

嵌入式系统原理与实验

Cortex M3 实验教程（上）

上海交通大学 嵌入式系统原理与实验课程组

2012. 3

目 录

第一章	通用输入/输出接口	1
1.1	GPIO 模块基本特性	1
1.2	GPIO 寄存器映射	1
1.3	GPIO 初始化和配置	3
1.4	板上 GPIO 资源及特性	4
1.5	Stellaris 库函数	6
1.6	实验一 GPIO 操作编程	9
1.7	思考与练习题	16
第二章	系统控制单元：时钟控制模块	17
2.1	概述	17
2.2	功率控制	17
2.3	时钟控制	17
2.4	Stellaris 库函数	19
2.5	自编封装函数介绍	22
2.6	实验二 时钟控制编程	22
2.7	思考与练习题	29
第三章	SysTick 及通用定时器模块	30
3.1	概述	30
3.2	系统定时器 (SysTick)	30
3.3	Timer 模块	31
3.4	Stellaris 库函数	32
3.5	实验三 SysTick 及 Timer 编程	35
3.6	思考与练习题	41
第四章	中 断	42
4.1	中断基本概念	42
4.2	Cortex-M3 内核异常与 NVIC	43
4.3	外部中断/事件控制器	45
4.4	Stellaris 中断基本编程方法	46
4.5	实验四 中断编程	50
4.6	思考与练习题	57
第五章	看门狗定时器	58
5.1	看门狗(Watchdog)简介	58
5.2	WatchDog 功能概述	58
5.3	Stellaris 库函数	59
5.4	实验五 Watchdog Timer 编程	61
5.5	思考与练习题	67
第六章	通用异步收发传输器	68
6.1	UART 简介	68
6.2	UART 功能概述	68
6.3	Stellaris 函数库	72

6.4	实验六 UART	76
6.5	思考与练习题	84
附录 A	Keil 建立工程文件步骤	85
附录 B	S700 实验板硬件电路使用说明	95

第一章 通用输入/输出接口

1.1 GPIO 模块基本特性

Stellaris LM3S9B96微控制器的通用输入/输出端口(General-Purpose Input Outputs, 简称为GPIOs)模块由9 个物理GPIO 模块组成, 每一个物理GPIO 模块对应一个端口(Port A~J)。GPIO 模块支持高达65 个可编程的输入/输出引脚, 具体取决于正在使用的外设。GPIO 模块基本特性包括:

- 所有的GPIO 引脚都可以用作GPIO 或是一种或多种的外设功能, 当配置为输入模式可承受5V 电压, 配置为数字输入的引脚均为施密特触发。
- 通过编程, 可使GPIO端口具备中断功能: 屏蔽中断发生; 中断触发方式可配置为上升沿、下降沿、双边沿、(高或低) 电平触发。
- 读写操作时可通过地址线进行位屏蔽的操作。
- 通过编程可控制的GPIO 引脚配置方式: 弱上拉或下拉电阻、2mA, 4mA 或8mA 驱动电流, 对于需要大电流的应用最多可以有四个引脚可以配置为18mA; 8mA 驱动的斜率控制; 开漏使能、数字输入使能。

1.2 GPIO 寄存器映射

为了利用GPIO端口达到所需的功能, 首先应对GPIO端口进行配置(编程)。配置方法就是对与某一GPIO端口相关的寄存器进行读写编程。可以通过两种方法来访问GPIO模块, 一种称为先进外设总线(APB), 向后兼容以前的Stellaris®产品, 这是一种较陈旧的方法。另外一种先进高端总线(AHB), 它和APB一样拥有相同的寄存器映射, 但是提供了比APB更好的访问性能。但是这两种访问方式只能选择一种使用。可以通过GPIOHBCTL寄存器来确定使用哪种方式访问。

与GPIO端口对应的基本地址如下:

表1.1

GPIO 端口	基本地址(APB)	基本地址(AHB)
A	0x4000.4000	0x4005.8000
B	0x4000.5000	0x4005.9000
C	0x4000.6000	0x4005.A000
D	0x4000.7000	0x4005.B000
E	0x4002.4000	0x4005.C000
F	0x4002.5000	0x4005.D000
G	0x4002.6000	0x4005.E000
H	0x4002.7000	0x4005.F000
J	0x4003.D000	0x4006.0000

与GPIO端口相关的寄存器的地址、名称、读写类型、复位值及其含义如表1.2所示。这里寄存器的地址是通过基本地址的偏移量给出的。例如, 端口A的GPIODIR寄存器的地址为0x4000.4400 (APB访问方式) 或0x4005.8400

(AHB访问方式)。

表1.2

偏移量	名称	读写类型	复位值	功能
0x000	GPIODATA	R/W	0x0000.0000	GPIO数据
0x400	GPIODIR	R/W	0x0000.0000	GPIO方向
0x404	GPIOIS	R/W	0x0000.0000	GPIO 中断检测
0x408	GPIOIBE	R/W	0x0000.0000	GPIO 中断双边沿检测
0x40C	GPIOIEV	R/W	0x0000.0000	GPIO 中断事件
0x410	GPIOIM	R/W	0x0000.0000	GPIO 中断屏蔽
0x414	GPIORIS	RO	0x0000.0000	GPIO 原始中断状态
0x418	GPIONIS	RO	0x0000.0000	GPIO 屏蔽后的中断状态
0x41C	GPIOICR	W1C	0x0000.0000	GPIO 中断清除
0x420	GPIOAFSEL	R/W	-	GPIO 备用功能选择
0x500	GPIONR2R	R/W	0x0000.00FF	GPIO 2mA 驱动选择
0x504	GPIONR4R	R/W	0x0000.0000	GPIO 4mA 驱动选择
0x508	GPIONR8R	R/W	0x0000.0000	GPIO 8mA 驱动选择
0x50C	GPIONDR	R/W	0x0000.0000	GPIO 开漏选择
0x510	GPIONPUR	R/W	-	GPIO 上拉选择
0x514	GPIONPDR	R/W	0x0000.0000	GPIO 下拉选择
0x518	GPIONSLR	R/W	0x0000.0000	GPIO 斜率控制选择
0x51C	GPIONDEN	R/W	-	GPIO 数字使能
0x520	GPIONLOCK	R/W	0x0000.0001	GPIO 锁定
0x524	GPIONCR	-	-	GPIO 确认
0x528	GPIONAMSEL	R/W	0x0000.0000	GPIO 模拟模块选择
0x52C	GPIONPCTL	R/W	-	GPIONGPIO 端口控制
0xFD0	GPIONPeriphID4	RO	0x0000.0000	GPIO 外设标识 4
0xFD4	GPIONPeriphID5	RO	0x0000.0000	GPIO 外设标识 5
0xFD8	GPIONPeriphID6	RO	0x0000.0000	GPIO 外设标识 6
0xFDC	GPIONPeriphID7	RO	0x0000.0000	GPIO 外设标识 7
0xFE0	GPIONPeriphID0	RO	0x0000.0061	GPIO 外设标识 0
0xFE4	GPIONPeriphID1	RO	0x0000.0000	GPIO 外设标识 1
0xFE8	GPIONPeriphID2	RO	0x0000.0018	GPIO 外设标识 2
0xFEC	GPIONPeriphID3	RO	0x0000.0001	GPIO 外设标识 3
0xFF0	GPIONPCellID0	RO	0x0000.000D	GPIO PrimeCell 标识 0
0xFF4	GPIONPCellID1	RO	0x0000.00F0	GPIO PrimeCell 标识 1
0xFF8	GPIONPCellID2	RO	0x0000.0005	GPIO PrimeCell 标识 2
0xFFC	GPIONPCellID3	RO	0x0000.00B1	GPIO PrimeCell 标识 3

注：R/W表示软件可以读或写此域；RO表示软件可以读取此域，写入此域无效；W1C表示软件可以写此域，向W1C位写入0不影响寄存器中的位值，写入1 清零寄存器中位的值，剩余的位保持变，读寄存器返回的数据没有意义。

注意：GPIODATA寄存器是数据寄存器，在软件控制模式时，在通过将方向寄存器设置为输出的情况下，写入GPIODATA 寄存器的数据将会传送到GPIO 的引脚上。为了对GPIODATA 寄存器执行写操作，由地址总线位[9:2]

产生的相关屏蔽位必须为高电平。否则，该位的值不会被写操作改变。同样，从该寄存器读取的值由从访问数据寄存器的地址处获取的屏蔽位[9:2]的情况来决定。如果地址屏蔽位为1，那么读取GPIODATA 中相应位的值；如果地址屏蔽位为0，那么不管GPIODATA 中相应位的值是什么，都会将它们读作0。如果各自的管脚被配置成输出，那么读取GPIODATA 将返回最后写入的位值；或者当这些管脚被配置成输入时，将返回相应的输入管脚上的值。所有位在复位时都被清零。

这样，为了对端口进行有效的读写，GPIODATA地址的偏移量应取0x3FC。例如PortF的GPIODATA地址为0x400253FC或0x400553FC。

表中各寄存器每一位的含义可参阅《Stellaris® LM3S9B96 微控制器数据手册》。

1.3 GPIO 初始化和配置

要使用 GPIO 端口的引脚，必须通过给 RCGC2 寄存器(相应的位置位来使能该端口的时钟信号。复位时，所有的 GPIO 引脚都被配置为非驱动状态(三态)：GPIOAFSEL=0,GPIODEN=0,GPIOPDR=0, GPIOPUR=0。

GPIO 端口的各种配置方法如表 1.3 所示。

表 1.3

配 置	寄存器位的值									
	GPIOAFSEL	GPIODIR	GPIODR	GPIODEN	GPIOPUR	GPIOPDR	GPIODR2R	GPIODR4R	GPIODR8R	GPIOSLR
数字输入（GPIO）	0	0	0	1	?	?	X	X	X	X
数字输出（GPIO）	0	1	0	1	?	?	?	?	?	?
开漏输入（GPIO）	0	0	1	1	X	X	X	X	X	X
开漏输出（GPIO）	0	1	1	1	X	X	?	?	?	?
数字输入（定时器CCP）	1	X	0	1	?	?	X	X	X	X
数字输出（PWM）	1	X	0	1	?	?	?	?	?	?
数字输出（定时器PWM）	1	X	0	1	?	?	?	?	?	?
数字输入/输出（SSI）	1	X	0	1	?	?	?	?	?	?
数字输入/输出（UART）	1	X	0	1	?	?	?	?	?	?
模拟输入（比较器）	0	0	0	0	0	0	X	X	X	X
数字输出（比较器）	1	X	0	1	?	?	?	?	?	?

表格中X代表忽略，可为任意值（0/1）；? 代表是0或1由具体情况决定，取决于配置

举例：GPIO 端口的第 2 引脚配置为上升沿驱动，配置方法如下表：

表 1.4

寄存器	期望的中断事件触发	端口各位的值							
		7	6	5	4	3	2	1	0
GPIOIS	0=边沿；1=电平	X	X	X	X	X	0	X	X
GPIOIBE	0=单边沿；1=双边沿	X	X	X	X	X	0	X	X

GPIOIEV	0=低电平，或负边沿； 1=高电平，或正边沿	X	X	X	X	X	1	X	X
GPIOIM	0=屏蔽；1=不屏蔽	0	0	0	0	0	1	0	0

1.4 板上 GPIO 资源及特性

GPIO 模块支持高达 65 个可编程的输入/输出引脚。具体取决于正在使用的外设。整体结构框图如下：

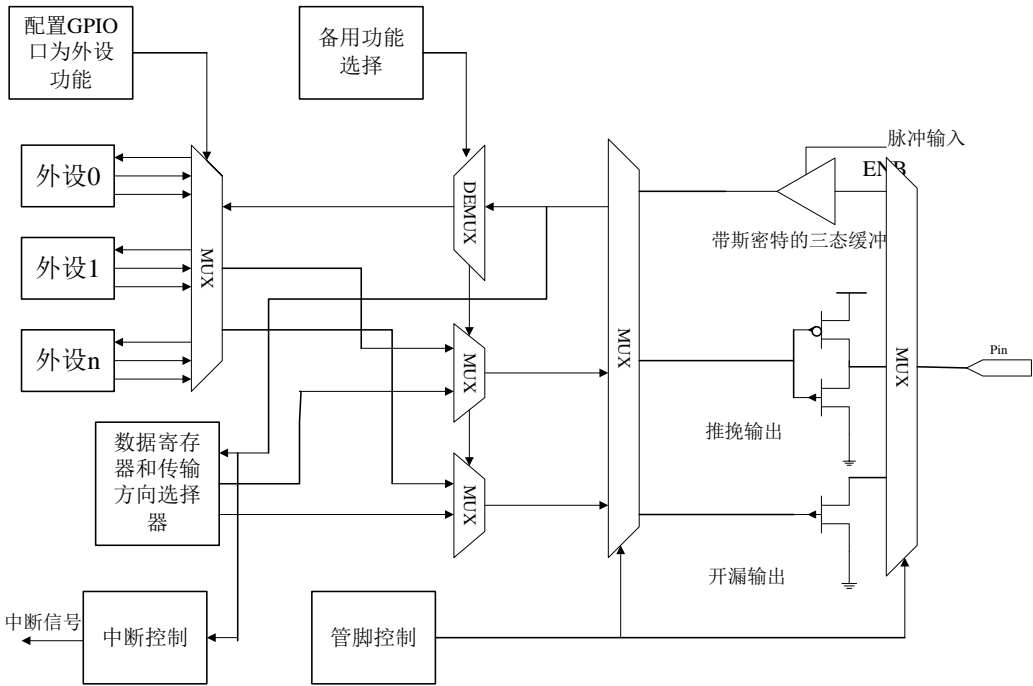


图 1.1 GPIO 整体示意图

本实验涉及部分电路图如下所示

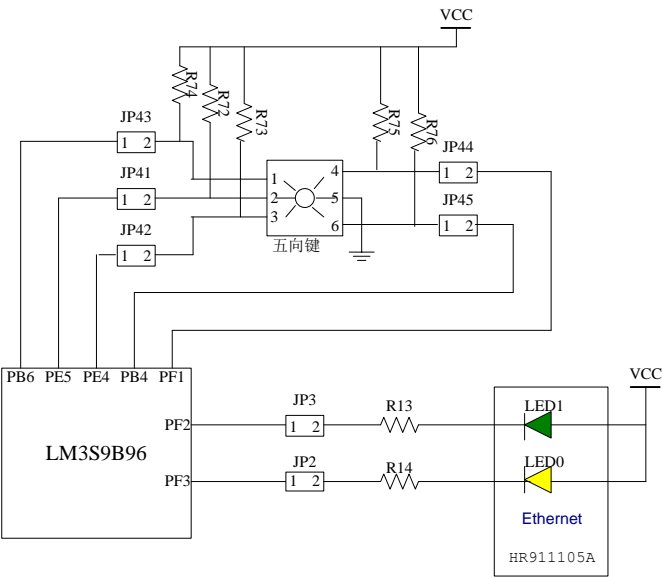


图 1.2 GPIO 实验涉及电路部分

跳线说明如下图:

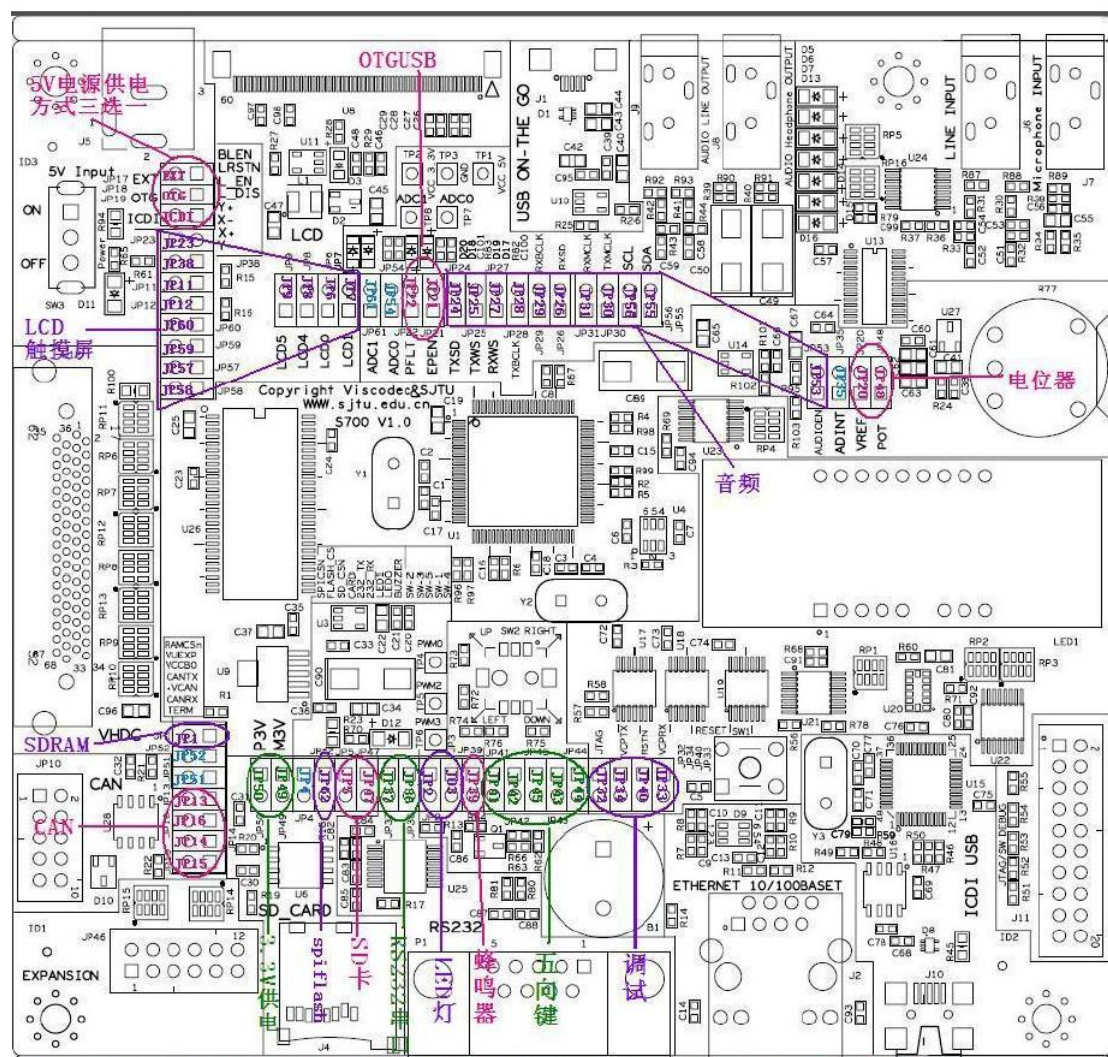


图 1.3 跳线说明

本实验使用的资源包括:

LED: 低电平有效

表 1.5

LED	对应端口
LED0(黄灯)	PF3
LED1(绿灯)	PF2

五向键：低电平触发

表 1.6

按键	对应端口	对应操作
SW_1	PB6	右
SW_2	PE5	按下
SW_3	PE4	上
SW_4	PF1	下
SW_5	PB4	左

1.5 Stellaris 库函数

作为高端嵌入式处理器，32 位 ARM 既可以用传统单片机的寄存器方式操作，也可以用类似操作系统驱动的方式，在拥有大量宏定义和封装函数的基础上作上层调用。

Tips: 两种操作方式的优劣明显，前者代码效率极高，但繁琐异常（发展到极限就是汇编语言）；后者效率一般，但编写者负担较轻，代码有很好的通用性、可重用性和可读性。在硬件技术飞速发展的今天，后者已渐渐成为了主流。

对于 LM3S9B96 处理器，TI 已配套有全套的 Stellaris 驱动库，可作为头文件加载与 Keil 的程序中。该驱动库不仅包含大量封装函数与高层宏定义，并且包括基本寄存器的宏定义。更为幸运的是，为配合 Keil 的烧写，其所有寄存器宏定义对用户都是透明的，并未深埋在 IDE 中，使用户可以更好地观察嵌入式可编程处理器的工作方式。

对寄存器读写，可参见 LM3S9B96 库函数介绍文件夹中的文件 LM3SLib_hw_types.pdf。

对于上层代码的编写，可参见 LM3S9B96 库函数介绍文件夹中的文件 LM3SLib_GPIO.pdf。

本实验涉及的库函数有：

表1.7 函数SysCtlPeripheralEnable

原型	void SysCtlPeripheralEnable(unsigned long ulPeripheral)
参数	ulPeripheral 是要使能的外设。
返回	None.
说明	<p>此函数使能外设。上电时全部的外设都被禁止；为了使外设能工作或响应寄存器的读/写操作，它们必须被使能。 ulPeripheral 参数必须取下面的其中一个值：</p> <p>SYSCTL_PERIPH_ADC0, SYSCTL_PERIPH_ADC1, SYSCTL_PERIPH_CAN0, SYSCTL_PERIPH_CAN1, SYSCTL_PERIPH_CAN2, SYSCTL_PERIPH_COMP0, SYSCTL_PERIPH_COMP1, SYSCTL_PERIPH_COMP2, SYSCTL_PERIPH_EPI0, SYSCTL_PERIPH_ETH, SYSCTL_PERIPH_GPIOA, SYSCTL_PERIPH_GPIOB, SYSCTL_PERIPH_GPIOC, SYSCTL_PERIPH_GPIOD, SYSCTL_PERIPH_GPIOE, SYSCTL_PERIPH_GPIOF, SYSCTL_PERIPH_GPIOG, SYSCTL_PERIPH_GPIOH, SYSCTL_PERIPH_GPIOJ, SYSCTL_PERIPH_HIBERNATE, SYSCTL_PERIPH_I2C0, SYSCTL_PERIPH_I2C1, SYSCTL_PERIPH_I2S0, SYSCTL_PERIPH_PWM, SYSCTL_PERIPH_QEI0, SYSCTL_PERIPH_QEI1, SYSCTL_PERIPH_SSI0, SYSCTL_PERIPH_SSI1, SYSCTL_PERIPH_TIMER0, SYSCTL_PERIPH_TIMER1, SYSCTL_PERIPH_TIMER2, SYSCTL_PERIPH_TIMER3, SYSCTL_PERIPH_TEMP, SYSCTL_PERIPH_UART0, SYSCTL_PERIPH_UART1, SYSCTL_PERIPH_UART2, SYSCTL_PERIPH_UDMA, SYSCTL_PERIPH_USB0, SYSCTL_PERIPH_WDOG0, or SYSCTL_PERIPH_WDOG1</p>
功能	使能一个外设

表1.8 函数GPIODirModeSet

原型	void GPIODirModeSet(unsigned long ulPort,unsigned char ucPins,unsigned long ulPinIO)
参数	<p>ulPort是GPIO端口的基址。</p> <p>ucPins是管脚的位组合（bit-packed）。</p> <p>ulPinIO是管脚方向“与/或”模式。</p>
返回	None.
说明	<p>这个函数在软件控制下将所选GPIO端口的指定管脚设置成输入或输出，或者，也可以将管脚设置成由硬件来控制。</p> <p>参数ulPinIO是一个枚举数据类型，它可以是下面的其中一个值：</p> <p>GPIO_DIR_MODE_IN; GPIO_DIR_MODE_OUT; GPIO_DIR_MODE_HW。</p> <p>在上面的值中，GPIO_DIR_MODE_IN表明管脚将被编程用作一个软件控制的输入，GPIO_DIR_MODE_OUT表明管脚将被编程用作一个软件控制的输出，GPIO_DIR_MODE_HW表明管脚将被设置成由硬件进行控制。</p> <p>管脚用一个位组合（bit-packed）的字节来指定，这里的每个字节，置位的位用来识别被访问的管脚，字节的位0代表GPIO端口管脚0、位1代表GPIO端口管脚1等等。</p>
功能	设置指定管脚的方向和模式。

表1.9 函数GPIOPadConfigSet

原型	void GPIOPadConfigSet(unsigned long ulPort, unsigned char ucPins, unsigned long ulStrength, unsigned long ulPinType)
参数	<p>ulPort是GPIO端口的基址。</p> <p>ucPins是特定管脚的位组合（bit-packed）表示。</p> <p>ulStrength指定输出驱动强度。</p> <p>ulPinType指定管脚类型。</p>
返回	None.
说明	<p>这个函数设置所选GPIO端口指定管脚的驱动强度和类型。对于配置用作输入端口的管脚，端口按照要求配置，但是对输入唯一真正的影响是上拉或下拉终端的配置。</p> <p>参数ulStrength可以是下面的一个值：</p> <p>GPIO_STRENGTH_2MA; GPIO_STRENGTH_4MA; GPIO_STRENGTH_8MA; GPIO_STRENGTH_8MA_SC。</p> <p>在上面的值中，GPIO_STRENGTH_xMA指示2、4或8mA的输出驱动强度；而GPIO_OUT_STRENGTH_8MA_SC指定了带斜率控制（slew control）的8mA输出驱动。</p> <p>参数ulPinType可以是下面的其中一个值：</p>

	<p>GPIO_PIN_TYPE_STD; GPIO_PIN_TYPE_STD_WPU; GPIO_PIN_TYPE_STD_WPD; GPIO_PIN_TYPE_OD; GPIO_PIN_TYPE_OD_WPU; GPIO_PIN_TYPE_OD_WPD; GPIO_PIN_TYPE_ANALOG。</p> <p>在上面的值中，GPIO_PIN_TYPE_STD* 指定一个推挽管脚，GPIO_PIN_TYPE_OD*指定一个开漏管脚，*_WPU指定一个弱上拉，*_WPD指定一个弱下拉，GPIO_PIN_TYPE_ANALOG指定一个模拟输入（对于比较器来说）。</p> <p>管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位0代表GPIO端口管脚0、位1代表GPIO端口管脚1等等。</p>
功能	设置指定管脚的配置。

表1.10 函数GPIOPinTypeGPIOOutput

原型	void GPIOPinTypeGPIOOutput(unsigned long ulPort, unsigned char ucPins)
参数	<p>ulPort是GPIO端口的基址。</p> <p>ucPins是管脚的位组合（bit-packed）表示。</p>
返回	None.
说明	<p>GPIO管脚必须正确配置，以便作为GPIO输出能正常工作。这一点，特别是对于Furry-class器件来说是很重要的，在Furry-class器件中，数字输入使能在默认状态下是关闭的。这个这个函数为用作GPIO管脚提供了正确的配置。管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位0代表GPIO端口管脚0、位1代表GPIO端口管脚1等等。</p>
功能	配置管脚用作GPIO输出。

表1.11 函数GPIOPinTypeGPIOInput

原型	void GPIOPinTypeGPIOInput(unsigned long ulPort, unsigned char ucPins)
参数	<p>ulPort是GPIO端口的基址。</p> <p>ucPins是管脚的位组合（bit-packed）表示。</p>
返回	None.
说明	<p>GPIO管脚必须正确配置，以便GPIO输入能正常工作。这一点，特别是对于Furry-class器件来说是很重要的，在Furry-class器件中，数字输入使能在默认状态下是关闭的。这个这个函数为用作GPIO管脚提供了正确的配置。管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位0代表GPIO端口管脚0、位1代表GPIO端口管脚1等等。</p>
功能	配置管脚用作GPIO输入。

表1.12 函数GPIOPinWrite

原型	void GPIOPinWrite(unsigned long ulPort, unsigned char ucPins, unsigned char ucVal)
参数	ulPort是GPIO端口的基址。 ucPins是管脚的位组合（bit-packed）表示。 ucVal是写入到指定管脚的值。
返回	None.
说明	将对应的位值写入ucPins指定的输出管脚。向配置用作输入的管脚写入一个值不会产生任何影响。管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位0代表GPIO端口管脚0、位1代表GPIO端口管脚1等等。
功能	向指定管脚写入一个值。

表1.13 函数GPIOPinRead

原型	long GPIOPinRead(unsigned long ulPort, unsigned char ucPins)
参数	ulPort是GPIO端口的基址。 ucPins是管脚的位组合（bit-packed）表示。
返回	None.
说明	读取指定管脚（由ucPins指定的）的值。输入和输出管脚的值都能返回，ucPins未指定的管脚的值被设置成0。管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位0代表GPIO端口管脚0、位1代表GPIO端口管脚1等等。
功能	读取指定管脚上出现的值。

1.6 实验一 GPIO 操作编程

■ 实验概述

本实验通过多种方法来控制GPIO端口的读写，通过GPIO端口的读写来控制J2网络接口上的两个独立LED灯（LED0黄色JP2，LED1绿色JP3）的点亮和熄灭。

前三个实验分别用了三种方法来控制GPIO端口，以此控制LED灯的亮灭。最后一个实验，通过GPIO端口读入五向键的状态，并以此来控制LED灯的亮灭。

本次实验重点在于，熟悉ARM的集成开发环境，理解GPIO的工作原理，掌握与GPIO有关的各寄存器的作用和设置方法。主要涉及以下知识点：

- 通过寄存器操作控制GPIO
- 使用Hw_types库中的函数读写寄存器
- 学习使用Luminary库函数及其中GPIO部分
- GPIO的Input和Output

实验1.1：寄存器直接操作GPIO实验

■ 本实验流程图

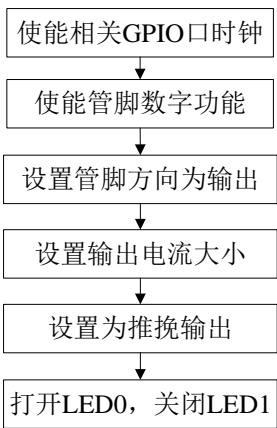


图 1.4 程序流程图

■ Step1: 准备

创建一个新的工程文件，添加 main.c 文件。为了后续实验的方便，工程所在文件夹建议命名为“GPIO_Output(Register)”。

■ Step2: 添加声明

在 main.c 中加入声明：

```
//寄存器地址定义（参见 ARM 的 DATASHEET:SysCtl 模块和 GPIO 模块）
#define RCGC2      (*((volatile unsigned long *) (0x400FE108))) //各模块时钟门控控制寄
//寄存器 *注意不是
//DATASHEET 上 的
0x000
#define GPIOFDATA  (*((volatile unsigned long *) (0x400253fc))) //PortF 数据寄存器
#define GPIOFDIR   (*((volatile unsigned long *) (0x40025400))) //PortF 方向寄存器
#define GPIOFAFSEL  (*((volatile unsigned long *) (0x40025420))) //Port 备用功能选择寄
寄存器
#define GPIOFDR2R   (*((volatile unsigned long *) (0x40025500))) //PortF 2-mA 驱动寄存
器
#define GPIOFDR4R   (*((volatile unsigned long *) (0x40025504))) //PortF 4-mA 驱动寄存
器
#define GPIOFDR8R   (*((volatile unsigned long *) (0x40025508))) //PortF 8-mA 驱动寄存
器
#define GPIOFODR    (*((volatile unsigned long *) (0x4002550c))) //PortF 开漏选择寄存器
#define GPIOFDEN    (*((volatile unsigned long *) (0x4002551c))) //PortF 数字使能寄存器
#define GPIOFSLR    (*((volatile unsigned long *) (0x40025518))) //PortF 斜率寄存器
```

■ Step3: 创建主函数

在声明的下方创建 main 函数

```
int main(void)
```

```
{  
return 0;  
}
```

■ Step4: 添加 GPIO 初始化设置

在 main 函数内加入如下代码：

```
RCGC2=0x00000020;    //使能 PORTF，需等待几个时钟周期  
__nop();              //asm 空语句  
__nop();  
  
GPIOFDEN=0x0000000c; //使能 PORTF 的数字模块  
GPIOFDIR=0x0000000c; //设为输出模式  
GPIOFAFSEL=0x00000000; //关闭备用功能  
GPIOFDR8R=0x0000000c; //设置输出电流为 8-mA（2-mA 和 4-mA 的寄存器自动  
清零）  
GPIOFODR=0x00000000; //设为推挽输出(可以不设置，默认)  
GPIOFSLR=0x0000000c; //打开斜率控制（跳变速度控制）
```

■ Step5: 添加 GPIO 控制代码

主函数（main）中添加 GPIO 控制的代码，打开 LED0，关闭 LED1：

```
GPIOFDATA=0x00000004; //打开 LED0，关闭 LED1  
while(1);
```

■ Step6: 添加库函数

新建一个 Group，命名为 Libraries，在 Libraries Group 中加入实验涉及到的 Stellaris Ware for C1/driverlib/rvmdk/driverlib.lib 文件。

■ Step 7: 编译和下载

■ Step 8: 查看实验效果

网口上两个 LED 等，LED0 亮，LED1 灭

实验1.2：底层驱动库操作GPIO实验

■ 本实验流程图

见图 1.4。

■ Step1: 准备

复制上一个工程所在文件夹：“GPIO_Output(Register)”，重命名为：“GPIO_Output(Hardwaretypes)”。打开工程文件.uvproj。将 main 函数内容修改为：

```
int main(void)
```

```
{
while(1);
return 0;
}
```

■ Step2: 添加 GPIO 初始化设置

在 while（1）之前添加如下代码：

```
HWREG(RCGC2)=0x00000020;    //使能 PORTF，需等待几个时钟周期
__nop();                    //asm 空语句
__nop();

HWREG(GPIOFDEN)=0x0000000c;    //使能 PORTF 的数字模块
HWREG(GPIOFDIR)=0x000c;        //设为输出模式
HWREGH(GPIOFAFSEL)=0x0000;    //关闭备用功能
HWREGH(GPIOFDR8R)=0x000c;    //设置输出电流为 8-mA（2-mA 和 4-mA
的寄存器自动清零）
HWREGB(GPIOFODR)=0x00;        //设为推挽输出(可以不设置，默认)
HWREGB(GPIOFSLR)=0x0c;        //打开斜率控制（跳变速度控制）
```

可以发现，相对于上一个实验，这里避免了寄存器的直接操作，而是使用了 *HWREG*，*HWREGH*，*HWREGB* 三个库来进行操作。

■ Step3: 添加 GPIO 控制代码

在 GPIO 初始化之后，在 main 函数中添加 GPIO 控制代码，打开 LED0，关闭 LED1：

```
HWREGB(GPIOFDATA)=0x04;    //打开 LED0，关闭 LED1
```

■ Step 4: 编译和下载

■ Step 5: 查看实验效果

网口上两个 LED 灯，LED0 亮，LED1 灭

实验 1.3: Luminary 驱动库操作 GPIO 实验

■ 本实验流程图

见图 1.4。

■ Step1: 准备

复制上一个工程文件夹，将原有的 main.c 文件内容全部删除。修改 main.c 文件。为了后续实验的方便，新的工程所在文件夹建议命名为“GPIO_Output(LuminaryLibrary)”。

■ Step2: 包含库函数相关的头文件

```
#include "inc/hw_memmap.h"    //基址宏定义
#include "inc/hw_types.h"      //数据类型宏定义，寄存器访问函数
#include "driverlib/debug.h"    //调试用
#include "driverlib/gpio.h"     //通用 IO 口宏定义
#include "driverlib/sysctl.h"   //系统控制宏定义
```

■ Step3: 创建主函数

在声明的下方创建 main 函数

```
int main(void)
{
while(1);
return 0;
}
```

■ Step4: 添加 GPIO 初始化设置

在 main 函数中添加如下设置，使能端口 F:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF); //使能端口（无需空等待语句）
```

在使能后，进一步设置端口为输出:

```
GPIODirModeSet(GPIO_PORTF_BASE,           //设置为输出
                GPIO_PIN_2|GPIO_PIN_3,
                GPIO_DIR_MODE_OUT);
```

进一步设置为 8mA、带转换速率控制的推挽输出:

```
GPIOPadConfigSet(GPIO_PORTF_BASE,
                  GPIO_PIN_2|GPIO_PIN_3,
                  GPIO_STRENGTH_8MA_SC,
                  GPIO_PIN_TYPE_STD);
```

■ Step5: 添加 GPIO 控制代码

```
GPIOPinWrite(GPIO_PORTF_BASE,           //写入端口，LED0,LED0 齐亮
              GPIO_PIN_2|GPIO_PIN_3, 0x00);
```

■ Step 6: 编译和下载

■ Step 7: 查看实验效果

网口上两个 LED 灯全亮。

实验 1.4: 驱动库操作 GPIO 综合实验

■ 本实验流程图

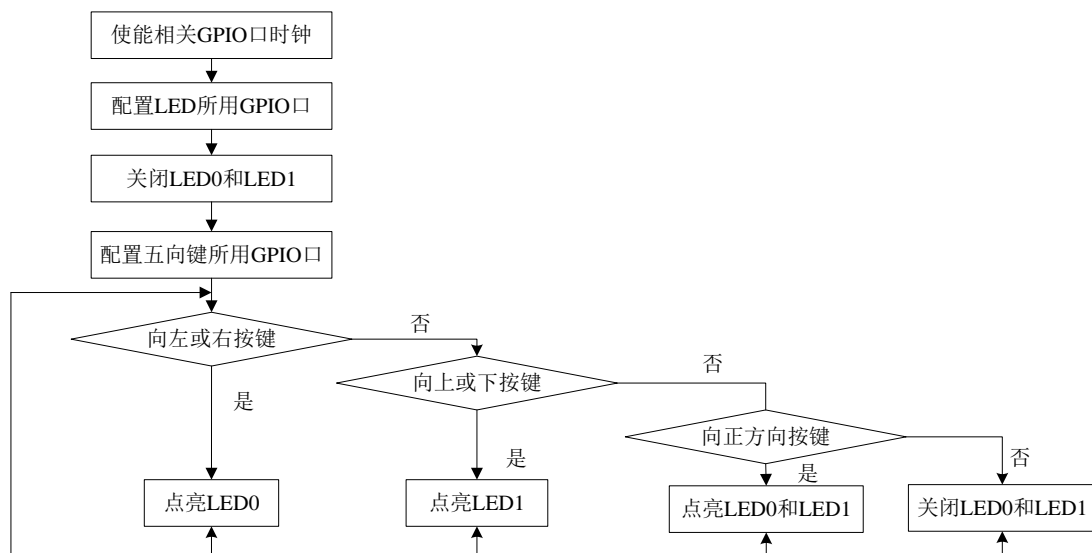


图 1-12 程序流程图

■ Step1: 准备

创建一个新的工程文件，添加 main.c 文件。为了后续实验的方便，工程所在文件夹建议命名为“GPIO_Output(LuminaryLibrary)”。

■ Step2: 包含库函数相关的头文件

```

#include "inc/hw_memmap.h"    //基址宏定义
#include "inc/hw_types.h"     //数据类型宏定义，寄存器访问函数
#include "driverlib/debug.h"   //调试用
#include "driverlib/gpio.h"    //通用 IO 口宏定义
#include "driverlib/sysctl.h"  //系统控制宏定义
  
```

■ Step3: 创建主函数

在声明的下方创建 main 函数：

```

int main(void)
{
    return 0;
}
  
```

■ Step4: 添加 GPIO 初始化设置

在 main 函数中添加如下设置，使能端口 B、E、F，并设置相关端口为输入或输出：

```

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);    //使能端口 B
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);    //使能端口 E
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);    //使能端口 F
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,         //PF2,PF3 设置为 2mA 推挽输出
                       GPIO_PIN_2|GPIO_PIN_3);
GPIOPinWrite(GPIO_PORTF_BASE, //写入 PF2,PF3, LED0,LED0 齐灭
  
```

```

        GPIO_PIN_2|GPIO_PIN_3,
        0x0c);
    GPIOPinTypeGPIOInput(GPIO_PORTB_BASE, //设置 PB6,PE5,PE4,PF1,PB4 为输入
        GPIO_PIN_4|GPIO_PIN_6);
    GPIOPinTypeGPIOInput(GPIO_PORTC_BASE,
        GPIO_PIN_4|GPIO_PIN_5);
    GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, GPIO_PIN_1);

```

■ Step5: 加入程序的主循环部分

下面的主循环主要功能是：读入五向键的按键状态，并通过 LED 灯显示出来。

```

while(1)
{
    if (((GPIOPinRead(GPIO_PORTB_BASE,GPIO_PIN_6)>>2) &
        GPIOPinRead(GPIO_PORTB_BASE,GPIO_PIN_4))==0)
        //左或右
        GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2|GPIO_PIN_3,0x04);
        //LED0 亮
    else
    {
        if (((GPIOPinRead(GPIO_PORTC_BASE,GPIO_PIN_4)>>3) &
            GPIOPinRead(GPIO_PORTF_BASE,GPIO_PIN_1))== 0)
            //上或下
            GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2|GPIO_PIN_3,0x08);
            //LED1 亮
        else
        {
            if ((GPIOPinRead(GPIO_PORTC_BASE,GPIO_PIN_5)>>5)==0)
                //按下
                GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2|GPIO_PIN_3,0x00);
                //LED0 及 LED1 亮
            else
                GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_2|GPIO_PIN_3,0x0c);
                //关闭 LED0 和 LED1
        }
    }
}

```

■ Step 6: 编译和下载

■ Step 7: 查看实验效果

网口上两个 LED 灯。拨动五向键，查看实验效果。

1.7 思考与练习题

1.1 在实验 1.1 中采用了 APB 访问方式还是 AHB 访问方式？请采用另一种访问方式改写程序。

1.2 阅读头文件 `hw_types.h`、`hw_memmap.h`、`hw_sysctl.h`、`hw_gpio.h`，了解它们的作用。函数 `HWREG()` 的作用是什么？

1.3 在实验 1.3 中，如要求配置 GPIO 引脚的输出为 4mA，LED0 点亮，LED1 熄灭，试修改、调试程序。

1.4 在实验 1.4 中，每按一次五向键，LED 灯产生相应的反应，试修改程序，达到如下功能：每连续按两次五向键，LED 灯产生相应的反应。

第二章 系统控制单元：时钟控制模块

2.1 概述

LM3S9B96 微控制器的系统控制单元主要提供如下系统控制功能：

- 器件标识：有些只读寄存器可以向软件提供关于微控制器的信息，比如版本、器件编号、SRAM 大小、Flash 存储器大小以及其它特性。这些只读寄存器包括 DID0 寄存器、DID1 寄存器、DC0-DC9 寄存器和 NVMSTAT 寄存器等，各寄存器的含义可参阅《Stellaris® LM3S9B96 微控制器数据手册》。
- 本机控制，如复位控制、功率控制和时钟控制等。
- 系统控制(运行模式、睡眠模式和深度睡眠模式)。
- Cortex-M3 处理器的内核级外设 in Stellaris®系列微控制器中的实现，包括 SysTick、NVIC、SCB 以及 MPU 等。

本章主要介绍系统控制单元中的功率控制和时钟控制模块。

2.2 功率控制

Stellaris® 微控制器提供了一个集成的LDO调节器，它用来对大多数微控制器的内部逻辑提供电源。为减小功耗，可以用一个不可编程的LDO来将微控制器的3.3V输入电压变为1.2V。电压输出最小为1.08V，最大为1.35V。该LDO可提供高达60mA的电流。电源结构如图2.1所示。

LDO是“Low Drop-Out”的缩写，是一种线性直流电源稳压器。LDO的显著特点是输入与输出之间的低压差，能达到数百毫伏，而传统线性稳压器一般在1.5V以上。这种低压差特性具有降低功耗、缩小体积等优点。

Stellaris系列ARM内部集成一个内部的LDO稳压器，为处理器内核及片内外设提供稳定的电源。LDO输出电压默认值是2.50V，可以根据应用程序在2.25 V到2.75V之间调整。降低LDO输出电压可以节省功耗。

片内LDO输入电压是芯片电源VDDA（额定3.3V），LDO输出到一个名为“LDO”的管脚。

注意：在LDO管脚和GND之间必须接一个1~3.0μF的瓷片电容，推荐值是2.2μF。在启用片内锁相环PLL(见本章2.3节)之前，必须要将LDO电压设置在最高的2.75V，否则可能造成系统工作异常。

2.3 时钟控制

如图 2.2 所示，为主时钟逻辑框图。外设由系统时钟信号驱动并且可以独立使能/禁止。ADC 时钟信号为了进行合适的 ADC 操作自动分频到 16MHz。PWM 时钟信号是系统时钟的同步分频，这样可以向 PWM 电路提供更宽的范围（通过设置 RCC 中的 PWMDIV）。

振荡器包括主振荡器、精确内部振荡器、内部 30KHz 振荡器和外部休眠振荡器。

主振荡器MOSC可以连接一个1~16MHz的外部晶体。典型接法如图2.3所

示，电阻R1可以加速起振过程，C1和C2取值要适当，不宜过大或过小。如果不使用晶体，则外部的有源振荡信号也可以从OSC0管脚输入，要求信号幅度介于0~3.3V之间，此时OSC1管脚应当悬空。

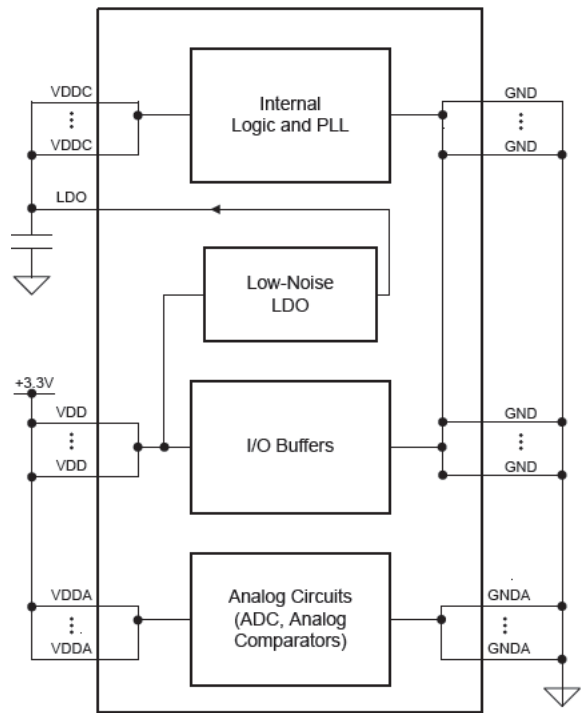


图 2.1 电源结构

在微控制器中有四个时钟源可以使用。表 2.3 列出了各时钟源的选项。

Stellaris系列ARM集成有两个内部振荡器，一个是12MHz高速精确内部振荡器IOSC，一个是30kHz低速振荡器INT30。内部振荡器误差较大，约 $\pm 30\%$ ，这是由于IC制造工艺的特点形成的，因此对时钟精度有要求严格的场合不适宜采用内部振荡器。IOSC经4分频后（IOSC/4）标称为3MHz，也可以作为系统时钟的一个来源。芯片在较低的时钟速率下运行能够明显节省功耗。

表 2.3 时钟源选项

时钟源	驱动 PLL?		用作 SysClk?	
精确内部振荡器	YES	BYPASS=0 OSCSRC=0	YES	BYPASS=1 OSCSRC=0x1
4 分频的精确内部振荡器 (4MHz $\pm 10\%$)	NO	BYPASS=1	YES	BYPASS=1 OSCSRC=0x2
主振荡器	YES	BYPASS=0 OSCSRC=0	YES	BYPASS=1 OSCSRC=0
内部 30KHz 振荡器	NO	BYPASS=1	YES	BYPASS=1 OSCSRC=0x3

通过PLL（Phase Locked Loop，锁相环）可以提高振荡器的工作频率。Stellaris系列ARM内部集成有一个PLL。PLL输出频率固定为400MHz，误差 $\pm 1\%$ 。如果选用PLL，则MOSC频率必须在3.579545~8.192MHz之间才能使PLL精确地输出400MHz。

经OSC或PLL产生的时钟可以经过1~64分频后得到系统时钟（System Clock），分频数越大越省电。

注意：由于Cortex-M3内核最高运行频率为50MHz，因此如果要使用PLL，则至少要进行4以上的分频（硬件会自动阻止错误的软件配置）。启用PLL后，系统功耗将明显增大。

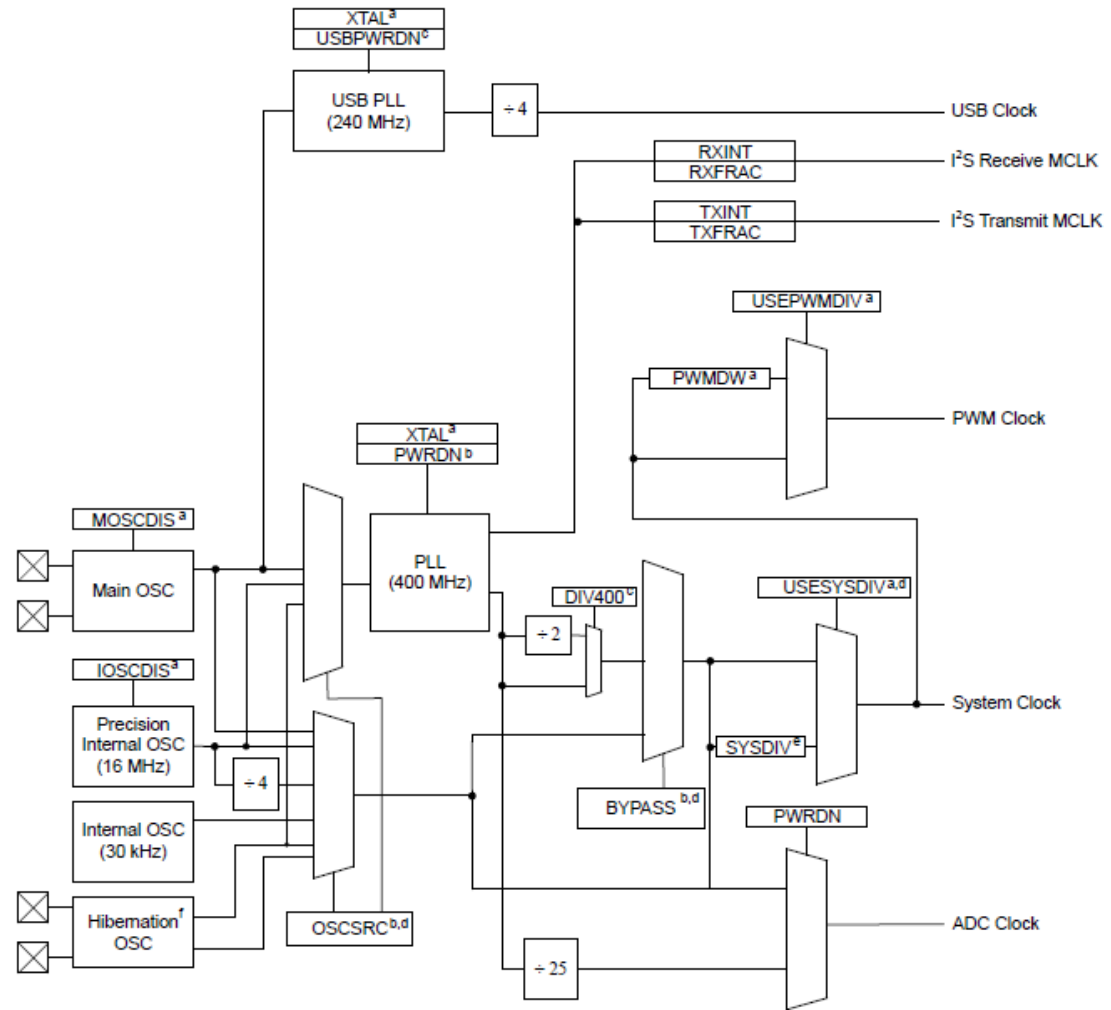


图 2.2 主时钟逻辑

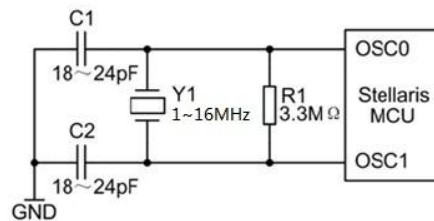


图 2.3 MOSC 外接晶振典型用法

2.4 Stellaris 库函数

驱动库函数 SysCtlLDOSet() 用来设置 LDO 的输出电压，在需要节省功耗时可以调得低一些，在耗电较大的应用场合要调得高一些，在启用 PLL 之前必须设置在最高的 2.75V。函数 SysCtlLDOGet() 用来获取 LDO 当前的输出电压值。详细参数参见表 2.1 和表 2.2。

表 2.1 函数 SysCtlLDOSet()

原型	void SysCtlLDOSet(unsigned long ulVoltage)
参数	ulVoltage 是所需的LDO的输出电压，它必须是下面的其中一个值： SYSCTL_LDO_2_25V, SYSCTL_LDO_2_30V, SYSCTL_LDO_2_35V, SYSCTL_LDO_2_40V, SYSCTL_LDO_2_45V, SYSCTL_LDO_2_50V, SYSCTL_LDO_2_55V, SYSCTL_LDO_2_60V, SYSCTL_LDO_2_65V, SYSCTL_LDO_2_70V, SYSCTL_LDO_2_75V.
返回	None
说明	这个函数设置 LDO 的输出电压。默认的电压是 2.5V；它可以在+/-10 % 范围内调整。
功能	设置 LDO 的输出电压

表2.2 函数SysCtlLDOGet()

原型	unsigned long SysCtlLDOGet(void)
参数	None
返回	返回 LDO 的当前电压；返回的是下面的其中一个值： SYSCTL_LDO_2_25V, SYSCTL_LDO_2_30V, SYSCTL_LDO_2_35V, SYSCTL_LDO_2_40V, SYSCTL_LDO_2_45V, SYSCTL_LDO_2_50V, SYSCTL_LDO_2_55V, SYSCTL_LDO_2_60V, SYSCTL_LDO_2_65V, SYSCTL_LDO_2_70V, SYSCTL_LDO_2_75V.
说明	这个函数决定了 LDO 的输出电压，就如控制寄存器指定的一样。
功能	获取 LDO 当前的输出电压值

驱动库函数SysCtlClockSet()可以用来完成系统时钟功能的设置。详细参数参见表2.4。函数SysCtlClockGet()用来获取已设置的系统时钟频率。

表2.4 函数SysCtlClockSet()

功能	系统时钟设置
原型	void SysCtlClockSet(unsigned long ulConfig)

参数	ulConfig is the required configuration of the device clocking: <ul style="list-style-type: none"> 系统时钟分频值 SYSCTL_SYSDIV_1 // 振荡器不分频（不可用于PLL） SYSCTL_SYSDIV_2 // 振荡器 2分频（不可用于PLL） SYSCTL_SYSDIV_3 // 振荡器 3分频（不可用于PLL） SYSCTL_SYSDIV_4 // 振荡器 4分频，或对PLL的分频结果为50MHz SYSCTL_SYSDIV_64 // 振荡器64分频，或对PLL的分频结果为3.125MHz 使用OSC还是PLL SYSCTL_USE_PLL // 采用锁相环PLL作为系统时钟源 SYSCTL_USE_OSC // 采用OSC（主振荡器或内部振荡器）作为系统时钟源 注：由于启用PLL时会消耗较大的功率，因此在启用PLL之前，要求必须先将LDO电压设置在2.75V，否则可能造成芯片工作不稳定。 OSC时钟源选择 SYSCTL_OSC_MAIN // 主振荡器作为OSC SYSCTL_OSC_INT // 内部12MHz振荡器作为OSC SYSCTL_OSC_INT4 // 内部12MHz振荡器4分频后作为OSC SYSCTL_OSC_INT30 // 内部30kHz振荡器作为OSC SYSCTL_OSC_EXT32 // 外接32.768kHz有源振荡器作为OSC 外接晶体频率 SYSCTL_XTAL_1MHZ // 外接晶体 1MHz SYSCTL_XTAL_1_84MHZ // 外接晶体 1.8432MHz SYSCTL_XTAL_16MHZ // 外接晶体 5.12MHz 	
	返回	None
示例	<pre> SysCtlClockSet(SYSCTL_SYSDIV_1 //振荡器不分频 SYSCTL_USE_OSC //采用OSC（主振荡器或内部振荡器） SYSCTL_OSC_MAIN // 主振荡器作为OSC SYSCTL_XTAL_16MHZ); // 采用16MHz晶振作为系统时钟 </pre>	

驱动库函数SysCtlDelay()提供一个产生一个固定长度延时的方法。参见表2.5的描述。它是用内嵌汇编语言的方式来编写的，可以在使用不同软件开发工具情况下而让程序的延时保持一致，具有较好的可移植性。

表2.5 函数SysCtlDelay()

功能	延时
原型	void SysCtlDelay(unsigned long ulCount)
参数	ulCount is the number of delay loop iterations to perform.
返回	None
示例	<pre> SysCtlDelay(20); // 延时60个系统时钟周期 SysCtlDelay(150 * (SysCtlClockGet() / 3000)); // 延时150ms </pre>

对于系统控制其他功能的详细介绍，参见 LM3S9B96 库函数介绍文件夹中的 LM3SLib_SysCtl.pdf。

2.5 自编封装函数介绍

表2.6 函数GPIOInitial()

原型	void GPIOInitial(void)
参数	Void
返回	Void
说明	None
功能	GPIO 初始化

表2.7 函数LED_On()

原型	void LED_On(unsigned char led_num)
参数	unsigned char led_num
返回	Void
说明	None
功能	打开 LED

表2.8 函数LED_Off()

原型	void LED_Off(unsigned char led_num)
参数	unsigned char led_num
返回	Void
说明	None
功能	关闭 LED

表 2.9 函数 LED_Overturn()

原型	void LED_Overturn (unsigned char led_num)
参数	unsigned char led_num
返回	Void
说明	关闭原来亮的 LED，打开原来俺的 LED
功能	翻转 LED

表 2.10 函数 KeyPress ()

原型	unsigned char KeyPress(unsigned char key_num)
参数	unsigned char key_num
返回	unsigned char
说明	None
功能	控制五向键

2.6 实验二 时钟控制编程

■ 实验概述：

本实验通过控制五向键将黄色网络灯(LED0)和绿色网络灯(LED1)轮流点亮，每次点亮时长为0.5秒； 放开按键在一个闪烁周期结束后停止闪烁。

本次实验学习目的：了解SysCtl模块的功能；学会根据具体实验板设置时钟频率；学会使用SysCtl库函数中的延时函数；学会将必须的初始函数或常用函数封装，自制延时函数。

■ 本实验用到的电路图如下

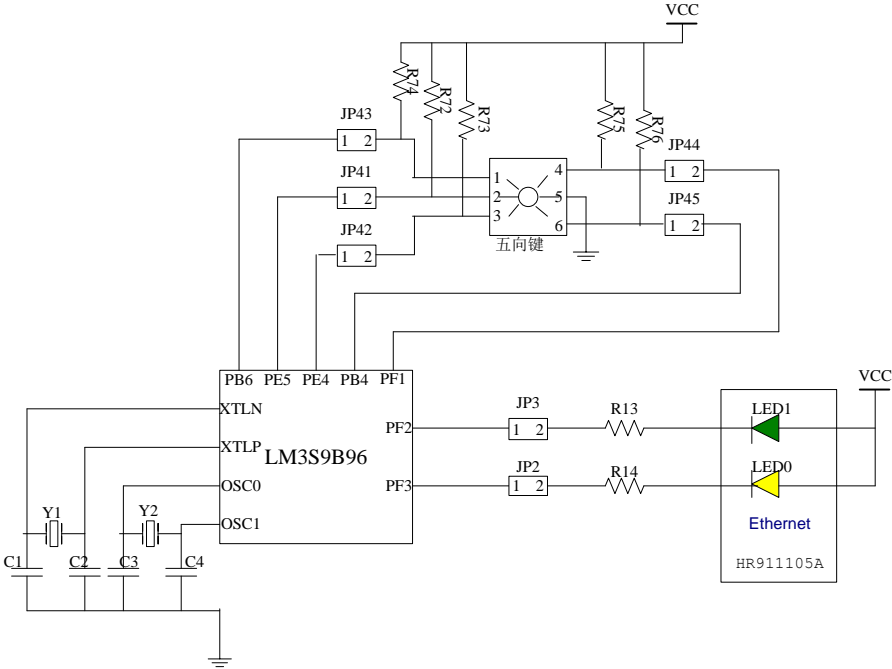


图 2.4 本实验电路图

■ 实验流程图

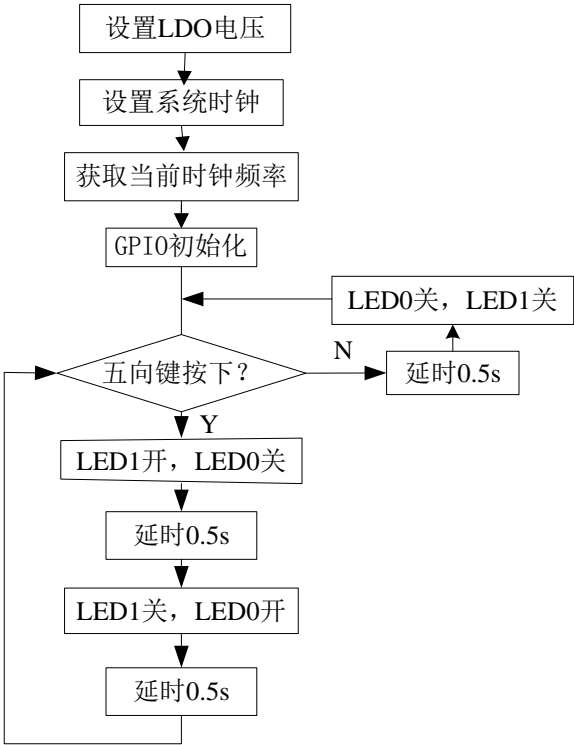


图 2-5 实验流程图

■ Step1: 准备

检查跳帽连接是否正确，如表 2.11 所示：

表 2.11 跳帽连接及对应端口

连接跳帽	对应端口	断开跳帽	对应按键
JP3	PF2	JP35	LED1
JP2	PF3	JP39	LED0
JP43	PB6	JP20,JP28	SW_1
JP41	PE5	JP24	SW2-PRESS
JP42	PE4	JP25	SW3-UP
JP44	PF1	JP30	SW4-DOWN
JP45	PB4	JP48	SW5-LEFT

在 StellarisWare for C1\codes\LM3S9B96_Experiment 下创建一个名为 SysCtl_OscMain 的文件夹，建立 SysCtl_OscMain 工程。

■ Step2: 定义 GPIODriverConfigure.h 文件

创建空白文档保存至 StellarisWare for C1/codes/LM3S9B96_Experiment /SysCtl_OscMain /下，命名为 GPIODriverConfigure.h，然后输入代码。

```
#ifndef __GPIODRIVERCONFIGURE_H__
#define __GPIODRIVERCONFIGURE_H__
extern void GPIOInitial(void);
extern void LED_On(unsigned char); //0-led0 1-led1 2-all
extern void LED_Off(unsigned char); //0-led0 1-led1 2-all
extern void LED_Overturn(unsigned char); //0-led0 1-led1 2-all
extern unsigned char KeyPress(unsigned char); //0-Press 1-Left 2-Right 3-Up 4-Down
#endif
```

■ Step3: GPIO 函数封装

建立一个空白文档以键入 GPIO 驱动程序，文档保存至 StellarisWare for C1/codes/LM3S9B96_Experiment/SysCtl_OscMain/ 下，命名为 GPIODriverConfigure.c，然后将该文件加入工程的 Source Group 中。

在文档中嵌入相关的头文件：

```
#include "GPIODriverConfigure.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_gpio.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"
```

封装 GPIO 初始化函数 GPIOInitial():

```
void GPIOInitial(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB); //KEY RIGHT | KEY LEFT
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE); //KEY PRESS | KEY UP
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF); //KEY DOWN | LED1 |
    LED0
    //引脚功能使能
    HWREG(GPIO_PORTF_BASE+GPIO_O_PCTL)=GPIO_PCTL_PF2_LED1 |
        GPIO_PCTL_PF3_LED0;
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,GPIO_PIN_3);//Set LED0
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,GPIO_PIN_2);//Set LED1
    LED_Off(2);
    GPIOPinTypeGPIOInput(GPIO_PORTE_BASE,GPIO_PIN_5); //Set Key Press
    GPIOPinTypeGPIOInput(GPIO_PORTB_BASE,GPIO_PIN_4); //Set Key Left
    GPIOPinTypeGPIOInput(GPIO_PORTB_BASE,GPIO_PIN_6); //Set Key Right
    GPIOPinTypeGPIOInput(GPIO_PORTE_BASE,GPIO_PIN_4); //Set Key Up
    GPIOPinTypeGPIOInput(GPIO_PORTF_BASE,GPIO_PIN_1); //Set Key Down
}
```

封装开 LED 函数 LED_On(), 关 LED 函数 LED_Off(), 翻转 LED 函数 LED_Overturn()和五向键控制函数 KeyPress()。

```
void LED_On(unsigned char led_num)
{
    if (led_num == 2)
    {
        HWREGBITB(0x400253fc,2)=0;
        HWREGBITB(0x400253fc,3)=0;
    }
    if (led_num == 1)    HWREGBITB(0x400253fc,2)=0;
    if (led_num == 0)    HWREGBITB(0x400253fc,3)=0;
}
```

```
void LED_Off(unsigned char led_num)
{
    if (led_num == 2)
    {
        HWREGBITB(0x400253fc,2)=1;
        HWREGBITB(0x400253fc,3)=1;
    }
    if (led_num == 1)    HWREGBITB(0x400253fc,2)=1;
```

```
    if (led_num == 0)        HWREGBITB(0x400253fc,3)=1;
}
```

```
void LED_Overturn(unsigned char led_num)
{
    if (led_num==0)
        HWREGBITB(0x400253fc,3)=~HWREGBITB(0x400253fc,3);
    if (led_num==1)
        HWREGBITB(0x400253fc,2)=~HWREGBITB(0x400253fc,2);
    if (led_num==2)
    {
        HWREGBITB(0x400253fc,3)=~HWREGBITB(0x400253fc,3);
        HWREGBITB(0x400253fc,2)=~HWREGBITB(0x400253fc,2);
    }
}
```

```
unsigned char KeyPress(unsigned char key_num)
{
    if (key_num == 1)
    {
        if (GPIOPinRead(GPIO_PORTA_BASE,GPIO_PIN_5)==0)
            return 1;
        else
            return 0;
    }
    if (key_num == 2)
    {
        if (GPIOPinRead(GPIO_PORTB_BASE,GPIO_PIN_4)==0)
            return 1;
        else
            return 0;
    }
    if (key_num == 3)
    {
        if (GPIOPinRead(GPIO_PORTB_BASE,GPIO_PIN_6)==0)
            return 1;
        else
            return 0;
    }
    if (key_num == 4)
    {
        if (GPIOPinRead(GPIO_PORTA_BASE,GPIO_PIN_4)==0)
```

```

        return 1;
    else
        return 0;
    }
    if (key_num == 5)
    {
        if (GPIOPinRead(GPIO_PORTF_BASE,GPIO_PIN_1)==0)
            return 1;
        else
            return 0;
    }
    return 0;
}

```

■ Step4: 创建主函数文档

创建空白文档保存至 StellarisWare for C1/codes/LM3S9B96_Experiment/SysCtl_OscMain/下，命名为 main.c，然后将该文件加入工程的 Source Group 中。

■ Step5: 包含库函数相关的头文件

```

#include "inc/hw_memmap.h"           //基址
#include "inc/hw_types.h"           //数据类型设置，寄存器访问封装
#include "driverlib/debug.h"         //调试
#include "driverlib/gpio.h"         //通用 IO 口
#include "driverlib/sysctl.h"       //使能寄存器
#include "GPIODriverConfigure.h"

```

■ Step6: 定义时钟变量

定义秒、毫秒、微秒和全局的系统时钟变量。

```

#define Second 4000000
#define MilliSecond 4000
#define MicroSecond 4
unsigned long TheSysClock = 12000000UL; // 定义全局的系统时钟变量

```

■ Step7: 自制延时函数

```

void delay_s(int time)               //延时时间单位为秒
{
    time=time*Second;
    while((time--)>0);
}

void delay_ms(int time)             //延时时间单位为毫秒
{
    time=time*MilliSecond;
}

```

```

        while((time--)>0);
    }
    void delay_us(long int time)    //延时时间单位为微秒
    {
        time=time*MicroSecond;
        while((time--)>0);
    }

```

■ Step8: 创建主函数

创建 main()主函数，调用 SysCtlLDOSet () 设置 LDO 输出电压为 2.50V；调用 SysCtlClockSet()函数设置系统时钟，要求振荡器不分频、采用 OSC 作为系统时钟源、采用主振荡器作为 OSC、晶振频率 16MHZ。然后获取当前系统时钟频率，最后初始化 GPIO。

```

SysCtlLDOSet(此处添加代码);           //设置 LDO 输出电压
SysCtlClockSet(此处添加代码);         // 设置系统时钟
TheSysClock = SysCtlClockGet();        //获取当前系统时钟频率
GPIOInitial();                        //初始化 GPIO

```

■ Step9: 添加循环函数

在 GPIO 初始化之后，在 main 函数的循环函数中调用开、关 LED 函数和延时函数轮流点亮 LED0 和 LED1。

```

while(1)
{
    if (KeyPress(1))
    {
        LED_On(1);
        LED_Off(0);
        SysCtlDelay(TheSysClock/6);    //延时一段时间，使用库函数，
                                         //延长时间=3×参数×系统时钟周期。该处为 0.5S
        LED_On(0);
        LED_Off(1);
        delay_ms(500);                //延时一段时间，使用自制 delay 函数，此处为 0.5S
    }
    else
    {
        LED_Off(2);
    }
}

```

■ Step10: 添加库函数

新建一个 Group，命名为 Libraries，在 Libraries Group 中加入实验涉及到的 StellarisWare for C1/driverlib/rvmdk/driverlib.lib 文件。

■ Step11: 烧写设置

■ Step 12: 编译和下载

详细操作见 Chapter0。

2.7 思考与练习题

2.1 查阅实验板的电路图和相关资料，看看该实验板使用了几片晶振。芯片 LM3S9B96 的主晶振的频率为多少？

2.2 在本实验的主程序中包含如下语句：

```
#define Second 4000000
#define MilliSecond 4000
#define MicroSecond 4
```

试分析其含义。如果将主程序中的语句

```
SysCtlClockSet(SYSCTL_SYSDIV_1 |
                SYSCTL_USE_OSC |
                SYSCTL_OSC_MAIN |
                SYSCTL_XTAL_16MHZ);
```

修改为

```
SysCtlClockSet(SYSCTL_SYSDIV_1 |
                SYSCTL_USE_OSC |
                SYSCTL_OSC_MAIN |
                SYSCTL_XTAL_16MHZ);
```

试重新改写 Second、MilliSecond、MicroSecond 定义值。

第三章 SysTick 及通用定时器模块

3.1 概述

微处理器中的可编程定时器可对驱动定时器输入引脚的外部事件进行计数或定时。

定时器的的工作过程实际上是对时钟脉冲计数，时钟脉冲有 1ms、10ms、100ms 等不同规格。因工作需要，定时器除了占有自己编号的存储器位外，还占有一个设定值寄存器（字），一个当前值寄存器（字）。设定值寄存器（字）存储编程时赋值的计时时间设定值。当前值寄存器记录计时当前值。这些寄存器为 16 位二进制存储器。其最大值乘以定时器的计时单位值即是定时器的最大计时范围值。定时器满足计时条件开始计时，当前值寄存器则开始计数，当当前值与设定值相等时定时器动作，起常开触点接通，常闭触点断开，并通过程序作用于控制对象，达到时间控制的目的。定时器相当于继电器电路中的时间继电器，可在程序中作延时控制。

Stellaris LM3S 系列微控制器的通用定时器模块（General-Purpose Timer Module, GPTM）包含 2~4 个 GPTM 模块（定时器 0、定时器 1、定时器 2 和定时器 3）。通用定时器模块是 Stellaris LM3S 系列微控制器的一个定时器资源，其他定时器资源还包括系统定时器（SysTick）和 PWM 定时器等。

3.2 系统定时器（SysTick）

Cortex-M3 内核集成有一个系统定时器 SysTick，提供简单易用、配置灵活的 24 位单调递减计数器，还具有写入即清零、过零自动重载等特性。该计数器的用途广泛，举例来说，可以：

- 用作 RTOS 的节拍定时器，按照可编程的频率（例如 100Hz）定时触发，调用系统定时器服务子程序；
- 用作高速报警定时器，采用系统时钟作为时钟源；
- 用作频率可变的报警或信号定时器——其周期取决于所采用的参考时钟源以及计数器的动态范围；
- 用作简单计数器，测量任务的完成时刻、总体耗时等等；
- 用于实现基于失配/匹配周期的内部时钟源控制。此时通过查询 STCTRL 控制及状态寄存器的 COUNT 标志位，可以判定某个动作是否在指定的时间内完成；以此作为动态时钟管理控制环的一部分。

系统定时器包含以下 3 个寄存器：

- 系统定时器控制及状态寄存器（STCTRL）：该寄存器用于配置系统定时器的时钟、使能计数器、使能 SysTick 中断、判定计数器状态；
- 系统定时器重载值寄存器（STRELOAD）：该寄存器包含计数器重载值，每当计数器过零时自动重载；
- 系统定时器当前值寄存器（STCURRENT）：该寄存器包含计数器的当前值。

使能系统定时器后，计数器将在每个时钟递减一次，从重载值逐个递减

到0，之后在下一个时钟沿翻转（重载STRELOAD 寄存器的值），之后继续每个时钟递减一次，如此周而复始。如果将STRELOAD 寄存器清零，则会在下次重载时终止计数器的运行。当计数器递减到0 时，COUNT 标志位将置位。读取COUNT 标志位后其自动清零。

对 STCURRENT 寄存器进行写操作，即可将此寄存器清零，同时还将清零COUNT 标志位。这个写操作并不会触发SysTick异常逻辑。读取该寄存器时，返回值是该寄存器被访问时刻的内容。

系统定时器的计数器是按照处理器时钟运行的。假如在某些低功耗模式下停止提供该时钟信号，则系统定时器的计数器也将停止运行。软件在访问系统定时器的寄存器时，应确保始终采用字对齐操作予以访问。

注：在调试过程中若处理器暂停，那么此计数器也不再递减。

3.3 Timer 模块

在 Stellaris LM3S 系列微控制器的通用定时器模块（GPTM）包含 2~4 个通用定时器模块(GPTM) ——Timer0、Timer1、Timer2 和 Timer3。其中有 2~3 个定时器可用于捕获、比较及 PWM 功能。每个 GPTM 模块包含两个 16 位的定时/计数器（称作 TimerA 和 TimerB）。用户可以将它们设置成独立的定时器或事件计数器，或将它们配置成 1 个 32 位定时器或 1 个 32 位实时时钟（RTC）。定时器还可以用来触发模/数转换（ADC）。由于所有通用定时器的触发信号在到达 ADC 模块前一起进行或操作，因而只需使用一个定时器来触发 ADC 事件。

通用定时器包含四个模块，它们具有如下特性：

- 可以向上或向下计数
- 16 位或 32 位可编程的单次定时器
- 16 位或 32 位可编程的周期定时器
- 具有 8 位预分频的 16 位通用定时器
- 当有 32.768KHz 的外部时钟源时可作为 32 位的实时时钟
- 8 个捕捉比较 PWM 引脚(CCP)
- 菊花链式的定时器模块允许一个定时器开始计时多路时钟事件
- 模数转换(ADC)触发器
- 当调试时，CPU 出现暂停标识时，用户可以停止定时器事件
- 16 位输入沿计数或定时捕获模块
- 16 位可通过软件实现 PWM 信号的反相输出
- 可以确定从产生中断到进入中断服务程序所经过的时间
- 用微型直接内存访问有效的传输数据
 - 每个定时器具有专用通道
 - 定时器中断响应突发请求

通用定时器模块结构图如图 3.1 所示。

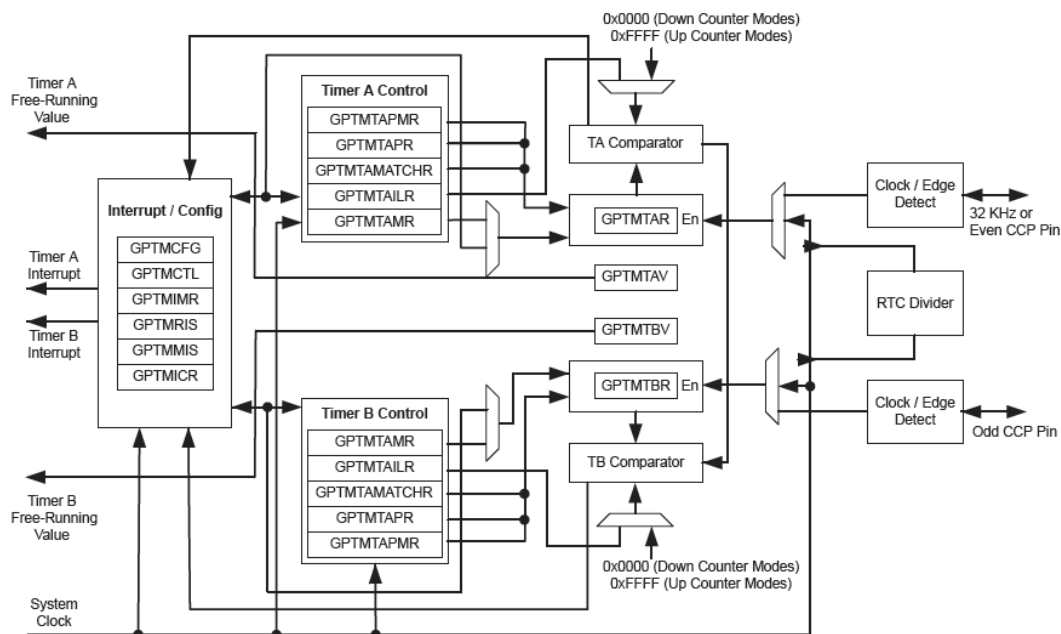


图 3.1 通用定时器模块结构图

3.4 Stellaris 库函数

3.4.1 SysTick 库函数

函数SysTickPeriodSet()用来设置SysTick计数器的周期值。函数SysTickPeriodGet()用来获取SysTick计数器的周期值。函数SysTickEnable()用来使能SysTick计数器，使其开始倒数。函数SysTckDisable()用来关闭SysTick计数器，停止计数。函数SysTickValueGet()用来获取SysTick当前的计数值。详细参数见表3.1、表3.2、表3.3、表3.4和表3.5。

表3.1 函数SysTickPeriodSet()

原型	Unsigned long SysTickPeriodSet(void)
参数	ulPeriod是每个 SysTick 计数器周期的时钟节拍数；它的值必须在 1~16,777,216 之间（1 和 16,777,216 包括在内）
返回	None.
说明	这个函数设置 SysTick 计数器绕回计数（wrap）的速率；它与相邻中断之间的处理器时钟数相等。
功能	设置 SysTick 计数器的周期

表3.2 函数SysTickPeriodGet()

原型	Unsigned long SysTickPeriodGet(void)
参数	None.
返回	返回 SysTick 计数器的周期。
说明	这个函数返回 SysTick 计数器绕回计数（wrap）的速率；它与两个中断之间的处理器时钟数相等。
功能	获取 SysTick 计数器的周期

表3.3 函数SysTickEnable()

原型	void SysTickEnable(void)
参数	None.
返回	None.
说明	这个函数将启动 SysTick 计数器。如果已经注册了一个中断处理程序，当 SysTick 计数器翻转时，中断处理程序将被调用。
功能	使能 SysTick 计数器

表3.4 函数SysTickDisable()

原型	void SysTickDisable(void)
参数	None.
返回	None.
说明	这个函数停止 SysTick 计数器。如果已经注册了一个中断处理程序，则这个中断处理程序在 SysTick 重新启动之前不会被调用。
功能	禁止 SysTick 计数器

表3.5函数SysTickValueGet()

原型	unsigned long SysTickValueGet(void)
参数	None.
返回	返回 SysTick 计数器的当前值
说明	这个函数返回 SysTick 计数器的当前值；它的值将在(周期-1)到 0 之间[(周期-1) 和 0 两个值包括在内]。
功能	获取 SysTick 计数器的当前值

3.4.2 Timer 库函数

Timer 库函数主要有以下 5 个部分：配置与控制、计数值的装载与获取、运行控制、匹配与预分频、中断控制等，这里重点介绍配置与控制以及计数值的装载与获取这两部分。

函数 TimerConfigure() 用来配置 Timer 的工作模式，这些模式包括：32 位单次触发定时器、32 位周期定时器、32 位 RTC 定时器、16 位输入边沿计数捕获、16 位输入边沿定时捕获和 16 位 PWM。对 16 位模式，Timer 被拆分为两个独立的定时/计数器 TimerA 和 TimerB，该函数能够分别对它们进行配置。详见表 3.6 的描述。

表 3.6 函数 TimerConfigure()

原型	void TimerConfigure (unsigned long ulBase, unsigned long ulConfig)
参数	ulBase 是定时器模块的基址。 ulConfig 是定时器的配置。
返回	无
说明	这个函数配置定时器的工作模式。定时器模块在配置前被禁止，并保持在禁止状态。 ulConfig 指定的配置为下面的其中一个： TIMER_CFG_32_BIT_OS：32 位单次触发定时器； TIMER_CFG_32_BIT_PER：32 位周期定时器； TIMER_CFG_32_RTC：32 位实时时钟定时器； TIMER_CFG_16_BIT_PAIR：2 个 16 位的定时器。当配置成一对 16 位

	<p>的定时器时，每个定时器单独配置。通过将 ulConfig 设置成下列其 中一个值和 ulConfig 的逻辑或结果的方法来配置第一个定时器：</p> <p>TIMER_CFG_A_ONE_SHOT: 16 位的单次触发定时器；</p> <p>TIMER_CFG_A_PERIODIC: 16 位的周期定时器；</p> <p>TIMER_CFG_A_CAP_COUNT: 16 位的边沿计数捕获；</p> <p>TIMER_CFG_A_CAP_TIME: 16 位的边沿时间捕获；</p> <p>TIMER_CFG_A_PWM: 16 位 PWM 输出。类似地，通过将 ulConfig 设置成一个相应的 TIMER_CFG_B_*值和 ulConfig 的逻辑或 结果的方法来配置第二个定时器。</p>
功能	配置定时器

函数 TimerLoadSet() 用来设置 Timer 的装载值。装载寄存器与计数器不同，它是独立存在的。在调用 TimerEnable()时会自动把装载值加载到计数器里，以后每输入一个脉冲计数器值就加 1 或减 1（取决于配置的工作模式），而装载寄存器不变。另外，除了单次触发定时器模式以外，在计数器溢出时会自动重新加载装载值。函数 TimerLoadGet()用来获取装载寄存器的值。参见表 3.7 及表 3.8 的描述。

函数 TimerValueGet() 用来获取当前 Timer 计数器的值。但在 16 位输入边沿定时捕获模里，获取的是捕获寄存器的值，而非计数器值。参见表 3.9 的描述。

表 3.7 TimerLoadSet ()

原型	void TimerLoadSet(unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
参数	ulBase 是定时器模块的基址。 ulTimer 指定调整的定时器；它的值必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 中 的一个。当定时器配置成执行 32 位的操作时，只使用 TIMER_A。 ulValue 是装载值。
返回	None.
说明	这个函数设置定时器装载值；如果定时器正在运行，则该值将立刻被装载入定时器中。
功能	设置定时器装载值

表 3.8 TimerLoadGet ()

原型	unsigned long TimerLoadGet(unsigned long ulBase, unsigned long ulTimer)
参数	ulBase 是定时器模块的基址。 ulTimer 指定定时器；它的值必须是 TIMER_A 或 TIMER_B 中的一个。当定时器配置成 执行 32 位的操作时，只使用 TIMER_A。
返回	返回定时器的装载值
说明	这个函数获取指定定时器的当前可编程时间间隔装载值
功能	获取定时器装载值

表 3.9 TimerValueGet ()

原型	unsigned long TimerValueGet (unsigned long ulBase, unsigned long ulTimer)
参数	ulBase 是定时器模块的基址。 ulTimer 指定定时器；它的值必须是

	TIMER_A 或 TIMER_B 中的一个。当定时器配置成 执行 32 位的操作时，只使用 TIMER_A。
返回	返回定时器的当前值
说明	这个函数读取指定定时器的当前值
功能	获取当前的定时器值

3.5 实验三 SysTick 及 Timer 编程

实验3.1: SysTick

■ 实验概述:

本实验通过控制五向键中的Press，将黄色网络灯(LED0)和绿色网络灯(LED1)轮流点亮，每次点亮时长为0.5秒；放开按键后立即停止闪烁。

本次实验学习目的：了解SysTick模块的功能；学会使用SysTick库函数设置时钟计数器；比较使用计数器和延时函数的区别。

■ 实验电路图

见图2.4

■ 实验流程图

本实验流程图如图 3.2。

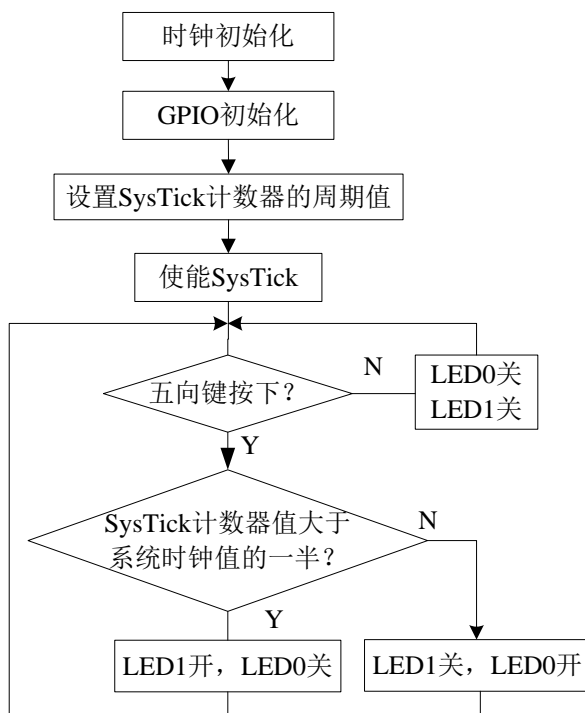


图 3.2 实验 3.1 流程图

■ Step1: 准备

检查跳帽连接是否正确，连接如下所示：

连接跳帽	对应端口	断开跳帽	对应按键
JP3	PF2	JP35	LED1
JP2	PF3	JP39	LED0
JP43	PB6	JP20,JP28	SW_1-RIGHT
JP41	PE5	JP24	SW2-PRESS
JP42	PE4	JP25	SW3-UP
JP44	PF1	JP30	SW4-DOWN
JP45	PB4	JP48	SW5-LEFT

在 StellarisWare for C1\codes\LM3S9B96_Experiment 下创建一个名为 SysTick 的文件夹，建立 SysTick 工程。

■ Step2: 配置 GPIO 驱动和 SysCtl 头文件

将文档 GPIODriverConfigure.c 、 GPIODriverConfigure.h 和 SysCtlConfigure.h 复制到 StellarisWare for C1\codes\LM3S9B96_Experiment/<PROJECT_NAME>/下，然后将 GPIODriverConfigure.c 文件加入工程的 Source Group 中。

■ Step3: SysCtl 函数的封装

建立一个空白文档，将文档保存至 StellarisWare for C1\codes\LM3S9B96_Experiment/<PROJECT_NAME>/下，命名为 SysCtlConfigure.c，然后将该文件加入工程的 Source Group 中。

在文件中嵌入相关的头文件

```
#include "SysCtlConfigure.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
```

定义全局系统时钟周期

```
unsigned long TheSysClock = 12000000UL;
```

封装系统时钟初始化函数 ClockInitial()

```
void ClockInitial(void)
{
    SysCtlLDOSet(SYSCTL_LDO_2_50V);           // 设置 LDO 输出电压
    SysCtlClockSet(SYSCTL_USE_OSC |           // 系统时钟设置
                   SYSCTL_OSC_MAIN |          // 采用主振荡器
                   SYSCTL_XTAL_16MHZ |        // 外接 16MHz 晶振
                   SYSCTL_SYSDIV_1);          // 不分频
    TheSysClock = SysCtlClockGet();            // 获取当前的系统时钟频率
}
```

■ Step4: 创建主函数文档

创建空白文档保存至 StellarisWare for C1/codes/LM3S9B96_Experiment/<PROJECT_NAME>/下，命名为 main.c，然后将该文件加入工程的 Source Group 中。

■ Step5: 包含库函数相关的头文件

```
#include "inc/hw_memmap.h"           //基址
#include "inc/hw_types.h"            //数据类型设置，寄存器访问封装
#include "driverlib/debug.h"          //调试
#include "driverlib/gpio.h"           //通用 IO 口
#include "driverlib/sysctl.h"         //使能寄存器
#include "driverlib/systick.h"        //使能 SysTick 寄存器

#include "GPIODriverConfigure.h"
#include "SysCtlConfigure.h"
```

■ Step6: 创建主函数

创建 main 函数，在 main 函数中加入系统时钟初始化函数 ClockInitial(); GPIO 初始化函数 GPIOInitial(); 使用 SysTickPeriodSet()函数设置 SysTick 计数器的周期值，溢出时间为 1s；使能 SysTick 函数 SysTickEnable()。

```
ClockInitial();           //系统时钟初始化
GPIOInitial();            //GPIO 初始化
SysTickPeriodSet(TheSysClock); // 设置 SysTick 计数器的周期值,溢出时间为 1s。
SysTickEnable();          // 使能 SysTick;
```

■ Step7: 添加循环函数

```
while(1)
{
    if (KeyPress(1)&(SysTickValueGet()>(TheSysClock/2)))
    {
        LED_On(1);
        LED_Off(0);
    }
    else
    {
        if (KeyPress(1)&(SysTickValueGet()<=(TheSysClock/2)))
        {
            LED_On(0);
            LED_Off(1);
        }
        else LED_Off(2);
    }
}
```



```
}
```

■ Step8: 添加库函数

新建一个 Group，命名为 Libraries，在 Libraries Group 中加入涉及到的 StellarisWare for C1/driverlib/rvmdk/driverlib.lib 文件。

■ Step9: 烧写设置

■ Step 10: 编译和下载

实验3.2: Timer

■ 实验概述:

本实验通过按住五向键中的 Press，将黄色网络灯(LED0)和绿色网络灯(LED1)将分别闪烁：黄色网络灯（LED0）每次点亮时长为 0.5S，闪烁周期为 1S；绿色网络灯（LED1）每次点亮时长为 0.17 秒,闪烁周期为 0.33S；放开 Press 立即停止。

本次实验学习目的：了解 Timer 模块的功能；学会设置 Timer 计数器；从两个计数器并行工作中理解计数器的并行性。

■ 实验流程图

实验流程图如图 3.3 所示。

■ Step1: 准备

检查跳帽连接是否正确，连接如下所示：

连接跳帽	对应端口	断开跳帽	对应按键
JP3	PF2	JP35	LED1
JP2	PF3	JP39	LED0
JP43	PB6	JP20,JP28	SW_1
JP41	PE5	JP24	SW2-PRESS
JP42	PE4	JP25	SW3-UP
JP44	PF1	JP30	SW4-DOWN
JP45	PB4	JP48	SW5-LEFT

在 StellarisWare for C1\codes\LM3S9B96_Experiment 下创建一个名为 Timer 的文件夹，建立 Timer 工程。

■ Step2: 配置 GPIO 驱动

将文档 GPIODriverConfigure.c 和 GPIODriverConfigure.h 复制到 StellarisWare for C1/codes/LM3S9B96_Experiment/<PROJECT_NAME>/下，然后将 GPIODriverConfigure.c 文件加入工程的 Source Group 中。

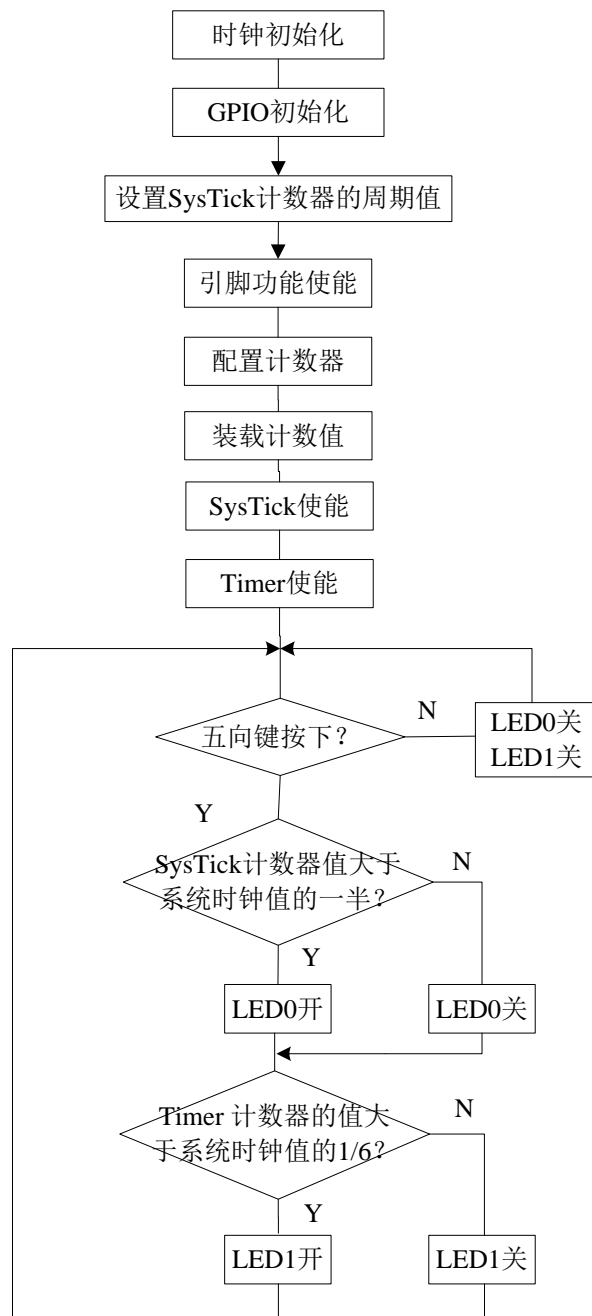


图 3.4 实验 3.3 流程图

■ Step3: 设置系统控制

将文档 SysCtlConfigure.c 和 SysCtlConfigure.h 复制到 StellarisWare for C1/codes/LM3S9B96_Experiment/<PROJECT_NAME>/ 下，然后将 SysCtlConfigure.c 文件加入工程的 Source Group 中。

■ Step4: 创建主函数文档

创建空白文档保存至 StellarisWare for C1/codes/LM3S9B96_Experiment/<PROJECT_NAME>/下，命名为 main.c，然后将该文件加入工程的 Source Group 中。

■ Step5: 包含库函数相关的头文件

```
#include "inc/hw_memmap.h"           //基址
#include "inc/hw_types.h"             //数据类型设置，寄存器访问封装
#include "driverlib/debug.h"           //调试
#include "driverlib/gpio.h"           //通用 IO 口
#include "driverlib/sysctl.h"          //使能寄存器
#include "driverlib/systick.h"        //使能 SysTick 寄存器

#include "GPIODriverConfigure.h"
#include "SysCtlConfigure.h"
```

■ Step6: 创建主函数

创建 main 函数，在 main 函数中加入下列函数。

```
ClockInitial(); //时钟初始化
GPIOInitial(); //GPIO 初始化
SysTickPeriodSet(TheSysClock);           // 设置 SysTick 计数器的周期值,溢出时间为
1s。
SysTickEnable();                         // 使能 SysTick;
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0); //使能 TIMER0
TimerConfigure(TIMER0_BASE,TIMER_CFG_32_BIT_PER); //配置 Time0r
TimerLoadSet(TIMER0_BASE,TIMER_A,(TheSysClock/3)); //溢出周期为 0.33s
SysTickEnable();                         //SysTick 开始计数
TimerEnable(TIMER0_BASE,TIMER_A);        //TIMER0 开始计数
```

■ Step7: 添加循环函数

```
while(1)
{
    if (KeyPress(1))
    {
        if (SysTickValueGet()>(TheSysClock/2))        LED_On(0);
        else LED_Off(0);

        if(TimerValueGet(TIMER0_BASE,TIMER_A)>(TheSysClock/6))
        LED_On(1);
        else LED_Off(1);
    }
    else
    {
        LED_Off(2);
    }
}
```

■ Step8: 添加库函数

新建一个 Group, 命名为 Libraries, 在 Libraries Group 中加入 Driver 中涉及到的 StellarisWare for C1/driverlib/rvmdk/driverlib.lib 文件。

■ Step9: 烧写设置

■ Step 10: 编译和下载

详细操作见 Chapter0。

3.6 思考与练习题

3.1 试改写实验 3.1 中的程序, 使黄色网络灯(LED0)和绿色网络灯(LED1)轮流点亮的时间为 1s。

3.2 试改写实验 3.2 中的程序, 使黄色网络灯(LED0)和绿色网络灯(LED1)的闪烁周期时间为 0.8s。

第四章 中 断

4.1 中断基本概念

中断（Interrupt）是 MCU 对内部或外部事件进行实时处理的一种机制。它允许 MCU 在执行程序的过程中，能够被某种内部或外部事件发生打断，转而对中断事件进行处理，中断事件处理完毕后，再返回被中断的程序处，继续执行原来的操作。

中断过程如图 4.1 所示。主程序在执行的过程中遇到中断请求（Interrupt Request）时，MCU 暂停主程序的执行，转而去执行中断服务程序（Interrupt Service Routine, ISR），即响应，中断服务例程执行完毕后返回到主程序断点处并继续执行主程序。常见的中断源有 GPIO 输入、Timer 溢出、UART 收到字符、ADC 转换完成，等等。

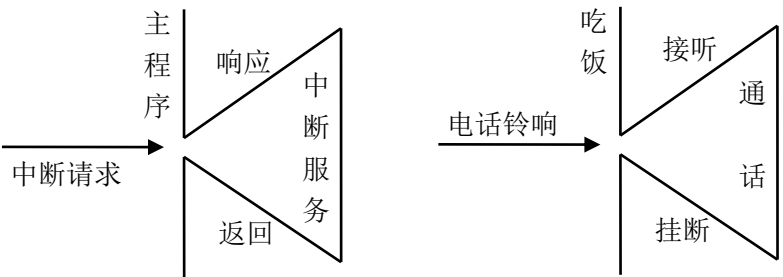


图4.1 中断过程示意图

Tips: 中断跟子程序调用有着本质的不同：子程序调用是程序员在主程序的特定位置预先安排的，通过指令来调用；中断请求对于主程序来说是随机产生的，进入中断服务例程是由硬件逻辑自动生成的跳转来实现的，而不是执行特定的指令（软中断例外）。

若系统中存在多个中断，它们都向 MCU 发出中断申请，MCU 该响应哪一个呢？因此，便有了优先级的概念，并且多个中断可以相互嵌套。正在执行的较低优先级中断可以被较高优先级的中断所打断，在执行完高级中断后返回到低级中断里继续执行，如图 4.2 所示。反之，则不行。

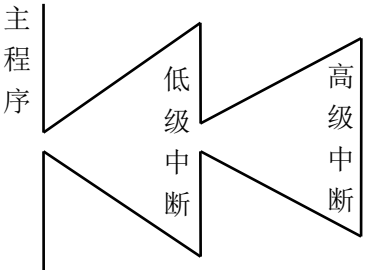


图4.2 中断嵌套示意图

4.2 Cortex-M3 内核异常与 NVIC

内核异常 (Exception) 是指在正常执行主程序以外的其它情况, 如复位、NMI (不可屏蔽中断)、存储器管理、外设中断, 等等。

Cortex-M3处理器和嵌套向量中断控制器 (Nested Vectored Interrupt Controller, NVIC) 对所有异常按优先级进行排序并处理。所有异常都在处理模式 (Handler mode) 中操作。出现异常时, 自动将处理器状态保存到堆栈中, 并在中断服务例程 (ISR) 结束时自动从堆栈中恢复。在状态保存的同时取出向量快速地进入中断。

以下特性使异常能够得到高效的、低延迟地处理:

- 自动的状态保存和恢复。处理器在进入ISR之前将状态寄存器压栈, 退出ISR之后将其弹出, 实现上述操作时不需要多余的指令;
- 自动读取代码存储器或数据SRAM中包含ISR地址的向量表入口。该操作与状态保存同时进行;
- 支持末尾连锁 (tail-chaining)。在末尾连锁中, 处理器在两个ISR之间不需要对寄存器进行出栈和压栈操作的情况下处理背对背中断;
- 中断优先级可动态重新设置;
- Cortex-M3与NVIC之间采用紧耦合接口, 通过该接口可以及早地对中断和高优先级的迟来中断进行处理;
- 中断可配置数目为1~240 (总共256个, 前16个为内核专用);
- 中断优先级可配置数目, 从3到8位, 即8~256级 (Stellaris系列仅实现3位, 即8级嵌套, 足够用);
- 处理模式 (Handler mode, 相当于中断服务程序) 和线程模式 (Thread mode, 相当于主程序) 具有独立的堆栈和特权等级;
- 采用C/C++标准的调用规范: ARM架构的过程调用标准 (Procedure Call Standard for the ARM Architecture, PCSAA) 执行ISR控制传输;
- 优先级屏蔽支持临界区 (critical regions)。

4.2.1 异常类型

CM3 内核上的异常架构支持为数众多的系统异常和外部中断。其中, 编号为 1~15 的对应系统异常, 大于等于 16 的均为外部中断, 大多数异常的优先级都是可配置的, 只有个别的异常的优先级无法改变, 如复位、NMI 等。

CM3 的所有中断机制都由 NVIC 实现, 支持 240 个外部中断, 11 个内部异常源, 加上四个未定义的异常类型, 共 256 个异常类型。具体见表 4.1。

表4.1 异常类型

类型	位置	优先级	描述
—	0	—	在复位时栈顶从向量表的第一个入口加载
Reset 复位	1	—3(最高)	在上电和热复位 (warm reset) 时调用, 在第一条指令上优先级降到最低 (线程模式), 异步的
不可屏蔽中断 (NMI)	2	—2	不能被除复位之外的任何异常停止或占先。异步的。
Hard Fault 硬故障	3	—1	由于优先级的原因或可配置的故障处理被禁止而导致不能将故障激活时的所有类型故障, 同步的

Memory Management 存储器管理	4	可配置	MPU不匹配，包括违反访问规范以及不匹配，是同步的，即使MPU被禁止或不存在，也可以用它来支持默认的存储器映射的XN区域
Bus Fault 总线故障	5	可配置	预取指故障，存储器故障，以及其它相关的地址/存储故障，精确时是同步，不精确时时异步
Usage Fault 使用故障	6	可配置	使用故障，例如，执行未定义的指令或尝试不合法的状态转换，是同步的
—	7~10	—	保留
SVCall 系统服务调用	11	可配置	系统服务调用
Debug Monitor 调试监控器	12	可配置	调试监控器，在处理器没有停止时出现，是同步的，但只有在使能时是有效的，如果它的优先级比当前有效的异常的优先级低，则不能被激活
—	13	—	保留
PendSV 可挂起的系统服务请求	14	可配置	可挂起的系统服务请求，是异步的，只能由软件来实现挂起
SysTick 系统节拍定时器	15	可配置	系统节拍定时器（System tick timer）已启动，是异步的
External Interrupt 外部中断	16至 255	可配置	在内核的外部产生（外部设备），INTISR[239:0]，通过NVIC（设置优先级）输入，都是异步的

4.2.2 优先级的定义

在 CM3 中，优先级对于异常来说非常重要，它决定了一个异常是否被响应，以及在未被屏蔽的情况下何时可以响应。优先级的数值越小，优先级越高。CM3 支持中断嵌套，高优先级的异常可以打断低优先级的异常。原则上，CM3 支持 3 个固定的优先级和多达 256 个可编程的优先级。并且支持 128 级抢占。但是，绝大多数的 CM3 芯片都会精简设计，以致实际上支持的优先级数更少，如 8 级、16 级、32 级等。它们在设计时会裁掉表达优先级的几个低端有效位，以减少优先级的级数。

4.2.3 中断向量表

异常发生后，对应的异常处理程序（exception handler）就会执行。但是，如何找到异常处理程序的入口地址呢？CM3 使用了“向量表查表机制”。向量表其实是一个 WORD 型（32 位整数）数组，存放着一个个异常的入口地址，每个下标对应一种异常。向量表的存储位置是可以设置的，通过 NVIC 中的一个重定位寄存器来指出向量表的地址。在复位后，该寄存器的值为 0。因此，在地址 0 处必须包含一张向量表，用于初始时的异常分配。

表4.2 向量表结构

异常类型	表项地址偏移量	异常向量
0	0x00	MSP 的初始值
1	0x04	复位
2	0x08	NMI
3	0x0c	硬故障
4	0x10	MemManage故障

5	0x14	总线故障
6	0x18	使用故障
7-10	0x1c - 0x28	保留
11	0x2c	SVC
12	0x30	调试监视器
13	0x34	保留
14	0x38	PendSV
15	0x3c	SysTick
16	0x40	IRQ #0
17	0x44	IRQ #1
18-255	0x48 - 0x3FF	IRQ #2 - #239

比如说，发生了异常 16（IRQ #0），那么，NVIC 就会计算出偏移移量： $16 \times 4 = 0x40$ ，然后从 0x41 处取出异常处理程序的入口地址并转移到该地址。

Tip: 类型 0 并不是什么入口地址，而是给出了复位后主堆栈指针 MSP 的初值。

4.3 外部中断/事件控制器

外部中断/事件控制器由 19 个产生事件/中断请求的边沿检测器组成。每根输入线可以独立地配置以选择类型（脉冲或挂起）和对应的触发事件（上升沿或下降沿或者双边沿都可触发）。每根输入线都可以被独立地屏蔽。挂起寄存器保持着中断请求线的状态。

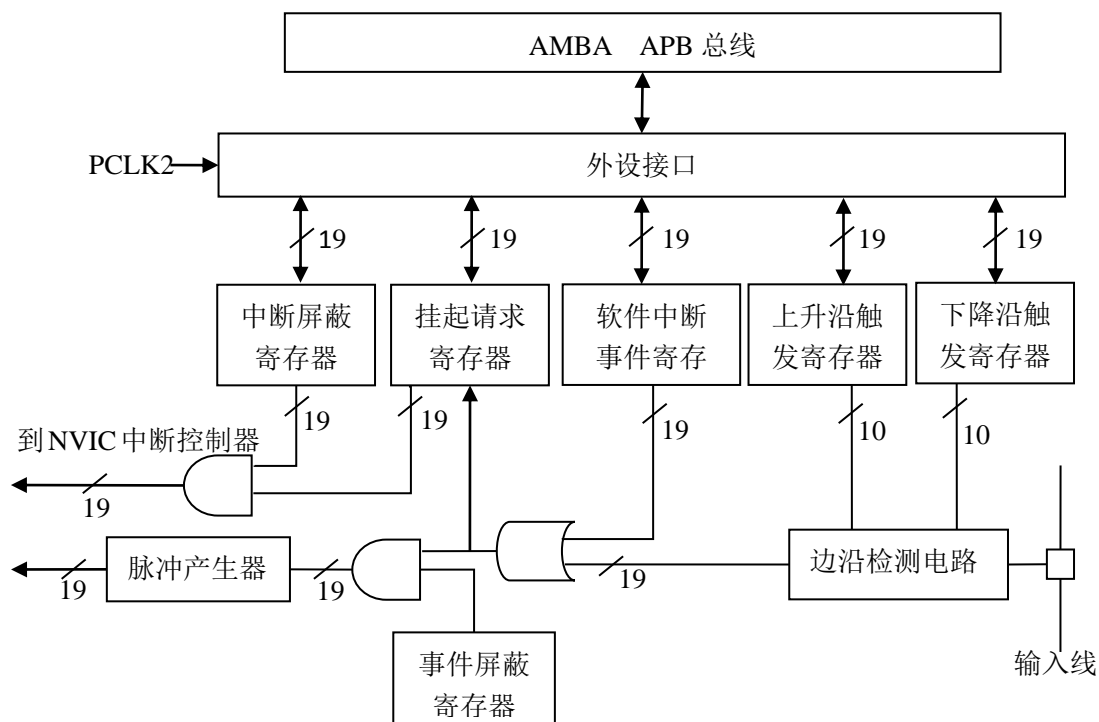


图4.3 外部中断/时间控制器框

4.4 Stellaris 中断基本编程方法

利用《Stellaris 外设驱动库》编写一个中断程序的基本方法如下：

- 使能相关片内外设，并进行基本的配置
 - 设置具体中断的类型或触发方式
 - 使能中断
 - 编写中断服务函数
 - 注册中断服务函数

(1) 使能相关片内外设，并进行基本的配置

如果要使用片内外设的中断源，首先需要对其进行使能。使能的方法是调用头文件<sysctl.h>中的函数 SysCtlPeripheralEnable()。(参见第一章相关说明)

表 4.3

原型	void SysCtlPeripheralEnable(unsigned long ulPeripheral)
参数	ulPeripheral 是要使能的外设
返回	None
说明	此函数使能外设。上电时全部的外设都被禁止；为了使外设能工作或响应寄存器的读/写操作，它们必须被使能。 ulPeripheral 参数必须取下面的其中一个值： SYSCTL_PERIPH_ADC0, SYSCTL_PERIPH_ADC1, SYSCTL_PERIPH_CAN0, SYSCTL_PERIPH_CAN1, SYSCTL_PERIPH_CAN2, SYSCTL_PERIPH_COMP0, SYSCTL_PERIPH_COMP1, SYSCTL_PERIPH_COMP2, SYSCTL_PERIPH_EPI0, SYSCTL_PERIPH_ETH, SYSCTL_PERIPH_GPIOA, SYSCTL_PERIPH_GPIOB, SYSCTL_PERIPH_GPIOC, SYSCTL_PERIPH_GPIOD, SYSCTL_PERIPH_GPIOE, SYSCTL_PERIPH_GPIOF, SYSCTL_PERIPH_GPIOG, SYSCTL_PERIPH_GPIOH, SYSCTL_PERIPH_GPIOJ, SYSCTL_PERIPH_HIBERNATE, SYSCTL_PERIPH_I2C0, SYSCTL_PERIPH_I2C1, SYSCTL_PERIPH_I2S0, SYSCTL_PERIPH_PWM, SYSCTL_PERIPH_QEI0, SYSCTL_PERIPH_QEI1, SYSCTL_PERIPH_SSI0, SYSCTL_PERIPH_SSI1, SYSCTL_PERIPH_TIMER0, SYSCTL_PERIPH_TIMER1, SYSCTL_PERIPH_TIMER2, SYSCTL_PERIPH_TIMER3, SYSCTL_PERIPH_TEMP, SYSCTL_PERIPH_UART0, SYSCTL_PERIPH_UART1, SYSCTL_PERIPH_UART2, SYSCTL_PERIPH_UDMA, SYSCTL_PERIPH_USB0, SYSCTL_PERIPH_WDOG0, or SYSCTL_PERIPH_WDOG1
功能	使能一个外设
示例	SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE); //PortE使能

(2) 设置具体中断的类型或触发方式

不同片内外设具体中断的类型或触发方式也各不相同。在使能中断之前，必须对其进行正确的设置。例如 GPIO，分为边沿触发、电平触发两大类，共 5 种，这要通过调用函数 GPIOIntTypeSet()来进行设置。

表 4.4

原型	void GPIOIntTypeSet(unsigned long ulPort, unsigned char ucPins, unsigned long ulIntType)
参数	ulPort是GPIO端口的基址。

	ucPins是特定管脚的位组合（bit-packed）表示。 ulIntType指定中断触发机制的类型。
返回	None.
说明	<p>这个函数为所选GPIO端口上特定的管脚设置不同的中断触发机制。</p> <p>参数ulIntType是一个枚举数据类型，它可以是下面其中的一个值：</p> <p>GPIO_FALLING_EDGE; GPIO_RISING_EDGE; GPIO_BOTH_EDGES; GPIO_LOW_LEVEL; GPIO_HIGH_LEVEL。</p> <p>在上面的值中，不同的值描述了中断检测机制（边沿或电平）和特定的触发事件（边沿检测的上升沿、下降沿或上升/下降沿，电平检测的低电平或高电平）。</p> <p>管脚用一个位组合（bit-packed）的字节来指定，这里的每个字节，置位的位用来识别被访问的管脚，字节的位0代表GPIO端口管脚0、位1代表GPIO端口管脚1等等。</p> <p>注：为了避免伪中断，用户必须确保GPIO输入在这个函数的执行过程中保持稳定。</p>
功能	设置指定管脚的中断类型
示例	GPIOIntTypeSet(GPIO_PORTE_BASE,GPIO_PIN_5,GPIO_FALLING_EDGE); //PE5下降沿触发

（3）使能中断

对于 Stellaris 系列 ARM，使能一个片内外设的具体中断，需要以下三个步骤：

- 调用片内外设具体中断的使能函数
- 调用函数IntEnable()，使能片内外设的总中断
- 调用函数IntMasterEnable()，使能处理器总中断

不同片内外设的中断使能函数形如 XXXIntEnable，例如，GPIO 端口的中断使能函数是 GPIOPinIntEnable()。

表 4.5

原型	void GPIOPinIntEnable(unsigned long ulPort, unsigned char ucPins)
参数	ulPort: 所选GPIO端口的基址 ucPins: 指定管脚的位组合表示
返回	无
功能	使能所选GPIO端口指定管脚的中断
示例	GPIOPinIntEnable(GPIO_PORTE_BASE,GPIO_PIN_5); //使能PE5中断

表 4.6

原型	void IntEnable(unsigned long ulInterrupt)
参数	ulInterrupt指定被使能的中断
返回	None

说明	指定的中断在中断控制器中被使能。其它的中断使能（例如外设级）不受这个函数的影响
功能	使能一个中断
示例	IntEnable(INT_GPIOE); //开启GPIOE中断源

表 4.7

原型	tBoolean IntMasterEnable(void)
参数	None
返回	None
说明	允许处理器响应中断。这不会影响在中断控制器中已使能的中断集；它只是控制控制器到处理器的个别中断
功能	使能处理器中断

（4）编写中断服务函数

中断服务函数命名：中断服务函数的名称可以由程序员自行指定，但是为了提高程序的可移植性，建议采用标准的中断服务函数名称，参见表 4.8。例如，GPIOB 端口的中断服务函数名称是 GPIO_Port_B_ISR，对应的函数头应当是 void GPIO_Port_B_ISR(void)。中断服务函数的参数和返回值都必须是 void 类型。

表 4.8 中断服务函数标准名称

向量号	中断服务函数名	向量号	中断服务函数名	向量号	中断服务函数名
0	（堆栈初值）	22	UART1_ISR	44	System_Control_ISR
1	reset_handler	23	SSI_ISR或SSI0_ISR	45	FLASH_Control_ISR
2	Nmi_ISR	24	I2C_ISR或I2C0_ISR	46	GPIO_Port_F_ISR
3	Fault_ISR	25	PWM_Fault_ISR	47	GPIO_Port_G_ISR
4	（MPU）	26	PWM_Generator_0_ISR	48	GPIO_Port_H_ISR
5	（Bus fault）	27	PWM_Generator_1_ISR	49	UART2_ISR
6	（Usage fault）	28	PWM_Generator_2_ISR	50	SSI1_ISR
7	（Reserved）	29	QEI_ISR或QEI0_ISR	51	Timer3A_ISR
8	（Reserved）	30	ADC_Sequence_0_ISR	52	Timer3B_ISR
9	（Reserved）	31	ADC_Sequence_1_ISR	53	I2C1_ISR
10	（Reserved）	32	ADC_Sequence_2_ISR	54	QEI1_ISR
11	SVCALL_ISR	33	ADC_Sequence_3_ISR	55	CAN0_ISR
12	（Debug monitor）	34	Watchdog_Timer_ISR	56	CAN1_ISR
13	（Reserved）	35	Timer0A_ISR	57	CAN2_ISR
14	PendSV_ISR	36	Timer0B_ISR	58	ETHERNET_ISR
15	SysTick_ISR	37	Timer1A_ISR	59	HIBERNATE_ISR
16	GPIO_Port_A_ISR	38	Timer1B_ISR	60	USB0_ISR
17	GPIO_Port_B_ISR	39	Timer2A_ISR	61	PWM_Generator_3_ISR
18	GPIO_Port_C_ISR	40	Timer2B_ISR	62	uDMA_ISR
19	GPIO_Port_D_ISR	41	Analog_Comparator_0_ISR	63	uDMA_Error_ISR
20	GPIO_Port_E_ISR	42	Analog_Comparator_1_ISR		
21	UART0_ISR	43	Analog_Comparator_2_ISR		

中断状态查询：一个具体的片内外设可能存在多个子中断源，但是都共用同一个中断向量。例如，GPIO 的中断状态查询函数是 `GPIOPinIntStatus()`。

表 4.9

原型	<code>long GPIOPinIntStatus(unsigned long ulPort, tBoolean bMasked)</code>
参数	<code>ulPort</code> 是GPIO端口的基址。 <code>bMasked</code> 指定返回的是屏蔽的中断状态还是原始的中断状态。
返回	返回一个位填充（bit-packed）的字节，在这个字节中，置位的位用来识别一个有效的屏蔽或原始中断，字节的位0代表GPIO端口管脚0、位1代表GPIO端口管脚1等等。位31:8应该忽略。
说明	如果 bMasked 被设置成 True ，则返回屏蔽的中断状态；否则，返回原始的中断状态。
功能	获取所指定GPIO端口的中断状态

中断清除：对于 Stellaris 系列 ARM 的所有片内外设，在进入其中断服务函数后，中断状态并不能自动清除，而必须采用软件清除。清除中断的方法是调用相应片内外设的中断清除函数。例如，GPIO 端口的中断清除函数是 `GPIOPinIntClear()`。

表 4.10

原型	<code>void GPIOPinIntClear(unsigned long ulPort, unsigned char ucPins)</code>
参数	<code>ulPort</code> 是GPIO端口的基址。 <code>ucPins</code> 是特定管脚的位组合（bit-packed）表示。
返回	None
说明	清除指定管脚的中断。 管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位0代表GPIO端口管脚0、位1代表GPIO端口管脚1等等。
功能	清除指定管脚的中断

（5）注册中断服务函数

中断服务函数写完成后，还需要对其进行注册，否则当中断事件产生时程序还无法找到它。注册方法有两种：

- 利用中断注册函数。好处是操作简单、可移植性好，缺点是由于把中断向量表重新映射到SRAM中而导致执行效率下降；
 - 直接修改启动文件。好处是执行效率很高，缺点是可移植性不够好。

推荐大家采用后一种方法，因为效率较高而且操作也并不复杂。

在 Keil 开发环境下，启动文件“`Startup.s`”是用汇编写的。以注册中断服务函数“`void I2C_ISR(void)`”为例，找到中断向量表“The vectors table”，在其前面插入 I2C_ISR 中断服务函数的声明：`EXTERN I2C_ISR`，再根据“Vectors”表格的注释内容找到外设 I2C0 的中断向量位置，把相应的“`IntDefaultHandler`”替换为“`I2C_ISR`”即可。

4.5 实验四 中断编程

■ 实验概述

本次实验重点在于了解中断的概念，理解中断向量表的作用，学会GPIO,TIMER,SYSTICK模块的中断设置及中断服务函数的编写和注册。

主要涉及以下知识点：

- 外部中断和定时中断的设置
- 中断服务函数的编写
- 中断服务函数的注册

■ 实验内容

本实验在前几次实验的基础上，使用中断的方式获取五向键的状态，并以此来控制LED灯的亮灭和闪烁：按下五向键中除Press外的另外4个键，黄色网络灯(LED0)和绿色网络灯(LED1)将开始闪烁。黄灯闪烁周期为2s；绿灯0.67s；若按下五向键中的Press键则停止闪烁。其中，五向键状态采用外部中断获取，而灯的闪烁则使用定时器中断来控制。这样系统大部分时间都处于空闲状态，外部中断或者定时中断产生时只需要很少的时间做相应的处理即可，系统效率大大提高。

■ 本实验使用的资源说明

LED：低电平有效

LED	对应端口
LED0(黄灯)	PF3
LED1(绿灯)	PF2

五向键：低电平触发

按键	对应端口	对应操作
SW_1	PB6	右
SW_2	PE5	按下
SW_3	PE4	上
SW_4	PF1	下
SW_5	PB4	左

■ 本实验涉及的部分电路图

见图 1.2。

■ 程序流程图：

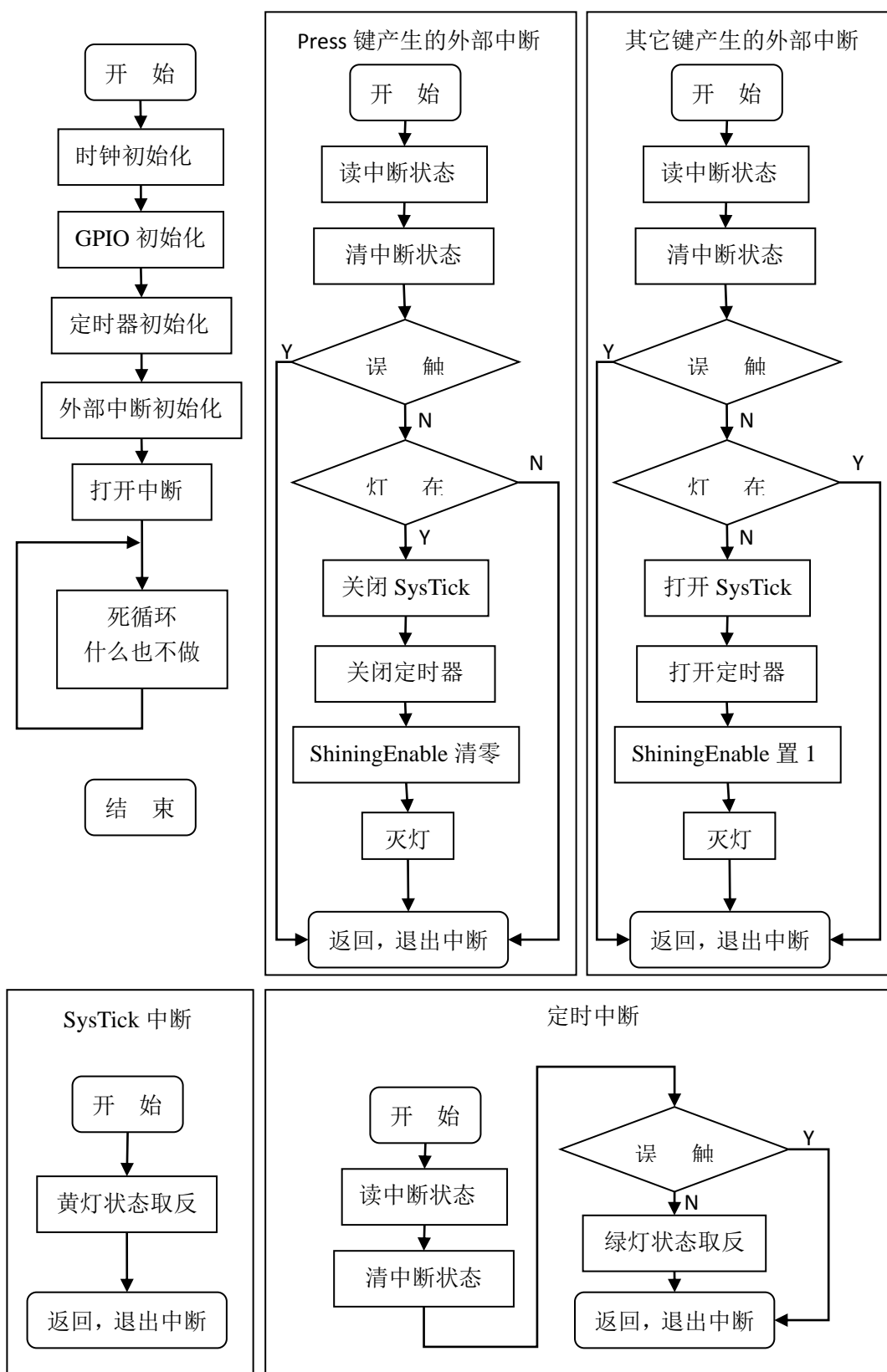


图4.4 程序流程图

■ 实验步骤

■ Step1: 准备

创建一个新的工程文件，添加 main.c 文件。为了后续实验的方便，工程所在文件夹建议命名为“Interrupt”。将前面用到的 *GPIODriverConfigure.c*、*GPIODriverConfigure.h*、*SysCtlConfigure.c*、*SysCtlConfigure.h* 拷到该目录下，并将 *main.c*、*GPIODriverConfigure.c* 和 *SysCtlConfigure.c* 添加到 Source Group 中。然后添加 driverlib.lib 至 Library 分组中。

■ Step2: 包含库函数相关的头文件

```
#include "inc/hw_memmap.h"    //基址
#include "inc/hw_types.h"     //数据类型设置，寄存器访问封装
#include "inc/hw_ints.h"      //中断向量号
#include "driverlib/debug.h"   //调试
#include "driverlib/gpio.h"   //通用 IO 口
#include "driverlib/sysctl.h" //系统控制寄存器
#include "driverlib/systick.h" //SysTick 寄存器
#include "driverlib/timer.h"   //Timer 控制寄存器
#include "driverlib/interrupt.h"

#include "GPIODriverConfigure.h"
#include "SysCtlConfigure.h"
```

■ Step3: 创建主函数

在声明的下方创建 main 函数

```
int main(void)
{
    while(1);
}
```

■ Step4: 添加初始化设置

在 main 函数中添加如下设置，系统时钟初始化、GPIO 初始化：

```
ClockInitial();
GPIOInitial();
```

然后是 SysTick 和定时器 0 初始化：

```
SysTickPeriodSet(TheSysClock);           //SysTick 计数值装载
SysTickIntEnable();                       //SysTick 中断使能

SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0); //TIMER0 使能
TimerConfigure(TIMER0_BASE,TIMER_CFG_32_BIT_PER); //设置为 32 位周期
定时器
TimerLoadSet(TIMER0_BASE,TIMER_A,(TheSysClock/3)); //TIMER0A 装载
计数值
```

```
TimerIntEnable(TIMER0_BASE,TIMER_TIMA_TIMEOUT);    //Timer0A 超时中  
断使能
```

接着进行 GPIO 中断初始化:

```
GPIOIntTypeSet(GPIO_PORTE_BASE,GPIO_PIN_5,GPIO_FALLING_EDGE);//Press  
中断  
GPIOPinIntEnable(GPIO_PORTE_BASE,GPIO_PIN_5);  
  
GPIOIntTypeSet(GPIO_PORTB_BASE,GPIO_PIN_4,GPIO_FALLING_EDGE);//Left  
中断  
GPIOPinIntEnable(GPIO_PORTB_BASE,GPIO_PIN_4);  
  
GPIOIntTypeSet(GPIO_PORTB_BASE,GPIO_PIN_6,GPIO_FALLING_EDGE);//Right  
中断  
GPIOPinIntEnable(GPIO_PORTB_BASE,GPIO_PIN_6);  
  
GPIOIntTypeSet(GPIO_PORTE_BASE,GPIO_PIN_4,GPIO_FALLING_EDGE);//Up 中  
断  
GPIOPinIntEnable(GPIO_PORTE_BASE,GPIO_PIN_4);  
  
GPIOIntTypeSet(GPIO_PORTF_BASE,GPIO_PIN_1,GPIO_FALLING_EDGE);//Down  
中断  
GPIOPinIntEnable(GPIO_PORTF_BASE,GPIO_PIN_1);
```

最后，使能所用到的中断:

```
IntEnable(INT_TIMER0A);    //开启 TIMER0A 中断源  
IntEnable(INT_GPIOE);    //开启 GPIOE 中断源  
IntEnable(INT_GPIOB);    //开启 GPIOB 中断源  
IntEnable(INT_GPIOF);    //开启 GPIOF 中断源  
IntMasterEnable();        //外围中断开启（除了 SysTick 外的所有中断都要开  
启）  
  
SysTickEnable();          //SysTick 开始计数  
TimerEnable(TIMER0_BASE,TIMER_A); //TIMER0 开始计数
```

■ Step5: 编写中断服务函数

添加 SysTick 中断服务函数:

```
void SysTick_ISR(void)    //SysTick 中断函数（需在 Startup.S 中 extern）  
{  
    LED_Overtun(0);  
}
```

添加 Timer0A 中断服务函数:

```
void Timer0A_ISR(void)  
{
```



```

unsigned long ulStatus;
ulStatus=TimerIntStatus(TIMER0_BASE,true);    //读取中断状态
TimerIntClear(TIMER0_BASE,ulStatus);          //清除该中断状态
//此处不直接填 TIMER_TIMA_TIMEOUT 为防止 TimerA 的其他中断误触发，并在
下面分支

if(ulStatus & TIMER_TIMA_TIMEOUT)              //防止误触发
{
    LED_Overturn(1);
}
}

```

添加 GPIO_Port_E 中断服务函数：

```

void GPIO_Port_E_ISR(void)
{
    unsigned long ulStatus;

    ulStatus=GPIOPinIntStatus(GPIO_PORTE_BASE,true);
    GPIOPinIntClear(GPIO_PORTE_BASE,ulStatus);

    if (ulStatus & GPIO_PIN_5)
    {
        if(ShiningEnable)
        {
            SysTickDisable();
            TimerDisable(TIMER0_BASE,TIMER_A);
            ShiningEnable=0;
            LED_Off(2);
        }
    }

    if (ulStatus & GPIO_PIN_4)
    {
        if(!ShiningEnable)
        {
            SysTickEnable();
            TimerEnable(TIMER0_BASE,TIMER_A);
            ShiningEnable=1;
            LED_On(2);
        }
    }
}

```

添加 GPIO_Port_B 中断服务函数：

```

void GPIO_Port_B_ISR(void)
{

```

```

unsigned long ulStatus;

ulStatus=GPIOPinIntStatus(GPIO_PORTB_BASE,true);
GPIOPinIntClear(GPIO_PORTB_BASE,ulStatus);

if (ulStatus & GPIO_PIN_4)
{
    if(!ShiningEnable)
    {
        SysTickEnable();
        TimerEnable(TIMER0_BASE,TIMER_A);
        ShiningEnable=1;
        LED_On(2);
    }
}

if (ulStatus & GPIO_PIN_6)
{
    if(!ShiningEnable)
    {
        SysTickEnable();
        TimerEnable(TIMER0_BASE,TIMER_A);
        ShiningEnable=1;
        LED_On(2);
    }
}
}

```

添加 GPIO_Port_F 中断服务函数：

```

void GPIO_Port_F_ISR(void)
{
    unsigned long ulStatus;

    ulStatus=GPIOPinIntStatus(GPIO_PORTF_BASE,true);
    GPIOPinIntClear(GPIO_PORTF_BASE,ulStatus);

    if (ulStatus & GPIO_PIN_1)
    {
        if(!ShiningEnable)
        {
            SysTickEnable();
            TimerEnable(TIMER0_BASE,TIMER_A);
            ShiningEnable=1;
            LED_On(2);
        }
    }
}

```

```
}
}
```

至此，main 函数完成。

■ Step6: 注册中断服务函数

首先，打开 Startup.s，加入 SysTick_ISR 等中断服务函数的 EXTERN 声明，代码如下（如下粗体部分）：

```
*****
;
;
; Place code into the reset code section.
;
*****
AREA    RESET, CODE, READONLY
THUMB

    EXTERN  SysTick_ISR
    EXTERN  Timer0A_ISR
    EXTERN  GPIO_Port_B_ISR
    EXTERN  GPIO_Port_E_ISR
    EXTERN  GPIO_Port_F_ISR
```

然后，根据“Vectors table”中的注释内容找到 SysTick_Handler 等中断向量位置，将相应的“IntDefaultHandler”替换为定义对应中断服务函数时的名称（如下粗体部分）：

```
*****
;
;
; The vector table.
;
*****
EXPORT  __Vectors
__Vectors
    DCD    StackMem + Stack                ; Top of Stack
    .....
    DCD    IntDefaultHandler                ; PendSV Handler
    DCD    SysTick_ISR                      ; SysTick Handler
    DCD    IntDefaultHandler                ; GPIO Port A
    DCD    GPIO_Port_B_ISR                  ; GPIO Port B
    DCD    IntDefaultHandler                ; GPIO Port C
    DCD    IntDefaultHandler                ; GPIO Port D
    DCD    GPIO_Port_E_ISR                  ; GPIO Port E
    DCD    IntDefaultHandler                ; UART0
    .....
    DCD    IntDefaultHandler                ; Watchdog
    DCD    Timer0A_ISR                      ; Timer 0A
    DCD    IntDefaultHandler                ; Timer 0B
```

```

.....
DCD      IntDefaultHandler          ; Flash Control
DCD      GPIO_Port_F_ISR            ; GPIO Port F
DCD      IntDefaultHandler          ; GPIO Port G
.....

```

■ Step 7: 编译和下载

4.6 思考与练习题

4.1 使用中断的方式获取五向键的状态，并以此来控制 LED 灯的亮灭和闪烁：按下五向键中除 **Press** 外的另外 4 个键，黄色网络灯(LED0)和绿色网络灯(LED1)将开始闪烁：

- 左键：黄灯闪烁周期为 0.5s；绿灯 1s；
 - 右键：黄灯闪烁周期为 1s；绿灯 0.5s；
 - 前键：黄灯闪烁周期为 1s；绿灯 1s；
 - 后键：黄灯闪烁周期为 0.5s；绿灯 0.5s；
- 若按下五向键中的 **Press** 键则停止闪烁。

第五章 看门狗定时器

5.1 看门狗(Watchdog)简介

在实际嵌入式应用中，微处理器常常会受到来自外界电磁场的干扰，造成程序跑飞或陷入死循环，导致系统无法继续工作，整个系统可能陷入停滞状态，发生不可预料的后果。出于对微处理器运行的安全可靠考虑，引入了一种专门的复位监控电路，使微控制器在进入错误状态后的一定时间内进行复位和自动恢复，这就是 WatchDog，俗称看门狗。看门狗定时器对微控制器提供了独立的保护系统，当系统出现故障时，在选定的超时周期之后，看门狗将 RESET 信号送出从而使微处理器复位。

5.2 WatchDog 功能概述

启动了看门狗的定时器之后，看门狗就开始自动递减计数，如果到了一定的时间还不去看门狗，即执行喂狗动作，那么看门狗定时器就会溢出从而引起不可屏蔽的看门狗中断，造成系统复位。当系统由于软件错误而无法响应或外部器件不能以期望的方式响应时，使用看门狗定时器可重新获得控制。

Stellaris® LM3S9B96 有两个看门狗定时器模块，一个模块(watchdog timer 0)使用系统时钟驱动，另一个模块(watchdog timer 1)由高精度内部振荡器(PIOSC)驱动。这两个模块是完全相同的，只是 WDT1 在不同的时钟域，因此需要同步器。

对于 WDT1 需要注意的是：在两次连续访问 WDT1 内部寄存器的时候要有一些时间间隙，看门狗定时器控制寄存器(WDTCTL)里面的 WRC 位表明了所需要的延时是否足够，当写操作开始时该位被清零，写操作结束后该位被置位。软件在访问另一个寄存器之前必须查询 WDTCTL 寄存器的 WRC 位是否为 1，为 1 时才可以安全的对下一个寄存器进行读写。WDT0 没有该限制，因为它是运行在系统时钟下的。

Stellaris® LM3S9B96 的两个看门狗定时器模块具有如下特性：

- 带可编程装载寄存器的32位递减计数器
- 独立的看门狗时钟使能控制
- 带中断屏蔽的可编程中断产生逻辑
- 软件跑飞时由锁定寄存器提供保护
- 复位使能/禁止产生逻辑
- 在调试过程中用户可控制看门狗暂停

看门狗定时器模块结构如图5.1所示。如图中所示，看门狗定时器模块包括一个32位递减计数器、一个可编程的装载寄存器、一个中断产生逻辑和一个锁定寄存器。看门狗使能后开始递减计数，由Comparator判断是否递减到0，若是则在产生第一个超时信号的同时，计数器重新装载看门狗定时器装载寄存器(WDTLOAD)里的值，并从该值开始递减计数。如果第二次超时发生之前，第一次超时中断已经清除了，即执行了喂狗操作，计数器将会重新装载 WDTLOAD 寄存器里的值，并继续计数。如果在第二次超时发生之时，第一

次超时中断没有被清除，即没有进行喂狗操作，并且使能了复位信号，看门狗定时器将会将复位信号通知给系统。

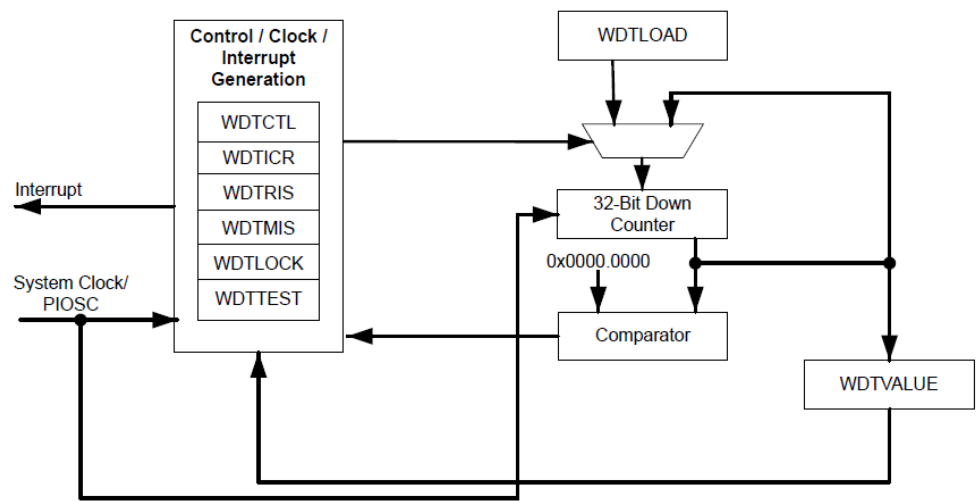


图5.1 看门狗定时器模块结构图

需要注意的是，看门狗定时器被配置后，需要将配置的状态进行锁定，以防止意外的软件修改，也就是写看门狗锁定寄存器(WDTLOCK)。因此以后要修改看门狗模块的配置，包括清除中断状态，都必须要首先解锁。另外，为了防止在调试软件时看门狗产生复位，看门狗模块还提供了允许其暂停计数的功能。

大多数微处理器都有看门狗，这可防止程序跑飞，也可以防止程序在线运行时候出现死循环。设计者必须清楚看门狗的溢出时间以决定在合适的时候清看门狗。清看门狗也不能太过频繁，否则会造成资源浪费。程序正常运行时，软件每隔一定的时间（小于定时器的溢出周期）给定时器置数，即可预防溢出中断而引起的误复位。

另外，在具体调试系统时，我们需要在硬件软件经过反复测试已经通过，而考虑到实际环境中可能出现的电磁干扰而造成程序跑飞的意外情况，再加入看门狗功能以提高系统的工作可靠性。

5.3 Stellaris 库函数

5.3.1 运行控制

函数 WatchdogEnable()的作用是使能看门狗。该函数实际执行的操作是使能看门狗中断功能，即等同于函数 WatchdogIntEnable()。中断功能一旦被使能，则只有通过复位才能被清除。函数 WatchdogResetEnable()使能看门狗定时器的复位功能，一旦看门狗定时器产生了二次超时事件，将引起处理器复位。函数 WatchdogResetDisable()禁止看门狗定时器的复位功能，此时可以把看门狗作为一个普通定时器来使用。函数 WatchdogStallEnable()允许看门狗定时器暂停计数，可防止在调试时引起不期望的处理器复位。

表5.1 WatchdogEnable 函数

原型	void WatchdogEnable(unsigned long ulBase)
参数	ulBase 是看门狗定时器模块的基址

返回	None.
说明	这个函数将使能看门狗定时器计数器和中断。 注：如果看门狗定时器已经被锁定了，则这个函数就没有任何效果了。
功能	使能看门狗定时器
示例	WatchdogEnable(WATCHDOG_BASE); //使能看门狗

表5.2 WatchdogResetEnable 函数

原型	void WatchdogResetEnable(unsigned long ulBase)
参数	ulBase 是看门狗定时器模块的基址
返回	None.
说明	当又一个超时条件出现时，使能看门狗定时器向处理器发布一次复位。 注：如果看门狗定时器已经被锁定了，那么这个函数就没有任何效果了。
功能	使能看门狗定时器复位
示例	WatchdogResetEnable(WATCHDOG_BASE); //使能看门狗复位功能

表5.3 WatchdogStallEnable 函数

原型	void WatchdogStallEnable(unsigned long ulBase)
参数	ulBase 是看门狗定时器模块的基址
返回	None.
说明	当调试器将处理器停止时，这个函数允许看门狗定时器停止计数。通过这样做来防止看门狗到达计时时间（从人类的时间角度看这通常是一个极短的时间）和复位系统（如果复位使能）。在调试执行完一定数量的处理器周期后（或在处理器重新启动后的适当时间），看门狗将继续计数到达计时时间。
功能	在调试事件过程中使能终止看门狗定时器

5.3.2 装载与锁定

函数 WatchdogReloadSet() 设置看门狗定时器的装载值，函数 WatchdogLock() 用来锁定看门狗定时器的配置，一旦锁定，则拒绝软件对配置的修改操作。函数 WatchdogUnlock() 用来解除锁定。

表5.4 WatchdogReloadSet 函数

原型	Void WatchdogReloadSet(unsigned long ulBase, unsigned long ulLoadVal)
参数	ulBase 是看门狗定时器模块的基址。 ulLoadVal 是看门狗定时器的装载值。
返回	None.
说明	当计数第一次达到零时，这个函数设置载入看门狗定时器的值；如果调用这个函数时看门狗定时器正在运行，那么这个值将立刻被载入看门狗定时器计数器。如果参数 ulLoadVal 为 0，则立刻产生一个中断。 注：如果看门狗定时器已经被锁定了，那么这个函数就没有任何效果了。
功能	设置看门狗定时器重载值
示例	WatchdogReloadSet(WATCHDOG_BASE, 2*SysCtlClockGet()); //设置看门狗装载值,定时2s

表5.5 WatchdogLock 函数

原型	void WatchdogLock(unsigned long ulBase)
参数	ulBase 是看门狗定时器模块的基址
返回	None.
功能	停止写看门狗定时器配置寄存器

表5.6 WatchdogUnlock 函数

原型	void WatchdogUnLock(unsigned long ulBase)
参数	ulBase 是看门狗定时器模块的基址
返回	None.
功能	使能对看门狗定时器配置寄存器的写访问

5.3.3 中断控制

函数 WatchdogIntEnable()用来使能看门狗定时器中断，该函数功能等同于函数 WatchdogEnable()。函数 WatchdogIntClear()用来清除中断状态。

表5.7 WatchdogIntEnable 函数

原型	void WatchdogIntEnable(unsigned long ulBase)
参数	ulBase 是看门狗定时器模块的基址
返回	None.
功能	使能看门狗定时器中断。注：如果看门狗定时器已经被锁定了，则这个函数就没有任何效果了。

表5.8 WatchdogIntClear()

原型	Void WatchdogIntClear(unsigned long ulBase)
参数	ulBase 是看门狗定时器模块的基址
返回	None.
说明	清除看门狗定时器中断，使其不再有效。注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免在真正清除中断源之前从中断处理程序中返回。如果操作失败可能会导致器件立即再次进入中断处理程序。（因为 NVIC 仍会把中断源看作是有效的）。
功能	清除看门狗定时器中断

5.4 实验五 Watchdog Timer 编程

■ 实验概述：

本实验主要是为了了解看门狗定时器的原理，学会怎样初始化和启用看门狗模块，熟悉与 Watchdog 有关的各个库函数的用法，并理解看门狗在系统

中的作用。主要涉及以下知识点：

- 看门狗定时器的原理
- 学习使用Luminary库函数中的Watchdog部分
- Watchdog的初始化设置和喂狗操作

■ 实验内容

- 初始状态下黄色网络灯（LED0）与绿色网络灯（LED1）齐亮；
- 按下Up，Down，Left与Right中的任一一键，黄色网络灯(LED0)将以警报灯的方式闪烁，具体闪烁循环如下：
亮0.1S，暗0.1S，亮0.1S，暗0.1S，亮0.1S，暗0.5S，周期为1S；
- 独立地，垂直按下五向键（Press键），将点亮绿色网络灯（LED1）0.1S，并执行喂狗操作。
- 看门狗周期设定为2S；因此，若在黄灯闪烁4个周期后，仍未喂狗，则系统会重启，回到初始状态。

■ 本实验使用的资源说明

LED：低电平有效

LED	对应端口
LED0(黄灯)	PF3
LED1(绿灯)	PF2

五向键：低电平触发

按键	对应端口	对应操作
SW_1	PB6	Right
SW_2	PE5	Press
SW_3	PE4	Up
SW_4	PF1	Down
SW_5	PB4	Left

■ 本实验涉及的部分电路图

见图 1.2。

■ 本实验程序流程图：

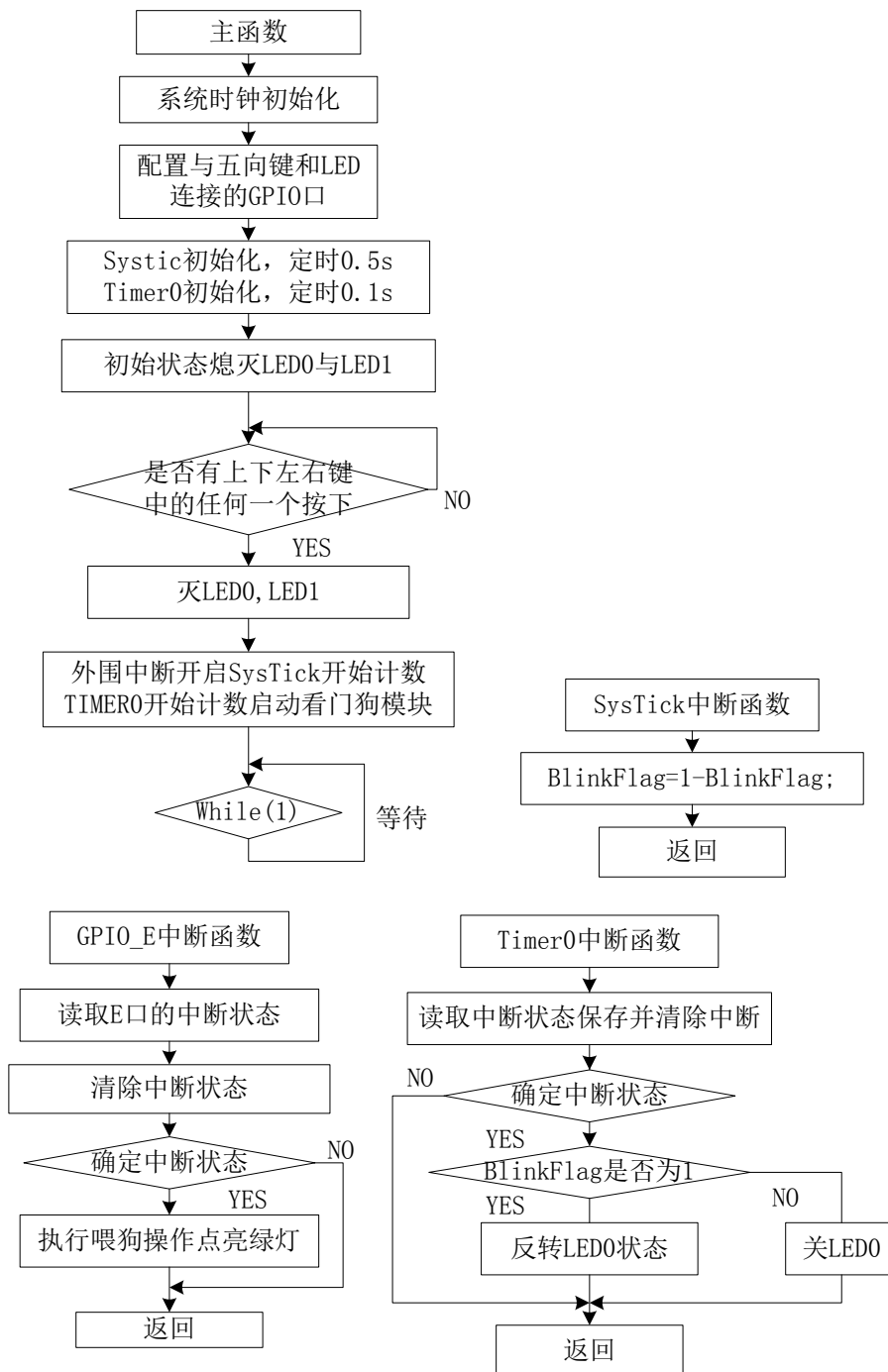


图 5.2 程序流程图

■ 实验步骤

■ Step1: 准备

创建一个新的工程文件，为了后续实验的方便，工程所在文件夹建议命名为“Watchdog(LuminaryLibrary)”。在工程文件夹里添加 main.c 和自己的库函数：

GPIO 初始化以及功能模块：“GPIODriverConfigure.c”；

系统时钟设置模块：“SysCtlConfigure.c”；

SysTick 计数器初始化模块：“SysTickConfigure.c”。

然后添加 driverlib.lib 至 Library 分组中。

■ Step2: Timer0 初始化模块封装以及添加

为了后续试验的方便，这里将 timer0 的初始化函数模块进行封装。首先在文件夹目录下点击新建，将以下代码输入，保存名字建议用“TimerConfigure.c”。

```
#include "TimerConfigure.h"
#include "SysCtlConfigure.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "driverlib/sysctl.h"
#include "driverlib/interrupt.h"
#include "driverlib/Timer.h"

void TimerInitial(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);           //TIMER0 使能
    TimerConfigure(TIMER0_BASE,TIMER_CFG_32_BIT_PER);       //设置为 32 位周期
    定时器
    TimerLoadSet(TIMER0_BASE,TIMER_A,(TheSysClock/10)); //TIMER0A 装载计 数
    值
    TimerIntEnable(TIMER0_BASE,TIMER_TIMA_TIMEOUT); //使能 Timer0A 超时中
    断
    IntEnable(INT_TIMER0A);                                //开启 TIMER0A 中 断
    源
}
```

将 TimerConfigure.c 文件添加到工程文件夹里。

■ Step3: 添加声明

新建 main.c 文件并添加声明

```
#include "inc/hw_memmap.h"           //基址
#include "inc/hw_types.h"            //数据类型设置，寄存器访问封装
#include "inc/hw_ints.h"             //中断向量号
#include "driverlib/debug.h"          //调试
#include "driverlib/gpio.h"           //通用 IO 口
#include "driverlib/sysctl.h"         //系统控制寄存器
#include "driverlib/systick.h"        //SysTick 寄存器
#include "driverlib/timer.h"          //Timer 控制寄存器
#include "driverlib/interrupt.h"      //中断控制寄存器
#include "driverlib/watchdog.h"       //看门狗控制寄存器

#include "GPIODriverConfigure.h"      // GPIO 初始化以及功能模块
#include "SysCtlConfigure.h"          //系统时钟设置模块
```

```
#include "SysTickConfigure.h"           //SysTick 计数器初始化模块
#include "TimerConfigure.h"             // timer0 初始化模块
```

■ Step4: 创建 Watchdog 初始化以及喂狗操作函数

1. 在声明的下方创建 Watchdog 初始化函数

```
unsigned char BlinkFlag=1;
void WatchDogInitial(void)
{
    unsigned long ulValue=2*TheSysClock;           //准备定时 2S,两倍时间不喂狗将重启
    SysCtlPeripheralEnable(SYSCTL_PERIPH_WDOG);    //使能看门狗模块
    WatchdogResetEnable(WATCHDOG_BASE);            //使能看门狗复位功能
    WatchdogStallEnable(WATCHDOG_BASE);            //使能调试器暂停看门狗计数
    WatchdogReloadSet(WATCHDOG_BASE, ulValue);     //设置看门狗装载值
    WatchdogEnable(WATCHDOG_BASE);                 //使能看门狗
    WatchdogLock(WATCHDOG_BASE);                   //锁定看门狗
}
```

2. 创建 WatchDogFeed 函数

```
void WatchDogFeed(void)
{
    WatchdogUnlock(WATCHDOG_BASE);                //解除锁定
    WatchdogIntClear(WATCHDOG_BASE);              //清除中断状态，即喂狗操作
    WatchdogLock(WATCHDOG_BASE);                  //重新锁定
}
```

■ Step5: 添加 SysTick、timer0、GPIO_E 的中断函数

1. SysTick 中断函数

```
void SysTick_ISR(void)                          //SysTick 中断函数
{
    BlinkFlag=1-BlinkFlag;
}
```

2. timer0 中断函数

```
void Timer0A_ISR(void)                          //TIMER0 中断函数
{
    unsigned long ulStatus;
    ulStatus=TimerIntStatus(TIMER0_BASE,true);   //读取中断状态
    TimerIntClear(TIMER0_BASE,ulStatus);         //清除该中断状态
    //此处不直接填 TIMER_TIMA_TIMEOUT 为防止 TimerA 的其他中断误触发
    if(ulStatus & TIMER_TIMA_TIMEOUT)             //确认中断源
    {
        if (BlinkFlag) LED_Overturn(0);          //若 BlinkFlag=1,LED0 状态反转
        else LED_Off(0);                          //若 BlinkFlag=0, 熄灭 LED0
    }
```

```
}  
}
```

3. GPIO_E 中断函数

```
void GPIO_Port_E_ISR(void)  
{  
    unsigned long ulStatus;  
  
    ulStatus=GPIOPinIntStatus(GPIO_PORTE_BASE,true);    //读取屏蔽之后的中断状态  
    GPIOPinIntClear(GPIO_PORTE_BASE,ulStatus);          // 清除该中断  
  
    if (ulStatus & GPIO_PIN_5)                            //确认中断源  
    {  
        WatchDogFeed();                                   //执行喂狗操作  
        LED_On(1);                                        // 点亮绿灯（LED0）  
        SysCtlDelay(TheSysClock/30);                     // 延时 100ms  
        LED_Off(1);  
    }  
}
```

■ Step6: 创建主函数

```
int main(void)  
{  
    while(1);  
}
```

■ Step7: 主函数中调用各个模块初始化设置函数

在 main 函数中添加如下设置：

```
ClockInitial();          //系统时钟设置  
GPIOInitial();           //初始化连接五向键与 LED 的 GPIO  
SysTickInitial();        //SysTick 计数值装载与使能,定时 0.5s  
TimerInitial();          //设置 timer0 为 32 位周期定时器,定时 0.1s
```

■ Step8: 继续添加控制代码

```
LED_On(2);                //初始状态，LED0 与 LED1 齐亮  
while (!(KeyPress(1) | KeyPress(2) | KeyPress(3) | KeyPress(4)));  
                           //没有按上下左右键则等待  
LED_Off(2);
```

■ Step9: 开启中断以及计数模块

```
IntMasterEnable();  
    //外围中断开启（除了 SysTick 外的所有中断都要开启）使能处理器中断  
SysTickEnable();          //SysTick 开始计数  
TimerEnable(TIMER0_BASE,TIMER_A);    //TIMER0 开始计数
```

WatchDogInitial();	//启动看门狗模块
--------------------	-----------

■ Step 10: 编译和下载

5.5 思考与练习题

5.1 修改程序，将看门狗周期设定为 4S。

第六章 通用异步收发传输器

6.1 UART 简介

计算机与外部设备的连接，使用的最多的两类接口就是：串行接口与并行接口。并行接口是指数据的各个位同时进行传送，其特点是传输速度快，但当传输距离远、位数又多时，通信线路变复杂且成本提高。串行通信是指数据一位一位地顺序传送，其特点是通信线路简单，只要一对传输线就可以实现双向通信，因而在远距离通信时可以大大降低通信成本。

串行通信又分为异步串行通信 ASYNC（Asynchronous Data Communication）与同步串行通信 SYNC（Synchronous Data Communication）两类。UART（Universal Asynchronous Receiver/Transmitter，通用异步收发器）正是设备间进行异步通信的关键模块。它的重要作用如下所示：

- 处理数据总路线和串行口之间的串/并、并/串转换；
- 只要通信双方采用相同的帧格式和波特率，就能在未共享时钟信号的情况下，自动同步信号，用两根信号线（Rx和Tx）完成通信过程；
- 采用异步方式，数据收发完成后，可通过激发中断或置位标志位的方式通知处理器进行处理，从而大大提高微控制器的工作效率。

若再加入一个电平转换器，比如 SP3232E、SP3485 等，UART 还能与 RS-232、RS-485 等通信，或者与计算机的串口对接通信。UART 在现实中得到了广泛的应用，在移动终端、工业控制、个人电脑等应用中都会用到 UART。

6.2 UART 功能概述

Stellaris® LM3S9B96 微控制器集成了三个 UART 通用异步收发器模块：UART0、UART1 和 UART2，每个 UART 都具有以下特性：

- 可编程的波特率发生器，在常规模式（16分频）下最高可达5Mbps，在高速模式（8分频）下最高可达10Mbps
- 相互独立的16×8发送FIFO和接收FIFO，可减低中断服务对CPU的占用
- FIFO出发深度有如下级别可选：1/8、1/4、1/23/4或7/8
- 标准的异步通讯位：起始位、停止位、奇偶校验位
- 线路中止（Line-break）的产生与检测
- 完全可编程的串行接口特性
 - 可包含5、6、7或8个数据位
 - 支持偶校验、奇校验、黏着校验或无校验位的产生/检测
 - 可产生1或2个停止位
- IrDA串行红外（SIR）编码器/解码器
 - 可编程选择用IrDA串行红外输入输出或普通UART输入输出
 - 支持IrDA SIR编解码器功能，半双工时数据传输率可高达115.2kbps
 - 支持正常3/16位持续时间以及低功耗（1.41~2.23us）位持续时间
 - 可编程的内部时钟发生器，可对参考时钟源进行1~256分频以提供低功耗模式位持续时间
- 支持与ISO7816智能卡的通讯

- 支持全调制解调器握手信号（仅UART1模块）
- 支持LIN协议
- 提供标准的基于FIFO深度和发送结束的中断
- 结合微直接存储器访问（uDMA）控制器使用，可以实现高效的数据传输
 - 相互独立的发送和接收通道
 - 支持单次请求接收（当接收FIFO中有数据时）和突发请求接收（当接收FIFO到达预设的触发深度时）
 - 支持单次请求发送（当发送FIFO中有空闲单元时）和突发请求发送（当发送FIFO达到预设的触发深度时）

Stellaris LM3S9B96 的 UART 模块结构框图如图 6.1 所示。

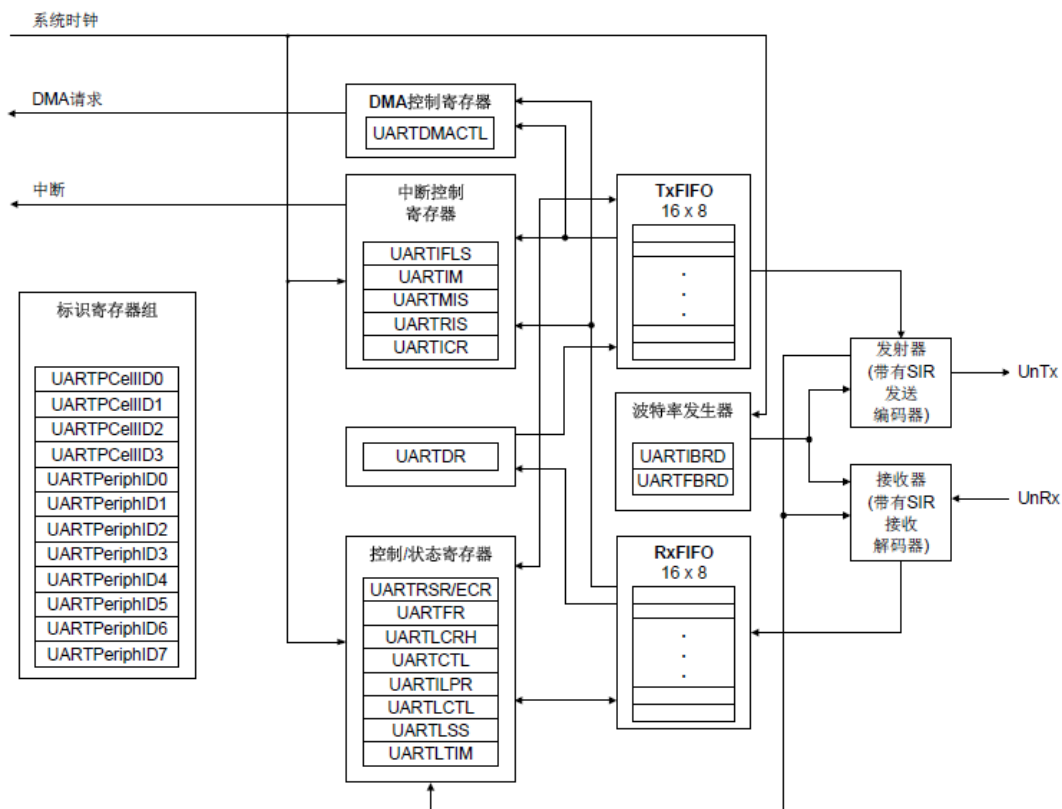


图 6.1 UART 模块结构图

6.2.1 发送/接收逻辑

发送逻辑电路对从“发送FIFO”读取的数据执行“并→串”转换。控制逻辑输出起始位在前的串行比特流，并且根据控制寄存器中已数值的配置，后面紧跟着数据位（注意：最低位LSB先输出）、奇偶校验位和停止位。参见图6.2的描述。

在检测到一个有效的起始脉冲后，接收逻辑电路对接收到的比特流执行“串→并”转换。此外还会对溢出错误、奇偶校验错误、帧错误和线路中止（line-break）错误进行检测，并将检测到的状态附加到被写入到“接收FIFO”的数据中。

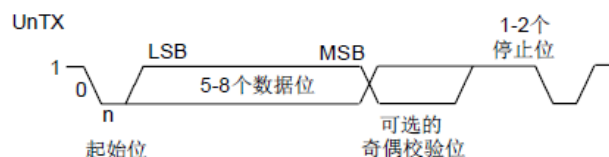


图 6.2 UART 字符帧（LSB 在前）

6.2.2 波特率的产生

波特率除数（baud-rate divisor）是一个 22 位数，它由 16 位整数和 6 位小数组成。波特率发生器使用波特率除数来决定位周期。通过带有小数波特率的除法器，在足够高的系统时钟速率下，UART 可以产生所有标准的波特率，而误差很小。

波特率除数公式：

$$BRD = BRDI.BRDF = \text{UARTSysClk} / (\text{ClkDiv} * \text{BaudRate})$$

其中：

BRD 是 22 位的波特率除数，由 16 位整数和 6 位小数组成

BRDI 是 UART 波特率分频系数整数寄存器，保存 BRD 的整数部分

BRDF 是 UART 波特率分频系数小数寄存器，保存 BRD 的小数部分

UARTSysClk 是连接到 UART 上的系统时钟

ClkDiv 是分频系数，取 16 或 8

BaudRate 是波特率（9600，38400，115200 等）

配置 UART 端口的波特率、数据帧格式的库函数是 UARTConfigSet() 宏函数。

6.2.3 数据收发

UART 的收发数据是放在两个 16 个字节的 FIFO 中的：一个用于发送，另一个用于接收。UART 在发送状态时，数据被写入“发送 FIFO”。如果 UART 被使能，则会按照预先设置好的参数（波特率、数据位、停止位、校验位等）发送数据帧，一直到“发送 FIFO”中没有数据。一旦有数据写入“发送 FIFO”（即 FIFO 未空），UART 的忙标志位 BUSY 就有效，并且在发送停止之前一直保持有效。BUSY 位仅在“发送 FIFO”为空，且已从移位寄存器发送最后一个字符，包括停止位时才变无效。即使 UART 不再使能，它也可以指示其忙状态。BUSY 位的相关库函数是 UARTBusy()。

在 UART 接收器空闲时，如果数据输入变成“低电平”，即接收到了起始位，则接收计数器开始运行，并且数据在 Baud16 的第 8 个周期或 Baud8 的第四个周期（取决于分频系数的设置）被采样。如果数据信号在 Baud16 的第 8 周期（或 Baud8 的第四个周期）仍然为低电平，则起始位有效，否则会被认为是错误的起始位并将其忽略。

如果起始位有效，则根据数据字符被编程的长度，在 Baud16 的每第 16 个周期（或 Baud8 的第 8 个周期）对连续的数据位（即一个位周期之后）进行采样。如果奇偶校验模式使能，则还会检测奇偶校验位。

最后，如果数据信号为高电平，则有效的停止位被确认，否则发生帧错误。每当接收到一个完整的字符时，都会将数据（包括任何错误位）存放到“接收 FIFO”中。

6.2.4 FIFO 操作

在进行UART通信时，中断方式比轮询方式要简便且效率高。UART模块的两个16字节收发FIFO主要就是为了解决UART收发中断过于频繁而导致CPU效率不高的问题而引入的。可以将两个FIFO分别配置为以不同深度触发中断，可供选择的配置包括：1/8、1/4、1/2、3/4和7/8深度。例如，如果接收FIFO深度选择1/4，则在UART接收到4个字节数据后会产生接收中断。复位时，两个FIFO的中断触发深度配置为1/2。用来设置收发FIFO触发中断时的深度级别的库函数是UARTFIFOLevelSet()。

● 发送FIFO的基本工作过程

只要有数据填充到发送FIFO里，就会立即启动发送过程。由于发送本身是个相对缓慢的过程，因此在发送的同时其它待发送的数据还可以继续填充到发送FIFO里。当发送FIFO被填满时就不能再继续填充了，否则会造成数据丢失，此时只能等待。这个等待并不会很久，以9600的波特率为例，等待出现一个空位的时间在1ms上下。发送FIFO会按照填入数据的先后顺序把数据一个个发送出去，直到发送FIFO全空时为止。已发送出去的数据会被自动清除，在发送FIFO里同时会多出一个空位。

● 接收FIFO的基本工作过程

当硬件逻辑接收到数据时，就会往接收FIFO里填充接收到的数据。程序应当及时取走这些数据，数据被取走也是在接收FIFO里被自动删除的过程，因此在接收FIFO里同时会多出一个空位。如果在接收FIFO里的数据未被及时取走而造成接收FIFO已满，则以后再接收到数据时因无空位可以填充而造成数据丢失。

● 发送FIFO的中断处理过程

发送数据时，触发FIFO中断的条件是当发送FIFO里剩余的数据减少到预设的深度时触发中断，而不是填充到预设的深度时触发中断。为了减少中断次数提高发送效率，发送FIFO中断触发深度级别越浅越好，如1/8深度。在需要发送大量数据时，首先要填充FIFO以启动发送过程，一定要填充到超过预设的触发深度（最好填满），然后就可以做其它事情了，剩余数据的发送工作会在中断里自动完成。当FIFO里剩余的数据减少到预设的触发深度时会自动触发中断。在中断服务函数里，继续填充发送数据，填满时退出。下次中断时继续填充，直到所有待发送数据都填充完毕为止（可以设置一个软标志来通知主程序）。

● 接收FIFO中断处理过程

接收数据时，触发FIFO中断的条件是当接收FIFO里累积的数据增加到预设的深度时触发中断。为了减少中断次数提高接收效率，接收FIFO中断触发深度级别越深越好，如7/8深度。在需要接收大量数据时，接收过程可以完全自动地完成，每次中断产生时都要及时地从接收FIFO里取走已接收到的数据（最好全部取走），以免接收FIFO溢出。

需要注意的是，在使能接收中断的同时一般都还要使能接收超时中断。

如果没有接收超时功能，则在接收 FIFO 未填充到预设深度而对方已经发送完毕的情况下并不会触发中断，结果造成最后接收的有效数据得不到处理的问题。另一种情况是对方发送过程中出现间隔，也不会触发中断，已在接收 FIFO 里的数据同样得不到及时处理。如果使能了接收超时中断，则在对方发送过程中出现 3 个数据的传输时间间隔时内就会触发超时中断，从而确保数据能够得到及时的处理。

6.2.5 中断控制

出现以下情况时，可使 UART 产生中断：

- FIFO 溢出错误
- 线路中止错误(即 Rx 信号一直为 0 的状态,包括校验位和停止位在内)
- 奇偶校验错误
- 帧错误（停止位不为 1）
- 接收超时（接收 FIFO 已有数据但未满，而后续数据长时间不来）
- 发送
- 接收

由于所有中断事件在发送到中断控制器之前会在一起做一个“或运算”操作，因此任意时刻 UART 模块都只能向中断产生一个中断请求。通过查询中断状态函数 UARTIntStatus()，程序可以在同一个中断服务函数里处理多个事件的中断。

6.3 Stellaris 函数库

6.3.1 配置与控制

函数 UARTConfigSetExpClk() 用来对 UART 端口的波特率、数据格式进行配置。在实际编程时，往往用形式更简单的宏函数 UARTConfigSet() 来代替上述库函数。参见表 6.1 和表 6.2 的描述。

表 6.1 函数 UARTConfigSetExpClk()

原型	void UARTConfigSetExpClk(unsigned long ulBase, unsigned long ulUARTClk, unsigned long ulBaud, unsigned long ulConfig)
参数	ulBase 是 UART 端口的基址。 ulUARTClk 是提供给 UART 模块的时钟速率。 ulBaud 是希望的波特率。 ulConfig 是端口的数据格式（数据位的数目、停止位的数目和奇偶位）
返回	None.
说明	此函数将配置 UART 在指定的数据格式下工作。波特率由 ulBaud 参数提供，数据格式由 ulConfig 参数提供。 ulConfig 参数是数据位数目、停止位数目和奇偶位 3 个值的逻辑或。 UART_CONFIG_WLEN_8 、 UART_CONFIG_WLEN_7 、 UART_CONFIG_WLEN_6 和 UART_CONFIG_WLEN_5 分别用来选择每个字节含有

	8 ~ 5 个数据位。UART_CONFIG_STOP_ONE 和 UART_CONFIG_STOP_TWO 分别用来选择 1 或 2 个停止位。UART_CONFIG_PAR_NONE、UART_CONFIG_PAR_EVEN、UART_CONFIG_PAR_ODD、UART_CONFIG_PAR_ONE 和 UART_CONFIG_PAR_ZERO 选择奇偶模式（分别选择无奇偶位、偶校验位、奇校验位、奇偶位总是为 1 和奇偶位总是为 0）。外设时钟将与处理器的时钟相同。该时钟值将会是 SysCtlClockGet() 返回的值，或如果该时钟为已知常量时（调用 SysCtlClockGet() 时用来保存代码/执行体），可以明确此时钟是硬编码。这个函数取代了最初的 UARTConfigSet() API，并执行相同的操作。uart.h 中提供一个宏来把最初的 API 映射到这个新的 API 中。
功能	设置一个 UART 的配置。

表 6.2 宏函数 UARTConfigSet()

原型	#define UARTConfigSet(a, b, c) UARTConfigSetExpClk(a, SysCtlClockGet(), b, c)
参数	详见表6-1的描述
返回	无
说明	本宏函数常常用来代替函数UARTConfigSetExpClk()，在调用之前应当先调用SysCtlClockSet()函数设置系统时钟
功能	UART配置（自动获取时钟速率）
示例	<pre>UARTConfigSet(UART0_BASE, // 配置UART0 9600, // 波特率：9600 UART_CONFIG_WLEN_8 // 数据位：8位 UART_CONFIG_STOP_ONE // 停止位：1位 UART_CONFIG_PAR_NONE); // 校验位：无</pre>

6.3.2 使能与禁止

函数 UARTEnable() 和 UARTDisable() 用来使能和禁止 UART 端口的收发功能。一般是先配置 UART，最后使能收发。当需要修改 UART 配置时，应当先禁止，配置完成后再使能。参见表 6.3 和表 6.4 的描述。

表 6.3 函数 UARTEnable()

原型	void UARTEnable(unsigned long ulBase)
参数	ulBase 是 UART 端口的基址。
返回	None.
说明	设置 UARTEN、TXE 和 RXE 位，并使能发送和接收的 FIFO。
功能	使能发送和接收。
示例	UARTEnable(UART0_BASE); //使能 UART0

表 6-4 函数 UARTDisable()

原型	void UARTDisable(unsigned long ulBase)
参数	ulBase 是 UART 端口的基址。
返回	None.
说明	清零 UARTEN、TXE 和 RXE 位，再等待当前字符发送结束，然后刷新

	发送 FIFO。
功能	禁止发送和接收。

6.3.3 数据收发

函数 UARTCharPut()以轮询的方式发送数据，如果发送 FIFO 有空位则填充要发送的数据，否则一直等待。函数 UARTCharGet()以轮询的方式接收数据，如果接收 FIFO 里有数据则读出数据并返回，否则一直等待。参见表 6.5 和表 6.6 的描述。

表 6.5 函数 UARTCharPut()

原型	void UARTCharPut(unsigned long ulBase, unsigned char ucData)
参数	ulBase 是 UART 端口的基址。 ucData 是要发送的字符。
返回	None.
说明	把字符 ucData 发送到指定端口的发送 FIFO 中。如果发送 FIFO 中没有多余的可用空间， 这个函数将会一直等待， 直至在返回前发送 FIFO 中有可用的空间。
功能	等待着把指定端口的字符发送出去。
示例	<pre>while(*message!='\0') { //向 UART0 发送 FIFO 写入一个字符串 UARTCharPut(UART0_BASE,*(message++)); }</pre>

表 6.6 函数 UARTCharGet()

原型	long UARTCharGet(unsigned long ulBase)
参数	ulBase 是 UART 端口的基址。
返回	Returns the character read from the specified port, cast as an long.
说明	从指定端口的接收 FIFO 中获取一个字符。如果没有可用的字符，这个函数将一直等待， 直至接收到一个字符， 然后再返回。
功能	等待指定端口的一个字符。
示例	UARTCharGet(UART0_BASE); //从 UART0 接收 FIFO 读取一个字符

为了避免UART收发过程中长时间的等待，函数UARTSpaceAvail()用来探测发送FIFO里是否有可用的空位。该函数一般用在正式发送之前，通常与“无阻塞”数据发送函数 UARTCharPutNonBlocking()配合使用。函数 UARTCharsAvail()用来探测接收FIFO里是否有接收到的数据。该函数一般用在正式接收之前，通常与“无阻塞”数据接收函数UARTCharGetNonBlocking()配合使用。参见表6.7~表6.10的描述。

表6.7 函数UARTSpaceAvail()

原型	tBoolean UARTSpaceAvail(unsigned long ulBase)
参数	ulBase is the base address of the UART port.
返回	true: 在发送FIFO里有可用空间 false: 在发送FIFO里没有可用空间（发送FIFO已满）
说明	通常， 本函数需要跟函数UARTCharPutNonBlocking()配合使用
功能	确认在指定UART端口的发送FIFO里是否有可用的空间

表6.8 函数UARTCharsAvail()

原型	tBoolean UARTCharsAvail(unsigned long ulBase)
参数	ulBase is the base address of the UART port.
返回	true: 在接收FIFO里有字符 false: 在接收FIFO里没有字符（接收FIFO为空）
说明	通常，本函数需要跟函数UARTCharGetNonBlocking()配合使用
功能	确认在指定UART端口的接收FIFO里是否有字符

表6.9 函数UARTCharPutNonBlocking()

原型	tBoolean UARTCharPutNonBlocking(unsigned long ulBase, unsigned char ucData)
参数	ulBase: UART端口的基址，取值UART0_BASE、UART1_BASE或UART2_BASE ulData: 要发送的字符
返回	如果发送FIFO里有可用空间，则将数据放入发送FIFO，并立即返回true 如果发送FIFO里没有可用空间，则立即返回false（发送失败）
说明	通常，在调用本函数之前应当先调用UARTSpaceAvail()确认发送FIFO里有可用空间
功能	发送1个字符到指定的UART端口（不等待）

表6.10 函数UARTCharGetNonBlocking()

原型	long UARTCharGetNonBlocking(unsigned long ulBase)
参数	ulBase: UART端口的基址，取值UART0_BASE、UART1_BASE或UART2_BASE
返回	如果接收FIFO里有字符，则立即返回接收到的字符（自动转换为long型） 如果接收FIFO里没有字符，则立即返回-1（接收失败）
说明	通常，在调用本函数之前应当先调用UARTCharsAvail()来确认接收FIFO里有字符
功能	从指定的UART端口接收1个字符（不等待）
示例	<pre>// Loop while there are characters in the receive FIFO. while(UARTCharsAvail(UART0_BASE)) { // Read the next character from the UART0 and write it back to the UART0. UARTCharPutNonBlocking(UART0_BASE, UARTCharGetNonBlocking(UART0_BASE)); }</pre>

6.3.4 中断控制

函数UARTIntEnable()用来使能UART端口的一个或多个中断。函数UARTIntClear()用来清除UART的中断状态，函数UARTIntStatus()用来获取UART的中断状态。参见表6.11~表6.13的描述。

表6.11 函数UARTIntEnable()

原型	void UARTIntEnable(unsigned long ulBase, unsigned long ulIntFlags)
----	--

参数	<ulbase: uart端口的基址,="" 取值uart0_base、uart1_base或uart2_base<br=""></ulbase:> ulIntFlags: 指定的中断源, 应当取下列值之一或者它们之间的任意“或运算”组合形式: UART_INT_OE // FIFO溢出错误中断 UART_INT_BE // BREAK错误中断 UART_INT_PE // 奇偶校验错误中断 UART_INT_FE // 帧错误中断 UART_INT_RT // 接收超时中断 UART_INT_TX // 发送中断 UART_INT_RX // 接收中断 注: 接收中断和接收超时中断通常要配合使用, 即UART_INT_RX UART_INT_RT
返回	无
功能	使能指定UART端口的一个或多个中断
示例	<pre>// Enable the UART0 receive interrupt and receive timeout interrupt IntEnable(INT_UART0); UARTIntEnable(UART0_BASE, UART_INT_RX UART_INT_RT);</pre>

表6.12 函数UARTIntClear()

原型	void UARTIntClear(unsigned long ulBase, unsigned long ulIntFlags)
参数	参见表6-11的描述
返回	无
功能	清除指定UART端口的一个或多个中断

表6.13 函数UARTIntStatus()

原型	unsigned long UARTIntStatus(unsigned long ulBase, tBoolean bMasked)
参数	ulBase: UART端口的基址, 取值UART0_BASE、UART1_BASE或UART2_BASE bMasked: false表示需要获取原始的中断状态, true表示需要获取屏蔽的中断状态
返回	原始的或屏蔽的中断状态
功能	获取指定UART端口当前的中断状态
示例	<pre>unsigned long ulStatus = UARTIntStatus(UART0_BASE, true); // Get the interrupt status UARTIntClear(UART0_BASE, ulStatus); // Clear the asserted interrupts</pre>

6.4 实验六 UART

■ 实验概述:

本实验通过五向键来控制UART端口的数据读写, 并进行字符的大小写转换, 同时练习网络接口上的两个独立LED灯(LED0黄色JP2, LED1绿色JP3)的点亮和熄灭。

本次实验重点在于, 了解UART的基本协议、功能, 学会使用UART总线与上位机作交互, 并理解char型数据与ASCII码的关系, 会用转义字符。主要涉及以下知识点:

- UART模块的配置
- UART模块数据的收发

● ASCII字符表与大小写转换

■ 实验内容

- 初始状态下黄色网络灯（LED0）与绿色网络灯（LED1）齐亮；初始化完成后UART向上位机发送信息：
Initial Done!
Press the KEY to continue~
- 按下Press键，系统启动。启动后系统会处于以下4种状态：
状态0：默认状态或按下DOWN后进入。此时仅将上位机发送的信息原样返回。
该状态下LED0与LED1齐灭。
状态1：按下LEFT后进入。此时将上位机发送的小写字符‘a’~‘z’转换为大写‘A’~‘Z’后返回，其他字符原样返回。
该状态下LED1亮。
状态2：按下RIGHT后进入。此时将上位机发送的大写字符‘A’~‘Z’转换为小写‘a’~‘z’后返回，其他字符原样返回。
该状态下LED0亮。
状态3：按下UP后进入，此时将上位机发送的小写字符转换为大写字符，大
写字符转换为小写字符并返回，其他字符原样返回。
该状态下LED0与LED1齐亮。
- 注意：若有字符输入，系统返回时每隔最多1S会自动换行

■ 本实验使用的资源说明

LED：低电平有效

LED	对应端口
LED0(黄灯)	PF3
LED1(绿灯)	PF2

五向键：低电平触发

按键	对应端口	对应操作
SW_1	PB6	Right
SW_2	PE5	Press
SW_3	PE4	Up
SW_4	PF1	Down
SW_5	PB4	Left

UART：

UART-0	对应端口
RX(接收)	PA0

TX(发送)	PA1
--------	-----

跳帽连接如下所示：

连接跳帽	对应端口	断开跳帽	对应按键
JP3	PF2	JP35	LED1
JP2	PF3	JP39	LED0
JP43	PB6	JP20,JP28	SW_1
JP41	PE5	JP24	SW2-PRESS
JP42	PE4	JP25	SW3-UP
JP44	PF1	JP30	SW4-DOWN
JP45	PB4	JP48	SW5-LEFT

■ 本实验涉及的部分电路图

见图 1.2。

■ 本实验程序流程图

见图 6.3。

■ 实验步骤

■ Step1: 准备

创建一个新的工程文件，添加 main.c 文件。为了后续实验的方便，工程所在文件夹建议命名为“UART”。再将前面实验里完成的相关模块一起添加到新工程：

GPIODriverConfigure.c,
SysCtlConfigure.c,
SysTickConfigure.c,
TimerConfigure.c

最后，添加名为 UARTConfigure.c 的文件。

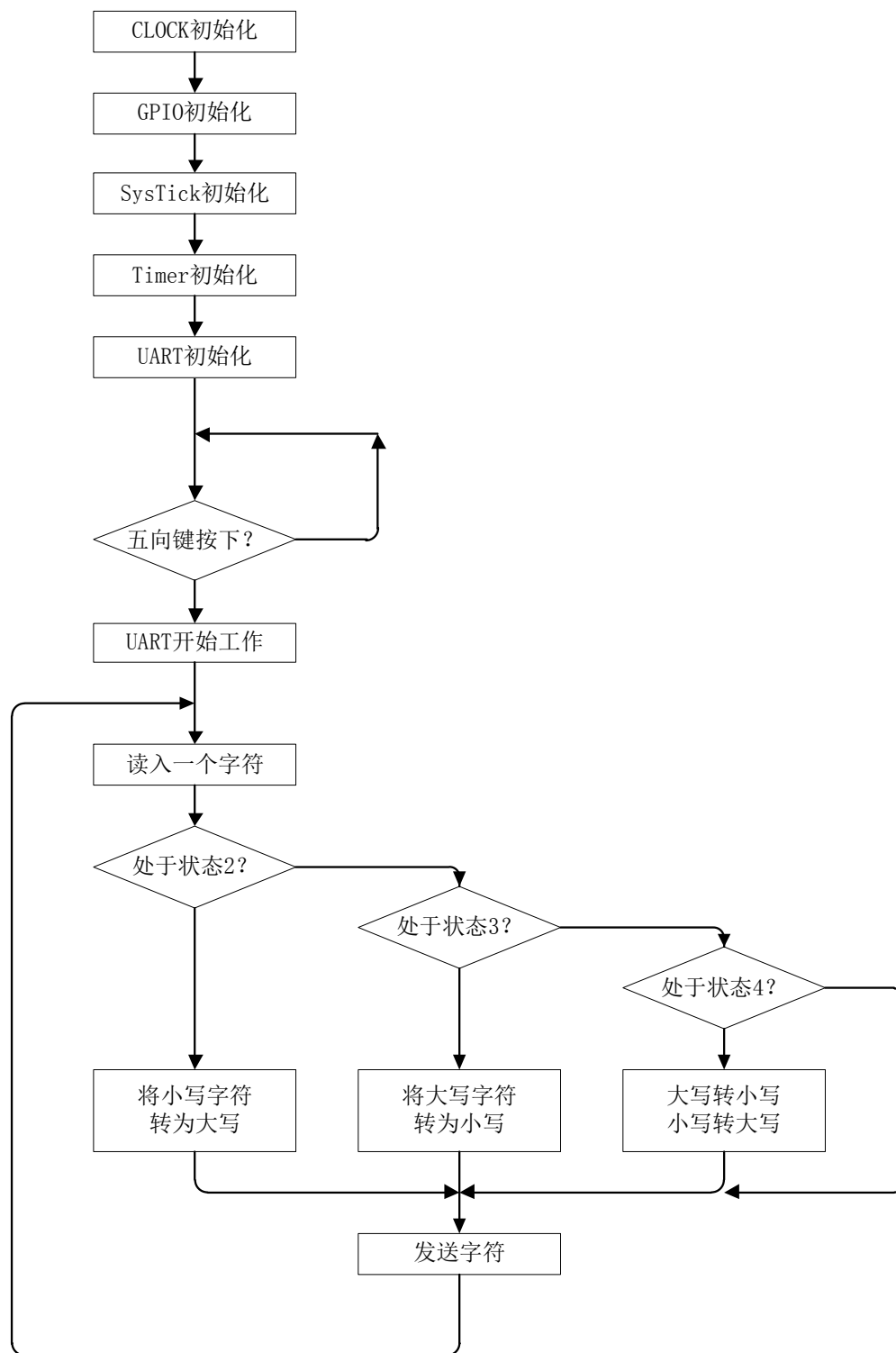


图6.3 程序流程图

■ Step2: 添加声明

在 main.c 中加入声明：

```

#include "inc/hw_memmap.h"           //基址
#include "inc/hw_types.h"           //数据类型设置，寄存器访问封装
#include "inc/hw_ints.h"            //中断向量号

```

```

#include "driverlib/debug.h"           //调试
#include "driverlib/gpio.h"           //通用 IO 口
#include "driverlib/sysctl.h"         //系统控制寄存器
#include "driverlib/systick.h"        //SysTick 寄存器
#include "driverlib/timer.h"          //Timer 控制寄存器
#include "driverlib/interrupt.h"      //中断控制寄存器
#include "driverlib/watchdog.h"
#include "driverlib/UART.h"           //UART 控制接口
#include "GPIODriverConfigure.h"
#include "SysCtlConfigure.h"
#include "SysTickConfigure.h"
#include "TimerConfigure.h"
#include "WatchDogConfigure.h"
#include "UARTConfigure.h"

```

```

static unsigned char Send_Flag=0; //发送换行标志位
static unsigned char Status=0;    //转换标志位

```

■ Step3: 创建主函数

在声明的下方创建 main 函数

```

int main(void)
{
    return 0;
}

```

■ Step4: 添加 GPIO 初始化设置

在 GPIOConfigure.c 的 GPIOInitial 函数中加入如下代码:

```

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);    //UART0
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);    //KEY RIGHT | KEY LEFT
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);    //KEY PRESS | KEY UP
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);    //KEY DOWN | LED1 | LED0

//引脚功能使能
HWREG(GPIO_PORTA_BASE+GPIO_O_PCTL)= GPIO_PCTL_PA0_U0RX |
                                       GPIO_PCTL_PA1_U0TX;

```

■ Step5: 添加 UART 初始化设置

在 UARTConfigure.c 内加入如下代码:

```

#include "UARTConfigure.h"
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/SysCtl.h"
#include "driverlib/UART.h"

```

```

void UART0Initial(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);           //使能 UART0 模块
    UARTConfigSet(UART0_BASE,                               //配置为 UART0 端口
                  115200,                                    //波特率: 115200
                  UART_CONFIG_WLEN_8 |                      //数据位: 8
                  UART_CONFIG_STOP_ONE |                   //停止位: 1
                  UART_CONFIG_PAR_NONE);                  //校验位: 无
    UARTEnable(UART0_BASE);                                //使能 UART0
}

void UARTStringPut(unsigned long ulBase,const char *message) //从串口发送字符串
{
    while(*message!='\0')
    {
        UARTCharPut(ulBase,*(message++));
    }
}

```

在 UARTConfigure.h 中加入如下代码:

```

#ifndef __UARTCONFIGURE_H_
#define __UARTCONFIGURE_H_

//Uart0 通过 FT2232D 通信芯片与上位机连接, 无需跳帽
//通信波特率设为 115200
extern void UART0Initial(void);
extern void UARTStringPut(unsigned long ulBase,const char *);

#endif

```

■ Step6: 添加中断处理代码

在 main.c 中添加中断处理的代码, 打开:

```

void SysTick_ISR(void)                                     //SysTick 中断函数
{
    if (Send_Flag)
    {
        UARTCharPut(UART0_BASE,'\r');                    //定时换行
        UARTCharPut(UART0_BASE,'\n');
        Send_Flag=0;
    }
}

void GPIO_Port_E_ISR(void)                                //PortE 有 KEY PRESS 和 KEY UP

```

```

{
    unsigned long ulStatus;
    ulStatus=GPIOPinIntStatus(GPIO_PORTE_BASE,true);
    GPIOPinIntClear(GPIO_PORTE_BASE,ulStatus);
    if (ulStatus & GPIO_PIN_4)                //按下 UP 键
    {
        Status=3;                            //小写转大写，大写转小写
        LED_On(2);
    }
}

// KEY RIGHT 和 KEY LEFT 的中断服务函数
void GPIO_Port_B_ISR(void)
{
    unsigned long ulStatus;
    ulStatus=GPIOPinIntStatus(GPIO_PORTB_BASE,true);
    GPIOPinIntClear(GPIO_PORTB_BASE,ulStatus);
    if (ulStatus & GPIO_PIN_4)                //按下 LEFT 键
    {
        Status=1;                            //小写转大写
        LED_On(1);
        LED_Off(0);
    }
    if (ulStatus & GPIO_PIN_6)                //按下 RIGHT 键
    {
        Status=2;                            //大写转小写
        LED_On(0);
        LED_Off(1);
    }
}

void GPIO_Port_F_ISR(void)
{
    unsigned long ulStatus;
    ulStatus=GPIOPinIntStatus(GPIO_PORTF_BASE,true);
    GPIOPinIntClear(GPIO_PORTF_BASE,ulStatus);
    if (ulStatus & GPIO_PIN_1)                //按下 DOWN 键
    {
        Status=0;                            //不转换
        LED_Off(2);
    }
}

```

■ Step7: 添加控制初始化代码

在 main.c 中添加初始化的代码:

```

char tempChar;

ClockInitial();
GPIOInitial();
SysTickInitial();
TimerInitial();
UART0Initial();
UARTStringPut(UART0_BASE,"Initial Done!\r\n");
UARTStringPut(UART0_BASE,"Press the KEY to continue~\r\n");

LED_On(2);
while (!KeyPress(1));
LED_Off(2);
UARTStringPut(UART0_BASE,"GO! ! \r\n");

IntMasterEnable();           //外围中断开启（除了 SysTick 外的所有中断都要开启）
SysTickEnable();              //打开 SysTick 中断
TimerEnable(TIMER0_BASE,TIMER_A); //打开 TIMER0

```

■ Step8: 添加 UART 控制代码

在 main.c 中添加 UART 的控制代码：

```

while(1)
{
    tempChar=UARTCharGet(UART0_BASE);    //从 UART 读入一个字符
    if (Status==1)                        //转换实现过程
    {
        if (tempChar>= 97 && tempChar<= 122)    //小写转大写
        {
            tempChar=tempChar-32;
        }
    }
    else if (Status==2)
    {
        if (tempChar >= 65 && tempChar <= 90)    //大写转小写
        {
            tempChar=tempChar+32;
        }
    }
    else if (Status==3)
    {
        if (tempChar>= 97 && tempChar<= 122)    //大小写互换
        {
            tempChar=tempChar-32;
        }
    }
}

```

```

        else
        {
            if (tempChar >= 65 && tempChar <= 90)
            {
                tempChar=tempChar+32;
            }
        }
    }
    UARTCharPut(UART0_BASE,tempChar);           //发送转化好的字符
    Send_Flag=1;
    if (tempChar=='\r')                         //如果发现'\r'就补发一个回车
    {
        UARTCharPut(UART0_BASE,'\n');
    }
}

```

■ Step 9: 编译和下载

6.5 思考与练习题

6.1 修改例程，完成中断方式的 UART 数据接收：

- 1) 在 main 函数中使能 UART0 接收中断和接收超时中断，并进入无限循环；
- 2) 编写 UART 中断服务程序 void UARTIntHandler(void)，获取中断状态并清中断，同时获取接收 FIFO 中的所有字符，完成状态 3 的操作，即大小写字母互换后回显；
- 3) 注册 UART 中断服务程序。

附录 A Keil 建立工程文件步骤

建立工程前请确保拥有该处理器的函数库文件，常用文件名为 StellarisWare for C1，其中包含 driverlib, usblib, grlib 等基础库。并检查 StellarisWare for C1/codes/LM3S9B96/driver 是否存在。（在实验室条件下，建议将 StellarisWare for C1 文件夹拷贝到 D 盘，后续的工作都在 D 盘的这个文件夹下进行）。

■ Step1: 建立工程

打开 Keil uVision4 软件，点击 Project→NewuVersion Project...，如图 A.1 所示。

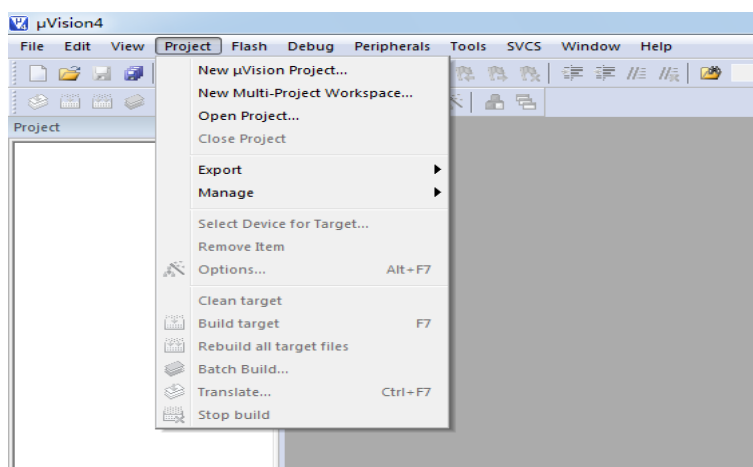
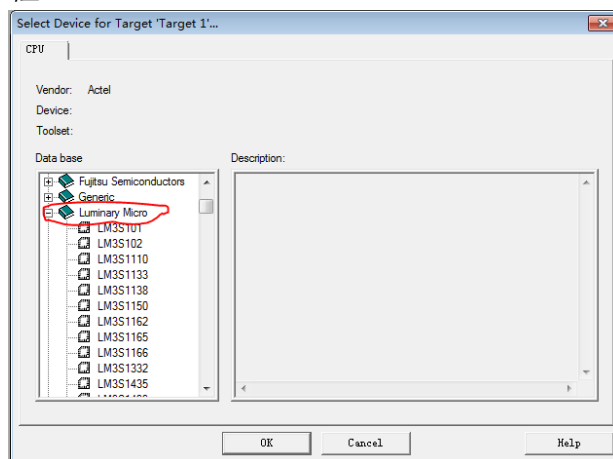


图 A.1

选择工程路径为工程模板文件夹下的 StellarisWare for C1/codes/LM3S9B96/<PROJECT_NAME>，最后一级文件夹需新建，并建议将工程名称与该文件夹名称取同名。

■ Step2: 选择芯片型号

选择 Luminary Micro 下的 LM3S9B96，如图 A.2 所示。并允许复制并添加文件进入该工程。



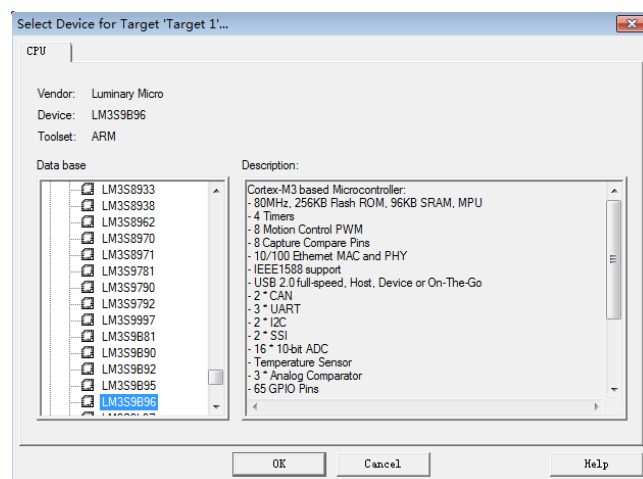



图 A.2

■ Step3: 建立主函数

点击 File-New（或快捷键 ）建立一个空白文档，键入主程序(main.c)，保存至 StellarisWare for C1/codes/LM3S9B96/<PROJECT_NAME>/下，注意添加后缀为 .c。如图 A.3 所示。

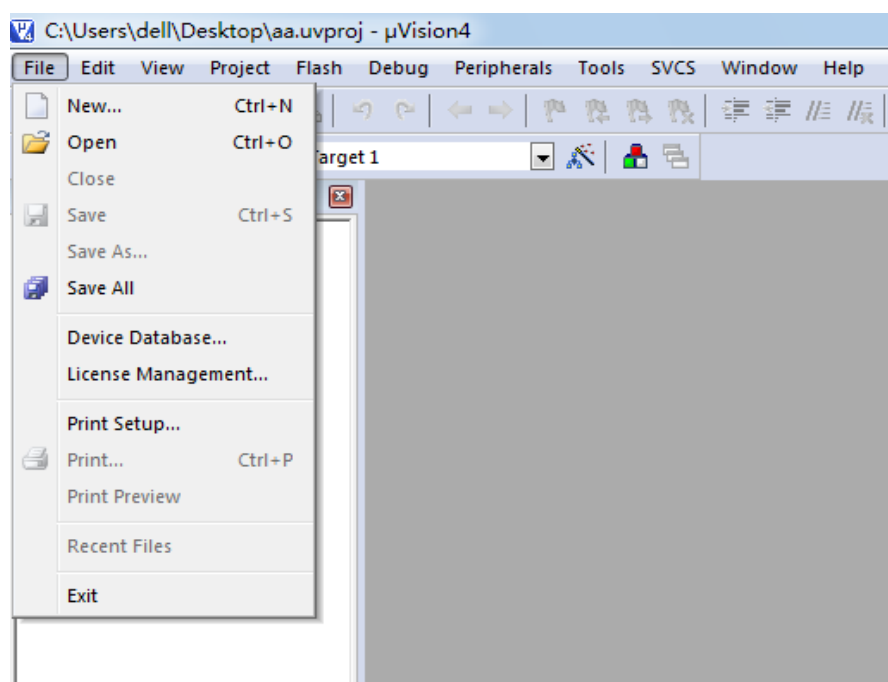


图 A.3

在 main.c 文件中添加如下代码：

```
int main(void)
{
    return 0;
}
```

将该文件加入工程中，如图 A.4 所示。（注意：添加文件后地址窗口将不

会自动，此时不用重复添加，关闭窗口即可。)

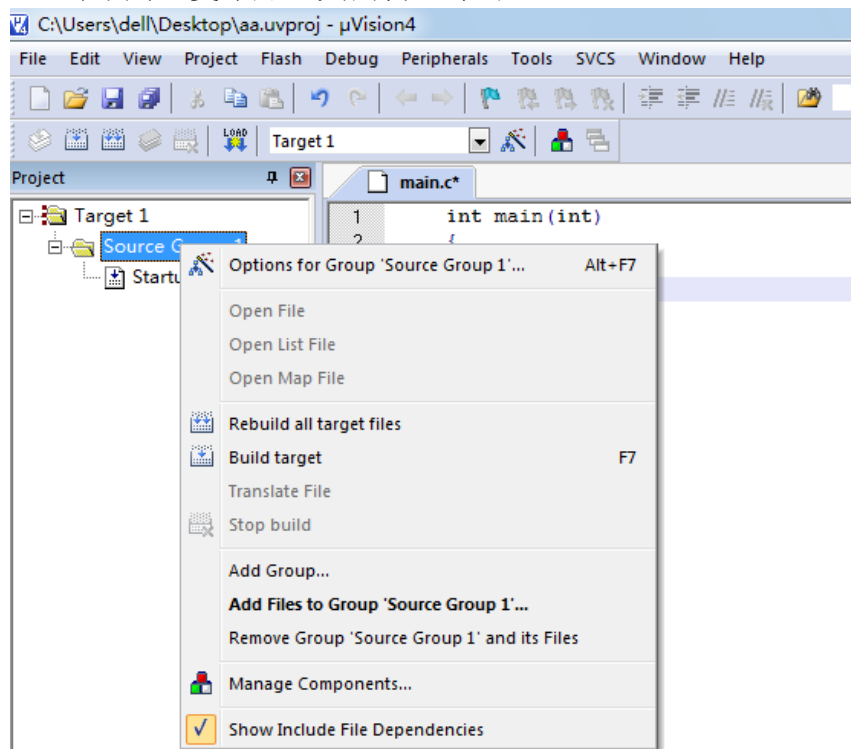


图 A.4

■ Step4: 添加库函数

根据硬件电路的不同设计，需要在 ARM 启动时对 Cortex-M3 进行初始化和设置，类似操作系统中的驱动。一般不把驱动写入主程序，而是建立公共的函数库保证效率。StellarisWare for C1/codes/LM3S9B96/driver 中即包含针对 S700 实验板的驱动模板及常用函数，需将程序中涉及的部分加入工程中，并作必须的修改。添加方法与上一步相同。添加后效果如图 A.5 所示。

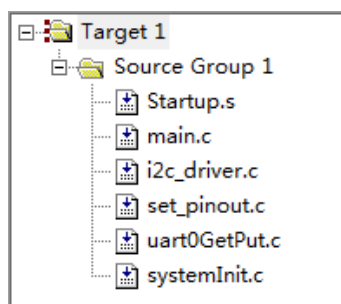


图 A.5

新建两个 Group，分别取名 Libraries 和 Documentation，如图 A.6 所示。

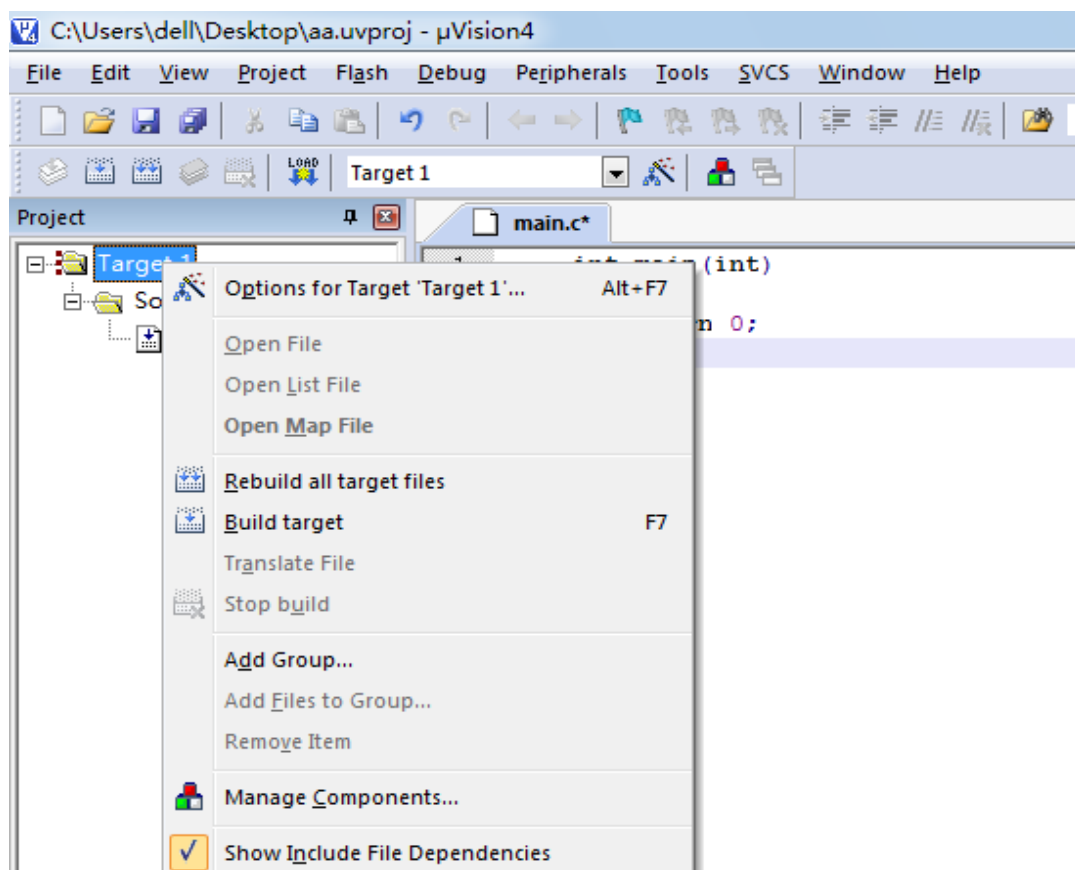


图 A.6

在 Libraries Group 中加入 Driver 中涉及到的 .lib 文件。例如 StellarisWare for C1/driverlib/rvmdk/driverlib.lib 和 StellarisWare for C1/grlib/rvmdk/grlib.lib。Documentation Group 包含说明文件，可有可无。完成后的目录结构如图 A.7 所示。

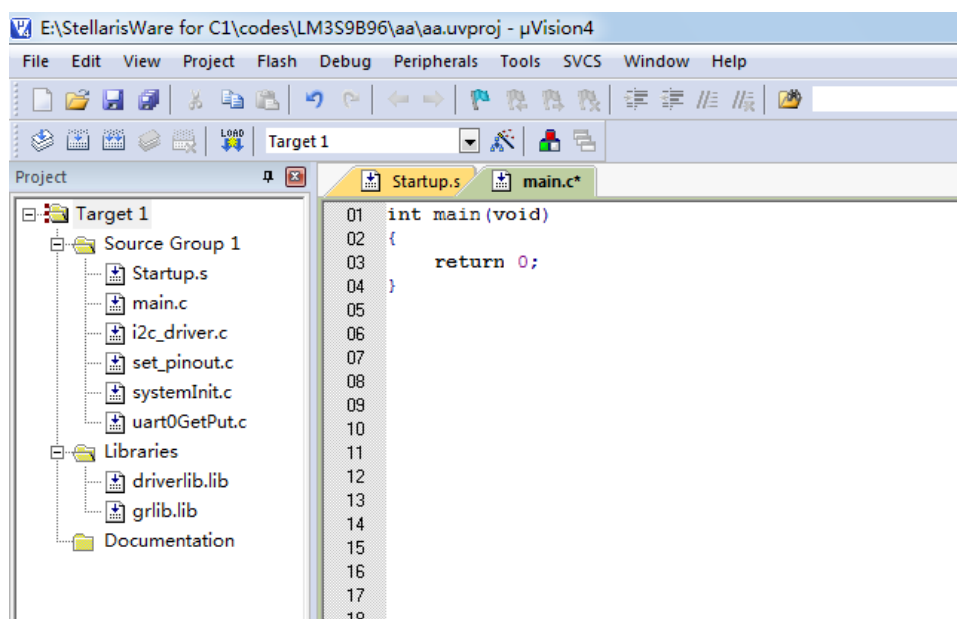


图 A.7

■ Step5: 烧写设置

新建文件夹 StellarisWare for
C1/codes/LM3S9B96/<PROJECT_NAME>/rvmdk

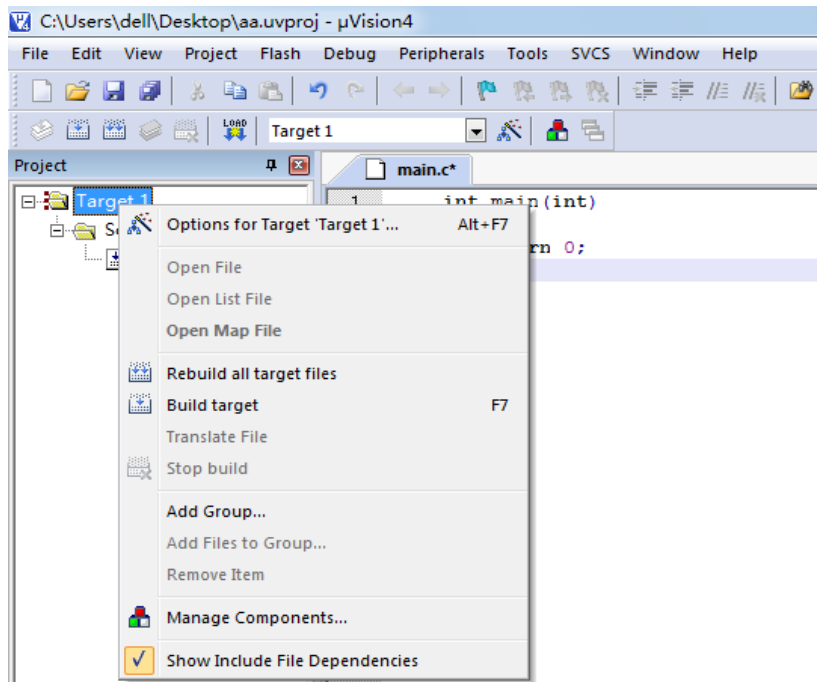


图 A.8

如图 A.8 所示，设置 Target。依次设置下列版面，注意红色部分为需改动部分。未注释部分根据图中式样选填即可。

注意：晶振频率根据实际电路频率设置。

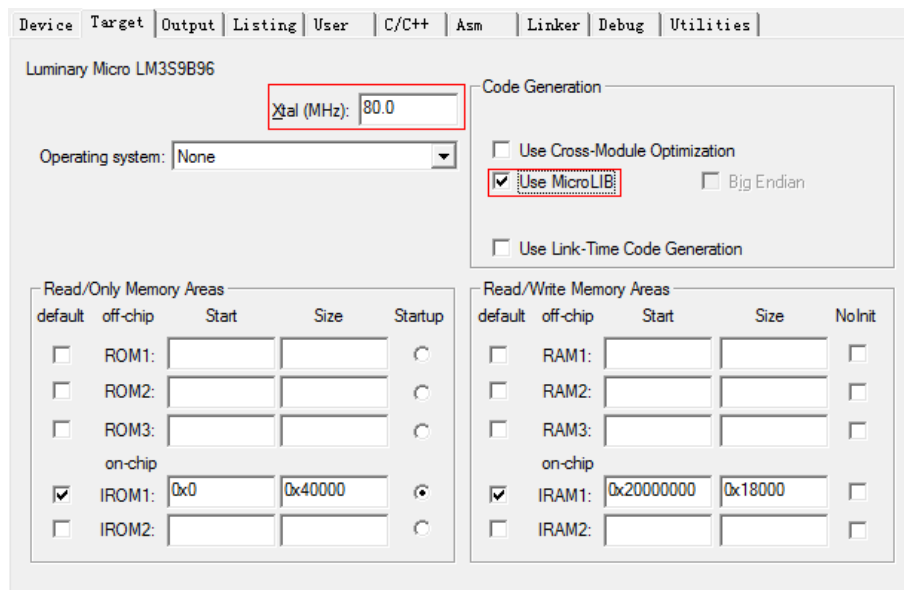


图 A.9

注意：设置路径为\rvmdk

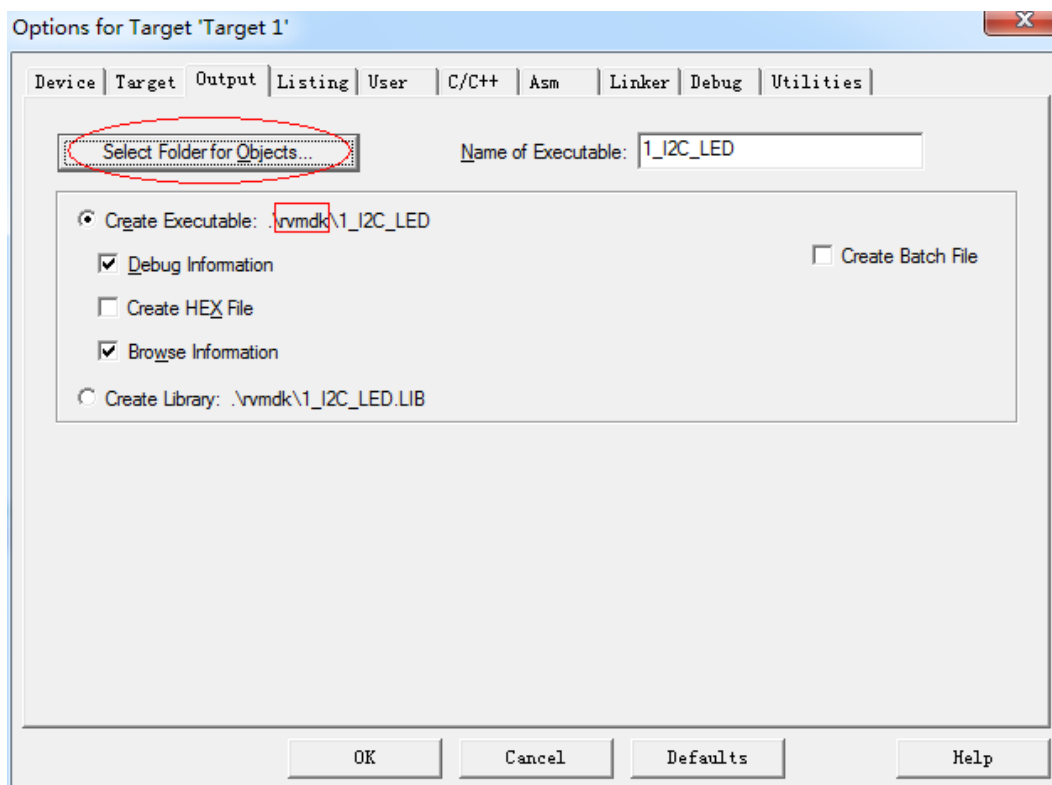


图 A.10

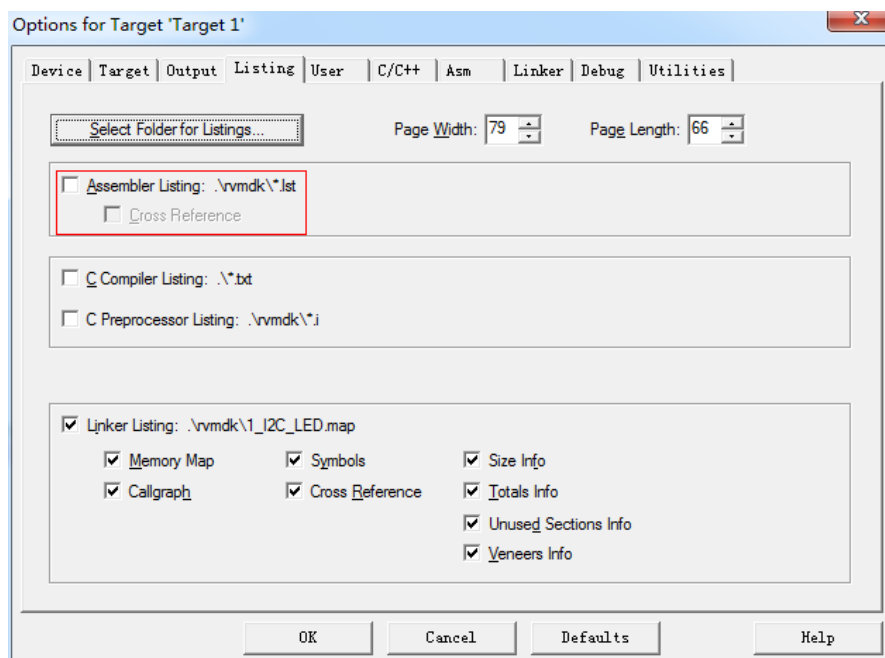


图 A.11

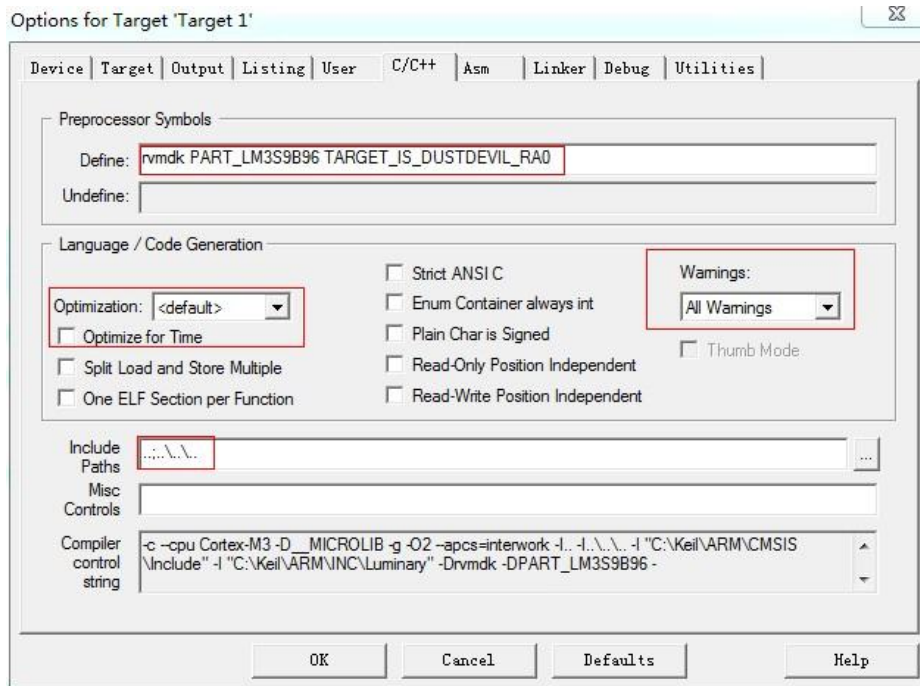


图 A.12

其中 Include Paths 包含前面添加的所有库文件的地址。..表示工程文件的上一级，可包含多个路径，用分号隔开。一般情况下需包含基本库函数文件夹 driverlib, usblib, grlib 等的根目录，以及驱动文件夹 driver 的根目录。

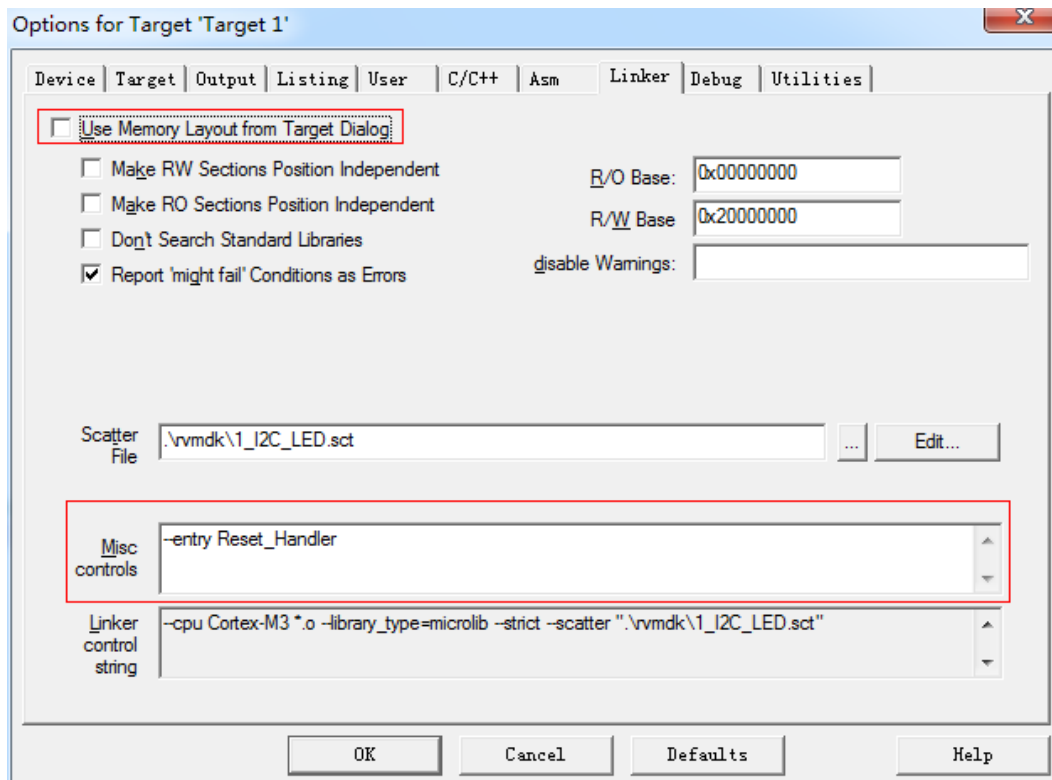


图 A.13

上图页面设置预留的地址区，首先在 Misc Controls 中填入 --entry Reset_Handler，再取消 Use Memory Layout from Target Dialog 和 Make RW Sections Position Independent 选项前的勾。此时在 Scatter File 中会出现.\rvmdk\<PROJECT_NAME>.sct。要注意的是该 .sct 文件的名称必须与工程名称相同。

若上述方法失败，可打开一个记事本文件，键入下段文字。保存并把文件名改为<PROJECT_NAME>.sct，移入工程文件下.\rvmdk 文件夹下。并在上述设置框中使用“...”按钮选中该 .sct 文件。

```
; *****
; *** Scatter-Loading Description File generated by uVision ***
; *****

LR_IROM1 0x00000000 0x00040000 {      ; load region size_region
  ER_IROM1 0x00000000 0x00040000 {      ; load address = execution address
    *.o (RESET, +First)
    *(InRoot$$Sections)
    .ANY (+RO)
  }
  RW_IRAM1 0x20000000 0x00018000 {      ; RW data
    .ANY (+RW +ZI)
  }
}
```

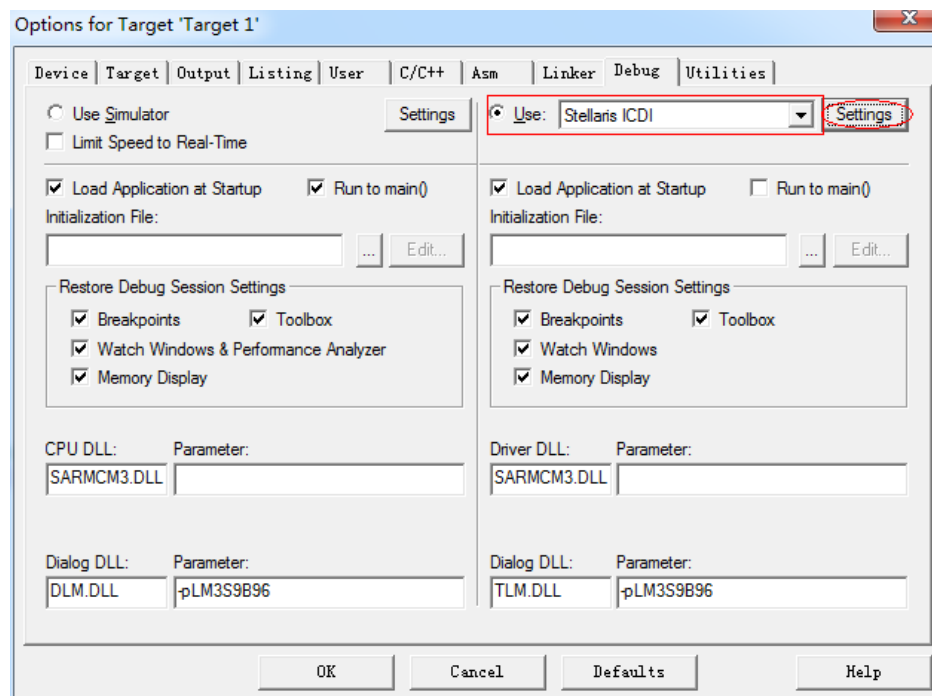


图 A.14

点击 Settings，弹出下对话框

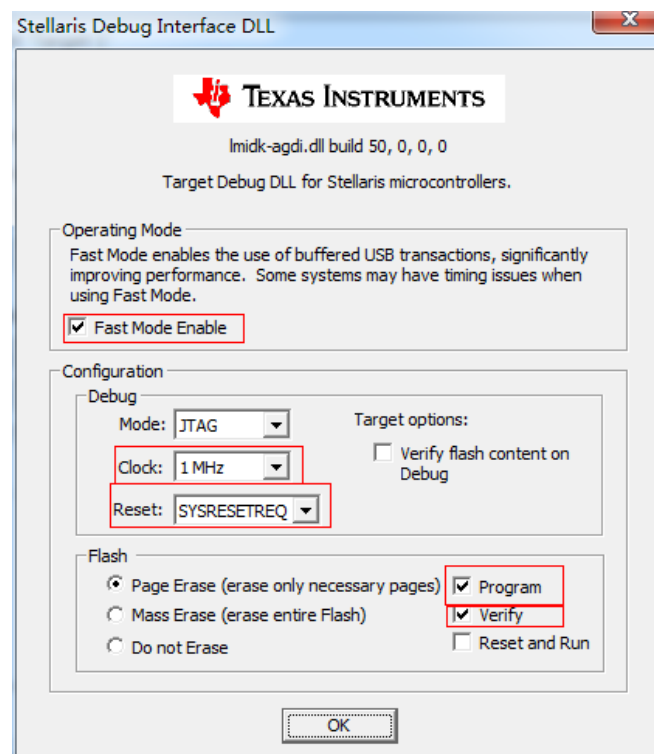


图 A.15

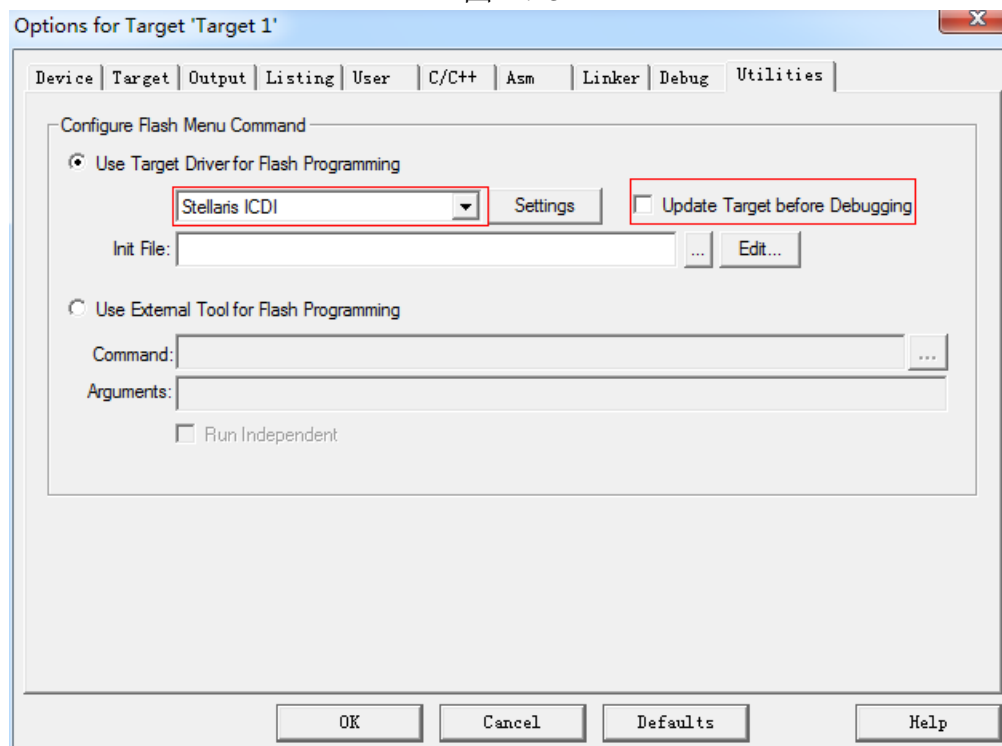


图 A.16

■ Step7: 编译与下载

完成上述设置后，分布点击下图中红圈和蓝圈中的按钮编译主函数并建立



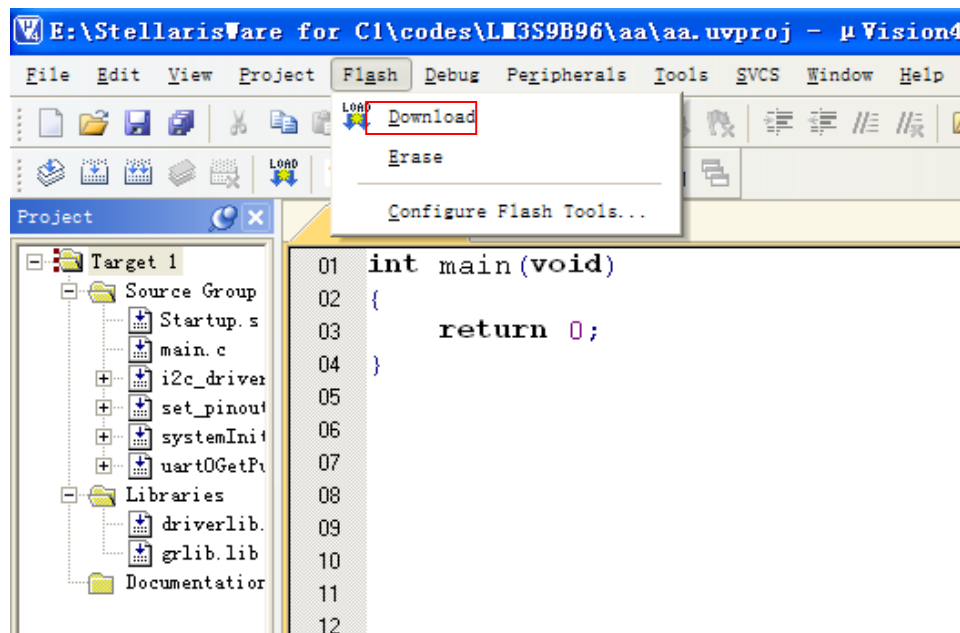
连接

若不是第一次编译，可点击蓝圈右边的 Rebuild 按钮重建。

```
Build Output
..\drivers\i2c_driver.c(15): warning: #177-D: variable "i
compiling set_pinout.c...
..\drivers\set_pinout.c(241): warning: #111-D: statement
compiling systemInit.c...
compiling uartOGetPut.c...
..\drivers\uartOGetPut.c(59): warning: #223-D: function "
..\drivers\uartOGetPut.c(60): warning: #223-D: function "
..\drivers\uartOGetPut.c(14): warning: #550-D: variable "
linking...
Program Size: Code=360 RO-data=16 RW-data=0 ZI-data=256
".\rvmdk\aa.axf" - 0 Error(s), 10 Warning(s).
```

上图最后一句，看到*.axf 文件的生成表示我们已经编译成功了。这个文件包含了我们要烧写到目标板内的所有数据。

最后点击 LOAD 按钮完成下载，注意此时需打开实验板电源。



```
Load "E:\\StellarisWare for C1\\codes\\LM3S9B96\\aa\\rvmdk\\aa.AXF"
Connecting: Mode=JTAG, Speed=1000000Hz
Erase Done.
Programming Done.
Verify OK.
```

看到“Verify OK”表示烧写已经完成了。

附录 B S700 实验板硬件电路使用说明

B.1 硬件使用注意事项

Cortex-M3 属于 ARM 处理器，乃静电敏感器件。使用时需注意不要用手触摸其表面或引脚（特别是上电以后），另外板上的三块晶振（图 0-2 中白框部分）在上电以后也严禁触碰，在操作跳帽和按钮时要特别注意。

如果需要触摸电路板或重新跳帽，请先断开电源，然后用手触碰机箱壳，金属窗框等，以释放人体静电。

B.2 供电

一般情况下使用右下角的Mini USB接口，兼有供电和烧写两大功能。

使用方法：保证开关（SW3）切至OFF档，将USB线缆大头端插入到电脑的USB插口，并将USB线缆的小头端插入到开发板的USB接口（J10），并确认跳线JP19已经用跳线帽连好，JP17和JP18处于断开状态。确认无误后，将开关（SW3）拨至开状态，实验板就可以上电运行了。实验板正常运行时，D11和D12指示灯会常亮。

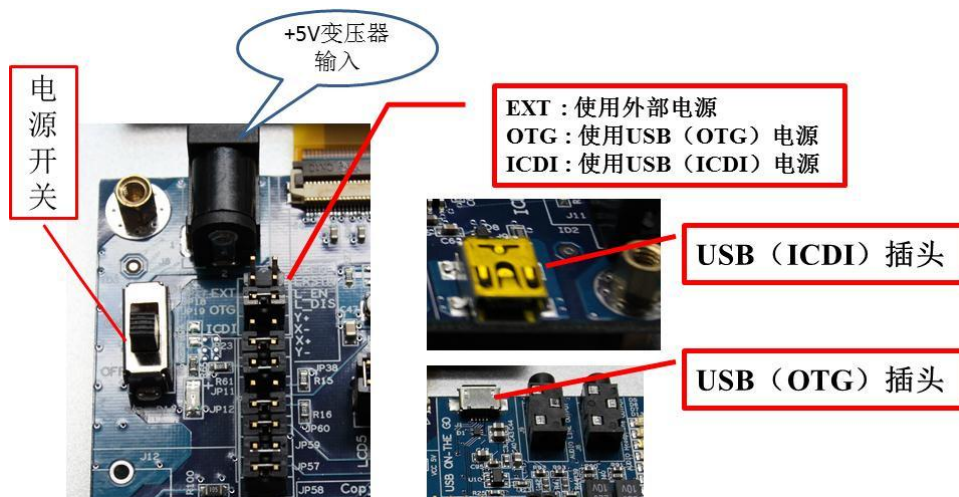


图 B.1 相关电源插头位置图

【补充说明】S700实验板的供电方式：S700供电为**直流5V**，共有三种供电模式：EXT(外部供电)、ICDI、OTG。

1. 如果采用外部5V 电源（J5）供电，需要连接跳线JP17（断开JP18、JP19）；
2. 如果采用 ICDI（J10 的Mini USB接口）供电，需要连接跳线JP19（断开JP17、JP18）；
3. 如果采用 OTG（J1 的Micro USB接口）供电，需要连接跳线JP18（断开JP17、JP19）。

此外，如果需要保证所有芯片的正常上电，需连接JP49或JP50。

注意事项： 1. 同一时刻只能用一种方式供电，故 J17、J18、J19 中只能有一个连接。

2. USB(OTG)接口容易脱落，插接时需要小心。

B.3 外设

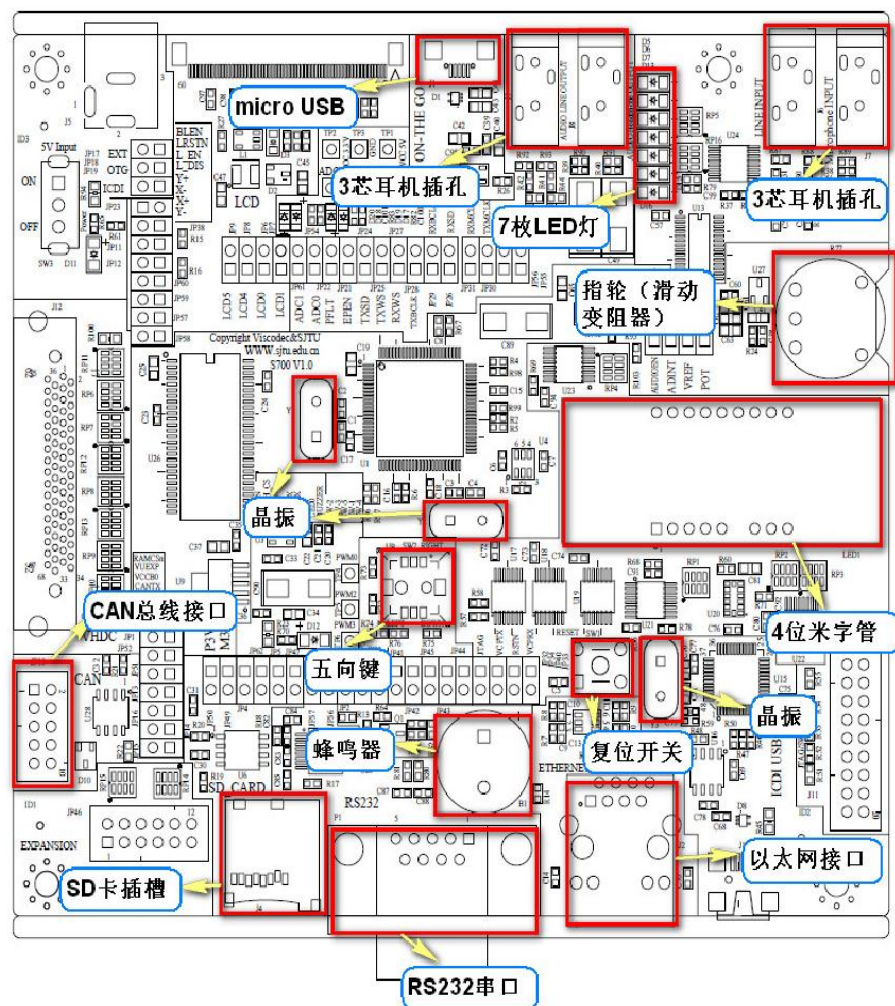


图 B.2 S700 实验板结构板图

程序设计中可涉及的外设如上图红框中所示，列表清单如下：

0. 主控制器重启按钮 (Reset)
1. 4 位米字管 S05441A
2. 指轮 Thumbwheel (滑动变阻器)
3. 蜂鸣器
4. 五向键 (上下左右相对板子轴线偏移 45 度)
5. SD 卡插槽
6. RS232 串口
7. Micro USB (可使用 NOKIA 连接线)
8. 3 芯耳机插孔 (音频及视频输出)
9. 3 芯耳机插孔 (音频及视频输入)
10. 以太网接口 (接标准网线)
11. 触摸式液晶屏
12. 7 枚贴片 LED 灯
13. CAN 总线接口

另外还有两个可扩展接口，一个电源接口和一个烧写接口。

【补充说明】

1. 上述外设中有一些共用接口，故不能同时使用
2. 米字管与贴片 LED 灯皆用 I2C 总线连接，不可用 GPIO 直接控制
3. 以太网接口中嵌入了两枚独立 LED 灯，可单独用 GPIO 控制。

B.4 跳帽

为保证该实验板的通用性，板上设置 60 个跳帽，可根据需要对芯片连接的外设线路作调整。

板上绘有所有跳帽的号码和名称，然字体太小，可参见《S700 实验板电路元件标绘图》。下表整理出了 S700 实验板跳线功能说明，相同颜色代表同种外设。

表 B.1

S700 实验板跳线功能说明			
跳线编号	字符说明	功能说明	冲突的跳线
JP1	RAMCSn	SDRAM 的片选信号	
JP2	LED0	网络灯 LED0	JP39
JP3	LED1	网络灯 LED1	JP35
JP4	SPICSn	通信扩展口预留的 spi 片选信号	JP5、JP62(spiflash 片选,两者不能同时工作)
JP5	SDCSn	SD 卡的片选	JP4、JP62(spiflash 片选,两者不能同时工作)
JP6	LCD0	LCD 的数据线 D0	JP29、JP36
JP7	LCD1	LCD 的数据线 D1	JP27、JP37
JP8	LCD4	LCD 的数据线 D4	JP26
JP9	LCD5	LCD 的数据线 D5	JP31
JP11	L_EN	LCD 开关, 连接表示拉低使能 LCD	
JP12	L_DIS	LCD 开关, 连接表示拉高禁止 LCD	
JP13	CANTX	CAN 总线 TX	JP22
JP14	CANRX	CAN 总线 RX	JP21
JP15	TERM	CAN 总线扩展口相关跳线	
JP16	+VCAN	CAN 总线扩展口的+5V 信号	
JP17	EXT	5V 供电选择: 外部供电	
JP18	OTG	5V 供电选择: OTG 供电	
JP19	ICDI	5V 供电选择: ICDI 供电	
JP20	VREF	ADC 采用的参考电压+3V	JP28、JP43
JP21	EPEN	OTG-USB 的 EPEN 信号	JP14
JP22	PFLT	OTG-USB 的 PFLT 信号	JP13
JP23	BLN	LCD 的背光开关, 断开表	

		示使能	
JP24	TXSD		JP41
JP25	TXWS		JP42
JP26	RXSD		JP8
JP27	RXWS		JP7、JP37
JP28	TXBCLK		JP20、JP43
JP29	RXBCLK		JP6、JP36
JP30	TXMCLK		JP44
JP31	RXMCLK		JP9
JP32	JTAG	JTAG 接口的 3.3V 供电	
JP33	VCPRX	ICDI 接口的虚拟串口信号	
JP34	VCPTX	ICDI 接口的虚拟串口信号	
JP35	ADINT	3D 加速传感器的中断信号	JP3
JP36	232_RX	RS232 接口信号	JP6、JP29
JP37	232_TX	RS232 接口信号	JP7、JP27
JP38	LRSTN	LCD 复位信号	
JP39	BUZZER	输出给蜂鸣器的 PWM 信号	JP2
JP40	RSTN	ICDI 接口的复位信号	
JP41	SW_2	五向键的某个 GPIO 引脚	JP24
JP42	SW_3	五向键的某个 GPIO 引脚	JP25
JP43	SW_1	五向键的某个 GPIO 引脚	JP20、JP28
JP44	SW_4	五向键的某个 GPIO 引脚	JP30
JP45	SW_5	五向键的某个 GPIO 引脚	JP48
JP47	CARD	SD 卡的相关信号	
JP48	POT	ADC 采集电压输入引脚	JP45
JP49	M3V	给系统供电 3.3V	
JP50	P3V	给系统供电 3.3V	
JP51	VUEXP	VHDC 接口的+5V	
JP52	VCCB0	VHDC 接口的+3.3V	
JP53	AUDIOEN	音频的使能，音频芯片的 3.3V 供电	
JP54	ADC0	ADC0 测试点的输入	JP60
JP55	SDA	连接音频芯片 i2c 的 SDA	
JP56	SCL	连接音频芯片 i2c 的 SCL	
JP57	X+	触摸屏模拟信号	JP61
JP58	Y-	触摸屏模拟信号	
JP59	X-	触摸屏模拟信号	
JP60	Y+	触摸屏模拟信号	JP54
JP61	ADC1	ADC1 测试点的输入	JP57

JP62	FLASH_CS	spiFlash 的片选信号	JP4、JP5
------	----------	----------------	---------

板上一根固定 I2C 总线连接 7 枚 LED 灯，米字管和加速度传感器，无需跳帽设置。

下表列出各外设工作时需要连接和断开的跳帽：

表 B.2

外设	连接	断开
加速度传感器	JP35	JP2
指轮	JP48	JP45
音频处理	JP24~JP31, JP53, JP54	JP41~JP44
LCD 触摸屏	JP6~JP9, JP11, JP38	JP12, JP26, JP27, JP29, JP31, JP36, JP37
SPI Flash	-	JP5
蜂鸣器	JP39	JP2
网络灯 LED0	JP2	JP39
网络灯 LED1	JP3	JP35
五向键	JP41~JP45	JP24, JP25, JP28, JP30, JP48
Ethernet 以太网口	-	-
Micro USB	JP21, JP22	JP13, JP14
SD RAM	JP1	-
RS232	JP36, JP37	JP6, JP7, JP27, JP29
SD CARD	JP5, JP47	JP4, JP62
CAN 总线	JP13~16	JP21, JP22