```
……
gpio_local_init();
gpio_local_enable_pin_output_driver(GPIO_PIN_EXAMPLE);
while (1)
{
    #define INSERT_GPIO_LOCAL_TGL_GPIO_PIN(idx, pin) \
    gpio_local_tgl_gpio_pin(pin);
    MREPEAT
        (
            128,
            INSERT_GPIO_LOCAL_TGL_GPIO_PIN, GPIO_PIN_EXAMPLE
        )
    #undef INSERT_GPIO_LOCAL_TGL_GPIO_PIN
}
……
```

Hello world !

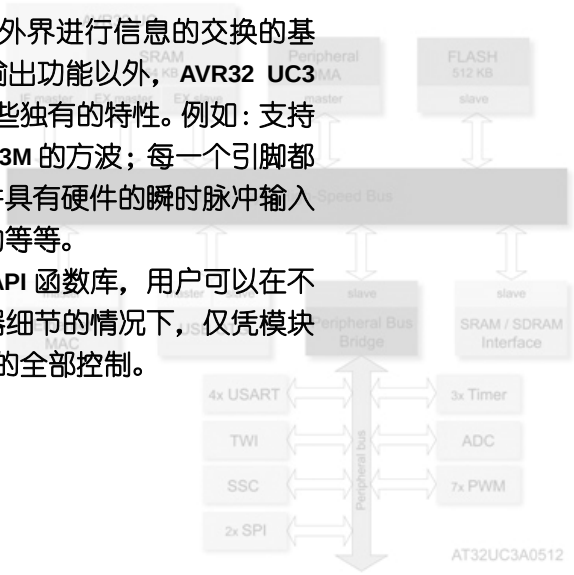
>> GPIO 模块

第八章

本章引言

GPIO 是嵌入式微处理器与外界进行信息的交换的基础通道。除了基本的电平输入输出功能以外，AVR32 UC3 系列还在设计时赋予了 GPIO 一些独有的特性。例如：支持以 CPU 时钟工作，输出最高达 33M 的方波；每一个引脚都可以由电平变化而触发中断，并具有硬件的瞬时脉冲输入滤波功能；4mA 的引脚驱动能力等等。

ATMEL 官方提供了统一的 API 函数库，用户可以在不知晓 AVR32 UC3 底层硬件寄存器细节的情况下，仅凭模块功能和特性信息，完成对 GPIO 的全部控制。



AVR32.UC3.GPIO.Summary 综述

8.1.1 GPIO 模块简介

AVR32 UC3 系列拥有丰富的 GPIO 资源，最多可支持 109 个 IO 同时工作。并且每个 IO 都有以下特性：

A、支持电平变化，上升沿，下降沿三种中断触发模式。

B、支持可编程内部 10K 欧姆上拉电阻。

C、支持用于中断触发的瞬时脉冲滤波器 Input Glitch Filter

D、支持最多 4 个设备功能复用

E、提供 4mA 的电流负载能力

F、可以承受 5V 电压输入

G、支持 66MHz 高速 IO

AVR32.UC3.GPIO.Feature 产品特性

8.2.1 灵活的设备引脚分配

现在的芯片在片上集成了越来越多的外设，然而因为体积的局限，IO 的数量总是有限的，系统往往会因为 IO 分配冲突，而无法发挥所有外设的功能。AVR32 UC3 系列的芯片同样集成了各种丰富的外设，在有限的 IO 空间中，AVR32 UC3 的设计者采用了另一种灵活的 IO 设计策略保证了最大限度发挥外设的功能：相同的引脚功能映射到多个物理 IO 上。

在 AVR32 UC3 的 IO 功能列表上，你会发现在多个引脚上有同样的复用功能，比如 UC3B 系列的以下两个管脚：

表 8-2-1 GPIO 引脚复用示例

引脚号	GPIO 引脚号	复用功能 A	复用功能 B	复用功能 C
PA15	GPIO 15	SPI – SCK	PWM - PWM[4]	USART2 - CLK
PA17	GPIO 17	SPI - NPCS[1]	TC - CLK2	SPI - SCK

注：AVR32 UC3 的 IO 最多可支持 4 个 IO 复用功能，在已推出的系列中 IO 支持 A、B、C 三个复用功能，第四个复用功能未开放，用于以后的扩展

在实际应用中，假如我们在使用的 PA15 的 PWM - PWM[4]功能，因为引脚冲突 SPI 功能因为缺少 SCK 引脚而不能使用，但从表 3-8-1 中我们发现 PA15 的复用功能 A 与 PA17 的复用功能 C 是重复的，都是 SPI-SCK 功能。这就为我们的设计提供了方便，我们可以通过寄存器来设置 PA17 的复用功能为 SPI-SCK。这样就能同时使用 PWM 及 SPI 功能，最大程度的发挥丰富外设的优势。AVR32 UC3 还有其他功能都有类似情况，读者可查询具体的数据手册。这种单功能多映射策略给与设计者相当大的 IO 调整空间，既方便了 IO 功能的分配也为 PCB 设计提供了便利。

8.2.2 中断输入瞬时脉冲滤波器 Glitch Filter

通常我们在一般的 IO 输入应用中，都需要考虑引脚的输入滤波问题。环境的电磁干扰，系统电路的波动以及人为的误操作都有可能产生错误的信号输入。为了减少这类错误的出

现，人们想了很多方法，比如软件滤波、在引脚上加滤波电容等。**AVR32 UC3** 的每个 **IO** 引脚都集成了独立的硬件瞬时脉冲滤波器，在使用 **GPIO** 引脚输入电平中断功能时开启，即可在一定程度上降低错误信号输入的概率（工程上不一定能减少软硬件滤波的开销）。

在 **IO** 的电平中断输入应用中，芯片会在时钟信号上升沿对输入电平进行采样，因为这一过程时间非常短，假如在这个期间发生瞬间的电平异动，就会产生对电平的错误采样。而常见的正常输入电平信号都会保持几个时钟信号周期，**AVR32 UC3** 的输入瞬时脉冲滤波器正是基于这样的原理——增加对电平信号的上跳沿采样数量要求，忽略所有电平保持不足一个时钟周期的脉冲信号。只有在信号保证两个上跳沿都被采样到确定电平时，才能触发相应的引脚电平变化中断。如图 **3-8-1** 所示。

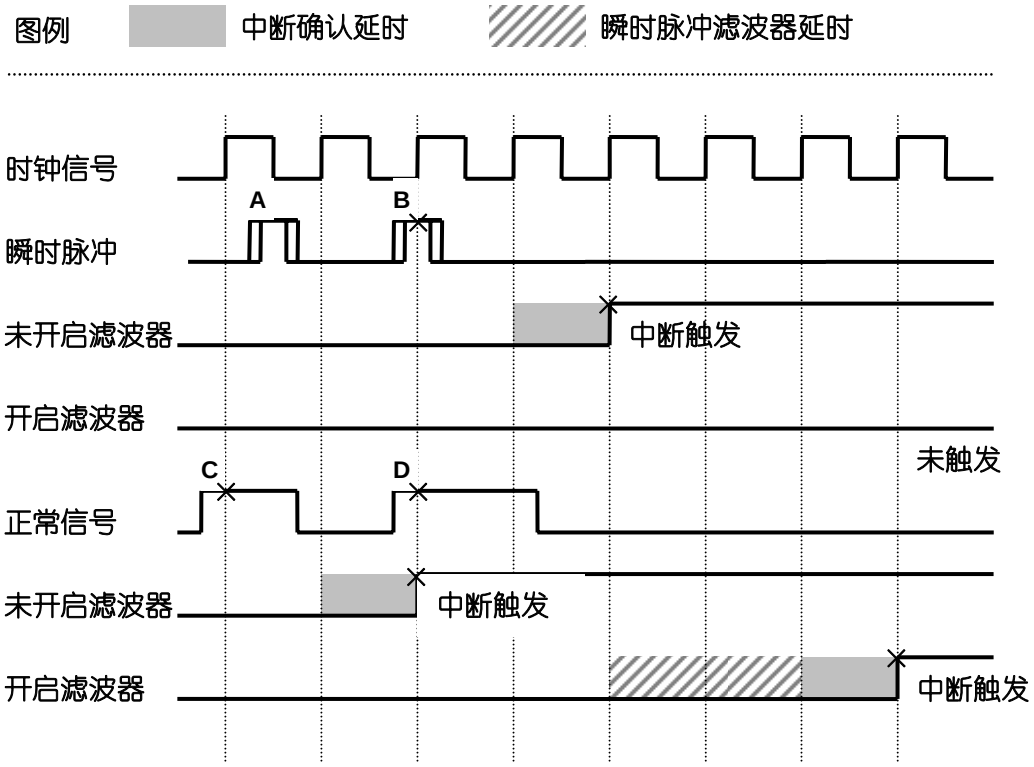


图 **8-2-1 Glitch Filter** 滤波时序演示

在图中，当瞬时脉冲 **B** 在时钟上跳沿被采样到时，系统在中断确认周期后触发中断信号，而当开启滤波器时这样的中断就不会被触发，因为瞬时脉冲保持时间过短没有达到两个时钟上跳沿采样的要求。需要注意的是，开启了滤波器以后，对于保持时间超过一个周期的电平信号但没有满足两个上升沿采样周期的信号同样也无法触发中断，比如图中的信号 **C**。所以，只有电平信号保持两个时钟周期才能 **100%**触发中断。另外，开启了滤波器以后，在确认采样成功的第二个上跳沿以后，相应的中断触发需要增加两个瞬时脉冲滤波器延时才能进入正常地中断确认操作。所以当使用瞬时脉冲滤波器时，从信号跳变开始到最终中断触发需要经过 **5** 个时钟周期（**2** 个确认信号跳变时钟周期**+2** 个瞬时脉冲滤波器延时**+1** 个中断确认延时）。

8.2.3 通过本地总线 Local Bus 操作 GPIO

AVR32 UC3 系列提供两种方式访问 **GPIO**，在默认的情况下 **CPU** 通过设备总线 **PBA** 操作 **GPIO**，虽然理论上最高可以达到 **PBA 33MHz** 的访问速度，但是因为在实际的应用中多个设

图 8-2-2 AVR32 UC3 访问 GPIO 的两种方式

备会同时使用 **PBA** 交换数据，所以不可避免的会造成 **IO** 访问的延时，这在某些高速 **IO** 应用中是不允许的。所以针对高速 **IO** 应用场合，**AVR32 UC3** 系列提供了另外一种名为 **Local Bus** 的访问形式。

Local Bus 是一条与 **CPU** 直接相连的独立总线，仅提供高速 **IO** 使用不用顾虑总线占用的情况，**Local Bus** 可以与 **CPU** 工作在同一频率，因此最高可达到 **66MHz** 的 **IO** 访问速度，足以应对普通高速 **IO** 应用要求。另外，因为 **Local Bus** 是一条独立于芯片主体总线架构的总线，仅支持 **IO** 置位，清除及翻转操作，**GPIO** 的多个复杂功能无法使用，在实际使用中需要注意。

AVR32.UC3.GPIO.Interface	寄存器和硬件接口

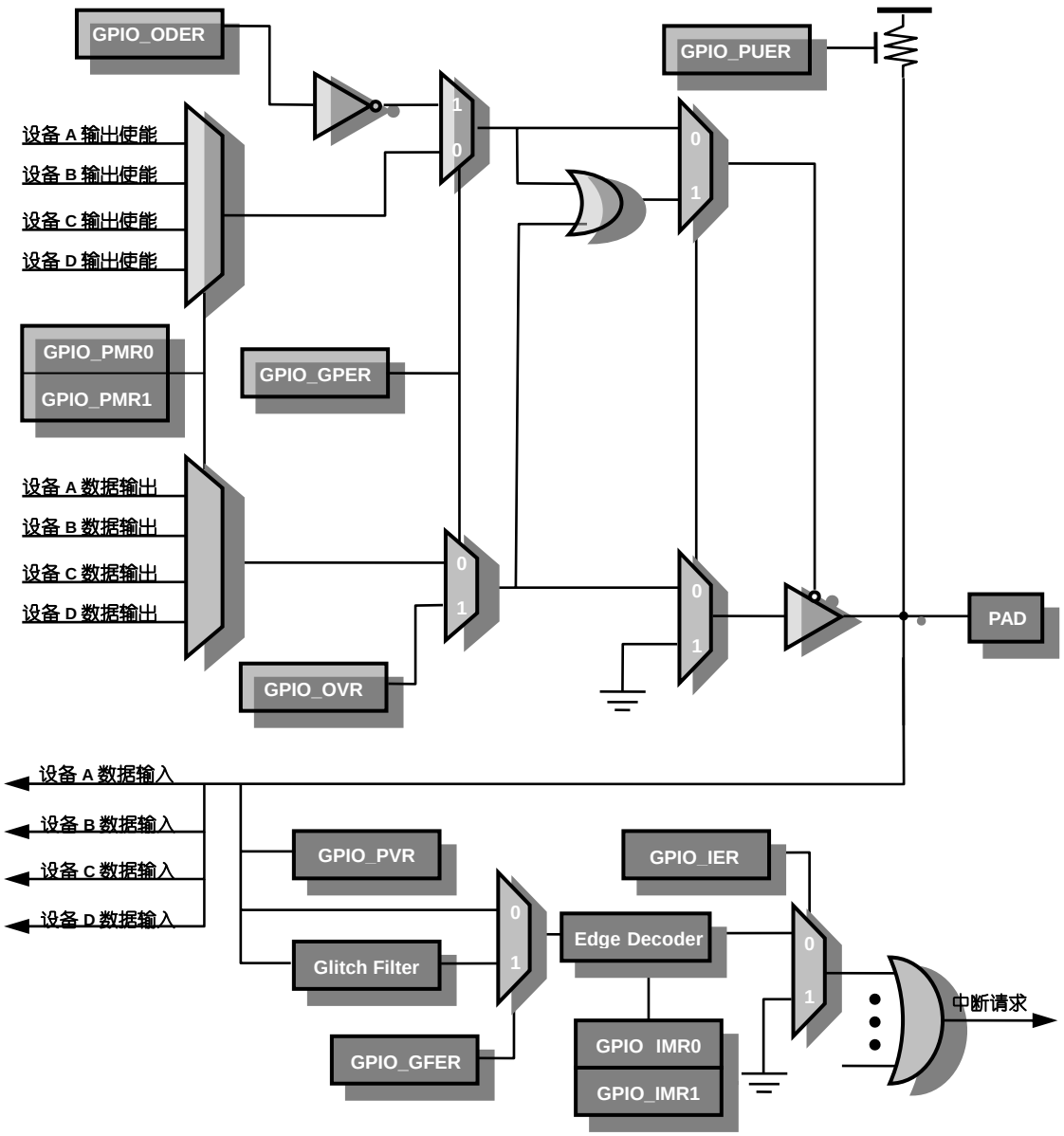


图 8.3.1 GPIO 寄存器逻辑结构图

8.3.1 GPIO 寄存器功能概述

AVR32 UC3 系列中，所有的 **GPIO** 寄存器都是 **32** 位的，其中每一位对应一个引脚。因此，每 **32** 个引脚对应一个端口。

使用 **GPIO** 时，首先需要通过 **GPIO** 使能寄存器 **GPER**（**GPIO Enable Register**）来决定对应得引脚是否使用普通的端口输入输出功能。当我们开启的引脚的普通输入输出功能以后，随时都可以通过寄存器 **PVR**（**Pin Value Register**）来读取引脚上的电平，而不用关心我们是否开启了引脚的电平输出（驱动）功能。**AVR32 UC3** 在复位以后，**GPIO** 默认是工作在这一模式下的。

当GPIO工作在普通的输入输出模式时，默认情况下是关闭引脚电平输出功能的，也就是说，默认情况下我们无法控制引脚输出的电平，而此时引脚处于一个电平并不确定的开漏状态。开漏是一种很重要的状态，著名的I^2C总线就是借助引脚的线与特性来实现设备时钟的同步和主机仲裁，而线与特性正是式借助开漏输出来实现的。在我们试图让引脚输出确定的电平前，必须要借助寄存器ODER（Output Driver Enable Register）来开启GPIO的电平驱动功能。一旦开启了引脚的驱动功能，就可以通过设置寄存器OVR（Output Value Register）实现高低电平的输出。需要补充强调的是，此时仍然可以通过PVR寄存器获取当前引脚上的实际电平。

当寄存器 GPER 对应位置被清零时，GPIO 将作为设备引脚，接受内部模块的控制。AVR32 UC3 系列支持在同一引脚上分别复合最多 4 种不同的设备功能。我们可以利用 PMR0 和 PMR1 两个寄存器进行组合，从 4 种引脚设备功能中进行选择。其中，PMR 是英文 Peripheral Mux Register 的缩写。

无论 GPIO 工作在什么模式下，都可以通过寄存器 PUER 来开启芯片内部的上拉电阻；同样，GPIO 引脚中断的开启也与 GPIO 的当前工作模式无关。寄存器 IER（Interrupt Enbale Reigster）控制对应引脚的使能。AVR32 UC3 系列支持 3 种电平触发方式，依次分别为：边沿触发、上升沿触发和下降沿触发。寄存器 IMR0 和 IMR1 共同完成中断触发方式的选择。其中 IMR 是英文 Interrupt Mode Register 的缩写。

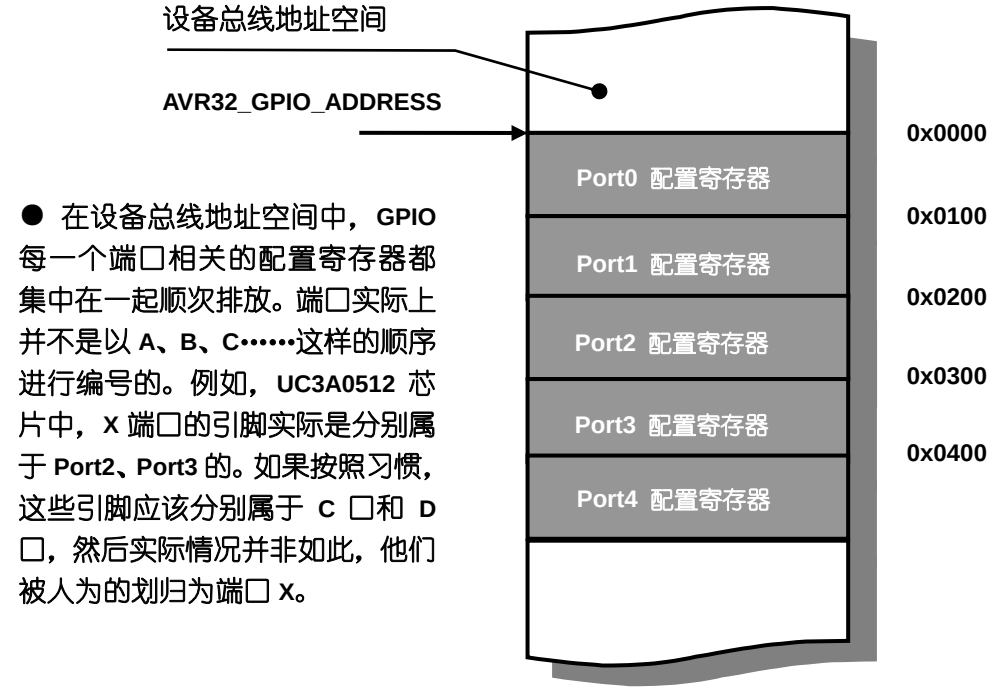
开启引脚电平变化中断，必须要首先开启 GPIO 的时钟，因为该模块总是在时钟的上升沿对引脚电平进行采样。当 GPIO 在时钟的上升沿探测到了一个符合要求的电平变化，将在延迟一个时钟周期以后，于第三个时钟周期的上升沿设置中断标志寄存器 IFR（Interrupt Flag Register）。同样，如果一个引脚电平的变化没有发生在时钟的上升沿，并且该电平的存活周期又不足以到达下一个时钟的上升沿，将无法被 GPIO 模块所感知到，因此，既不会表现在寄存器 PVR 上，也不会有任何机会触发中断。

借助寄存器 GFER（Glitch Filter Enable Register）开启瞬时脉冲滤波器，实际上只是要求 GPIO 在时钟的上升沿读取到有效的电平（也就是能够触发中断的电平）以后，必须在接下来的第二个上升沿对这一电平进行确认，以消除可能的瞬时脉冲产生的中断误触发。需要强调，开启瞬时脉冲缓冲器，将会在电平得到确认以后，引入额外两个周期的延时，加上原先设置中断标志位 IFR 所需的一个周期，实际上，系统将在上升沿确认有效电平之后的第四个上升沿设置中断标志寄存器 IFR。开启瞬时脉冲滤波器仅对中断有效，并不影响寄存器 PVR。

8.3.2 GPIO 寄存器的访问

在 AVR32 体系构架中，所有的设备寄存器都可以通过 HSB 经由设备总线 PB（Peripheral Bus）来进行访问。在 GPIO 中，每一个端口都有对应的一组寄存器，这些寄存器顺次连续的映射在设备总线的地址空间中（如图 8.3.2 所示）。

在 AVR32 UC3 系列的 GPIO 中，所有的引脚和编号实际上并不是以 A、B、C……这样的字母顺序来编号的。所有的引脚都有一个唯一的数字编号，这些编号每 32 个为一组，隶属于一个端口。这些端口从 0 开始向后依次进行编号。因此，我们就可以得到一个引脚编号和所属端口，以及该引脚与对应端口的配置寄存器所属二进制位的一个对应关系：



如图 8.3.2 GPIO 寄存器映射结构图

端口号 = [引脚编号 / 32]	注意，这里的“[]”表示只保留整数部分
端口内引脚编号 = 引脚编号 & 0x1F	注意，这里的“& 0x1F”相当于对 32 进行求余数操作

例如，某一个引脚编号为 **44**，根据公式，很容易知道它隶属于 **1** 号端口（**[44 / 32] = 1**），其端口内的引脚编号为 **12**（**44 & 0x1F = 12**）。习惯上，将字母 **A**、**B**、**C**……与端口号 **0**、**1**、**2**……一一对应，因此，对于 **44** 号引脚，我们就用 **PB12** 来表示。然而，这种对应关系并不是统一的，例如，在 **UC3A0512** 芯片中，引脚 **PX34** 的实际编号为 **70**，隶属于 **Port2**，而不是端口 **23**——不仅在命名上没有划归为端口 **C**，而且其端口内编号 **34** 也并不表示该引脚在对应寄存器中的位置。因此，我们并不能认为引脚的名称与实际编号之间有简单的线性对应关系。大部分情况下，我们可以利用引脚名称推测出引脚可能的编号。所幸官方驱动包定义了一些列常数宏，提供了引脚名称与实际编号的对应关系，我们只要通过这些宏就可以访问到正确的引脚而无需关心引脚实际的所属的端口和内部编号。

官方驱动包为我们定义了 **GPIO** 在设备总线 **PB** 地址空间中的位置，同时也定义了每一个具体寄存器相对端口起始地址的偏移量，利用这些信息，我们可以实现对某一特定引脚的寄存器进行直接访问。（请参照对编译器 **include** 目录下应器件对应的头文件以及 **gpio_100.h** 或 **gpio_110.h**）

例如，在前面的例子中，引脚 **44** 隶属于 **1** 号端口，其端口内引脚编号为 **12**，我们可以通过下面的方法实现对其 **GPER** 寄存器的访问：

```
/* 一个必要的宏 */
# define GPIO_REG(__PORT_NUM,__REG_OFFSET)      \
            (\
                * (volatile U32 *)\
                (AVR32_GPIO_ADDRESS + ((__PORT_NUM) * 0x0100) + (__REG_OFFSET))\
```

```
                )

/* 访问 Port1 的 GPER，其中宏 AVR32_GPIO_GPER 定义了 GPER 的偏移量 0x00000000 */
GPIO_REG(1, AVR32_GPIO_GPER) |= (1<<12);          /* 开启引脚 44 的普通输入输出功能 */
```

直接访问 **AVR32** 的寄存器是一种并不被提倡的做法，除非您非常了解寄存器细节，并且极端追求代码的效率。**将寄存器细节交给编译器，将具体实现方法交给官方驱动，是我们从事 32 位嵌入式系统开发的一个原则**。详细的资料请参考 **Datasheet** 和官方驱动函数库。

AVR32.UC3.GPIO.API

硬件驱动

8.4.1 数据结构

在设备总线 **PB** 地址空间中，寄存器以端口为单位组成寄存器文件，从 **GPIO** 模块的起始地址开始依次展开（**如图 8.3.2** 所示）。其中每个端口寄存器文件的内容如**表 8.4.1** 所示。

表 8.4.1 GPIO 端口寄存器映射表

偏移	寄存器	功能	名称	操作权限	复位默认值
0x00	GPIO Enable Register	读/写	GPER	读/写	有效引脚均为 1
0x04		Set	GPERS	只写	
0x08		Clear	GPERC	只写	
0x0C		Toggle	GPERT	只写	
0x10	Peripheral Mux Register 0	读/写	PMR0	读/写	0x00000000
0x14		Set	PMR0S	只写	
0x18		Clear	PMR0C	只写	
0x1C		Toggle	PMR0T	只写	
0x20	Peripheral Mux Register 1	读/写	PMR1	读/写	0x00000000
0x24		Set	PMR1S	只写	
0x28		Clear	PMR1C	只写	
0x2C		Toggle	PMR1T	只写	
0x30~ 0x3C	RESERVED	- -	- -	- -	
0x40	Output Driver Enable Register	读/写	ODER	读/写	0x00000000
0x44		Set	ODERS	只写	
0x48		Clear	ODERC	只写	
0x4C		Toggle	ODERT	只写	
0x50	Output Value Register	读/写	OVR	读/写	0x00000000
0x54		Set	OVRS	只写	
0x58		Clear	OVRC	只写	
0x5C		Toggle	OVRT	只写	
0x60	Pin Value Register	只读	PVR	只读	取决于引脚电平
0x64		-	-	-	
0x68		-	-	-	
0x6C		-	-	-	
0x70	Pull-up Enable Register	读/写	PUER	读/写	0x00000000
0x74		Set	PUERS	只写	
0x78		Clear	PUERC	只写	
0x7C		Toggle	PUERT	只写	
0x80~ 0x8C	RESERVED	- -	- -	- -	
0x90	Interrupt Enable Register	读/写	IER	读/写	0x00000000
0x94		Set	IERS	只写	

0x98		Clear	IERC	只写	
0x9C		Toggle	IERT	只写	
0xA0	Interrupt Mode Register 0	读/写	IMR0	读/写	0x00000000
0xA4		Set	IMR0S	只写	
0xA8		Clear	IMR0C	只写	
0xAC		Toggle	IMR0T	只写	
0xB0	Interrupt Mode Register 1	读/写	IMR1	读/写	0x00000000
0xB4		Set	IMR1S	只写	
0xB8		Clear	IMR1C	只写	
0xBC		Toggle	IMR1T	只写	
0xC0	Glitch Filter Enable Register	读/写	GFER	读/写	0x00000000
0xC4		Set	GFERS	只写	
0xC8		Clear	GFERC	只写	
0xCC		Toggle	GFERT	只写	
0xD0	Interrupt Flag Register	读取	IFR	只读	0x00000000
0xD4		-	-	-	
0xD8		Clear	IFRC	只写	
0xDC		-	-	-	
0xE0-0xFF	RESERVED	-	-	-	

针对端口寄存器文件，官方驱动包在 **gpio_100.h** 以及 **gpio_110.h** 中定义了相关的结构体，以方便对端口相关的寄存器进行访问。其中 **gpio_110.h** 提供了使用 **Local Bus** 访问端口寄存器的结构体定义，而 **gpio_100.h** 则没有。对应寄存器文件，一个典型的通过设备总线访问寄存器文件的结构体定义如下：

```
/*完整内容请参阅 include/gpio_100.h 或者 include/gpio_110.h */
typedef struct avr32_gpio_port_t {
        unsigned long           gper        ;//0x0000
        unsigned long           gpers       ;//0x0004
        unsigned long           gperc       ;//0x0008
        unsigned long           gpert       ;//0x000c
        ……
    const unsigned long         ifr         ;//0x00d0
        unsigned int           :32          ;//0x00d4
        unsigned long           ifrc        ;//0x00d8
        ……
} avr32_gpio_port_t;
```

官方将该结构体定义为一个新的类型 **avr32_gpio_port_t**，并在该类型的基础上通过数组的形式定义了 **GPIO** 模块完整的寄存器影射方式：

```
/* GPIO 在设备总线地址空间中的完整影射 */
typedef struct avr32_gpio_t {
  avr32_gpio_port_t      port[AVR32_GPIO_PORT_LENGTH];
} avr32_gpio_t;
```

在结构体 **avr32_gpio_t** 中，宏 **AVR32_GPIO_PORT_LENGTH** 用于描述整个 **GPIO** 模块在设备总线地址空间中共有多少端口寄存器文件，该宏由处理器对应型号的头文件定义（例如：

include\uc3a0512.h）。借助这些结构体，我们只要拥有了 GPIO 在设备总线地址空间中的起始地址，就能够以一种优雅的方式很快访问到我们所需的寄存器（当然，包括笔者在内，一开始往往并不习惯官方驱动包将寄存器名称全部小写的表达方式）：

```
/* 这些宏可以在具体的芯片头文件中找到，例如 include\uc3a0512.h */
#define AVR32_GPIO_ADDRESS      0xFFFF1000
#define AVR32_GPIO              (*((volatile avr32_gpio_t*)AVR32_GPIO_ADDRESS))

……

/* 访问 Port2 的 OVR 寄存器 */
AVR32_GPIO. port[2].ovr = 0x12345678;

/* 已知引脚编号 44，开启该引脚的上拉电阻 */
AVR32_GPIO.port[44>>5].puers = (1<<(44 & 0x1F));
/* 或者利用库函数提供的宏得到以下的等效写法 */
AVR32_GPIO.port[AVR32_PIN_PB12 >> 5].puers = (1 << (AVR32_PIN_PB12 & 0x1F));
```

通过 Local Bus 访问 GPIO 的方法与使用设备总线的方法类似，只不过需要借助专用的结构体类型 avr32_gpio_local_port_t 来描述端口寄存器文件；借助 avr32_gpio_local_t 来描述 GPIO 模块的映射方式，例如：

```
#define AVR32_GPIO_LOCAL_ADDRESS                0x40000000
#define AVR32_GPIO_LOCAL        \
            (*((volatile avr32_gpio_local_t*)AVR32_GPIO_LOCAL_ADDRESS))
……

/* 首先要完成对 Local Bus 的初始化 */
gpio_local_init();
……
/* 通过 Local Bus 来访问 Port2 的 OVR 寄存器 */
AVR32_GPIO_LOCAL.port[2].ovr = 0x12345678;
```

针对 AVR32 UC3 系列微处理器支持在引脚上复用多路设备功能的特性，官方驱动包采用了一种类似面向对象的处理方法：将引脚编号与引脚对应的设备功能封装在一起构成结构体，并定义了具有数组属性的用户类型 gpio_map_t[]，我们不妨称之为引脚功能地图。您可以在官方驱动包的头文件 DRIVERS\GPIO\gpio.h 中找到对应的代码：

```
typedef struct
{
   unsigned char pin;              //!< Module pin.
   unsigned char function;         //!< Module function.
} gpio_map_t[];
```

利用这一数据结构，我们可以使用类似面向对象的思想（注意只是类似），在程序的一开始使用声明数组的方式集中指定功能模块所涉及到的引脚并指定引脚的设备功能，最后再将定义的引脚功能地图传递给相应的接口函数 **gpio_enable_module()**完成引脚功能的统一设定，例如：

```
/*您可以在 DRIVERS\USART\USART_EXAMPLE\ usart_example.c 中找到以下代码片断*/

/* 定义引脚宏 */
#   define EXAMPLE_USART_RX_PIN          AVR32_USART1_RXD_0_0_PIN
#   define EXAMPLE_USART_RX_FUNCTION     AVR32_USART1_RXD_0_0_FUNCTION
#   define EXAMPLE_USART_TX_PIN          AVR32_USART1_TXD_0_0_PIN
#   define EXAMPLE_USART_TX_FUNCTION     AVR32_USART1_TXD_0_0_FUNCTION

/* 定义了引脚功能地图 */
static const gpio_map_t USART_GPIO_MAP =
{
    {EXAMPLE_USART_RX_PIN, EXAMPLE_USART_RX_FUNCTION},
    {EXAMPLE_USART_TX_PIN, EXAMPLE_USART_TX_FUNCTION}
};

……
/* 传递定义的引脚功能地图给接口函数完成引脚设备功能的设置. */
gpio_enable_module
    (
        USART_GPIO_MAP,
        sizeof(USART_GPIO_MAP) / sizeof(USART_GPIO_MAP[0])
    );
```

8.4.2 硬件驱动 API 函数

官方提供的 **GPIO** 驱动函数库将针对 **GPIO** 模块的访问分为两类：一种是通过设备总线的方式；另一种是通过本地总线 **Local Bus** 来进行访问。其中使用 **Local Bus** 进行访问的函数接口对于不含该功能的芯片，系统会通过条件编译的方法自动的进行屏蔽，而无需用户手动进行设置。

8.4.2.1 通过设备总线 PB 来访问 GPIO 模块

GPIO 在所有的寄存器在默认的情况下都可以通过设备总线 **PB** 进行直接访问，唯一的缺点就是操作所消耗的时间是不确定的。虽然在 **AVR32 UC3** 的器件手册上明确的指出，从控制 **OVR** 引脚输出电平到电平信息真实的出现在引脚上需要确定的两个时钟周期（**GPIO** 的时钟，而非 **CPU** 时钟）；这一电平信息出现在 **PVR** 上又需要额外的两个时钟周期；但这都是建立在寄存器已经得到操作的基础上。由于设备总线 **PB** 为很多外设所共享，**CPU** 针对 **GPIO** 的操作往往并不能以一个确定的时间到达模块，加之设备总线的工作频率总是小于等于 **CPU** 工作时钟，因而，通过设备总线 **PB** 来访问 **GPIO**（相对采用 **Local Bus** 而言）是比较缓慢的，信息交换的延迟也是不确定的。

官方驱动函数库提供了两种操作引脚的方式：

1、 以引脚编号为输入信息提供针对单个引脚的操作；

2、 以引脚功能地图（多个引脚及其功能的说明信息所组成的结构体）为输入信息，成批的处理引脚的操作。

第一种方式主要用于引脚的普通输入输出、引脚设备功能的单独设定、中断操作；第二种方式则仅针对引脚设备功能的批量设定。适应不同的操作方式，官方驱动包分别定义了两套常数宏，用于建立标称引脚（就是我们在原理图上看到的引脚名称）与内部引脚编号之间的关联。

当我们的工程中通过**#include <avr32/io.h>**包含了对器件基本输入输出的支持库时，预编译系统很根据头文件 **io.h** 以及工程的设定自动选择对应的器件支持库，例如 **uc3a0512.h**，这些基本的头文件都放置在编译器的 **include** 文件夹下。

在器件对应的支持库中，定义了引脚名称到内部编号的宏，例如：

```
#define AVR32_PIN_PA00     0
#define AVR32_PIN_PA01     1
#define AVR32_PIN_PA02     2
……
#define AVR32_PIN_PX37     103
#define AVR32_PIN_PX38     102
#define AVR32_PIN_PX39     101
```

同时，为了方便用户利用库函数设定端口的第二功能，该头文件还在每个设备相关的区域内定义了所需的功能引脚编号，以及对应的引脚功能编号，例如：

```
/* USART 模块定义相关引脚和功能编号的宏 */
#define AVR32_USART1_CLK_0_PIN                7
#define AVR32_USART1_CLK_0_FUNCTION           0
……
#define AVR32_USART1_RXD_0_0_PIN              5
#define AVR32_USART1_RXD_0_0_FUNCTION         0
#define AVR32_USART1_RXD_0_1_PIN              96
#define AVR32_USART1_RXD_0_1_FUNCTION         1
#define AVR32_USART1_TXD_0_0_PIN              6
#define AVR32_USART1_TXD_0_0_FUNCTION         0
#define AVR32_USART1_TXD_0_1_PIN              95
#define AVR32_USART1_TXD_0_1_FUNCTION         1
```

利用这些宏定义，用户就可以基于语义完成端口功能等的设置，而无需关心不同芯片在内部端口编号和引脚功能编号上的差异，极大的减小了同系列芯片间源代码移植的工作量，提高了代码的重用性和可维护性。关于如何利用这些宏完成设备功能引脚批量设定的例子，请参考 **8.4.1 数据结构**章节的相关说明。

● 以单个引脚作为操作对象的接口函数

```
/* 设定指定引脚的设备功能，并自动关闭该引脚普通输入输出功能*/
/* 如果操作成功返回 0，操作失败返回 1 */
extern int gpio_enable_module_pin(unsigned int pin, unsigned int function);

/* 开启指定引脚的普通输入输出功能，并自动关闭电平输出功能，进入开漏状态*/
extern void gpio_enable_gpio_pin(unsigned int pin);

/* 开启/关闭指定引脚的内部上拉电阻 */
extern void gpio_enable_pin_pull_up(unsigned int pin);
extern void gpio_disable_pin_pull_up(unsigned int pin);

/* 读取指定引脚的电平信息 */
extern int gpio_get_pin_value(unsigned int pin);

/* 读取指定引脚上次曾要求输出的电平信息 */
extern int gpio_get_gpio_pin_output_value(unsigned int pin);

/* 在指定的引脚上输出高电平，自动切换到普通输入输出模式，并打开电平输出功能 */
extern void gpio_set_gpio_pin(unsigned int pin);

/* 在指定的引脚上输出低电平，自动切换到普通输入输出模式，并打开电平输出功能 */
extern void gpio_clr_gpio_pin(unsigned int pin);

/* 将指定引脚上的电平取反，自动切换到普通输入输出模式，并打开电平输出功能 */
extern void gpio_tgl_gpio_pin(unsigned int pin);

/* 打开/关闭指定引脚的瞬时脉冲滤波功能 */
extern void gpio_enable_pin_glitch_filter(unsigned int pin);
extern void gpio_disable_pin_glitch_filter(unsigned int pin);

/* 设置制定引脚的中断模式，并自动打开中断使能和瞬间脉冲滤波功能 */
/* 如果操作成功返回 0，操作失败返回 1 */
extern int gpio_enable_pin_interrupt(unsigned int pin, unsigned int mode);

/* 关闭制定引脚的中断使能 */
extern void gpio_disable_pin_interrupt(unsigned int pin);

/* 读取指定引脚的中断标志位 */
extern int gpio_get_pin_interrupt_flag(unsigned int pin);

/* 清除指定引脚的中断标志 */
extern void gpio_clear_pin_interrupt_flag(unsigned int pin);
```

- 以多个引脚的功能地图为操作对象的接口函数

```
/* 通过引脚功能地图，批量设置指定数量的引脚功能，并自动关闭引脚的普通输入输出功能 */
extern int gpio_enable_module(const gpio_map_t gpiomap, unsigned int size);

/* 通过引脚功能地图，批量设置指定数量的引脚为普通输入输出引脚，默认为开漏模式*/
extern void gpio_enable_gpio(const gpio_map_t gpiomap, unsigned int size);
```

8.4.2.2 通过本地总线 Local Bus 来访问 GPIO 模块

AVR32 UC3 系列使用 **Local Bus** 访问 **GPIO** 时，仅对部分需要经常操作的关键寄存器有效，分别为：**ODER**、**OVR** 和 **PVR**（对寄存器的 **Set**、**Clear**、**Toggle** 版本同样有效）。因此，驱动包提供了对端口进行输入、输出以及电平输出使能等功能进行控制的接口函数。它们的使用方法与采用设备总线 **PB** 进行访问的同类函数相同，可以看作仅仅是在相关操作函数的名称中插入了一个单词“**local**”。对应的接口声明如下：

```
/* local bus 初始化函数，在使用 local bus 访问 GPIO 之前，必须先调用这个函数 */
#if __GNUC__
__attribute__((__always_inline__))
#endif
extern __inline__ void gpio_local_init(void)
```

```
/* 开启指定编号引脚的电平输出功能，在此之前，在此之前需要确定 GPER 是否设置正确*/
#if __GNUC__
__attribute__((__always_inline__))
#endif
extern __inline__ void gpio_local_enable_pin_output_driver(unsigned int pin)
```

```
/* 关闭指定编号引脚的电平输出功能，这将导致引脚处于开漏状态。*/
#if __GNUC__
__attribute__((__always_inline__))
#endif
extern __inline__ void gpio_local_disable_pin_output_driver(unsigned int pin)
```

```
/* 读取指定引脚的电平值，在此之前需要确定 GPER 是否设置正确*/
#if __GNUC__
__attribute__((__always_inline__))
#endif
extern __inline__ int gpio_local_get_pin_value(unsigned int pin)
```

```
/* 设置指定引脚的输出高电平，在此之前，需要确认开启了引脚的电平输出功能 */
#if __GNUC__
__attribute__((__always_inline__))
#endif
extern __inline__ void gpio_local_set_gpio_pin(unsigned int pin)
```

```
/* 设置指定引脚的输出低电平，在此之前，需要确认开启了引脚的电平输出功能 */
#if __GNUC__
__attribute__((__always_inline__))
#endif
extern __inline__ void gpio_local_clr_gpio_pin(unsigned int pin)
```

```
/*对指定引脚上的电平取反后输出，在此之前，需要确认开启了引脚的电平输出功能 */
#if __GNUC__
__attribute__((__always_inline__))
#endif
extern __inline__ void gpio_local_tgl_gpio_pin(unsigned int pin)
```

以上的接口声明可以从官方驱动包 **DRIVERS\GPIO\GPIO.h** 中找到。为了保证代码效率，最大的发挥使用 **Local Bus** 访问的优势，函数库强行指定这些接口为内联函数。使用 **Local Bus** 访问 **GPIO** 之前，首先需要通过函数 **gpio_local_init()**对总线进行初始化；同时需要确认此时寄存器 **GPER** 的设置是否正确，有必要的情况下，需要通过使用设备总线 **PB** 的接口函数对寄存器 **GPER** 进行访问。

用户试图对整个端口而非单个引脚进行操作时，可以借助宏 **AVR32_GPIO_LOCAL** 宏对相关寄存器进行直接访问，例如：

```
/* 设置端口 Port1 的 BIT8~BIT15 输出 0xAA */
AVR32_GPIO_LOCAL.port[1].ovrc = 0x55 << 8;
AVR32_GPIO_LOCAL.port[0].ovrs = 0xAA;
```

8.4.3 端口开漏技术的实现

引脚的开漏输出是实现“线与”特性、在一定范围内由外界决定输出逻辑高电平电压的关键。由于缺乏必要的硬件开漏控制寄存器，我们只能通过一种变通的方法来实现这一功能：

首先设置寄存器 **GPER** 开启目标引脚的普通输入输出功能；接下来，在未开启引脚电平输出功能的情况下，设置寄存器 **OVR**，使引脚对应位为“**0**”——一旦通过寄存器 **ODER** 开启了引脚的驱动使能，将直接输出低电平；反之将处于开漏输出状态。实际上，经过上面的步骤，我们已经可以通过控制 **ODER** 寄存器来控制开漏输出高低电平，完成了原先需要专用的开漏寄存器 **ODMER**（**Open Drain Mode Enable Register**）才能实现的功能。

```
# define PB      AVR32_GPIO.port[1]
# define BIT(__VALUE)        (1<<(__VALUE))

/* 在 GPIO 普通输入输出模式下，再未开启 ODER 的情况下设置 OVR，完成初始化 */
PB.ovrc = BIT(12);            /* PB12 输出低电平 */

/* 控制 ODER 的使能来选择输出低电平或者开漏 */
PB.oderc = BIT(12);           /* PB12 开漏输出高电平，电平电压由外部电路决定 */
PB.oders = BIT(12);           /* PB12 输出低电平 */
```

```
/* 在这个基础上，我们可以定义对应的功能宏 */
# define PB12_CLEAR        PB.oders = BIT(12);        /* 输出低电平 */
# define PB12_RELEASE      PB.oderc = BIT(12);        /* 开漏输出 */
```

AVR32.UC3.GPIO.Examples 范例

8.5.1 使用设备总线访问 GPIO

［需求与分析］

本范例将通过对 **LED** 和按键的基本操作，分别演示官方驱动包对 **GPIO** 的 **3** 种访问方式。它们分别为：直接使用 **gpio.h** 中定义的接口函数、使用 **Software Framework** 定义的 **GPIO** 相关结构体、利用驱动函数库提供的地址常数直接访问寄存器。在该实例中，我们将利用本书配套的开发套件 **OpenUC3** 作为演示蓝本。与 **OpenUC3** 相关的宏定义，都可以在 **SOFTWARE_FRAMEWORK\BOARDS\OpenUC3\OpenUC3.h** 中找到，如：

```
/* SOFTWARE_FRAMEWORK\BOARDS\OpenUC3\OpenUC3.h */
……
/* 定义LED引脚 */
# define LED1_PIN          AVR32_PIN_PB27
# define LED2_PIN          AVR32_PIN_PB28
# define LED3_PIN          AVR32_PIN_PB29
# define LED4_PIN          AVR32_PIN_PB30

/* 定义按钮 */
# define SW1_PIN           AVR32_PIN_PA20
# define SW2_PIN           AVR32_PIN_PA26
# define SW3_PIN           AVR32_PIN_PA27
# define SW4_PIN           AVR32_PIN_PA28
……
```

［相关设计资料］

参考硬件电路原理如图 **8.5.1** 所示。核心代码如下：

```
/ Include Files
#include "board.h"
#include "gpio.h"
```

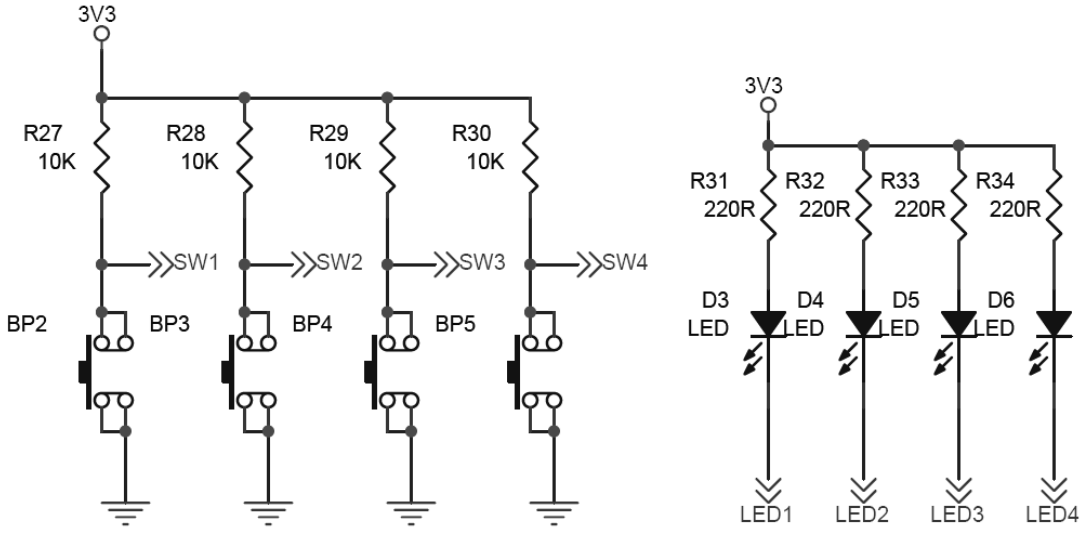


图 8.5.1 OpenUC3 按键和 LED 部分原理图

```
void Delay(void)
{
    U16 n = 5000;
    while(n--);
}

void Short_Delay(void)
{
    U16 n = 1000;
    while(n--);
}

int main(void)
{
    /* 使用库函数直接点亮 LED1 */
    gpio_clr_gpio_pin(LED1_PIN);
    Delay();

    /* 使用库函数直接点亮 LED2，并使用系统定义的引脚宏*/
    /* #define LED2_PIN        AVR32_PIN_PB28 */
    gpio_clr_gpio_pin(AVR32_PIN_PB28);
    Delay();

    /* 通过库函数定义的结构体直接访问寄存器，点亮 LED3*/
    AVR32_GPIO.port[LED3_PIN >> 5].ovrc = BIT(LED3_PIN & 0x1F);
    AVR32_GPIO.port[LED3_PIN >> 5].oders = BIT(LED3_PIN & 0x1F);
    AVR32_GPIO.port[LED3_PIN >> 5].gpers = BIT(LED3_PIN & 0x1F);
    Delay();
```

```
/* 通过地址的方式直接访问寄存器，点亮 LED4 */
# define GPIO_REG(__PORT_NUM,__REG_OFFSET)      \
        (\
            * (volatile U32 *)\
            (AVR32_GPIO_ADDRESS + ((__PORT_NUM) * 0x0100) + (__REG_OFFSET))\
        )

GPIO_REG(LED4_PIN >> 5,AVR32_GPIO_OVRC) = BIT(LED4_PIN & 0x1F);
GPIO_REG(LED4_PIN >> 5,AVR32_GPIO_ODERS) = BIT(LED4_PIN & 0x1F);
GPIO_REG(LED4_PIN >> 5,AVR32_GPIO_GPERS) = BIT(LED4_PIN & 0x1F);
Delay();

/* 开启按钮的上拉电阻 */
gpio_enable_pin_pull_up(SW1_PIN);                                  /* SW1_PIN */
gpio_enable_pin_pull_up(AVR32_PIN_PA26);                           /* SW2_PIN */
AVR32_GPIO.port[SW3_PIN >> 5].puers = BIT(SW3_PIN & 0x1F);         /* SW3_PIN */
GPIO_REG(SW4_PIN >> 5,AVR32_GPIO_PUERS) = BIT(SW4_PIN & 0x1F);  /* SW4_PIN */

/* 开启按钮的普通输入输出功能，默认关闭引脚电平输出功能 */
gpio_enable_gpio_pin(SW1_PIN);                                     /* SW1_PIN */
gpio_enable_gpio_pin(AVR32_PIN_PA26);                              /* SW2_PIN */
AVR32_GPIO.port[SW3_PIN >> 5].oderc = BIT(SW3_PIN & 0x1F);         /* SW3_PIN */
AVR32_GPIO.port[SW3_PIN >> 5].gpers = BIT(SW3_PIN & 0x1F);
GPIO_REG(SW4_PIN >> 5,AVR32_GPIO_ODERC) = BIT(SW4_PIN & 0x1F);  /* SW4_PIN */
GPIO_REG(SW4_PIN >> 5,AVR32_GPIO_GPERS) = BIT(SW4_PIN & 0x1F);

while(TRUE)
{
    /* SW1_PIN */
    if (gpio_get_pin_value(SW1_PIN) == LOW)
    {
        /* 按钮被按下 */
        gpio_set_gpio_pin(LED1_PIN);
    }

    /* SW2_PIN */
    if (gpio_get_pin_value(AVR32_PIN_PA26) == LOW)
    {
        /* 按钮被按下 */
        gpio_clr_gpio_pin(LED1_PIN);
    }

    /* SW3_PIN */
    if (!(AVR32_GPIO.port[SW3_PIN >> 5].pvr & BIT(SW3_PIN & 0x1F)))
```

```
        {
            /* 按键去抖 */
            Short_Delay();
            if (!(AVR32_GPIO.port[SW3_PIN >> 5].pvr & BIT(SW3_PIN & 0x1F)))
            {
                /* 按钮被按下 */
                AVR32_GPIO.port[LED2_PIN >> 5].ovrt = BIT(LED2_PIN & 0x1F);
            }
        }

        if (!(GPIO_REG(SW4_PIN >> 5,AVR32_GPIO_PVR) & BIT(SW4_PIN & 0x1F)))
        {
            /* 按键去抖 */
            Short_Delay();
            if (!(GPIO_REG(SW4_PIN >> 5,AVR32_GPIO_PVR) & BIT(SW4_PIN & 0x1F)))
            {
                AVR32_GPIO.port[LED3_PIN >> 5].ovrt = BIT(LED3_PIN & 0x1F)
                                                    | BIT(LED4_PIN & 0x1F);
            }
        }
    }
    return 0;
}
```

［演示效果］

正确下载程序到 **OpenUC3**，我们会看到评估版上的 **4** 个 **LED** 以差不多 **1** 秒为间隔，依次被点亮。这时，当我们按下按钮 **1**（**SW1**）时，**LED1** 熄灭；当我们按下按钮 **2**（**SW2**）时，**LED1** 又将再次被点亮；每次我们按下按钮 **3**（**SW3**），**LED2** 的明暗状况都会被取反；当按钮 **4**（**SW4**）被按住不放时，**LED3** 和 **LED4** 都会被同时点亮或者熄灭。

［小结与分析］

在这个范例中，我们分别使用 **4** 种不同的方法对相同的功能进行演示。比如，依次点亮 **LED1**~**LED4** 所使用的方法就是各不相同的。其中，点亮 **LED1** 所用的方式，最为简单，也是本书所提倡的方法：直接调用芯片厂商提供的硬件接口驱动函数来实现。这种操作方式在实现相同功能的代码效率上可能有所牺牲，但这一牺牲也并非是绝对的。而最关键的优点在于，该方式体现了当前以及未来 **32** 位嵌入式系统开发的一个潜原则：**将寄存器的访问细节交给编译器；将功能的具体实现交给官方 API 驱动。**

点亮 **LED1** 和 **LED2** 的方法虽然都是通过调用官方 **API** 函数，但在开发思想上还是有所差别的。其中，点亮 **LED1** 时，我们借助硬件细节描述宏 **LED1_PIN** 将所要操作的引脚以及该引脚的功能在代码中直观的表现出来，不仅提高了代码的可读性，更重要的是，通过修改 **LED1_PIN** 的内容，就可以实现原代码在不同硬件间的移植。以上优势在 **LED2** 的操作中并不存在，同样是使用宏来指定函数所要操作的目标引脚，然而 **AVR32_PIN_PB28** 这样的宏与处理器相关性太大；阅读这段代码时，必须参照处理器的引脚封装图和硬件原理图，从代码中并不能直观的获得更多地信息；当试图将源代码移植到其他硬件上时，我们不得不面对一一

修正相关代码的庞杂问题，难免不出错误。

点亮 **LED3** 的代码，为我们介绍了一种利用官方驱动函数包提供的结构体定义直接访问目标寄存器的方式。除非很有必要，否则这种要求用户知晓寄存器细节，并且亲自配置的操作方式是不被推荐的。在代码效率方面，由于同样是通过结构体进行间接寻址，因此在编译器优化选项开启的情况下，并不见得比第一种和第二种方法有优势。与之相对应，点亮 **LED4** 的方式则是利用官方驱动包中最底层的地址信息进行直接寄存器寻址访问，很多情况下，这种对寄存器采用立即数直接寻址的方式会带来最高的效率，同时也为程序员带来最多的细节问题。如果您不是一个底层驱动开发人员，如果您的项目不在速度和效率上有较为严格的要求，本书不推荐这种方式。

作为四种常见的操作方法，我们有义务的本书的第一个实例中对其逐个进行列举。但是在普通的应用中，我们更推荐通过 **API** 来操作硬件的方式：它使得程序员将更多地精力从缺乏技术含量的寄存器操作中解放出来，投入到真正需要设计者发挥聪明才智的算法和应用设计中去。因此，除非有特殊需要，在本书以后的讲解和范例中，我们将统一采用第一种方式——也就是调用官方驱动的方式来操作硬件。

8.5.2 使用 Local Bus 产生 33MHz 方波范例

［需求与分析］

本范例通过 **Local Bus** 访问 **GPIO**，并产生一路 **33MHz** 方波，为读者展示 **AVR32 UC3** 系列的高速 **IO** 功能。

软件环境设置：使用 **OSC0 12MHz** 晶振 **PLL** 锁相环至 **66MHz**。

硬件连接说明：**33MHz** 方波输出←→**PB18**

［相关设计资料］

核心代码如下：

```
#include "board.h"
#include "gpio.h"
#include "pm.h"

int main(void)
{

    /* 将系统时钟设置为 66M */
    pm_switch_to_osc0(&AVR32_PM, FOSC0, OSC0_STARTUP);
    pm_pll_setup(&AVR32_PM, 0,   // pll.
                 10,   // mul.
                 0,    // div.
                 0,    // osc.
                 16); // lockcount.
    pm_pll_set_option(&AVR32_PM, 0, // pll.
                      1,   // pll_freq.
                      1,   // pll_div2.
                      0); // pll_wbwdisable.
    pm_pll_enable(&AVR32_PM, 0);
```

```
    pm_wait_for_pll0_locked(&AVR32_PM);
    pm_cksel(&AVR32_PM,
             1,    // pbadiv.
             0,    // pbasel.
             1,    // pbbdiv.
             0,    // pbbsel.
             1,    // hsbdiv.
             0);   // hsbsel.
    pm_switch_to_clock(&AVR32_PM, AVR32_PM_MCCTRL_MCSEL_PLL0);

    /* 初始化 local bus */
    gpio_local_init();

    /* 使能 GPIO 的电平输出功能 */
    gpio_local_enable_pin_output_driver(AVR32_PIN_PB18);

    while(TRUE)
    {

        #define INSERT_GPIO_LOCAL_TGL_GPIO_PIN(idx, pin) \
                    gpio_local_tgl_gpio_pin(pin);
        MREPEAT(255, INSERT_GPIO_LOCAL_TGL_GPIO_PIN, AVR32_PIN_PB18)
        #undef INSERT_GPIO_LOCAL_TGL_GPIO_PIN

    }

     return 0;
}
```

［演示效果］

借助数字示波器，我们可以在屏幕上看到一个明显的方波，其频率为 **32.9M**（如图 **8.5.2** 所示）：（有待补充图片）

［小结与分析］

在这个范例中，我们很容易忽略一个细节：当示波器显示引脚输出波形的频率为 **32.9M** 时，很多人会将这种偏差归咎于示波器的测量误差或者 **AVR32 UC3** 处理器所使用的时钟源存在不稳定因素。这些我们都不否认，不过如果细心观察就会发现一个疑点：为什么范例的设计者在每一个超级循环周期里都要使用 **MREPAT** 宏将端口的翻转操作重复 **256** 次呢？原因很简单：通过 **Local Bus**，**GPIO** 的引脚操作可以和内核的工作频率相同，当内核工作在 **66M** 频率下时，**GPIO** 的翻转操作将产生 **33M** 的方波——当然这是理想情况，因为这要求代码沿着线形的流程一直运行下去，如果中途有任何类似跳转指令的存在，就无法达到 **33M** 的输出频率——这是显而易见的。所以，为了使输出波形尽可能的接近 **33M**，范例的设计者采用“列波”的方式，以 **256** 的长度为单位，输出一个个 **33M** 的列波，虽然其间插入了 **1** 个指令周期的跳转语句，但波形的总体频率看起来就是及其接近 **33M** 的，这就是该范例输出小于

33M 频率波形的理论必然性。

AVR32.UC3.GPIO.Reference	相关参考信息

8.6 如何获取进一步的参考信息

作为一本 **AVR32 UC3** 工程技术指南，我们不可能涵盖官方文档以及驱动函数的所有信息。除了本章提炼的信息以外，更详细的内容可以通过以下的渠道获得：

- 关于**GPIO**寄存器的详细信息以及相互间的制约关系，可以参阅官方器件手册的相关章节：**22. General-Purpose Input/Output Controller (GPIO)**。本书参考的章节版本为**REV 1.1.0.2**。
- 关于**GPIO**的电器参数，可以参照官方器件手册的相关章节：**38.2 DC Characteristices**。
- 在官方器件手册**Peripheral**章节中，有关于**GPIO**开漏寄存器的相关描述。
- 在官方期间手册**Blockdiagram**章节中，有关于**GPIO**与**Local Bus**的模块结构说明，根据这些信息，也很容易找到设备总线桥**PBB**、系统高速总线**HSB**以及**GPIO**三者之间的关系。
- 如果您想知道更多关于引脚电平变化中断、以及如何发挥引脚速度优势的内容，可以补充阅读器件手册中关于系统电源管理**Power Manager**和中断控制器**Interrupt Controller**的内容。系统电源管理模块**PM**对**GPIO**来说尤为重要。
- **ATMEL**官方提供的驱动函数库中，**include**文件夹下的**gpio_100.h**、**gpio_110.h**以及**Driver\GPIO**文件夹下的**gpio.h**和**gpio.c**将会为我们带来更多的信息。