TI Stellaris LM4F 定时器(Timer)指南

[作者: Richard Ma]
[Email: mxschina@gmail.com]

Texas Instruments 在 Stellaris LM4F 上提供了更多更强大的定时器(Timer)模块，除了传统的 32/16-bit 定时器，更增加了 64/32-bit 定时器。这些定时器均支持定时器(Timer)模式、捕捉(Capture)模式以及 PWM 模式。本文主要介绍 Timer 和 Capture 模式的用法。

1. 功能介绍

LM4F 的定时器模块有两种，一种是 32/16-bit 的，另一种是 64/32-bit 的。每一个定时器模块，可以单独工作(如 32/16-bit 作为 32-bit 定时器使用)，或拆为两个独立定时器 A 和 B 进行工作(如 32/16-bit 作为两个 16-bit 定时器使用)。

下表对定时器模块支持的功能做了一个汇总：

工作模式		拆分模式	计数方向	计数大小	
				32/16-bit 型	64/32-bit 型
定时器模式 (Timer)	单次运行	整体	双向	32-bit	64-bit
		拆分	双向	16-bit	32-bit
	连续运行	整体	双向	32-bit	64-bit
		拆分	双向	16-bit	32-bit
	实时时钟	整体	加计数	32-bit	64-bit
捕捉模式 (Capture)	边沿计数	拆分	双向	16-bit	32-bit
	边沿计时	拆分	双向	16-bit	32-bit
PWM 模式	PWM	拆分	减计数	16-bit	32-bit

2. 定时器模式(Timer)

2.1. 单次运行与连续运行模式

定时器的基本功能为计数(包括加计数和减计数两种)，Stellaris LM4F 中以系统时钟为计数节拍(当计数器被拆分使用时可以使用预分频功能，为了简单起见这里不作讨论)。当加计数时，计数器由零开始，逐步加一，直到到达用户预设值；减计数则由某一用户预设值开始，逐步减一，直到计数为零。每当计数完成，则会置相应的状态位(包括中断)，提示计时完成。

单次运行与连续运行工作时没有区别，不同的是单次运行在完成一次计时后会自动停止，连续模式下定时器会自动从计数起点开始(根据计数方向为零或用户设定值)继续计时。

2.2. 定时器程序设置

让定时器模块正常工作起来需要以下几步：

1) 启用时钟模块

使用 SysCtlPeripheralEnable 函数启用相应的定时器模块。

程序示例：

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_WTIMER0);
```

在 StellarisWare 中，32/16-bit 定时器模块为 TIMER，64/32-bit 定时器模块为 WTIMER (Wide Timer)。除了名字不同、计数范围不同外没有其它区别。

2) 设置时钟模块工作模式

使用 TimerConfigure 函数对定时器模块的工作模式进行设置，将其设置为定时器功能。

程序示例：

```
TimerConfigure(TIMER0_BASE, TIMER_CFG_ONE_SHOT);

TimerConfigure(WTIMER2_BASE, TIMER_CFG_SPLIT_PAIR |
        TIMER_CFG_A_ONE_SHOT | TIMER_CFG_B_PERIODIC);
```

在不拆分的情况下，可以用下面参数中的一个将模块设置成所需的定时器模式：

TIMER_CFG_ONE_SHOT – 单次减计数模式
TIMER_CFG_ONE_SHOT_UP – 单次加计数模式
TIMER_CFG_PERIODIC – 连续减计数模式

TIMER_CFG_PERIODIC_UP – 连续加计数模式
TIMER_CFG_RTC – 实时时钟模式

如果需要将计时器拆分，则需使用参数 TIMER_CFG_SPLIT_PAIR 然后用“|”号连接被拆分定时器 A、B 的设置。如果只使用了一个可以只设置用到的那个。拆分出来的定时器 A 和 B 的设置方法是一样的，只是函数名中各自用 A 和 B：

TIMER_CFG_A_ONE_SHOT – 定时器 A 单次减计数
TIMER_CFG_A_ONE_SHOT_UP –定时器 A 单次加计数
TIMER_CFG_A_PERIODIC – 定时器 A 连续减计数
TIMER_CFG_A_PERIODIC_UP – 定时器 A 连续加计数

TIMER_CFG_B_ONE_SHOT – 定时器 B 单次减计数
TIMER_CFG_B_ONE_SHOT_UP –定时器 B 单次加计数
TIMER_CFG_B_PERIODIC – 定时器 B 连续减计数
TIMER_CFG_B_PERIODIC_UP – 定时器 B 连续加计数

3) 设置时钟的计数范围

使用 TimerLoadSet、TimerLoadSet64 函数可以为计数设置范围。设置未拆分使用的 64/32-bit 定时器模块，需要使用 TimerLoadSet64 函数，对其它模块、其它状况的设置使用 TimerLoadSet 函数。计数范围为设置值到零(加计数: 0~预设值，减计数: 预设值~0)。

程序示例：

```
TimerLoadSet64(TIMER3_BASE, 80000);
TimerLoadSet(WTIMER0_BASE, TIMER_B, 10000);
```

4) 启动时钟

使用 TimerEnable 函数启动定时器。可以用的参数有 TIMER_A、TIMER_B 和 TIMER_BOTH。可以分别或同时启动 A、B 定时器。如果定时器没有拆分，直接使用 TIMER_A 即可。

程序示例：

```
TimerEnable(WTIMER0_BASE, TIMER_B);
```

2.3. 定时器读取及中断设置

1) 计数值读取

可以使用 TimerValueGet 函数和 TimerValueGet64 函数获得定时器当前的计数值。需要注意的是 TimerValueGet64 返回的是 64 位结果。

程序示例：

```
long val = TimerValueGet(TIMER1_BASE, TIMER_A);
long long timer_val = TimerValueGet64(WTIMER3_BASE);
```

2) **中断设置**

一般定时器多用中断响应以满足时间要求。可以用 TimerIntRegister 向系统注册中断处理函数，用 TimerIntEnable 来允许某个定时器的中断请求。需要注意的是，在 M4 中还应该用 IntEnable 在系统层使能定时器的中断。当然，系统总中断开关也必须用 IntMasterEnable 使能。

TimerIntEnable 在该模式下可以支持：

```
TIMER_TIMA_TIMEOUT
TIMER_TIMB_TIMEOUT
```

程序示例：

```
TimerIntRegister(WTIMER0_BASE, TIMER_B, WTimer0BIntHandler);
IntMasterEnable();
TimerIntEnable(WTIMER0_BASE, TIMER_TIMB_TIMEOUT);
IntEnable(INT_WTIMER0B);
```

在 Timer 中断中，需要手工清除中断标志位，可以使用如下代码：

```
unsigned long ulstatus = TimerIntStatus(TIMER4_BASE,
                                        TIMER_TIMA_TIMEOUT | TIMER_TIMB_TIMEOUT);
TimerIntClear(TIMER4_BASE, ulstatus);
```

2.4. 示例程序

下面的程序利用定时器每隔一秒向串口发送一个计数：

```
// ================================================
// Code Begin
// ================================================

// Stellaris 硬件定义及 StellarisWare 驱动定义头文件
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_timer.h"
#include "inc/hw_ints.h"
#include "driverlib/timer.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "utils/uartstdio.h"

// 用于记录进入定时器中断的次数
unsigned long g_ulCounter = 0;
```

```
// 初始化 UART 的函数
extern void InitConsole(void);

// 定时器的中断处理函数
Void WTimer0BIntHandler(void)
{
    // 清除当前中断标志
    TimerIntClear(WTIMER0_BASE, TIMER_TIMB_TIMEOUT);

    // 更新进入中断次数的计数
    g_ulCounter++;
}

// 主程序
int main(void)
{
    // 用来记录上次计数以判断是否计数改变
    unsigned long ulPrevCount = 0;

    // 设置 LM4F 时钟为 50MHz
    SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL |
                   SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);

    // 使能 64/32-bit 的时钟模块 WTIMER0
    SysCtlPeripheralEnable(SYSCTL_PERIPH_WTIMER0);

    // 初始化 UART
    InitConsole();

    // 打印程序信息
    UARTprintf("32-Bit Timer Interrupt ->");
    UARTprintf("\n   Timer = Wide Timer0B");
    UARTprintf("\n   Mode = Periodic");
    UARTprintf("\n   Rate = 1s\n\n");

    // 设置 WTimer0-B 模块为连续减计数
    TimerConfigure(WTIMER0_BASE, TIMER_CFG_SPLIT_PAIR |
                   TIMER_CFG_B_PERIODIC);

    // 设置定时器的计数值，这里用系统频率值，即每秒一个中断
    TimerLoadSet(WTIMER0_BASE, TIMER_B, SysCtlClockGet());

    // 设置 WTimer0-B 的中断处理函数
    TimerIntRegister(WTIMER0_BASE, TIMER_B, WTimer0BIntHandler);

    // 启用系统总中断开关
    IntMasterEnable();

    // 启用 WTimer0-B 超时中断
```

```
    TimerIntEnable(WTIMER0_BASE, TIMER_TIMB_TIMEOUT);


    // 在系统层面(NVIC)使能 WTimer0-B 中断
    IntEnable(INT_WTIMER0B);

    // 启动定时器
    TimerEnable(WTIMER0_BASE, TIMER_B);

    while(1)
    {
        // 循环等待 WTimer0-B 中断更新 g_ulCounter 计数
        // 若计数改变则进行 UART 输出
        if(ulPrevCount != g_ulCounter)
        {
            // UART 输出计数值
            UARTprintf("Number of interrupts: %d\r", g_ulCounter);
            ulPrevCount = g_ulCounter;
        }
    }
}
```

3. 捕捉模式(Capture)

3.1. 边沿计数模式

3.1.1. 功能介绍

计时器模块可以通过器件的 I/O 脚来捕捉电平边沿，支持分别捕捉上升沿、下降沿，以及同时捕捉上升下降沿。硬件层面上对电平保持时间的要求是宽度要大于 2 个时钟周期。在这个模式下，因为可以使用分频器 (Prescaler)，计数范围扩大到 24/48-bit(64-bit 未拆分模式下不可以用 Capture)。

边沿计数值也可以加计数、减计数。加计数的计数范围为从零到用户预设的(Match)值。减计数的范围为从预设(Preload)值到预设(Match)值。

加计数与减计数模式的另一个重要差异是计数终止时的差异。加计数模式下，计数器会自动清零并自动重新开始计数；减计数模式下，计数器会自动重新载入用户预设(Preload)值，但计数会停止，需要用户重新手工使能计数。不同模式如下表所示：

边沿计数模式	起始值	结束值	计数结束动作
加计数	0	Match 值	重置，继续工作
减计数	Preload 值	Match 值	重置，停止工作

当计数结束时，定时器模块会产生中断通知系统。中断标志位需要在中断处理函数中手工清除。

3.1.2. 边沿计数程序设置

要让定时器模块工作在捕捉-边沿计数模式下，主要需要以下几步：

1) I/O 管脚配置

边沿的捕捉需要将 I/O 脚作为定时器模块的捕捉输入使用，并配置相应的驱动类型(上拉、下拉或者开漏等)。

首先使用 GPIOPinConfigure 函数 为 I/O 配置使用定时器输入功能(TnCCP0 或 TnCCP1)，在 pinmap.h 头文件里可以找到针对不同器件的管脚定义。如

GPIOPinConfigure(GPIO_PM0_T4CCP0)表示将 GPIO_M 的第 0 脚作为 Timer4A 的捕捉输入。

接下来使用 GPIOPinTypeTimer 函数 和 GPIOPadConfigSet 函数完成 GPIO 的其它设置。

程序示例：

```
GPIOPinConfigure(GPIO_PM0_T4CCP0);
GPIOPinTypeTimer(GPIO_PORTM_BASE, GPIO_PIN_0);
GPIOPadConfigSet(GPIO_PORTM_BASE, GPIO_PIN_0,
                 GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);
```

2) 配置定时器模块为捕捉-边沿计数模式

使用 TimerConfigure 函数对定时器模块进行设置，因为定时器只能在拆分时作为捕捉(Capture)模式使用，所以 TIMER_CFG_SPLIT_PAIR 是必须的，然后可以使用下列参数分别设置边沿计数模式：

TIMER_CFG_A_CAP_COUNT – 模块 A 捕捉-边沿减计数模式
TIMER_CFG_A_CAP_COUNT_UP –模块 A 捕捉-边沿加计数模式
TIMER_CFG_B_CAP_COUNT –模块 B 捕捉-边沿减计数模式
TIMER_CFG_B_CAP_COUNT_UP –模块 B 捕捉-边沿加计数模式

程序示例：

```
TimerConfigure (TIMER4_BASE, TIMER_CFG_SPLIT_PAIR |
                             TIMER_CFG_A_CAP_COUNT);
```

3) 设置要捕捉的边沿

使用 TimerControlEvent 函数设置要捕捉的边沿，可以捕捉的边沿有：

TIMER_EVENT_POS_EDGE – 只捕捉上升沿
TIMER_EVENT_NEG_EDGE – 只捕捉下降沿
TIMER_EVENT_BOTH_EDGES – 同时捕捉上升和下降沿

程序示例：

```
TimerControlEvent(TIMER4_BASE, TIMER_A, TIMER_EVENT_NEG_EDGE);
```

4) 设置计数范围

加计数使用 TimerMatchSet 函数进行设置，计数范围为 0~设定值。
减计数使用 TimerLoadSet 函数设置起始值，使用 TimerMatchSet 函数设置结束值。

程序示例

```
TimerLoadSet(TIMER4_BASE, TIMER_A, 0x8FFF);
TimerMatchSet(TIMER4_BASE, TIMER_A, 0x8FFA);
```

5) 中断设置

可以用 TimerIntRegister 向系统注册中断处理函数，用 TimerIntEnable 来允许某个定时器的中断请求。需要注意的是，在 M4 中还应该用 IntEnable 在系统层使能定时器的中断。当然，系统总中断开关也必须用 IntMasterEnable 使能。

TimerIntEnable 在该模式下可以支持：

TIMER_CAPA_MATCH – 模块 A 计数到达预设值
TIMER_CAPB_MATCH – 模块 B 计数到达预设值

程序示例：

```
TimerIntRegister(WTIMER0_BASE, TIMER_B, WTimer0BIntHandler);
IntMasterEnable();
TimerIntEnable(WTIMER0_BASE, TIMER_CAPB_MATCH);
IntEnable(INT_WTIMER0B);
```

在 Timer 中断中，需要手工清除中断标志位，可以使用如下代码：

```
unsigned long ulstatus = TimerIntStatus(TIMER4_BASE,
                                        TIMER_CAPA_MATCH | TIMER_CAPB_MATCH);
TimerIntClear(TIMER4_BASE, ulstatus);
```

6) 启动定时器模块

使用 TimerEnable 函数启动定时器捕捉模式。可以用的参数有 TIMER_A、TIMER_B 和 TIMER_BOTH。可以分别或同时启动 A、B。

程序示例：

```
TimerEnable(WTIMER0_BASE, TIMER_B);
```

3.1.3. 边沿计数程序示例

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_timer.h"
#include "inc/hw_ints.h"
#include "inc/hw_gpio.h"
#include "driverlib/timer.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "utils/uartstdio.h"

// UART 初始化程序
extern void InitConsole(void);

void Timer4AIntHandler(void)
{
    unsigned long ulstatus;

    // 读取中断标志位
    ulstatus = TimerIntStatus(TIMER4_BASE, TIMER_CAPA_EVENT);

    // 清除中断标志位
    TimerIntClear(TIMER4_BASE, ulstatus);

    // 输出计数完成提示
    UARTprintf("Counting Finished!\n");

    // 因为减计数会自动停止，所以需要重新启用计数模块
    TimerEnable(TIMER4_BASE, TIMER_A);
}

int main(void)
{
    // 设置系统时钟为 50MHz
```

```
SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL |
                    SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);

// 启用 Timer4 模块
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER4);

// 启用 GPIO_M 作为脉冲捕捉脚
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOM);

// 配置 GPIO 脚为使用 Timer4 捕捉模式
GPIOPinConfigure(GPIO_PM0_T4CCP0);
GPIOPinTypeTimer(GPIO_PORTM_BASE, GPIO_PIN_0);

// 为管脚配置弱上拉模式
GPIOPadConfigSet(GPIO_PORTM_BASE, GPIO_PIN_0,
                    GPIO_STRENGTH_2MA,  GPIO_PIN_TYPE_STD_WPU);

// 初始化 UART 模块
InitConsole();

// 配置使用 Timer4 的 TimerA 模块为边沿触发减计数模式
TimerConfigure(TIMER4_BASE, TIMER_CFG_SPLIT_PAIR |
                    TIMER_CFG_A_CAP_COUNT);

// 使用下降沿触发
TimerControlEvent(TIMER4_BASE, TIMER_A, TIMER_EVENT_NEG_EDGE);

// 设置计数范围为 0x8FFF~0X8FFA
TimerLoadSet(TIMER4_BASE, TIMER_A, 0x8FFF);
TimerMatchSet(TIMER4_BASE, TIMER_A, 0x8FFA);

// 注册中断处理函数以响应触发事件
TimerIntRegister(TIMER4_BASE, TIMER_A, Timer4AIntHandler);

// 系统总中断开
IntMasterEnable();
```

```
    // 时钟中断允许，中断事件为 Capture 模式中边沿触发，计数到达预设值
    TimerIntEnable(TIMER4_BASE, TIMER_CAPA_MATCH);

    // NVIC 中允许定时器 A 模块中断
    IntEnable(INT_TIMER4A);

    // 启动捕捉模块
    TimerEnable(TIMER4_BASE, TIMER_A);

    // 循环，等待边沿计数完成
    while(1)
    {

    }
}
```

3.2. 边沿计时模式

3.2.1. 功能介绍

除了使用 I/O 脚捕捉边沿的个数，定时器模块可以捕捉上升、下降沿，在边沿到来时记录计数值，以测定脉冲间的时间间隔。加计时的计时范围为从零到用户预设的(Preload)值。减计时的范围为从预设(Preload)值到零，预设(Preload)值需要使用 TimerLoadSet 函数进行设置。最大计数范围与边沿计数模式相同，为 48/24-bit。

边沿计时模式	起始值	结束值	计数结束动作
加计时	0	Preload 值	重置，继续工作
减计时	Preload 值	0	重置，继续工作

每当边沿到来，定时器模块会产生中断并记录当前计时值，计时值可以稍后读出。中断标志位需要在中断处理函数中手工清除。

3.2.2. 边沿计时程序设置

边沿计时模块与边沿计数模块的设置方法基本相同，以下几个部分有所区别：

1) 需要配置定时器模块为捕捉-边沿计时模式

使用 TimerConfigure 函数对定时器模块进行设置，因为定时器只能在拆分时作为捕捉(Capture)模式使用，所以 TIMER_CFG_SPLIT_PAIR 是必须的，然后可以使用下列参数分别设置边沿计时模式：

TIMER_CFG_A_CAP_TIME – 模块 A 捕捉-边沿减计时模式
TIMER_CFG_A_CAP_ TIME _UP –模块 A 捕捉-边沿加计时模式
TIMER_CFG_B_CAP_ TIME–模块 B 捕捉-边沿减计时模式
TIMER_CFG_B_CAP_ TIME _UP –模块 B 捕捉-边沿加计时模式

程序示例：

```
TimerConfigure (TIMER4_BASE, TIMER_CFG_SPLIT_PAIR |
                                    TIMER_CFG_A_CAP_COUNT);
```

2) 计时范围设置

无论加计时与减计时，均需使用 TimerLoadSet 函数设置计时范围。计时范围：加计时为 0 ~ 设定值；减计时为设定值 ~ 0。

程序示例：

```
TimerLoadSet(TIMER4_BASE, TIMER_A, 0x5000);
```

3) 中断设置

可以用 TimerIntRegister 向系统注册中断处理函数，用 TimerIntEnable 来允许某个定时器的中断请求。需要注意的是，在 M4 中还应该用 IntEnable 在系统层使能定时器的中断。当然，系统总中断开关也必须用 IntMasterEnable 使能。

TimerIntEnable 在该模式下可以支持：

TIMER_CAPA_EVENT – 模块 A 计数到达预设值
TIMER_CAPB_EVENT – 模块 B 计数到达预设值

程序示例：

```
TimerIntRegister(TIMER4_BASE, TIMER_B, Timer4BIntHandler);
IntMasterEnable();
TimerIntEnable(TIMER4_BASE, TIMER_CAPB_EVENT);
```

IntEnable(INT_TIMER4B);

在 Timer 中断中，需要手工清除中断标志位，可以使用如下代码：

```
unsigned long ulstatus = TimerIntStatus(TIMER4_BASE,
                                        TIMER_CAPA_EVENT | TIMER_CAPB_EVENT);
TimerIntClear(TIMER4_BASE, ulstatus);
```

3.2.3. 边沿计时程序示例

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_timer.h"
#include "inc/hw_ints.h"
#include "inc/hw_gpio.h"
#include "driverlib/timer.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "utils/uartstdio.h"

// UART 初始化程序
extern void InitConsole(void);

void Timer4AIntHandler(void)
{
    unsigned long ulstatus;

    // 读取中断标志位
    ulstatus = TimerIntStatus(TIMER4_BASE, TIMER_CAPA_EVENT);

    // 清除中断标志位
    TimerIntClear(TIMER4_BASE, ulstatus);

    // 输出捕捉到的计数值
    UARTprintf("Captured Value: 0x%04X\n",
                          TimerValueGet(TIMER4_BASE, TIMER_A));
```

```
}

int main(void)
{
    // 设置系统时钟为 50MHz
    SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL |
                   SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);

    // 启用 Timer4 模块
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER4);

    // 启用 GPIO_M 作为脉冲捕捉脚
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOM);

    // 配置 GPIO 脚为使用 Timer4 捕捉模式
    GPIOPinConfigure(GPIO_PM0_T4CCP0);
    GPIOPinTypeTimer(GPIO_PORTM_BASE, GPIO_PIN_0);

    // 为管脚配置弱上拉模式
    GPIOPadConfigSet(GPIO_PORTM_BASE, GPIO_PIN_0,
                     GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);

    // 初始化 UART 模块
    InitConsole();

    // 配置使用 Timer4 的 TimerA 模块为沿触发加计时模式
    TimerConfigure(TIMER4_BASE, TIMER_CFG_SPLIT_PAIR |
                                    TIMER_CFG_A_CAP_TIME_UP);

    // 使用下降沿触发
    TimerControlEvent(TIMER4_BASE, TIMER_A, TIMER_EVENT_NEG_EDGE);

    // 设置计数范围为 0~0x8FFF
    TimerLoadSet(TIMER4_BASE, TIMER_A, 0x8FFF);

    // 注册中断处理函数以响应触发事件
```

```
    TimerIntRegister(TIMER4_BASE, TIMER_A, Timer4AIntHandler);

    // 系统总中断开
    IntMasterEnable();

    // 时钟中断允许，中断事件为 Capture 模式中边沿触发
    TimerIntEnable(TIMER4_BASE, TIMER_CAPA_EVENT);

    // NVIC 中允许定时器 A 模块中断
    IntEnable(INT_TIMER4A);

    // 启动捕捉模块
    TimerEnable(TIMER4_BASE, TIMER_A);

    // 循环，等待边沿触发
    while(1)
    {

    }
}
```

4. 附录

InitConsole 函数示例代码：

```
//*****************************************************************
// This function sets up UART0 to be used for a console to
// display information as the example is running.
//*****************************************************************
void InitConsole(void)
{
    // Enable GPIO port A which is used for UART0 pins.
    // TODO: change this to whichever GPIO port you are using.
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    // Configure the pin muxing for UART0 functions on port A0
    // and A1.
    // This step is not necessary if your part does not support
    // pin muxing.
    // TODO: change this to select the port/pin you are using.
    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);

    // Select the alternate (UART) function for these pins.
    // TODO: change this to select the port/pin you are using.
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    // Initialize the UART for console I/O.
    UARTStdioInit(0);
}
```