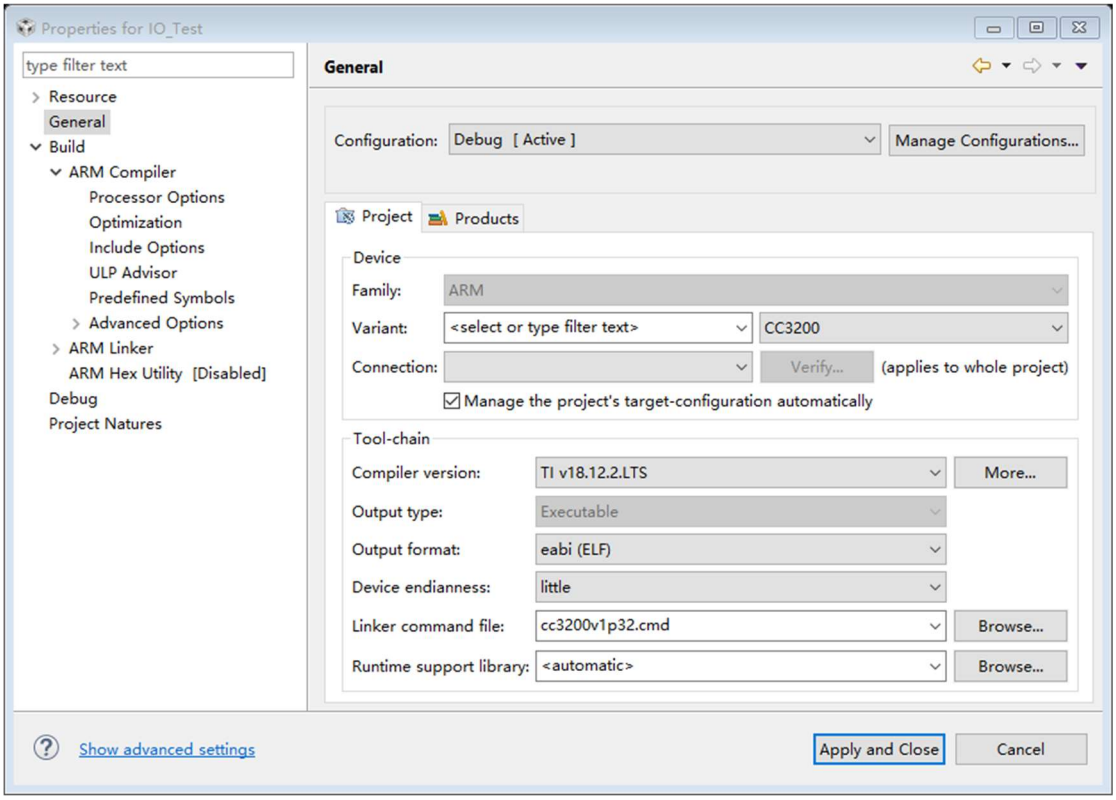
大家好，我是 cug-xiao，今天为大家带来的是 CC3200 外设之 GPIO 详解

首先呢，我们先来了解一下如何开发 CC3200 的外设。毫无疑问的，两种方式：寄存器和库函数。

咱们首先来说简单的库函数，ti 官方有提供 driverlib，咱们只需要在 include 里面添加好驱动库的路径，在使用时只需要 include 相应头文件就可以愉快的使用库函数了。

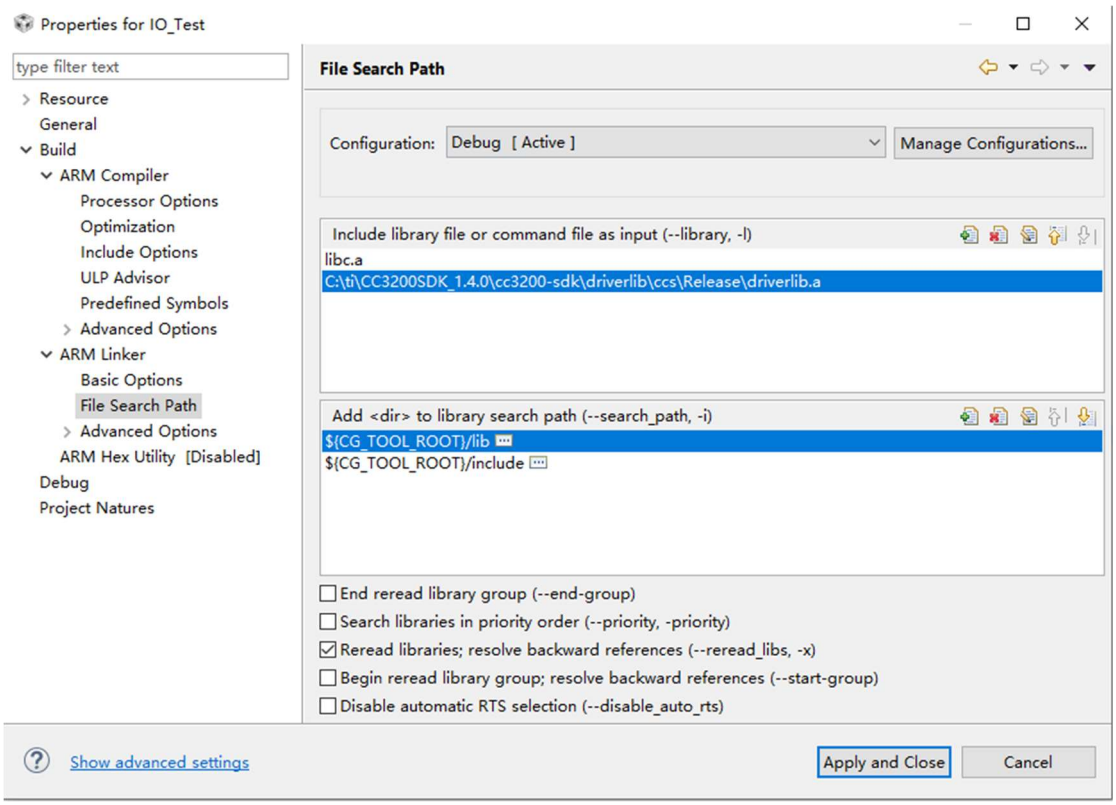
首先咱们来建立工程，直接 File -> New -> CCS Project，取好名字点 Finsh 即可。新建好工程模板之后我们还需要修改 cmd 文件（可直接从官方例程复制）和新建一个 ccxml 文件（可直接从官方例程复制），这俩文件是用来 Debug 的。

然后呢我们需要在 include 下添加驱动库路径，右键工程名选择最下方的 Properties 会弹出如下窗口。



咱们点击 ARM Compiler 下的 include Options，点击加号即可选择我们的驱动库路径，官方例程中的 common 文件路径可以不用添加，因为这个是专门为 CC3200 这个开发板写的接口，若是我们以后自己用 CC3200 绘制板子做应用，这个接口就用不到了，这种接口我们一般将其称之为 BSP（Board Support Package，即板级支持包），不是通用接口，像 driverlib 这种底层一点的驱动库才是我们需要的。

好了，将驱动库路径添加好之后不要着急关闭这个页面，因为我们还有一步需要完成，点击 ARM Linker，选择 File Search Path，添加 driverlib.a 这个文件，这是告诉编译器去哪找头文件相对应的源文件，如果没有这一步是无法编译通过的哟（提示无法找到函数定义）。添加好后如下图。



好啦，工程模板建好了，下面就要开始愉快的编码了。本来以为是挺简单的一个过程（如果很简单的话就没有这个帖子了~~），结果还是有许多坑的。

首先咱们看看官方怎么写的，嗯，相比于我们的工程多了 pinmux 文件和 gpio_if 文件。不用多说，pinmux 是引脚配置文件，将与 LED 相连的机构引脚配置为输出模式，这个文件可以使用我们之前提到的 Pin Mux Tool 这个工具来直接生成；gpio_if 这个文件呢是官方 BSP 里面的，可以看到我们的 LED 驱动就只在这一个文件夹里实现的，咱们自己实现的话，最好不要使用这个文件，当然可以用来参考。不过我个人觉得这个驱动实现的有点七拐八绕的，不容易看明白具体如何实现的，特别是在 CC3200 的 GPIO 分组奇奇怪怪的情况下。

接下来咱们来聊一聊 CC3200 的 GPIO，作为一个第一次接触 ti 的开发板的开发者来讲，这个板子（我不知道其他的开发板是不是也是这样）的 GPIO 实在是有点奇怪，在程序中即用到了芯片的引脚号（PIN_01~PIN_64），也用到了 GPIO 编号（GPIO00~GPIO31），还用到了 GPIO 组号（GPIO_A0~A3，GPIO_PIN_00~07），而且在使用库函数时这三个参数都需要使用到。而这其中的对应关系，说实话阻隔了我一段时间。

这其中的映射关系我们有一下几种方式了解到：

① 查表，在 pin.c 下有这么一个表，存放的就是 PIN 至 GPIO 的映射关系。

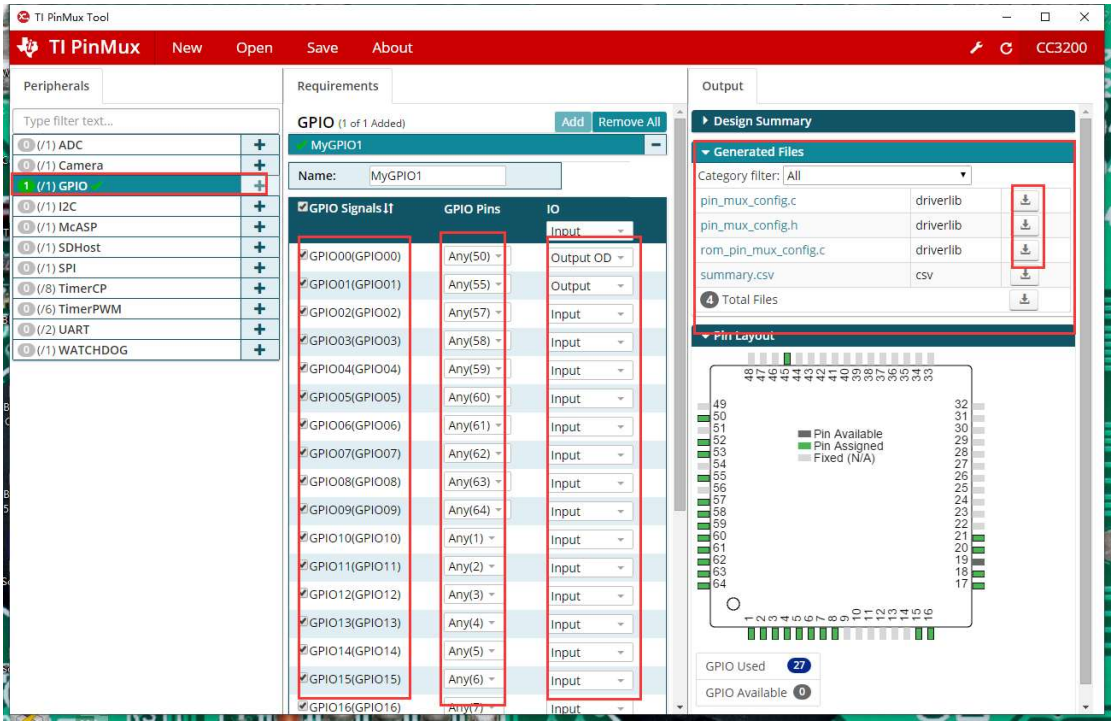
```
//*****************************************************************************
// PIN to PAD matrix
//*****************************************************************************
static const unsigned long g_ulPinToPadMap[64] =
{
    10,11,12,13,14,15,16,17,255,255,18,
    19,20,21,22,23,24,40,28,29,25,255,
    255,255,255,255,255,255,255,255,255,255,255,
    255,255,255,255,255,255,255,255,255,255,255,
    31,255,255,255,255,0,255,32,30,255,1,
    255,2,3,4,5,6,7,8,9
};
```

② 看官方数据手册（72 页的），从第 8 页开始，也可以找到对应关系

Table 3-1. Pin Multiplexing (continued)

General Pin Attributes						Function					Pad States		
Pkg Pin	Pin Alias	Use	Select as Wakeup Source	Config Addl Analog Mux	Muxed with JTAG	Dig. Pin Mux Config Reg	Dig. Pin Mux Config Mode Value	Signal Name	Signal Description	Signal Direction	LPDS[1]	Hib[2]	nRESET = 0
3	GPIO12	I/O	No	No	No	GPIO_PAD_CONFIG_12 (0x4402 E0D0)	0	GPIO12	General Purpose I/O	I/O	Hi-Z	Hi-Z	Hi-Z
							3	McACLK	I2S Audio Port Clock O	O	Hi-Z		
							4	pVS (VSYNC)	Parallel Camera Vertical Sync	I	Hi-Z		
							5	I2C_SCL	I2C Clock	I/O (Open Drain)	Hi-Z		
							7	UART0_TX	UART0 TX Data	O	1		
							12	GT_CCP03	Timer Capture Port	I	Hi-Z		
4	GPIO13	I/O	Yes	No	No	GPIO_PAD_CONFIG_13 (0x4402 E0D4)	0	GPIO13	General-Purpose I/O	I/O	Hi-Z	Hi-Z	Hi-Z
							5	I2C_SDA	I2C Data	I/O (Open Drain)			
							4	pHS (HSYNC)	Parallel Camera Horizontal Sync	I			
							7	UART0_RX	UART0 RX Data	I			
							12	GT_CCP04	Timer Capture Port	I			
5	GPIO14	I/O		No	No	GPIO_PAD_CONFIG_14 (0x4402 E0D8)	0	GPIO14	General-Purpose I/O	I/O	Hi-Z	Hi-Z	Hi-Z
							5	I2C_SCL	I2C Clock	I/O (Open Drain)			
							7	GSPI_CLK	General SPI Clock	I/O			
							4	pDATA8 (CAM_D4)	Parallel Camera Data Bit 4	I			
							12	GT_CCP05	Timer Capture Port	I			

③ 使用 Pin Mux Tool 这个工具，添加 GPIO 后就可以直接看到对应关系，配置需要使用到的 GPIO 就可以直接生成配置函数。值得一提的是这个工具生成的函数 GPIO_PIN 入口参数竟然是数字，作为一个个读程序的人来说，纯数字是没有任何意义的（大大的吐槽一下），而且驱动库里明明有相应的宏定义，官方却没有使用。



好了，通过以上几种方式我们知道了芯片引脚编号和 GPIO 引脚的映射关系了，就差 GPIO 的分组了，这个就稍微简单一点，GPIO00~31 共 32 个 IO（能用的没这么多）可以分

4 组（其实有 5 组），后面的编号对 8 取模就是组号，取余就是对于引脚号（这个算法可以在官方 BSP 中 gpio_if.c 中找到），比如 GPIO00 就对应 GPIOA0，GPIO_PIN0，GPIO09 就对应 GPIOA1，GPIO_PIN1。

到此为止终于是弄清了这个映射关系，终于可以愉快的写代码了，首先咱们让我们板子上的 led 闪烁起来：首先外部声明全局向量表和 BoardInit 这俩直接从官方例程复制即可，咱们只需要编写 GPIO_Init 和 main 函数即可，GPIO 的初始化首先要使能相应 GPIO 组的时钟，通过前面的方式我们只知道板子上的 LED7 为 PIN_64 对应 GPIO09，对应 GPIOA1，GPIO_PIN_1，设置引脚模式，和输入输出方向即可成功初始化，这里我们初始化之后默认输出 0，再在主循环里翻转 GPIO 的输出电平即可达到 LED 闪烁效果。

下面是驱动板子上的 LED 代码

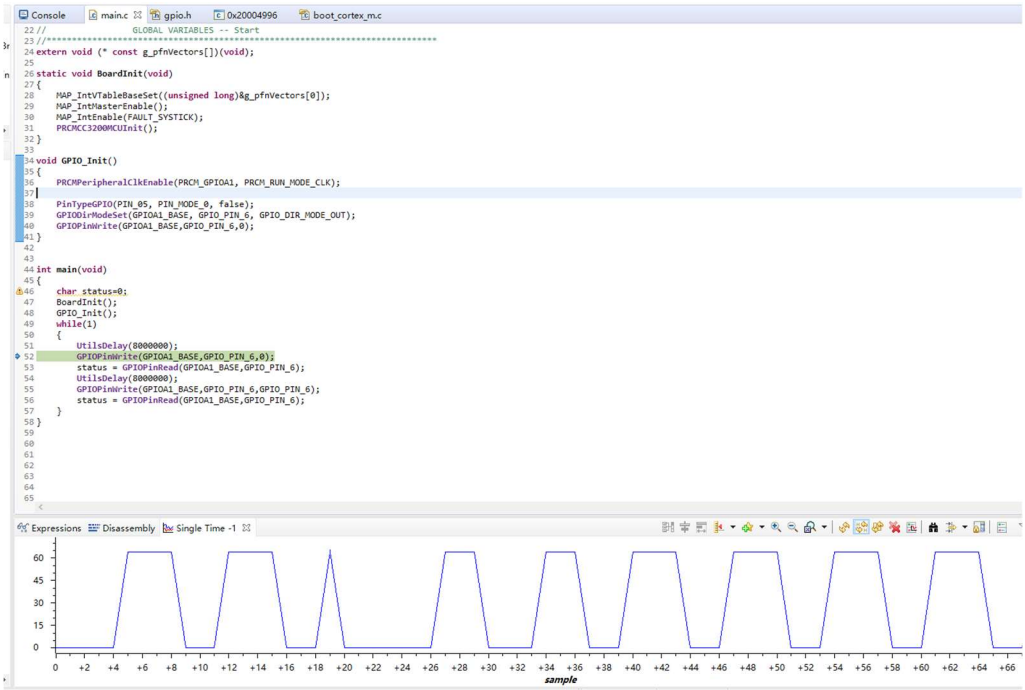
Console　main.c

```c
#include "prcm.h"
#include "gpio.h"
#include "utils.h"


//*****************************************************************************
//                    GLOBAL VARIABLES -- Start
//*****************************************************************************
extern void (* const g_pfnVectors[])(void);

static void BoardInit(void)
{
    MAP_IntVTableBaseSet((unsigned long)&g_pfnVectors[0]);
    MAP_IntMasterEnable();
    MAP_IntEnable(FAULT_SYSTICK);
    PRCMCC3200MCUInit();
}

void GPIO_Init()
{
    PRCMPeripheralClkEnable(PRCM_GPIOA1, PRCM_RUN_MODE_CLK);

    //PIN_64 --> D7
    PinTypeGPIO(PIN_64, PIN_MODE_0, false);
    GPIODirModeSet(GPIOA1_BASE, GPIO_PIN_1, GPIO_DIR_MODE_OUT);
    GPIOPinWrite(GPIOA1_BASE,GPIO_PIN_1,0);
}


int main(void)
{
    BoardInit();
    GPIO_Init();
    while(1)
    {
        UtilsDelay(8000000);
        GPIOPinWrite(GPIOA1_BASE,GPIO_PIN_1,0);
        UtilsDelay(8000000);
        GPIOPinWrite(GPIOA1_BASE,GPIO_PIN_1,GPIO_PIN_1);
    }
}
```

代码写好了，我们来试试 Debug（注意，需要验证代码时我们只需要 Debug 即可，没有必要每次都烧写到 flash 里面，毕竟烧写是比较麻烦的，咱们直接下载到 RAM 里调试即可，这个我们直接 Debug，cmd 文件官方已经写好，我们无需修改。需要 Debug 时直接右

键工程名选择 Debug As -> CCDS，进入调试界面后可以一直运行或者单步运行等等，包括断点等等调试手段都是有的），点击一直运行后可以发现板子上的 LED 闪烁起来了，不过复位或者掉电之后就啥也没有了。

下面我们来试试其他 IO 行不行，其他 IO 没接东西，想知道他的状态咱们可以使用 read 函数读取。可以看到也是可以的。



OK，以上就是使用库函数点灯的方式，下面咱们来讲一讲更底层的寄存器（有兴趣的朋友可以继续看下去）。

首先呢咱们使用寄存器开发的话得需要更详细的资料，咱们可以去找 CC3200 的寄存器手册（swru367，572 页），刚才的数据手册（72 页）就没多大作用了。

这次呢，同样的使 LED7 闪烁起来，只不过是直接使用寄存器实现的。

初始化 GPIO，即使能 GPIO 相应端口的时钟，咱们使用到的 GPIO 为 GPIOA1 的 GPIO_PIN_1，所以呢咱们翻到手册第 487 页，需要使能 GPIO1 的时钟

15.7.1.16 GPIO1CLKEN Register (offset = 58h) [reset = 0h]

GPIO1CLKEN is shown in Figure 15-19 and described in Table 15-19.

Figure 15-19. GPIO1CLKEN Register

31	30	29	28	27	26	25	24
RESERVED							
R-0h							
23	22	21	20	19	18	17	16
RESERVED							DSLPCLKEN
R-0h							R/W-0h
15	14	13	12	11	10	9	8
NU1							SLPCLKEN
R-0h							R/W-0h
7	6	5	4	3	2	1	0
NU2							RUNCLKEN
R-0h							R/W-0h

Table 15-19. GPIO1CLKEN Register Field Descriptions

Bit	Field	Type	Reset	Description
31-17	RESERVED	R	0h	
16	DSLPCLKEN	R/W	0h	GPIO_B_DSLP_CLK_ENABLE 0h = Disable GPIO_B clk during deep-sleep mode 1h = Enable GPIO_B clk during deep-sleep mode
15-9	NU1	R	0h	
8	SLPCLKEN	R/W	0h	GPIO_B_SLP_CLK_ENABLE 0h = Disable GPIO_B clk during sleep mode 1h = Enable GPIO_B clk during sleep mode
7-1	NU2	R	0h	
0	RUNCLKEN	R/W	0h	GPIO_B_RUN_CLK_ENABLE 0h = Disable GPIO_B clk during run mode 1h = Enable GPIO_B clk during run mode

可以看到想要使能时钟需要将 GPIO1CLKEN 寄存器的最低位置 1，即使能时钟在运行模式，那么这个寄存器的地址是多少呢？首先这个寄存器是属于 PRCM（Power，Reset and Clock Management）寄存器组的，我们得知道他的基地址，见下图可看出 PRCM 基地址为 0x44025000，而上面的土中则可以看到 GPIO1CLKEN 的偏移地址为 0x58，所以呢，GPIO1CLKEN 的最终地址为 0x44025058。

Table 2-4. Memory Map (continued)

Start Address	End Address	Description	Comment
0x04003.3000	0x4003.3FFF	General-purpose timer A3	
0x400F.7000	0x400F.7FFF	Configuration registers	
0x400F.E000	0x400F.EFFF	System control	
0x400F.F000	0x400F.FFFF	µDMA	
0x4200.0000	0x43FF.FFFF	Bit band alias of 0x4000.0000 through 0x400F.FFFF	
0x4401.C000	0x4401.EFFF	McASP	
0x4402.0000	0x4402.0FFF	FlashSPI	Used for external serial flash
0x4402.1000	0x4402.2FFF	GSPI	Used by application processor
0x4402.5000	0x4402.5FFF	MCU reset clock manager	
0x4402.6000	0x4402.6FFF	MCU configuration space	
0xE000.0000	0xE000.0FFF	Instrumentation trace Macrocell ™	
0xE000.1000	0xE000.1FFF	Data watchpoint and trace (DWT)	
0xE000.2000	0xE000.2FFF	Flash patch and breakpoint (FPB)	
0xE000.E000	0xE000.EFFF	Cortex-M4 Peripherals (NVIC, SysTick,SCB)	
0xE004.0000	0xE004.0FFF	Trace port interface unit (TPIU)	
0xE004.1000	0xE004.1FFF	Reserved for embedded trace macrocell (ETM)	

时钟使能之后呢，咱们需要配置引脚，寄存器地址如下

16.8.1 Pad Configuration Registers for Application Pins

Table 16-11 lists the configuration registers associated with the device pins.

Table 16-11. Pad Configuration Registers

Package Pin #	GPIO #[1]	Digital Pad Config Register	Physical Address
50	GPIO0 / (ANA)	GPIO_PAD_CONFIG_0	0x4402 E0A0
55	GPIO1	GPIO_PAD_CONFIG_1	0x4402 E0A4
57	GPIO2	GPIO_PAD_CONFIG_2	0x4402 E0A8
58	GPIO3	GPIO_PAD_CONFIG_3	0x4402 E0AC
59	GPIO4	GPIO_PAD_CONFIG_4	0x4402 E0B0
60	GPIO5	GPIO_PAD_CONFIG_5	0x4402 E0B4
61	GPIO6	GPIO_PAD_CONFIG_6	0x4402 E0B8
62	GPIO7	GPIO_PAD_CONFIG_7	0x4402 E0BC
63	GPIO8	GPIO_PAD_CONFIG_8	0x4402 E0C0
64	GPIO9	GPIO_PAD_CONFIG_9	0x4402 E0C4
1	GPIO10	GPIO_PAD_CONFIG_10	0x4402 E0C8
2	GPIO11	GPIO_PAD_CONFIG_11	0x4402 E0CC
3	GPIO12	GPIO_PAD_CONFIG_12	0x4402 E0D0

(1) Pins are referred either by number or by the GPIO mapped to that pin. Functionalities available on a pin are not limited to that name. Refer to the pin-mux table for all possible functional mapping.

Config 寄存器地址知道了，怎么配置呢，见下图

16.8.1.1.1 GPIO_PAD_CONFIG_0 to GPIO_PAD_CONFIG_32

Table 16-12. GPIO_PAD_CONFIG_0 to GPIO_PAD_CONFIG_32 Register Description

Bit	Field	Type	Reset	Description
31-12	Reserved	R	0	

Table 16-12. GPIO_PAD_CONFIG_0 to GPIO_PAD_CONFIG_32 Register Description (continued)

Bit	Field	Type	Reset	Description
11-0	MEM_GPIO_PAD_CONFIG	RW	0xC61	[3:0] CONFMODE. Determines which functional signal is routed to the pad. Refer to the pin mux table. [4] Enable open-drain mode (for example, when used as I2C). [7:5] DRIVESTRENGTH 111 = 14 mA 110 = 12 mA 101 = 10 mA 100 = 8 mA 011 = 6 mA 010 = 4 mA 001 = 2 mA 000 = Output driver not enabled [8] Enable internal weak pullup [9] Enable internal weak pulldown [10] Pad output enable override value. Level enables the pad output buffer. Otherwise, the output buffer is tristated. This does not affect the internal pullup or pulldown, which are controlled independently by bit 8 and bit 9. [11] This enables over-riding of the pad output buffer enable. When this bit is set to logic 1, the value in bit 4 controls the state of the pad output buffer. When this bit is set to logic 0, the state of the pad output buffer is directly controlled by the peripheral module to which the pad is functionally muxed.

可以看到低 4 位用来决定该引脚是用来干什么的，具体哪个值对应哪个功能呢，见下图

Table 16-7. Pin Multiplexing (continued)

General Pin Attributes						Function					Pad States		
Pkg Pin	Pin Alias	Use	Select as Wakeup Source	Config Addl Analog Mux	Muxed with JTAG	Dig. Pin Mux Config Reg	Dig. Pin Mux Config Mode Value	Signal Name	Signal Description	Signal Direction	LPDS[1]	Hib[2]	nRESET = 0
64	GPIO9	I/O	No	No	No	GPIO_PAD_CONFIG_9 (0x4402 E0C4)	0	GPIO9	General-Purpose I/O	I/O	Hi-Z	Hi-Z	Hi-Z
							3	GT_PWM05	Pulse Width Modulated O/P	O			
							6	SDCARD_DATA	SD Cad Data	I/O			
							7	McAXR0	I2S Audio Port Data (Rx/Tx)	I/O			
							12	GT_CCP00	Timer Capture Port	I			

这个寄存器我们还需要设置的是位【4】是否开漏输出，咱们驱动 LED 不需要，位【7】~【5】，设置驱动电流，一般为 2mA，后面的则是一些无关紧要的寄存器，后面需要用到再说，

引脚配置好之后呢咱们就可以配置 GPIO 这个外设了，在此之前咱们首先来看看寄存器表

5.5 GPIO_REGISTER_MAP Registers

Table 5-3 lists the memory-mapped registers for the GPIO_REGISTER_MAP. Each GPIO port can be accessed through the advanced peripheral bus (APB). The offset listed is a hexadecimal increment to the register address, relative to the base address of that GPIO port:

- GPIO Port A0: 0x4000.4000
- GPIO Port A1: 0x4000.5000
- GPIO Port A2: 0x4000.6000
- GPIO Port A3: 0x4000.7000

Each GPIO module clock must be enabled before the registers can be programmed. There must be a delay of 3 system clocks after the GPIO module clock is enabled before any GPIO module registers are accessed.

Table 5-3. GPIO_REGISTER_MAP Registers

Offset	Acronym	Register Name	Section
0h	GPIODATA	GPIO Data	Section 5.5.1.1
400h	GPIODIR	GPIO Direction	Section 5.5.1.2
404h	GPIOIS	GPIO Interrupt Sense	Section 5.5.1.3
408h	GPIOIBE	GPIO Interrupt Both Edges	Section 5.5.1.4
40Ch	GPIOIEV	GPIO Interrupt Event	Section 5.5.1.5
410h	GPIOIM	GPIO Interrupt Mask	Section 5.5.1.6
414h	GPIORIS	GPIO Raw Interrupt Status	Section 5.5.1.7
418h	GPIOMIS	GPIO Masked Interrupt Status	Section 5.5.1.8
41Ch	GPIOICR	GPIO Interrupt Clear	Section 5.5.1.9

咱们驱动 LED 的话只需要使用以上两个寄存器即可，其他寄存器都是用来配置中断的，首先咱们配置 GPIO 方向

5.5.1.2 GPIODIR Register (offset = 400h) [reset = 0h]

GPIODIR is shown in Figure 5-5 and described in Table 5-5.

The GPIODIR register is the data direction register. Setting a bit in the GPIODIR register configures the corresponding pin to be an output, while clearing a bit configures the corresponding pin to be an input. All bits are cleared by a reset, meaning all GPIO pins are inputs by default.

Figure 5-5. GPIODIR Register

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
RESERVED																								DIR							
R-0h																								R/W-0h							

Table 5-5. GPIODIR Register Field Descriptions

Bit	Field	Type	Reset	Description
31-8	RESERVED	R	0h	
7-0	DIR	R/W	0h	GPIO Data Direction 0h = Corresponding pin is an input. 1h = Corresponding pins is an output.

咱们把相应为配置为 1 即可输出。输入输出模式配好之后就可以控制 GPIO 输出高低电平啦，那怎么控制呢，见下图

5.5.1.1 GPIODATA Register (offset = 0h) [reset = 0h]

GPIODATA is shown in Figure 5-4 and described in Table 5-4.

The GPIODATA register is the data register. In software control mode, values written in the GPIODATA register are transferred onto the GPIO port pins if the respective pins have been configured as outputs through the GPIO Direction (GPIODIR) register.
The GPIODATA register has 256 aliased addresses from offset 0x000 to 0x3FF. A different address alias is used to directly read/ or write any combination of the 8 signal bits. This feature can help avoid time-consuming read-modify-writes and bit-masking operation for read-in software.
In this scheme, to write to GPIODATA, the corresponding bits in the mask, represented by the address bus bits [9:2], must be set. Otherwise, the bit values remain unchanged by the write.
Similarly, the values read from this register are determined for each bit by the mask bit derived from the alias address used to access the data register, bits [9:2]. Bits set in the address mask cause the corresponding bits in GPIODATA to be read, and bits that are clear in the address mask cause the corresponding bits in GPIODATA to be read as 0, regardless of their value.
A read from GPIODATA returns the last bit value written if the respective pins are configured as outputs, or it returns the value on the corresponding input pin when these are configured as inputs. All bits are cleared by a reset.

Figure 5-4. GPIODATA Register

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
RESERVED																								DATA							
R-0h																								R/W-0h							

Table 5-4. GPIODATA Register Field Descriptions

Bit	Field	Type	Reset	Description
31-8	RESERVED	R	0h	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7-0	DATA	R/W	0h	GPIO Data This register is virtually mapped to 256 locations in the address space. To facilitate the reading and writing of data to these registers by independent drivers, the data read from and written to the registers are masked by the eight address lines [9:2]. Reads from this register return its current state. Writes to this register only affect bits that are not masked by ADDR[9:2] and are configured as outputs.

DATA 里的 8 位数据分别控制 8 个 IO 口的电平，那前面这一堆英文是什么意思呢，这就需要看手册了

5.2.1.2 Data Register Operation

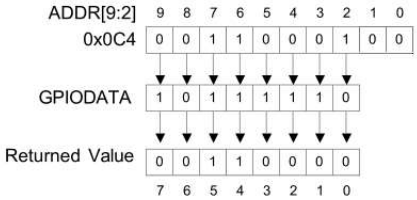
To aid in the efficiency of software, the GPIO ports allow for the modification of individual bits in the GPIO Data (GPIODATA) register by using bits [9:2] of the address bus as a mask. In this manner, software drivers can modify individual GPIO pins in a single instruction without affecting the state of the other pins. This method is more efficient than the conventional method of performing a read-modify-write operation to set or clear an individual GPIO pin. To implement this feature, the GPIODATA register covers 256 locations in the memory map.

During a write, if the address bit associated with that data bit is set, the value of the GPIODATA register is altered. If the address bit is cleared, the data bit is left unchanged. For example, writing a value of 0xEB to the address GPIODATA + 0x098 has the results shown in Figure 5-2, where u indicates that data is unchanged by the write. This example demonstrates how GPIODATA bits 5, 2, and 1 are written with a single operation by using GPIODATA address alias 0x098 (offset address with regards to the base of the respective GPIO instance A0 to A4).

Figure 5-2. GPIODATA Write Example

During a read, if the address bit associated with the data bit is set, the value is read. If the address bit associated with the data bit is cleared, the data bit is read as a zero, regardless of its actual value. For example, reading address GPIODATA + 0x0C4 yields as shown in Figure 5-3. This example shows how to read GPIODATA bits 5, 4, and 0 with a single operation by using GPIODATA address alias 0x0C4 (offset address with regard to the base of the respective GPIO instance S0 to S4).

Figure 5-3. GPIODATA Read Example



乍一看还是比较复杂难懂，在看了半天代码后终于有一点明悟了，对于写 GPIODATA 来说，0x098 代表什么意思？意思是说我只想让 GPIO_PIN_5、2、1 改变，其他 GPIO 就算你往里写 1 也不会改变，那为什么不直接使用低 8 位来表示，而要左移两位呢，因为我们的 GPIO 的的引脚数值是递增 1 的（原本可以是位【0】代表 GPIO0，位【1】代表 GPIO1），但是我们的 DATA 寄存器却是 32 位的，每加 1，地址需要往后移 4 个字节，所以就造成了需要左移两位的情况，就是说现在是这样的，8 个 GPIO 哪个需要改变哪个不需要改变总共有 256 中排列组合（2 的 8 次方），所以呢我们的 ti 工程师就在这一片地址空间里总共设置了 256 个地址，每个地址都可以独立配置，这样做可以方便软件（这样的硬件真贴心）,如何方便呢，可能你还没有明白，简单举个例子，比如地址（偏移）为 0x000，就是每个 GPIO 都不想改变，你往这个地址里写任何值 GPIO 的电平都不会有变化，那地址（偏移）为 0x004 呢（低 2 位必须为 0 ，因为我们的寄存器是 32 位的，一下得增加 4 字节），就是只想改变 GPIO_PIN_0，就算你往里写 0xff，也只会把 GPIO_PIN_0 写成 1，其他 GPIO 原来是什么电平现在就是什么电平，丝毫不为所动，这下大家懂了吧。

评价一下这种读写配置方式，嗯虽然对于软件的编写来说是方便了点，不过这个文档看起来就略微有点费劲了，而且感觉有点浪费地址。不过呢地址空间够大（32 位机最大能用 4 个 G 的地址空间），浪费一点也没啥关系。好了下面咱们开始愉快的编写代码吧。

编码的时候就很简单了，直接往寄存器里写值就好了，写好的代码如下图

Console | main.c | gpio.h | boot_cortex_m.c | cc3200v1p32.cmd | hw_memmap.h | gpio.c | hw_

```c
#include "interrupt.h"
#include "hw_apps_rcm.h"
#include "prcm.h"
#include "pin.h"
#include "rom.h"
#include "rom_map.h"
#include "prcm.h"
#include "gpio.h"
#include "utils.h"


//*****************************************************************************
//                    GLOBAL VARIABLES -- Start
//*****************************************************************************
extern void (* const g_pfnVectors[])(void);

static void BoardInit(void)
{
    MAP_IntVTableBaseSet((unsigned long)&g_pfnVectors[0]);
    MAP_IntMasterEnable();
    MAP_IntEnable(FAULT_SYSTICK);
    PRCMCC3200MCUInit();
}

void GPIO_Init()
{
    *((volatile unsigned long *)(0x44025000 + 0x58)) = 0x01;//enable GPIOA1 clock
    *((volatile unsigned long *)0x4402e0c4)          = 0x20;//GPIO09
    *((volatile unsigned long *)(0x40005000 + 0x400))= 0x02;//GPIOA1 GPIO_PIN_1 set to output

    *((volatile unsigned long *)(0x40005000 + 0x00 + (0x02<<2))) = 0xff;//GPIO_PIN_1 write 1   0x02 -> 0000 0010
}

int main(void)
{
    BoardInit();
    GPIO_Init();
    while(1)
    {
        UtilsDelay(8000000);
        *((volatile unsigned long *)(0x40005000 + 0x00 + (0x02<<2))) = 0x00;//only GPIO_PIN_1 will be changed
        UtilsDelay(8000000);
        *((volatile unsigned long *)(0x40005000 + 0x00 + (0x02<<2))) = 0xff;//only GPIO_PIN_1 will be changed
    }
}
```

可以看到我们往寄存器里写值是这样的*((volatile unsigned long *)值)，有些同学可能不知道这是什么意思啊，我稍微解释一下，(volatile unsigned long *)这个是强制转化，转换成什么呢，可变的无符号长整型指针，这个指针指向的地址就是这个“值”，然后我们使用*来引用这个指针，往里写值或者读取这个地址里的值。怎么样，是不是感觉很简单。

OK，既然代码写好了，下面咱们来 debug 一下（Debug 的时候板子上的跳线帽保持出厂时候的样子就行了，不像烧写代码的时候一样要插一个跳冒），右键工程名选择 Debug As -> CCDS，进入调试状态，点击一直运行就可以看到板子上的 LED7 欢快的闪烁起来啦。

好了，以上就是今天的全部内容，谢谢大家！！