[0]对于 SPI 通道 0；<br>值的含义为，<br>0：低电平；<br>1：高电平 |

| | | | |
|---|---|---|---|
| [19:16] | DATA_CTL | R/W | 使用某个 SPI 通道时，非激活状态下数据线的电平，由此位控制。<br>DATA_CTL[3]对于 SPI 通道 3；<br>DATA_CTL[2]对于 SPI 通道 2；<br>DATA_CTL[1]对于 SPI 通道 1；<br>DATA_CTL[0]对于 SPI 通道 0；<br>值的含义为，<br>0：低电平；<br>1：高电平 |
| [15:12] | SS_POL | R/W | 使用某个 SPI 通道时，对应的片选信号的极性，由此位控制。<br>SS_POL[3]对于 SPI 通道 3；<br>SS_POL[2]对于 SPI 通道 2；<br>SS_POL[1]对于 SPI 通道 1；<br>SS_POL[0]对于 SPI 通道 0；<br>值的含义为，<br>0：低电平有效；<br>1：高电平有效 |
| [11:8] | SS_CTL | R/W | 上一个寄存器提到当 SMC 为 0 时，设置 XCH 为 1 就可以启动 SPI 传输，那么启动多少个 burst 传输呢？<br>0：在 master 模式，只发送一个 burst；<br>1：在 master 模式，只要 TXFIFO 中有数据，SPI 控制器就会不断发送 burst，每个 burst 之间都会有一个“Negate Chip Select”。即 SPI 控制器使能片选，发送一个 burst，取消片选；再次使能片选，发送一个 burst，取消片选，如此循环有到 TXFIFO 为空。<br>SS_CTL[3]对于 SPI 通道 3；<br>SS_CTL[2]对于 SPI 通道 2；<br>SS_CTL[1]对于 SPI 通道 1；<br>SS_CTL[0]对于 SPI 通道 0 |
| [7:4] | SCLK_POL | R/W | 使用某个 SPI 通道时，对应的 SCLK 信号的极性，由此位控制。<br>SCLK_POL[3]对于 SPI 通道 3；<br>SCLK_POL[2]对于 SPI 通道 2；<br>SCLK_POL[1]对于 SPI 通道 1；<br>SCLK_POL[0]对于 SPI 通道 0；<br>值的含义为，<br>0：高电平有效，即空闲时为低电平；<br>1：低电平有效，即空闲时为高电平 |

| | | | |
|---|---|---|---|
| [3:0] | SCLK_PHA | R/W | 使用某个 SPI 通道时，对应的 SCLK 信号的相位，由此位控制。<br>SCLK_PHA[3]对于 SPI 通道 3；<br>SCLK_PHA[2]对于 SPI 通道 2；<br>SCLK_PHA[1]对于 SPI 通道 1；<br>SCLK_PHA[0]对于 SPI 通道 0；<br>值的含义为，<br>0：Phase 0 operation，在第一个时钟沿采样数据；<br>1：Phase 1 operation，在第二个时钟沿采样数据 |

➢ **ECSPIX_STATREG**

状态寄存器，该寄存器只有低 8 位有效，用于指示 SPI 控制器的操作状态，包括发送和接收 FIFO 的状态等信息。如果模块被禁止的话，读取到的值为 0x00000003。

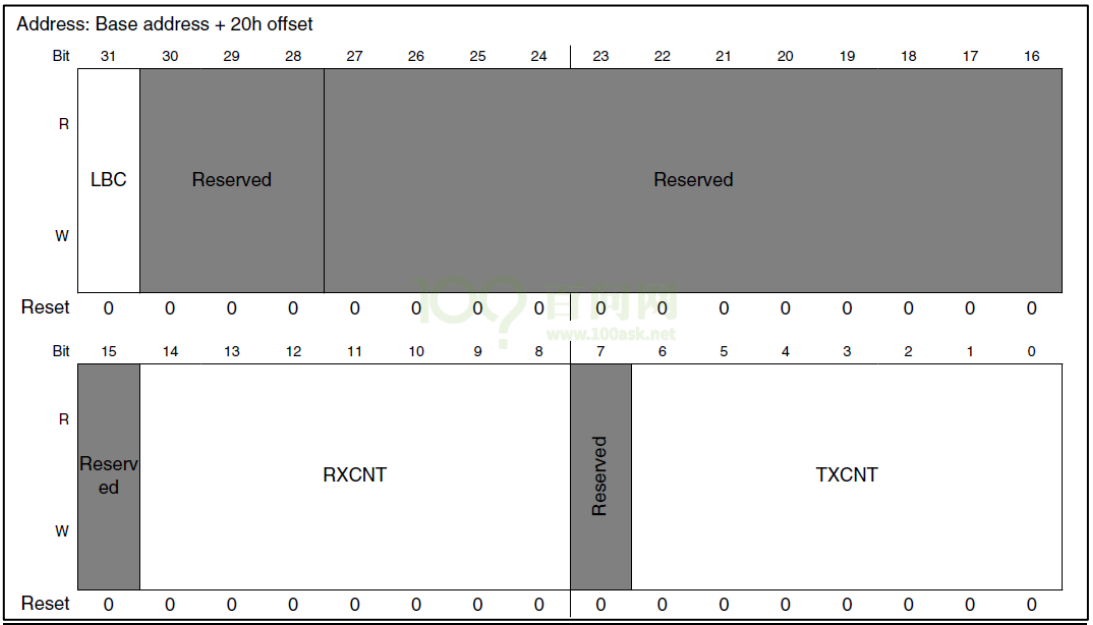



| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [7] | TC | W1C | Transfer Completeed Status bit，传输结束状态位，写 1 清 0，<br>读时，<br>0：正在传输；<br>1：传输结束 |
| [6] | RO | W1C | RXFIFO Overflow，RXFIFO 溢出标记，写 1 清 0，<br>读时，<br>0：RXFIFO 没溢出；<br>1：RXFIFO 溢出了 |
| [5] | RF | R/W | RXFIFO Full，<br>0：RXFIFO 没满；<br>1：RXFIFO 满了 |
| [4] | RDR | R/W | RXFIFO Data Request，RXFIFO 请求数据，<br>当 RXTDE=1 时，<br>0：RXFIFO 中的数据小于或等于 RX_THRESHOLD；<br>1：RXFIFO 中的数据大于 RX_THRESHOLD，<br>或“DMA TAIL DMA condition exists”<br>当 RXTDE=0 时，<br>0：RXFIFO 中的数据量小于或等于 RX_THRESHOLD； |

| | | | |
|---|---|---|---|
| | | | 1：RXFIFO 中的数据量大于 RX_THRESHOLD； |
| [3] | RR | R/W | RXFIFO ready，当 RXFIFO 中数据就绪，<br>0：RXFIFO 中无数据；<br>1：RXFIFO 中有数据 |
| [2] | TF | R/W | TXFIFO Full，TXFOFO 是否满了， |
| [1] | TDR | R/W | TXFIFO Data Request，TXFIFO 请求数据，<br>0：TXFIFO 中的有效数据量大于 TX_THRESHOLD；<br>1：TXFIFO 中的有效数据量小于或等于 TX_THRESHOLD； |
| [0] | TE | R/W | TXFIFO Empty，TXFIFO 空，<br>0：TXFIFO 中有数据；<br>1：TXFIFO 空了，无数据 |

➢ **ECSPIX_TESTREG**

测试寄存器，该寄存器提供了一个测试方式，通过软件在 SPI 控制器内部将接收和发送区域连接起来，这就是回乐(Loopback)。回环时，不会影响到设备的输出，但是外部的输入将会被忽略。



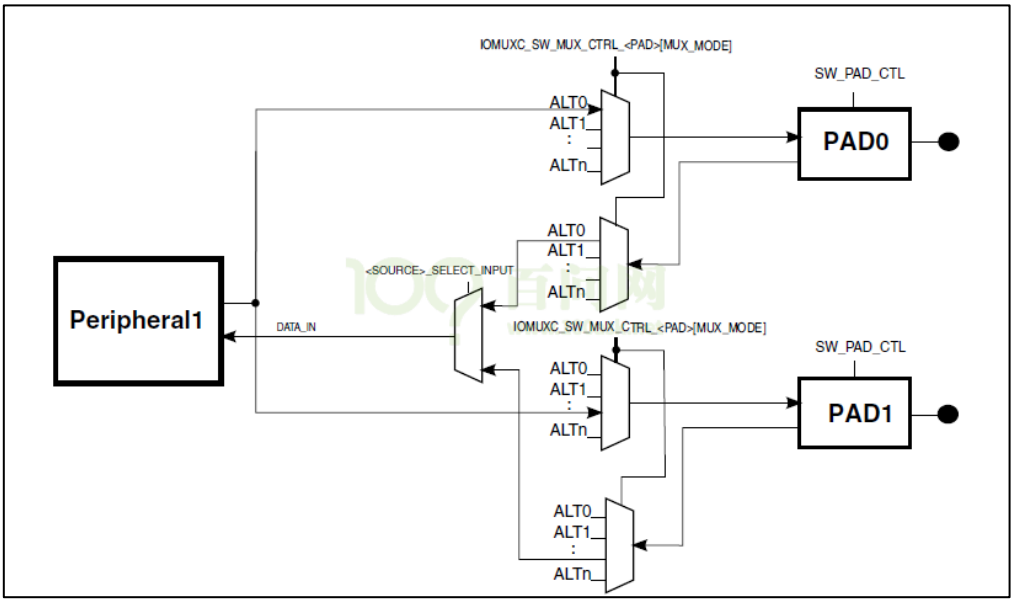

| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [31] | LBC | RW | Loop Back Control，使在 master 模式下使用，<br>0：不回环；<br>1：回环 |

| | | | |
|---|---|---|---|
| [14:8] | RXCNT | RW | RXFIFO Count，RXFIFO 中含有的数据个数(多少个 word) |
| [6:0] | TXCNT | RW | TXFIFO Count，TXFIFO 中含有的数据个数(多少个 word) |

可以通过将 bit31 设置为 1，然后进行数据发送并统计发送的数量，之后读取接收 fifo 的内容和数量进行对比，从而测试 SPI 控制器是否正常。

### 15.2.4 SPI 控制器引脚设置

为了使用 SPI 控制器，我们需要把对应的 IO 引脚配置为 SPI3 功能。PAD 的功能设置框图如下图所示，需要设置引脚工作模式、引脚上下拉和工作速度等。




➢ **初始化 SPI3 引脚**

把引脚配置为 SPI3 功能，涉及到的引脚模式配置的寄存器如下图所示。

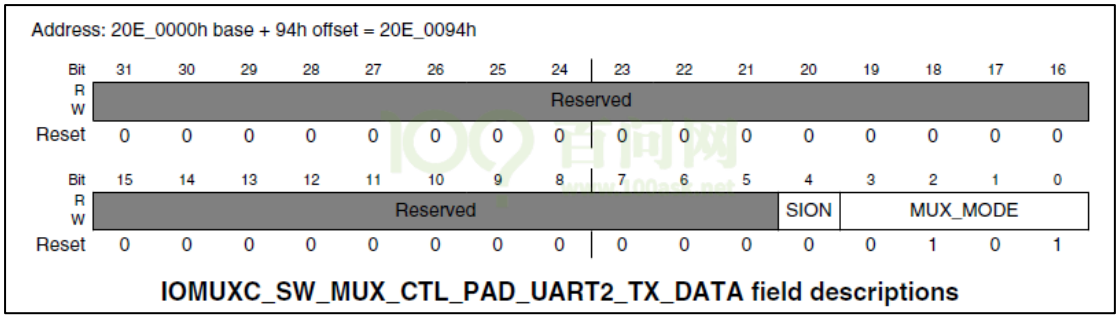
| Ball Name | Signal Name | Mux Mode |
|---|---|---|
| UART2_RTS_B | ecspi3.MISO | Alt 8 |
| UART2_CTS_B | ecspi3.MOSI | Alt 8 |
| UART2_RX_DATA | ecspi3.SCLK | Alt 8 |
| UART2_TX_DATA | ecspi3.SS0 | Alt 8 |

这四组引脚涉及到的具体寄存器如下。

① IOMUXC_SW_MUX_CTL_PAD_UART2_TX_DATA

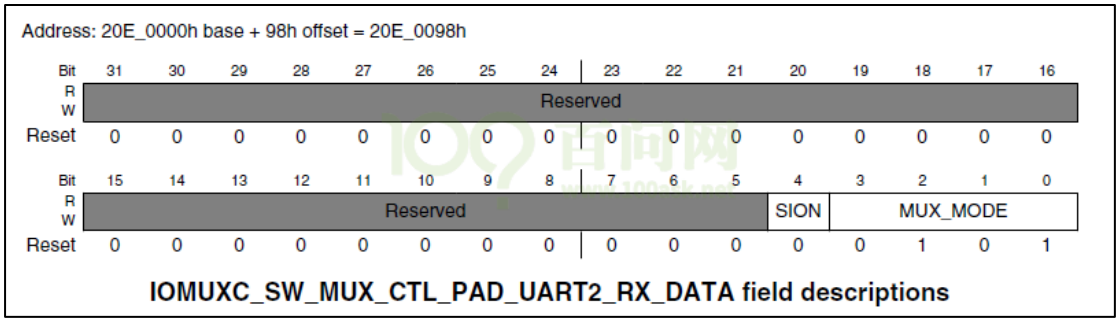
该寄存器用于设置 SPI3 的 SS0 片选引脚，把 MUX_MODE 设置为 8。

如果想自己来控制片选信号，可以将该引脚设置为普通 GPIO，自己来控制它的输出电平。



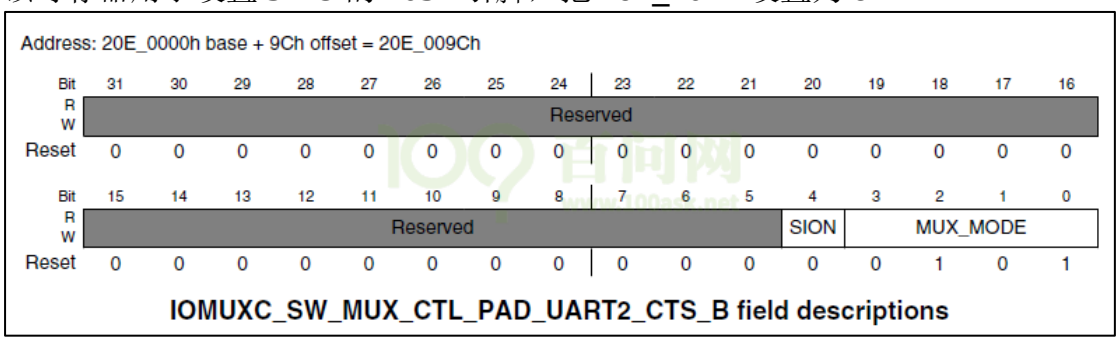

② IOMUXC_SW_MUX_CTL_PAD_UART2_RX_DATA

该寄存器用于设置 SPI3 的 SCLK 引脚，把 MUX_MODE 设置为 8。



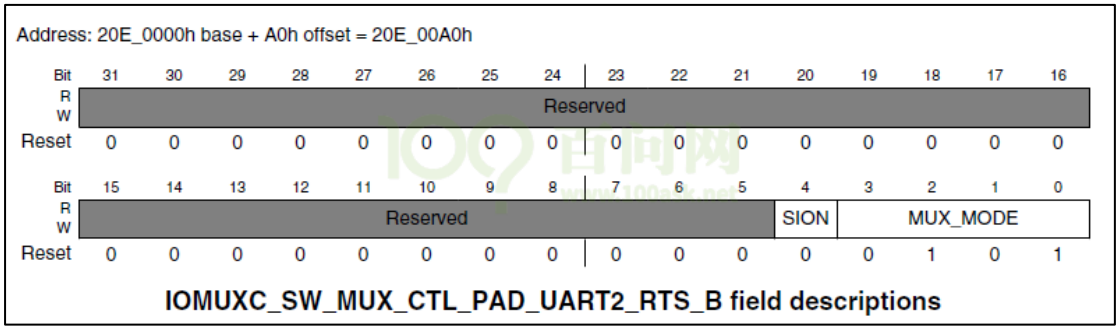

③ IOMUXC_SW_MUX_CTL_PAD_UART2_CTS_B

该寄存器用于设置 SPI3 的 MOSI 引脚，把 MUX_MODE 设置为 8。




④ IOMUXC_SW_MUX_CTL_PAD_UART2_RTS_B

该寄存器用于设置 SPI3 的 MISO 引脚，把 MUX_MODE 设置为 8。




➢ **初始化 SPI3 工作速率**

把所用引脚配置为 SPI3 功能，还需要设置引脚的上下拉阻、工作速度等，涉及到的 SW_CTL_PAD 下图所示。

| | | | | | |
|---|---|---|---|---|---|
| 20E_0320 | SW_PAD_CTL_PAD_UART2_TX_DATA SW PAD Control Register (IOMUXC_SW_PAD_CTL_PAD_UART2_TX_DATA) | 32 | R/W | 0000_10B0h | 32.6.167/ 1815 |
| **Absolute address (hex)** | **Register name** | **Width (in bits)** | **Access** | **Reset value** | **Section/ page** |
| 20E_0324 | SW_PAD_CTL_PAD_UART2_RX_DATA SW PAD Control Register (IOMUXC_SW_PAD_CTL_PAD_UART2_RX_DATA) | 32 | R/W | 0000_10B0h | 32.6.168/ 1817 |
| 20E_0328 | SW_PAD_CTL_PAD_UART2_CTS_B SW PAD Control Register (IOMUXC_SW_PAD_CTL_PAD_UART2_CTS_B) | 32 | R/W | 0000_10B0h | 32.6.169/ 1819 |
| 20E_032C | SW_PAD_CTL_PAD_UART2_RTS_B SW PAD Control Register (IOMUXC_SW_PAD_CTL_PAD_UART2_RTS_B) | 32 | R/W | 0000_10B0h | 32.6.170/ 1821 |
| | SW_PAD_CTL_PAD_UART3_TX_DATA SW PAD Control | | | | |

这四组引脚涉及到的具体寄存器如下。

① IOMUXC_SW_PAD_CTL_PAD_UART2_TX_DATA

该寄存器用于设置引脚 UART2_TX_DATA 上下拉、速度和驱动能力等，具体如下图所示。

Address: 20E_0000h base + 320h offset = 20E_0320h

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | Reserved | | | | | | | | | | | | | | | HYS |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | PUS | | PUE | PKE | ODE | Reserved | | | SPEED | | DSE | | | Reserved | | SRE |
| Reset | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

② IOMUXC_SW_PAD_CTL_PAD_UART2_RX_DATA

该寄存器用于设置引脚 UART2_RX_DATA 上下拉、速度等，具体如下图所示。

Address: 20E_0000h base + 324h offset = 20E_0324h

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | Reserved | | | | | | | | | | | | | | | HYS |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | PUS | | PUE | PKE | ODE | Reserved | | | SPEED | | DSE | | | Reserved | | SRE |
| Reset | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

③ IOMUXC_SW_PAD_CTL_PAD_UART2_CTS_B

该寄存器用于设置引脚 UART2_CTS_B 上下拉、速度等，具体如下图所示。

Address: 20E_0000h base + 328h offset = 20E_0328h

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | Reserved | | | | | | | | | | | | | | | HYS |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | PUS | | PUE | PKE | ODE | Reserved | | | SPEED | | DSE | | | Reserved | | SRE |
| Reset | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

④ IOMUXC_SW_PAD_CTL_PAD_UART2_RTS_B

该寄存器用于设置引脚 UART2_RTS_B 上下拉、速度等，具体如下图所示。

Address: 20E_0000h base + 32Ch offset = 20E_032Ch

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | Reserved | | | | | | | | | | | | | | | HYS |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R W | PUS | | PUE | PKE | ODE | Reserved | | | SPEED | | DSE | | | Reserved | | SRE |
| Reset | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

## 15.3 ICM-20608-G 操作方法

参考资料：网盘开发板配套资料“06_Datasheet（数据手册）/Base_board/100ask_imx6ull 底板_规格书/ICM-20608-G.pdf、icm20608-g-RM-000030-v1.0.pdf”。

### 15.3.1 ICM-20608-G 介绍

ICM-20608-G 采用 LGA16 封装（大小为 3*3*0.75m），6 轴姿态传感器，包括 3 轴加速度和 3 轴角速度，具有如下特性：

① 支持数字输出 X、Y 和 Z 轴角速度传感器（陀螺仪），具有±250、±500、±1000 和±2000 /秒的用户可编程量程，内部集成 16 位 ADC

② 数字输出 X、Y 和 Z 轴加速度位±2g、±4g、±8g 和±16g 的用户可编程满量程，内部集成 16 位 ADC

③ 512 字节的 FIFO 缓冲器，允许应用处理器来读取突然的数据

④ 数字输出温度传感器

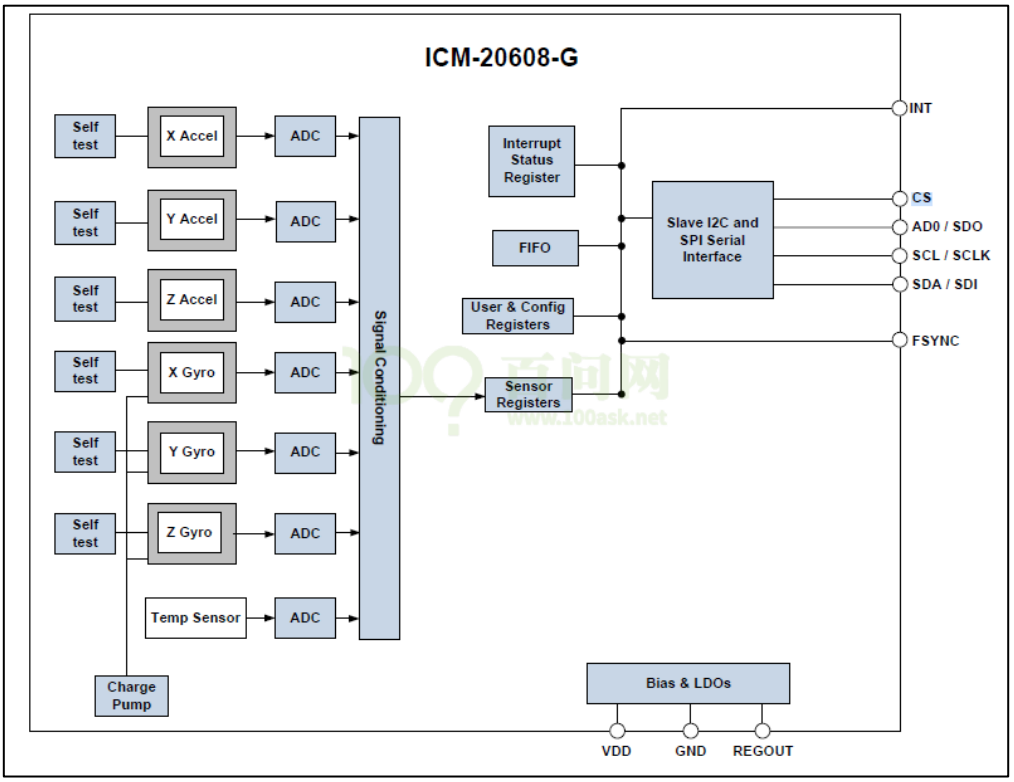
⑤ 支持用户可编程的角速度、加速度和温度传感器滤波设置

⑥ 支持 400kHz 的快速模式 I2C，或者 8MHz 的 SPI 串行接口

⑦ 支持 self-test

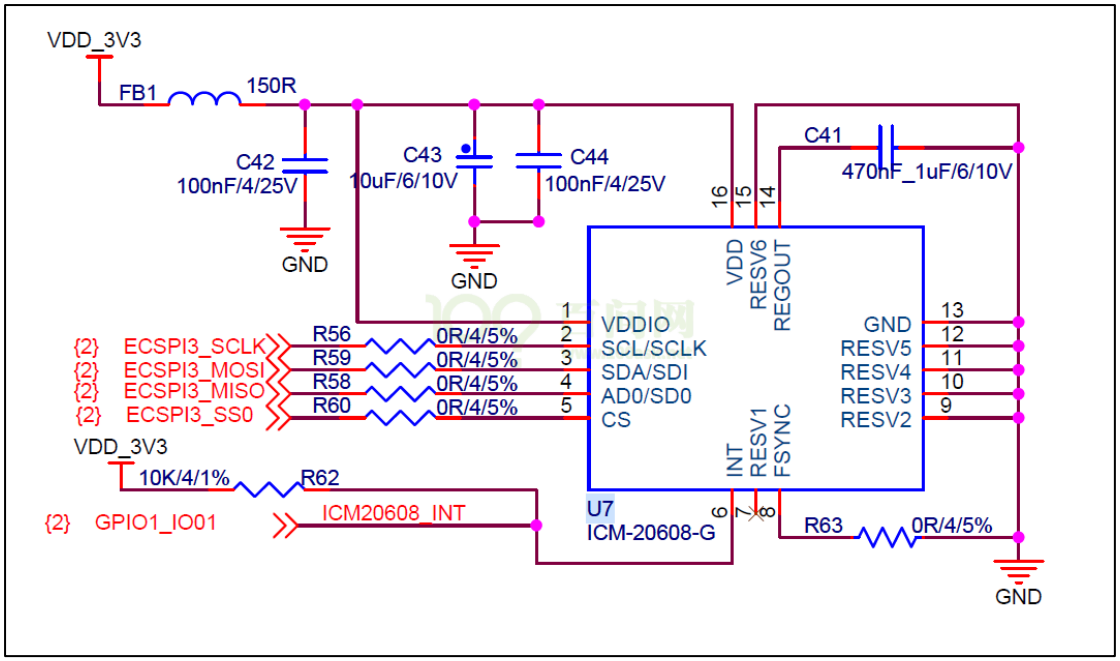
⑧ 支持动作唤醒和低功耗操作

ICM-20608-G 内部框图如下所示：




### 15.3.2 ICM-20608-G 接口

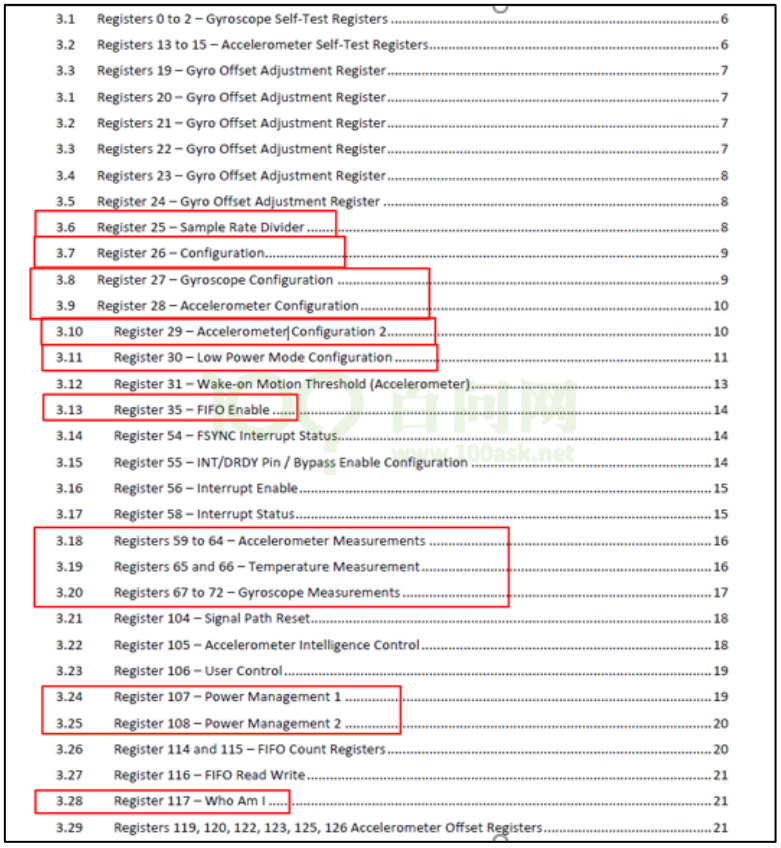
ICM-20608-G 支持 SPI 和 I2C 两种接口，这里选择 SPI 接口进行控制。具体连接方式如下图所示，使用 IMX6ULL 的 SPI3 接口去读写 ICM-206068-G 设备。




## 15.3.3 ICM-20608-G 相关寄存器

ICM-20608-G 内部有多个寄存器，可以通过 SPI 接口来选择不同寄存器地址来实现读取设备 ID、温度、加速度和加速度等多个信息，也可以设置 ICM-20608-G 的相关工作方式。

相关寄存器地址具体如下图所示，其中红框里的是此次实验涉及到的寄存器。




相关寄存器的含义具体如下：

(1) Sample rate divider：该寄存器用于设置采样频率，该值用于产生采样

频率来控制传感器数据输出和 FIFO 采样率。
(2) Configuration：该寄存器用于设置设备的工作模式，包括 FIFO、加速度、温度和加速度输出等。
(3) Gyroscope configuration：该寄存器用于配置加速度传感器，包括量程选择、是否自测等信息。
(4) Accelerometer configuration：该寄存器用于配置加速度传感器，包括量程选择、是否自测等。
(5) Accelerometer configuration2：该寄存器也同时用于配置加速度传感器，包括采样速度等。
(6) Low power mode configuration：该寄存器用于设置 ICM-20608-G 是否处于低功耗模式以及唤醒频率。
(7) Fifo enable：该寄存器用于设置数据存储 FIFO 是否进行使用，为 1 表示使用，为 0 表示禁止 FIFO。
(8) Power management1：该寄存器用于设置 ICM-20608-G 时钟来源、电源模式、以及是否清楚全部寄存器到默认状态等。
(9) Power management2：该寄存器用于设置是否使能相关传感器、包括加速度和角速度传感器。
(10) Who am i：该寄存器存储设备 ID，ICM-20608-G 的默认值为 0xAF。
(11) Accelerometer measurements：该寄存器存储角速度的测量结果，一共 6 个地址，每两个地址为一组 X/Y/Z 轴角速度的高低字节，具体内容如下图所示。
(12) Temperature measurements：该寄存器用于存储温度测量结果，由高低两个字节构成。
(13) Gyroscope measurement：该寄存器存储加速度的测量结果，一共 6 个地址，每两个地址为一组 X/Y/Z 轴加速度的高低字节，具体内容如下图所示。

- 要了解 ICM—20608-G 的接口，请参考：ICM-20608-G.pdf
- 要了解寄存器的内部细节，请参考：ICM-20608-g-RM-000030-v1.0.pdf

### 15.3.4 ICM-20608-G 读写方法

通过 SPI 接口连接 ICM-20608-G，ICM-20608-G 的操作要求如下：
① 数据传输首先传输 MSB 位，最后传输 LSB 位
② 数据在 sclk 上升沿被锁存
③ 数据应当在 sclk 的下降沿被传输
④ 最大支持 8MHz 的时钟频率
⑤ 数据读写操作在 16 个或者更多的时钟周期内完成。第一字节是寄存器地址，接下来的字节为相关数据。
⑥ 支持单字节或者多字节读写操作

ICM-20608-G 内部包含有多个寄存器，我们可以通过 SPI 对不同的地址进行操作读写操作来实现。

要读写 ICM-20608-G 的某个寄存器时，方法如下：

- ① 发送寄存器地址，并标记是要读还是要写：

寄存器地址作为第 1 个字节发送出去，最高位 MSB 表示读写操作(1：读，0：写)。格式如下：

SPI Address format

| MSB | | | | | | | LSB |
|---|---|---|---|---|---|---|---|
| R/W | A6 | A5 | A4 | A3 | A2 | A1 | A0 |

- ② 之后就是寄存器的数据

要写寄存器，就要 MOSI 上驱动数据；要读寄存器时，MISO 数据线上传输的 ICM-20608-G 返回的数据。

数据格式如下：

SPI Data format

| MSB | | | | | | | LSB |
|---|---|---|---|---|---|---|---|
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |

为了使用 ICM-20608-G，我们需要进行初始化。

选择时钟、设置传感器的采样率、加速度和角速度传感器的量程、加速度和角速度的相关滤波器以及是否使用低功耗模式。

当设置完成之后，使能传感器。最后，就可以读取相关传感器数据了。具体的 ICM-20608-G 操作过程请看后面的程序。

## 15.4 SPI 控制器编程

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/18_SPI 编程/001_spi_init”目录下：spi.c。

板卡通过 SPI3 接口连接到 ICM-20608-G，本章针对 SPI3 进行相关编程说明，其他的 SPI 接口可通过修改寄存器地址和初始化相关引脚来实现。

要想实现 SP 控制器的顺利使用，我们首先需要将对应的 SPI 控制器引脚初始化为 SPI 功能引脚，然后设置 SPI 控制器。SPI 控制器的设置包括时钟的配置、传输数据长度、工作模式和电平极性等。

### 15.4.1 SPI 控制器引脚设置

我们使用的是 SPI3 控制器，连接到 ICM-20608-G 的引脚如下图红框处所示。

| Signal | eCSPI1 | eCSPI2 | eCSPI3 | eCSPI4 |
|---|---|---|---|---|
| MISO | CSI_DATA07.alt3 | CSI_DATA03.alt3 | UART2_RTS_B.alt8 | ENET2_TX_CLK.alt3 |
| MOSI | CSI_DATA06.alt3 | CSI_DATA02.alt3 | UART2_CTS_B.alt8 | ENET2_TX_EN.alt3 |
| SCLK | CSI_DATA04.alt3 | CSI_DATA00.alt3 | UART2_RX_DATA.alt8 | ENET2_TX_DATA1.alt3 |
| SS0 | CSI_DATA05.alt3 | CSI_DATA01.alt3 | UART2_TX_DATA.alt8 | ENET2_RX_ER.alt3 |
| SS1 | LCD_DATA05.alt8 | LCD_HSYNC.alt8 | NAND_ALE.alt8 | NAND_DATA01.alt8 |
| SS2 | LCD_DATA06.alt8 | LCD_VSYNC.alt8 | NAND_RE_B.alt8 | NAND_DATA02.alt8 |
| SS3 | LCD_DATA07.alt8 | LCD_RESET.alt8 | NAND_WE_B.alt8 | NAND_DATA03.alt8 |

➢ **设置引脚的工作模式**

我们首先设置引脚的工作模式，然后设置引脚上下拉和驱动能力等

① **设置 SPI3 的 SS0 片选引脚**

针对 ICM-20608-G，它使用 SPI3 的 SS0 作为片选引脚。我们将该引脚设置成 GPIO,通过拉高拉低来进行片选操作。

SPI3 的 SS0 对应 IOMUX_SW_MUX_CTL_PAD_UART2_TX_DATA 寄存器，设置为 ALT5 模式，作为 GPIO 进行使用。

SPI3 的 SS0 引脚用作 GPIO 时，对应 GPIO1_20，需要设置它为输出引脚，以后设置它的输出值，涉及的 GPIO 寄存器如下图所示：

| Absolute address (hex) | Register name | Width (in bits) | Access | Reset value | Section/ page |
|---|---|---|---|---|---|
| 209_C000 | GPIO data register (GPIO1_DR) | 32 | R/W | 0000_0000h | 28.5.1/1358 |
| 209_C004 | GPIO direction register (GPIO1_GDIR) | 32 | R/W | 0000_0000h | 28.5.2/1359 |

② 设置 SPI3 的 SCLK 引脚

要将 IOMUXC_SW_MUX_CTL_PAD_UART2_RX_DATA 设置为 ALT8 模式。

③ 设置 SPI3 的 MOSI 引脚

要将 IOMUXC_SW_MUX_CTL_PAD_UART2_CTS_B 设置为 ALT8 模式。

④ 设置 SPI3 的 MISO 引脚

要将 IOMUXC_SW_MUX_CTL_PAD_UART2_RTS_B 设置为 ALT8 模式。

选择引脚的功能后，还可以设置它的属性：设置引脚上下拉和驱动能力等，我们使用默认值。

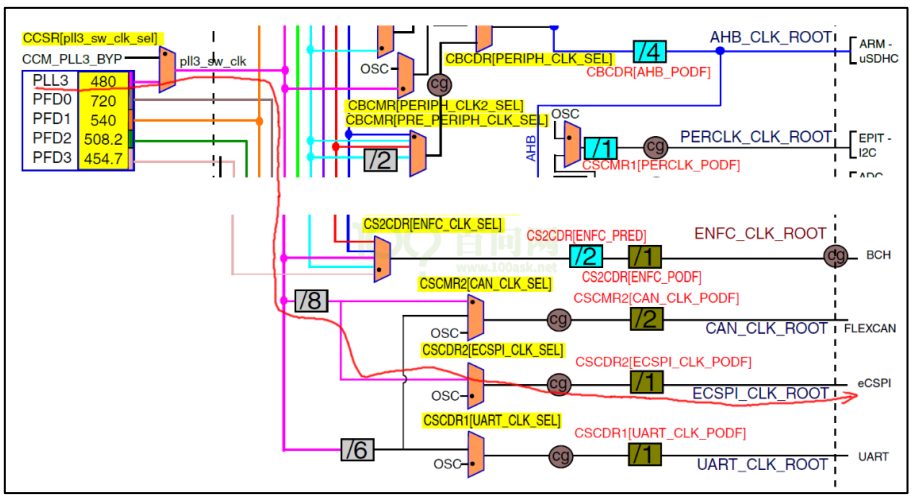
➢ 引脚初始化代码

SPI3 引脚初始化代码如下所示，将 CS 片选引脚设置成 ALT5 模式，其他设置成 ALT8 模式。 这部分的代码在程序文件 spi.c 中第 87 行~第 96 行。

```
87  //引脚初始化
88  iomuxc_sw_set(UART2_TX_BASE,5);//设置为 GPIO 作为片选来进行使用。GPIO1_IO20
89  GPIO1_GDIR  = (volatile unsigned int *)(0x209C000 + 0x4);
90  GPIO1_DR  = (volatile unsigned int *)(0x209C000);
91  *GPIO1_GDIR |= (1<<20);//设置为输出
92  *GPIO1_DR |= (1<<20);
94  iomuxc_sw_set(UART2_RX_BASE,8);
95  iomuxc_sw_set(UART2_RTS_BASE,8);
96  iomuxc_sw_set(UART2_CTS_BASE,8);
```

## 15.4.2 SPI 控制器时钟设置

首先选择 SPI 时钟来源，然后 CONREG 寄存器设置分频，得到 SPI 的时钟。SPI 控制器的时钟来源如下图所示，起始于 PLL3，经过分频之后提供给 SPI 控制器进行使用。PLL3 频率为 480MHz，经过 8 分频之后提供 60MHz 的时钟给 SPI 控制器进行使用。



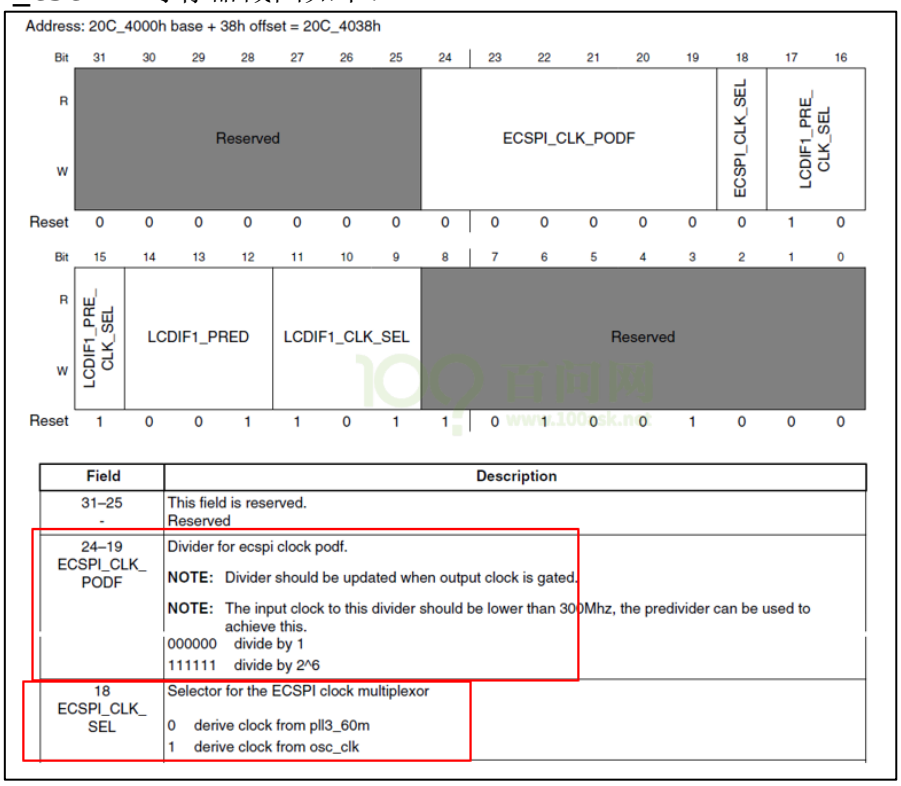
➢ **设置 SPI 时钟**

① 选择 SPI 控制器的时钟来源

从上图可知，SPI 的时钟来源有二：pll3_sw_clk/8 或 OSC。

在 CCM_CSCDR2 中选择 SPI 时钟源，并设置分频系数 ECSPI_CLK_PODF(本程序取值为 0，分频系数为 1)，所以提供给 SPI 控制器的时钟为：PLL3 / 8 / 1 = 480MHz / 8 / 1 = 60 MHz。

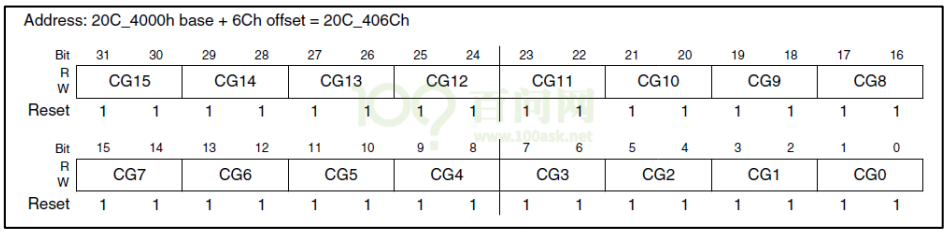
CCM_CSCDR2 寄存器截图如下：




| Field | Description |
|---|---|
| 31–25<br>- | This field is reserved.<br>Reserved |
| 24–19<br>ECSPI_CLK_PODF | Divider for ecspi clock podf.<br>**NOTE:** Divider should be updated when output clock is gated.<br>**NOTE:** The input clock to this divider should be lower than 300Mhz, the predivider can be used to achieve this.<br>000000 divide by 1<br>111111 divide by 2^6 |
| 18<br>ECSPI_CLK_SEL | Selector for the ECSPI clock multiplexor<br>0 derive clock from pll3_60m<br>1 derive clock from osc_clk |

② 使能 SPI 控制器时钟

在选择时钟来源之后，我们需要使能 SPI 控制器时钟，对应的寄存器 CCM_CCGR1 相关设置位如下图红框所示。




| | |
|---|---|
| CG4 | |
| 7–6<br>CG3 | ecspi4 clocks (ecspi4_clk_enable) |
| 5–4<br>CG2 | ecspi3 clocks (ecspi3_clk_enable) |
| 3–2<br>CG1 | ecspi2 clocks (ecspi2_clk_enable) |
| CG0 | ecspi1 clocks (ecspi1_clk_enable) |

CG 的设置参考值如下图所示，我们将 bit5:4 的值设置为 0bb11,在所有模式下 SPI 控制器时钟都进行使能(除 STOP 模式之外)。

因为所有的外设时钟都在文件《imximage.cfg.cfgtmp》进行了初始化，

所有我们不需要再次设置。

| CGR value | Clock Activity Description |
|---|---|
| 00 | Clock is off during all modes. Stop enter hardware handshake is disabled. |
| 01 | Clock is on in run mode, but off in WAIT and STOP modes |
| 10 | Not applicable (Reserved). |
| 11 | Clock is on during all modes, except STOP mode. |

③ 设置 SPI SCLK 时钟

前面只是设置了提供给 SPI 控制器的时钟，它输出的 SCLK 时钟是多少，还需要进一步设置。

为了与外部设备时钟相匹配，我们将实际的 SPI 时钟 SCLK 设置为 4MHz。

涉及到的寄存器为 ECSPI3_CONREG 寄存器 Bit15:12 和 Bit11:8，计算公式如下所示：

```
fre_spi=60MHz/((bit15:11+1)*(2^(bit11:8)))
```

根据上述公式，为了将 SPI 控制器频率设置为 4MHz，我们需要将 bit15:11 设置为 14 即可，计算结果如下：

```
60MHz/((14+1)*(2^0))=4MHz
```

➢ **SPI 时钟设置相关代码**

SPI 时钟设置相关代码如下所示，这部分的代码在程序文件 spi.c 第 71 行~第 85 行。

```
71  /*设置时钟相关的*/
72  /*
73  从 RM 手册 chapter18 中，我们得知时钟来源为 PLL3
74  1、pll3_sw_clk_sel 为 0，则选择 pll3；为 1 则选择 ccm_pll3_bys，时钟   默认选择 pll3 。输出 pll3_sw_clk 给 spi 进行使用   输出给 spi 的时钟为 480M/8=60Mhz
75  2、我们需要使能 spi 的时钟进行使用，通过 CCM_CCGR1 的 bit5:2 来进行设置 这部分在制作.imx 文件的时候初始化，可以不处理
76  3、计算时钟频率 CONREG 寄存器
77      bit15:12 div_1
78      bit11:8  div_2
79  最终提供给 spip 的时钟为
80  60M/(div+1)*(2^div_2))
81  假设我们要使用的时钟是 4M
82  则我们设置 bit15:12 = 15 即可  60M/4 = 15Mhz
83  */
84  uc_num->CONREG &= ~(0xf<<12|0xf<<8);//清除原先的时钟频率设置
85  uc_num->CONREG |= (14<<12); //设置 clk = 60/(14+1) = 4M
```

## 15.4.3 SPI 控制器控制和配置

我们将 SPI 控制器作为 master 来连接外部设备，除引脚设置外，涉及 SPI 控制器的 2 个寄存器：CONREG(控制寄存器)、CONFIGREG(配置寄存器)。

➢ **步骤 1_CONREG 寄存器**

CONREG 寄存器相关设置如下图所示：

Address: Base address + 8h offset

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R/W | BURST_LENGTH | | | | | | | | | | | | CHANNEL_SELECT | | DRCTL | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R/W | PRE_DIVIDER | | | | POST_DIVIDER | | | | CHANNEL_MODE | | | | SMC | XCH | HT | EN |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

在前面讲解寄存器时，对其中的位都有详细描述。

具体取值的如下：

- Bit31:20：表示一次 SPI 传输的数据长度，一次传输的最大大小为 2^12 位。我们一次传输一个字节，因此将该值设置为 7
- Bit19:18：表示 SPI 通道选择位，值为 0~3 分别对应通道 0~3。我们选择通道 0，将该值设置为 0。
- Bit17:16：表示 SPI_RDY#信号控制。该信号默认不进行使用，因此我们不进行处理。
- Bit15:12 和 Bit11:8：用于设置 SPI 时钟频率，稍后在 SPI 控制器时钟设置小节进行描述。
- Bit7:4：设置通道工作模式，分别对应四个通道。每个通道对应位位 0 表示 slave 模式。为 1 表示 master 模式。我们将通道 0 设置为 master 模式
- Bit3：开始传输控制。该值为 0 时，表示需要通过控制 SPI 交换位 XCH 来控制数据传输，当 XCH 为 1 则开始传输。该值为 1 时，当数据开始写入 TXFIFO 的时候，立即开始传输。我们将该值设置为 1，写入 TXFIFO 之后立即开始传输
- Bit2：SPI 交换位 XCH，这里不进行使用。
- Bit1：硬件触发使能位。这里不进行使用
- Bit0：SPI 模块使能位，该值位 1 表示使能 SPI 控制器。该值必须在写入除 CONREG 之外的寄存器或开始传输前使能。

根据以上描述，我们需要先将寄存器 CONREG 值清零，然后设置该寄存器，设置为 master 模式，每次传输为一个字节，使能立即传输并使能模块，具体对应的值为：

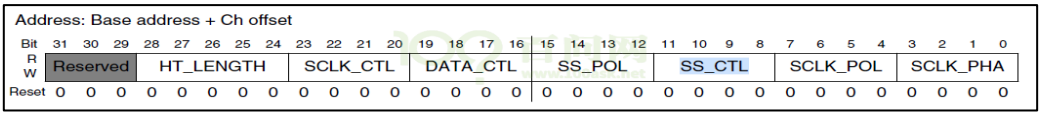
```
（7<<20）|（1<<4）|（1<<3）|（1<<0）
```

代码如下，在 spi.c 中：

```
40  /*
41       1、清除 CONREG 寄存器的 EN 位 来复位模块
42       2、在 ccm 中使能 spi 时钟
43       3、配置 control register，然后设置 CONREG 的 EN 位来使 spi 模块退出复位
44       4、配置 spi 对应的 IOMUX 引脚
45       5、根据外部 spi 设备规格来合适的配置 spi 寄存器
46
47  */
48       printf("spi 初始化开始\n\r") ;
49
50  /**/
51  uc_num->CONREG =  0;// clear all bits
52  /*
53       bit0:使能 SPI
54       bit3:写入 TXDATA 之后，立即发送
55       bit4:设置通道 0 为 master mode
56       bit31:20 设置 burst length ，7 表示为 8bits，一个字节
57  */
58  uc_num->CONREG |= (7<<20)|(1<<4)|(1<<3)|(1<<0);
```

➢ **步骤 2_CONFIGREG 寄存器**

CONFIGREG 寄存器相关设置如下图所示：




在前面讲解寄存器时，对其中的位都有详细描述。

具体值的含义如下：

- Bit28:24：设置 HT 模式下的数据长度，这里不进行使用。
- Bit23:20：设置通道 SCLK 空闲时的电平状态，默认空闲时为低电平，这里使用默认值。
- Bit19:16：设置通道数据线空闲时的电平状态，默认空闲时为高电平，这里使用默认值。
- Bit15:12：设置 SS 片选信号极性设置，默认为低电平有效，这里使用默认值。
- Bit11:8：通道 SS 片选信号， 当 SMC 为 1 的时候此位无效。
- Bit7:4：通道 SCLK 时钟初始电平，为 0 表示高电平有效，这里使用默认值。
- Bit3:0：通道 SCLK 时钟极性选择，为 0 表示第一个时钟沿采集数据，为 1 表示第二个时钟沿采集数据。默认第一个时钟沿采集数据，这里使用默认值。

根据以上描述，我们将 CONFIGREG 寄存器值设置为 0。

代码如下，在 spi.c 中：

```
59  /*  CONFIGREG 采用默认设置
60      *
61      *bit0        PHA=0
62      *bit7:4  sclk 高电平有效
63      *bit11:8 通道片选信号，当 SMC =1 的时候，无效（当前处于 SMC=1 模式）
64      *bit15:12    POL=0
65      *bit19:16    数据线空闲为高电平
66      *bit23:20    sclk 空闲为低电平
67      *bit28:24    设置消息长度 ，该产品不进行使用
68      *
69  */
70  uc_num->CONFIGREG = 0;//
```

## 15.4.4 发送和接收数据

要发送数据时，将值写入 TXDATA 寄存器；要读取数据时，读取 RXDATA 寄存器。

发送和接收的代码如下所示，这部分的代码在程序文件 spi.c 中 spi_writeread 函数：

```
144 while(!(spi_num->STATREG&(1<<0)));//如果 FIFO 时空的话，则填充数据以开始下一次发送
145 spi_num->TXDATA = uc_txdata;
146
147 while(!(spi_num->STATREG&(1<<3)));//等待接收数据完成，当为 1 的时候表示有接收数据存在，可以进行读取
148 return spi_num->RXDATA;
```

## 15.5 ICM-20608-G 编程

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/18_SPI 编程/003_spi_read_icm20608_id”目录下：icm20608g.c。

本节讲解 SPI 控制器的程序。

### 15.5.1 ICM-20608-G 配置初始化

初始化要做的事情挺多，

- **选择时钟，**

之后，设置传感器的采样率、加速度和角速度传感器的量程、加速度和角速度的相关滤波器以及是否使用低功耗模式。当设置完成之后，使能传感器。

- **选择内部晶振作为时钟**

设置 ICM20608G_PWR_MGMT_1 寄存器的 bit[7]为 1，可以复位其内部寄存器，复位成功后该位会自动清 0。实际上，板子上电后 ICM-20608-G 就会用动复位，bit[7]会自动清零。

**Register Name: PWR_MGMT_1**
**Register Type: READ/WRITE**
**Register Address: 107 (Decimal); 6B (Hex)**

| BIT | NAME | FUNCTION |
|---|---|---|
| [7] | DEVICE_RESET | 1 – Reset the internal registers and restores the default settings. The bit automatically clears to 0 once the reset is done. |
| [6] | SLEEP | When set to 1, the chip is set to sleep mode.<br>Note: The default value is 1, the chip comes up in Sleep mode |
| [5] | ACCEL_CYCLE | When set to 1, and SLEEP and STANDBY are not set to 1, the chip will cycle between sleep and taking a single accelerometer sample at a rate determined by SMPLRT_DIV<br>NOTE: When all accelerometer axes are disabled via PWR_MGMT_2 register bits and cycle is enabled, the chip will wake up at the rate determined by the respective registers above, but will not take any samples. |
| [4] | GYRO_STANDBY | When set, the gyro drive and pll circuitry are enabled, but the sense paths are disabled. This is a low power mode that allows quick enabling of the gyros. |
| [3] | TEMP_DIS | When set to 1, this bit disables the temperature sensor. |
| [2:0] | CLKSEL[2:0] | Code Clock Source<br>0 Internal 20MHz oscillator<br>1 Auto selects the best available clock source – PLL if ready, else use the Internal oscillator<br>2 Auto selects the best available clock source – PLL if ready, else use the Internal oscillator<br>3 Auto selects the best available clock source – PLL if ready, else use the Internal oscillator<br>4 Auto selects the best available clock source – PLL if ready, else use the Internal oscillator<br>5 Auto selects the best available clock source – PLL if ready, else use the Internal oscillator<br>6 Internal 20MHz oscillator<br>7 Stops the clock and keeps timing generator in reset |

在源文件 icm20608g.c 的 icm20608g_init 函数中，有如下代码：

```
60  //icm20608_write_addr(ICM20608G_PWR_MGMT_1,0x80);//设备复位，没必要
61  icm20608g_write_addr(ICM20608G_PWR_MGMT_1,0x01);//选择内部 20M 晶振作为时钟
```

- **设置采样率**

对应寄存器地址为 0x19，用于设置采样率，采样率计算公式如下：

```
AMPLE_RATE = INTERNAL_SAMPLE_RATE / (1 + SMPLRT_DIV)
```

其中 INTERNAL_SAMPLE_RATE = 1kHz。

我们使用 1K 的采样率，，因此该寄存器值设定为 0，寄存器如下图所示：

**Register Name: SMPLRT_DIV**
**Register Type: READ/WRITE**
**Register Address: 25 (Decimal); 19 (Hex)**

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | SMPLRT_DIV[7:0] | Divides the internal sample rate (see register CONFIG) to generate the sample rate that controls sensor data output rate, FIFO sample rate. NOTE: This register is only effective when FCHOICE_B register bits are 2'b00, and (0 < DLPF_CFG < 7). This is the update rate of the sensor register:<br>SAMPLE_RATE = INTERNAL_SAMPLE_RATE / (1 + SMPLRT_DIV)<br>Where INTERNAL_SAMPLE_RATE = 1kHz |

在源文件 icm20608g.c 的 icm20608g_init 函数中，有如下代码：

```
70  icm20608g_write_addr(ICM20608G_SMPLRT_DIV,0x00);//采样率默认 1K
```

➢ **配置寄存器**

该寄存器地址为 0x1A，可以设置 FIFO 的模式、外部 FSYNC 引脚功能以及 DLPF 配置，具体如下图所示。我们将低通滤波器设定为 250Hz，其余采用默认，因此该寄存器值为 0x00。

**Register Name: CONFIG**
**Register Type: READ/WRITE**
**Register Address: 26 (Decimal); 1A (Hex)**

| BIT | NAME | FUNCTION |
|---|---|---|
| [7] | - | Always set to 0 |
| [6] | FIFO_MODE | When set to '1', when the FIFO is full, additional writes will not be written to FIFO. When set to '0', when the FIFO is full, additional writes will be written to the FIFO, replacing the oldest data. |
| [5:3] | EXT_SYNC_SET[2:0] | Enables the FSYNC pin data to be sampled.<br>EXT_SYNC_SET / FSYNC bit location<br>0 / function disabled<br>1 / TEMP_OUT_L[0]<br>2 / GYRO_XOUT_L[0]<br>3 / GYRO_YOUT_L[0]<br>4 / GYRO_ZOUT_L[0]<br>5 / ACCEL_XOUT_L[0]<br>6 / ACCEL_YOUT_L[0]<br>7 / ACCEL_ZOUT_L[0]<br>FSYNC will be latched to capture short strobes. This will be done such that if FSYNC toggles, the latched value toggles, but won't toggle again until the new latched value is captured by the sample rate strobe. |
| [2:0] | DLPF_CFG[2:0] | For the DLPF to be used, FCHOICE_B[1:0] is 2'b00.<br>See the table below. |

低通滤波器 DLPF_CFG 的设定值范围如下所示，本程序设置为 0：

The DLPF is configured by *DLPF_CFG*, when *FCHOICE_B* [1:0] = 2b'00. The gyroscope and temperature sensor are filtered according to the value of *DLPF_CFG* and *FCHOICE_B* as shown in the table below.

| FCHOICE_B | | DLPF_CFG | Gyroscope | | | Temperature Sensor |
|---|---|---|---|---|---|---|
| <1> | <0> | | 3-dB BW (Hz) | Noise BW (Hz) | Rate (kHz) | 3-dB BW (Hz) |
| X | 1 | X | 8173 | 8595.1 | 32 | 4000 |
| 1 | 0 | X | 3281 | 3451.0 | 32 | 4000 |
| 0 | 0 | 0 | 250 | 306.6 | 8 | 4000 |
| 0 | 0 | 1 | 176 | 177.0 | 1 | 188 |
| 0 | 0 | 2 | 92 | 108.6 | 1 | 98 |
| 0 | 0 | 3 | 41 | 59.0 | 1 | 42 |
| 0 | 0 | 4 | 20 | 30.5 | 1 | 20 |
| 0 | 0 | 5 | 10 | 15.6 | 1 | 10 |
| 0 | 0 | 6 | 5 | 8.0 | 1 | 5 |
| 0 | 0 | 7 | 3281 | 3451.0 | 8 | 4000 |

在源文件 icm20608g.c 的 icm20608g_init 函数中，有如下代码：

```
71  icm20608g_write_addr(ICM20608G_CONFIG, 0x00);//禁止 FIFO
```

角速度传感器设置

该寄存器地址为 0x1B，可以设置角速度传感器自测和量程范围以及低通滤波器的范围，具体如下图所示。

我们选择量程范围为±250dps，其余使用默认值，因此该寄存器值设定为 0x00。

**Register Name: GYRO_CONFIG**
**Register Type: READ/WRITE**
**Register Address: 27 (Decimal); 1B (Hex)**

| BIT | NAME | FUNCTION |
|---|---|---|
| [7] | XG_ST | X Gyro self-test |
| [6] | YG_ST | Y Gyro self-test |
| [5] | ZG_ST | Z Gyro self-test |
| [4:3] | FS_SEL[1:0] | Gyro Full Scale Select:<br>00 = ±250dps<br>01= ±500dps<br>10 = ±1000dps<br>11 = ±2000dps |
| [2] | - | Reserved |
| [1:0] | FCHOICE_B[1:0] | Used to bypass DLPF as shown in table 1 above. |

在源文件 icm20608g.c 的 icm20608g_init 函数中，有如下代码：

```
72  icm20608g_write_addr(ICM20608G_GYRO_CONFIG,0x00);//使用默认量程和低通滤波器
```

➢ **加速度传感器设置 1**

该寄存器地址为 0x1C，可以设置加速度传感器自测和量程范围以及低通滤波器的范围，具体如下图所示：

**Register Name: ACCEL_CONFIG**
**Register Type: READ/WRITE**
**Register Address: 28 (Decimal); 1C (Hex)**

| BIT | NAME | FUNCTION |
|---|---|---|
| [7] | XA_ST | X Accel self-test |
| [6] | YA_ST | Y Accel self-test |
| [5] | ZA_ST | Z Accel self-test |
| [4:3] | ACCEL_FS_SEL[1:0] | Accel Full Scale Select:<br>±2g (00), ±4g (01), ±8g (10), ±16g (11) |
| [2:0] | - | Reserved |

我们选择量程范围为±2g，其余使用默认值，因此该寄存器值设定为 0x00。

在源文件 icm20608g.c 的 icm20608g_init 函数中，有如下代码：

```
73  icm20608g_write_addr(ICM20608G_ACC_CONFIG,0x00);//使用默认量程
```

➢ **加速度传感器设置 2**

该寄存器地址为 0x1D，可以设置加速度传感器采样和低通滤波器的范围，具体如下图所示：

**Register Name: ACCEL_CONFIG2**
**Register Type: READ/WRITE**
**Register Address: 29 (Decimal); 1D (Hex)**

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:6] | - | Reserved |
| [5:4] | DEC2_CFG[1:0] | Averaging filter settings for Low Power Accelerometer mode:<br>0 = Average 4 samples<br>1 = Average 8 samples<br>2 = Average 16 samples<br>3 = Average 32 samples |
| [3] | ACCEL_FCHOICE_B | Used to bypass DLPF as shown in the table below. |
| [2:0] | A_DLPF_CFG | Accelerometer low pass filter setting as shown in the table below. |

我们选择低通滤波器为 218.1Hz，其余使用默认值，因此该寄存器值设定为 0x00。

在源文件 icm20608g.c 的 icm20608g_init 函数中，有如下代码：

```
74  icm20608g_write_addr(ICM20608G_ACC_CONFIG2,0x00);//使用默认低通滤波器
```

➢ **低功耗模式设置**

该寄存器地址为 0x1E，可以设置低功耗模式和低功耗模式下唤醒频率等，具体如下图所示：

**Register Name: LP_MODE_CFG**
**Register Type: READ/WRITE**
**Register Address: 30 (Decimal); 1E (Hex)**

| BIT | NAME | FUNCTION |
|---|---|---|
| [7] | GYRO_CYCLE | When set to '1' low-power gyroscope mode is enabled. Default setting is '0' |
| [6:4] | G_AVGCFG[2:0] | Averaging filter configuration for low-power gyroscope mode. Default setting is '000' |
| [3:0] | LPOSC_CLKSEL | Sets the frequency of waking up the chip to take a sample of accel data – the low power accel Output Data Rate |

| LPOSC_CLKSEL | Output Frequency (Hz) |
|---|---|
| 0 | 0.24 |
| 1 | 0.49 |
| 2 | 0.98 |
| 3 | 1.95 |
| 4 | 3.91 |
| 5 | 7.81 |
| 6 | 15.63 |
| 7 | 31.25 |
| 8 | 62.50 |
| 9 | 125 |
| 10 | 250 |
| 11 | 500 |
| 12-15 | Reserved |

我们不需要低功耗模模式，因此将该寄存器设置为 0，关闭低功耗模式。
在源文件 icm20608g.c 的 icm20608g_init 函数中，有如下代码：

```
75  icm20608g_write_addr(ICM20608G_LP_MODE_CFG,0x00);//关闭低功耗模式
```

- **FIFO 使能设置**

该寄存器地址为 0x23，用于使能传感器的 FIFO，具体如下图所示：

Register Name: FIFO_EN
Register Type: READ/WRITE
Register Address: 35 (Decimal); 23 (Hex)

| BIT | NAME | FUNCTION |
|---|---|---|
| [7] | TEMP_FIFO_EN | 1 – Write TEMP_OUT_H and TEMP_OUT_L to the FIFO at the sample rate; If enabled, buffering of data occurs even if data path is in standby.<br>0 – function is disabled |
| [6] | XG_FIFO_EN | 1 – Write GYRO_XOUT_H and GYRO_XOUT_L to the FIFO at the sample rate; If enabled, buffering of data occurs even if data path is in standby.<br>0 – function is disabled |
| [5] | YG_FIFO_EN | 1 – Write GYRO_YOUT_H and GYRO_YOUT_L to the FIFO at the sample rate; If enabled, buffering of data occurs even if data path is in standby.<br>0 – function is disabled<br>NOTE: Enabling any one of the bits corresponding to the Gyros or Temp data paths, data is buffered into the FIFO even though that data path is not enabled. |
| [4] | ZG_FIFO_EN | 1 – Write GYRO_ZOUT_H and GYRO_ZOUT_L to the FIFO at the sample rate; If enabled, buffering of data occurs even if data path is in standby.<br>0 – function is disabled |
| [3] | ACCEL_FIFO_EN | 1 – write ACCEL_XOUT_H, ACCEL_XOUT_L, ACCEL_YOUT_H, ACCEL_YOUT_L, ACCEL_ZOUT_H, and ACCEL_ZOUT_L to the FIFO at the sample rate;<br>0 – function is disabled |
| [2:0] | - | Reserved |

由于我们不需要 FIFO，因此我们将该寄存器值设定为 0，关闭所有 FIFO。
在源文件 icm20608g.c 的 icm20608g_init 函数中，有如下代码：

```
76  icm20608g_write_addr(ICM20608G_FIFO_EN,0x00);//禁止传感器 FIFO
```

- **电源管理设置 2**

该寄存器地址为 0x6C，可以设置传感器使能等，具体如下图所示：

Register Name: PWR_MGMT_2
Register Type: READ/WRITE
Register Address: 108 (Decimal); 6C (Hex)

| BIT | NAME | FUNCTION |
|---|---|---|
| [7] | FIFO_LP_EN | 1 – Enable FIFO in low-power accelerometer mode. Default setting is 0. |
| [6] | - | Reserved. |
| [5] | STBY_XA | 1 – X accelerometer is disabled<br>0 – X accelerometer is on |
| [4] | STBY_YA | 1 – Y accelerometer is disabled<br>0 – Y accelerometer is on |
| [3] | STBY_ZA | 1 – Z accelerometer is disabled<br>0 – Z accelerometer is on |
| [2] | STBY_XG | 1 – X gyro is disabled<br>0 – X gyro is on |
| [1] | STBY_YG | 1 – Y gyro is disabled<br>0 – Y gyro is on |
| [0] | STBY_ZG | 1 – Z gyro is disabled<br>0 – Z gyro is on |

我们使用所有的传感器，因此将该值设置为 0x00。
注意：另一个寄存器“电源管理设置 1”在前面已经设置过，我们用来选择时钟。
在源文件 icm20608g.c 的 icm20608g_init 函数中，有如下代码：

```
77  icm20608g_write_addr(ICM20608G_PWR_MGMT_2,0x00);//使能传感器
```

## 15.5.2 读取 ICM-20608-G 的设备 ID

初始化好 ICM-20608-G 后，可以试试读出设备 ID。
设备 ID 寄存器地址为 0x75，可以读取 WHO_AM_I 信息。WHO_AM_I 是一个

8 位设备 ID，该值默认为 0xAF，具体如下图所示。

Register Name: WHO_AM_I
Register Type: READ only
Register Address: 117 (Decimal); 75 (Hex)

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | WHOAMI | Register to indicate to user which device is being accessed. |

This register is used to verify the identity of the device. The contents of *WHOAMI* is an 8-bit device ID. The default value of the register is 0xAF. This is different from the I2C address of the device as seen on the slave I2C controller by the applications processor. The I2C address of the ICM-20608-G is 0x68 or 0x69 depending upon the value driven on AD0 pin.

读取设备 ID 相关的代码如下所示，这部分的代码这 2 个程序里都有：

```
003_spi_read_icm20608_id
004_read_sensor_data
```

源文件是 icm20608g.c，在 icm20608g_init 函数中第 62 行~第 69 行，如下：

```
62  //读取设备 id 并对比，如果不等于 0xaf，则退出初始化
63  uc_dev_id = icm20608g_read_addr(ICM20608G_WHO_AM_I);
64  printf("read icm20608g id is 0x%x\n\r",uc_dev_id);
65  if(uc_dev_id!=0xAF)
66  {
67      printf("read id fail\n\r");
68      return -1;
69  }
```

## 15.5.3 读取 ICM-20608-G 的温度信息

温度传感器数据，其值分为高低两个字节，分别对应寄存器地址 0x41 和 0x42，把这 2 个寄存器的值读出后组合得到一个 16 位的值：Temp_out。

使用 Temp_out 可以算出温度值，对应计算公式为：

```
Temp_degC = (Temp_out-RoomTemp_Offset)/Temp_sensitivity)+25degC
```

寄存器如下图所示：

Register Name: TEMP_OUT_H
Register Type: READ only
Register Address: 65 (Decimal); 41 (Hex)

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | TEMP_OUT[15:8] | High byte of the temperature sensor output |

Register Name: TEMP_OUT_L
Register Type: READ only
Register Address: 66 (Decimal); 42 (Hex)

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | TEMP_OUT[7:0] | Low byte of the temperature sensor output<br>TEMP_degC = ((TEMP_OUT – RoomTemp_Offset)/Temp_Sensitivity) + 25degC |

在芯片手册中也找不到上述计算公式中的 RoomTemp_Offset 和 Temp_sensitivity，程序中直接打印寄存器的值。

读取温度相关的代码如下所示，这部分的代码在程序文件《004_read_sensor_data/icm20608g.c》的 icm20608g_read_temp 函数中：

```
181 icm20608g_read_len(0x41,uc_buf,2);
182 icm20608g_get.temp_adc = (signed short)((uc_buf[0]<<8)|uc_buf[1]);
```

## 15.5.4 读取 ICM-20608-G 的加速度信息

ICM-20608-G 支持加速度测量，对应地址为从寄存器地址 59 到 64。其中 59 和 60 地址对应 X 轴加速度高低字节，61 和 62 地址对应 Y 轴加速度高低字

节，63 和 64 地址对应 Z 轴高低字节。高低字节拼接之后进行处理即可。寄存器如下图所示

Register Name: ACCEL_XOUT_H
Register Type: READ only
Register Address: 59 (Decimal); 3B (Hex)

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | ACCEL_XOUT_H[15:8] | High byte of accelerometer x-axis data. |

Register Name: ACCEL_XOUT_L
Register Type: READ only
Register Address: 60 (Decimal); 3C (Hex)

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | ACCEL_XOUT_L[7:0] | Low byte of accelerometer x-axis data. |

Register Name: ACCEL_YOUT_H
Register Type: READ only
Register Address: 61 (Decimal); 3D (Hex)

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | ACCEL_YOUT_H[15:8] | High byte of accelerometer y-axis data. |

Register Name: ACCEL_YOUT_L
Register Type: READ only
Register Address: 62 (Decimal); 3E (Hex)

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | ACCEL_YOUT_L[7:0] | Low byte of accelerometer y-axis data. |

Register Name: ACCEL_ZOUT_H
Register Type: READ only
Register Address: 63 (Decimal); 3F (Hex)

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | ACCEL_ZOUT_H[15:8] | High byte of accelerometer z-axis data. |

Register Name: ACCEL_ZOUT_L
Register Type: READ only
Register Address: 64 (Decimal); 40 (Hex)

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | ACCEL_ZOUT_L[7:0] | Low byte of accelerometer z-axis data. |

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/18_SPI 编程/004_read_sensor_data”目录下：icm20608g.c。读取加速度相关的代码如下所示，icm20608g_read_acc 函数：

```
139 icm20608g_read_len(0x3b,uc_buf,6);
140 icm20608g_get.acc_x_adc = (signed short)((uc_buf[0]<<8)|uc_buf[1]);
141 icm20608g_get.acc_y_adc = (signed short)((uc_buf[2]<<8)|uc_buf[3]);
142 icm20608g_get.acc_z_adc = (signed short)((uc_buf[4]<<8)|uc_buf[5]);
```

## 15.5.5 读取 ICM-20608-G 的角速度信息

ICM-20608-G 支持角速度测量，对应地址为从寄存器地址 67 到 72。其中 67 和 68 地址对应 X 轴角速度高低字节，69 和 70 地址对应 Y 轴角速度高低字节，71 和 72 地址对应 Z 轴角速度高低字节。高低字节拼接之后进行处理即可。寄存器如下图所示

**Register Name: GYRO_XOUT_H**
**Register Type: READ only**
**Register Address: 67 (Decimal); 43 (Hex)**

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | GYRO_XOUT[15:8] | High byte of the X-Axis gyroscope output |

**Register Name: GYRO_XOUT_L**
**Register Type: READ only**
**Register Address: 68 (Decimal); 44 (Hex)**

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | GYRO_XOUT[7:0] | Low byte of the X-Axis gyroscope output<br>**GYRO_XOUT =** Gyro_Sensitivity * X_angular_rate<br>Nominal Conditions: FS_SEL = 0<br>Gyro_Sensitivity = 131 LSB/(º/s) |

**Register Name: GYRO_YOUT_H**
**Register Type: READ only**
**Register Address: 69 (Decimal); 45 (Hex)**

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | GYRO_YOUT[15:8] | High byte of the Y-Axis gyroscope output |

**Register Name: GYRO_YOUT_L**
**Register Type: READ only**
**Register Address: 70 (Decimal); 46 (Hex)**

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | GYRO_YOUT[7:0] | Low byte of the Y-Axis gyroscope output<br>**GYRO_YOUT =** Gyro_Sensitivity * Y_angular_rate<br>Nominal Conditions: FS_SEL = 0<br>Gyro_Sensitivity = 131 LSB/(º/s) |

**Register Name: GYRO_ZOUT_H**
**Register Type: READ only**
**Register Address: 71 (Decimal); 47 (Hex)**

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | GYRO_ZOUT[15:8] | High byte of the Z-Axis gyroscope output |

**Register Name: GYRO_ZOUT_L**
**Register Type: READ only**
**Register Address: 72 (Decimal); 48 (Hex)**

| BIT | NAME | FUNCTION |
|---|---|---|
| [7:0] | GYRO_YOUT[7:0] | Low byte of the Z-Axis gyroscope output<br>**GYRO_ZOUT =** Gyro_Sensitivity * Z_angular_rate<br>Nominal Conditions: FS_SEL = 0<br>Gyro_Sensitivity = 131 LSB/(º/s) |

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/18_SPI 编程/004_read_sensor_data”目录下：icm20608g.c。读取角速度相关的代码如下所示，icm20608g_read_gyro 函数：

```
160 icm20608g_read_len(0x43,uc_buf,6);
161 icm20608g_get.gyro_x_adc = (signed short)((uc_buf[0]<<8)|uc_buf[1]);
162 icm20608g_get.gyro_y_adc = (signed short)((uc_buf[2]<<8)|uc_buf[3]);
```

```
163 icm20608g_get.gyro_z_adc = (signed short)((uc_buf[4]<<8)|uc_buf[5]);
```

## 15.6 上机实验

### 15.6.1 SPI 控制器初始化测试（无示波器可不进行观察试验）

执行函数 spi_init(ESCPI3_BASE);即可初始化 SPI 控制器，通过 spi_writeread（0x54）执行向外数据的输出，可以通过测试控制器的输出时钟 SCLK 来测试控制器是否初始化成功。这部分的试验代码在《001_spi_init/main.c》，具体代码如下所示：

```
18 int  main()
19 {
20     unsigned char uc_cnt;
21     spi_init(ESCPI3_BASE);
22     while(1)
23     {
24         spi_writeread(ESCPI3_BASE,0x55);
25        delay(1000);
26     }
27     return 0;
28 }
```

### 15.6.2 SPI 控制器回环测试

该实验主要用于测试 SPI 控制器本身功能，通过将收发连接到一块来进行测试。

首先，在《002_spi_loopback/spi.c》中实现 spi_test_rw 函数，然后在《002_spi_loopback/main.c》中调用此函数。spi_test_rw 函数功能如下：

① 首先设置 SPI 控制器进入回环模式；
② 然后在缓冲区中构造数据；
③ 进行读写测试。

写入数据、读出来是本程序的核心，第 181 行是写数据，把 uc_buf_write 数组中的数据通过 SPI 发送出去：

```
181     spi_writeread(spi_num,uc_buf_write[uc_cnt]);
```

然后，立刻，通过第 183 行把数据马上读回来，存入 uc_buf_read 数组：

```
183     uc_buf_read[uc_cnt]=spi_writeread(ESCPI3_BASE,0xff);
```

在第 186～194 行对比读、写缓冲区。

spi_test_rw 函数代码在程序文件《002_spi_loopback/spi.c》中，具体如下所示：

```
169 //设置进入 loop 模式，进行测试
170 spi_num->TESTREG = (1<<31);
171 printf("spi 进入回环测试模式\n\r");
172 //造数
173 for(uc_cnt=0;uc_cnt<20;uc_cnt++)
174 {
175     uc_buf_write[uc_cnt] = 0x20+uc_cnt;
176 }
177 //进行读写测试
178 for(uc_cnt=0;uc_cnt<20;uc_cnt++)
179 {
180     printf("write_cnt %d\t",uc_cnt);
181     spi_writeread(spi_num,uc_buf_write[uc_cnt]);
182     printf("write %d\t",uc_buf_write[uc_cnt]);
```

```
183     uc_buf_read[uc_cnt]=spi_writeread(ESCPI3_BASE,0xff);
184     printf("read %d\n\r",uc_buf_read[uc_cnt]);
185 }
186 //进行数据对比
187 for(uc_cnt=0;uc_cnt<20;uc_cnt++)
188 {
189     if(uc_buf_read[uc_cnt]!=uc_buf_write[uc_cnt])
190     {/*表示回环测试失败，存在问题*/
191         printf("!!! spi 回环测试失败\n\r");
192         return -1;
193     }
194 }
195 printf("@@@spi 回环测试成功\n\r");
196 printf("spi 退出回环测试模式\n\r");
197     //exit loopback mode
198 spi_num->TESTREG = 0;
```

之后在《002_spi_loopback/main.c》中调用 spi_test_rw 实现回环测试，main 函数代码如下：

```
int  main()
{
    unsigned char uc_cnt;
    spi_init(ESCPI3_BASE);
    spi_test_rw(ESCPI3_BASE);
    return 0;
}
```

编译、运行程序，观察串口输出信息。测试结果记录如下，可以看到读取和写入的数据相同：

```
spi 初始化开始
spi 初始化结束
spi进入回环测试模式
write_cnt 0     write 32        read 32
write_cnt 1     write 33        read 33
write_cnt 2     write 34        read 34
write_cnt 3     write 35        read 35
write_cnt 4     write 36        read 36
write_cnt 5     write 37        read 37
write_cnt 6     write 38        read 38
write_cnt 7     write 39        read 39
write_cnt 8     write 40        read 40
write_cnt 9     write 41        read 41
write_cnt 10    write 42        read 42
write_cnt 11    write 43        read 43
write_cnt 12    write 44        read 44
write_cnt 13    write 45        read 45
write_cnt 14    write 46        read 46
write_cnt 15    write 47        read 47
write_cnt 16    write 48        read 48
write_cnt 17    write 49        read 49
write_cnt 18    write 50        read 50
write_cnt 19    write 51        read 51
@@@spi回环测试成功
spi退出回环测试模式
```

## 15.6.3 读取 ICM-20608-G 的设备 ID

该实验用于测试读取与 ICM-20608-G 的 SPI 通讯接口功能正常与否。如果能够正确的读取到设备 ID，则表示 SPI 接口通讯正常。

在《003_spi_read_icm20608_id/main.c》中调用 icm20608g_init 实现 ICM-20608-G 的初始化和 ID 的读取，调用代码如下所示：

```
04 int  main()
05 {
06   unsigned char uc_cnt;
07   icm20608g_init();//初始化传感器 ICM-20608-G
08   return 0;
09 }
```

编译、运行程序，观察串口输出信息，可以看到读取到的 ID 是 0xAF，与 ICM-20608-G 的 ID 一致。

### 15.6.4 读取 ICM-20608-G 的传感器信息

该实验读取寄存器，得到温度、三轴角速度和加速度值。在《004_read_sensor_data/main.c》中调用 icm20608g_read_acc、icm20608g_read_ac 和 icm20608g_read_temp 函数来实现传感器数值读取，调用代码如下所示：

```
04 int  main()
05 {
06  unsigned char uc_cnt;
07  icm20608g_init();//初始化传感器 ICM-20608-G
08  for(uc_cnt=0;uc_cnt<1;uc_cnt++)
09  {
10      icm20608g_read_acc();
11      icm20608g_read_gyro();
12      icm20608g_read_temp();
13  }
14
15  return 0;
16 }
```

编译、运行程序，观察串口输出信息。

# 第16章 百问网传感器模块介绍

## 16.1 模块的分类

100ASK_IMX6ULL 开发板由核心板和底板组成。

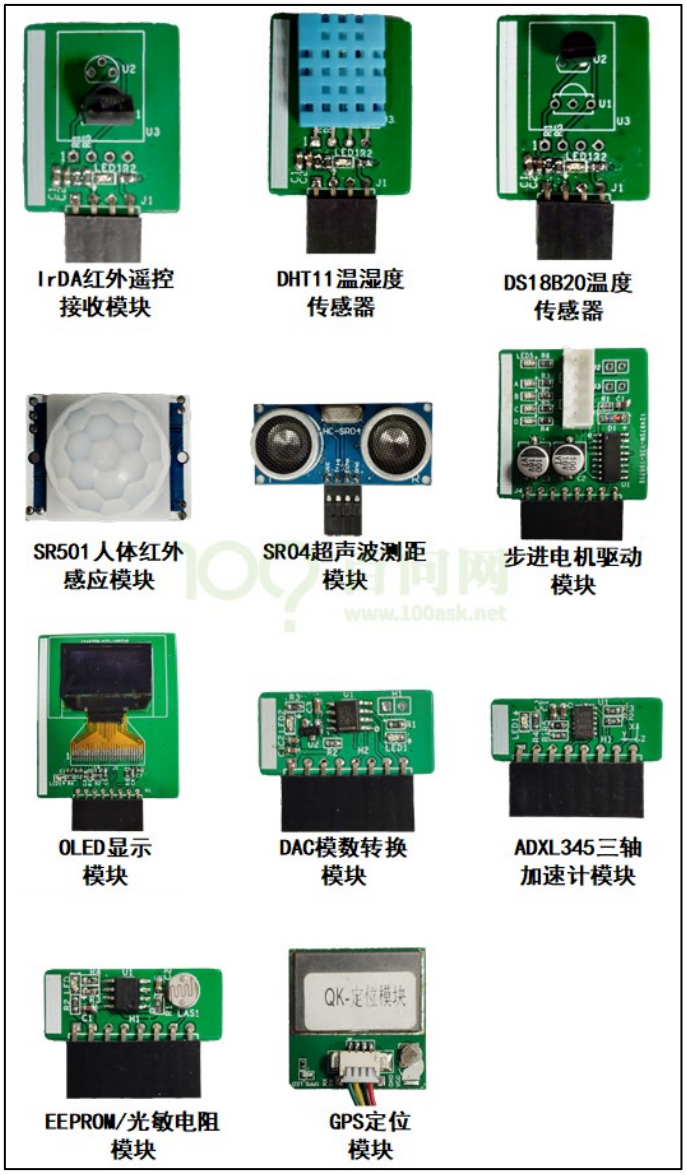
核心板是一个最小系统，包含 CPU、RAM、Flash 和电源芯片等必须的芯片，以及引出各种引脚。

底板上有网口、USB 口、LCD 接口、TTL 电平的串口等。此外还包含少量的模块，比如 WIFI/Bluetooth 等。底板上的模块还不够你使用，怎么办？通过扩展板，接上更多模块。

这些通用模块，接口协议都比较简单，适合裸机开发练习。

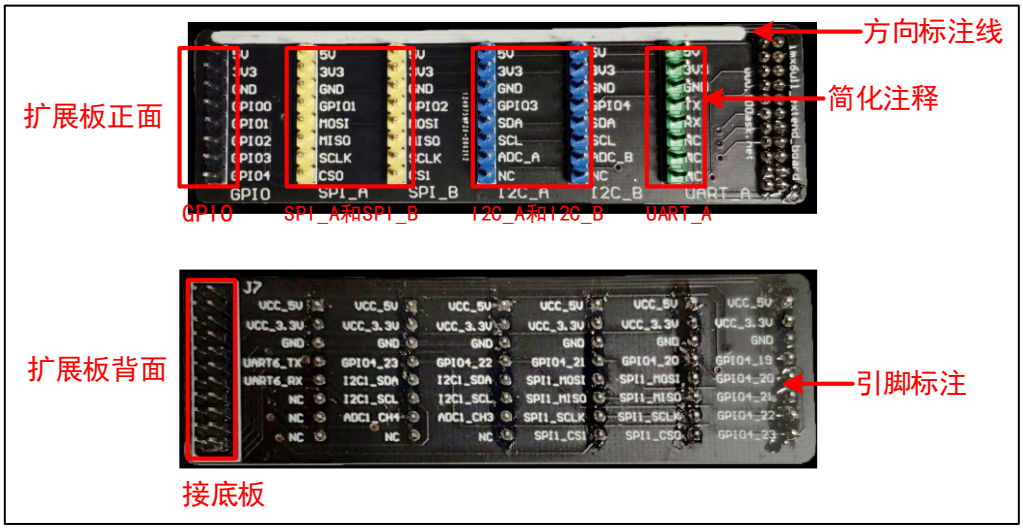
## 16.2 通用模块介绍

目前设计好的通用模块有十余个，后续可能会继续生产更多模块，现有模块如下图所示。



每个开发板的接口引脚顺序各有不同，需要接上它自己的扩展板，才能在扩展板上使用各种通用模块。

100ASK_IMX6ULL 扩展板示意图如下图所示。




扩展板正面从左到右依次是一列 GPIO 接口、两列 SPI 接口(SPI_A 和 SPI_B)、两列 I2C 接口(I2C_A 和 I2C_B)，一列 UART 接口(UART_A)，其中 I2C 接口所在列还包含 ADC 接口(ADC_A 和 ADC_B)。每列接口都提供 5V、3V3 电源，以便支持工作电压不同的模块。

扩展板正面文字注释为简化注释，方便统一称呼。上方有根白色的方向标注线用于方向指引，配合模块上的白色标注线，接模块时，它的白线或扩展板的白线都在同一边(白线对齐，就不会插反)。

扩展板背面文字注释为 100ASK_IMX6ULL 原理图上的引脚标注，参考附录一，供想深入学习的用户参考。

使用扩展板的 GPIO 时要留意：

① 对于只需要一个 GPIO 的模块

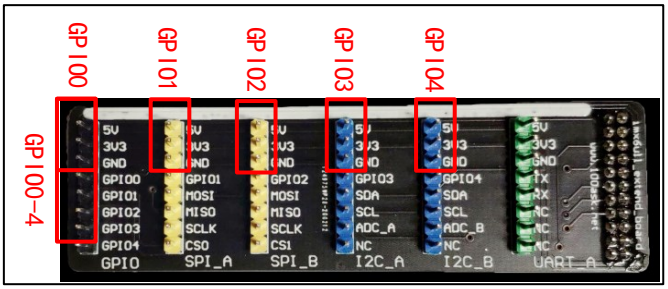
硬件上可以插在五个排针座中的任意一个的上半部分，如下图所示的 GPIO0、GPIO1、GPIO2、GPIO3、GPIO4。

但是如果软件用的是 GPIOx，你就应该接到 GPIOx 对应的插针上。

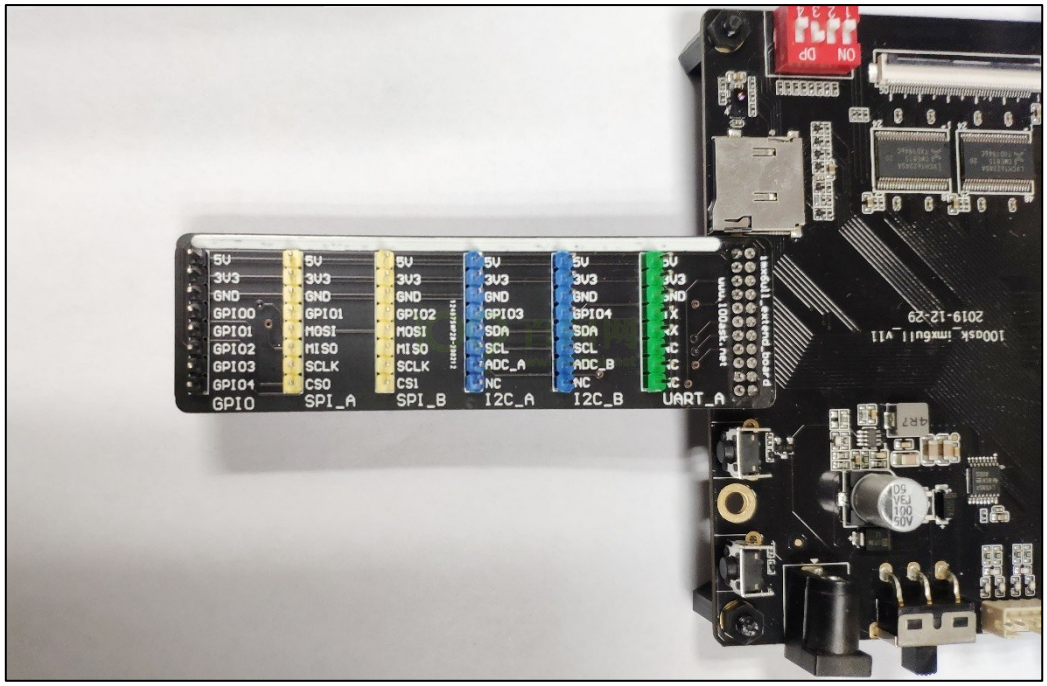
② 对于步进电机驱动板这种需要两个及以上 GPIO 的模块，只能插在最左边的黑色排针 GPIO0 上；

③ 同一个 GPIO，不能同时用于多个模块

比如不能同时使用电机驱动板和 OLED 模块。因为电机驱动板占用了 GPIO0～3；而 OLED 模块要用到 SPI 接口、一个 GPIO 口，本来可以接在 SPI_A 或 SPI_B，但所需的 GPIO1～2 被电机驱动板占用了。




扩展板插在底板的“J5 Camera & Extend”上(在 TF 卡槽附近)，方向如下图所示。




## 16.3 模块原理图和芯片资料

所有模块的资料都在百度网盘开发板配套资料中，下载到 100ASK_IMX6ULL 的配套资料包后，有以下目录：05_Hardware (原理图)/Extend_modules/

每个模块对应一个压缩包，里面既有原理图，也有芯片资料。

# 第17章 IRDA 红外遥控模块

模块原理图及资料：网盘开发板配套资料“05_Hardware（原理图)/Extend_modules/irda.zip”。

## 17.1 红外遥控简介

红外遥控被广泛应用于家用电器、工业控制和智能仪器系统中，像我们熟知的有电视机盒子遥控器、空调遥控器。红外遥控器系统分为发送端和接收端，如图下图所示。


发送端就是红外遥控器，上面有许多按键，当我们按下遥控器按键时，遥控器内部电路会进行编码和调制，再通过红外发射头，将信号以肉眼不可见的红外线发射出去。红外线线虽然肉眼不可见，但可以通过手机摄像头看到，常用该方法检查遥控器是否正常工作。

接收端是一个红外接收头，收到红外信号后，内部电路会进行信号放大和解调，再将数据传给板子上的 GPIO，板子收到数据后再解码才能确定是哪个按键被按下。

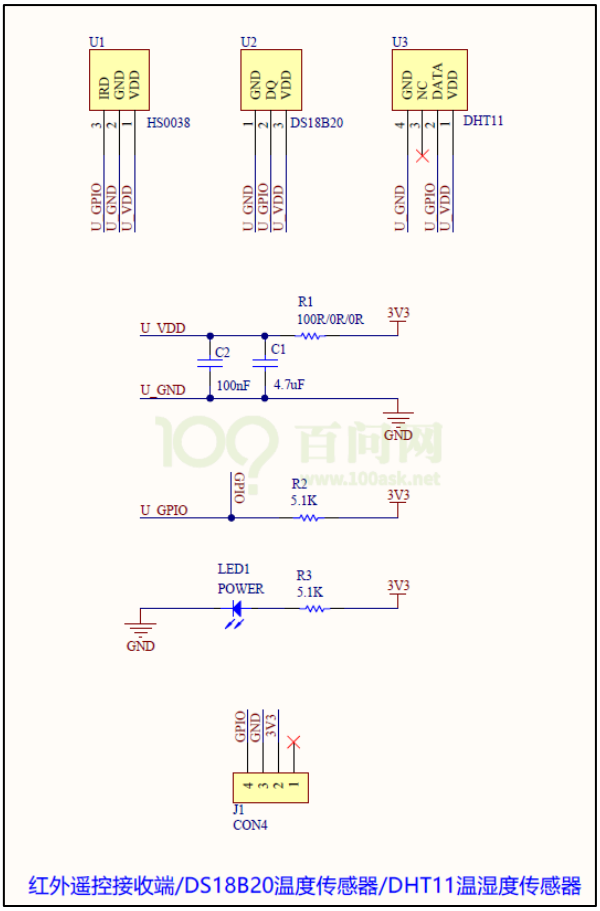

## 17.2 IRDA 红外遥控模块硬件设计

IRDA 红外接收头，只需要一个 GPIO 即可实现数据的传输，这种传输协议叫做“1-Wire 单总线”。顾名思义，即只有一根数据线，系统中的数据交换、控制都由这根线完成。

后面的 DS18B20 和 DHT11 也是使用该协议，且电路设计类似。因此将三个模块的 PCB 电路板做成了同一个，生产的时候，分别只焊接“U1”、“U2”、“U3”来实现不同的功能。

原理图中的 U1（HS0038）即为 IRDA 红外接收头，1 脚 VDD 接到了 3V3，2 脚 GND 接到了 GND，3 脚 IRD 外接 GPIO。

在使用的时候，将模块插到 100ASK_6ULL 扩展板上，同时扩展板插到 100ASK_6ULL 底板的“Extend”接口，这样 IRDA 的 IRD 脚就连到到 6ULL 的 GPIO 上。




红外遥控接收端/DS18B20温度传感器/DHT11温湿度传感器

# 17.3 IRDA 红外遥控模块软件设计

## 17.3.1 红外遥控器协议

我们按下遥控器按键的时候，遥控器自动发送某个红外信号，接收头接收到红外信号，然后把红外信号转换成电平信号，通过 IRD 这根线，传给 SOC。整个传输，只涉及单向传输，由 HS0038 向主芯片传送。

因此，我们只需要编写程序，从 IRD 上获取数据即可，在这之前，我们需要先了解下数据是怎么表示的，也就是传输的红外数据的格式。

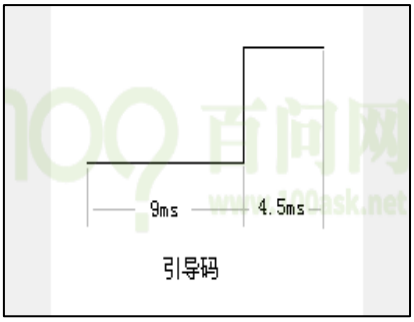
红外协议有：NEC、SONY、RC5、RC6 等，常用的就是 NEC 格式，因此我们主要对 NEC 进行讲解。

在分析文章中的波形之前，我们先想象一下怎么在一条数据线上传输信号。

开始传输数据之前，一般都会发出一个 start 起始信号，通知对方我开始传输数据了，后面就是每一位每一位的数据。

NEC 协议的开始是一段引导码：


这个引导码由一个 9ms 的低脉冲加上一个 4.5ms 的高脉冲组成，它用来通知接收方我要开始传输数据了。



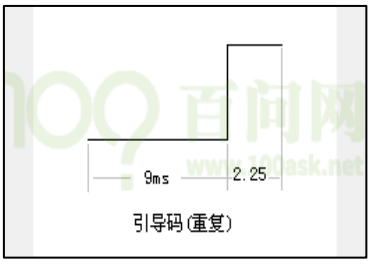

然后接着的是数据，数据由 4 字节组成：地址、地址(取反)、数据、数据(取反)，取反是用来校验用的。

地址是指遥控器的 ID，每一类遥控器的 ID 都不一样，这样就可以防止操控电视的遥控器影响空调。数据就是遥控器上的不同按键值。

从前面的图可以知道，NEC 每次要发 32 位（地址、地址取反、数据、数据取反，每个 8 位）的数据。数据的 1 和 0，开始都是 0.56ms 的低脉冲，对于数据 1，后面的高脉冲比较长，对于数据 0，后面的高脉冲比较短。


第一次按下按键时，它会发出引导码，地址，地址取反，数据，数据取反。

如果这时还没松开按键，这就是“长按”，怎么表示“长按”？遥控器会发送一个不一样的引导码，这个引导码由 9ms 的低脉冲，2.25ms 的高脉冲组成，表示现在按的还是上次一样的按键，也叫连发码，它会一直发送，直到松开。




## 17.3.2 编程思路

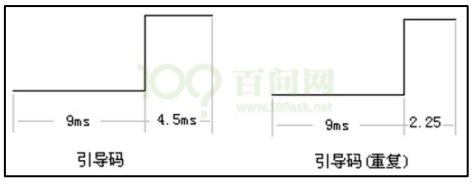
有了上述基础知识后就可以开始编写程序了。

编程思路如下：

① 平时 GPIO 为高；

② 发现 GPIO 为低时，判断它有 9ms 的低电平：

对于引导码，或连发码，它们都有 9ms 的低电平，如下图：




③ 分辨是引导码，还是连发码：

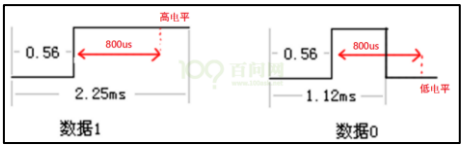
在 9ms 的低电平之后，判断高电平持续时间，引导码的高电平维持时间是 4.5ms，连发码的高电平维持时间是 2.25ms。

发现是连发码时，直接结束译码。

发现是引导码时，还得继续接收 32 位数据。

④ 接收数据：

关键在于如何得到一位数据，看看下图：




先等待低电平结束，一直等到出现高电平；然后延时 800us，读取 GPIO 值：这就是该位的数据值。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/20_IRDA 红外遥控器模块/gpio_irda”目录下。

## 17.3.3 编写 GPIO 函数

首先是 GPIO 的控制函数，我们可以设置 GPIO 引脚输出高低电平，也可以设置 GPIO 为输入，读取它的电平。

如果将 IRDA 模块插在扩展板的 GPIO0 上，对应的 GPIO 即为 GPIO4_19。

GPIO 函数在 irda.c 中，代码如下：

```
//设置引脚输出电平
static void set_gpio_value(int value)
{
    /* IRDA GPIO4_IO19 */
    GPIO4->GDIR |= (1<<19);

   if (1 == value)
       GPIO4->DR |= (1<<19);
   else
       GPIO4->DR &= ~(1<<19);
}

//读取引脚电平
static int get_gpio_value(void)
{
    GPIO4->GDIR &= ~(1<<19);

    if((GPIO4->DR >> 19 ) & 0x1)
        return 1;
    else
        return 0;
}
```

## 17.3.4 高精度的延时函数

在前面讲解 GPT 定时器时，就实现了这个函数，代码在 timer.c 里：

```
//精确延时
void delay_us(unsigned int n)
{
  gpt2_chan1_delay_us(n);
}

void delay_ms(unsigned int n)
{
    for(; n>0; n--)
        delay_us(1000);
}
```

### 17.3.5 接收起始信号

首先获取开始信号，即一个长达 9ms 低电平。只要这 9ms 中出现高电平，那就表示出错了。

代码在 irda.c 中：

```
//接收判断开始信号
static int irda_start(void)
{
    int i;
    int start_flag = 1;

    while(get_gpio_value());//一直等到有低电平

    for(i=0; i<9; i++)  //循环十次，检测每个 800 微秒内(共 9ms 内)如果出现高电平就退出解码
程序
    {
        delay_us(800);
        if (get_gpio_value() == 1)  //出现高电平，错误
        {
            start_flag = 0;
            break;
        }
    }

    return start_flag;
}
```

### 17.3.6 判断是引导码还是连发码

检测完开始信号后，判断是引导码还是连发码。两者区别就是引导码的高电平持续 4.5ms，而连发码持续 2.25ms，即延时大于 2.25ms 后检测引脚电平。

代码在 irda.c 中：

```
//判断开始的是引导码还是连发码
static int irda_mode(void)
{
    int mode_flag = 1;

    while(0 == get_gpio_value());//一直等到前面的 9ms 完

    delay_ms(2);//2.5ms
    delay_us(500);

    if(get_gpio_value())
        mode_flag = 1; //是引导码
    else
        mode_flag = 0; //是连发码

    while(1 == get_gpio_value()); //一直待整个判断周期结束

    return mode_flag;
}
```

### 17.3.7 接收 1 位数据

就 1 位数据而言，无论它是 1 还是 0，都是先收到一个 0.56ms 的低脉冲，接着不同宽度的高脉冲。

我们只需要得到高脉冲那刻延时 800us(大于 0.56ms 即可)后，再次读取电平，如果为高就是 1，反之为 0。

代码在 irda.c 中：

```
//读 1bit
static int read_byte(void)
{
    while(0 == get_gpio_value()); //一直等到高电平，即过掉 0.56ms 的共同低电平
    delay_us(800);

    if(get_gpio_value())
    {
         while(1 == get_gpio_value());
        return 1;
    }
    else
     {
         while(1 == get_gpio_value());
        return 0;
     }
}
```

## 17.3.8 解码函数

value_show 函数用来解析红外数据，并把它打印出来。代码在 irda.c 中，如下：

```
//接收显示
void value_show(void)
{
    int start_flag, mode_flag, i, j;
    unsigned char data;

    set_gpio_value(1);

    while(1)
    {
        start_flag = irda_start();     //判断是否有开始信号

        if (start_flag)
            mode_flag = irda_mode();   //再判断是何种码

        if (start_flag && mode_flag)
        {
            for(i=0; i<4; i++)         //连续接收四个字节
            {
                for(j=0; j<8; j++)
                {
                    data >>= 1;

                    if (read_byte())
                        data |= 0x80;
                }

                result[i] = data;
            }

            printf("addr code: %d \n\r", result[0]);
```

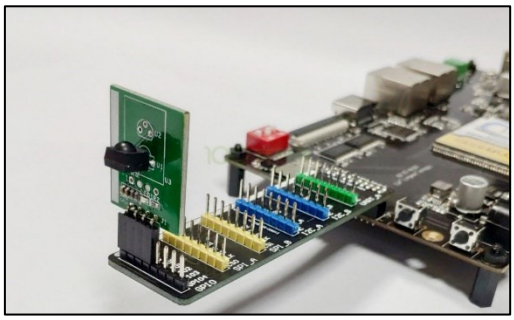
```
            printf("addr anti-code: %d \n\r", result[1]);
            printf("data code: %d \n\r", result[2]);
            printf("data anti-code: %d \n\r", result[3]);

            printf("\n\r");
        }
        else if ((start_flag && !mode_flag))
        {
            printf("addr code: %d \n\r", result[0]);
            printf("addr anti-code: %d \n\r", result[1]);
            printf("data code: %d \n\r", result[2]);
            printf("data anti-code: %d \n\r", result[3]);

            printf("\n\r");
        }
    }
}
```

## 17.4 IRDA 红外遥控模块测试

IMX6ULL 先断电，按下图所示，将模块插在扩展板的 GPIO0，将扩展板插在底板上。

注意：为了防止用户接错方向，模块和扩展板都有一条长白线，连接时需要模块上的白线和扩展板的白线在同一侧。

然后准备好配套的红外遥控器，如果是第一次使用红外遥控器，要先取出电池上的隔离薄膜。



编译、运行程序，打开串口观察，运行效果如下：

```
hello world
IRDA
addr code: 0
addr anti-code: 255
data code: 69
data anti-code: 186

addr code: 0
addr anti-code: 255
data code: 22
data anti-code: 233

addr code: 0
addr anti-code: 255
data code: 25
data anti-code: 230

addr code: 0
addr anti-code: 255
data code: 25
data anti-code: 230
```

# 第18章 DHT11 温湿度模块

模块原理图及资料：网盘开发板配套资料“05_Hardware（原理图)/Extend_modules/dht11.zip”。

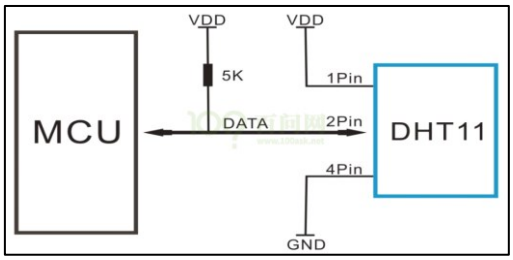
## 18.1 DHT11 简介

DHT11 是一款可测量温度和湿度的传感器。比如市面上一些空气加湿器，会测量空气中湿度，再根据测量结果决定是否继续加湿。

DHT11 数字温湿度传感器是一款含有已校准数字信号输出的温湿度复合传感器，具有超小体积、极低功耗的特点，使用单根总线与主机进行双向的串行数据传输。DHT11 测量温度的精度为±2℃，检测范围为-20℃-60℃。湿度的精度为±5%RH，检测范围为 5%RH-95%RH，常用于对精度和实时性要求不高的温湿度测量场合。

## 18.2 DHT11 模块硬件设计

和 IRDA 模块的电路基本一致，主机通过一条数据线与 DH11 连接，主机通过这条线发命令给 DHT11，DHT11 再通过这条线把数据发送给主机。

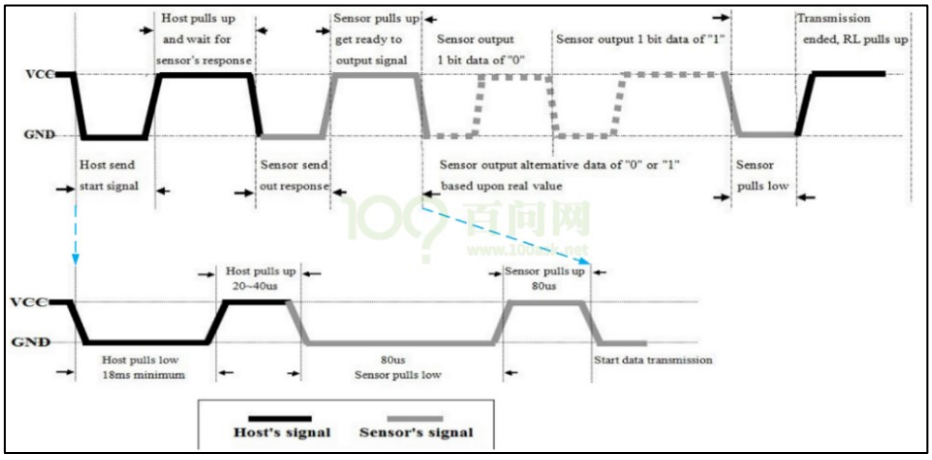



## 18.3 DHT11 模块软件设计

DHT11 的硬件电路比较简单，核心要点就是 主机发给 DHT11 的命令格式和 DHT11 返回的数据格式。

### 18.3.1 DHT11 通信协议

与 IRDA 的通信不同，需要先发一个开始信号给 DHT11，才能接收数据。下图为一次完整的传输示例：



其中深黑色信号表示由主机驱动，即主机向 DHT11 发信号，浅灰色信号表示 DHT11 驱动，即 DHT11 发向主机发信号。

- 当主机没有与 DHT11 通信时，总线处于空闲状态，此时总线电平由于上拉电阻的作用处于高电平。

- 当主机与 DHT11 正在通信时，总线处于通信状态，一次完整的通信过程如下：

a) 主机将对应的 GPIO 管脚配置为输出，准备向 DHT11 发送数据；

b) 主机发送一个开始信号：

开始信号 = 一个低脉冲 + 一个高脉冲。低脉冲至少持续 18ms，高脉冲持续 20-40us。

c) 主机将对应的 GPIO 管脚配置为输入，准备接受 DHT11 传来的数据，这时信号由上拉电阻拉高；

d) DHT11 发出响应信号：

响应信号 = 一个低脉冲 + 一个高脉冲。低脉冲持续 80us，高脉冲持续 80us。

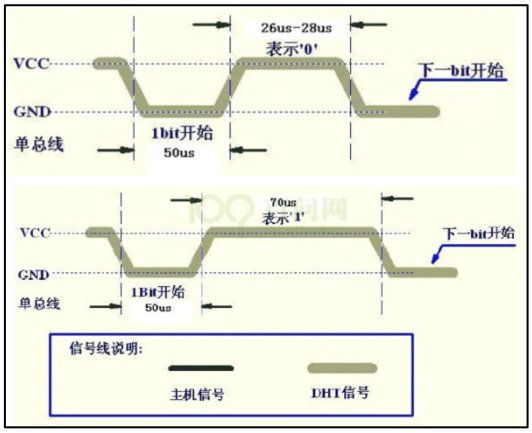
e) DHT11 发出数据信号：

- 数据为 0 的一位信号 = 一个低脉冲 + 一个高脉冲。低脉冲持续 50us，高脉冲持续 26～28us。

- 数据为 1 的一位信号 = 一个低脉冲 + 一个高脉冲。低脉冲持续 50us，高脉冲持续 70us。

f) DHT11 发出结束信号：

最后 1bit 数据传送完毕后，DHT11 拉低总线 50us，然后释放总线，总线由上拉电阻拉高进入空闲状态。




## 18.3.2 数据格式

数据格式：

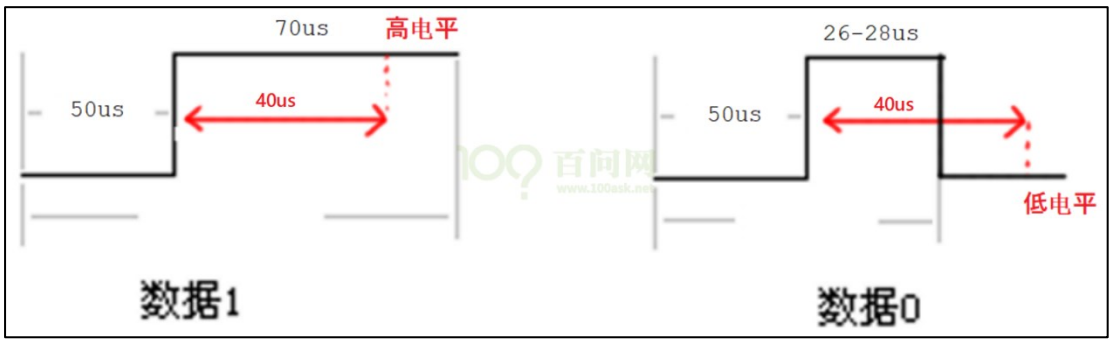
8bit 湿度整数数据+8bit 湿度小数数据
+8bi 温度整数数据+8bit 温度小数数据
+8bit 校验和。

数据传送正确时，校验和等于“8bit 湿度整数数据+8bit 湿度小数数据+8bi 温度整数数据+8bit 温度小数数据”所得结果的末 8 位。

## 18.3.3 编程思路

有了上述基础知识后就可以开始编写程序了。编程思路如下：

① 设置好 GPIO；
② 主机把 GPIO 设置为输出引脚，发送开始信号，然后把 GPIO 设置为输入引脚；
③ 主机判断是否收到 DHT11 的回应信号；
④ 接收到回应信号后，开始读取数据；
关键在于如何得到一位数据，看看下图：




先等待低电平结束，一直等到出现高电平；然后延时 40us，读取 GPIO 值：这就是该位的数据值。
⑤ 接收完数据后，校验、解析。
代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/21_DHT11 温湿度模块/dht11”目录下。

## 18.3.4 编写 GPIO 函数

① 初始化 GPIO
本次实验使用 GPIO4_19 管脚，首先使能 GPIO4 时钟，再配置 IOMUXC_SW_MUX_CTL_PAD_CSI_VSYNC 寄存器设置 GPIO4_19 用作 GPIO 作用。代码在 dht11.c 中，如下：

```
static void dht11_gpio_init(void)
{
    unsigned int val;

    CCM_CCGR3                               = (volatile unsigned int *)(0x20C4074);
    IOMUXC_SW_MUX_CTL_PAD_CSI_VSYNC         = (volatile unsigned int *)(0x20E01DC);
    GPIO4_GDIR                              = (volatile unsigned int *)(0x20A8000 +
0x4);
    GPIO4_DR                            = (volatile unsigned int *)(0x20A8000);

    /* 使能 GPIO4
     * set CCM to enable GPIO4
     * CCM_CCGR3[CG6] 0x20C4074
     * bit[13:12] = 0b11
     */
    *CCM_CCGR3 |= (3<<12);

    /* 设置 GPIO4_IO19 用于 GPIO
     * set IOMUXC_SW_MUX_CTL_PAD_CSI_VSYNC
     *      to configure GPIO4_IO19 as GPIO
     * IOMUXC_SW_MUX_CTL_PAD_CSI_VSYNC  0x20E01DC
     * bit[3:0] = 0b0101 alt5
     */
    val = *IOMUXC_SW_MUX_CTL_PAD_CSI_VSYNC;
    val &= ~(0xf);
```

```
    val |= (5);
    *IOMUXC_SW_MUX_CTL_PAD_CSI_VSYNC = val;
}
```

② 配置 GPIO，实现输入、输出功能

在初始化 GPIO4_19 管脚后，设置 GPIO4_GDIR 的 bit[19]为 0 或 1，设置为输入或输出引脚。

代码在 dht11.c 中，如下：

```
static void dht11_gpio_as_input(void)
{
    /*
     * 设置 GPIO4_IO19 作为 input 引脚
     * set GPIO4_GDIR to configure GPIO4_IO19 as input
     * GPIO4_GDIR  0x20A8000
     * bit[19] = 0b0
     */
     *GPIO4_GDIR &= ~(1<<19);
}

static void dht11_gpio_as_output(void)
{
    /*
     * 设置 GPIO4_IO19 作为 output 引脚
     * set GPIO4_GDIR to configure GPIO4_IO19 as output
     * GPIO4_GDIR  0x20A8000
     * bit[19] = 0b1
     */
     *GPIO4_GDIR |= (1<<19);
}
```

③ 设置输出电平或读取管脚数据

在设置管脚为输入时，通过读取 GPIO4_DR 的 bit[19]，可以判断此时管脚电平状态。

在设置管脚为输出时，通过设置 GPIO4_DR 的 bit[19]，可以设置管脚输出电平为高或低。

代码在 dht11.c 中，如下：

```
static void dht11_data_set(int val)
{
    if(val)
        *GPIO4_DR |= (1<<19);
    else
        *GPIO4_DR &= ~(1<<19);
}

static int dht11_data_get(void)
{
    if((*GPIO4_DR>>19) & 0x1)
        return 1;
    else
        return 0;
}
```

## 18.3.5 编写 DHT11 温湿度模块初始化函数

DHT11 模块上电后，要等待 1s，以越过不稳定状态，在此期间无需发送命令。设置 GPIO，让它输出高电平。

代码在 dht11.c 中，如下：

```
static void dht11_init(void)
{
    dht11_gpio_as_output();
    dht11_data_set(1);
    gpt2_chan1_delay_us(2000000);
}
```

### 18.3.6 编写 DHT11 开始信号函数

在管脚为输出管脚前提下，输出低电平并延时至少 18ms，表示开始信号。
代码在 dht11.c 中，如下：

```
static void dht11_start(void)
{
    dht11_data_set(0);
    gpt2_chan1_delay_us(20000);
}
```

### 18.3.7 编写等待响应、结束信号函数

在管脚为输入管脚前提下，等待高或低电平一段时间，读出数据时会调用此函数。可以用来等待高电半、等待低电平，并传入超时值。
代码在 dht11.c 中，如下：

```
static int dht11_wait_for_val(int val, int timeout_us)
{
    while(timeout_us--)
    {
        if(dht11_data_get() == val)
            return 0;
        gpt2_chan1_delay_us(1);
    }
    return -1;
}
```

### 18.3.8 编写读取一个字节函数

用 dht11_recv_byte 函数读取一字节，它的核心是如何读到 1 位数据：
① 第 196 行，等待数据信号中的高电平；
② 第 201 行，延时 40us；
③ 第 203 行，读取 1 位数据。
代码在 dht11.c 中，如下：

```
189 static unsigned char dht11_recv_byte(void)
190 {
191     int i;
192     int data = 0;
193
194     for (i = 0; i < 8; i++)
195     {
196             if (dht11_wait_for_val(1, 1000))
197             {
198                     printf("dht11 wait for high data err!\n\r");
199                     return -1;
200             }
201             gpt2_chan1_delay_us(40);
202             data <<= 1;                             //数据左移一位
```

```
203             if (dht11_data_get() == 1)      //如果引脚上是高电平，数据置 1
204                     data |= 1;                              //否则数据左移自动补 0
205
206             if (dht11_wait_for_val(0, 1000))
207             {
208                     printf("dht11 wait for low data err!\n\r");
209                     return -1;
210             }
211     }
212
213      return data;
214 }
```

## 18.3.9 综合起来，编写读取数据的函数

在读取 dht11 数据时：

① 先初始化 GPIO；

② 然后初始化 dht11；

③ 发送开始信号：拉高等待 20~40us 后，GPIO 管脚设置为输入，准备接收 dht11 数据。

④ 读出 5byte 数据：把前 4byte 数据加起来判断与第 5byte 数据是否相等，如果相等，校验通过。

⑤ 打印。

代码在 dht11.c 中，如下：

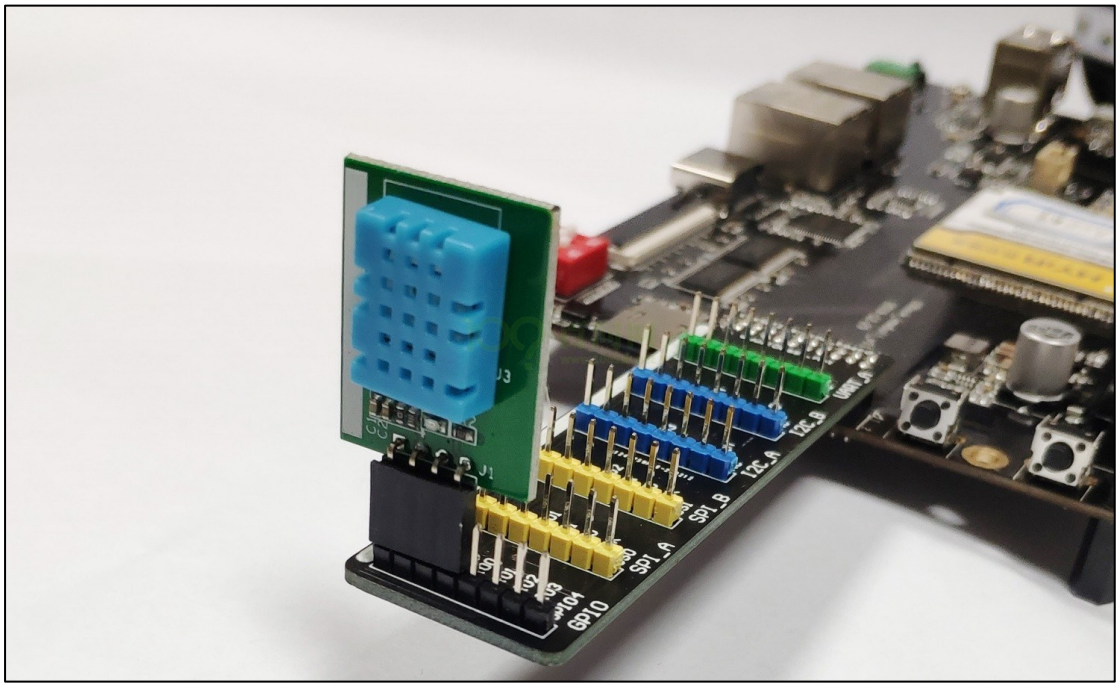
```
void dht11_data_read(void)
{
    unsigned char val[5],tmp;
    dht11_gpio_init();                          //初始化 GPIO 管脚
    dht11_init();                               //初始化 dht11
    dht11_start();                              //发出开始信号
    dht11_data_set(1);                          //主机拉高并延时 20~40us
    gpt2_chan1_delay_us(40);
    dht11_gpio_as_input();
    if(!dht11_data_get())                   //判断是否接收到 dht11 响应信号
    {
        while(!dht11_data_get());           //等待响应信号结束
        while(dht11_data_get());            //dht11 拉高 80us，等待 80us
        val[0] = dht11_recv_byte();             //读取 8bit 湿度整数数据
        val[1] = dht11_recv_byte();             //读取 8bit 湿度小数数据
        val[2] = dht11_recv_byte();             //读取 8bit 温度整数数据
        val[3] = dht11_recv_byte();             //读取 8bit 温度小数数据
        val[4] = dht11_recv_byte();             //读取 8bit 校验和
        tmp = val[0] + val[1] + val[2] + val[3];
        if(tmp==val[4])                         //校验通过，通过串口打印数据
        {
            printf("Humidity    = %d.%d\n\r", val[0], val[1]);
            printf("Temperature = %d.%d\n\r", val[2], val[3]);
        }
    }
}
```

# 18.4 DHT11 模块测试

IMX6ULL 先断电，按下图所示，将模块插在扩展板的 GPIO0，将扩展板插在

底板上。

注意：为了防止用户接错方向，模块和扩展板都有一条长白线，连接时需要模块上的白线和扩展板的白线在同一侧。




编译、运行程序，打开串口观察，运行效果如下：

```
Humidity    = 55.0
Temperature = 23.3

Humidity    = 54.0
Temperature = 23.2

Humidity    = 54.0
Temperature = 23.3

Humidity    = 54.0
Temperature = 23.1
```

# 第19章 DS18B20 温度模块

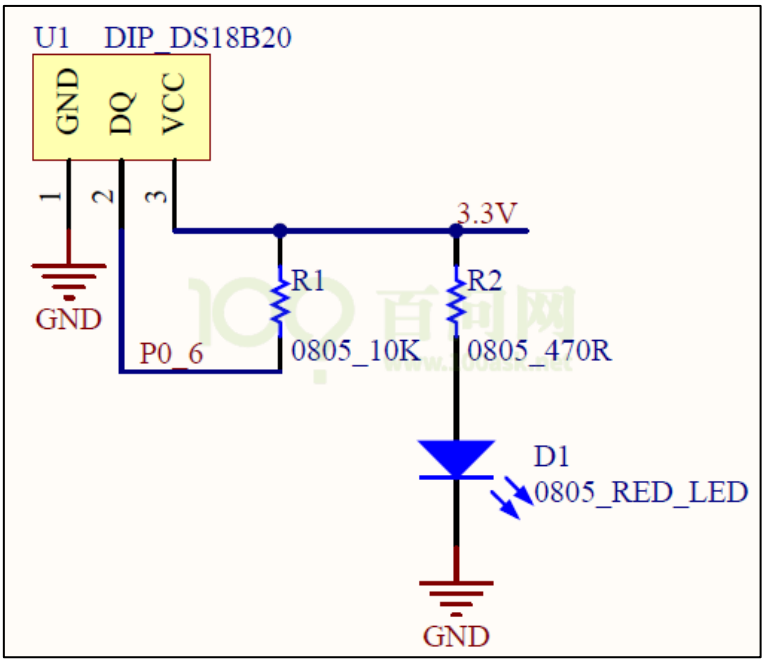
模块原理图及资料：网盘开发板配套资料“05_Hardware（原理图)/Extend_modules/ds18b20.zip”。

## 19.1 DS18B20 简介

DS18B20 温度传感器具有线路简单、体积小的特点，用来测量温度非常简单，在一根通信线上可以挂载多个 DS18B20 温度传感器。用户可以通过编程实现 9~12 位的温度读数，每个 DS18B20 有唯一的 64 位序列号，保存在 rom 中，因此一条总线上可以挂载多个 DS18B20。

## 19.2 DS18B20 模块硬件设计

DS18B20 也使用的是“1-Wire 单总线”，只通过一条数据线传输数据，既要控制器发送数据给芯片，又要通过芯片发送数据给控制器，是双向传输数据。DS18B20 的硬件设计电路与前面的 DHT11 基本一致，原理图如下：




## 19.3 DS18B20 模块软件设计

### 19.3.1 存储器介绍

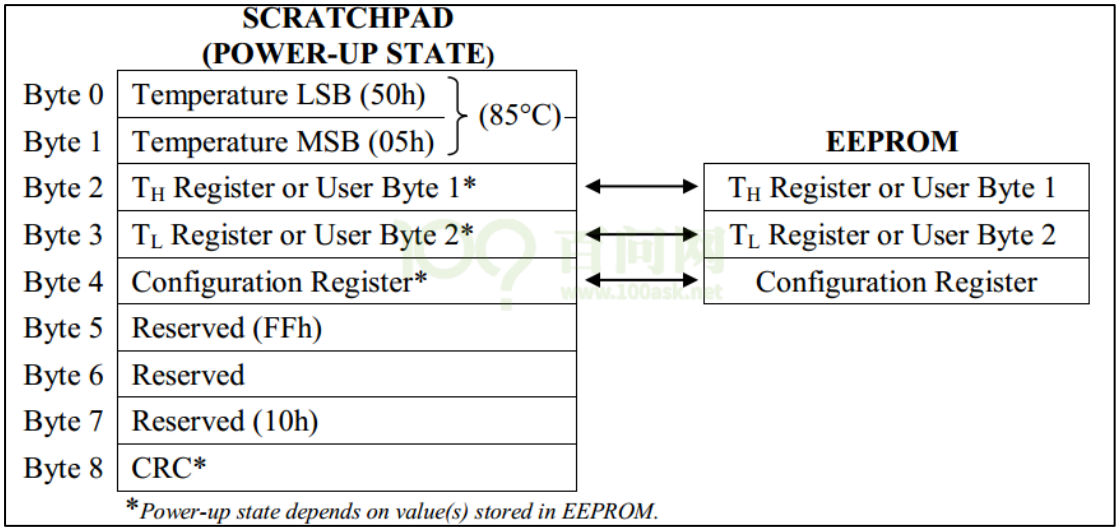
DS18B20 内部有个 64 位只读存储器(ROM)和 64 位配置存储器(SCRATCHP)。

64 位只读存储器（ROM）包含序列号等，具体格式如下图：

| 8-BIT CRC | | 48-BIT SERIAL NUMBER | | 8-BIT FAMILY CODE (28h) | |
|---|---|---|---|---|---|
| MSB | LSB | MSB | LSB | MSB | LSB |

低八位用于 CRC 校验，中间 48 位是 DS18B20 唯一序列号，高八位是该系列产品系列号(固定为 28h)。因此，根据每个 DS18B20 唯一的序列号，可以实现一条总线上可以挂载多个 DS18B20 时，获取指定 DS18B20 的温度信息。

64 位配置存储器（SCRATCHP）由 9 个 Byte 组成，包含温度数据、配置信息等，具体格式如下图：




- Byte[0:1]：温度值。也就是当我们发出一个测量温度的命令之后，还需要发送一个读内存的命令才能把温度值读取出来。

- Byte[2:3]：TL 是低温阈值设置，TH 是高温阈值设置。当温度低于/超过阈值，就会报警。 TL、TH 存储在 EEPROM 中，数据在掉电时不会丢失；

- Byte4：配置寄存器。用于配置温度精度为 9、10、11 或 12 位。配置寄存器也存储在 EEPROM 中，数据在掉电时不会丢失；

- Byte[5:7]：厂商预留；

- Byte[8]：CRC 校验码。

## 19.3.2 通信时序

① ① 初始化时序

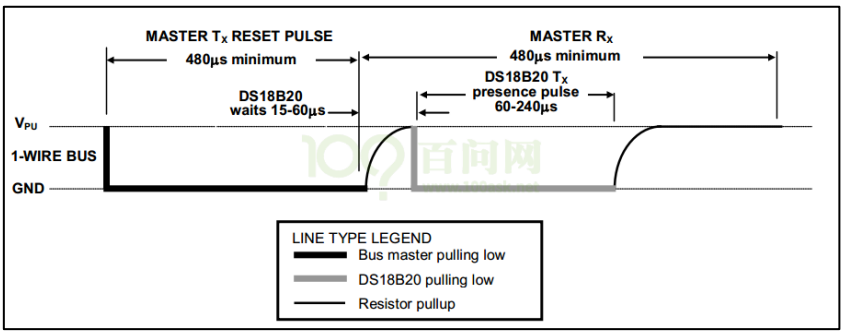
类似前面的 DHT11，主机要跟 DS18B20 通信，首先需要发出一个开始信号。
深黑色线表示由主机驱动信号，浅灰色线表示由 DS18B20 驱动信号。
最开始时引脚是高电平，想要开始传输信号，

a) 必须要拉低至少 480us，这是复位信号；

b) 然后拉高释放总线，等待 15~60us 之后，

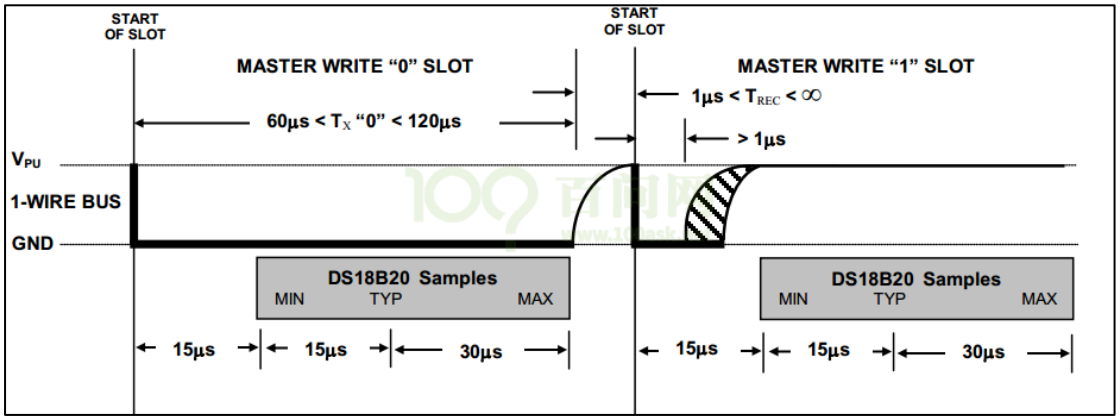
c) 如果 GPIO 上连有 DS18B20 芯片，它会拉低 60~240us。

如果主机在最后检查到 60～240us 的低脉冲，则表示 DS18B20 初始化成功。



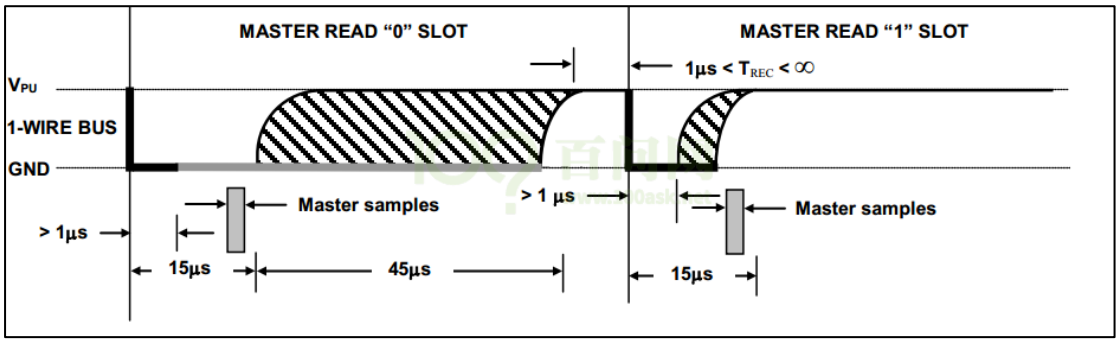
② 写时序

- 如果写 0，拉低至少 60us(写周期为 60-120us)即可；
- 如果写 1，先拉低至少 1us，然后拉高，整个写周期至少为 60us 即可。




③ 读时序

- 主机先拉低至少 1us，随后读取电平，如果为 0，即读到的数据是 0，如果为 1，即可读到的数据是 1。
- 整个过程必须在 15us 内完成，15us 后引脚都会被拉高。




## 19.3.3 常用命令

现在我们知道怎么发 1 位数据，收 1 位数据。发什么数据才能得到温度值，这需要用到“命令”。

DS18B20 中有两类命令：ROM 命令、功能命令，列表如下：

| ROM Commands | 命令名称 | 描述 |
|---|---|---|
| F0H | Search ROM | 搜索ROM 用于确定挂接在同一总线上DS18B20的个数，识别64位ROM地址 |
| 33H | Read ROM | 读ROM 读DS18B20芯片中的编码值，即64位ROM值 |
| 55H | Match ROM | 匹配ROM 发出此命令后，接着发出64位ROM编码，用于选中某个设备 |
| CCH | Skip ROM | 忽略ROM 表示后续发出的命令将会发给所有设备 如果总线上只有一个DS18B20，则特别适用此命令 |
| ECH | Alarm ROM | 警报搜索 执行此命令后，只有温度超过设定值上限或下限的芯片才会做出响应 |

| Function Commands | 命令名称 | 描述 |
|---|---|---|
| 44H | Convert Teamperature | 启动温度转换，注意不同精度需要不同转换时间，<br>结果存入内部RAM |
| 4EH | Write Scratchpad | 写内部RAM，可以写入3字节：TH，TL，配置值(用于选择精度)TH，TL可用于设置报警上下限，或给用户自己使用 |
| BEH | Read Scratchpad | 读整个内部RAM，9字节 |
| 48H | Copy Scratchpad | 把内部RAM中的TH、TL、配置值，复制给EEPROM |
| B8H | Recall EEPROM | 从EEPROM中把TH、TL、配置值，读到内部RAM |
| B4H | Read Power Supply | 分辨DS18B20的供电方式：用电源引脚供电，或从数据线偷电 |

## 19.3.4 怎么使用命令：流程图

DS18B20 芯片手册中有 ROM 命令、功能命令的流程图，先贴出来，下一小节再举例。

① ROM 命令流程图

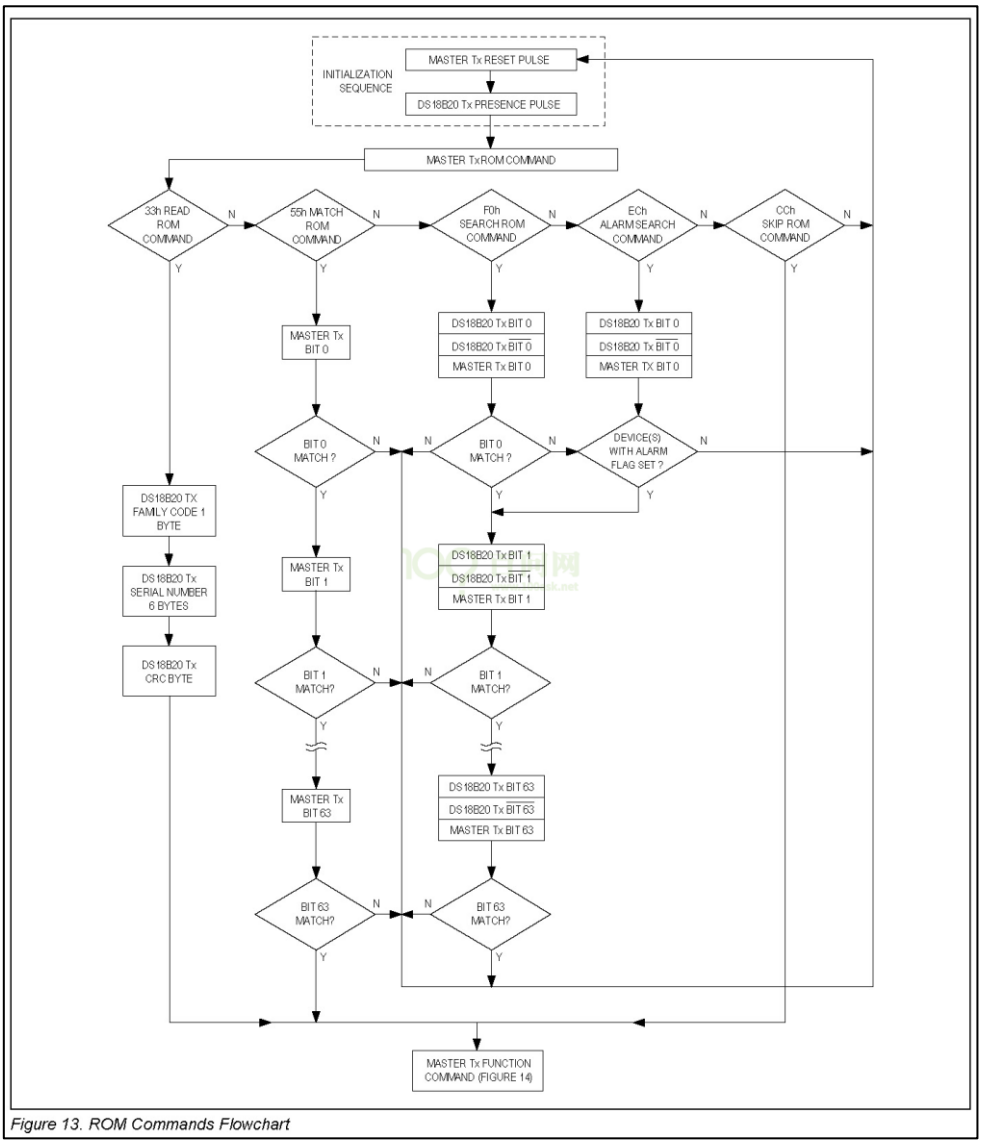



*Figure 13. ROM Commands Flowchart*

② 功能命令流程图

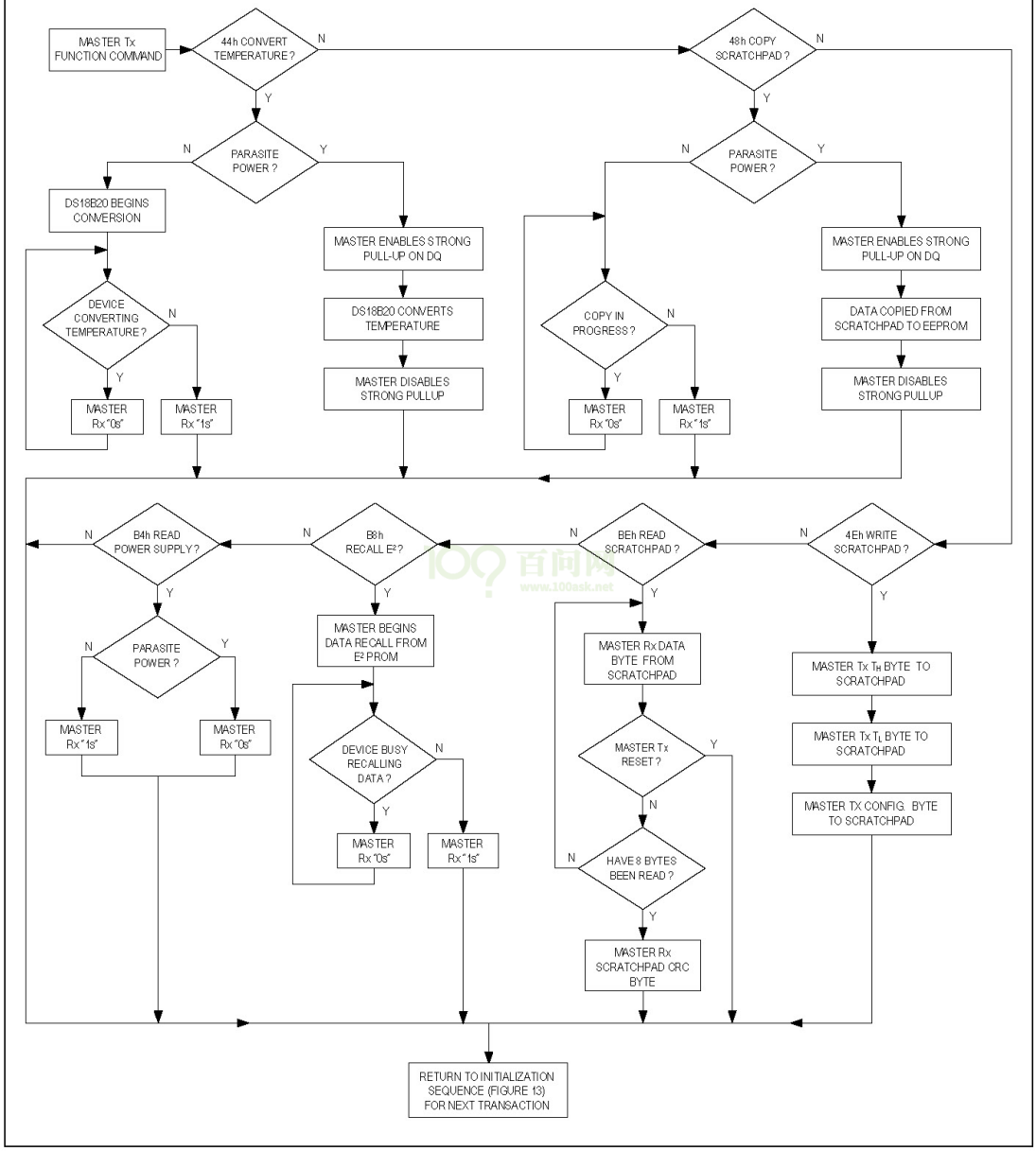



*Figure 14. DS18B20 Function Commands Flowchart*

## 19.3.5 命令示例 1：单个 DS18B20 温度转换

总线上只一个 DS18B20 设备时，根据下表发送命令、读取数据。因为只有一个 DS18B20，所以不需要选择设备，发出“Skip ROM”命令。然后发户“Convert T”命令启动温度转换；

等待温度转换成功后，要读数据前，也要发出“Skip ROM”命令。下表列得很清楚：

| 主机模式 | 数据 | 描述 |
|---|---|---|
| 发送 | 复位 | 主机发出复位脉冲 |
| 接收 | 回应 | 总线上可能有多个 DS18B20，它们都可以拉低信号，回应 |
| 发送 | CCh | 主机发出“Skip ROM”命令(忽略 ROM) |

| 发送 | 44h | 主机发出“Convert T”命令(启动温度转换) |
|---|---|---|
| 发送 | 保持高电平 | 主机使用强上拉保存数据线为高电平，至少 tCONV() |
| 发送 | 复位 | 主机发出复位脉冲 |
| 接收 | 回应 | 总线上可能有多个 DS18B20，它们都可以拉低信号，回应 |
| 发送 | CCh | 主机发出“Skip ROM”命令(忽略 ROM) |
| 发送 | BEh | 主机发出“Read Scratchpad”命令(读内存) |
| 接收 | 9 字节数据 | 主机读 9 字节数据 |

### 19.3.6 命令示例 2：指定 DS18B20 温度转换

总线上有多个 DS18B20 设备时，根据下表发送命令、读取数据。首先肯定是要选中指定设备：使用“Match ROM”命令发出 ROM Code 来选择中设备；然后发户“Convert T”命令启动温度转换；等待温度转换成功后，要读数据前，也要发出“Match ROM”命令、ROM Code。下表列得很清楚：

| 主机模式 | 数据 | 描述 |
|---|---|---|
| 发送 | 复位 | 主机发出复位脉冲 |
| 接收 | 回应 | 总线上可能有多个 DS18B20，它们都可以拉低信号，回应 |
| 发送 | 55h | 主机发出“Match ROM”命令(匹配 ROM) |
| 发送 | 64 位 ROM code | 主机发出想访问的 DS18B20 的“ROM Code” |
| 发送 | 44h | 主机发出“Convert T”命令(启动温度转换) |
| 发送 | 保持高电平 | 主机使用强上拉保存数据线为高电平，至少 tCONV() |
| 发送 | 复位 | 主机发出复位脉冲 |
| 接收 | 回应 | 总线上可能有多个 DS18B20，它们都可以拉低信号，回应 |
| 发送 | 55h | 主机发出“Match ROM”命令(匹配 ROM) |
| 发送 | 64 位 ROM code | 主机发出想访问的 DS18B20 的“ROM Code” |
| 发送 | BEh | 主机发出“Read Scratchpad”命令(读内存) |
| 接收 | 9 字节数据 | 主机读 9 字节数据 |

### 19.3.7 编程思路

按照流程图来编程即可：

① 实现复位函数；
② 实现等待回应的函数；
③ 实现发送 1 位数据的函数，进而实现发送 1 字节的函数；
④ 实现读取 1 位数据的函数，进而实现读取 1 字节的函数；
⑤ 按照流程图或是表格，发送、接收数据。

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/22_DS18B20 温度/ds18b20001_ds18b20_test”目录下。

### 19.3.8 编写 GPIO 函数

① 初始化 GPIO

本次实验使用 GPIO4_19 管脚，首先使能 GPIO4 时钟，再配置 IOMUXC_SW_MUX_CTL_PAD_CSI_VSYNC 寄存器设置 GPIO4_19 用作 GPIO 作用。代码在 ds18b20.c 中，如下：

```
static void ds18b20_gpio_init(void)
{
    unsigned int val;

    CCM_CCGR3                               = (volatile unsigned int *)(0x20C4074);
    IOMUXC_SW_MUX_CTL_PAD_CSI_VSYNC         = (volatile unsigned int *)(0x20E01DC);
    GPIO4_GDIR                              = (volatile unsigned int *)(0x20A8000 +
0x4);
    GPIO4_DR                            = (volatile unsigned int *)(0x20A8000);

    /* 使能 GPIO4
     * set CCM to enable GPIO4
     * CCM_CCGR3[CG6] 0x20C4074
     * bit[13:12] = 0b11
     */
    *CCM_CCGR3 |= (3<<12);

    /* 设置 GPIO4_IO19 用于 GPIO
     * set IOMUXC_SW_MUX_CTL_PAD_CSI_VSYNC
     *      to configure GPIO4_IO19 as GPIO
     * IOMUXC_SW_MUX_CTL_PAD_CSI_VSYNC  0x20E01DC
     * bit[3:0] = 0b0101 alt5
     */
    val = *IOMUXC_SW_MUX_CTL_PAD_CSI_VSYNC;
    val &= ~(0xf);
    val |= (5);
    *IOMUXC_SW_MUX_CTL_PAD_CSI_VSYNC = val;
}
```

② 配置 GPIO，实现输入、输出功能

在初始化 GPIO4_19 管脚后，设置 GPIO4_GDIR 的 bit[19]为 0 或 1，设置为输入或输出引脚。

代码在 ds18b20.c 中，如下：

```
static void ds18b20_gpio_as_input(void)
{
    /*
     * 设置 GPIO4_IO19 作为 input 引脚
     * set GPIO4_GDIR to configure GPIO4_IO19 as input
     * GPIO4_GDIR  0x20A8000
     * bit[19] = 0b0
     */
     *GPIO4_GDIR &= ~(1<<19);
}
static void ds18b20_gpio_as_output(void)
{
    /*
     * 设置 GPIO4_IO19 作为 output 引脚
     * set GPIO4_GDIR to configure GPIO4_IO19 as output
     * GPIO4_GDIR  0x20A8000
     * bit[19] = 0b1
```

```
     */
     *GPIO4_GDIR |= (1<<19);
}
```

③ 设置输出电平或读取管脚数据

在设置管脚为输入时，通过读取 GPIO4_DR 的 bit[19]，可以判断此时管脚电平状态。

在设置管脚为输出时，通过设置 GPIO4_DR 的 bit[19]，可以设置管脚输出电平为高或低。

代码在 ds18b20.c 中，如下：

```
static void ds18b20_data_set(int val)
{
    if(val)
        *GPIO4_DR |= (1<<19);
    else
        *GPIO4_DR &= ~(1<<19);
}

static int ds18b20_data_get(void)
{
    if((*GPIO4_DR>>19) & 0x1)
        return 1;
    else
        return 0;
}
```

## 19.3.9 编写时间精确的电平输出函数

在 DS18B20 的时序中，输出高低电平的时长要很精确，这要用到高精度的连时函数。

代码在 ds18b20.c 中，如下：

```
static void ds18b20_data_set_val_for_time(int val, int us)
{
    ds18b20_gpio_as_output();
    ds18b20_data_set(val);
    gpt2_chan1_delay_us(us);
}
```

## 19.3.10 编写 DS18B20 初始化函数

最开始时 GPIO 是高电平，要开始访问 DS18B20，

① 必须要拉低至少 480us，然后释放总线；

② 等待 15~60us 之后，DS18B20 会回应主机：把这条线拉低 60~240us。

代码在 ds18b20.c 中，如下：

```
static int ds18b20_init(void)
{
    unsigned int val;
    ds18b20_data_set_val_for_time(1,6);
    ds18b20_data_set_val_for_time(0,500);
    ds18b20_gpio_as_input();
    gpt2_chan1_delay_us(80);

    val = ds18b20_data_get();
    gpt2_chan1_delay_us(250);
```

```
    return val;
}
```

## 19.3.11 编写读写 1 位数据函数

不论是写 0 还是写 1，时序都是大于 60us。

- 写 0 时，拉低总线维持 60us 以上；
- 写 1 时，信号线拉低 1us 时间，然后马上释放总线。

当总线控制器把数据线从高电平拉到低电平时，就产生了写信号，DS18B20 会在 15~60us 对数据线采样。

写 1 位数据的代码在 ds18b20.c 中，如下：

```
static void ds18b20_write_bit(int val)
{
    if (!val)
    {
        ds18b20_data_set_val_for_time(0, 60);
        ds18b20_gpio_as_input();
        gpt2_chan1_delay_us(2);
    }
    else
    {
        ds18b20_data_set_val_for_time(0, 2);
        ds18b20_gpio_as_input();
        gpt2_chan1_delay_us(60);
    }
}
```

一个读周期至少是 60us，每位的间隔也是 1us。

主机把数据线从高电平拉到低电平时，产生读信号，数据线至少维持 1us 低电平，然后释放总线；在 15~60us 内读取管脚电平，高电平表示 1，否则表示 0。

注意：主机把数据线从高电平拉到低电平时，就向 DS18B20 发出了写信号或读信号，是写信号还是读信号由 DS18B20 的状态决定。

读 1 位数据的代码在 ds18b20.c 中，如下：

```
static int ds18b20_read_bit(void)
{
    int val;

    ds18b20_data_set(1);
    ds18b20_data_set_val_for_time(0, 2);
    ds18b20_gpio_as_input();
    gpt2_chan1_delay_us(10);
    val = ds18b20_data_get();
    gpt2_chan1_delay_us(50);
    return val;
}
```

## 19.3.12 编写读写 1 字节数据的函数

- 写 1 字节时，调用写 1 位数据的写函数，注意移位，先发送最低位。
- 读 1 字节时，调用读 1 位数据的读函数，注意移位，先读到最低位。

代码在 ds18b20.c 中，如下：

```
static void ds18b20_write_byte(unsigned char data)
```

```
{
    /*
    优先传输最低位
    */
    int i;
    for (i = 0; i < 8; i++)
    {
        ds18b20_write_bit(data & (1<<i));
    }
}

static unsigned char ds18b20_read_byte(void)
{
    int i;
    unsigned char data = 0;

    for (i = 0; i < 8; i++)
    {
        if (ds18b20_read_bit() == 1)
            data |= (1<<i);
    }

    return data;
}
```

## 19.3.13 编写读温度数据函数

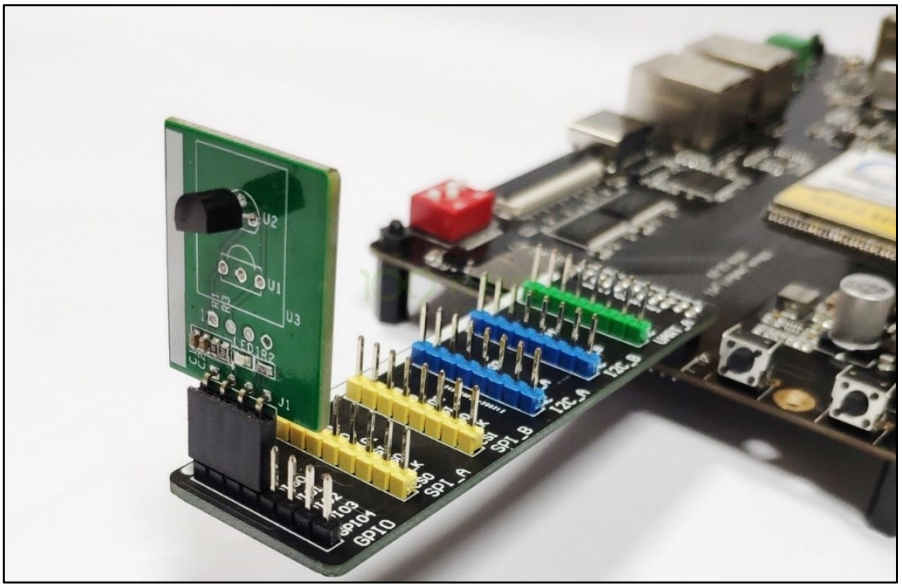
此函数实现了最简单的读 DS18B20 温度数据，读出数据后打印出来。代码在 ds18b20.c 中，如下：

```
int ds18b20_data_read(void)
{
    unsigned char tempL=0,tempH=0;
    unsigned int integer;
    unsigned char decimal1,decimal2,decimal;

    ds18b20_gpio_init();                                //GPIO 管脚初始化

    if(ds18b20_init() != 0)                             //ds18b20 初始化
    {
        printf("ds18b20_initialization err!\n\r");
        return -1;
    }
    ds18b20_write_byte(0xcc);                           //忽略 rom 指令，直接使用功能指令
    ds18b20_write_byte(0x44);                           //温度转换指令
    gpt2_chan1_delay_us(1000000);                       //转换需要时间，延时 1s
    if(ds18b20_init() != 0)                             //ds18b20 初始化
    {
        printf("ds18b20_initialization err!\n\r");
        return -1;
    }
    ds18b20_write_byte(0xcc);                           //忽略 rom 指令，直接使用功能指令
    ds18b20_write_byte(0xbe);                           //读暂存器指令
    tempL = ds18b20_read_byte();                        //读温度低 8 位
    tempH = ds18b20_read_byte();                        //读温度高 8 位
    if(tempH>0x7f)                                      //最高位为 1 时温度是负
    {
        tempL    = ~tempL;                                  //补码转换，取反加一
```

```
        tempH    = ~tempH+1;
        integer  = tempL/16+tempH*16;                   //整数部分
        decimal1 = (tempL&0x0f)*10/16;                  //小数第一位
        decimal2 = (tempL&0x0f)*100/16%10;              //小数第二位
        decimal  = decimal1*10+decimal2;                //小数两位
        printf("temperature = %d.%d\n\r\n", integer, decimal);
    }
    integer  = tempL/16+tempH*16;                       //整数部分
    decimal1 = (tempL&0x0f)*10/16;                      //小数第一位
    decimal2 = (tempL&0x0f)*100/16%10;                  //小数第二位
    decimal  = decimal1*10+decimal2;                    //小数两位
    printf("temperature = %d.%d\n\r\n", integer, decimal);

    return 0;
}
```

## 19.4 DS18B20 模块测试

IMX6ULL 先断电，按下图所示，将模块插在扩展板的 GPIO0，将扩展板插在底板上。




**注意：为了防止用户接错方向，模块和扩展板都有一条长白线，连接时需要模块上的白线和扩展板的白线在同一侧。**

编译、运行程序，打开串口观察，运行效果如下：

```
temperature = 25.37

temperature = 25.43

temperature = 25.50

temperature = 25.50
```

# 第20章 SR501 人体红外模块

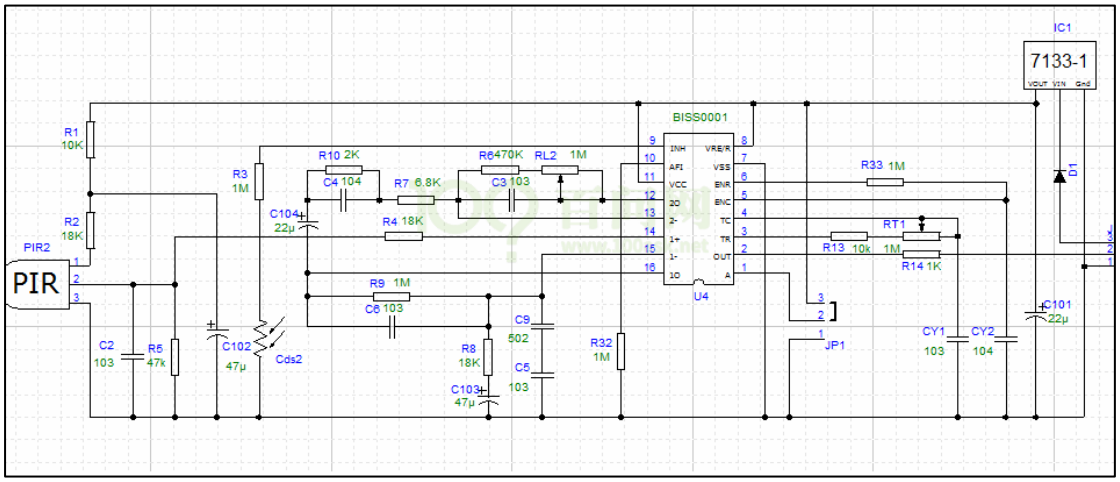
模块原理图及资料：网盘开发板配套资料“05_Hardware（原理图)/Extend_modules/人体红外感应.zip”。

## 20.1 人体红外模块简介

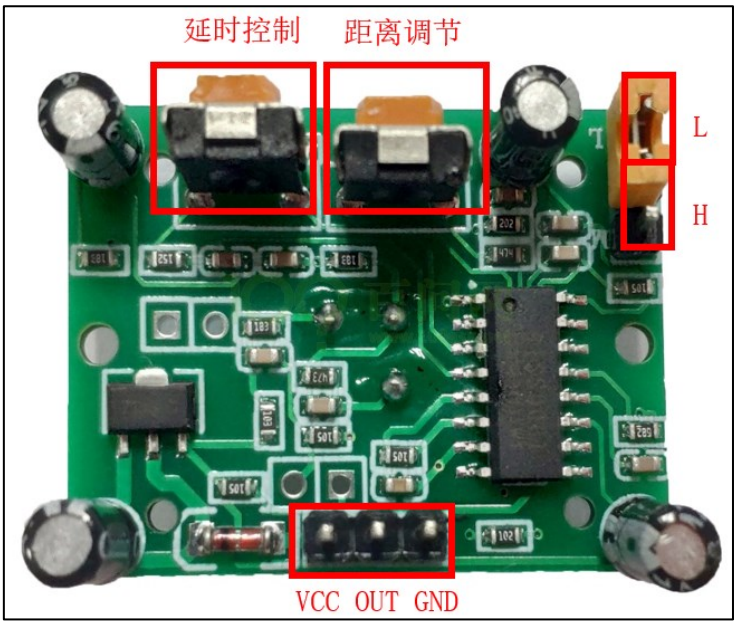
人体都有恒定的体温，一般在 37 度，所以会发出特定波长 10uM 左右的红外线，被动式红外探头就是靠探测人体发射的 10uM 左右的红外线而进行工作的。人体发射的 10uM 左右的红外线通过菲泥尔滤光片增强后聚集到红外感应源上。红外感应源通常采用热释电元件，这种元件在接收到人体红外辐射温度发生变化时就会失去电荷平衡，向外释放电荷，后续电路经检测处理后就能产生报警信号。

人体红外模块是一种能够检测人或动物发射的红外线而输出电信号的传感器。广泛应用于各种自动化控制装置中。比如常见的楼道自动开关、防盗报警等。如果有人在量程内运动，DO 引脚将会输出有效信号。

市面上人体红外模块有很多，但其外形和原理都差不多，如下是一个典型的人体红外模式原理图：




实物和使用方法如下图所示，可以设置探测距离、延迟控制等：



通过跳线来设置是否可以重复触发，默认为 L。其中 L 表示不可重复，H 表示可重复。含义如下：

① 不可重复触发方式：

感应到人体并输出高电平后，延时时间一结束，输出将自动从高电平变为低电平。

② 重复触发方式：

感应到人体后输出高电平后，在延时时间段内，如果有人体在其感应范围内活动，其输出将一直保持高电平，直到人离开后才延时将高电平变为低电平(感应模块检测到人体的每一次活动后会自动顺延一个延时时间段，并且以最后一次活动的时间为延时时间的起始点)。

可以通过电位器实现封锁时间和检测距离的调节：

① 调节检测距离：

即有效距离的远近。调节距离电位器顺时针旋转，感应距离增大（约 7 米）；反之，感应距离减小（约 3 米）。

② 封锁时间：

感应模块在每一次感应输出后(高电平变为低电平)，可以紧跟着设置一个封锁时间，在此时间段内感应器不接收任何感应信号。

此功能可以实现(感应输出时间和封锁时间)两者的间隔工作，可应用于间隔探测产品；同时此功能可有效抑制负载切换过程中产生的各种干扰。

调节延时电位器顺时针旋转，感应延时加长（约 300S），反之，感应延时减短（约 0.5S）。

## 20.2 SR501 人体红外模块软件设计

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/23_SR501 人体红外模块/gpio_sr501”目录下。

探测到人体时，红外模块会发生电平跳变，本程序中使用中断来进行检测。首先将引脚初始化为输入 GPIO，然后开启引脚沿触发类型中断。

代码是文件“main.c”中的 init_pins 函数，如下：

```
void init_pins(void)
{
    /* led pins */
    IOMUXC_SetPinMux(IOMUXC_SNVS_SNVS_TAMPER3_GPIO5_IO03, 0U);
    IOMUXC_SetPinConfig(IOMUXC_SNVS_SNVS_TAMPER3_GPIO5_IO03,
                        IOMUXC_SW_PAD_CTL_PAD_SRE_MASK |
                        IOMUXC_SW_PAD_CTL_PAD_DSE(1U) |
                        IOMUXC_SW_PAD_CTL_PAD_HYS_MASK);

    /* uart1 pins */
    IOMUXC_SetPinMux(IOMUXC_UART1_RX_DATA_UART1_RX, 0U);
    IOMUXC_SetPinConfig(IOMUXC_UART1_RX_DATA_UART1_RX,
                            IOMUXC_SW_PAD_CTL_PAD_DSE(2U) |
                            IOMUXC_SW_PAD_CTL_PAD_SPEED(2U) |
                            IOMUXC_SW_PAD_CTL_PAD_PKE_MASK);
    IOMUXC_SetPinMux(IOMUXC_UART1_TX_DATA_UART1_TX, 0U);
    IOMUXC_SetPinConfig(IOMUXC_UART1_TX_DATA_UART1_TX,
                            IOMUXC_SW_PAD_CTL_PAD_DSE(2U) |
                            IOMUXC_SW_PAD_CTL_PAD_SPEED(2U) |
                            IOMUXC_SW_PAD_CTL_PAD_PKE_MASK);
```

```
    /*  pins GPIO4_IO19 */
    IOMUXC_SetPinMux(IOMUXC_CSI_VSYNC_GPIO4_IO19, 0U);
    IOMUXC_SetPinConfig(IOMUXC_CSI_VSYNC_GPIO4_IO19,
                        IOMUXC_SW_PAD_CTL_PAD_DSE(6U) |
                        IOMUXC_SW_PAD_CTL_PAD_SPEED(2U) |
                        IOMUXC_SW_PAD_CTL_PAD_PKE_MASK |
                        IOMUXC_SW_PAD_CTL_PAD_HYS_MASK);
}
```

在初始化为 GPIO 引脚之后，需要设置中断触发方式，代码是文件“main.c”中的 key_irq_init 函数，如下：

```
void key_irq_init(void)
{
    /* if set detects any edge on the corresponding input signal*/
    GPIO4->EDGE_SEL |= (1 << 19);
    /* if set 1, unmasked, Interrupt n is enabled */
    GPIO4->IMR |= (1 << 19);
}
```

在中断处理函数中根据 GPIO 引脚电平的不同判断当前是否有人活动：

- 如果为高电平则表示为有效电平，有人在活动；
- 为低电平的话，则表示没有人活动。

相关的代码在程序文件“gic.c”的 handle_irq_c 中断处理函数中，如下：

```
void handle_irq_c(void)
{
    int nr;

    GIC_Type *gic = get_gic_base();
    /* The processor reads GICC_IAR to obtain the interrupt ID of the
     * signaled interrupt. This read acts as an acknowledge for the interrupt
     */
    nr = gic->C_IAR;
    printf("irq %d is happened\r\n", nr);

    switch(nr)
    {
        case GPIO4_Combined_16_31_IRQn:
        {
            /* read GPIO4_DR to get GPIO4_IO014 status*/
            if((GPIO4->DR >> 19 ) & 0x1)
            {
                printf("检测到人员活动\r\n");
                GPIO5->DR |= (1<<3); //led on
            }
            else
            {
                printf("没有检测到人员活动\r\n");
                GPIO5->DR &= ~(1<<3); //led off
            }
            /* write 1 to clear GPIO4_IO014 interrput status*/
            GPIO4->ISR |= (1 << 19);
            break;
        }

        default:
            break;
```

```
    }

    /* write GICC_EOIR inform the CPU interface that it has completed
     * the processing of the specified interrupt
     */
    gic->C_EOIR = nr;
}
```

SR501 由中断来驱动，main 函数可以在其他事，代码如下：

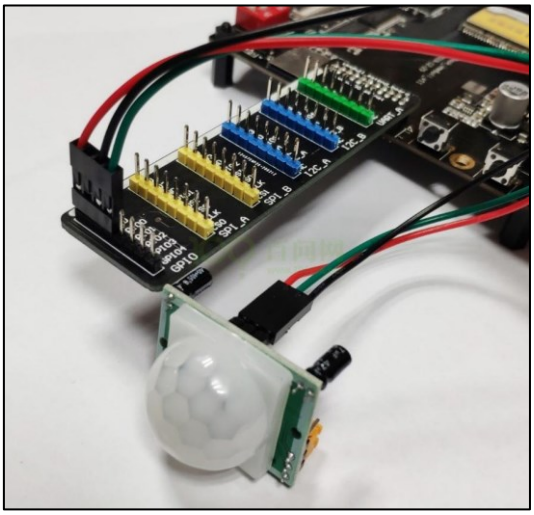
```
int  main()
{
    unsigned char c;

    while(1)
    {
        c = getchar();
        if (c == '\r')
        {
            putchar('\n');
        }

        if (c == '\n')
        {
            putchar('\r');
        }

        putchar(c);

    }

    return 0;
}
```

## 20.3 SR501 人体红外模块测试

IMX6ULL 先断电，按下图所示，将模块插在扩展板的 GPIO0，将扩展板插在底板上。

**注意：为了防止用户接错方向，模块和扩展板都有一条长白线，连接时需要模块上的白线和扩展板的白线在同一侧。**

**注意：注意 SR501 模块中红线、黑线、绿线的位置，如下图接线。**



编译、运行程序，打开串口观察：

```
hello world
irq 105 is happened
没有检测到人员活动
irq 105 is happened
检测到人员活动
irq 105 is happened
没有检测到人员活动
irq 105 is happened
检测到人员活动
irq 105 is happened
没有检测到人员活动
irq 105 is happened
检测到人员活动
irq 105 is happened
没有检测到人员活动
irq 105 is happened
检测到人员活动
irq 105 is happened
没有检测到人员活动
irq 105 is happened
检测到人员活动
```

可以看到，当有人经过的时候，串口会打印信息并且 LED 灯会熄灭。当人走后，过一段时间（即设置的延迟时间）之后，就会发现 LED 灯熄灭并且串口有打印信息。

具体打印店信息如下图所示（由于人体红外模块灵敏度设置的不同，观察到实际现象可能需要间隔一定时间，请根据具体来进行实验。建议先把延时时间逆时针选择改到最低，以方便测试）.

# 第21章 SR04 超声波测距模块

模块原理图及资料：网盘开发板配套资料“05_Hardware（原理图)/Extend_modules/超声波.zip”。

## 21.1 SR04 超声波简介

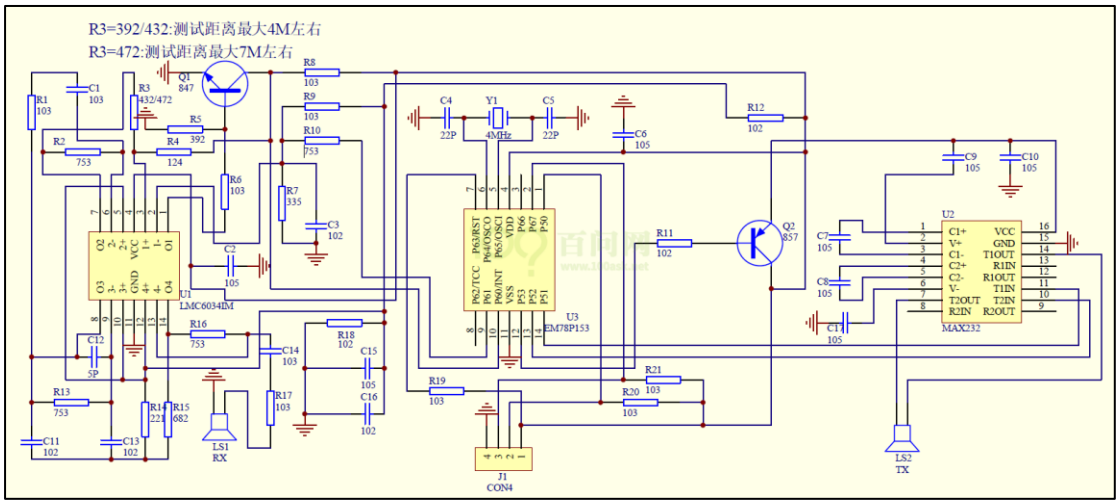
超声波测距模块是利用超声波来测距。模块先发送超声波，然后接收反射回来的超声波，由反射经历的时间和声音的传播速度 340m/s，计算得出距离。

SR04 是一款常见的超声波传感器，模块自动发送 8 个 40KHz 的方波，自动检测是否有信号返回，用户只需提供一个触发信号，随后检测回响信号的时间长短即可。

SR04 采用 5V 电压，静态电流小于 2mA，感应角度最大约 15 度，探测距离约 2cm-450cm。

## 21.2 SR04 超声波测距模块硬件设计

SR04 是市面上比较成熟模块，价格低廉，用户一般不用考虑重新设计硬件，直接购买使用即可，参考电路如下。



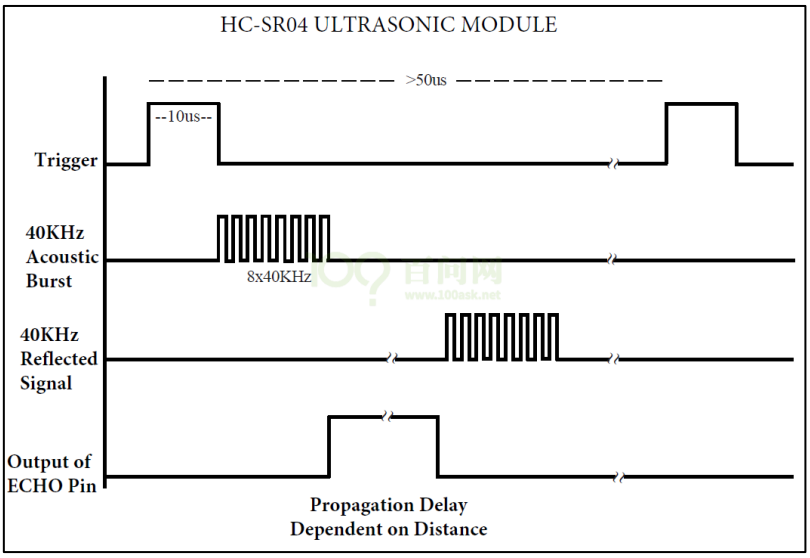

SR04 模块上面有四个引脚，分别为：VCC、Trig、Echo、GND。

- Trig 是脉冲触发引脚，即控制该脚让 SR04 模块开始发送超声波。
- Echo 是回响接收引脚，即 SR04 模块一旦接收到超声波的返回信号则输出回响信号，回响信号的脉冲宽度与所测距离成正比。

# 21.3 SR04 超声波测距模块软件设计

## 21.3.1 SR04 时序及编程思路

在开始写代码前，先简单介绍下思路，下图为超声波时序图。




要测距，需如下操作：

① 触发：向 Trig（脉冲触发引脚）发出一个大约 10us 的高电平。

② 发出超声波，接收反射信号：模块就自动发出 8 个 40Khz 的超声波，超声波遇到障碍物后反射回来，模块收到返回来的超声波。

③ 回响：模块接收到反射回来的超声波后，Echo 引脚输出一个与检测距离成比例的高电平。

我们只要在该引脚为高时，开启定时器计数，在该引脚变为低时，结束定时器计数。根据定时器的计数和定时器频率就可以算出经历时间，根据时间即可推导出距离。

明白原理后，便可开始写程序。

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/24_SR04 超声波测距模块/gpio_sr04”目录下。

## 21.3.2 编写 GPIO 函数

由分析可知道，这里至少需要两个引脚：

- 一个脉冲触发引脚(Trig 引脚)，这里为 GPIO4_19；
- 一个回响接收引脚(Echo 引脚)，这里为 GPIO4_20。

Trig 引脚设置为输出，Echo 引脚设置为中断引脚。

Trig 引脚设置函数如下，在 sr04.c 文件中，如下：

```
//设置 Trig 引脚输出电平
static void set_gpio_value(int value)
{
    /* IRDA GPIO4_IO19 */
    GPIO4->GDIR |= (1<<19);

   if (1 == value)
      GPIO4->DR |= (1<<19);
```

```
    else
        GPIO4->DR &= ~(1<<19);
}
```

Echo 引脚设置函数、中断使能函数如下，在 main.c 中，代码如下：

```
void init_pins(void)
{
    ……
     /*  pins GPIO4_IO20 */
    IOMUXC_SetPinMux(IOMUXC_CSI_HSYNC_GPIO4_IO20, 0U);
    IOMUXC_SetPinConfig(IOMUXC_CSI_HSYNC_GPIO4_IO20,
                        IOMUXC_SW_PAD_CTL_PAD_DSE(6U) |
                        IOMUXC_SW_PAD_CTL_PAD_SPEED(2U) |
                        IOMUXC_SW_PAD_CTL_PAD_PKE_MASK |
                        IOMUXC_SW_PAD_CTL_PAD_HYS_MASK);
}

void echo_irq_init(void)
{
    /* if set detects any edge on the corresponding input signal*/
    GPIO4->EDGE_SEL |= (1 << 20);
    /* if set 1, unmasked, Interrupt n is enabled */
    GPIO4->IMR |= (1 << 20);
}
```

### 21.3.3 编写定时器函数

在“timer.c”添加计数器启动和停止函数，参考前面定时器章节的程序，这里时钟源选择 Peripheral Clock (ipg_clk)为 66M，预分频值设置为 0（即预分频值为 1），代码如下：

```
void gpt2_start(void)
{
    GPT2->CR |= (1 << 15);
    while((GPT2->CR >> 15) & 0x1);
    GPT2->CR = (1 << 6) | (1 << 5) | (1 << 3) | (1 << 1);
    GPT2->PR = 0;
    GPT2->CR |= 1;
}

void gpt2_stop(void)
{
    GPT2->CR &= ~1;
    GPT2->SR = 0x1;
}
```

### 21.3.4 编写触发函数

准备工作差不多就做完了，可以开始发送触发信号，扔给中断处理。

下列函数设置 Trig 引脚，让它发出一个大于 10us 的高脉冲，并使能 Echo 引脚的 GPIO 中断。

代码在 sr04.c 中，如下：

```
void start_sr04(void)
{
    set_gpio_value(0);
    set_gpio_value(1);
    delay_us(15);
```

```
    gic_enable_irq(GPIO4_Combined_16_31_IRQn);
    set_gpio_value(0);
}
```

### 21.3.5 编写中断处理函数

在“gic.c”的中断处理函数里：

① 先判断发生中断的引脚是高电平还是低电平；

② 如果是高电平则表示回响信号开始：启动定时器并记录当前定时器值。

③ 如果是低电平则表示回响信号结束：

记录下当前定时器值，停止定时器。

代码如下：

```
static unsigned long int sr04_echo_start = 0;
static unsigned long int sr04_echo_stop = 0;

void handle_irq_c(void)
{
    int nr;

    GIC_Type *gic = get_gic_base();
    nr = gic->C_IAR;

    switch(nr)
    {
        case GPIO4_Combined_16_31_IRQn:
        {
            if((GPIO4->DR >> 20 ) & 0x1)
            {
                gpt2_start();
                sr04_echo_start = GPT2->CNT;
            }
            else
            {
                sr04_echo_stop = GPT2->CNT;
                gpt2_stop();

                gic_disable_irq(GPIO4_Combined_16_31_IRQn);

                int distance = (sr04_echo_stop-sr04_echo_start)*17/66/100;
                printf("distance=%d mm\n\r", distance);

            }
            GPIO4->ISR |= (1 << 20);
            break;
        }
        default:
            break;
    }
    gic->C_EOIR = nr;
}
```

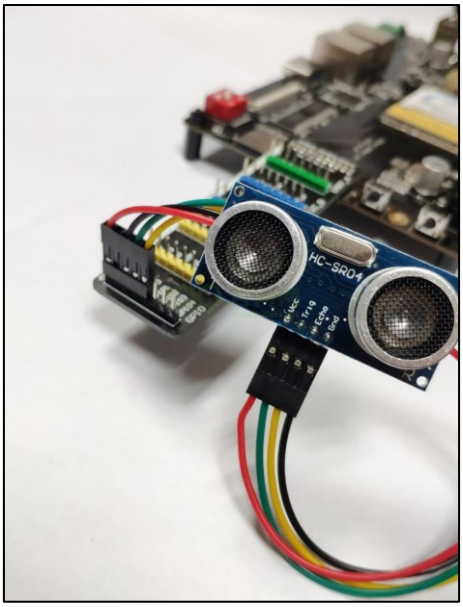
把 2 个节点得到定时器的差值，用来计算距离。

假设差值为 108930，除以频率 66MHz，即可得到 1650us，换算成 0.00165s，乘以声速 340m/s，得到距离为 0.56m。

这 0.56m 为声音发出再返回的距离，还得除以 2，得到真实距离为 280mm。

## 21.4 SR04 超声波测距模块测试

IMX6ULL 先断电，按下图所示，将模块插在扩展板的 GPIO0，将扩展板插在底板上。




**注意：为了防止用户接错方向，模块和扩展板都有一条长白线，连接时需要模块上的白线和扩展板的白线在同一侧。**

**注意：注意 SR04 模块中的红线、黑线、绿线、黄线的位置，如下图接线。**

编译、运行程序，打开串口观察。将手由远到近靠近超声波模块正面，可以看到如下效果，注意模块距离最小识别距离为 2cm：

```
hello world
SR04
distance=230 mm
distance=210 mm
distance=204 mm
distance=123 mm
distance=94 mm
distance=69 mm
distance=76 mm
```

# 第22章 步机电机模块

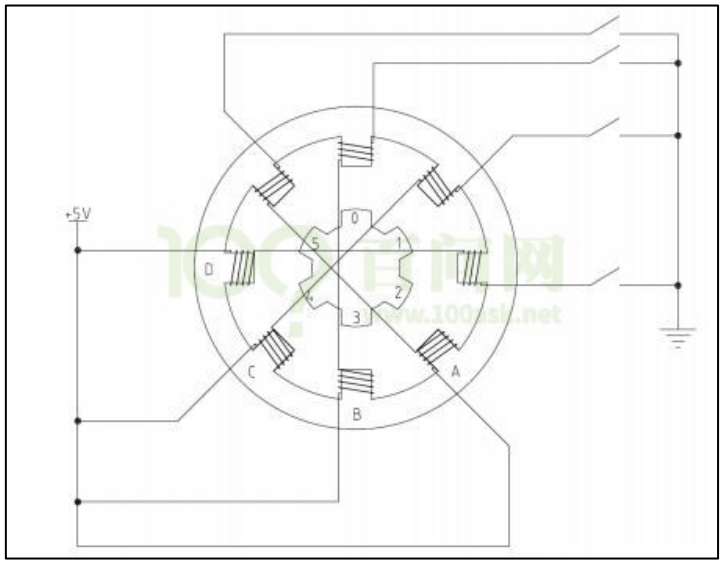
模块原理图及资料：网盘开发板配套资料“05_Hardware (原理图)/Extend_modules/步进电机驱动模块.zip”。

## 22.1 28BYJ-48 电机原理

28BYJ-48 是一款常见的步机电机，其名称的含义为外径 28 毫米四相八拍式永磁减速型步进电机。型号的含义如下：

① 28：步进电机的有效最大外径是 28 毫米
② B：表示是步进电机
③ Y：表示是永磁式
④ J：表示是减速型（减速比 1:64）
⑤ 48：表示四相八拍

先说什么是“4 相永磁式”的概念，28BYJ-48 的内部结构示意图如下所示：




先看里圈，它上面有 6 个齿，分别标注为 0～5，这个叫做转子，顾名思义，它是要转动的，转子的每个齿上都带有永久的磁性，是一块永磁体，这就是“永磁式”的概念。

再看外圈，这个就是定子，它是保持不动的，实际上它是跟电机的外壳固定在一起的，它上面有 8 个齿，而每个齿上都缠上了一个线圈绕组，正对着的 2 个齿上的绕组又是串联在一起的，也就是说正对着的 2 个绕组总是会同时导通或关断的，如此就形成了 4 相，在图中分别标注为 A-B-C-D，这就是“4 相”的概念。每组绕组各有一端连接到公共端，另外一端由独立的引线引出，如下图：

步进电机一共有 5 根线引出，红色是公共端，接 5v 电源，其他四根分别对应 A,B,C ,D 四个绕组的另外一端。

怎样让电机转动起来呢？

首先，假设我们让 B 线导通，此时转子 0 和 3 都对应 B 有一个吸引力；可以看到的是，A 和 2 之间有一个很小的夹角，当我们截断 B，导通 A 时，转子就会顺时针转动对齐 A。

此时，我们 D 和 4 之间的夹角也减到了最小（再小就是正对），为了让转子再旋转一点，我们可以先关闭 A，导通 D，此时 4 和 D 之间产生的吸引力，使电

机又顺时针转动了一点。

当我们依次单独导通 BADC 时，电动机就顺时针转到起来了。

那么很明显，当完成一个 B-A-D-C 的四节拍操作后，转子的 3 号齿原来对准 B 定子，现在对准 C 定子，即转子转过了八分之一圈。

依此类推，8 个四节拍以后转子将转过完整的一圈，而其中单个节拍使转子转过的角度就很容易计算出来了，即 360 度/(8*4)=11.25 度，这个值就叫做步进角度。而上述这种工作模式就是步进电机的单四拍模式-单相绕组通电四节拍。同样的道理，如果想让转子逆时针转动，可以依次单独导通 BCDA。

我们再来介绍一种具有更优性能的工作模式，那就是在单四拍的每两个节拍之间再插入一个双绕组导通的中间节拍，组成八拍模式。

比如，在逆时针转动过程中，从 B 相导通到 C 相导通的过程中，加入一个 B 相和 C 相同时导通的节拍，这个时候，由于 B、C 两个绕组的定子齿对它们附近的转子齿同时产生相同的吸引力，这将导致这两个转子齿的中心线对比到 B、C 两个绕组的中心线上，也就是新插入的这个节拍使转子转过了上述单四拍模式中步进角度的一半，即 5.625 度。这样一来，就使转动精度增加了一倍，而转子转动一圈则需要 8*8=64 拍了。另外，新增加的这个中间节拍，还会在原来单四拍的两个节拍引力之间又加了一把引力，从而可以大大增加电机的整体扭力输出，使电机更“有劲”了，而且更平顺。

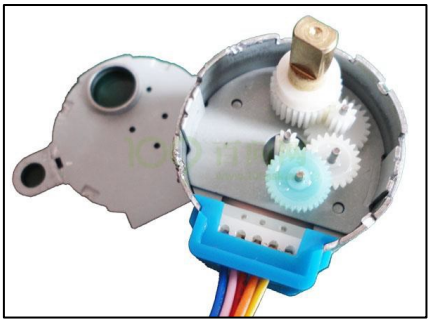
下表给出八拍模式下电机绕组激励时序（电机引线颜色可能因厂家不同而不同）：

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| P1-红 | VCC | VCC | VCC | VCC | VCC | VCC | VCC | VCC |
| P2-橙 | GND | GND | | | | | | GND |
| P3-黄 | | GND | GND | GND | | | | |
| P4-粉 | | | | GND | GND | GND | | |
| P5-蓝 | | | | | | GND | GND | GND |

本文将以八拍模式展开编程演示，当按照下表的数值，连续给电机提供激励时，电机就转了起来。

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| D | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| C | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| B | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| A | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

另外，28BYJ-48 为减速电机，电机输出的转速并不等于转子的转速。下图是这个 28BYJ-48 步进电机的拆解图：



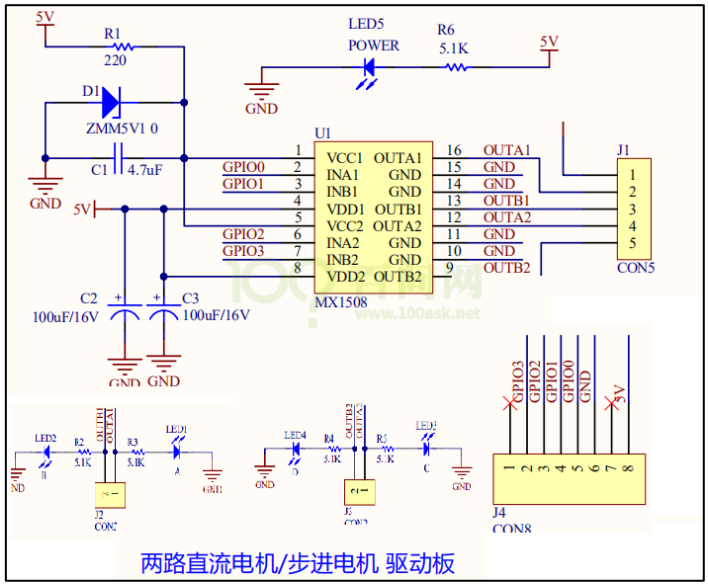
从图中可以看到，位于最中心的那个白色小齿轮才是步进电机的转子输出，64 个节拍只是让这个小齿轮转了一圈，然后它带动那个浅蓝色的大齿轮，这就是一级减速。

大家看一下右上方的白色齿轮的结构，除电机转子和最终输出轴外的 3 个传动齿轮都是这样的结构，由一层多齿和一层少齿构成，而每一个齿轮都用自己的少齿层去驱动下一个齿轮的多齿层，这样每 2 个齿轮都构成一级减速，一共就有了 4 级减速。

## 22.2 步进电机模块硬件设计

根据 28BYJ-48 电机原理，我们只需要将 100ASK_IMX6ULL 的四个引脚（通常为 GPIO）分别连接到电机，再按照电机的驱动逻辑给出一定的激励信号。

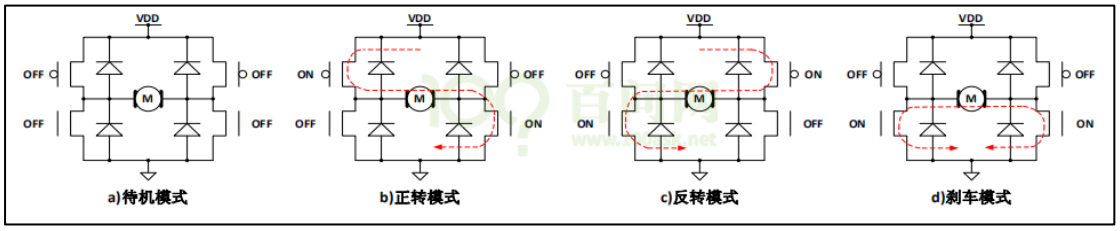
因 100ASK_IMX6ULL 的 GPIO 驱动能力有限，需要在 100ASK_IMX6ULL 和电机之间加入驱动电路，本案选择了双路有刷直流马达驱动芯片 MX1508，驱动电路原理图见下图：




J1 连接到 100ASK_IMX6ULL 基板的扩展接口，J4 连接到 28BYJ-48 电机的接线插座，这样 100ASK_IMX6ULL 就通过驱动芯片 MX1508 与电机建立了控制关系。

MX1508 芯片采用 H 桥电路结构设计，采用高可靠性功率管工艺，特别适合驱动线圈、马达等感性负载，内置两路驱动电路，可同时驱动两个线圈马达，工作电压范围覆盖 2V 到 9.6V。电路设计有芯片级温度检测电路，实时监控芯片内部发热，当芯片内部温度超过设定值时(典型值 150℃)，启动保护功能，芯片内置的温度迟滞电路，确保电路恢复到安全温度后，才允许重新对功率管进行控制。

MX1508 芯片内部电路和基本工作模式：



① 待机模式

在待机模式下，INAx=INBx=L。包括驱动功率管在内的所有内部电路都处于关断状态。电路消耗极低极低的电流。此时马达输出端 OUTAx 和 OUTBx 都为高阻状态（记为 Z）。

② 正转模式

正转模式的定义为：INAx=H，INBx=L，此时马达驱动端 OUTAx 输出高电平，马达驱动端 OUTBx 输出低电平时，马达驱动电流从 OUTAx 流入马达，从 OUTBx 流到地端，此时马达的转动定义为正转模式。

③ 反转模式

反转模式的定义为：INAx=L，INBx=H，此时马达驱动端 OUTBx 输出高电平，马达驱动端 OUTAx 输出低电平时，马达驱动电流从 OUTBx 流入马达，从 OUTAx 流到地端，此时马达的转动定义为反转模式。

④ 刹车模式

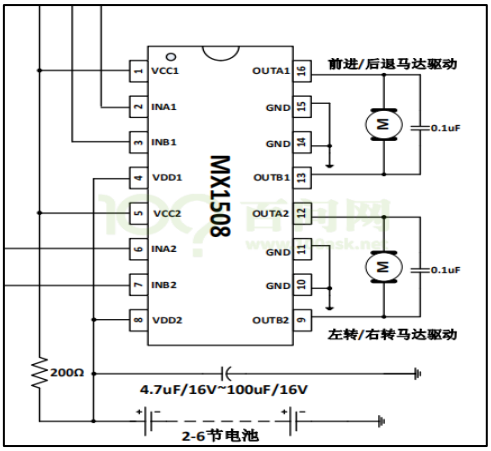
刹车模式的定义为：INAx=H，INBx=H，此时马达驱动端 OUTAx 以及 OUTBx 都输出低电平，马达内存储的能量将通过 OUTAx 端 NMOS 管或者 OUTBx 端 NMOS 快速释放，马达在短时间内就会停止转动。注意在刹车模式下电路将消耗静态功耗。

⑤ PWM 模式 A

当输入信号 INAx 为 PWM 信号，INBx=0 或者 INAx=0，INBx 为 PWM 信号时，马达的转动速度将受到 PWM 信号占空比的控制。在这个模式下，马达驱动电路是在导通和待机模式之间切换，在待机模式下，所有功率管都处于关断状态，马达内部储存的能量只能通过功率 MOSFET 的体二极管缓慢释放。

当输入信号 INAx 为 PWM 信号，INBx=1 或者 INAx=1，INBx 为 PWM 信号时，马达的转动速度将受到 PWM 信号占空比的控制。在这个模式下，马达驱动电路输出在导通和刹车模式之间，在刹车模式下马达存储的能量通过低边的 NMOS 管快速释放。

MX1508 芯片一个应用场景是电池玩具车，一个通道驱动车子的驱动轮（下图中的 M1），一个通道驱动转向（M2），这样就可以实现玩具车的前进、后退、左转、右转。




由上图可知，MX1508 芯片用来单独驱动两个单线圈电机时，可以通过给输入引脚 INA1、INB1、INA2、INB2 提供不同的信号轻松控制电机转向和转速。如果将驱动电路的输出端 OUTA1、OUTB1、OUTA2、OUTB2 分别接到步进电机 28BYJ-48 的 A、B、C、D 四个相的线圈，OUTA1、OUTB1、OUTA2、OUTB2 能够按照电机要求的时序，按照节拍输出相应的激励，就可以将步进电机驱动起来。

# 22.3 步进电机模块软件设计

## 22.3.1 确定转动节拍对应的引脚值

在了解了 28BYJ-48 电机和驱动芯片 MX1508 的原理之后，想要通过编写程序控制电机的转动，还要搞清楚控制信号值的映射关系。

从扩展板原理图和电机驱动板原理图，可以得到 GPIO 和相位的关系如下：

| 电机相位 | 驱动芯片输出 | 驱动芯片输入 | 100ASK_IMX6ULL |
|---|---|---|---|
| D | OUTB2 | INB2 | GPIO4_22 |
| C | OUTA2 | INA2 | GPIO4_21 |
| B | OUTB1 | INB1 | GPIO4_20 |
| A | OUTA1 | INA1 | GPIO4_19 |

我们知道，按照下图的时序就可以驱动电机转动，那么可以沿着信号传输线路推导出 100ASK_IMX6ULL 的 GPIO 的信号序列，这样我们就可以开始编程。

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| D | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| C | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| B | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| A | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

下图为驱动芯片 MX1508 输入输出真值表：

**逻辑真值表**

| INAx | INBx | OUTAx | OUTBx |
|---|---|---|---|
| L | L | Z | Z |
| H | L | H | L |
| L | H | L | H |
| H | H | L | L |

注：x 代表 1 或者 2。

可以看出 OUTB1、OUTA1 不能同时输出 1（只能同时为高阻态），因步进电机的公共端是接到 5V 的，可以用高阻态代替 1,OUTB2、OUTA2 也是如此。

根据电机激励序列推出驱动芯片的输出序列；根据 MX1508 输入输出真值表，可以从输出信号序列推出输入序列；由电机驱动芯片的输入推导出 100ASK_IMX6ULL 的 GPIO 的信号序列：

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| D | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| C | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| B | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| A | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| OUTB2 | 0 | 0 | 1 | 1 | Z | Z | Z | 0 |
| OUTA2 | 1 | 0 | 0 | 0 | Z | Z | Z | 1 |
| OUTB1 | Z | Z | Z | 0 | 0 | 0 | 1 | 1 |
| OUTA1 | Z | Z | Z | 1 | 1 | 0 | 0 | 0 |
| INB2 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| INA2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| INB1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| INA1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| GPIO4_22 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| GPIO4_21 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| GPIO4_20 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

| GPIO4_19 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|
| NUM[3:0] | 4 | c | 8 | 9 | 1 | 3 | 2 | 6 |

这个表可能让人难以理解，举例说明一下。

以第 1 个节拍为例，想让 D 输出 0，ABC 输出高电平或是高阻态，怎么办？换句话说，想让 OUTB2=L，OUTA2、OUTA1、OUTB1 等于 H 或 Z，怎么办？

根据真值表，要设置 INA2=H、INB2=L，真值表如下：

**逻辑真值表**

| INAx | INBx | OUTAx | OUTBx |
|---|---|---|---|
| L | L | Z | Z |
| H | L | H | L |
| L | H | L | H |
| H | H | L | L |

注：x 代表 1 或者 2。

根据真值表，设置 INA1=L、INB1=L 时，可以让 OUTA1=Z、OUTB1=Z，真值表如下：

**逻辑真值表**

| INAx | INBx | OUTAx | OUTBx |
|---|---|---|---|
| L | L | Z | Z |
| H | L | H | L |
| L | H | L | H |
| H | H | L | L |

注：x 代表 1 或者 2。

所以，要让 D 输出 0，ABC 输出高电平或是高阻态时，需要：INA1=L、INB1=L、INA2=H、INB2=L。

即：GPIO4_19=0、GPIO4_20＝0、GPIO4_21=1、GPIO4_22=0，用二进制表示即为：0b0100，即 0x04。

将信号序列即 8 个节拍对应的 GPIO 值存到数组中：

```
S_CW[8]=  {0x04,0x0c,0x08,0x09,0x01,0x03,0x02,0x06};
```

配置好 GPIO 后，将数组中的数据循环写到 GPIO 输出寄存器，就可以驱动电机转动起来。

按照反向的顺序循环写，就可以实现电机反转。调节两个节拍间的周期，可以改变电机转速。

根据电机的参数：空载牵入频率≥600Hz 可知，两个节拍之间的时间间隔不宜小于 1.6ms。下面开始写程序，

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/25_步进电机模块/gpio_motor”目录下。

## 22.3.2 先写 main 函数

后面会实现 2 个函数：

- motor_sw 实现控制电机的顺时针转动；
- motor_aw 实现控制电机的逆时针转动。

在 main 函数中调用初始化函数后，调用 motor_sw、motor_aw 转动电机，代码在 main.c 中，如下：

```
int  main()
{
    gpio_init();
    while(1)
    {
```

```
        motor_sw(10,2000);  //顺时针 转 2000 步 速度 10          速度范围：1-10
        motor_aw(7,2000);   //逆时针 转 2000 步 速度 7
    }
   return 0;
}
```

在初始化好 GPIO 的配置寄存器后，调用电机转动函数，函数第一个参数为转速（1-10 级可调），第二个参数为转动步数。

这样电机有顺时针和逆时针交替的转动起来。

### 22.3.3 编写 GPIO 函数

代码在 motor.c 中，如下：

```
/************************************************************************
 * 函数名称： gpio_init
 * 功能描述： 初始化 GPIO4   _19 _20 _21 _22
 * 输入参数： 无
 * 输出参数： 无
 * 返 回 值： 无
 * 修改日期        版本号     修改人       修改内容
 * -----------------------------------------------
 * 2020/02/07         V1.0      韦东山       创建
 ***********************************************************************/
void gpio_init(void)
{

/* a. 使能 GPIO4
 * set CCM to enable GPIO4
 * CCM_CCGR1[CG15] 0x20C4074
 * bit[13:12] = 0b11
 */
    *CCM_CCGR3 |= (3<<12);

//设置 IOMUX 来选择引脚用于 GPIO4 4_19,4_20,4_21,4_22。
    *IOMUXC_SW_MUX_CTL_PAD_CSI_VSYNC      = 5;
    *IOMUXC_SW_MUX_CTL_PAD_CSI_HSYNC      = 5;
    *IOMUXC_SW_MUX_CTL_PAD_CSI_DATA00     = 5;
    *IOMUXC_SW_MUX_CTL_PAD_CSI_DATA01     = 5;

//设置 GPIO4_GDIR 中  4_19,4_20,4_21,4_22 引脚设置为输出功能。
    GPIO4->GDIR |= (15 << 19);
}
```

- 第一步使能 GPIO4 模块，
- 第二步设置 IOMUX 寄存器，使其对应的引脚连接到 GPIO 模块，
- 第三步将 GPIO4_19、GPIO4_20、GPIO4_21、GPIO4_22 设置为输出功能。

### 22.3.4 编写电机顺时针转动函数 motor_sw

代码在 motor.c 中，如下：

```
static int  S_CW[8]=  {0x2,0x3,0x1,0x9,0x8,0xc,0x4,0x6};
static int  Speed[10]={200,100,75,50,25,12,6,3,2,1};
static char phase = 0;
```

```
/*************************************************************************
 * 函数名称： motor_sw
 * 功能描述： 电机顺时针转
 * 输入参数：
 *     speed :转动速度     1-10
       degree:转动步数
 * 输出参数： 无
 * 返 回 值： 无
 *************************************************************************/
void motor_sw(int speed ,int degree)
{
   if(speed  > 10 || speed  < 1)
       speed  = 5;
   int  i,num;
   for(i=0;i<degree;i++)
    {
        phase ++;
        num  = S_CW[phase&7];
        num  = num << 19;
        GPIO4->DR = num;
        delaynms(Speed[speed-1]);
    }
}
```

motor_sw 函数根据传入的步数，循环将 S_CW 数组里的数据依次取出，写到 GPIO4_19、GPIO4_20、GPIO4_21、GPIO4_22 这 4 个引脚上，步进电机就顺时针转动了起来。调整两次写 GPIO 之间的时间间隔，就实现了电机转速的调整。

## 22.3.5 编写电机逆时针转动函数 motor_aw

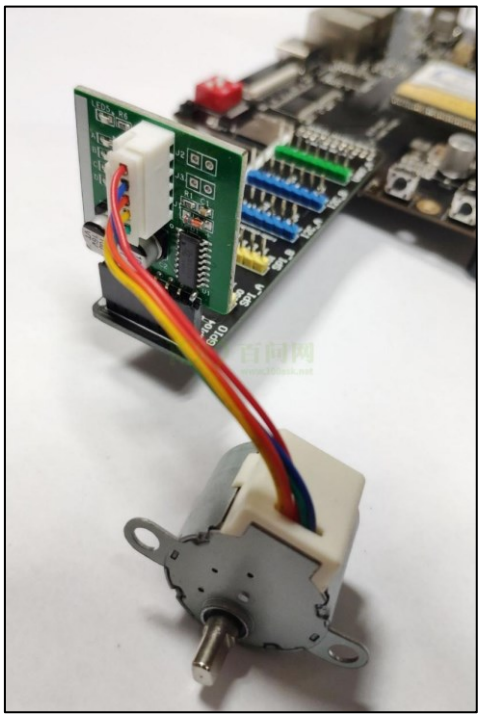
代码在 motor.c 中，如下：

```
/*************************************************************************
 * 函数名称： motor_aw
 * 功能描述： 电机逆时针转
 * 输入参数：
 *     speed :转动速度     1-10
       degree:转动步数
 * 输出参数： 无
 * 返 回 值： 无
 * 修改日期        版本号     修改人       修改内容
 * -----------------------------------------
  *************************************************************************/
void motor_aw(int speed ,int degree)
{
   if ((speed  > 10)||(speed  < 1))
       speed  = 5;

   int  i,num;
   for(i=0;i<degree;i++)
    {
        phase --;
        num  = S_CW[phase&7];
        num  = num << 19;
        GPIO4->DR = num;
        delaynms(Speed[speed-1]);
    }
}
```

motor_aw 函数根据传入的步数，循环将 S_CW 数组里的数据逆序取出，写到 GPIO4_19、GPIO4_20、GPIO4_21、GPIO4_22 这 4 个引脚上，步进电机就逆时针转动了起来。调整两次写 GPIO 之间的时间间隔，就实现了电机转速的调整。

## 22.4 步进电机模块测试

IMX6ULL 先断电，按下图所示，将模块插在扩展板的 GPIO0，将扩展板插在底板上：



注意：为了防止用户接错方向，模块和扩展板都有一条长白线，连接时需要模块上的白线和扩展板的白线在同一侧。

编译、运行程序，步机电机先快速顺时针转一圈，再慢速逆时针转一圈，如此反复。

# 第23章 OLED 显示模块

模块原理图及资料：网盘开发板配套资料“05_Hardware （原理图)/Extend_modules/oled.zip”。

## 23.1 OLED 简介

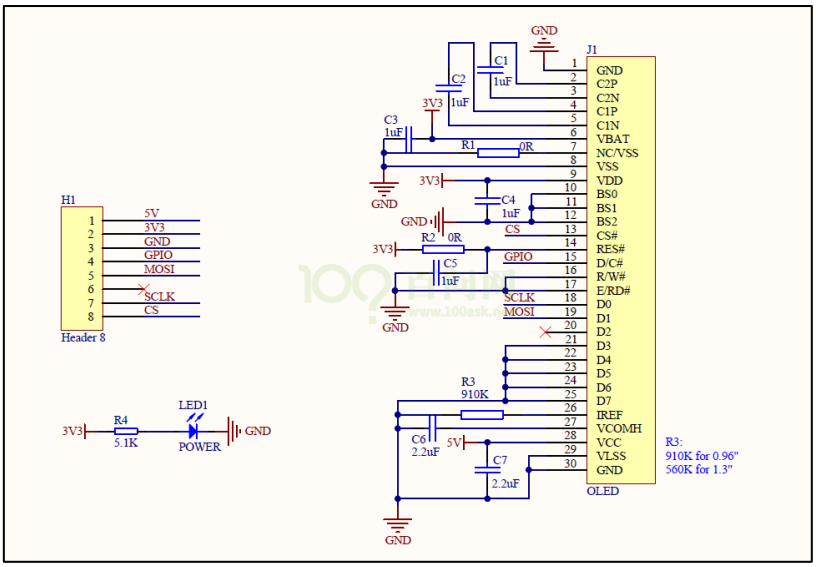
OLED，即有机发光二极管（Organic Light Emitting Diode），属于一种电流型的有机发光器件，是通过载流子的注入和复合而致发光的现象，发光强度与注入的电流成正比。OLED 由于同时具备自发光、不需要背光、对比度高、厚度薄、视角广和反应速度块等，因此被广泛应用到挠性面板。OLED 的使用温度范围广，构造及制程较简单。OLED 由于自发光，因此显示效果较好。

## 23.2 OLED 模块硬件设计

在这个实验中，我们使用百问网制作的 OLED 模块，该模块为 0.96 寸的屏幕，支持 128*64 像素显示。

该屏幕使用驱动芯片 SSD1306。SSD1306 是一块内置 CMOS OLED/PLED 驱动控制器的 IC 芯片，芯片可以驱动共阴型 OLED 面板。芯片内部包含晶振、显示 RAM、对比度控制模块以及 256 级亮度控制模块，大大降低了外围元器件数量和功耗。

主机可以通过 6800/8000 并行接口、I2C 接口或者 SPI 接口实现对 SSD1306 的控制，百问网的 OLED 模块是使用的 SPI 接口进行控制。原理图如下：



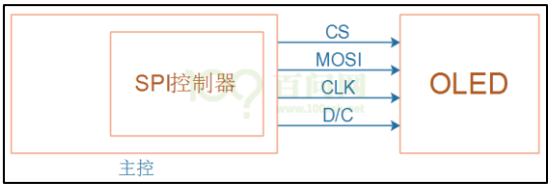

原理图中 BS[2:0]用于选择接口的类型，列表如下图所示。原理图上 BS[2:0]都设置为低电平，使用的是 4 线 SPI 接口：CS、MOSI、MISO、CLK。

| SSD1306 Pin Name | I²C Interface | 6800-parallel interface (8 bit) | 8080-parallel interface (8 bit) | 4-wire Serial interface | 3-wire Serial interface |
|---|---|---|---|---|---|
| BS0 | 0 | 0 | 0 | 0 | 1 |
| BS1 | 1 | 0 | 1 | 0 | 0 |
| BS2 | 0 | 1 | 1 | 0 | 0 |

实际上，我们只需要把数据发给 OLED，不需要从 OLED 读取数据，所以 MISO

引脚用不到。另外，还需要一个 GPIO 来表示发送给 OLED 的是命令，还是数据。

主机跟 OLED 之间的简化连接图，如下：

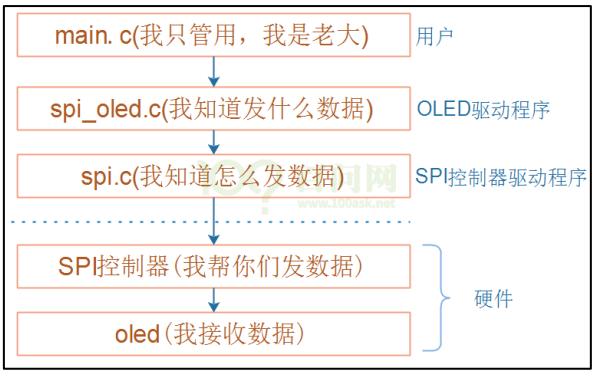



## 23.3 模块软件设计

### 23.3.1 程序框架

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/26_OLED 显示模块/spi_oled”目录下。

所涉及文件如下图：




spi.c 中实现了 SPI 的操作，在前面的章节有 SPI 的详细介绍；这不是本章的重点。

重点在于 spi_oled.c，它实现了 OLED 的操作：怎么初始化、怎么发命令和数据、怎么显示。

### 23.3.2 OLED 的基本操作:发命令、发数据

通过 SPI 接口发送给 OLED 的信息，可以是命令，也可以是数据：

- 当 DC 引脚是低电平时，是命令；
- 当 DC 引脚是高电平时，是数据。

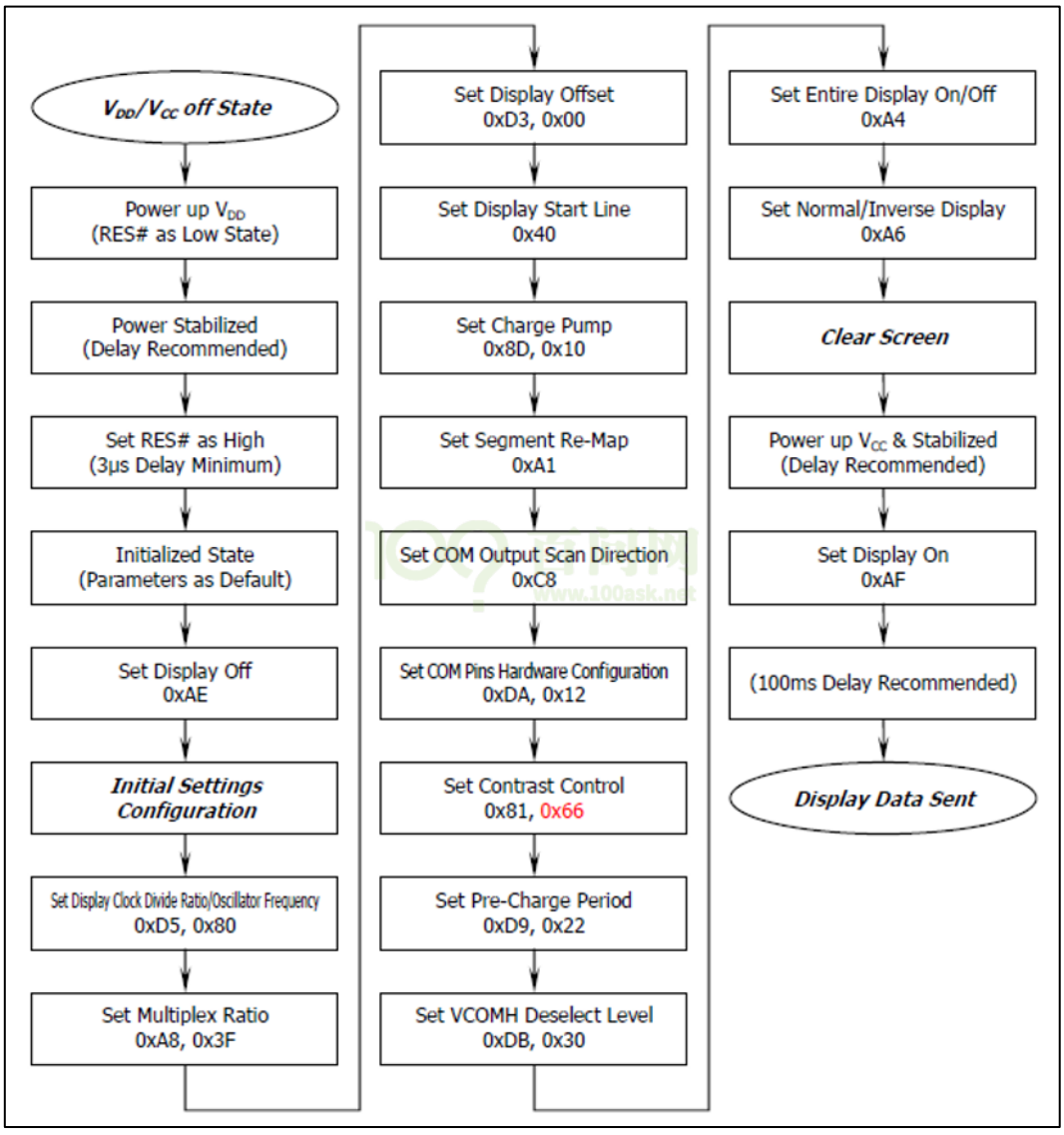
发命令、发数据的代码在“spi.c”的 oled_write_cmd_data 函数中，如下：

```
22 void oled_write_cmd_data(unsigned char uc_data,unsigned char uc_cmd)
23 {
24     unsigned char uc_read=0;
25     if(uc_cmd==0)
26     {
27             *GPIO4_DR_s &= ~(1<<20);//拉低，表示写入指令
28     }
29     else
30     {
31             *GPIO4_DR_s |= (1<<20);//拉高，表示写入数据
32     }
33     spi_writeread(ESCPI1_BASE,uc_data);//写入
34 }
```

OLED 的主控芯片是 SSD1306，SSD1306 芯片手册中有很多命令。实际上我们不用关心这些命令，在 OLED 手册里它会告诉我们应该发什么命令来初始化 OLED。要在 OLED 上显示字符时，也会用到几个命令，用到时再讲解。

### 23.3.3 编写 OLED 的初始化函数

初始化 OLED 并不复杂，芯片手册中列出了需要发出的一系列命令，如下：



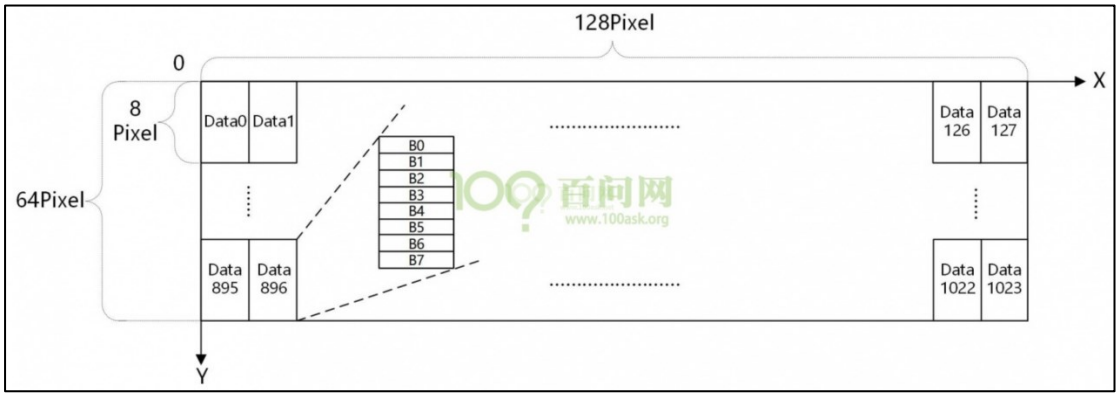

上图为 OLED 屏幕要求的初始化流程，我们按照流程对 OLED 屏幕进行初始化。相关的初始化代码在程序文件“spi_oled.c”的 oled_init 函数，如下：

```
45 int oled_init(void)
46 {
47     unsigned char uc_dev_id = 0;
48
49     GPIO4_GDIR_s = (volatile unsigned int *)(0x20a8000 + 0x4);
50     GPIO4_DR_s   = (volatile unsigned int *)(0x20a8000);
51     spi_init(ESCPI1_BASE);
52
53     oled_write_cmd_data(0xae,OLED_CMD);//关闭显示
54
55     oled_write_cmd_data(0x00,OLED_CMD);//设置 lower column address
56     oled_write_cmd_data(0x10,OLED_CMD);//设置 higher column address
```

```
57
58      oled_write_cmd_data(0x40,OLED_CMD);//设置 display start line
59
60      oled_write_cmd_data(0xB0,OLED_CMD);//设置 page address
61
62      oled_write_cmd_data(0x81,OLED_CMD);// contract control
63      oled_write_cmd_data(0x66,OLED_CMD);//128
64
65      oled_write_cmd_data(0xa1,OLED_CMD);//设置 segment remap
66
67      oled_write_cmd_data(0xa6,OLED_CMD);//normal /reverse
68
69      oled_write_cmd_data(0xa8,OLED_CMD);//multiple ratio
70      oled_write_cmd_data(0x3f,OLED_CMD);//duty = 1/64
71
72      oled_write_cmd_data(0xc8,OLED_CMD);//com scan direction
73
74      oled_write_cmd_data(0xd3,OLED_CMD);//set displat offset
75      oled_write_cmd_data(0x00,OLED_CMD);//
76
77      oled_write_cmd_data(0xd5,OLED_CMD);//set osc division
78      oled_write_cmd_data(0x80,OLED_CMD);//
79
80      oled_write_cmd_data(0xd9,OLED_CMD);//ser pre-charge period
81      oled_write_cmd_data(0x1f,OLED_CMD);//
82
83      oled_write_cmd_data(0xda,OLED_CMD);//set com pins
84      oled_write_cmd_data(0x12,OLED_CMD);//
85
86      oled_write_cmd_data(0xdb,OLED_CMD);//set vcomh
87      oled_write_cmd_data(0x30,OLED_CMD);//
88
89      oled_write_cmd_data(0x8d,OLED_CMD);//set charge pump disable
90      oled_write_cmd_data(0x14,OLED_CMD);//
91
92      oled_write_cmd_data(0xaf,OLED_CMD);//set dispkay on
93
94      return 0;
95 }
```

## 23.3.4 OLED 的像素显示

OLED 上有 128*64 个像素，每个像素只有 2 种状态：亮、灭。如下图所示：




怎么控制某个像素的状态呢？OLED 内部有显存。显存中每个字节用来表示一列的 8 个像素，注意：是一列，不是一行。

显存中数据和像素的对应关系见上图，初始化好 OLED 后，只要往 OLED 显存中写入数据即可显示。怎么写入数据？有 2 个问题要解决：

① 怎么发地址？

对于 64 行像素，可以分为 8 页，每页对应 8 行。

要寻址显存中某个字节时，先确定它是处于哪页(Page)，再确定它处于哪列(Col)。

通过命令发出 Page 和 Col 即可，后面再详细介绍。

② 怎么发数据？

让 DC 为高电平，通过 SPI 发送数据即可。

## 23.3.5 OLED 设置地址函数

OLED 主控的手册里介绍了三种地址模式，我们常用的是页地址模式(Page addressing mode)，它把显存的 64 行分为 8 页，每页对应 8 行；选中某页后，再选择某列，然后就可以往里面写数据了，每写一个数据，地址就会加 1，一直写到最右端的位置，它会自动跳到最左端。

通过命令来实现发送页地址和列地址，其中列地址分为两次发送，先发送低字节，再发送高字节。

代码为“spi_oled.c”的 OLED_DIsp_Set_Pos 函数，这个函数里 y 就对应 Page，它的值从 0 到 7；x 就对应 Col(列)，它的值从 0 到 127。

代码如下：

```
176 void OLED_DIsp_Set_Pos(int x, int y)
177 {   oled_write_cmd_data(0xb0+y,OLED_CMD);
178     oled_write_cmd_data((x&0x0f),OLED_CMD);
179     oled_write_cmd_data(((x&0xf0)>>4)|0x10,OLED_CMD);
180 }
```

## 23.3.6 OLED 清屏函数

把显存全部写为 0，即可清屏。

代码为“spi_oled.c”的 OLED_DIsp_Clear 函数，如下：它一页一页地发送数据，第 137 行设置地址，第 138～139 行发送 128 个数据，代码如下：

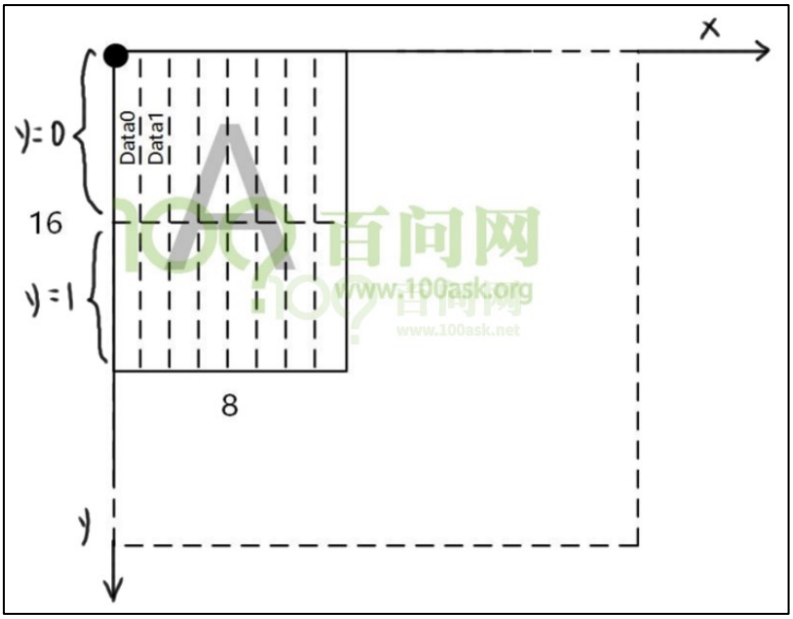
```
132 void OLED_DIsp_Clear(void)
133 {
134     unsigned char x, y;
135     for (y = 0; y < 8; y++)
136     {
137         OLED_DIsp_Set_Pos(0, y);
138         for (x = 0; x < 128; x++)
139             oled_write_cmd_data(0, OLED_DATA); /* 清零 */
140     }
141 }
```

## 23.3.7 OLED 显示字符的函数

要显示一个字符，要先得到它的点阵，即字模。

然后在显存中根据字模修改数据，让对应的点亮、灭。

我们使用 8*16 的字模，以字符“A”为例，看看如何在 OLED 上显示出来，如下图所示：




假设要在(x，y)位置显示字符 A，要做的事为：

① 使用命令，发出位置(x，y)；
② 发出 8 个数据，对应字符的上半部；
③ 使用命令，发出位置(x，y+1)；
④ 发出 8 个数据，对应字符的下半部。

代码如下，

```
193 void OLED_DIsp_Char(int x, int y, unsigned char c)
194 {
195     int i = 0;
196     /* 得到字模 */
197     const unsigned char *dots = oled_asc2_8x16[c - ' '];
198
199     /* 发给 OLED */
200     OLED_DIsp_Set_Pos(x, y);
201     /* 发出 8 字节数据 */
202     for (i = 0; i < 8; i++)
203             oled_write_cmd_data(dots[i], OLED_DATA);
204
205     OLED_DIsp_Set_Pos(x, y+1);
206     /* 发出 8 字节数据 */
207     for (i = 0; i < 8; i++)
208             oled_write_cmd_data(dots[i+8], OLED_DATA);
209 }
```

## 23.3.8 OLED 显示字符串的函数

需要注意的是换行：OLED 一行只能显示 16 个字符(8*16 点阵)，如果字符太长，要换行。

代码如下 ：

```
224 void OLED_DIsp_String(int x, int y, char *str)
225 {
226     unsigned char j=0;
227     while (str[j])
228     {
229             OLED_DIsp_Char(x, y, str[j]);//显示单个字符
230             x += 8;
```

```
231             if(x > 127) /* 换行 */
232             {
233                     x = 0;
234                     y += 2;
235             }//移动显示位置
236             j++;
237     }
238 }
```

## 23.3.9 OLED 显示汉字的函数

原理也是一样的，也需要先得到字模。为了简单化处理，我们只制作了“百问网”三个字的字模，存放 font.h 头文件里的 hz_1616 数组中，这是 16x16 的点阵。

显示函数在 spi_oled.c 中，代码如下：

```
252 void OLED_DIsp_CHinese(unsigned char x,unsigned char y,unsigned char no)
253 {
254     unsigned char t,adder=0;
255     OLED_DIsp_Set_Pos(x,y);
256     for(t=0;t<16;t++)
257     {//显示上半截字符
258         oled_write_cmd_data(hz_1616[no][t*2],OLED_DATA);
259         adder+=1;
260     }
261     OLED_DIsp_Set_Pos(x,y+1);
262     for(t=0;t<16;t++)
263     {//显示下半截字符
264         oled_write_cmd_data(hz_1616[no][t*2+1],OLED_DATA);
265         adder+=1;
266     }
267 }
```

## 23.3.10 编写测试函数

在 main.c 中，代码如下：

```
04 int  main()
05 {
06      //使用 SPI_A 接口调试
07      unsigned char uc_cnt;
08      oled_init(); //初始化 TLC5615
09      OLED_DIsp_Clear(); //清屏
10      //OLED_DIsp_All();
11
12      OLED_DIsp_Test();
13      return 0;
14 }
```

## 23.3.11 初始化 SPI1 控制器

我们把 OLED 模块插在扩展板的 SPIA 接口上，这对应 SPI1 控制器。所以需要初始化 SPI，代码在程序文件“spi.c”中，是 spi_init 函数，如下：

```
#include "spi.h"

/*spi1 对应的 iomux 基址*/
#define CSI_DATA7_BASE    0x20e0200
#define CSI_DATA6_BASE    0x20e01fc
```

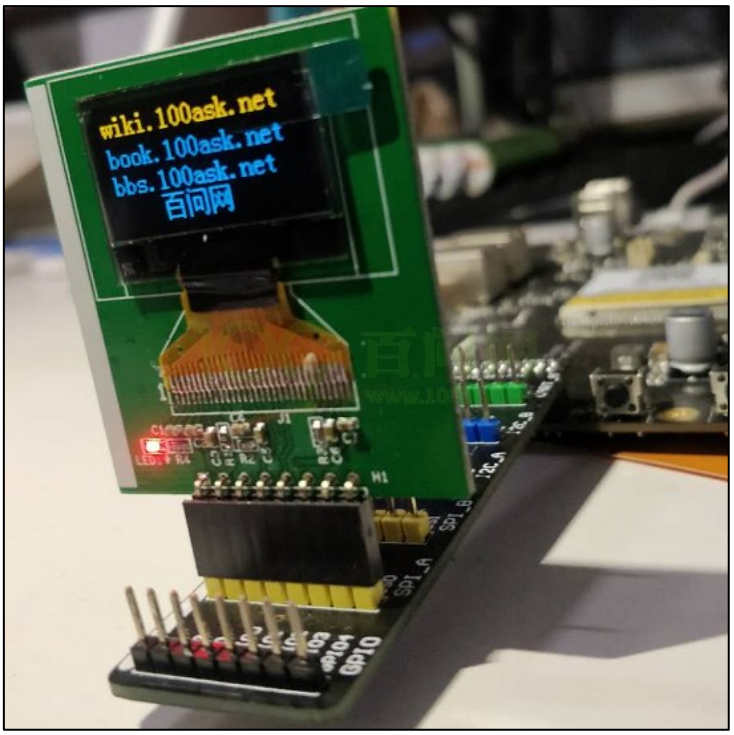
```
#define CSI_DATA5_BASE    0x20e01f8
#define CSI_DATA4_BASE    0x20e01f4
#define CSI_HSYNC         0x20e01e0

unsigned char  spi_init(SPI_CTRL *uc_num)
{
    /*
        1、清除 CONREG 寄存器的 EN 位 来复位模块
        2、在 ccm 中使能 spi 时钟
        3、配置 control register，然后设置 CONREG 的 EN 位来使 spi 模块退出复位
        4、配置 spi 对应的 IOMUX 引脚
        5、根据外部 spi 设备规格来合适的配置 spi 寄存器
    */

    uc_num->CONREG =  0;// clear all bits
    /*
        bit0:使能 SPI
        bit3:写入 TXDATA 之后，立即发送
        bit4:设置通道 0 为 master mode
        bit31:20 设置 burst length ，7 表示为 8bits，一个字节
    */
    uc_num->CONREG |= (7<<20)|(1<<4)|(1<<3)|(1<<0);
    /*  CONFIGREG 采用默认设置
        *bit0        PHA=0
        *bit7:4  sclk 高电平有效
        *bit11:8 通道片选信号，当 SMC =1 的时候，无效（当前处于 SMC=1 模式）
        *bit15:12    POL=0
        *bit19:16    数据线空闲为高电平
        *bit23:20    sclk 空闲为低电平
        *bit28:24    设置消息长度 ，该产品不进行使用
        *
    */

    uc_num->CONFIGREG = 0;
    /*设置时钟相关的*/
    /*
    从 RM 手册 chapter18 中，我们得知时钟来源为 PLL3
    1、pll3_sw_clk_sel 为 0，则选择 pll3；为 1 则选择 ccm_pll3_bys，时钟默认选择 pll3。
输出 pll3_sw_clk 给 spi 进行使用  输出给 spi 的时钟为 480M/8=60Mhz。
    2、我们需要使能 spi 的时钟进行使用，通过 CCM_CCGR1 的 bit5:2 来进行设置。
这部分在制作.imx 文件的时候初始化，可以不处理。
    3、计算时钟频率 CONREG 寄存器
        bit15:12  div_1
        bit11:8   div_2
    最终提供给 spip 的时钟为：60M/(div+1)*(2^div_2))
    假设我们要使用的时钟是 4M，则我们设置 bit15:12 = 15 即可，60M/4 = 15Mhz
    */

    uc_num->CONREG &= ~(0xf<<12|0xf<<8);//清除原先的时钟频率设置
    uc_num->CONREG |= (14<<12|3<<8); //设置 clk = 60/(14+1)/2……3 = 4M/8 = 500KHz

    //引脚初始化
    iomuxc_sw_set(CSI_HSYNC,5);//设置为 GPIO 作为片选来进行使用。GPIO4_IO20
    GPIO1_GDIR  = (volatile unsigned int *)(0x20a8000 + 0x4);
    GPIO1_DR  = (volatile unsigned int *)(0x20a8000);
    *GPIO1_GDIR |= (1<<20);//设置为输出
    *GPIO1_DR |= (1<<20);
    iomuxc_sw_set(CSI_DATA5_BASE,3);
```

```
    iomuxc_sw_set(CSI_DATA4_BASE,3);
    iomuxc_sw_set(CSI_DATA6_BASE,3);
    iomuxc_sw_set(CSI_DATA7_BASE,3);
    return 0;
}
```

## 23.4 OLED 模块测试

IMX6ULL 先断电，按下图所示，将模块插在扩展板的 SPI_A，将扩展板插在底板上。

注意：为了防止用户接错方向，模块和扩展板都有一条长白线，连接时需要模块上的白线和扩展板的白线在同一侧。




编译、运行程序，效果如上图所示。

# 第24章 DAC 模块

模块原理图及资料：网盘开发板配套资料“05_Hardware (原理图)/Extend_modules/dac.zip”。

## 24.1 DAC 简介

DAC(Digital Analog Convector)即数模转换器，是把数字量转变成模拟量的器件。主要包括以下几个主要性能指标：

① 分辨率：指最小输出电压与最大输出电压直逼。如 N 位 DAC 转换器，其分辨率为：1/（2^N-1）。在实际使用中，分辨率的大小也可以用输入数字量的位数来表示；

② 精度线性度；

③ 转换精度：影响转换精度的主要因素有失调误差、增益误差、非线性误差和微分非线性误差灯；

④ 转换速度。

## 24.2 DAC 硬件模块设计

不同类型的 DAC 有不同的通讯方式，这里我们选择 SPI 接口 DAC 芯片 TLC5615。TLC5615 是一个 10 位的 DAC，具有如下特性：

① 10 位 CMOS 电压输出 DAC

② 5V 供电电压

③ 3 线串行操作

④ 高阻抗参考电压输入

⑤ 输出电压最大为 2 倍的输入参考电压

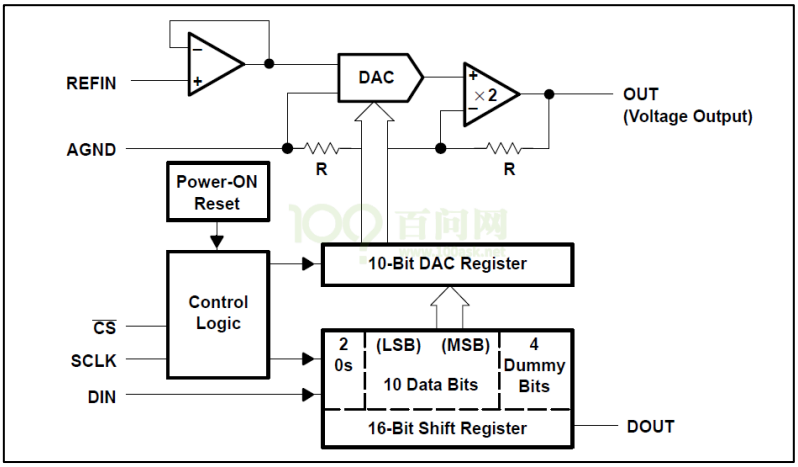
⑥ 内部有上电复位电路

⑦ 低电源功耗，最大 1.75mW

⑧ 最大更新速度为 1.21MHz

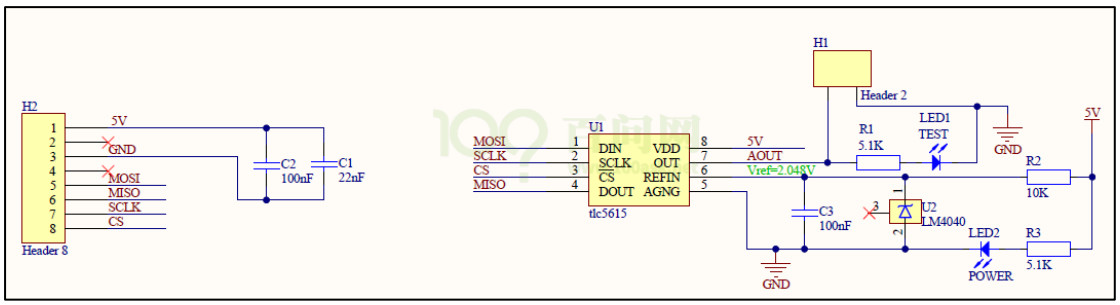
TLC5615 使用简单，只需要提供 5V 供电和外部参考电压即可，无需其他配置。主控通过三线 SPI 连接 TLC5615，将 10bit 的数据发送给 TLC5615；TLC5615 将数值转换为电压输出。

TLC5615 内部框图如下图所示：



百问网 DAC 模块的设计电路如下图所示，通过外部连接器提供 5V 供电电压。

该模块中，TLC5615 从 SPI 接口得到数据后，经过 DAC 转换为电压值，用来驱动 LED1。我们可以通过 LED 灯的亮度来判断 DAC 模块的输出电压高低，亮度越大，表示输出电压越高。

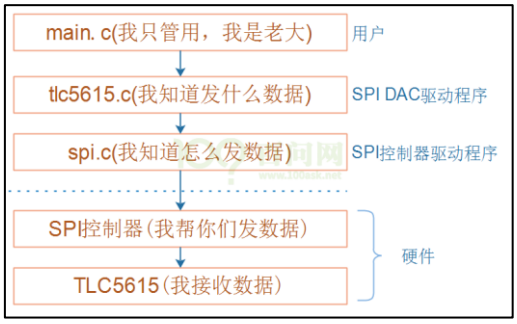



## 24.3 DAC 模块软件设计

### 24.3.1 程序框架

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/27_DAC 模块/spi_dac”目录下。

所涉及文件如下图：




spi.c 中实现了 SPI 的操作，在前面的章节有 SPI 的详细介绍；这不是本章的重点。重点在于 tlc5615.c，它实现了 DAC 的操作：怎么初始化、怎么发数据。

### 24.3.2 DAC 操作方法

向 SPI DAC 模块直接传输 10 位数据，即可控制电压输出。电压计算如下，不同的参考电压会导致不同的输出电压，我们使用 2.048V 参考电压，最大可以输出 4.096V 电压。

| INPUT (1) | | | OUTPUT |
|---|---|---|---|
| 1111 | 1111 | 11(00) | $2(V_{REFIN})\frac{1023}{1024}$ |
| : | | | : |
| 1000 | 0000 | 01(00) | $2(V_{REFIN})\frac{513}{1024}$ |
| 1000 | 0000 | 00(00) | $2(V_{REFIN})\frac{512}{1024} = V_{REFIN}$ |
| 0111 | 1111 | 11(00) | $2(V_{REFIN})\frac{511}{1024}$ |
| : | | | : |
| 0000 | 0000 | 01(00) | $2(V_{REFIN})\frac{1}{1024}$ |
| 0000 | 0000 | 00(00) | 0 V |

(1) A 10-bit data word with two bits below the LSB bit (sub-LSB) with 0 values must be written since the DAC input latch is 12 bits wide.

由于 TLC5615 是 10 位 DAC,它允许主控每次发送 12 位或者 16 位的数据，12 位和 16 位的发送数据格式要求如下图所示。

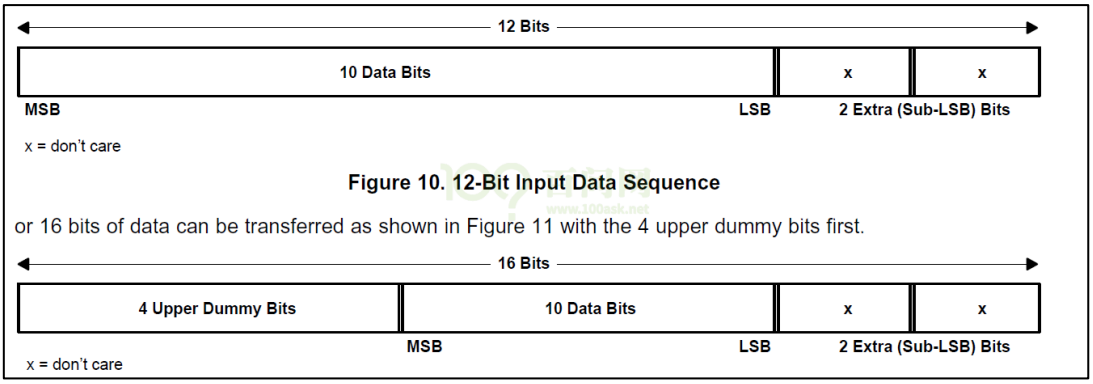



Figure 10. 12-Bit Input Data Sequence

or 16 bits of data can be transferred as shown in Figure 11 with the 4 upper dummy bits first.




## 24.3.3 初始化 SPI1 控制器

SPI DAC 模块使用 SPI1 控制器，我们选择每次发送 12 位数据，时钟频率为 500KHz。

相关的 SPI1 控制器初始化代码在程序文件“spi.c”的 spi_init 函数，如下：

```
unsigned char  spi_init(SPI_CTRL *uc_num)
{
    /*
        1、清除 CONREG 寄存器的 EN 位 来复位模块
        2、在 ccm 中使能 spi 时钟
        3、配置 control register，然后设置 CONREG 的 EN 位来使 spi 模块退出复位
        4、配置 spi 对应的 IOMUX 引脚
        5、根据外部 spi 设备规格来合适的配置 spi 寄存器

    */
        printf("spi 初始化开始\n\r") ;

    uc_num->CONREG =  0;// clear all bits
    /*
        bit0:使能 SPI
        bit3:写入 TXDATA 之后，立即发送
        bit4:设置通道 0 为 master mode
        bit31:20 设置 burst length ，11 表示为 12bits，即每次发送 12 位数据
    */
    uc_num->CONREG |= (11<<20)|(1<<4)|(1<<3)|(1<<0);

    /*  CONFIGREG 采用默认设置
        *
        *bit0        PHA=0
        *bit7:4  sclk 高电平有效
        *bit11:8 通道片选信号，当 SMC =1 的时候，无效（当前处于 SMC=1 模式）
        *bit15:12    POL=0
        *bit19:16    数据线空闲为高电平
        *bit23:20    sclk 空闲为低电平
        *bit28:24    设置消息长度 ，该产品不进行使用
        *
    */
    uc_num->CONFIGREG = 0;//
    /*设置时钟相关的*/
    /*
```

```
    从 RM 手册 chapter18 中，我们得知时钟来源为 PLL3
    1、pll3_sw_clk_sel 为 0，则选择 pll3；为 1 则选择 ccm_pll3_bys，时钟默认选择 pll3。
输出 pll3_sw_clk 给 spi 进行使用，输出给 spi 的时钟为 480M/8=60Mhz
    2、我们需要使能 spi 的时钟进行使用，通过 CCM_CCGR1 的 bit5:2 来进行设置。
这部分在制作.imx 文件的时候初始化，可以不处理。
    3、计算时钟频率 CONREG 寄存器
        bit15:12 div_1
        bit11:8  div_2
    最终提供给 spip 的时钟为：60M/(div+1)*(2^div_2))
    假设我们要使用的时钟是 4M，则我们设置 bit15:12 = 15 bit11:=3 即可，60M/15/2^3 =
0.5Mhz
    */
    uc_num->CONREG &= ~(0xf<<12|0xf<<8);//清除原先的时钟频率设置
    uc_num->CONREG |= (14<<12|3<<8); //设置 clk = 60/(14+1)/2^3 = 0.5M
    printf("spi 初始化结束\n\r");
    //引脚初始化
    iomuxc_sw_set(CSI_DATA5_BASE,3);
    iomuxc_sw_set(CSI_DATA4_BASE,3);
    iomuxc_sw_set(CSI_DATA6_BASE,3);
    iomuxc_sw_set(CSI_DATA7_BASE,3);
    return 0;
}
```

## 24.3.4 DAC 发送数据的函数

初始化 SPI 控制器之后，我们就可以对 TLC5615 写入数据了，通过写入 TLC5615 来设置 DAC 输出电压大小。

发送数据给 DAC 的代码在文件“tlc5615.c”中，是 tlc5615_write_addr 函数，如下：

```
void tlc5615_write_addr(unsigned short uc_data)
{
/*
* 一共 10 位数据，其中低 2 位和高四位需要为 0,在调用该函数之前已经进行处理，因此此处不操作写
入的数据
* 按照写入 12 位数据，需要将写入的数据左移位，最低 2 位保持位 0
*/
    spi_writeread(ESCPI1_BASE,uc_data);
}
```

## 24.3.5 测试函数

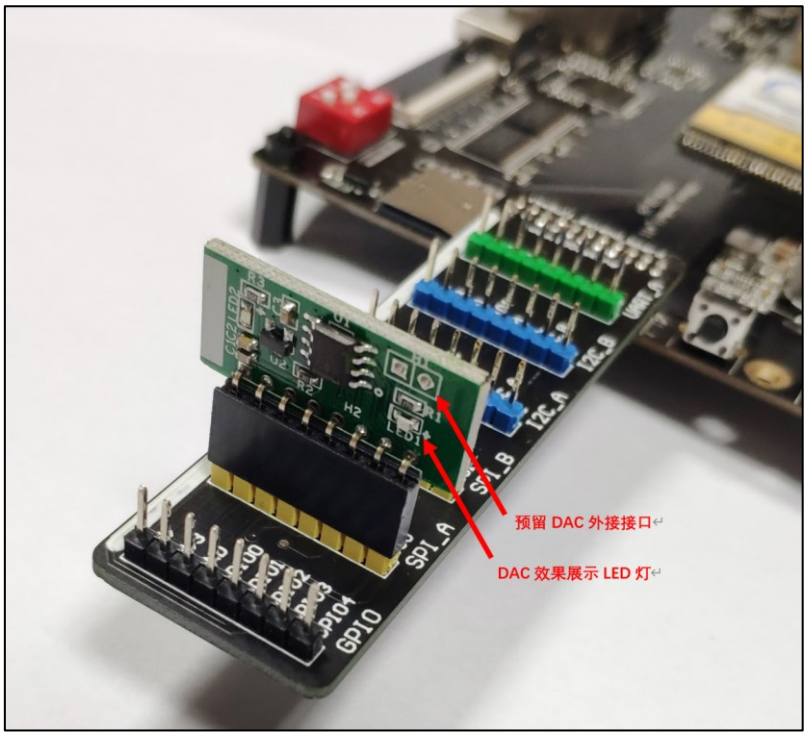
代码在文件“main.c”的 main 函数中，如下：

```
#include "uart.h"
#include "spi.h"
#include "my_printf.h"
#include "tlc5615.h"

void delay_x(unsigned short uc_x)
{
    unsigned short x,y;
    for(x=0;x<uc_x;x++)
    {
        for(y=0;y<1000;y++);
    }
}

int  main()
```

```
{
    unsigned short uc_cnt=0;
    unsigned short key_read_x=0;
    unsigned short key_read_his=0;
   Uart_Init();
   tlc5615_init();//初始化 TLC5615
    while(1)
    {
        for(uc_cnt=0;uc_cnt<1024;uc_cnt+=50)
        {
            printf("write data %d\n\r",uc_cnt);
            tlc5615_write_addr(uc_cnt<<2);
            delay_x(1000);
        }
    }
    return 0;
}
```

## 24.4 DAC 模块测试

IMX6ULL 先断电，按下图所示，将模块插在扩展板的 SPI_A，将扩展板插在底板上：




**注意：为了防止用户接错方向，模块和扩展板都有一条长白线，连接时需要模块上的白线和扩展板的白线在同一侧。**

编译、运行程序，打开串口观察。可以看到 LED1 在逐渐从灭到变亮，依次循环，同时串口打印 DAC 值。

# 第25章 EEPROM 模块

模块原理图及资料：网盘开发板配套资料“05_Hardware（原理图)/Extend_modules/eeprom.zip”。

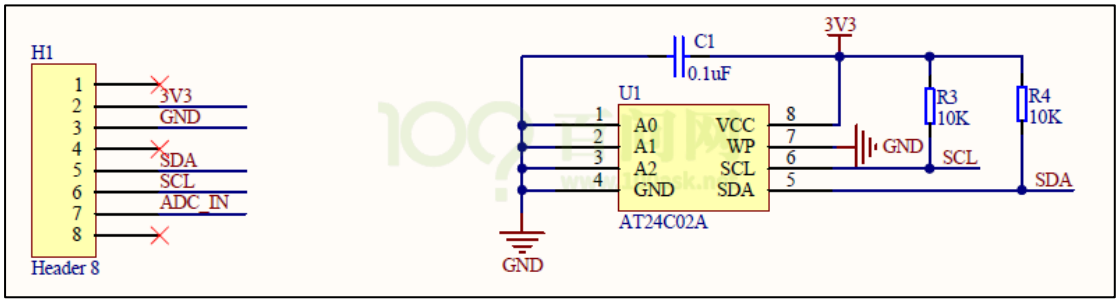
## 25.1 AT24C02 简介

AT24C02 是基于 I2C 总线的存储器件，由于接口方便，体积小，数据掉电不丢失等特点，在仪器仪表及工业自动化控制中得到大量的应用。

百问网提供的 EEPROM 模块使用的就是 AT24C02，使用 8 位地址，存储容量为 2K bit，即 2048bit = 256*8bit = 256 Byte，其中它被分为 32 页，每页 8Byte。

## 25.2 EEPROM 模块硬件设计

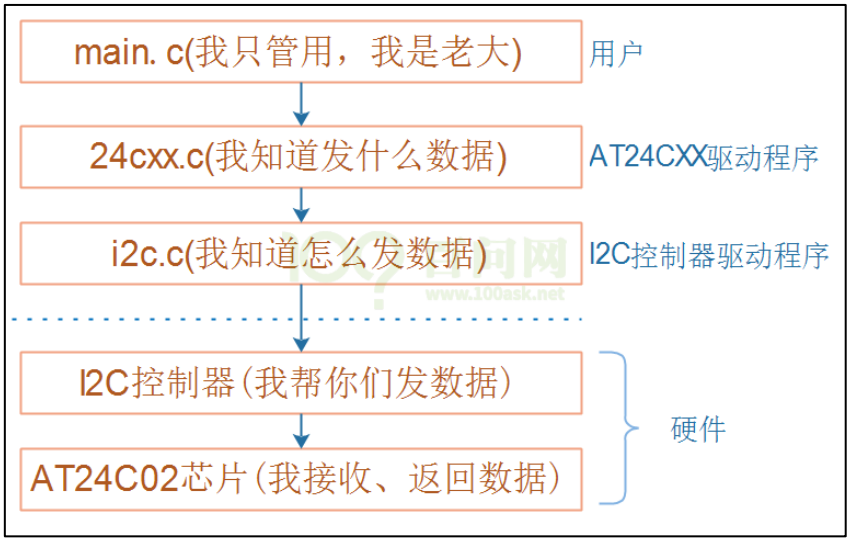
EEPROM 模块的原理图如下：




## 25.3 EEPROM 模块软件设计

### 25.3.1 程序框架

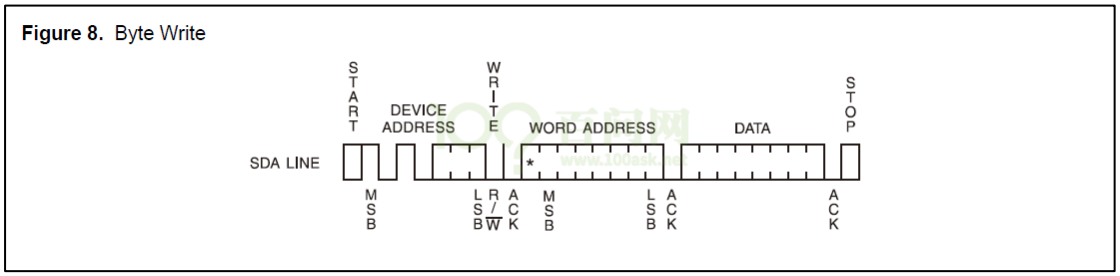
代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/28_EEPROM 模块/i2c_eeprom”目录下。所涉及文件如下图：



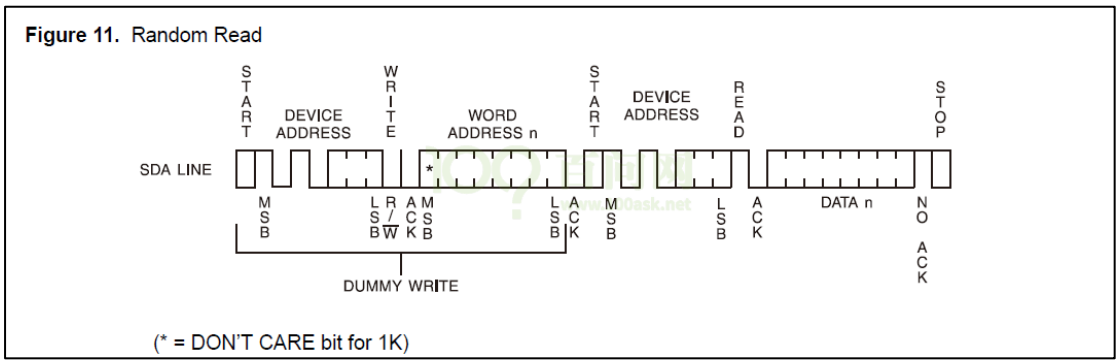

i2c.c 中实现了 SPI 的操作，在前面的章节有 SPI 的详细介绍；这不是本章的重点。重点在于 24cxx.c，它实现了 EEPROM 的读写操作。

## 25.3.2 AT24C02 操作方法

基于 I2C 接口的设备操作，我们主要关心的是 I2C 数据格式。AT24C02 作为 EEPROM 存储设备，显然我们关心的是：怎么读写某个地址，即怎么发地址、怎么读写数据。查阅《AT24CXX.pdf》手册，要写入一个字节，可如下操作：先发出设备地址，再发出 WORD 地址(即存储地址)，再发出数据。




反之，要读 AT24C02，如下图：涉及 2 次 I2C 传输。先发出设备地址，WORD 地址；再次发出设备地址，读到数据。




## 25.3.3 AT24C02 读函数

调用 i2c_read_one_byte 来读 AT24C02，代码在 24cxx.c 中，如下：

```
void AT24CXX_Read(unsigned short ReadAddr, unsigned char *pBuffer, unsigned short
NumToRead)
{
    while(NumToRead) /* 循环发送 */
    {
        *pBuffer++ = i2c_read_one_byte(AT24CXX_ADDR, ReadAddr++);
        NumToRead--;
    }
}
```

AT24CXX_Read 函数用来进行连续内存块读操作，ReadAddr 为起始读出地址，pBuffer 用来保存读出的数据，NumToRead 为需要读出的数据长度。

i2c_read_one_byte 函数为 I2C 的 1 个字节读操作，具体参考前面的 I2C 章节。

## 25.3.4 AT24C02 写函数

调用 i2c_write_one_byte 来写 AT24C02，代码在 24cxx.c 中，如下：

```
void AT24CXX_Write(unsigned short WriteAddr,unsigned char *pBuffer,unsigned short
NumToWrite)
{
    while(NumToWrite--)
    {
        i2c_write_one_byte(AT24CXX_ADDR, WriteAddr, *pBuffer);
```

```
        WriteAddr++;
        pBuffer++;
    }
}
```

AT24CXX_Write 函数用来进行连续内存块写操作，WriteAddr 为写入起始地址，pBuffer 为保存需要写入的数据，NumToRead 为需要写入的数据长度。

i2c_write_one_byte 函数为 I2C 的 1 个字节写操作,具体参考前面的 I2C 章节。

### 25.3.5 测试函数

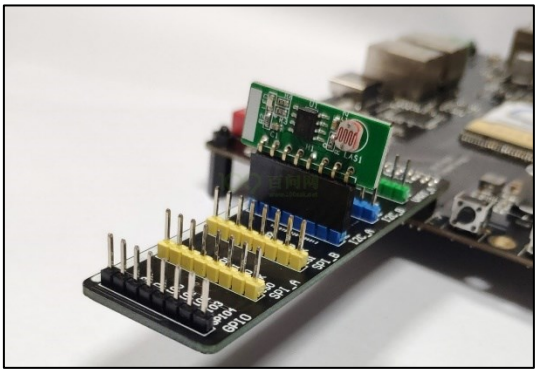
main 函数部分如下：

```
while(1) {

    // 24cxx
    if(read_state)
    {
        read_state = 0;
        printf("Start Read 24CXX....    \r\n\r\n");
        AT24CXX_Read(0, print_data, TEXT_SIZE);
        printf("The Data Readed Is: %s \r\n\r\n", print_data);
    } else if(write_state) {
        write_state = 0;
        printf("Start Write 24CXX....   \r\n\r\n");
        AT24CXX_Write(0, (unsigned char *)TEXT_Arr, TEXT_SIZE);
        printf("24CXX Write Finished!   \r\n\r\n");
    }

    if(temp){
        GPIO5->DR &= ~(1<<3);
    } else {
        GPIO5->DR |= (1<<3);
    }
    temp = !temp;
    gpt2_chan1_delay_us(1000000);
}
```

read_state 和 write_state 分别由两个按键中断控制，当 read_state 为 1 时读取对应的地址数据，write_state 为 1 时数据写入相对应的地址。

## 25.4 EEPROM 模块测试

IMX6ULL 先断电，按下图所示，将模块插在扩展板的 I2C_A，将扩展板插在底板上：



**注意：为了防止用户接错方向，模块和扩展板都有一条长白线，连接时需要模块上的白线和扩展板的白线在同一侧。**

**注意：本实验中，AT24C02 模块要插到 I2C_A 接口，如下图所示。**

编译、运行程序；打开串口观察，分别按下 KEY1 和 KEY2，可看出打印后的数据，如下图所示。

```
key 1 is release
Exception! cpsr is 0x200001d3
swi exception
SWI val = 0x123
Exception! cpsr is 0xa00000
swi exception
The Data Readed Is:

key 1 is press
key 1 is release
Start Write 24CXX....

24CXX Write Finished!

key 2 is press
key 2 is release
Start Read 24CXX....

The Data Readed Is: IMX6ULL 24CXX TEST!
```

# 第26章 GPS 模块

模块原理图及资料：网盘开发板配套资料“05_Hardware (原理图)/Extend_modules/gps.zip”。

## 26.1 GPS 简介

全球定位系统(Global Positioning System，GPS)是一种以空中卫星为基础的高精度无线电导航的定位系统，它在全球任何地方以及近地空间都能够提供准确的地理位置、车行速度及精确的时间信息。GPS 主要由三大组成部分：空间部分、地面监控部分和用户设备部分。GPS 系统具有高精度、全天候、用广泛等特点。

太空卫星部分由多颗卫星组成，分成多个轨道，绕行地球一周约 12 小时。每个卫星均持续发射载有卫星轨道数据及时间的无线电波，提供地球上的各种接收机来应用。

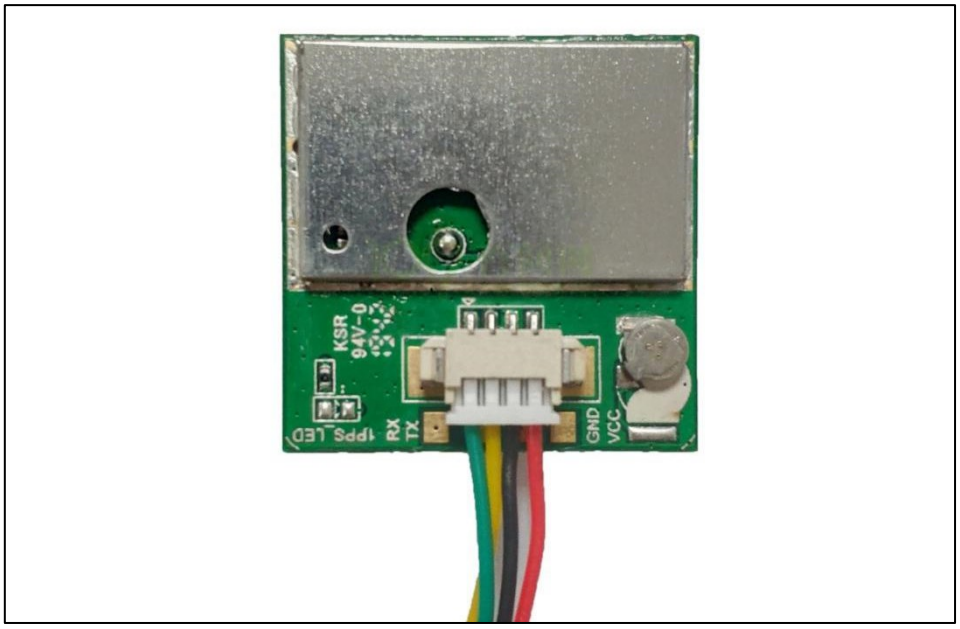
地面管制部分，这是为了追踪及控制太空卫星运行所设置的地面管制站，主要工作为负责修正与维护每个卫星能够正常运转的各项参数数据，以确保每个卫星都能够提供正确的讯息给使用者接收机来接收

使用者接收机（即用户设备），追踪所有的 GPS 卫星，并实时的计算出接收机所在位置的坐标、移动速度及时间。我们日常接触到的是用户设备部分，这里使用到的 GPS 模块即为用户设备接收机部分。

## 26.2 GPS 模块硬件设计

GPS 模块与外部控制器的通讯接口有多种方式，这里我们使用串口进行通讯，波特率为 9600bps,1bit 停止位，无校验位，无流控，默认每秒输出一次标准格式数据。

GPS 模块外观如下图所示，通过排线与控制器进行供电和通讯。该模块为集成模块，暂无相关原理图。

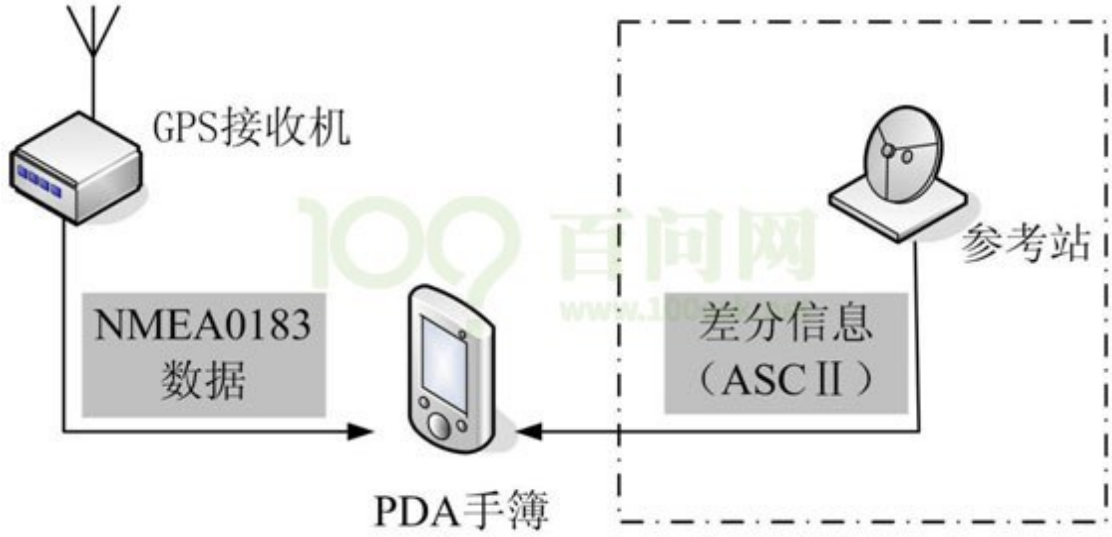


# 26.3 GPS 模块软件设计

## 26.3.1 GPS 数据格式

GPS 使用多种标准数据格式，目前最通用的 GNSS 格式是 NMEA0183 格式。NMEA0183 是最终定位格式，即将二进制定位格式转为统一标准定位格式，与卫星类型无关。这是一套定义接收机输出的标准信息，有几种不同的格式，每种都是独立相关的 ASCII 格式，逗点隔开数据流，数据流长度从 30-100 字符不等，通常以每秒间隔持续输出。

NVMEA0183 格式主要针对民用定位导航，与专业 RTCM2.3/3.0 和 CMR+的 GNSS 数据格式不同。通过 NMEA0183 格式，可以实现 GNSS 接收机与 PC 或 PDA 之间的数据交换，可以通过 USB 和 COM 口等通用数据接口进行数据传输，其兼容性高，数据传输稳定。这里我们使用串口进行是通讯，通信框图如下图所示。




我们使用串口接收数据，收到的数据包含：$GPGGA（GPS 定位数据）、$GPGLL（地理定位信息）、$GPGSA（当前卫星信息）、$GPGSV（可见卫星状态信息）、$GPRMC（推荐最小定位信息）、$GPVTG（地面速度信息）。

这里我们只分析$GPGGA (Global Positioning System Fix Data)即可，它包含了 GPS 定位经纬度、质量因子、HDOP、高程、参考站号等字段。其标准格式如下：

```
$GPGGA,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,M,<10>,M,<11>,<12>*hh<CR><LF>
```

$GPGGA 语句各字段的含义和取值范围各字段的含义和取值范围见下表所示。

| 字段 | 含义 | 取值范围 |
|---|---|---|
| <1> | UTC时间hhmmss.ss | 000000.00~235959.99 |
| <2> | 纬度，格式：ddmm.mmmm | 000.00000~8959.9999 |
| <3> | 南北半球 | N北纬<br>S南纬 |
| <4> | 经度格式dddmm.mmmm | 00000.0000~17959.9999 |
| <5> | 东西半球 | E表示东经<br>W表示西经 |
| <6> | GPS状态 | 0=未定位<br>1=GPS单点定位固定解 |

| | | |
|---|---|---|
| | | 2=差分定位<br>3=PPS解<br>4=RTK固定解<br>5=RTK浮点解<br>6=估计值<br>7=手工输入模式<br>8=模拟模式 |
| <7> | 应用解算位置的卫星数 | 00~12 |
| <8> | HDOP<br>水平图形强度因子 | 0.500~99.000（大于6不可用) |
| <9> | 海拔高度 | -9999.9~99999.9 |
| <l0> | 地球椭球面相对大地水准面的高度<br>（高程异常） | -9999.9~99999.9 |
| <11> | 差分时间 | 从最近一次接收到差分信号开始的秒数，如果不是差分定位将为空 |
| <12> | 参考站号 | 0000~1023；不使用DGPS时为空 |

例子：

```
$GPGGA，074529.82，2429.6717，N，11804.6973，E，1，8，1.098，42.110,,, M,, *76
```

## 26.3.2 编程思路

代码:GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/29_GPS 模块/uart_gps”目录下。

在实验中，我们使用串口 6 与 GPS 模块进行通讯，使用串口 1 将数据打印出来给我们观看。

通过串口 6 来读 GPS 数据，解析出位置信息，再通过串口 1 打印出来。

## 26.3.3 串口初始化函数

串口初始化函数为 Uart_Init，其中串口 1 波特率为 115200，串口 6 为 9600。

相关的代码在程序文“uart.c”中，如下：

```
void Uart_Init(UART_Type *uart_x,unsigned char uc_baudrate)
{

    if(uart_x==UART1)
    {
        IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA       = (volatile unsigned int
*)(0x20E0084);
        IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA       = (volatile unsigned int
*)(0x20E0088);

        *IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA = 0;
        *IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA = 0;

    }
    if(uart_x==UART6)
    {
        IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA       = (volatile unsigned int
```

```
*)(0x20E01D4);
        IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA       = (volatile unsigned int
*)(0x20E01D8);

        *IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA = 8;
        *IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA = 8;

        IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA       = (volatile unsigned int
*)(0x20E064C);
        *IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA = 03;
    }
    uart_x->UCR1 |= (1 << 0) ;        /*关闭当前串口*/

    /*
     *  设置 UART 传输格式：
     *  UART1 中的 UCR2 寄存器关键 bit 如下
     *  [14]:    1：忽略 RTS 引脚
     *  [8] :    0: 关闭奇偶校验 默认为 0，无需设置
     *  [6] :    0: 停止位 1 位     默认为 0，无需设置
     *  [5] :    1: 数据长度 8 位
     *  [2] :    1: 发送数据使能
     *  [1] :    1: 接收数据使能
     */
    uart_x->UCR2 |= (1<<14) |(1<<5) |(1<<2)|(1<<1);

    /*
     *  UART1 中的 UCR3 寄存器关键 bit 如下
     *  [2]:  1:根据官方文档表示，IM6ULL 的 UART 用了这个 MUXED 模型，提示要设置
     */

    uart_x->UCR3 |= (1<<2);

    /*
     * 设置波特率
     * 根据芯片手册得知波特率计算公式:
     * Baud Rate = Ref Freq / (16 * (UBMR + 1)/(UBIR+1))
     * 当我们需要设置 115200 的波特率
     * UART1_UFCR [9:7]=101，表示不分频，得到当前 UART 参考频率 Ref Freq ： 80M ，
     * 带入公式：115200 = 80000000 /(16*(UBMR + 1)/(UBIR+1))
     *
     * 选取一组满足上式的参数：UBMR、UBIR 即可
     *
     * UART1_UBIR    = 71
     * UART1_UBMR = 3124
     */
    if(uc_baudrate==1)  //15200
    {
        uart_x->UFCR = 5 << 7;       /* Uart 的时钟 clk：80MHz */
        uart_x->UBIR = 71;
        uart_x->UBMR = 3124;
    }
    else  //9600
    {
         uart_x->UFCR = 5 << 7;       /* Uart 的时钟 clk：80MHz */
        uart_x->UBIR = 5;
        uart_x->UBMR = 3124;
    }
```

```
    uart_x->UCR1 |= (1 << 0);     /*使能当前串口*/
}
```

## 26.3.4 读取 GPS 信息的函数

我们首先通过串口 6 将每帧数据独立的保存下来，每帧数据以‘$’开头，以 0x0a 结尾。

这部分相关代码在程序文件“uart.c”的 GetStr 函数，如下：

```
void GetStr(UART_Type *uart_x, unsigned char *uc_read_cnt_x)
{
static   unsigned char uc_read =0;
static   unsigned char uc_rec_cnt=0;
    uc_read = GetChar(uart_x);
    if(uc_read=='$')                    //表示一帧 GPS 数据的开头
    {
        uc_rec_cnt = 0;
        flag_rec_ok = 0;
    }
    buf[uc_rec_cnt]=uc_read;
    uc_rec_cnt++;
    PutChar(UART1,uc_read); //打印得到的 GPS 模块传送过来的信息

    if(uc_read ==0x0a)                      //表示一帧 GPS 数据的结尾
    {
        PutChar(UART1,'S');                 //发送特定数据，表示收到 一帧数据
        flag_rec_ok = 1;                //表示收到一帧完整的数据
        buf[uc_rec_cnt]='\0';           //将缓存末尾清 0
        *uc_read_cnt_x = uc_rec_cnt;  //传递接收到的帧数据长度
    }
}
```

## 26.3.5 提取信息的函数

将每帧数据接收下来之后，需要进行相关信息的提取。根据数据格式可以得知，每组信息以‘,’为间隔，因此我们编写‘,’位置计算函数。

相关代码在程序文件“uart.c”的 get_pos_dot 函数中，下面是对应的代码：

```
unsigned char  get_pos_dot(unsigned char *uc_buf,unsigned char uc_cnt)
{
    unsigned char *buf_x;
    buf_x = uc_buf;
    while(uc_cnt)
    {
        if(*buf_x==',') uc_cnt--;//如果是逗号，则减一
        buf_x++;//缓存地址+1
    }
    return buf_x-uc_buf;

}
```

得到‘,’提取函数之后，我们只需要计算每两个‘,’之间的字符内容即可得到相关信息。

具体代码在程序文件“uart.c“的 print_nmea_get_data 函数中，下面是对应的代码：

```
unsigned char print_nmea_get_data(unsigned char *uc_buf,unsigned char uc_posi)
{
```

```
        unsigned char uc_cnt =0;
        unsigned char uc_pos[24];
        unsigned char uc_pos_x =0;
        unsigned char uc_pos_y =0;
        unsigned char uc_len = 0;

        memset(uc_pos,24,0);
        uc_pos_x=get_pos_dot(uc_buf,uc_posi);//得到指定位置的逗号位置
        uc_pos_y=get_pos_dot(uc_buf,uc_posi+1);//得到指定位置+1 的逗号位置

        uc_len  = uc_pos_y-uc_pos_x;//计算两个逗号之间的数据长度

        for(uc_cnt=0;uc_cnt<uc_len;uc_cnt++)//将两个逗号之间的内容进行拷贝，就是实际要
获取的相关内容
            uc_pos[uc_cnt]=uc_buf[uc_cnt+uc_pos_x];

        uc_pos[uc_len-1]='\0';
        PutStr(UART1,uc_pos);

}
```

## 26.3.6 解析信息的函数

得到内容提取函数之后，我们就可以针对$GPGGA 进行操作了。

相关代码在程序文件“uart.c”的 anglysysrawdata 函数中，下面是对应的代码：

```
void anglysysrawdata(unsigned char *uc_buf)
{/*提取定位信息*/
    unsigned char uc_pos_x =0;
    unsigned char uc_pos_y =0;

    uc_pos_x=get_pos_dot(uc_buf,2);
    uc_pos_y=get_pos_dot(uc_buf,3);
    PutStr(UART1,uc_buf);//打印得到的原始数据
    PutStr(UART1,"\n\r");
    PutChar(UART1,'@');

/*$GPGGA,082559.00,4005.22599,N,11632.58234,E,1,04,3.08,14.6,M,-5.6,M,,*76"*/
    if(uc_pos_x==8)
    {
        PutStr(UART1,"Invalid data, Please move the module to the open area and
wait 3-5 minutes.\n\r");
    }
    else
    {
        PutStr(UART1,"@UTC Time:");
        print_nmea_get_data(uc_buf,1);
        PutStr(UART1,"\n\r");
        PutStr(UART1,"@定位卫星数量:");
        print_nmea_get_data(uc_buf,7);
        PutStr(UART1,"\n\r");
        PutStr(UART1,"@是否成功定位，0 表示没有定位，非 0 表示定位成功:");
        print_nmea_get_data(uc_buf,6);
        PutStr(UART1,"\n\r");
        PutStr(UART1,"@纬度:");
        print_nmea_get_data(uc_buf,3);
        print_nmea_get_data(uc_buf,2);
```

```
        PutStr(UART1,"\n\r");
        PutStr(UART1,"@经度:");
        print_nmea_get_data(uc_buf,5);
        print_nmea_get_data(uc_buf,4);
        PutStr(UART1,"\n\r");
        PutStr(UART1,"@海拔高度:");
        print_nmea_get_data(uc_buf,8);
        PutStr(UART1,"\n\r");
    }
}
```

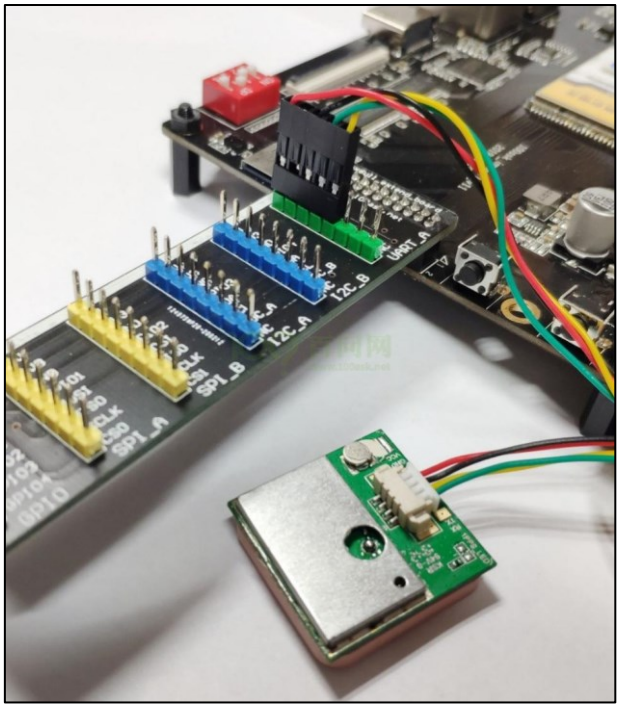
## 26.3.7 测试函数

测试代码在程序文件“main.c”main 函数，下面是对应的代码：

```
#include "uart.h"
#include <string.h>
extern unsigned char flag_rec_ok;//从外部读取，表示读取到一帧 GPS 信息
extern  unsigned char buf[128];//缓存有当前读取到的 GPS 帧信息
int  main()
{
    unsigned char cTestDataX =0;      /*用于记录读取到的数据长度*/

    Uart_Init(UART1,1)    ;//初始化串口 1，设置波特率为 115200

    PutStr(UART1,"Hello, world!\n\r");    /*发送字符串*/
    Uart_Init(UART6,0)    ;//初始化串口 6，设置波特率为 9600
    while(1)
    {
        GetStr(UART6,&cTestDataX);
        if(flag_rec_ok==1)
        {
            flag_rec_ok = 0;
            if(buf[4]=='G')//表示当前读取到的是 GPGGA 的相关内容，进行解析。如果需要其他
GPGSA、GPGSV 等语句的信息，按照相应格式进行提取即可
            {
            //PutStr(UART1,buf);//打印得到的原始数据
            //用于测试固定内容的解析，
//anglysysrawdata("$GPGGA,082559.00,4005.22599,N,11632.58234,E,1,04,3.08,14.6,M,-
5.6,M,,*76");
                anglysysrawdata(buf);//进行实际数据内容的解析
            }
        }
    }
    return 0;
}
```

## 26.4 GPS 模块测试

IMX6ULL 先断电，按下图所示，将模块插在扩展板的 GPIO0，将扩展板插在底板上：




**注意：为了防止用户接错方向，模块和扩展板都有一条长白线，连接时需要模块上的白线和扩展板的白线在同一侧。**

**注意：本实验中，AT24C02 模块要插到 I2C_A 接口。**

编译、运行程序，打开串口观察。如果是首次启动，或者信号不好，则会打印类似的下图信息，**需要将模块移动到空旷的地方**，比如窗户外，重新上电启动后等待 3～5 分钟。

```
@Invalid data, Please move the module to the open area and wait 3-5 minutes.
$GPGLL,,,,,,V,N*64
S$GPGSA,A,1,,,,,,,,,,,,,99.99,99.99,99.99*30
S$GPGSV,1,1,00*79
S$GPRMC,,V,,,,,,,,,,N*53
S$GPVTG,,,,,,,,,N*30
S$GPGGA,,,,,,0,00,99.99,,,,,,*48
S$GPGGA,,,,,,0,00,99.99,,,,,,*48
```

如果非首次启动，且当前位置没有信号或者信号不稳定，则会打印类似的下图信息：

```
@@UTC Time:094517.104
@定位卫星数量:03
@是否成功定位，0表示没有定位，非0表示定位成功:0
@纬度:
@经度:
@海拔高度:99.99
$GPGLL,,,,,,V,N*64
S$GPGSA,A,1,01,11,28,,,,,,,,,,99.99,99.99,99.99*3B
S$GPGSV,2,1,05,11,,,29,01,,,31,28,,,16,07,,,21*7E
S$GPGSV,2,2,05,03,,,23*7E
S$GPRMC,094517.104,V,,,,,,,,,,N*46
S$GPVTG,,,,,,,,,N*30
S$GPGGA,094518.90,,,,,0,03,99.99,,,,,,*6D
S$GPGGA,094518.90,,,,,0,03,99.99,,,,,,*6D
```

待模块在空旷地方等待一会后，就可以实现定位，完全定位好之后的信息打印如下图所示：

```
@@UTC Time:094658.00
@定位卫星数量:04
@是否成功定位，0表示没有定位，非0表示定位成功:1
@纬度:N2239.11677
@经度:E11407.16151
@海拔高度:2.30
$GPGLL,2239.11677,N,11407.16151,E,094658.00,A,A*62
S$GPGSA,A,3,01,11,28,03,,,,,,,,,2.50,2.30,0.99*0C
S$GPGSV,3,1,10,28,58,343,21,30,43,225,16,03,42,098,21,01,32,034,31*7E
S$GPGSV,3,2,10,07,26,189,18,06,24,232,13,11,10,052,29,08,03,081,*7D
S$GPGSV,3,3,10,193,,,,35,,,*45
S$GPRMC,094658.00,A,2239.11677,N,11407.16151,E,0.000,304.89,010420,,,A*64
S$GPVTG,,T,,M,0.000,N,0.000,K,A*23
S$GPGGA,094659.00,2239.11677,N,11407.16151,E,1,04,2.30,92.4,M,-1.9,M,,*79
S$GPGGA,094659.00,2239.11677,N,11407.16151,E,1,04,2.30,92.4,M,-1.9,M,,*79
```

# 第27章 ADC 实验_光敏模块

模块原理图及资料：网盘开发板配套资料“05_Hardware（原理图)/Extend_modules/eeprom.zip”。

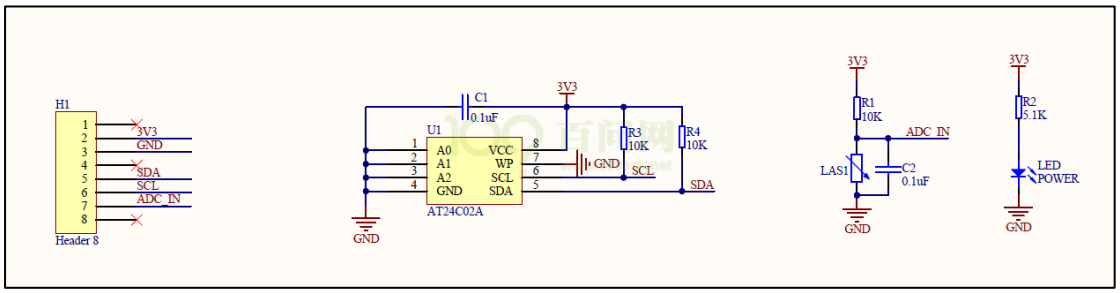
## 27.1 光敏电阻简介

光敏传感器的核心是光敏电阻，常用的制作材料为硫化镉等材料。这些制作材料具有在特定波长的光照射下，其阻值迅速减小的特性。这是由于光照产生的载流子都参与导电，在外加电场的作用下作漂移运动，电子奔向电源的正极，空穴奔向电源的负极，从而使光敏电阻器的阻值迅速下降。随着光照强度的升高，电阻值迅速降低，亮电阻值可小至 1KΩ以下。其在无光照时，呈高阻状态，暗电阻一般可达 1.5MΩ。

利用光敏电阻受光照阻值变化的特性，在光敏电阻两端施加一个电压，使用 ADC 去检测电压的变化，就能得知光照的变化。

## 27.2 光敏模块硬件设计

光敏模块的电路非常简单，因此和 EEPROM 模块做在了一起，光敏模块电路如下：




图中 LAS1 即为光敏电阻，串联了一个 R1，ADC 只要读取两个电阻之间的电压，再通过分压公式就能计算出 LAS1 两端的电压变化。

100ASK_IMX6ULL 引出了两个 ADC，分别是 ADC1 的通道 3 和 4，本次实验采用的是 ADC1_CH3。

# 27.3 IMX6ULL ADC 简介

## 27.3.1 概述

IMX6ULL 中有 2 个 ADC，每个 ADC 都有 16 个通道。它的结构图如下：

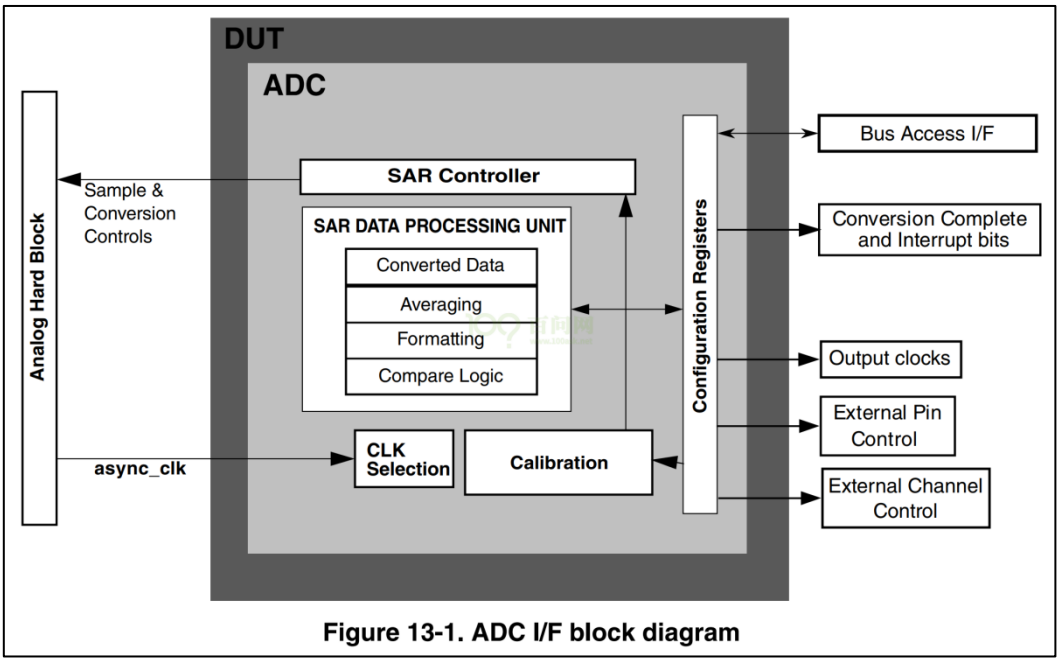



Figure 13-1. ADC I/F block diagram

ADC 精度最大为 12 位，最大采样率是 1M。它还有比较功能，当采样值小于、大于或是等于某个值时，可以设置让它发出中断。

ADC 有 3 种状态：禁止(Disabled)，空闲(IDLE)、正在转换(Performing conversions)。ADC 默认处于禁止状态，需要先配置它才能使用。

为了得到更精确到 ADC 值，需要校准。当 ADC 复位之后，应该进行校准。校准是 ADC 的内部功能。较准之后，就可以使用 ADC 了。

怎么启动 ADC？可以使用软件触发，也可以使用硬件触发。

ADC_HC0 寄存器即可以用于软件触发，也可以用于硬件触发；另一个 ADC_HCn

当 ADC 结束时，结果保存在 ADC_Rn 寄存器中，并且 ADC_HS[COCOn](硬件状态寄存器的转换结束标记)将被设置为 1，如果 ADC_HCn[AIEN]=1 的话还会触发中断。

ADC 还有自动比较功能:把 ADC 转换值跟 ADCx_CV 寄存器中的值进行比较，通过 ADC_GC[ACFE]位来使能比较功能。怎么比较呢？后面会讲。

ADC 还有自动求平均值的功能，这叫硬件平均功能。通过 ADC_GC[AVGE]位来使能。

## 27.3.2 ADC 时钟

ADC 时钟源有 4 个，如下表：

| ADICLK | Selected Clock Source |
|---|---|
| 00 | IPG clock |
| 01 | IPG clock divided by 2 |
| 10 | Reserved |
| 11 | Asynchronous clock (ADACK) |

使用哪个时钟，这可以通过 ADC_CFG[ADICLK]来选择；这个时钟还可以进一步分频，通过 ADC_CFG[ADIV]设置分频系数。

### 27.3.3 参考电压选择

IMX6ULL 中，参考电压没得选，用的是外部引脚 VREFH 和 VREFL 的电压。在 ADC_CFG[REFSEL]中有多个值，但是只有 1 个值可用，其他值是保留值。

### 27.3.4 转换控制

虽然 IMX6ULL 的 ADC_CFG[ADTRG]是用来设置触发方式的，但是它里面只有一个有效值：由软件触发。

所以，开始时，ADC 转换总是由软件触发：往 ADC_HC0 中写入值(选择通道)后，就会启动 ADC 转换。

之后呢？如果设置了 ADC_GC[ADCO]等于 1，它表示连续 ADC 转换，当第 1 次转换结束之后，ADC 会再次自动启动。它会一直自动转换，直接你取消。

如果使能了“hardware averaging”(ADC_GC[AVGE]=1)，也就是硬件求均值，假设你设置为 10 次求平均值，那么使用软件启动第 1 次 ADC 转换后，硬件也会自动启动后 9 次 ADC 转换。

如果既设置了连续转换 ADC_GC[ADCO]，也使能了“hardware averaging”，那 ADC 就会一直进行下去，转换到指定的次数后就算出平均值，然后接着进行下一次转换。

### 27.3.5 转换完成

我怎么知道一个 ADC 转换完成了？有状态位：ADCx_HS[COCO0]，这个状态位比较复杂，分开说明。

① **对于单次传输：**

如果 ADC_GC[ACFE]=0(不比较)，并且 ADC_GC[AVGE]=0(不使用硬件平均)，那么每次 ADC 完成时，ADCx_HS[COCO0]都会被置 1。

② **如果使能了比较功能(ADC_GC[ACFE]=1)：**

ADC 完成后并且比较结果为真时，才设置 ADCx_HS[COCO0]为 1。

③ **如果使能了硬件平均功能(DC_GC[AVGE]=1)：**

假设要求 10 次 ADC 的均值，那只有当 10 次 ADC 都完成，才设置 ADCx_HS[COCO0]为 1。

④ **如果既使能了比较功能，也使能在硬件平均功能：**

假设要求 10 次 ADC 的均值，那么只有 10 次 ADC 都完成，并且比较结果为真时，才设置 ADCx_HS[COCO0]为 1。

**注意：校准结束，或是自测结束时，ADCx_HS[COCO0]也会被置 1。**

**注意：写 ADC_HC0 或读 ADC_R0 时，ADCx_HS[COCO0]会被自动清零。**

### 27.3.6 硬件平均功能

可以设置 ADC_GC[ACFE]=1 使用这个功能，平均功能就是多次启动 ADC，把多次结果求平均。

那要计算多少次 ADC 结果的均值？可以在 ADCx_CFG[AVGS]中设置，有 4、8、16、32 共 4 种取值。

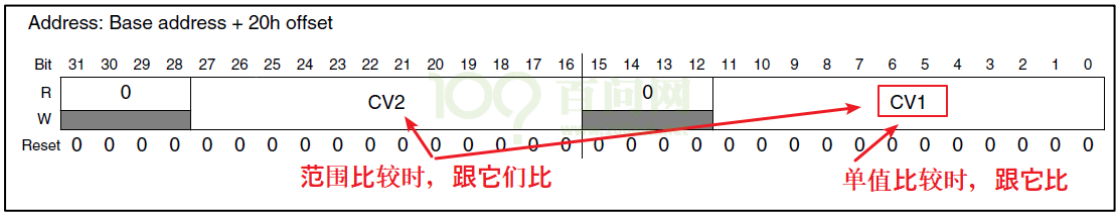
使用这个功能时，第 1 次启动 ADC 后，硬件会自动启动剩余的 ADC 转换。均值结果存放在 ADC_R0 中。

## 27.3.7 比较功能

可以把 ADC 结果跟单个数进行比较：小于、大于、等于。还可以把 ADC 结果进行范围比较，它处于某个范围之内，在是在范围之外？

比较结果为真时，ADCx_HS[COCO0]会被置 1，并触发中断(使能了的话)。

既然要进行比较，就得有被比较的值：




怎么比？当然也有寄存器 ADCx_GC：

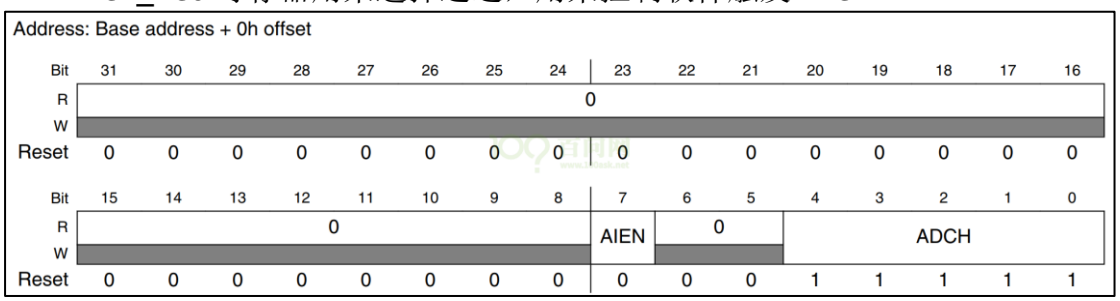
| ACFGT | ACREN | CV1/CV2 | 功能 | 比较结果为真的情况 |
|---|---|---|---|---|
| 0 | 0 | - | Less than threshold | ADC <= CV1 |
| 1 | 0 | - | Greater than or equal to threshold | ADC >= CV1 |
| 0 | 1 | CV1 <= CV2 | Outside range, not inclusive | ADC < CV1 或 ADC > CV2 |
| 0 | 1 | CV1 > CV2 | Inside range, not inclusive | ADC < CV1 && ADC > CV2 |
| 1 | 1 | CV1 <= CV2 | Inside range,inclusive | ADC >= CV1 && ADC <= CV2 |
| 1 | 1 | CV1 > CV2 | Outside range,inclusive | ADC >= CV1 或 ADC <= CV2 |

## 27.4 IMX6ULL ADC 寄存器

本实验使用查询方式使用 ADC，不涉及中断。下面介绍用到的寄存器。

### 27.4.1 控制寄存器 ADCx_HC0

ADCx_HC0 寄存器用来选择通道，用来控制软件触发 ADC。




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [7] | AIEN | R/W | 转换结束时，ADC_HS[COCO0]等于 1，能否产生中断由此位决定：<br>1：转换结束时中断使能；<br>0：转换结束时中断禁止 |
| [4:0] | ADCH | R/W | Input Channel Select，用来选择输入通道，<br>00000-01111：外部通道 0～15，实际上只有 8 个通道；<br>10000-10111：保留；<br>11000：保留； |

| | | | |
|---|---|---|---|
| | | | 11001：VREFSH=内部通道，用于自测，它会连接到 VRH；<br>11010：保留；<br>11011：保留；<br>11100-11110：保留；<br>11111：禁止转换，这会禁止 ADC 并且让外部输入跟内部通道完全隔离 |

## 27.4.2 状态寄存器 ADCx_HS

它的 BIT0 就是 COCO0，用来表示转换是否完成。更多细节请看前面的小章节《转换完成》。

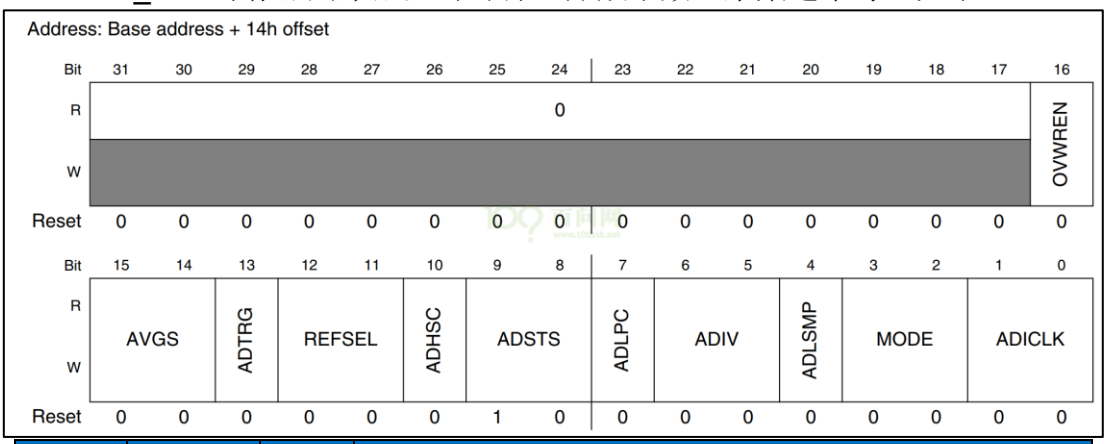
## 27.4.3 数据结果寄存器 ADCx_R0

ADC 的转换结果，保存在这个寄存器里，有效数据从 BIT0 开始存放，用不到的位值为 0，如下图：

| Conversion Mode | Data Result Register bits | | | | | | | | | | | | | | | | Format |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | D31 | D30 | ..... | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | |
| 12b single-ended | 0 | 0 | 0 | 0 | D | D | D | D | D | D | D | D | D | D | D | D | unsigned right justified |
| 10b single-ended | 0 | 0 | 0 | 0 | 0 | 0 | D | D | D | D | D | D | D | D | D | D | unsigned right justified |
| 8b single-ended | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | D | D | D | D | D | D | D | D | unsigned right justified |

## 27.4.4 配置寄存器 ADCx_CFG

ADCx_CFG 寄存器用来配置时钟源、分频系数、采集速率等，如下：



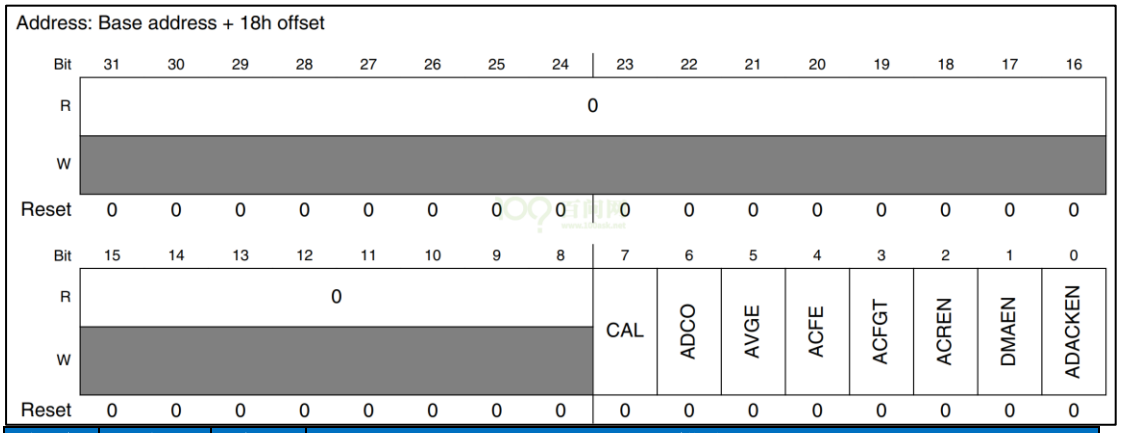

| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [16] | OVWREN | R/W | Data Overwrite Enable，数据能否覆盖，<br>1：可以覆盖；<br>0：不能覆盖 |
| [15:14] | AVGS | R/W | 硬件平均选择，即用多少个采样植来算平均值，仅当 ADC_GC[AVGE]=1 时有效，<br>00：4 个采样值；<br>01：8 个；<br>10：16 个；<br>11：32 个 |

| | | | |
|---|---|---|---|
| [13] | ADTRG | R/W | ADC 触发方式，<br>0：软件触发，目前仅有这种方式，写 ADC_HC0 寄存器时启动 ADC 转换；<br>1：保留 |
| [12:11] | REFSEL | R/W | 选择参考电压，<br>00：VREFH/VREFL 作为参考电压；<br>其他值：保留 |
| [10] | ADHSC | R/W | High Speed Configuration，高速转换，<br>0：正常转换；<br>1：高速转换，这是时 ADC 的内部时钟会比正常状态下的高 |
| [9:8] | ADSTS | R/W | 用来决定采样周期，即采样要花多少个 ADC 时钟。采样越快越精确，但是也越耗电；<br>如果输入电压变化没那么快，那即使采样慢一点也没关系。<br>ADC 有 2 种模式：短或长，ADLSMP=1 时表示“长采样时间”，ADLSMP=0 表示“短采样时间”。<br>ADLSMP=0 时，ADSTS 含义为：<br>00：采样时间为 2 个 ADC 时钟；<br>01：采样时间为 4 个 ADC 时钟；<br>10：采样时间为 6 个 ADC 时钟；<br>11：采样时间为 8 个 ADC 时钟；<br><br>ADLSMP=1 时，ADSTS 含义为：<br>00：采样时间为 12 个 ADC 时钟；<br>01：采样时间为 16 个 ADC 时钟；<br>10：采样时间为 20 个 ADC 时钟；<br>11：采样时间为 24 个 ADC 时钟 |
| [7] | ADLPC | R/W | Low Power Configuration. 让 ADC 进入低电源模式，<br>0：ADC 没进入低电模式；<br>1：ADC 进入低电模式 |
| [6:5] | ADIV | R/W | 生成内部时钟 ADCK 时，使用的分频系数，<br>00：ADCK = 输入时钟；<br>01：ADCK = 输入时钟/2；<br>10：ADCK = 输入时钟/4；<br>11：ADCK = 输入时钟/8 |
| [4] | ADLSMP | R/W | Long Sample Time Configuration，长采样时间，<br>0：短采样时间；<br>1：长采样时间 |
| [3:2] | MODE | R/W | 转换模式，即转换精度，<br>00：8 位；<br>01：10 位； |

| | | | |
|---|---|---|---|
| | | | 10：12 位；<br>11：保留 |
| [1:0] | ADICLK | R/W | 输入时钟选择，<br>00：IPG clock；<br>01：IPG clock/2；<br>10：保留；<br>11：ADACK |

## 27.4.5 通用控制寄存器 ADCx_GC

ADCx_GC 寄存器用来控制校准、转换模式和电压参考选择等。




| 位域 | 名 | 读写 | 描述 |
|---|---|---|---|
| [7] | CAL | R/W | 较准<br>设置这位为 1 时，启动校准；在较准过程中，此位一直是 1；校准结束时，此位被清 0。<br>使用 ADC_GS[CALF]来判断校准结果。<br>校准过程一旦开始，不要中止它，否则结果将无效，ADC_GS[CALF]=1。<br>设置 CAL 位时，当前的转换将被取消。 |
| [6] | ADCO | R/W | Continuous Conversion Enable，连续转换使能，AVGE=1 时，ADCO 的含义为：<br>0：启动转换时，只转换一次；<br>1：启动转换时，会连续地转换 |
| [5] | AVGE | R/W | Hardware average enable，硬件平均使能，<br>0：硬件平均功能被禁止；<br>1：硬件平均功能被使能 |
| [4] | ACFE | R/W | Compare Function Enable，比较功能是否使能，<br>0：禁止；<br>1：使能 |
| [3] | ACFGT | R/W | ACFE 使能后，这位用来决定如何进行比较，更详细的介绍请看前面章节《比较功能》 |
| [2] | ACREN | R/W | 比较功能中范围比较，是否使能，<br>0：禁止；<br>1：使能 |
| [1] | DMAE | R/W | DMA 使能位， |

| | | | |
|---|---|---|---|
| | N | | 0：禁止；<br>1：使能 |
| [0] | ADAC KEN | R/W | 异步时钟(ADACK)输出使能，<br>0：在 ADICLK 中选择 ADACK 时，并且正在进行 ADC 转换时，ADACK 才输出；<br>其他情况下 ADACK 不输出；<br>1：ADACK 一直输出，不管 ADC 状态是什么 |

## 27.5 编程

代码：GIT 下载后在“10_裸机开发/01_100ASK_IMX6ULL 裸机程序/30_光敏模块/adc_photoresistance”目录下。

### 27.5.1 ADC 初始化

刚上电时，ADC 处于禁止状态。需要对它进行初始他，设置时钟、设置参考电压等等。
最重要的是要进行较准，较准流程如下：
① 设置 ADC_CFG，选定 ADC 精度；
② 设置 ADC_GC[CAL]位，启动较准；
③ 检查 ADC_GS[CALF]状态位，如果它等于 1，表示较准失败；
④ 检查 ADC_GC[CAL]位，等待较准结束；
⑤ 当 ADC_GC[CAL]等于 0 时，检查 ADC_GS[CALF](表示是否较准失败)和 ADC_HS[COCO0](表示较准完成)。
代码在 adc.c 中，如下：

```
int adc_init(void)
{

    /* 1、先清零 CFG 寄存器 */
    ADC1->CFG = 0;

    /* 2、采用 VREFH/VREFL 的参考电压，12 位 ADC，ADICLK 的时钟源 */
    ADC1->CFG |= (2 << 2) | (3 << 0);

    /* 3、关闭 DMA，使能 ADACK */
    ADC1->GC = 0;
    ADC1->GC |= 1 << 0;

    /* 4、校准 ADC */
    ADC1->GS |= (1 << 1);                              // 清除 CALF 位，写 1 清零
    ADC1->GC |= (1 << 7);                              // 使能校准功能

    /* 5、校准完成之前 GC 寄存器的 CAL 位会一直为 1，直到校准完成此位自动清零 */
    while((ADC1->GC & (1 << 7)) != 0) {
        if((ADC1->GS & (1 << 1)) != 0) {
            return 2;
        }
    }
    /* 6、校准成功以后 HS 寄存器的 COCO0 位会置 1 */
    if((ADC1->HS  & (1 << 0)) == 0)
        return 1;
    if(ADC1->GS & (1 << 1)) return 1;
```

```
    return 0;
}
```

## 27.5.2 获取 ADC 值

往 ADC_HC0 寄存器中写入值，选择 ADC 通道，即可启动 ADC 转换。当 ADC_HS[COCO0]即 BIT0 等于 1 时，表示转换结束。这时就可以读取 ADC_R0 得到转换值。

代码在 adc.c 中，如下：

```
unsigned int getadc_value(void)
{
    /* 1、配置 ADC 通道 3 */
    ADC1->HC[0] = 0;
ADC1->HC[0] |= (3 << 0);
/* 2、等待转换完成 */
    while((ADC1->HS & (1 << 0)) == 0);
    return ADC1->R[0];
}
```

每调用 getadc_value 函数一次，得到一个 ADC 值，可以多次调用求平均值——这是使用软件求平均值。IMX6ULL 的 ADC 支持硬件平均值。

本程序使用 adc_avr 计算平均值，代码如下：

```
unsigned short adc_avr(unsigned char temp)
{
    unsigned int temp_val = 0, i = temp;
    while(i--) {
        temp_val += adc_value();
        gpt2_chan1_delay_us(5000);
    }
    return temp_val / temp;
}
```

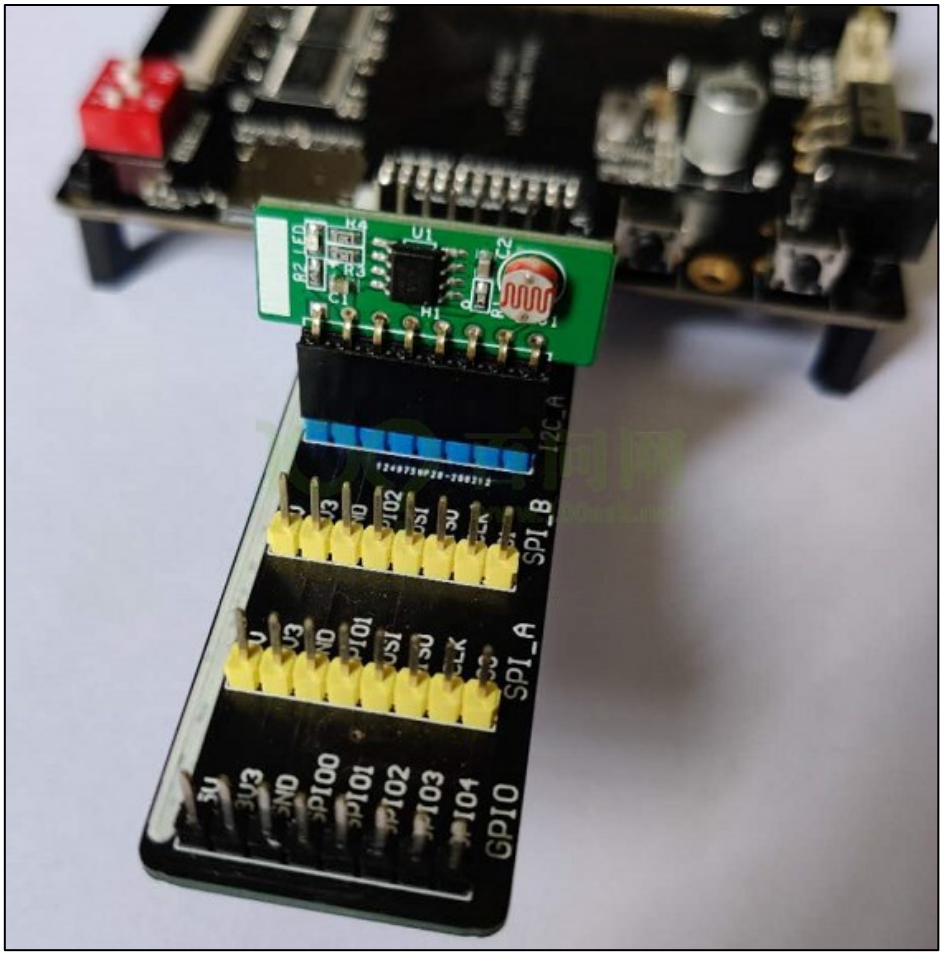
## 27.5.3 测试函数

代码在 main.c 中，关键代码如下：

```
        adcvalue = adc_avr(10);
       lightvalue = 100 - 100 * adcvalue / 4096;
        printf("ADC orig value = %d\r\n", adcvalue);
        printf("Light value = %d \r\n", lightvalue);
```

## 27.6 光敏模块测试

IMX6ULL 先断电，按下图所示，将模块插在扩展板的 I2C_A，将扩展板插在底板上：




**注意：为了防止用户接错方向，模块和扩展板都有一条长白线，连接时需要模块上的白线和扩展板的白线在同一侧。**

编译、运行程序，打开串口观察。将手遮挡光敏电阻，可以看到 ADC 和光照有变化：

```
Light value = 46
ADC orig value = 2237
Light value = 46
ADC orig value = 2240
Light value = 46
ADC orig value = 2733
Light value = 34
ADC orig value = 3706
Light value = 10
ADC orig value = 3701
Light value = 10
ADC orig value = 3699
Light value = 10
```