目 录

第1章 LPC2103 简介

1.1 概述

LPC2103 是一个基于支持实时仿真的 16/32 位 ARM7 TDMI-S CPU 的微控制器，并带有 32kB 的嵌入高速 Flash 存储器，128 位宽度的存储器接口和独特的加速结构使 32 位代码能够在最大时钟速率下运行。

较小的封装和极低的功耗使 LPC2103 适用于访问控制器和 POS 机等小型应用系统中；由于内置了宽范围的串行通信接口（2 个 UART、SPI、SSP 和 2 个 I^2C）和 8KB 的片内 SRAM，LPC2103 也适合用在通信网关和协议转换器中。32/16 位定时器、增强型 10 位 ADC、定时器输出匹配 PWM 特性、多达 13 个边沿、电平触发的外部中断、32 条高速 GPIO，使得 LPC2103 微控制器特别适用于工业控制和医疗系统中。

1.2 LPC2103 特性

- 16/32 位 ARM7 TDMI-S 微控制器，超小 LQFP48 封装；
- 8KB 的片内静态 RAM 和 32KB 的片内 Flash 程序存储器。128 位宽度接口/加速器可实现高达 70 MHz 工作频率；
- 通过片内 boot 装载程序实现在系统/在应用编程（ISP/IAP）。单个 Flash 扇区或整片擦除时间为 100ms，256 字节编程时间为 1ms；
- 嵌入式 ICE RT 通过片内 RealMonitor 软件提供实时调试；
- 10 位 A/D 转换器提供 8 路模拟输入（每个通道的转换时间低至 2.44us），以及特定的结果寄存器来最大限度地减少中断开销；
- 2 个 32 位定时器/外部事件计数器（带 7 路捕获和 7 路比较通道）；
- 2 个 16 位定时器/外部事件计数器（带 3 路捕获和 7 路比较通道）；
- 低功耗实时时钟（RTC）具有独立的电源和特定的 32KHz 时钟输入；
- 多个串行接口，包括 2 个 UART（16C550 协议标准）、2 个高速 I^2C 总线（400 Kbit/s）、SPI 和具有缓冲作用和数据长度可变功能的 SSP；
- 向量中断控制器（VIC），可配置优先级和向量地址；
- 多达 32 个通用 I/O 口（可承受 5V 电压）；
- 多达 13 个边沿、电平触发的外部中断管脚；
- 通过一个可编程的片内 PLL（100us 的设置时间）可实现最大为 70MHz 的 CPU 操作频率，其具有 10MHz～25MHz 的输入频率；
- 片内集成振荡器与外部晶体的操作频率范围为 1～25MHz；
- 低功耗模式包括空闲模式、带 RTC 的睡眠模式和掉电模式；
- 可通过个别使能/禁止外围功能和外围时钟分频来优化额外功耗；
- 通过外部中断或 RTC 将处理器从掉电模式中唤醒。

1.3 管脚信息

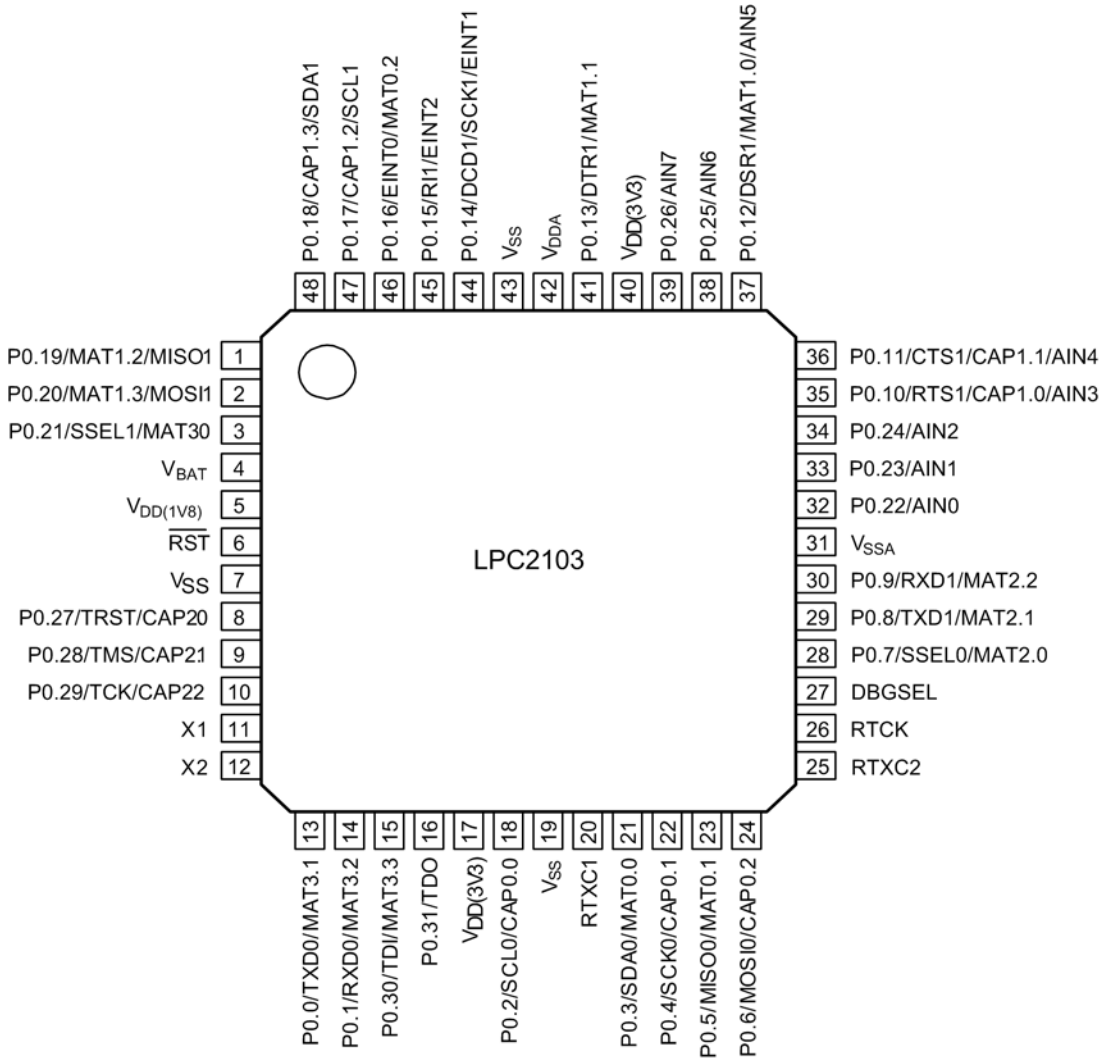


图 1.1 LQFP48 管脚配置

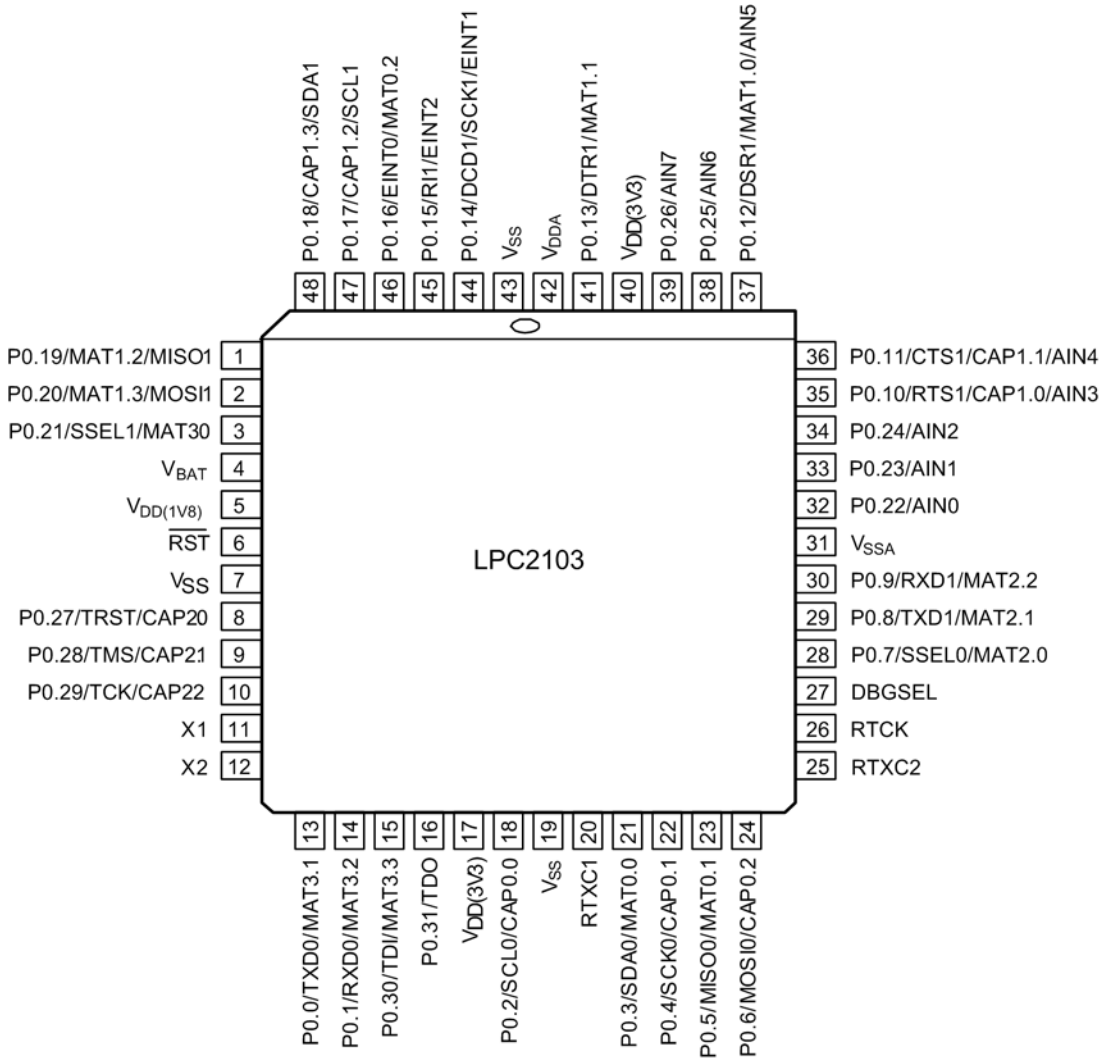


图 1.2 PLCC44 管脚配置

LPC2103 的管脚描述及其主要功能如表 1.1所示。

表 1.1 管脚描述

管脚名称	LQFP48	PLCC44	类型	描 述
P0.0~P0.31			I/O	P0 口：P0 口是一个 32 位 I/O 口。每个位都有独立的方向控制。有 31 个 P0 口可用作通用双向数字 I/O 口，P0.31 只用作输出口。P0 口管脚的操作取决于管脚连接模块所选择的功能
P0.0/ TXD0/ MAT3.1	13 [1]	18 [1]	I/O O O	P0.0－通用输入/输出数字管脚（GPIO） TXD0－UART0 的发送器输出 MAT3.1－定时器 3 的 PWM 输出 1
P0.1/ RXD0/ MAT3.2	14 [2]	19 [2]	I/O I O	P0.1－通用输入/输出数字管脚（GPIO） RXD0－UART0 的接收器输入 MAT3.2－定时器 3 的 PWM 输出 2
P0.2/ SCL0/ CAP0.0	18 [3]	22 [3]	I/O I/O I	P0.2－通用输入/输出数字管脚（GPIO） SCL0－I^2C0 时钟输入/输出。开漏输出（符合 I^2C 规范） CAP0.0－定时器 0 捕获输入 0

续上表

管脚名称	LQFP48	PLCC44	类型	描 述
P0.3/ SDA0/ MAT0.0	21 [3]	25 [3]	I/O I/O O	P0.3－通用输入/输出数字管脚（GPIO） SDA0－I^2C0 数据输入/输出。开漏输出（符合 I^2C 规范） MAT0.0－定时器 0 的 PWM 输出 0
P0.4/ SCK0/ CAP0.1	22 [4]	26 [4]	I/O I/O I	P0.4－通用输入/输出数字管脚（GPIO） SCK0－SPI0 串行时钟，主机输出或从机输入的时钟 CAP0.1－定时器 0 捕获输入 1
P0.5/ MISO0/ MAT0.1	23 [4]	27 [4]	I/O I/O O	P0.5－通用输入/输出数字管脚（GPIO） MISO0－SPI0 主机输入/从机输出，从机到主机的数据传输 MAT0.1－定时器 0 的 PWM 输出 1
P0.6/ MOSI0/ CAP0.2	24 [4]	28 [4]	I/O I/O I	P0.6－通用输入/输出数字管脚（GPIO） MOSI0－SPI0 主机输出/从机输入，主机到从机的数据传输 CAP0.2－定时器 0 捕获输入 2
P0.7/ SSEL0/ MAT2.0	28 [2]	31 [2]	I/O I O	P0.7－通用输入/输出数字管脚（GPIO） SSEL0－SPI0 从机选择，选择 SPI 接口用作从机。 MAT2.0－定时器 2 的 PWM 输出，通道 0
P0.8/ TXD1/ MAT2.1	29 [4]	32 [4]	I/O O O	P0.8－通用输入/输出数字管脚（GPIO） TXD1－UART1 的发送器输出 MAT2.1－定时器 2 的 PWM 输出，通道 1
P0.9/ RXD1/ MAT2.2	30 [2]	33 [2]	I/O I O	P0.9－通用输入/输出数字管脚（GPIO） RxD1－UART1 的接收器输入 MAT2.2－定时器 2 的 PWM 输出，通道 2
P0.10/ RTS1/ CAP1.0/ AIN3	35 [4]	38 [4]	I/O O I I	P0.10－通用输入/输出数字管脚（GPIO） RTS1－UART1 请求发送输出。 CAP1.0－定时器 1 捕获输入 0 AIN3－模拟输入 3
P0.11/ CTS1/ CAP1.1/ AIN4	36 [3]	39 [3]	I/O I I I	P0.11－通用输入/输出数字管脚（GPIO） CTS1－UART1 的清零发送输入 CAP1.1－定时器 1 捕获输入 1 AIN4－模拟输入 4
P0.12/ DSR1/ MAT1.0/ AIN5	37 [4]	40 [4]	I/O I O I	P0.12－通用输入/输出数字管脚（GPIO） DSR1－UART1 的数据设置就绪输入 MAT1.0－定时器 1 的 PWM 输出 0 AIN5－模拟输入 5
P0.13/ DTR1/ MAT1.1	41 [4]	43 [4]	I/O O O	P0.13－通用输入/输出数字管脚（GPIO） DTR1－UART1 的数据终端就绪输出 MAT1.1－定时器 1 的 PWM 输出 1
P0.14/ DCD1/ SCK1/ EINT1	44 [3]	2 [3]	I/O I I/O I	P0.14－通用输入/输出数字管脚（GPIO） DCD1－UART1 数据载波检测输入 SCK1－SPI1 的串行时钟。SPI 时钟的主机输出或从机输入 EINT1－外部中断 1 输入

续上表

管脚名称	LQFP48	PLCC44	类型	描 述
P0.15/ RI1/ EINT2	45 [4]	3 [4]	I/O I I	P0.15－通用输入/输出数字管脚（GPIO） RI1－UART1 铃声指示输入 EINT2－外部中断 2 输入
P0.16/ EINT0/ MAT0.2	46 [2]	4 [2]	I/O I O	P0.16－通用输入/输出数字管脚（GPIO） EINT0－外部中断 0 输入 MAT0.2－定时器 0 的 PWM 输出 2
P0.17/ CAP1.2/ SCL1	47 [1]	5 [1]	I/O I I/O	P0.17－通用输入/输出数字管脚（GPIO） CAP1.2－定时器 1 捕获输入 2 SCL1－I^2C1 时钟输入/输出。开漏输出（遵循 I^2C 总线规范）
P0.18/ CAP1.3/ SDA1	48 [1]	6 [1]	I/O I I/O	P0.18－通用输入/输出数字管脚（GPIO） CAP1.3－定时器 1 捕获输入 3 SDA1－I^2C1 数据输入/输出。开漏输出（遵循 I^2C 总线规范）
P0.19/ MAT1.2/ MISO1	1 [1]	7 [1]	I/O O I/O	P0.19－通用输入/输出数字管脚（GPIO） MAT1.2－定时器 1 的 PWM 输出 2 MISO1－SSP 主机输出/从机输入，主机到从机的数据传输
P0.20/ MAT1.3/ MOSI1	2 [2]	8 [2]	I/O O I/O	P0.20－通用输入/输出数字管脚（GPIO） MAT1.3－定时器 1 的 PWM 输出 3 MOSI1－SSP 主机输出/从机输入，主机到从机的数据传输
P0.21/ SSEL1/ MAT3.0	3 [4]	9 [4]	I/O I O	P0.21－通用输入/输出数字管脚（GPIO） SSEL1－SPI1 的从机选择。选择 SPI 接口作为从机。 MAT3.0－定时器 3 的 PWM 输出，通道 0
P0.22/ AIN0	32 [4]	35 [4]	I/O I	P0.22－通用输入/输出数字管脚（GPIO） AIN0－模拟输入 0
P0.23/ AIN1	33 [1]	36 [1]	I/O I	P0.23－通用输入/输出数字管脚（GPIO） AIN1－模拟输入 1
P0.24/ AIN2	34 [1]	37 [1]	I/O I	P0.24－通用输入/输出数字管脚（GPIO） AIN2－模拟输入 2
P0.25/ AIN6	38 [1]	41 [1]	I/O I	P0.25－通用输入/输出数字管脚（GPIO） AIN6－模拟输入 6
P0.26/ AIN7	39 [1]	n.c	I/O I	P0.26－通用输入/输出数字管脚（GPIO） AIN7－模拟输入 7
P0.27/ TRST/ CAP2.0	8 [4]	13 [4]	I/O I I	P0.27－通用输入/输出数字管脚（GPIO） TRST－JTAG 接口的测试复位 CAP2.0－定时器 2 捕获输入，通道 0
P0.28/ TMS/ CAP2.1	9 [4]	14 [4]	I/O I I	P0.28－通用输入/输出数字管脚（GPIO） TMS－JTAG 接口的测试模式选择 CAP2.1－定时器 2 捕获输入，通道 1
P0.29/ TCK/ CAP2.2	10 [4]	15 [4]	I/O I I	P0.29－通用输入/输出数字管脚（GPIO） TCK－JTAG 接口的测试时钟 CAP2.2－定时器 2 捕获输入，通道 2

续上表

管脚名称	LQFP48	PLCC44	类型	描 述
P0.30/ TDI/ MAT3.3	15 [4]	20 [4]	I/O I O	P0.30－通用输入/输出数字管脚（GPIO） TDI－JTAG 接口的测试数据输入 MAT3.3－定时器 3 的 PWM 输出 3
P0.31/ TDO	16 [4]	21 [4]	O O	P0.31－通用仅为输出的数字管脚（GPO） TDO－JTAG 接口的测试数据输出
RTXC1	20 [5]	24 [5]	I	RTC 振荡器电路的输入
RTXC2	25 [5]	29 [5]	O	RTC 振荡器电路的输出
RTCK	26 [5]	n.c	I/O	返回的测试时钟输出：JTAG 端口的额外信号。当处理器频率变化时帮助调试器保持同步。带内部上拉的双向口
X1	11	16	I	振荡器电路和内部时钟发生器电路的输入
X2	12	17	O	振荡放大器的输出
DBGSEL	27	30	I	调试选择：当该管脚为低电平时，器件正常操作。当该管脚为高电平时，进入调试模式。输入内部拉低
$\overline{RST}$	6	11	I	外部复位输入：该管脚的低电平将器件复位，并使 I/O 口和外围功能恢复默认状态，处理器从地址 0 开始执行。带迟滞的 TTL 电平，管脚可承受 5V 电压
V_{SS}	7，19，43	1，12，23	I	地：0V 参考点
V_{SSA}	31	34	I	模拟地：0V 参考点。标称电压与 V_{SS} 相同，但应当互相隔离以减少噪声和故障
V_{DDA}	42	44	I	模拟 3.3V 端口电源：标称电压与 $V_{DD(3V3)}$ 相同，但应当互相隔离以减少噪声和故障。该电压也用来向片内 PLL 供电
$V_{DD(1V8)}$	5	10	I	1.8V 内核电源：这是内部电路的电源电压
$V_{DD(3V3)}$	17，40	42	I	3.3V 端口电源：这是 I/O 口的电源电压
V_{BAT}	4	n.c	I	RTC 电源：RTC 的 3.3V 电源端

[1] 5V 电压容限端口提供带 TTL 电平、滞后和 10ns 转换速率控制的数字 I/O 功能；

[2] 5V 电压容限端口提供带 TTL 电平、滞后和 10ns 转换速率控制的数字 I/O 功能。如果配置为输入功能，该端口利用内置的干扰滤波器阻止短于 3ns 的脉冲；

[3] 可承受开漏 5V 电压的数字 I/O I^2C 总线 400kHz 规格的兼容端口。它需要外部上拉提供多种用途的输出；

[4] 5V 电压容限端口提供数字 I/O（带 TTL 电平、滞后和 10ns 转换速率控制）和模拟输入功能。如果配置为输入功能，该端口利用内置的干扰滤波器阻止短于 3ns 的脉冲。当配置为 ADC 输入时，端口的数字部分禁能；

[5] 端口提供特殊的模拟功能。

1.4 器件信息

表 1.2 LPC2103 器件信息

器件	管脚数	片内 SRAM	片内 FLASH	ADC 通道
LPC2103	48	8 KB	32 KB	8 输入

第2章 EasyARM2103 硬件说明

EasyARM2103 开发板是广州周立功单片机发展有限公司设计的 EasyARM 系列开发套件之一，它采用了 NXP 公司基于 ARM7 TDMI-S 核、LQFP48 封装的 LPC2103 芯片，具有 JTAG 仿真调试和 ISP 编程功能。

EasyARM2103 开发板上提供了按键、发光二极管等常用的功能器件，具有 RS-232 接口电路和 I^2C 存储器电路。用户可以更换兼容的 CPU 进行仿真调试，如 LPC2101 和 LPC2102 芯片等。开发板上所有的 I/O 口全部引出，灵活的跳线组合，极大的方便用户进行 32 位 ARM 嵌入式系统的开发实验。

2.1 功能特点

EasyARM2103 开发板的功能特点如下：

- 采用“底板＋PACK 板”的形式构成 EasyARM2103 开发套件，PACK 板的主芯片为 LPC2103；
- 板上所有的功能器件与 LPC2103 的引脚可通过跳线来连接；
- 配套有详细的开发板实验教程；
- I/O 口全部引出，方便用户连接外部电路进行开发；
- 可进行 GPIO 的输入输出实验，如按键输入、发光二极管输出等；
- 按键、发光二极管分别可用于外部中断、GPIO 输出等；
- 具有 RS-232 转换电路，可与上位机进行通信，完成 UART 通信实验；
- 具有 I^2C 接口和 SPI/SSP 接口输出；
- 提供基于 PC 的人机界面，方便调试实时时钟和串口通信等；
- 可进行外部中断实验，学习向量中断控制器（VIC）；
- 定时器实验，如定时输出和定时器捕获等；
- 复位芯片 CAT1025，完成 I^2C 总线实验；
- A/D 转换实验；
- 实时时钟控制实验；
- WDT 看门狗实验；
- 详细的实验教程和实验例程完整地验证了芯片的所有功能外设。

2.2 硬件原理

2.2.1 LPC2103 PACK板原理图

LPC2103 PACK板的电路设计如图 2.1所示，芯片LPC2103 的所有引脚全部连接到插针上，电容C1-C5 用于对LPC2103 芯片上 5 个电源端进行滤波。

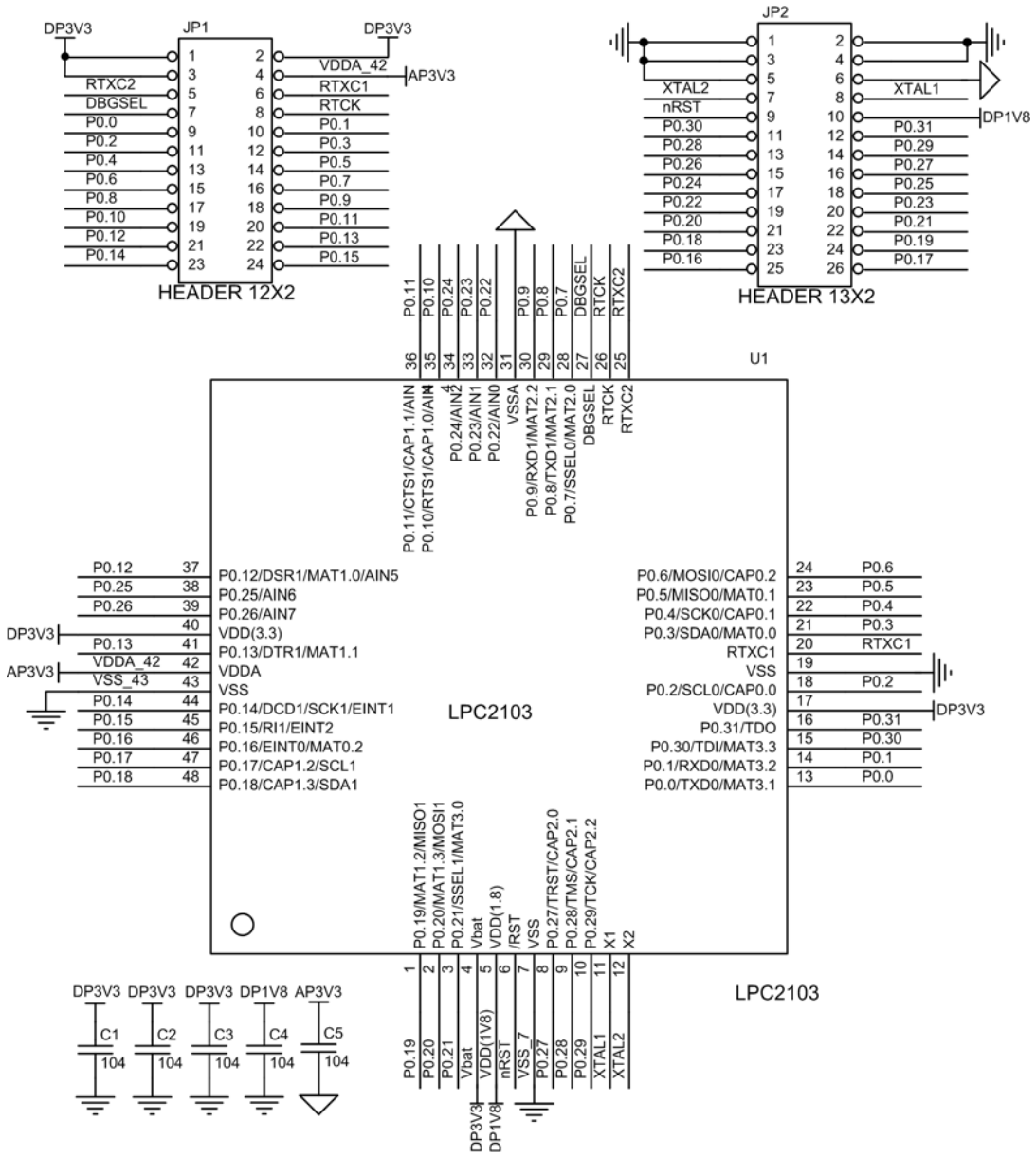


图 2.1 LPC2103 PACK 原理图

2.2.2 EasyARM2103 底板原理图

1. 电源电路

LPC2103 控制器需要双电源供电，1.8V内核电压和 3.3V功能外设电压，EasyARM2103 开发板的电源电路如图 2.2所示。5V电源由变压器或USB电源线输入，二极管 1N5819 可以防止电源反接。电路采用SPX1117 系列LDO芯片SPX1117M-3.3 和SPX1117M-1.8 将电压稳至 3.3V和 1.8V，0 欧的电阻用来隔离数字电源和模拟电源、数字地和模拟地。

SPX1117 系列 LDO 芯片是 EXAR 公司生产的低压差芯片，其特点是输出电流大，输出电压精度高，稳定性高，宽电压输入（这里选择的是 5V 输入）。此系列 LDO 芯片的输出电流高达 800mA，输出电压的精度在±1%之内，可广泛应用于手持式仪表、数字家电、工业控制等领域。注意在电路输入输出端接一个 10μF 的电容，可改善瞬态响应和稳定性。

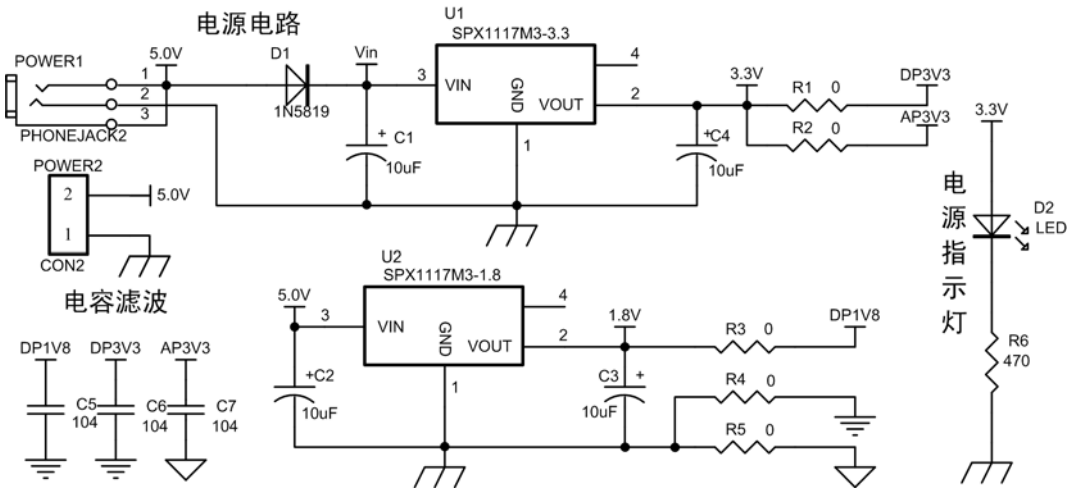


图 2.2 电源电路

2. 复位和I²C电路

由于ARM芯片的高速、低功耗和低工作电压的工作特性，导致其噪声容限低，对电源纹波、瞬态响应性能、时钟源的稳定性和电源监控的可靠性等诸多方面有很高的要求。开发板采用的是带有 256 字节存储空间、I²C接口的专用电源监控复位芯片CAT1025，保证了系统的可靠性，其电路原理如图 2.3所示。

nRST 连接到芯片 LPC2103 的复位引脚，当复位按键 RST1 按下时，CAT1025 的复位引脚输出有效信号，使芯片 LPC2103 复位。I²C 总线需要外接上拉电阻 R15、R16 。

注意：使用 CAT1025 芯片时，其 $\overline{\text{RESET}}$ 引脚上的上拉电阻 R14 是不能省略的。

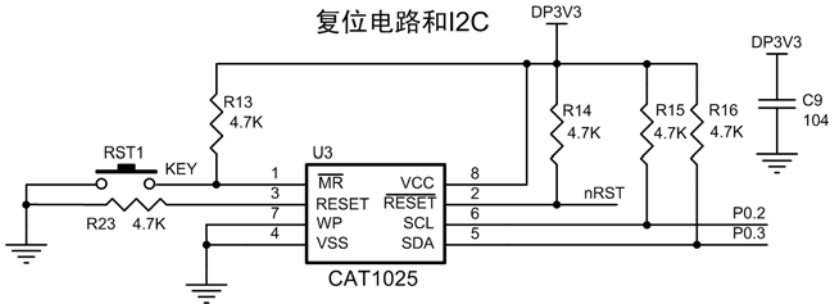


图 2.3 复位电路

3. 时钟电路

LPC2103 微控制器可使用外部晶振或外部时钟源，内部 PLL 电路可调整系统时钟，使系统运行速度更快（CPU 的操作频率最大可达 70MHz）。若不使用片内 PLL 功能及 ISP 下载功能，则外部晶振频率为 1～30MHz，外部时钟频率为 1～50MHz；若使用片内 PLL 功能或 ISP 下载功能，则外部晶振频率为 10～25MHz，外部时钟频率为 10～25MHz。

EasyARM2103 开发板使用外部晶振 11.0592MHz，实时时钟为 32.768KHz，电路原理如图 2.4所示。用 11.0592MHz的外部晶振使串口的波特率更精确，同时能支持LPC2103 微控制器内部的PLL电路及ISP（在系统编程）功能。

[1]ISP 在系统编程：通过 Boot 装在程序和串口对片内 Flash 存储器进行编程和再编程。

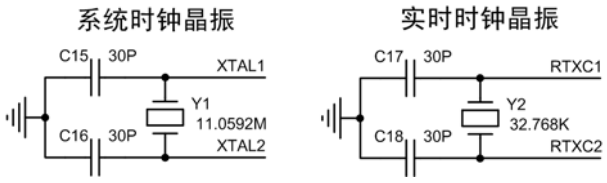


图 2.4 时钟电路

4. JTAG接口电路

JTAG接口电路采用ARM公司提出的标准 20 脚JTAG仿真调试接口，JTAG接口与LPC2103 引脚之间的连接如图 2.5所示。在RTCK引脚处接一个 4.7K的下拉电阻，将在系统复位后使能JTAG调试接口。

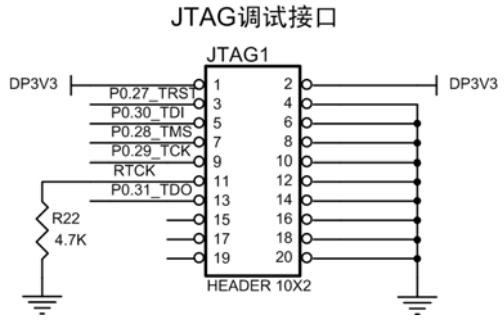


图 2.5 JTAG 接口电路

5. UART接口电路

由于开发板是 3.3V供电系统，需要使用电平转换芯片SP3232E来进行RS-232 电平转换。SP3232E的工作电压为 3.3V，电平转换电路如图 2.6所示。

当使用ISP 功能时，需要将PC 机的串口与开发板的串口相连，短接JP6 端口，短接P0.14，在系统复位时，进入 ISP 状态。同样，在程序仿真调试中，若用到串口 UART0，则需要短接 JP6 两个端口。

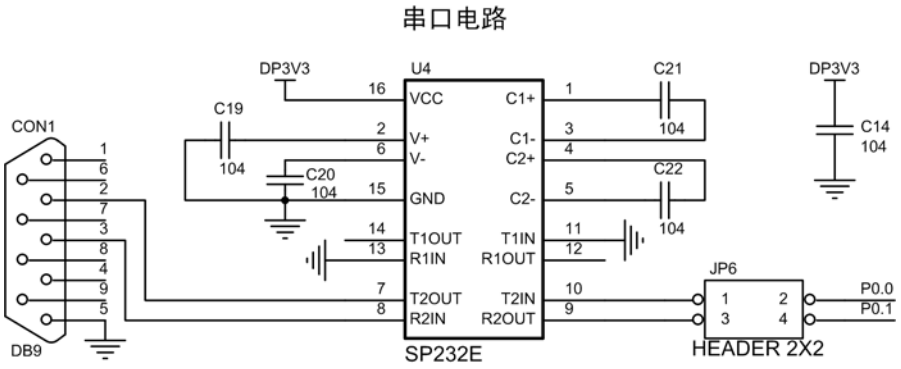


图 2.6 串口电路

6. 按键与显示电路

EasyARM2103 开发板提供了 3 个独立的按键、4 个发光二极管，按键与显示电路如图 2.7所示。当P0 口作为输入时，内部没有上拉电阻，需要外接上拉电阻R17、R18、R19。显

示电路中采用灌电流的驱动方式来驱动发光二极管，由于LPC2103 芯片I/O口提供的灌电流大于其拉电流，可以保证了发光二极管的亮度。

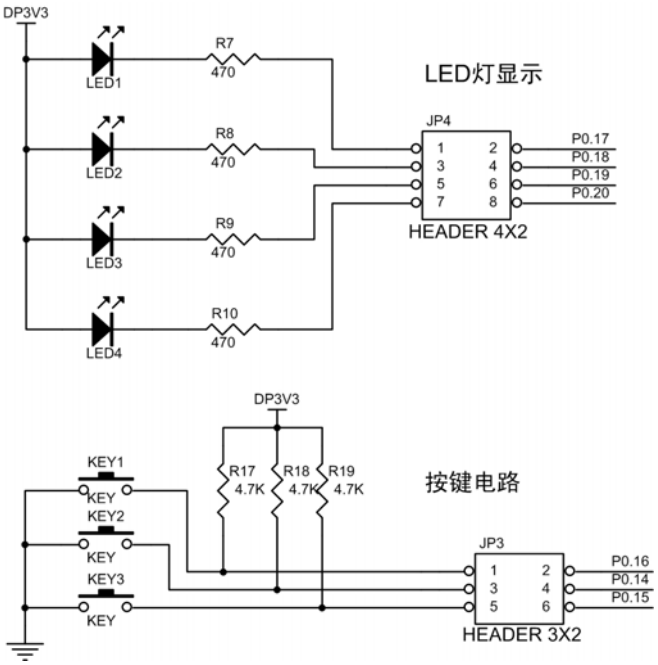


图 2.7 按键与 LED 电路

2.3 硬件结构

2.3.1 元件布局图

EasyARM2103 开发板的底板与PACK板的元件布局如图 2.8、图 2.9所示。

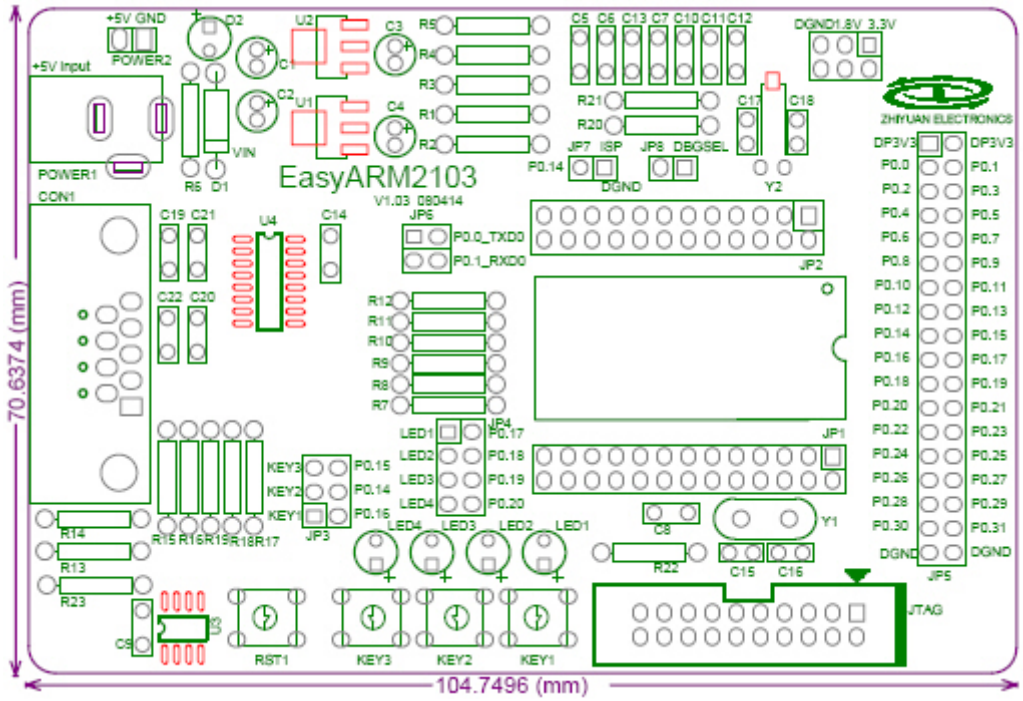


图 2.8 底板元件布局图

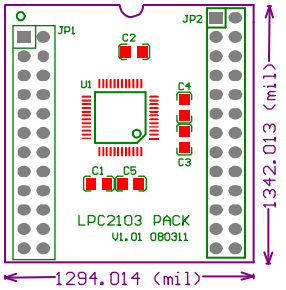


图 2.9 PACK 板元件布局图

2.3.2 开发板使用说明

EasyARM2103 开发板使用说明及注意事项如下：

1. 电源电压

- EasyARM2103 开发板系统电源电压为 5V，注意避免连接高于 5V 电压的电源；
- 注意跳线的连接，避免短路现象。

2. PACK板与JTAG

- 在使用开发板前，必须连接好 PACK 板与底板，注意不要将 PACK 板插反；
- 禁止频繁拔插 PACK，容易造成线路连接不良；
- 当使用 JTAG 调试程序时，需要短接 DBGSEL 引脚，将底板的 JP8 短接；

3. ISP使用

- 当使用 ISP 下载程序时，上电前需要将 P0.14 脚拉低，将底板的 JP7 短接；
- 不使用 ISP 功能时，上电前需要开路 JP7，否则系统无法进入调试状态。

第3章 EasyARM2103 快速入门

本章将首先介绍在 ADS1.2 开发环境里如何建立、编译连接工程及对工程进行调试的基本方法，进而说明基于 LPC2103 ARM 微控制器的工程模板、EasyJTAG-H 仿真器的安装与使用，最后对 ARM 芯片如何进行 ISP 操作进行简单了解。

3.1 ADS 1.2 集成开发环境的组成

ADS 集成开发环境是 ARM 公司推出的 ARM 核微控制器集成开发工具，英文全称为 ARM Developer Suite，成熟版本为 ADS1.2。ADS1.2 支持 ARM10 之前的所有 ARM 系列微控制器，支持软件调试及 JTAG 硬件仿真调试，支持汇编、C、C++源程序，具有编译效率高、系统库功能强等特点，可以在 Windows98、Windows2000、Windows XP 以及 RedHat Linux 上运行。

ADS 1.2 由 6 个部分组成，如表 3.1所示。

表 3.1 ADS 1.2 的组成部分

名称	描述	使用方式
代码生成工具	ARM 汇编器 ARM 的 C、C++编译器 Thumb 的 C、C++编译器 ARM 连接器	由 CodeWarrior IDE 调用
集成开发环境	CodeWarrior IDE	工程管理，编译连接
调试器	AXD ADW/ADU armsd	仿真调试
指令模拟器	ARMulator	由 AXD 调用
ARM 开发包	底层的例程 实用程序(如 fromELF)	实用程序由 CodeWarrior IDE 调用
ARM 应用库	C、C++函数库等	用户程序使用

由于用户一般直接操作的是 CodeWarrior IDE 集成开发环境和 AXD 调试器，所以本文只介绍这两部分的使用，其它部分的详细说明参考 ADS 1.2 的在线帮助文档。

3.1.1 CodeWarrior IDE简介

ADS 1.2 使用了 CodeWarrior IDE 集成开发环境，并集成了 ARM 汇编器、ARM 的 C/C++编译器、Thumb 的 C/C++编译器、ARM 连接器，包含工程管理器、代码生成接口、语法敏感（对关键字以不同颜色显示）编辑器、源文件和类浏览器等。

CodeWarrior IDE主窗口如图 3.1所示。

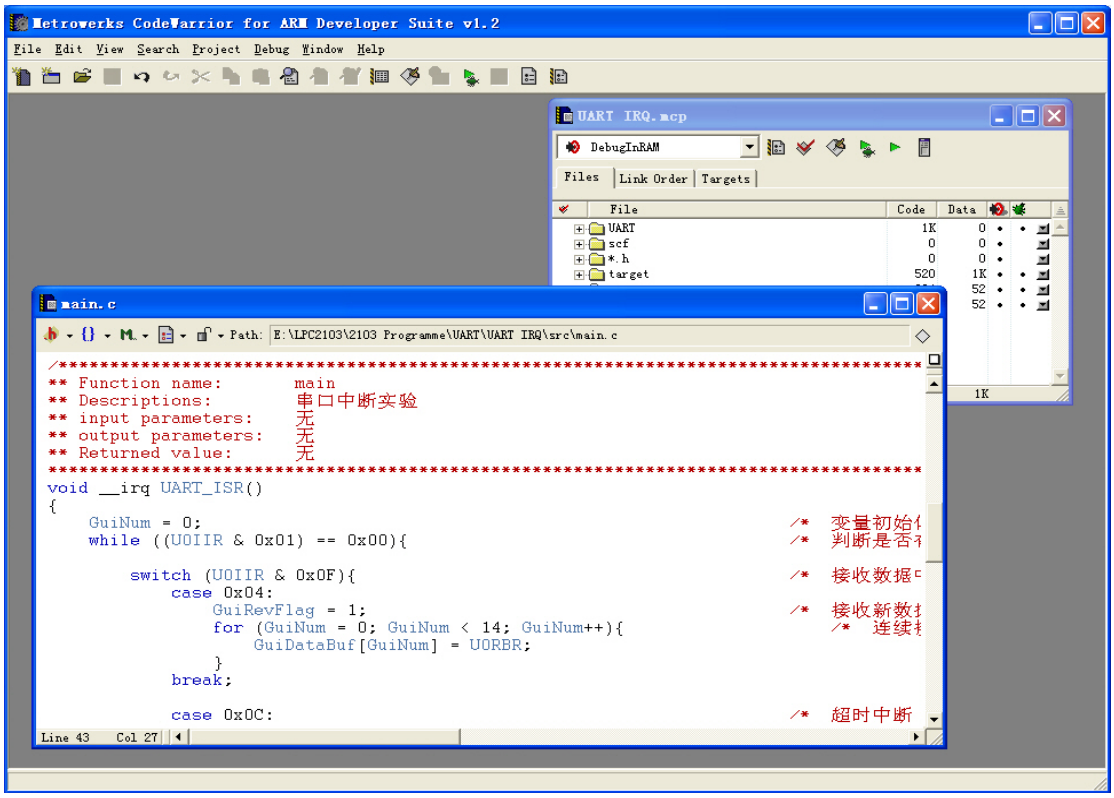


图 3.1 CodeWarrior 开发环境

3.1.2 AXD调试器简介

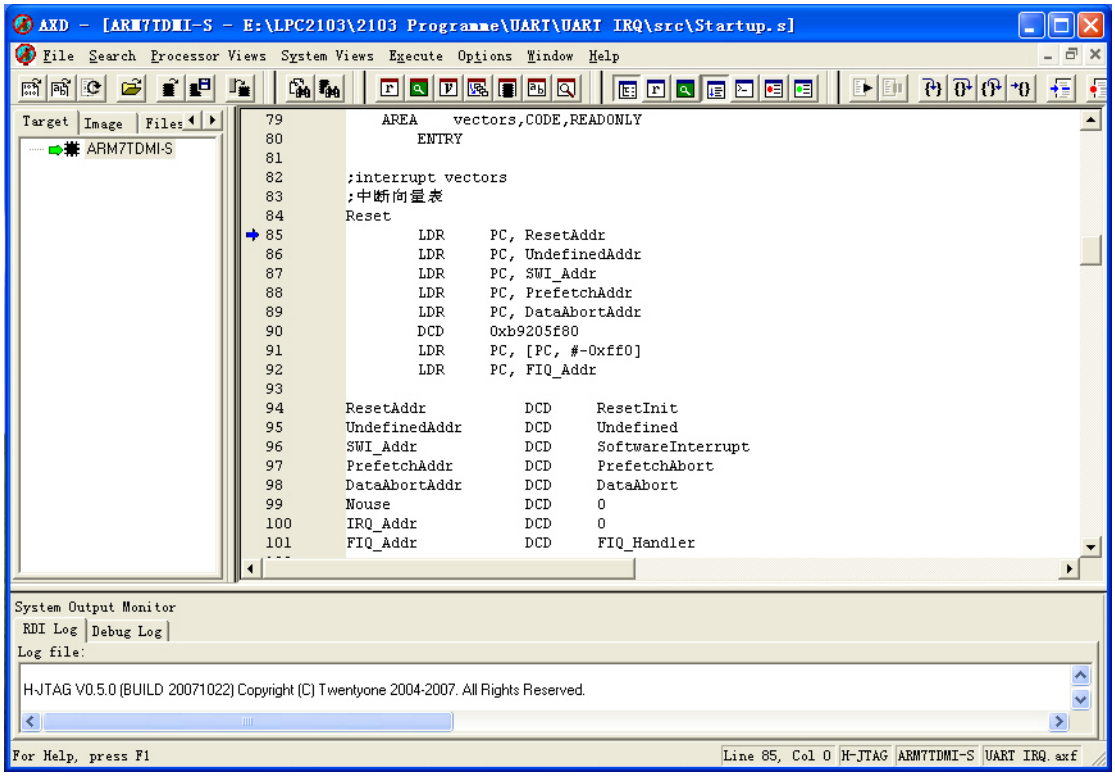


图 3.2 AXD 调试器

AXD调试器ARM Extended Debugger是ARM扩展调试器，包括ADW/ADU的所有特性，

支持硬件仿真和软件仿真(ARMulator)。AXD能够装载映像文件到目标内存，具有单步、全速和断点等调试功能，可以观察变量、寄存器和内存的数据等，AXD调试器主窗口如图 3.2 所示。

3.2 工程的编辑

3.2.1 建立工程

点击WINDOWS操作系统的【开始】→【程序】→【ARM Developer Suite v1.2】→【CodeWarrior for ARM Developer Suite】启动Metrowerks CodeWarrior；或双击“CodeWarrior for ARM Developer Suite”快捷方式启动，启动ADS1.2 IDE如图 3.3所示。

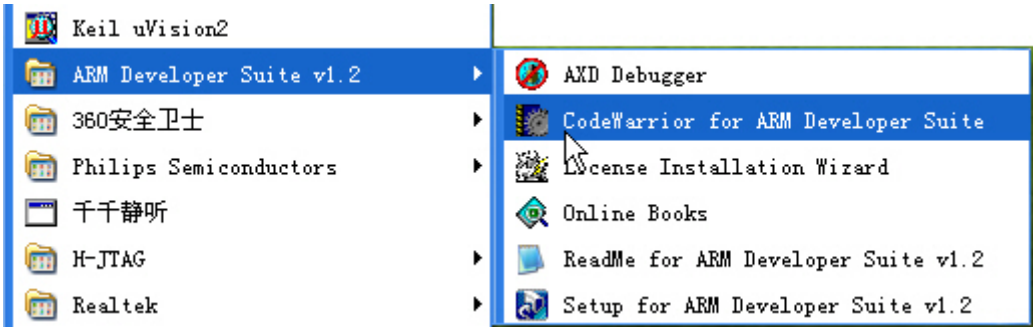


图 3.3 启动 ADS1.2 IDE

点击【File】菜单，选择【New...】即弹出New对话框，如图 3.4所示。

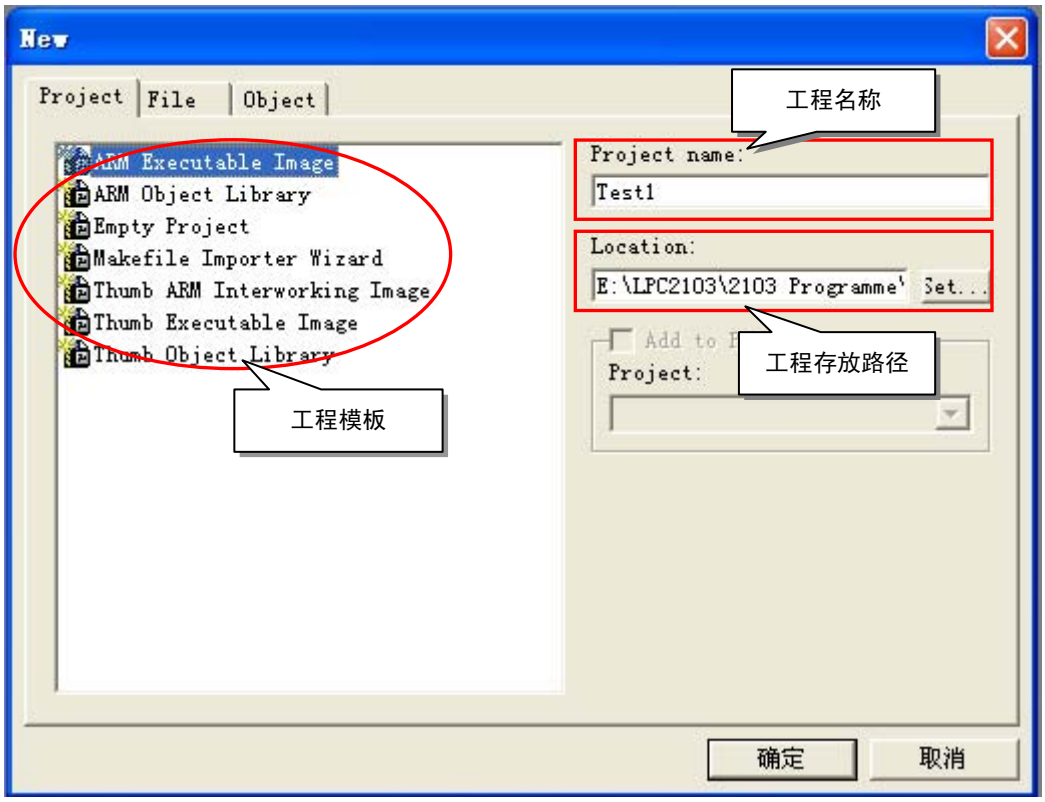


图 3.4 New 对话框

选择工程模板为 ARM 可执行映象（ARM Executable Image）或 Thumb 可执行映象(Thumb Executable Image)，或 Thumb、ARM 交织映象(Thumb ARM Interworking Image)，然后在【Location】项选择工程存放路径，并在【Project name】项输入工程名称，点击【确

定】按钮即可建立相应工程，工程文件名后缀为 mcp（下文有时也把工程称为项目)。

3.2.2 建立文件

建立一个文本文件以便输入用户程序。点击“New Text File”图标按钮，如图 3.5所示。

图 3.5 “New Text File”图标按钮

然后在新建的文件中编写程序，点击“Save”图标按钮将文件存盘(或从【File】菜单选择【Save】)，输入文件全名，如 TEST1.S。注意，请将文件保存到相应工程的目录下，以便于管理和查找。

当然，您也可以New对话框选择【File】页来建立源文件，如图 3.4所示，或使用其它文本编辑器建立或编辑源文件。

3.2.3 添加文件到工程

如图 3.6所示，在工程窗口中【Files】页空白处点击鼠标右键，弹出浮动菜单，选择“Add Files…”即可弹出“Select files to add…”对话框，选择相应的源文件（可按着Ctrl键一次选择多个文件），点击【打开】按钮即可。

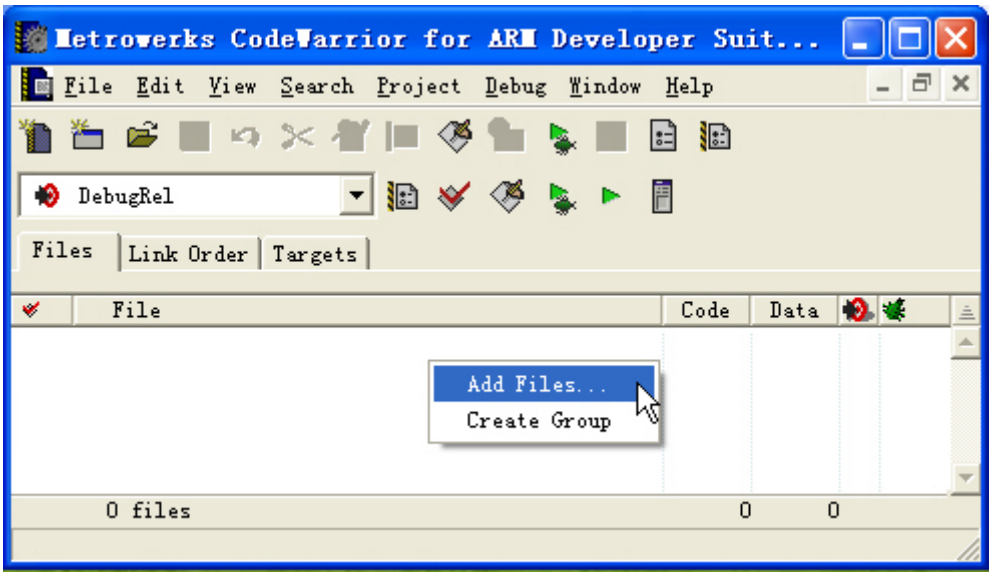


图 3.6 在工程窗口中添加源文件

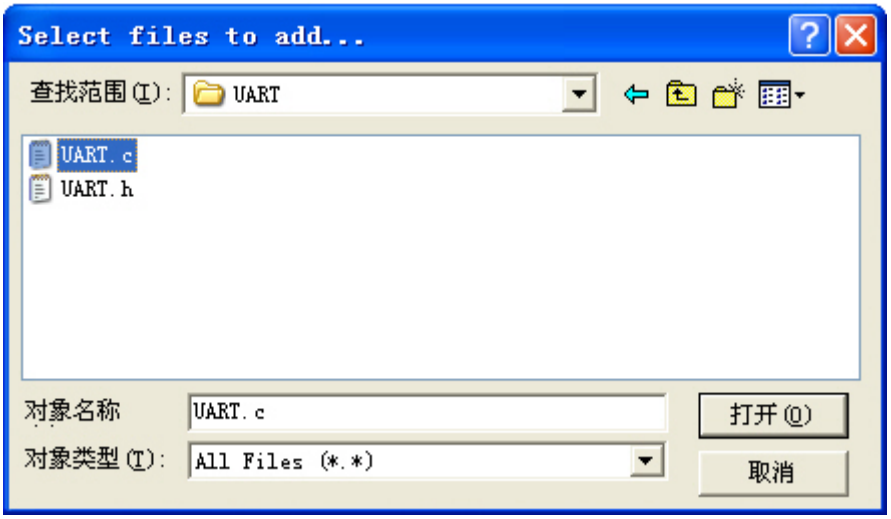


图 3.7　Select files to add…对话框

另外，用户也可以在【Project】菜单中选择【Add Files…】来添加源文件，或使用New对话框选择【File】页来建立源文件时选择加入工程，即选中“Add to Project”项。添加文件操作如图 3.6、图 3.7所示。

3.2.4 编辑连接工程

如图 3.8所示为工程窗口中的图标按钮，通过这些图标按钮，您可以快速地进行工程设置、编译连接、启动调试等等。在不同的菜单项上可以分别找到对应的菜单命令，它们从左至右分别为：

DebugRel Settings…	**工程设置，如地址设置、输出文件设置、编译选项等，其中 DebugRel 为当前的生成目标(target system)；**
Synchronize Modification Dates	同步修改日期，检查工程中每个文件的修改日期，若发现有更新（如使用其它编辑器编辑源文件），则在 Touch 栏标记“√”；
Make	**编译连接(快捷键为 F7)；**
Debug	**启动 AXD 进行调试(快捷键为 F5)；**
Run	启动 AXD 进行调试，并直接运行程序；
Project Inspector	工程检查，查看和配置工程中源文件的信息。

图 3.8　工程窗口中的图标按钮

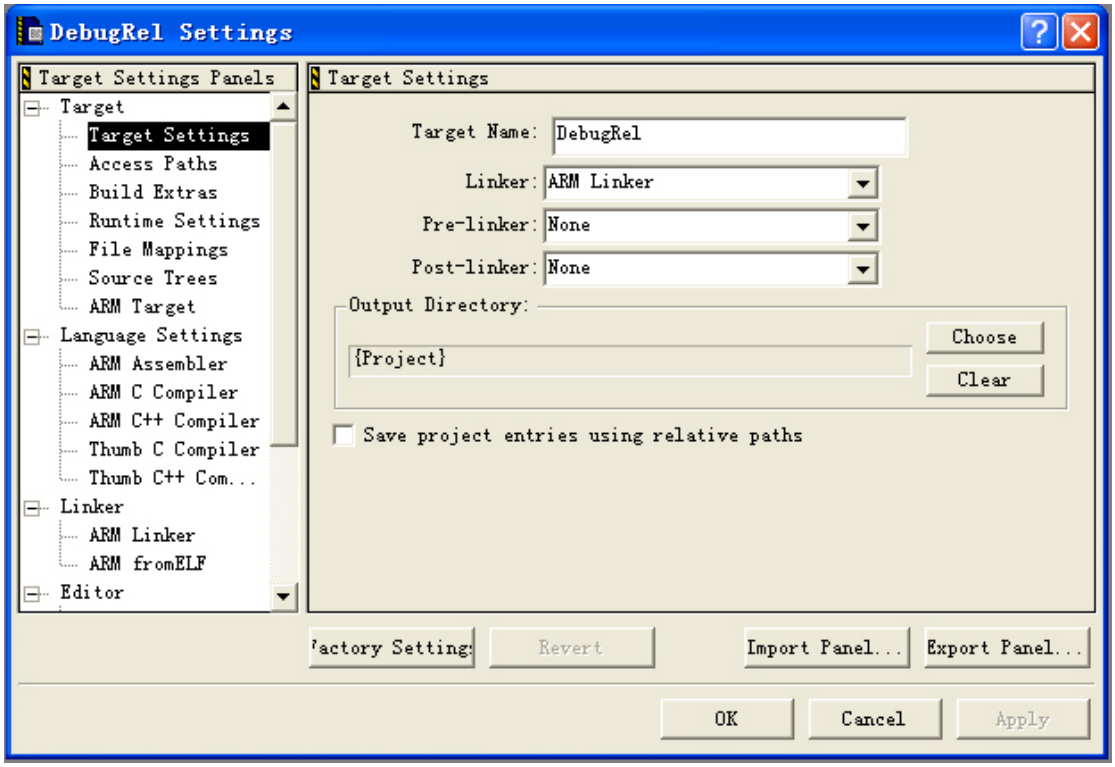


图 3.9 DebugRel Settings 窗口

点击“DebugRel Settings…”图标按钮，即可进行工程的地址设置、输出文件设置、编译选项等，如图 3.9所示。在“ARM Linker”对话框设置连接地址，在“Language Settings”中设置各编译器的编译选项。

对于简单的软件调试，可以不进行连接地址的设置，直接点击工程窗口的“Make”图标按钮，即可完成编译连接。**若编译出错，会有相应的出错提示，双击出错提示行信息，编辑窗即会使用光标指出当前出错的源代码行**。编译连接输出窗口如图 3.10所示。

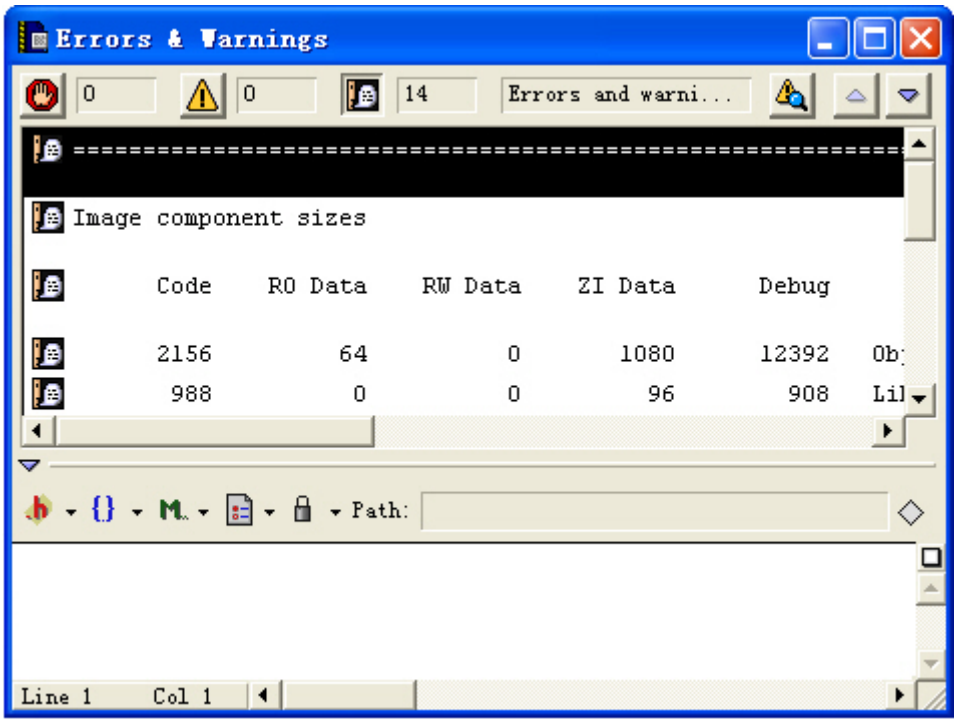


图 3.10 编译连接输出窗口

Touch栏用于标记文件是否已编译，若打上“√”则表明对应文件需要重新编译，如图3.11所示。可以通过单击该栏位置来设置/取消符号“√”，或将工程目录下的*.tdt文件删除也可以使整个工程源文件均打上“√”。

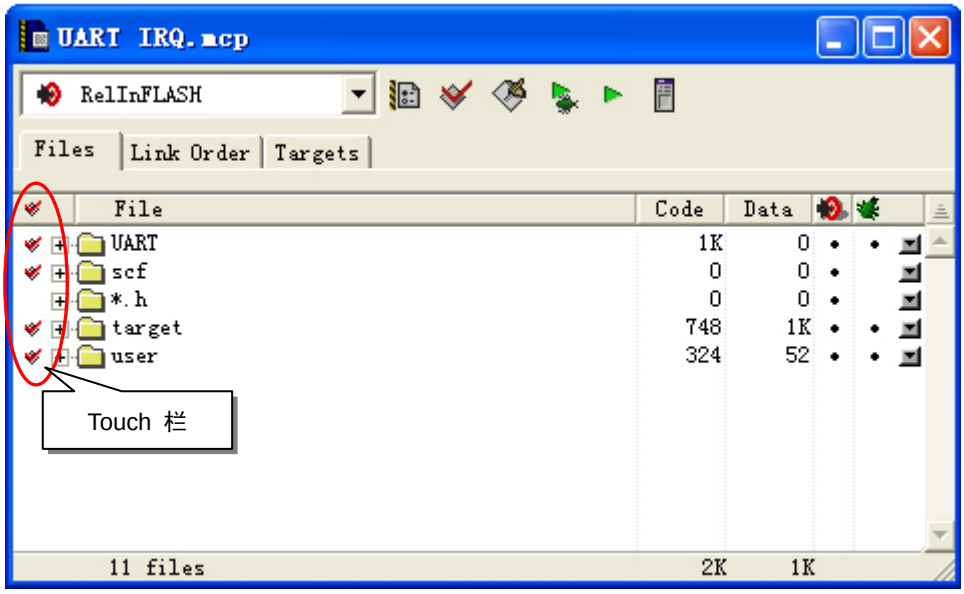


图 3.11 工程窗口中 Make 操作

重新编译之前，建议将原来生成的目标文件都删除，方法如下，点选“project”下拉菜单的“Remove Object code”->“All Targets”，如图 3.12所示，删除了旧目标文件后，所有文件都被“touch”上了，此时可对整个工程进行重新编译。

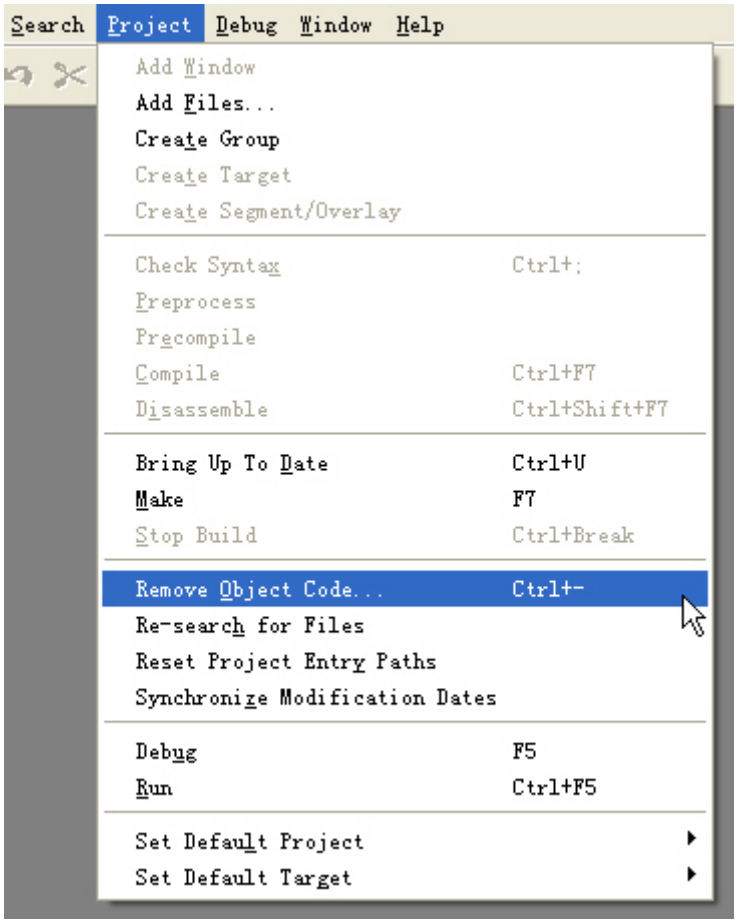


图 3.12 删除旧的目标文件

3.2.5 打开旧工程

点击【File】菜单，选择【Open】即弹出“打开”对话框，找到相应的工程文件(*.mcp)，单击【打开】即可。在工程窗口的【Files】页中，双击源程序的文件名即可打开该文件进行编辑。

3.3 工程的调试

3.3.1 选择调试方式

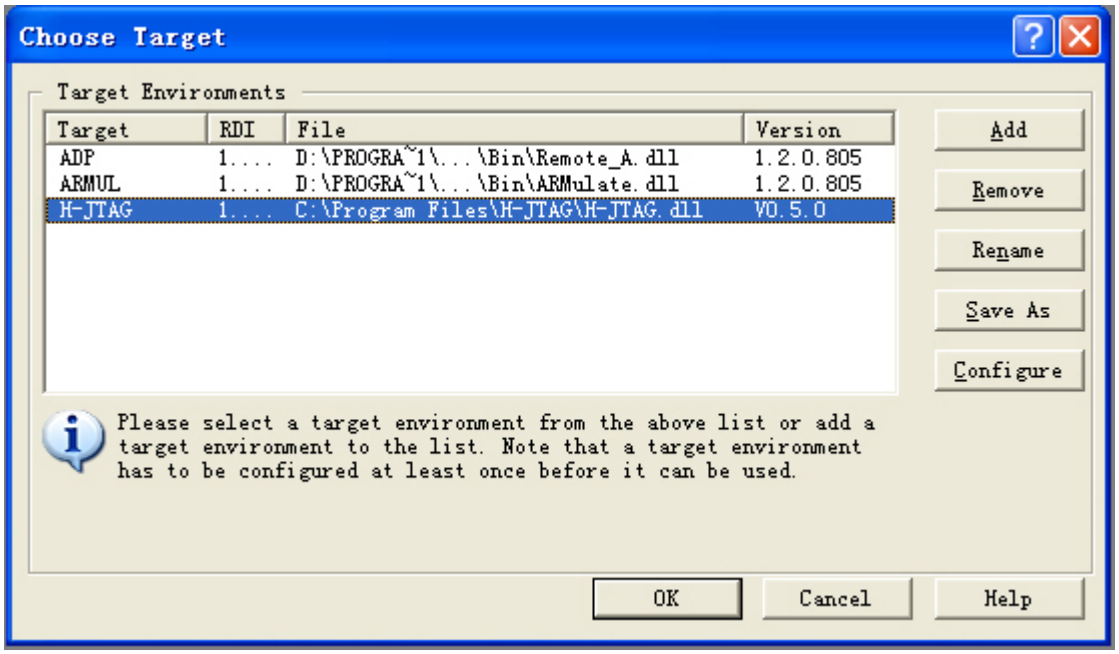


图 3.13 Choose Target 窗口

当工程编译连接通过后，在工程窗口中点击“Debug”图标按钮（或者使用快捷键F5），即可启动AXD（也可以通过【开始】菜单起动AXD）。点击菜单【Options】选择【Configure Target…】，即弹出Choose Target窗口，如图 3.13所示。在没有添加其它仿真驱动程序前，Target项中只有两项，分别为ADP（JTAG硬件仿真）和ARMUL（软件仿真)。

选择仿真驱动程序后，点击【File】选择【Load Image…】加载 ELF 格式的可执行文件，即*.axf 文件。（说明：当工程编译连接通过后，在“工程名\工程名_Data\当前的生成目标”目录下就会生成一个*.axf 调试文件。比如工程 TEST，当前的生成目标 Debug，编译连接通过后，则在…\TEST\TEST_Data\Debug 目录下生成 TEST.axf 文件）。

3.3.2 调试工具条

AXD运行调试工具条如图 3.14所示，调试观察窗口工具条如图 3.15所示，文件操作工具条如图 3.16所示。

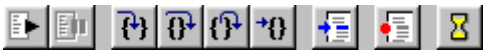


图 3.14 运行调试工具条

全速运行(Go)

停止运行(Stop)

单步运行(Step In)，与 Step 命令不同之处在于对函数调用语句，Step In 命令将进入该函数。

 　单步运行(Step)，每次执行一条语句，这时函数调用将被作为一条语句执行。

 　单步运行(Step Out)，执行完当前被调用的函数，停止在函数调用的下一条语句。

 　运行到光标(Run To Cursor)，运行程序直到当前光标所在行时停止。

 　设置断点(Toggle BreakPoint)

图 3.15　调试观察窗口工具条

 　打开寄存器窗口(Processor Registers)

 　打开观察窗口(Processor Watch)

 　打开变量观察窗口(Context Variable)

 　打开存储器观察窗口(Memory)

 　打开反汇编窗口(Disassembly)

图 3.16　文件操作工具条

 　加载调试文件(Load Image)

 　重新加载文件(Reload Current Image)。由于 AXD 没有复位命令，所以通常使用 Reload 实现复位(直接更改 PC 寄存器为零也能实现复位)。

3.4　LPC2103 微控制器工程模板

在第3.2节介绍新建立工程时，我们已经接触了ADS1.2 提供的几个标准工程模板，使用各个模板建立的工程，它们的各项设置均有不同之处，方便生成不同结构的代码，如ARM可执行映象（生成ARM指令的代码）或Thumb可执行映象（生成Thumb指令的代码），或Thumb、ARM交织映象（生成Thumb、ARM指令交织的代码）。

针对 LPC2103 微控制器，我们定义了 2 个工程模板，这些模板一般包含的设置信息有 FLASH 起始地址 0x00000000、片内 RAM 起始地址 0x40000000、编译连接选项及编译优化级别等等；模板中包含了 LPC2103 微控制器的启动文件，包括 Startup.S、tagec.c；模板还包含了 LPC2103 微控制器的头文件（如：LPC2294.h），分散加载描述文件（如：mem_a.scf、mem_b.scf、mem_c.scf）等等。

3.4.1 添加LPC2103 专用工程模板

将“配套光盘内容\光盘内容v1.00\5.工程模板”目录下的所有文件拷贝到“ADS1.2安装目录>\Stationery\”即可，操作如图 3.17所示。这个步骤只需 1 次，以后可以直接使用工程模板了。

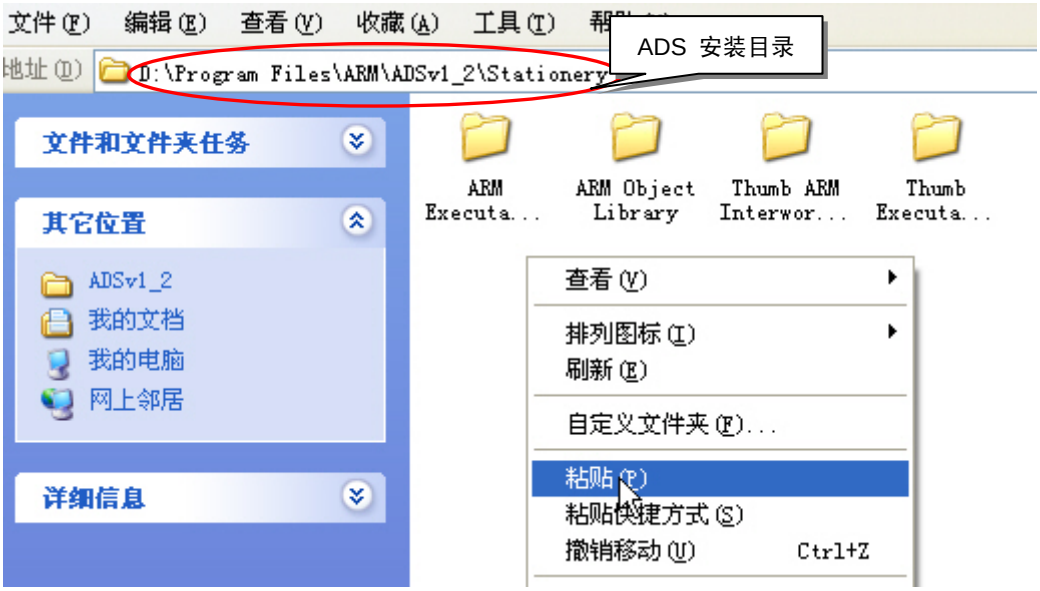


图 3.17 复制文件目录

3.4.2 应用LPC2103 模板建立工程

启动ADS1.2 IDE，点击【File】菜单，选择【New】即弹出New对话框，如图 3.18所示。由于事先增加了LPC2103 专用工程模板和其他模版，所以在工程模板栏中多出几项工程模板选项。

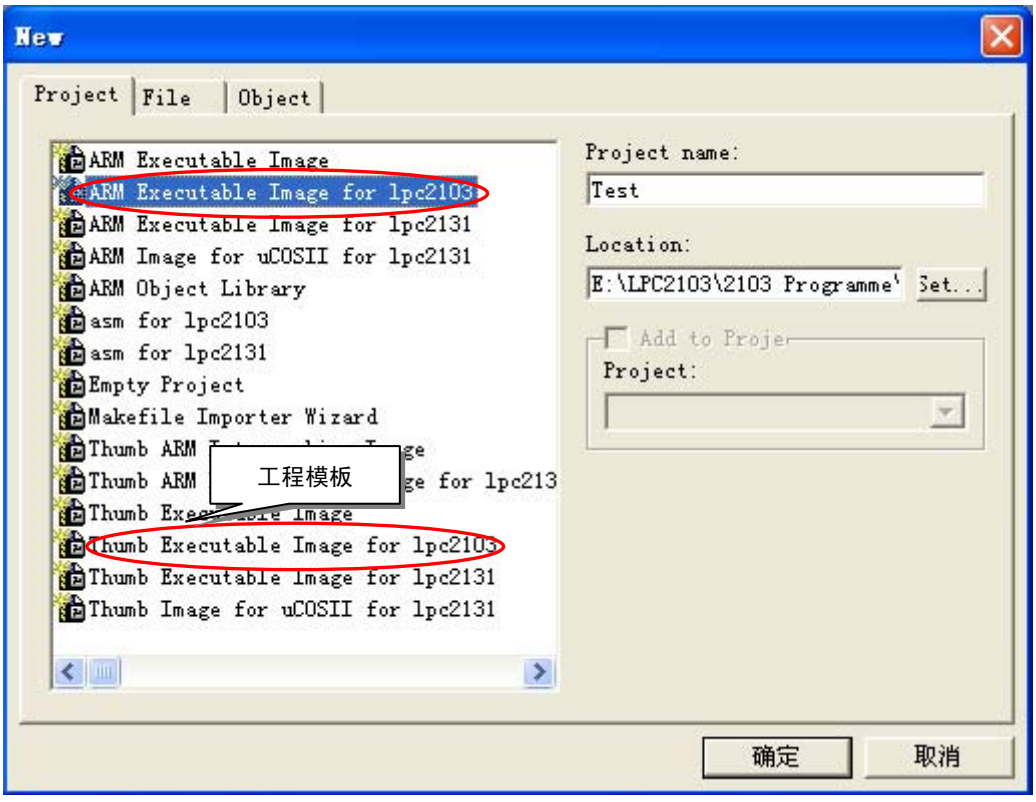


图 3.18 增加的工程模板

LPC2103 专用工程模板说明如下：

ARM Executable Image **for LPC2103**：无操作系统时，所有 C 代码均编译成 ARM 指令的工程模板。

Thumb Executable Image for LPC2103：无操作系统时，所有 C 代码均编译成 Thumb 指令的工程模板。

用户选择相应的工程模板建立工程，如图 3.19所示为使用ARM Executable Image for lpc2103 工程模板建立的一个工程。工程有 3 个生成目标(target system)：DebugInRAM、DebugInFlash、RelInFLASH。工程模板已经将相应的编译参数设置好了，直接使用即可。

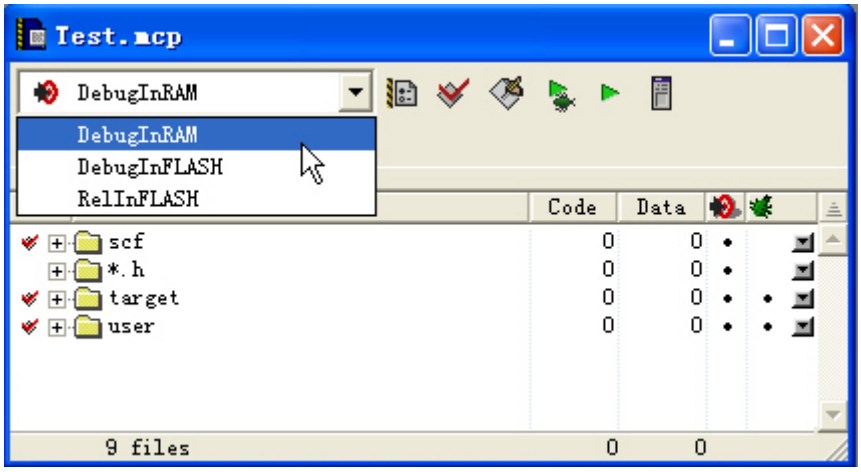


图 3.19 用 LPC2103 系列 ARM 专用工程模板建立的工程

注意：选用 RelInFLASH 目标时，将会对 LPC2103 芯片进行加密。加密的芯片只能使用 ISP 进行芯片整片擦除后，才能恢复 JTAG 调试及 ISP 读/写操作。

3.5 EasyJTAG-H简介

EasyJTAG-H 仿真器是一款新型的仿真器，目前，可以支持 LPC21037 微控制器和部分 ARM9 芯片，支持 ADS1.2 集成开发环境，支持单步、全速及断点等调试功能，支持下载程序到片内 FLASH 和特定型号的片外 FLASH，采用 ARM 公司提出的标准 20 脚 JTAG 仿真调试接口。这款仿真器需要 H-JTAG 软件（调试代理）的支持。

3.5.1 EasyJTAG-H安装

由于EasyJTAG-H仿真器需要H-JTAG软件的支持，所以使用EasyJTAG-H之前，必须要先安装H-JTAG。将H-JTAG压缩包解压，然后运行安装文件H-JTAG.EXE，如图 3.20所示，根据安装提示完成安装即可。

图 3.20 H－JTAG 安装界面

安装好的H-JTAG软件包含有H-JTAG Server(下文简称为H-JTAG)和H-Flasher，在桌面上有它们的快捷图示，如图 3.21所示。

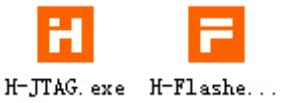


图 3.21 H-JTAG 快捷图示

3.5.2 H-JTAG配置

H-JTAG 设置较简单，只要进行以下两步操作，其它采用默认设置即可。

（1）单击任务栏的H 提示图标，将看见H-JTAG的主窗口，如图 3.22所示。单击“放大镜”图标按钮后，能看见调试代理搜索到ARM7 处理器。

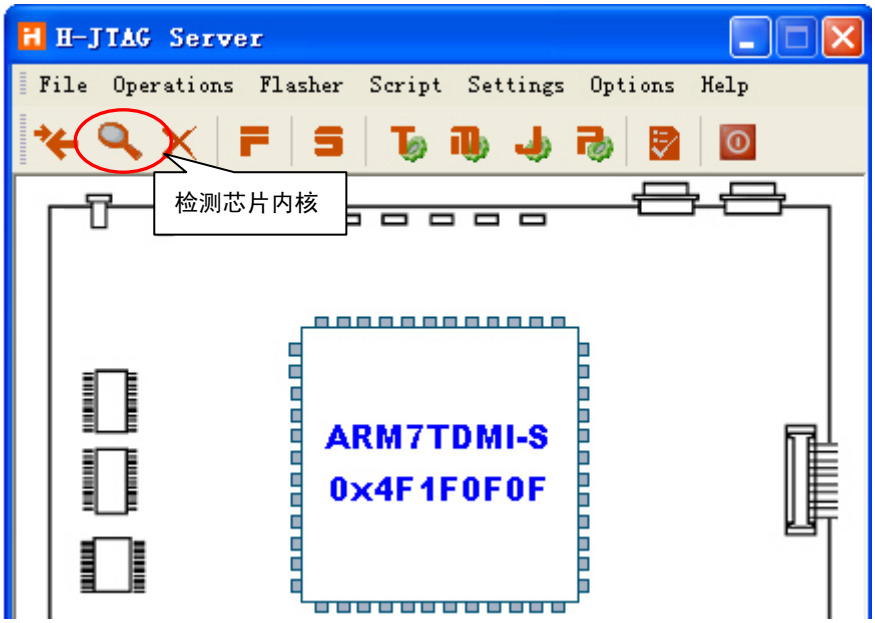


图 3.22 H-JTAG Server 检测到的芯片内核

（2）选择【Flasher】->【Auto Download】选择自动下载项。

注意，在 Flash 中调试时必须要选择“Auto Download”，而在 RAM 中调试可以不选择。

H-Flasher 主要用来完成 Flash 的烧写。如果需要手动下载 hex 文件，需要按照下面的步骤进行操作：

- 在 1 Flash Selection中选择CPU的型号。如图 3.23所示；

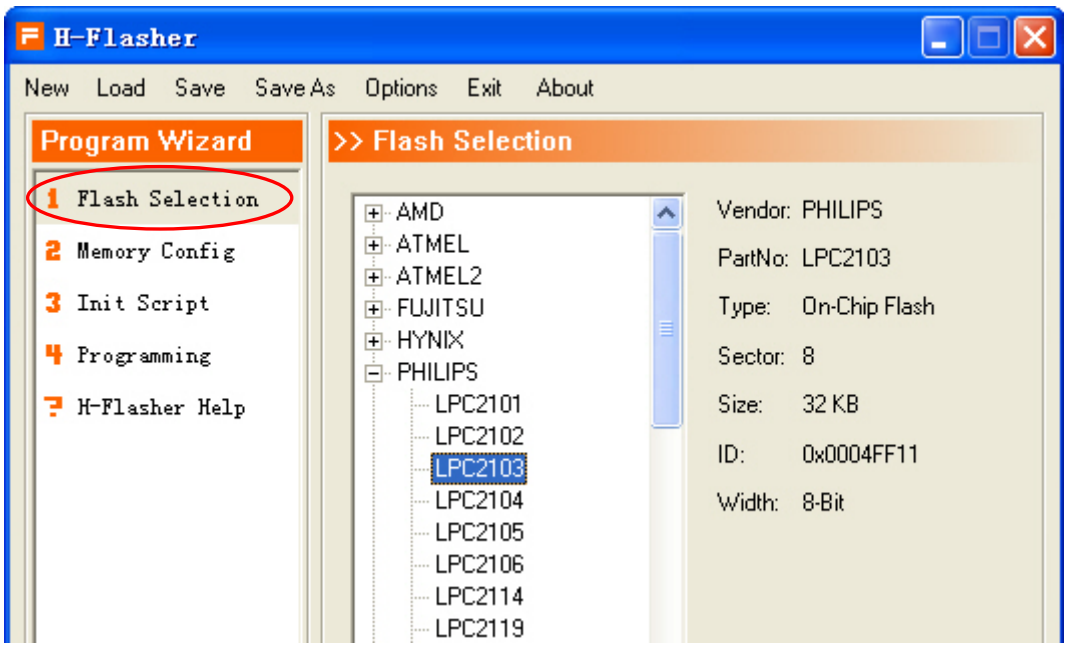


图 3.23 选择芯片型号

- 打开 4 Programing选项，如图 3.24所示。

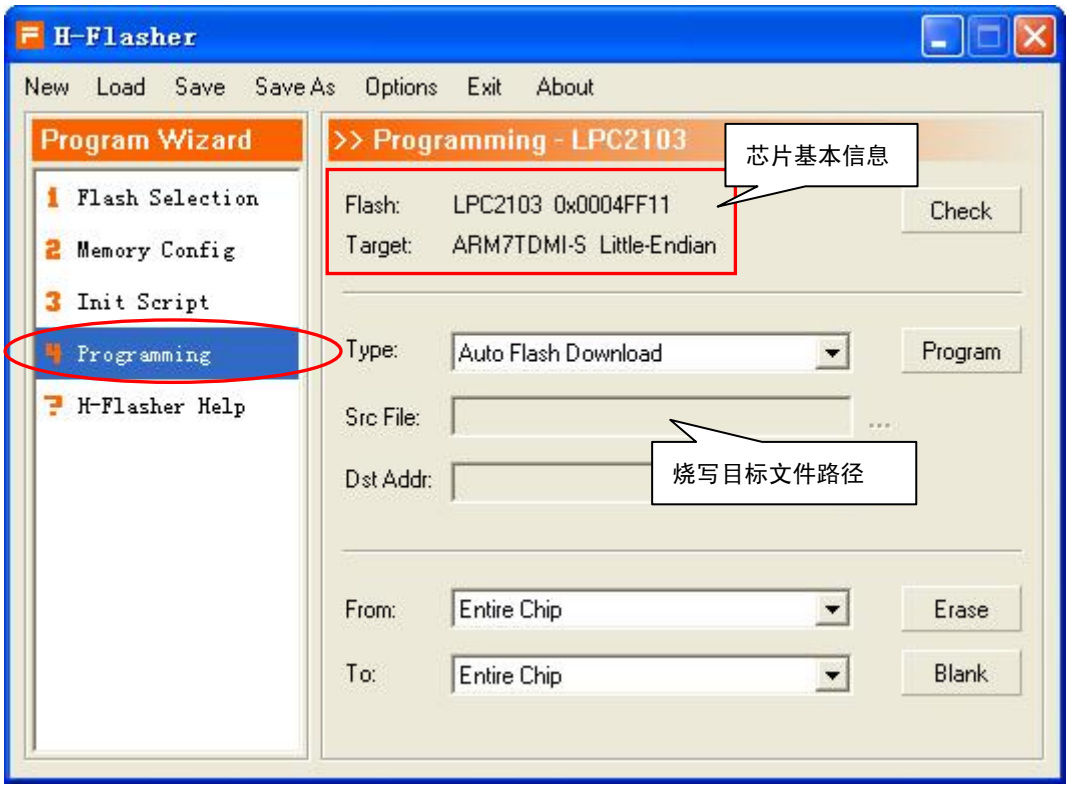


图 3.24 Flash 编程选项

Check：检测芯片内核。如果 EasyJTAG-H 连接正确，且芯片型号正确，则 Check 后会显示芯片的基本信息。

Type：烧写文件的类型。

Auto Flash Down：自动下载方式；

Intel Hex Format：下载 Hex 文件；

Plain Binary Format：下载 Bin 文件。

Src File：烧写文件的路径，如果 Type 选择为 **Auto Flash Down** 时，该项无效。

Dst Addr：目标地址信息，只有 Type 选择为 **Plain Binary Format** 时，该项才有效。

Program：对芯片进行编程操作。

Erase：对选中（由“From ～ To”指定）的扇区进行擦除操作。

工具栏中各选项功能说明：

New：新建一个配置文件。

Load：载入配置文件。H-Flasher 在启动时，总是自动载入最近一次的配置信息。

Save：将当前的配置信息保存为一个文件。

Save As：将配置信息另存。

Option：调试程序时，是否使能自动计算向量表前 32 字的累加和。默认为使能。

3.6 EasyJTAG－H仿真器的使用

本节将通过使用 EasyJTAG—H 仿真器在开发板上运行一个程序，来介绍仿真器的具体使用方法，最终脱机运行。

3.6.1 在开发板上运行第一个程序

下面让我们利用LPC2103 在ADS下的工程模板来编写程序，程序的功能设计为控制LED闪烁，硬件电路如图 3.25所示。

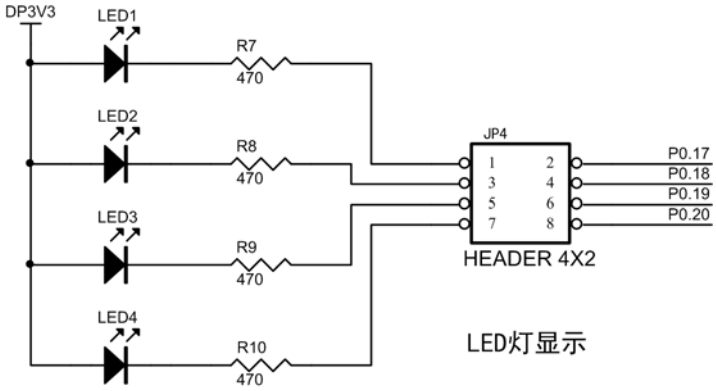


图 3.25 闪灯程序电路原理图

3.6.2 建立工程

1、打开ADS（ARM_Developer Suite v1.2-> CodeWarrior for ARM Developer Suite）开发环境，使用ARM Executable Image for lpc2103 模板建立工程——Example，如图 3.26所示。项目目录如图 3.27所示。

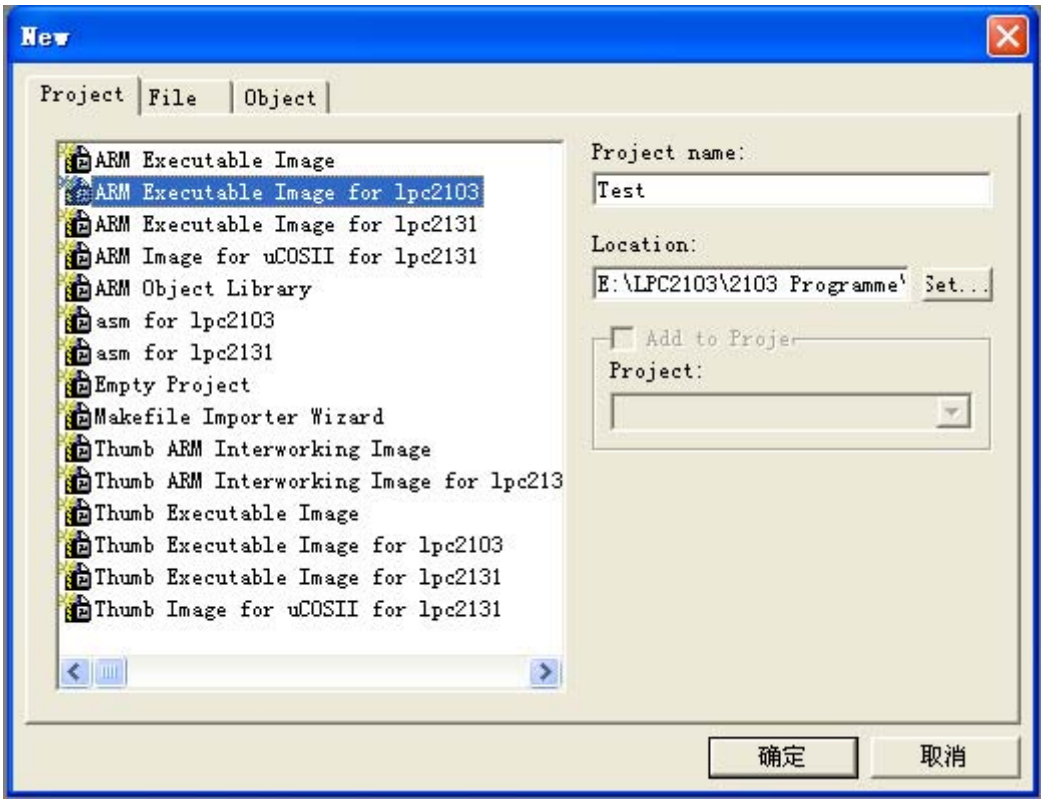


图 3.26 使用 LPC2103 工程模板建立工程——Example

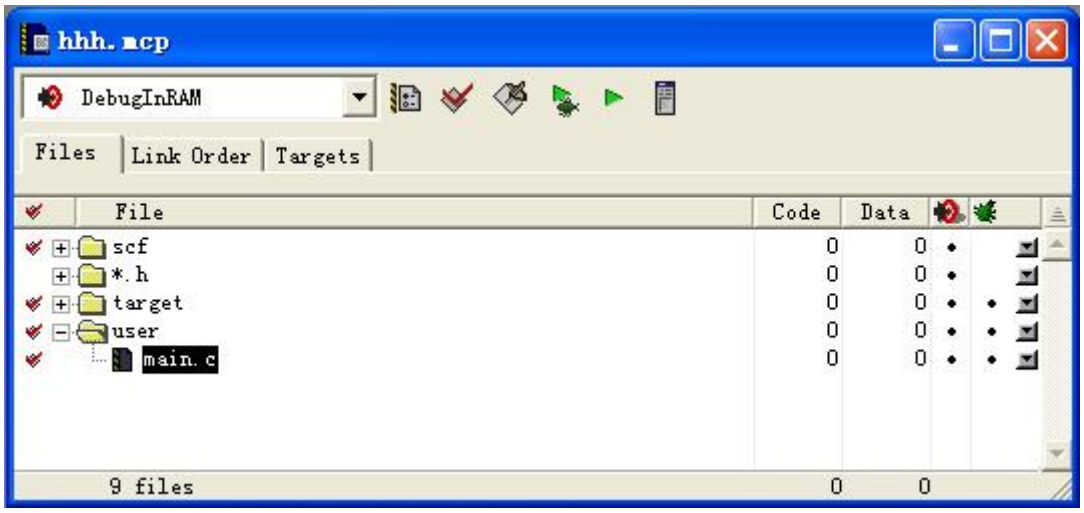


图 3.27　ADS 项目工程目录

2、在main.c文件中添加代码，如程序清单 3.1所示。

程序清单 3.1　LED 控制程序

```
#include        "config.h"
#define         LED1            (1<<17)                 /* LED1                                 */

/*********************************************************************************************
** 函数名称：  Delay
** 函数功能：  延时子程序
** 入口参数：  dly   延时参数
** 出口参数：  无
*********************************************************************************************/
void Delay(uint32 dly)
{
    uint32   i;
    for(;dly > 0;dly--)
        for(i = 0;i < 50000;i++);
}

/*********************************************************************************************
** 函数名称：  main
** 函数功能：  LED 控制程序
** 入口参数：  无
** 出口参数：  无
** 说    明：  短接跳线 JP4
*********************************************************************************************/
int main (void)
{
    PINSEL1  =    0x00000000;                   /* 设置管脚连接 GPIO                    */
    IO1DIR   =    LED1;                         /* 设置 LED1 控制口为输出               */
```

```
    while(1)    {
        IO1SET    =    LED1;                    /* LED1 熄灭                    */
        Delay(50);
        IO1CLR    =    LED1;                    /* LED1 点亮                    */
        Delay(50);
    }
    return 0;
}
```

3.6.3 编译连接工程

在项目目标栏处选择 DebugInRAM 模式，然后编译连接工程。

选择 DebugInRAM 生成目标时，编译连接生成的目标代码用于在片内 RAM 调试。

3.6.4 仿真调试

将计算机并口与EasyJTAG-H仿真器相连，然后再将EasyJTAG-H仿真器的JTAG接口连接到EasyARM2103 目标板上，打开H-JTAG Server，检测到芯片内核信息后，打开Auto Download选项，如图 3.28所示。此时会自动启动H-Flasher软件，选择目标芯片的型号，如图 3.23所示，将当前的配置信息保存起来，建议将配置信息保存到安装路径下的Hconfig文件夹内，如图 3.29所示。

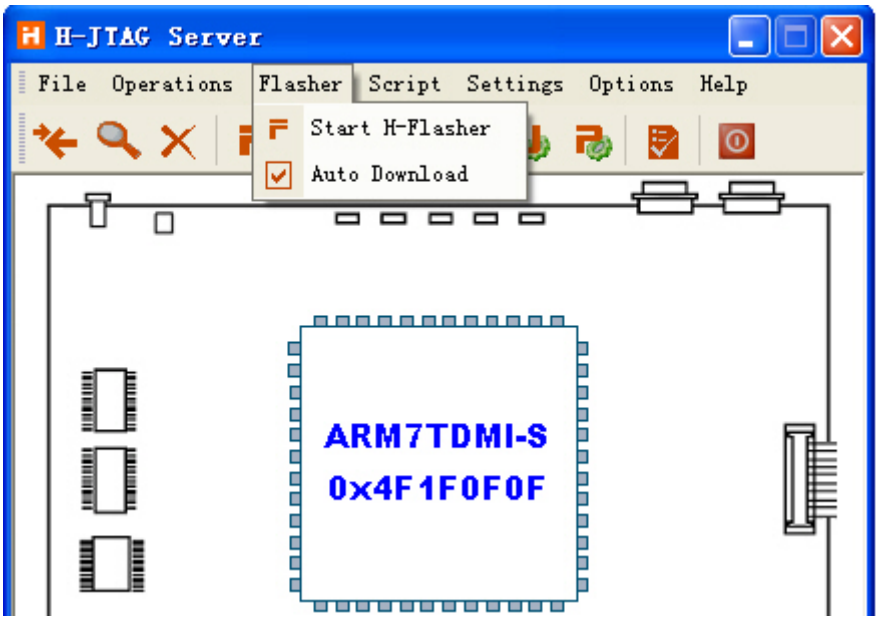


图 3.28　启动 H－JTAG Server

图 3.29 保存配置信息

在 H-JTAG Server 中选中 Auto Download 项后，H-Flasher 的只需设置芯片型号即可，至于烧写目标文件可以不设置。

设置完成后，关闭H-JTAG Server和H-Flasher（注意：不能使用Exit项关闭）。启动AXD，打开【Options】->【Configure Target...】，弹出Choose Target窗口，如图 3.30所示。点击“ADD”添加仿真器的驱动程序，在添加文件窗口选择如D:\Program Files\H-JTAG 目录下的H-JTAG .dll，点击“打开”即可。

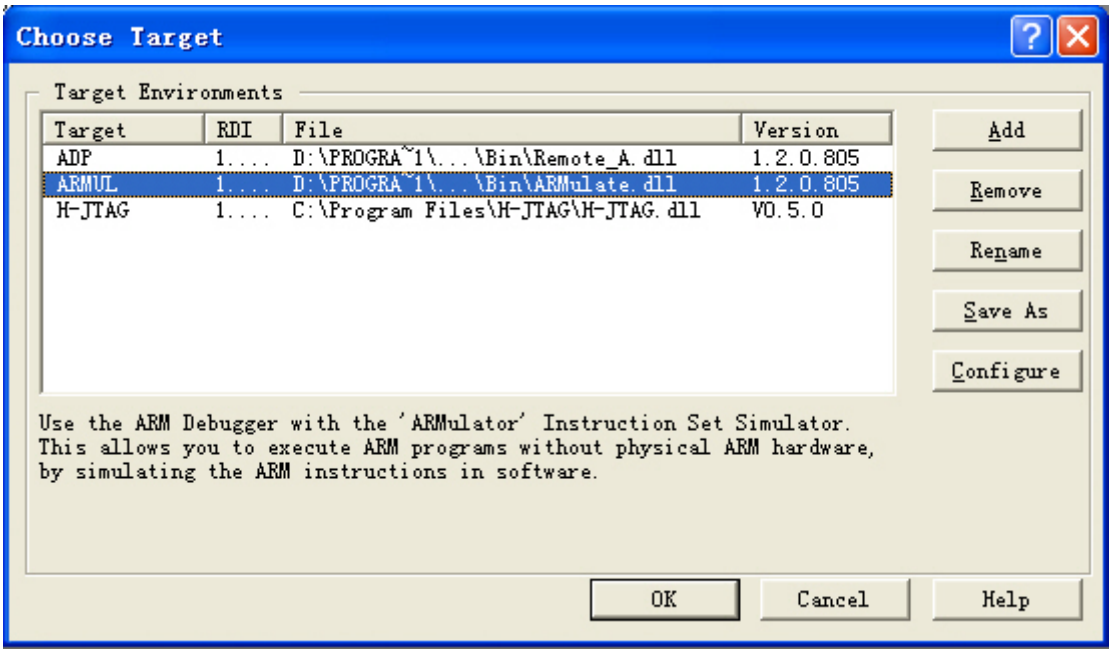


图 3.30 Choose Target 窗口

注意：若在添加文件窗口中没有显示 DLL 文件，请设置 WINDOWS 文件浏览窗口的“文件夹选项(O)...”，将查看页中的“隐藏文件”项选用“显示所有文件”。

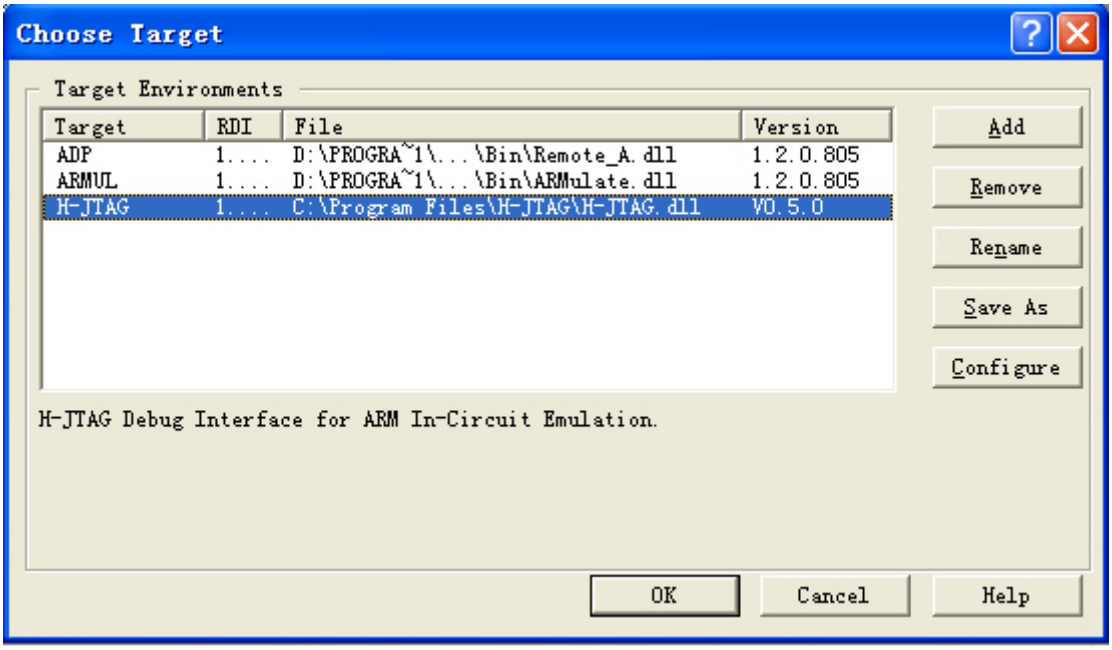


图 3.31 选择驱动程序

添加完EasyJTAG-H驱动后，选择该驱动程序，如图 3.31所示，然后点击OK，出现图3.32所示的界面。

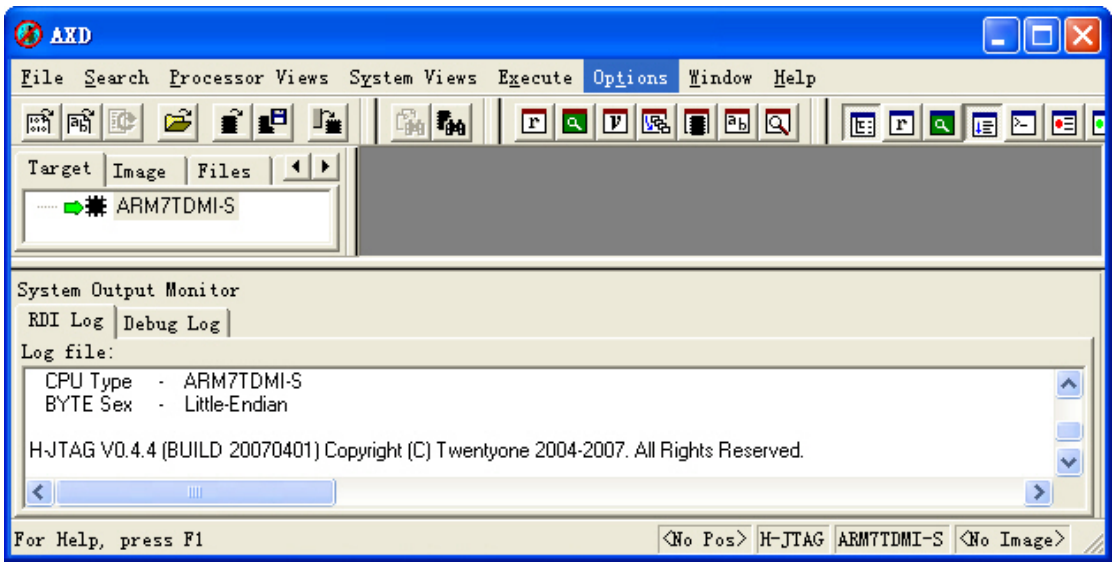


图 3.32 H-JTAG 检测到的 CPU 内核

关闭AXD界面，回到ADS中，在正常情况下，点击Debug仿真后，PC指针会指向中断向量表的起始处，如图 3.33所示。

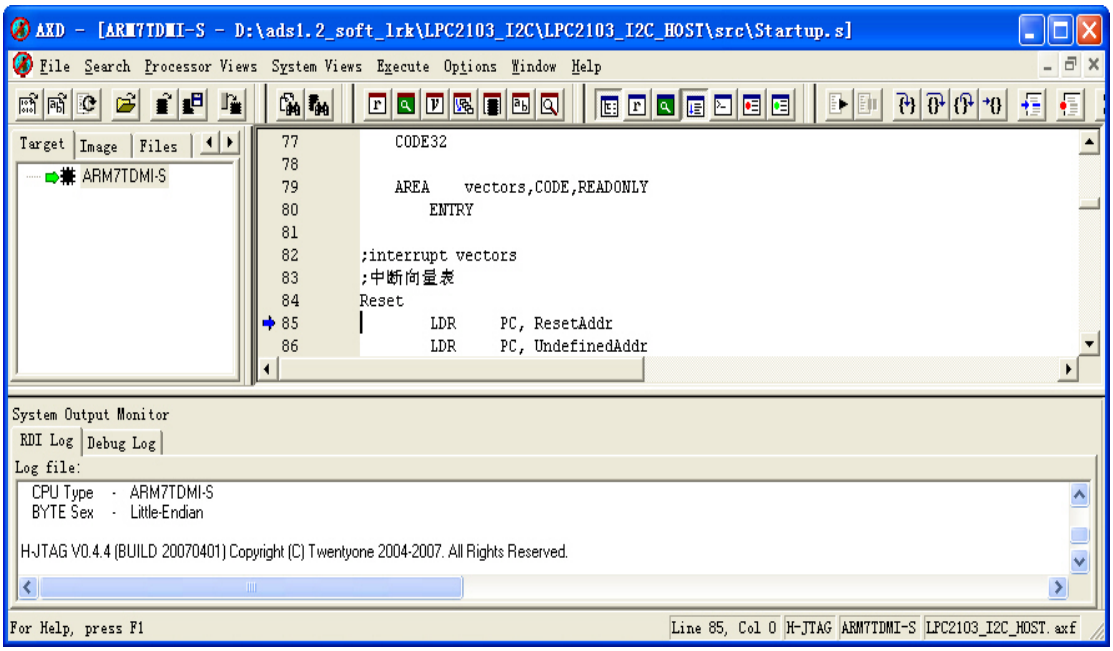


图 3.33 进入 AXD 仿真调试界面

如果工程文件的路径中存在中文，直接进入AXD调试环境会出现错误，此时需要重新设置配置信息，如图 3.31所示。因此，建议工程路径中不包含中文。

3.6.5 脱机运行

调试时若选择使用 DebugInFLASH 生成目标，并进行调试后（使用 EasyJTAG-H 仿真器），程序即烧写到片内 Flash 中。将 EasyJTAG-H 和电源断开，重新上电，程序将脱机运行，将会看到 LED 闪烁。

使用 RelInFlash 生成目标时，编译连接生成的目标代码会将芯片加密。此时不能再进行调试，除非使用 ISP 进行全片擦除，否则是不能再进行调试的。

3.7 EasyJTAG-H常见问题

（1） 在进行 AXD 仿真调试前，需要首先打开 H-JTAG Server 检测芯片内核，并且要选择 Auto Load 选项，同时要在 H-Flasher 中选择芯片型号；

（2） 在AXD调试过程中，将开发板硬件复位，如果需要继续调试，需要在H-JTAG Server中复位目标项——Reset Target，如所图 3.34示，然后Reload Image，此时会弹出“不能重新建立断点”的错误信息，如图 3.35所示；

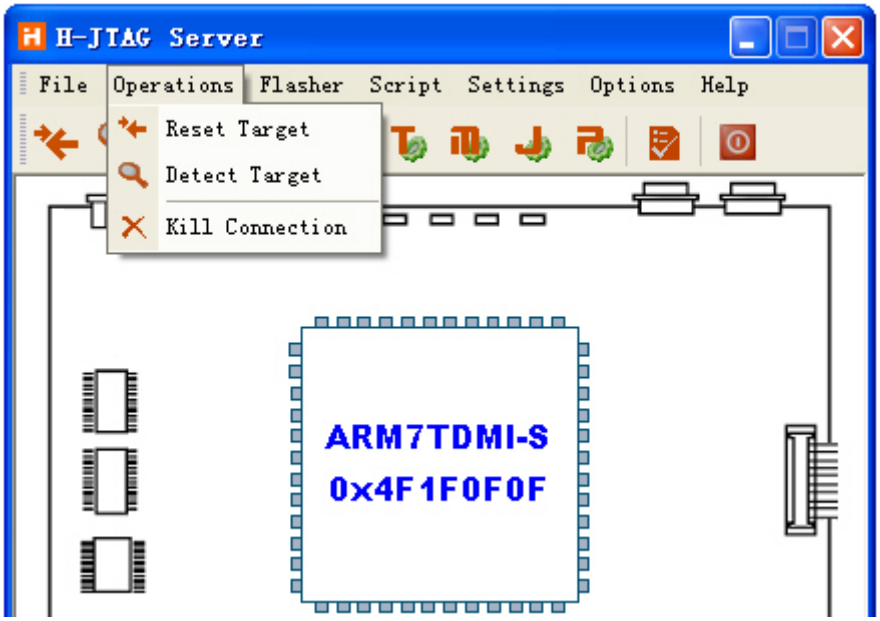


图 3.34 Reset Target

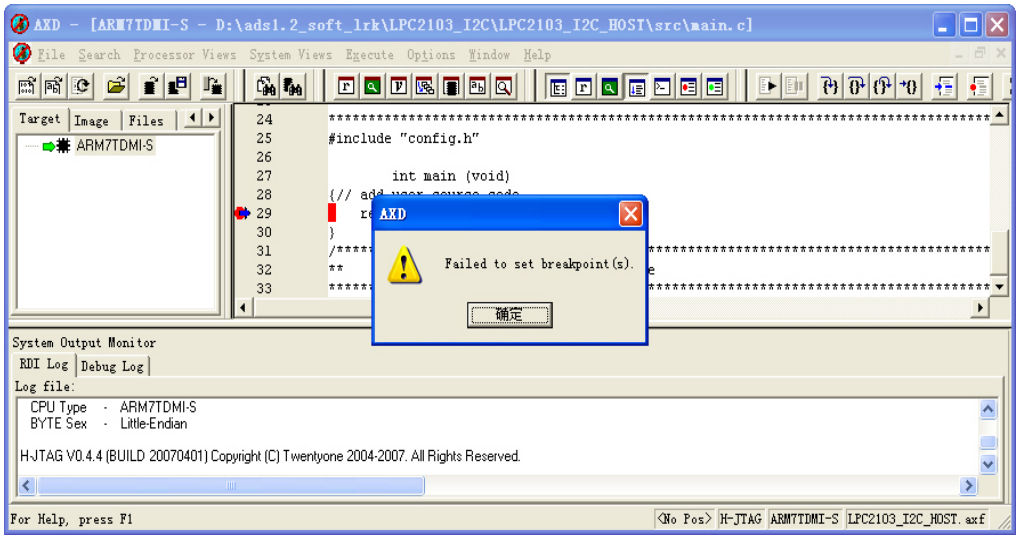


图 3.35 不能重新建立断点

选择确定，接下来在弹出的AXD错误信息中选择Restart，如图 3.36所示。

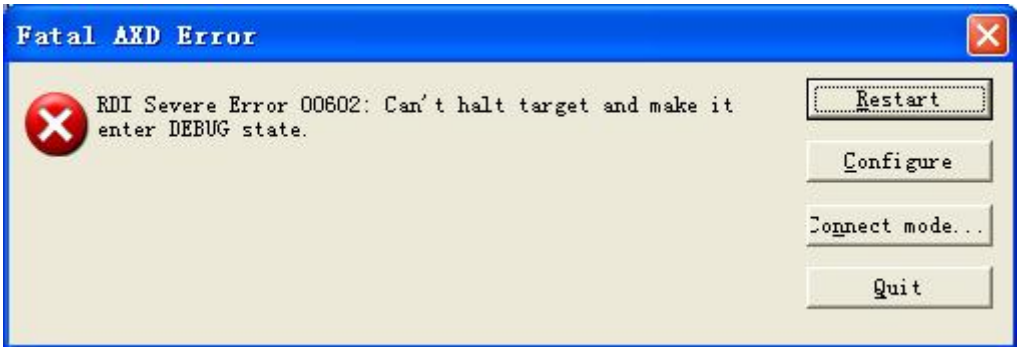


图 3.36 AXD 错误信息

最后重新载入image文件，如图 3.37所示。

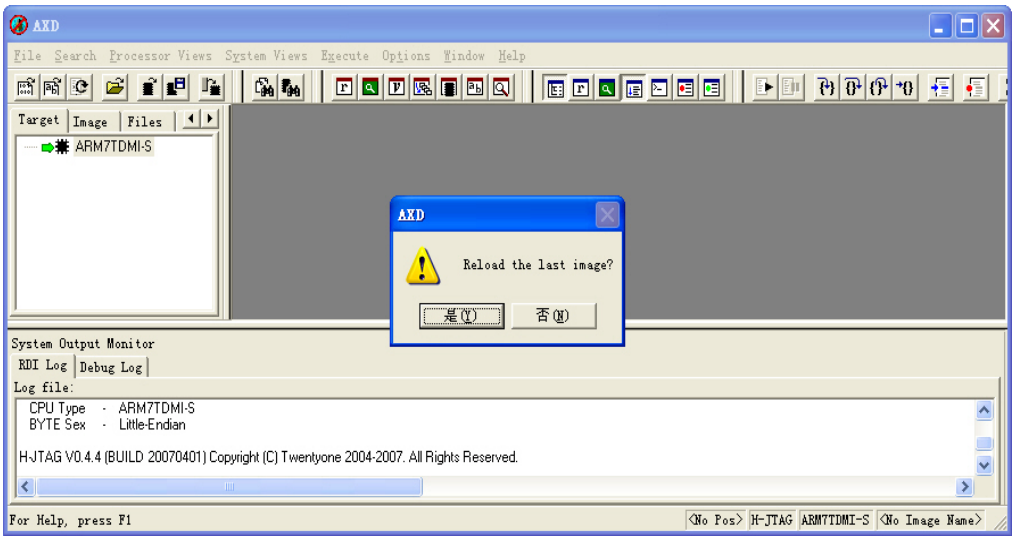


图 3.37 Reload the last image

（3） 如果芯片进入到Power Down模式，H-JTAG会出现错误，如图 3.38所示；

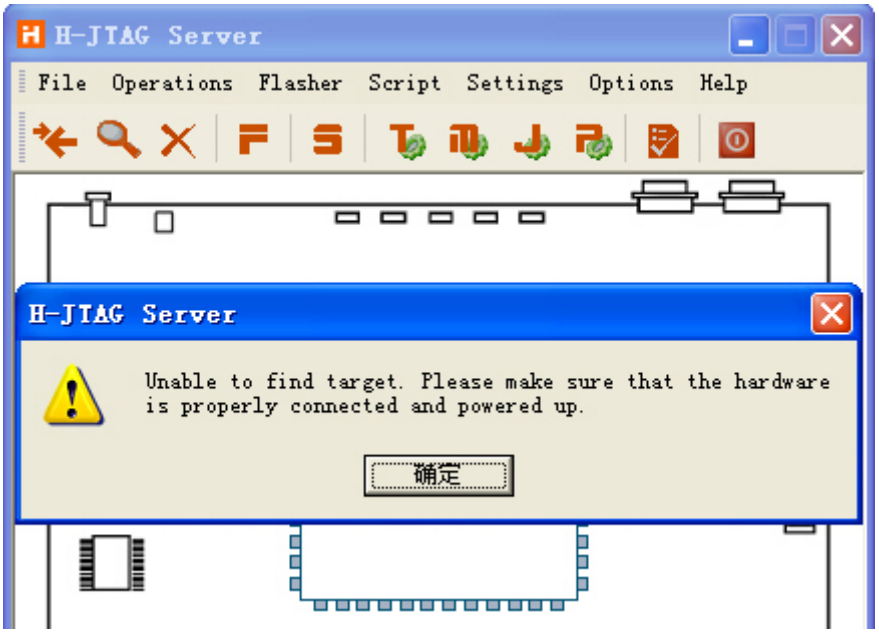


图 3.38 H-JTAG Server 连接出错

（4） 如果在ADS仿真调试时出现异常错误，一般可以重新选择EasyJTAG-H驱动，如图 3.39所示。如果还出现出错对话框，可以先把EasyJTAG-H断开，给开发板重新上电，再重新连接目标板，直到弹出连接成功界面，如图 3.40所示。

如果此时仍然连接错误，则需要检查目标板自身的电路，以及判断芯片是否加密、是否进入 Power Down 模式。

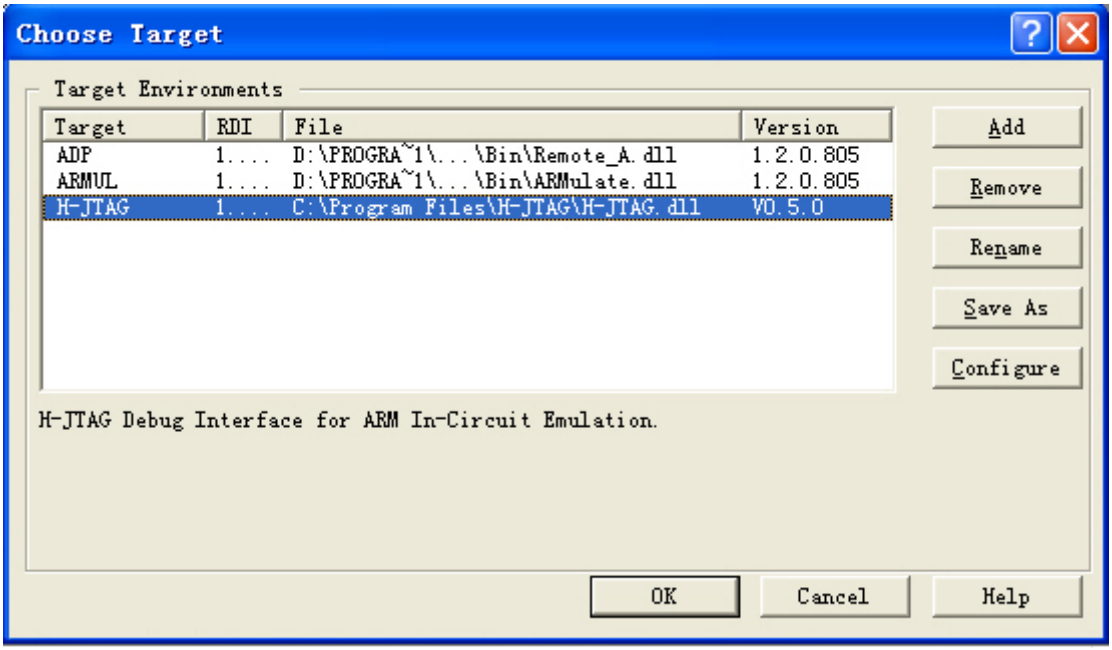


图 3.39 选择 H-JTAG 驱动

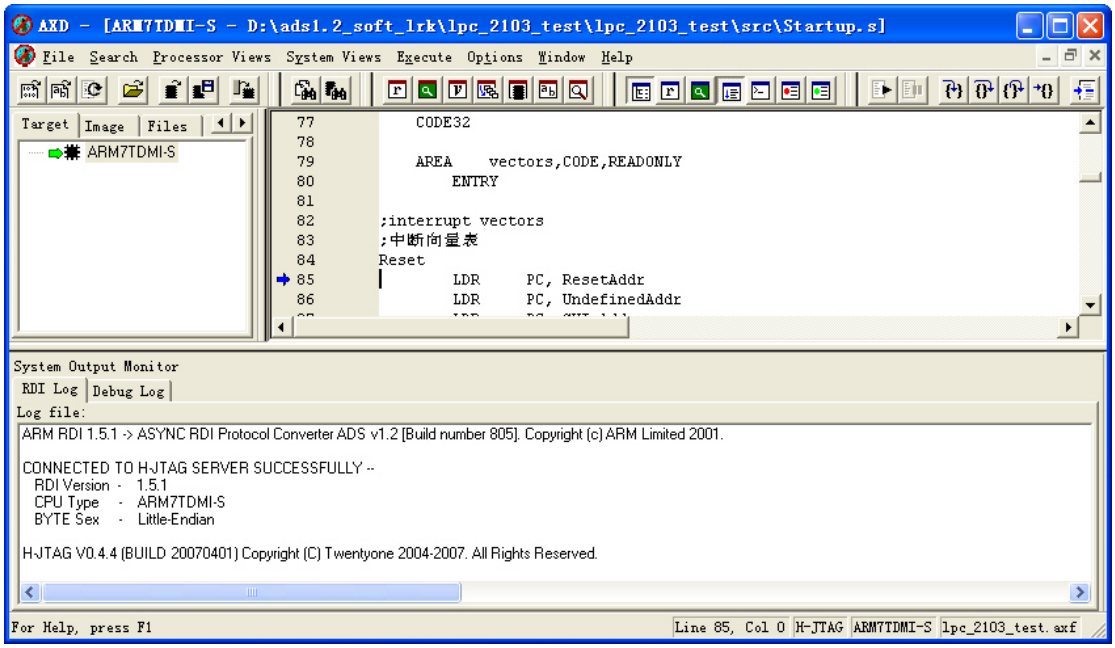


图 3.40 EasyJTAG-H 重新连接目标板

成功连接目标板后，关闭 AXD 界面，然后在 ADS 中直接 Debug 仿真调试即可。

第4章 LPC2103 功能部件详解

4.1 引脚连接模块

4.1.1 概述

LPC2103 控制器的引脚都具有多种功能，但是每个引脚在某一时刻只能选择一种功能。图 4.1是LPC2103 引脚P0.0 的一个功能选择示意图，通过配置引脚功能选择寄存器即可选择相应的功能。

当使用一个功能外设时，如果需要相应的引脚参与（如 GPIO 等），则必须在实现这一功能之前先设置好引脚的功能，否则无法实现该外设功能。

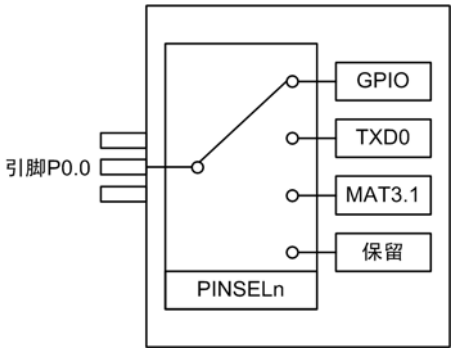


图 4.1 引脚连接模块示意图

4.1.2 寄存器描述

LPC2103 具有两个PINSEL寄存器，PINSEL0 和PINSEL1，它们都是 32 位宽度的，详细描述如表 4.1所列。

表 4.1 引脚连接模块寄存器描述

名称	描述	访问	复位值[1]	地址
PINSEL0	引脚功能选择寄存器 0	读/写	0x00000000	0xE002C000
PINSEL1	引脚功能选择寄存器 1	读/写	0x00000000	0xE002C004

[1]：复位值仅反映已使用位中保存的数据，不包含保留位的内容。

PINSEL0 和PINSEL1 寄存器中的每两个位控制着一个引脚的功能，所以一个引脚最多可以有 4 种不同的功能选择。PINSEL0 和PINSEL1 寄存器就是按照如表 4.2、表 4.3所列的设定来选择P0 口的引脚功能，使用方法如图 4.2所示。

表 4.2 引脚功能选择寄存器 0

PINSEL0	引脚名称	00	01	10	11	复位值
1:0	P0.0	GPIO P0.0	TxD0(UART0)	MAT3.1(定时器 3)	保留	00
3:2	P0.1	GPIO P0.1	RxD0(UART0)	MAT3.2(定时器 3)	保留	00
5:4	P0.2	GPIO P0.2	SCL0(I^2C0)	CAP0.0(定时器 0)	保留	00
7:6	P0.3	GPIO P0.3	SDA0(I^2C0)	MAT0.0(定时器 0)	保留	00
9:8	P0.4	GPIO P0.4	SCK0(SPI0)	CAP0.1 (定时器 0)	保留	00
11:10	P0.5	GPIO P0.5	MISO0(SPI0)	MAT0.1 (定时器 0)	保留	00

续上表

PINSEL0	引脚名称	00	01	10	11	复位值
13:12	P0.6	GPIO P0.6	MOSI0(SPI0)	CAP0.2(定时器 0)	保留	00
15:14	P0.7	GPIO P0.7	SSEL0(SPI0)	MAT2.0(定时器 2)	保留	00
17:16	P0.8	GPIO P0.8	TxD1 (UART1)	MAT2.1(定时器 2)	保留	00
19:18	P0.9	GPIO P0.9	RxD1(UART1)	MAT2.2(定时器 2)	保留	00
21:20	P0.10	GPIO P0.10	RTS1 (UART1)	CAP1.0(定时器 1)	AIN3	00
23:22	P0.11	GPIO P0.11	CTS1(UART1)	CAP1.1 (定时器 1)	AIN4	00
25:24	P0.12	GPIO P0.12	DSR1(UART1)	MAT1.0 (定时器 1)	AIN5	00
27:26	P0.13	GPIO P0.13	保留	MAT1.1 (定时器 1)	DTR1 (UART1)	00
29:28	P0.14	GPIO P0.14	EINT1	SCK1(SSP1)	DCD1 (UART1)	00
31:30	P0.15	GPIO P0.15	EINT2	保留	RI1 (UART1)	00

表 4.3 引脚功能选择寄存器 1

PINSEL1	引脚名称	00	01	10	11	复位值
1:0	P0.16	GPIO P0.16	EINT0	MAT0.2(定时器 0)	保留	00
3:2	P0.17	GPIO P0.17	SCL1 (I^2C1)	CAP1.2 (定时器 1)	保留	00
5:4	P0.18	GPIO P0.18	SDA1 (I^2C1)	CAP1.3 (定时器 1)	保留	00
7:6	P0.19	GPIO P0.19	MISO1 (SPI1)	MAT1.2(定时器 1)	保留	00
9:8	P0.20	GPIO P0.20	MOSI1 (SPI1)	MAT1.3(定时器 1)	保留	00
11:10	P0.21	GPIO P0.21	SSEL1 (SPI1)	MAT3.0(定时器 3)	保留	00
13:12	P0.22	GPIO P0.22	保留	保留	AIN0	00
15:14	P0.23	GPIO P0.23	保留	保留	AIN1	00
17:16	P0.24	GPIO P0.24	保留	保留	AIN2	00
19:18	P0.25	GPIO P0.25	保留	保留	AIN6	00
21:20	P0.26	GPIO P0.26	保留	保留	AIN7	00
23:22	P0.27	GPIO P0.27	TRST (JTAG)	CAP2.0(定时器 2)	保留	01
25:24	P0.28	GPIO P0.28	TMS (JTAG)	CAP2.1(定时器 2)	保留	01
27:26	P0.29	GPIO P0.29	TCK (JTAG)	CAP2.2(定时器 2)	保留	01
29:28	P0.30	GPIO P0.30	TDI (JTAG)	MAT3.3(定时器 3)	保留	01
31:30	P0.31	GPIO P0.31	TDO (JTAG)	保留	保留	00

说明：表中的“PINSELx 栏表示 PINSELx 存器的控制位，“引脚名称”栏表示控制位所控制的引脚，“00/01/10/11”栏表示控制位在这些设定值时的引脚功能。比如，P0.0 脚，控制位为 PINSEL0[1:0]，当 PINSEL0[1:0]=00 时引脚为 GPIO 功能(即 P0.0)，当 PINSEL0[1:0]=01 时引脚为 UART0 的 TxD 功能，当 PINSEL0[1:0]=10 时引脚为 MAT3.1 功能。

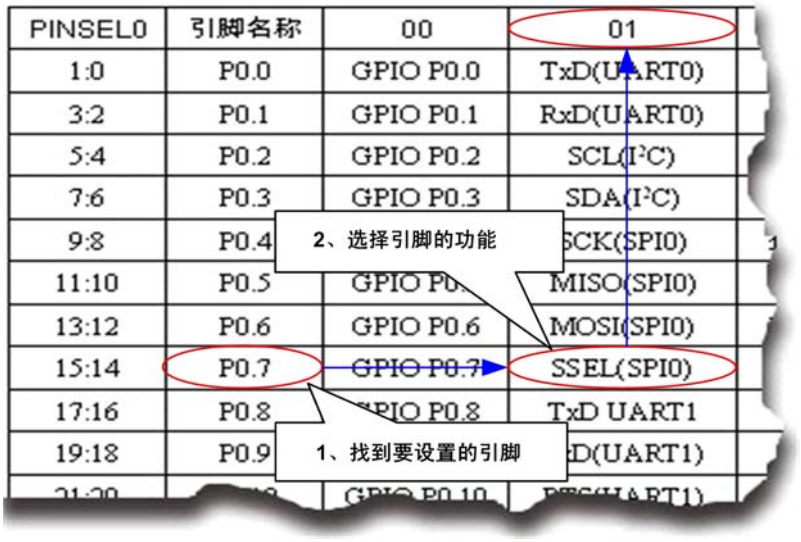


图 4.2 PINSEL 寄存器设置示意图

4.1.3 应用示例

由表 4.2可知引脚P0.0 有三种功能：GPIO、TxD0 和MAT3.1，如何设置该引脚的功能可参考下面的步骤。

1. 设置P0.0 为GPIO功能

```
PINSEL0 = 0x00;                                    /* 设置 P0.0～P0.15 都为 GPIO      */
```

上面的设置已经达到了设置 P0.0 为 GPIO 的目的，但同时也改变了 P0.1~P0.15 的引脚功能，为了不改变其它引脚的功能，可参照下面的设置。

```
PINSEL0 = PINSEL0 & 0xFFFFFFFC;                    /* 设置 P0.0 为 GPIO               */
```

2. 设置P0.0 为TxD0 功能

```
PINSEL0 = (PINSEL0 & 0xFFFFFFFC) | 0x01;           /* 设置 P0.0 的功能为 TxD0         */
```

为了不更改原先的引脚功能，可以采取“读取－修改－回写”的方式进行，即先读取寄存器值，然后再进行逻辑“与”、“或”操作，再回写到寄存器，可见如下例子。

```
PINSEL0 = (PINSEL0 & 0xFFF0FFFF) | (0x05 << 16);   /* 设置 P0.8/P0.9 的功能为 TxD1/RxD1*/
```

4.2 GPIO

4.2.1 概述

LPC2103 只有 1 个 32 位的通用 I/O 口 P0[31：0]，由于与引脚的其它功能复用，在使用前要进行相关的引脚设置，然后才能进行操作。其中，P0.27～P0.31 是 JTAG 调试引脚，在复位时，如果 DEBUG 引脚为高，则 P0[31：27]是不能作为 GPIO 使用的，只能作为 JTAG 调试引脚；反之，如果复位时 DEBUG 引脚为低，则 P0[31：27]引脚可以由用户设置，此时，调试禁止。

LPC2103 的 GPIO 有两种模式：高速 GPIO 和低速 GPIO（相对高速而言）。高速 GPIO 的控制寄存器位于 CPU 的局部总线上，可进行高速的读写操作，而低速 GPIO 的控制寄存器是挂在 VPB 总线上。P0 口作为高速 GPIO 使用时，将不能在调试环境下观察 GPIO 在 VPB 总线上的寄存器。

4.2.2 寄存器描述

LPC2103 的GPIO由表 4.4、表 4.5、表 4.6中所列的寄存器来控制。

表 4.4所列寄存器是用来选择GPIO的操作模式。当GPIO0M位的值为 0 时，选择低速GPIO，当GPIO0M位的值为 1 时，选择高速GPIO。在使用引脚的GPIO时，必须选择GPIO的操作模式。

表 4.5所列是低速GPIO的相关寄存器，挂在VPB总线上。而表 4.6所列是高速GPIO的相关寄存器，挂在局部总线上，对挂在局部总线上的这些寄存器的访问，就如访问片内存储器一样快速。

表 4.4 系统控制和状态标志寄存器（SCS-0xE01F C1A0）

位	名称	描述	复位值
0	GPIO0M	0：低速 GPIO 在 GPIO 端口 0 上使能，通过 VPB 地址访问 GPIO 端口 0，与先前的 LCP2000 器件兼容	0
		1：高速 GPIO 在 GPIO 端口 0 上使能，通过片内存储器范围的地址来访问，该模式包括端口屏蔽特性	
31：1	-	保留，用户软件不要向其写入 1。从保留位读出的值未被定义	NA

表 4.5 GPIO 控制寄存器（通过 VPB 总线访问的寄存器）

通用名称	描述	访问	复位	PORT0 地址&名称
IOPIN	GPIO 端口管脚值寄存器。不管管脚的方向如何设定，GPIO 配置端口管脚的当前状态都可从该寄存器中读出	R/W	NA	0xE002 8000 IO0PIN
IOSET	GPIO 端口输出置位寄存器。该寄存器和 IOCLR 寄存器一起控制输出管脚的状态。写入 1 使对应管脚输出高电平。写入 0 无效	R/W	0	0xE002 8004 IO0SET
IODIR	GPIO 端口方向控制寄存器。该寄存器单独控制每个 I/O 口的方向	R/W	0	0xE002 8008 IO0DIR
IOCLR	GPIO 端口输出清零寄存器。该寄存器控制输出管脚的状态。写入 1 使对应管脚输出低电平并清零 IOSET 寄存器中的对应位。写入 0 无效	WO	0	0xE002 800C IO0CLR

表 4.6 GPIO 控制寄存器（通过局部总线访问的寄存器）

通用名称	描述	访问	复位	PORT0 地址&名称
FIODIR	高速 GPIO 端口方向控制寄存器。该寄存器单独控制每个端口管脚的方向	R/W	0	0x3FFFC000 FIO0DIR
FIOPIN	使用 FIOMASK 的高速端口管脚值寄存器。不管管脚方向或交替的功能选择如何，可从该寄存器中读取数字端口管脚的当前状态（只要管脚不配置为 ADC 的输入）。读出的值通过和 FIOMASK 相与来屏蔽。写该寄存器，在 FIOMASK 中由 0 使能的所有位中放置对应的值	R/W	0	0x3FFF C014 FIO0PIN
FIOMASK	端口的高速屏蔽寄存器。写，设置，清除和读端口（通过写入 FIOPIN， FIOSET 和 FIOCLR，以及读 FIOPIN 完成）改变或返回仅在寄存器中被 0 使能的位	R/W	0	0x3FFFC010 FIO0MASK

续上表

通用名称	描述	访问	复位	PORT0 地址&名称
FIOSET	使用 FIOMASK 的高速端口输出设置寄存器。该寄存器控制输出管脚的状态。写 1 在相应的端口管脚产生高电平。写 0 无效。读该寄存器返回端口输出寄存器的当前内容	R/W	0	0x3FFF C018 FIO0SET
FIOCLR	使用 FIOMASK 的高速端口输出清零寄存器。该寄存器控制输出管脚的状态。写 1 在相应的端口管脚产生低电平。写 0 无效	WO	0	0x3FFF C01C FIO0CLR

高速GPIO和低速GPIO在LPC2103 结构中的位置，可以从图 4.3观察。

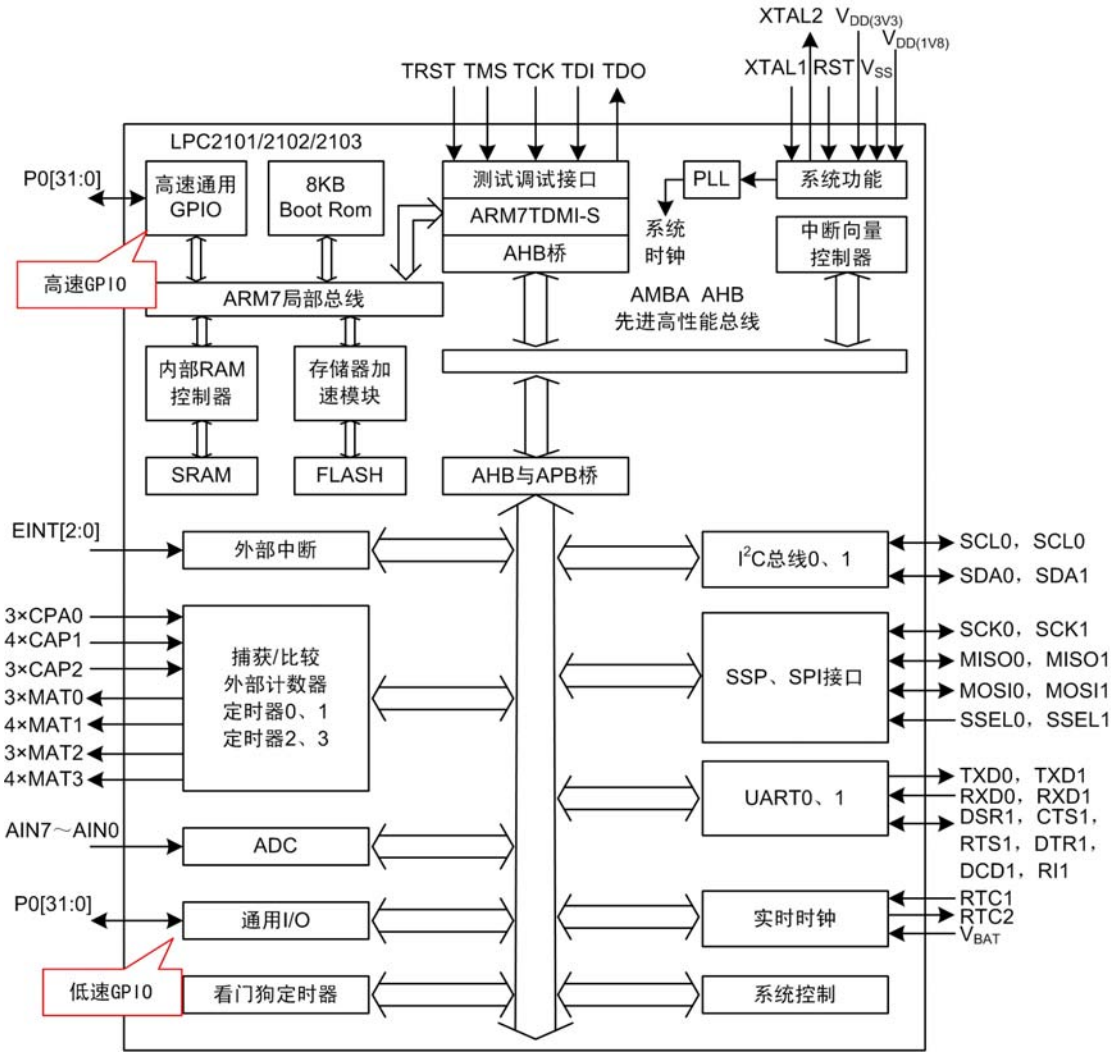


图 4.3 LPC2103 结构图

1. GPIO引脚值寄存器IOPIN

这个寄存器提供引脚作为数字信号时的逻辑值，而不管分配给引脚的功能是 GPIO，还是 PWM 等。但是，如果引脚作为模拟信号时（如作为 A/D 转换器的输入），此时寄存器中对应该引脚的值是无效的。

向寄存器 IOPIN 写入相应的值，即可确定引脚的状态，而不需要通过寄存器 IOSET 和

IOCLR 来完成。

低速GPIO的引脚值寄存器为IO0PIN，高速GPIO的引脚值寄存器为FIO0PIN。若通过FIOPIN寄存器来操作功能引脚时，需要配合FIOMASK寄存器。IO0PIN寄存器的描述如表4.7所示，FIO0PIN寄存器的描述如表 4.8所示。

表 4.7 GPIO 端口 0 引脚值寄存器（IO0PIN-0xE002 8000）

位	符号	描述	复位值
31:0	P0nVAL	反映低速 GPIO 引脚的值，寄存器中的 bit0～bit31 分别对应 P0.0~P0.31	NA

表 4.8 快速 GPIO 端口 0 引脚值寄存器（FIO0PIN-0x3FFF C014）

位	符号	描述	复位值
31:0	FP0nVAL	反映高速 GPIO 引脚的值，寄存器中的 bit0～bit31 分别对应 P0.0~P0.31	NA

2. GPIO端口 0 输出置位寄存器IOSET

当引脚配置为 GPIO 输出时，该寄存器可以使引脚输出高电平。向该寄存器写 1，对应的引脚将会输出高电平；向该寄存器写 0，无效。当引脚功能选择为 GPIO 输入或其它功能时，写 IOSET 无效。读该寄存器将返回相应的值，此值由以前对寄存器 IOSET 和 IOCLR（或 IOPIN）的写操作决定，此值不会反映外部环境对引脚的影响。

低速GPIO的输出置位寄存器为IO0SET，高速GPIO的输出置位寄存器为FIO0SET，通过FIOSET来设置引脚的输出时，需要跟FIOMASK寄存器配合。低速GPIO的输出置位寄存器的描述如表 4.9所示，高速GPIO的输出置位寄存器的描述如表 4.10所示。

表 4.9 低速 GPIO 输出置位寄存器（IO0SET-0xE002 8004）

位	符号	描述	复位值
31:0	P0nSET	写 1，对应位的引脚输出高电平，位 0～位 31 分别对应 P0.0～P0.31	0

表 4.10 高速 GPIO 输出置位寄存器（FIO0SET-0x3FFF C018）

位	符号	描述	复位值
31:0	FP0nSET	写 1，对应位的引脚输出高电平，位 0～位 31 分别对应 P0.0～P0.31	0

高速GPIO除了一个 32 位的输出置位寄存器之外，还有 4 个 8 位字节输出置位寄存器和 2 个 16 位半字输出置位寄存器。它们的功能同 32 位输出置位寄存器一样，它们的描述如表4.11所示。

表 4.11 高速 GPIO 端口 0 输出置位字节和半字寄存器描述

寄存器名称	寄存器长（位）	地址	描述	复位值
FIO0SET0	8（字节）	0x3FFF C018	写 1，对应位的引脚输出高电平，位 0～位 7 分别对应 P0.0～P0.7	0
FIO0SET1	8（字节）	0x3FFF C019	写 1，对应位的引脚输出高电平，位 0～位 7 分别对应 P0.8～P0.15	0
FIO0SET2	8（字节）	0x3FFF C01A	写 1，对应位的引脚输出高电平，位 0～位 7 分别对应 P0.16～P0.23	0

续上表

寄存器名称	8（字节）	地址	描述	复位值
FIO0SET3	8（字节）	0x3FFF C01B	写 1，对应位的引脚输出高电平，位 0～位 7 分别对应 P0.24～P0.31	0
FIO0SETL	16（半字）	0x3FFF C018	写 1，对应位的引脚输出高电平，位 0～位 15 分别对应 P0.0～P0.15	0
FIO0SETU	16（半字）	0x3FFF C01A	写 1，对应位的引脚输出高电平，位 0～位 15 分别对应 P0.16～P0.31	0

3. GPIO端口 0 方向寄存器IODIR

当引脚选择GPIO功能时，使用该寄存器可以控制引脚的方向。只有当引脚选择GPIO功能时，IODIR的设置才有意义。低速GPIO的方向控制寄存器为IO0DIR，高速GPIO的方向控制寄存器为FIO0DIR，如表 4.12、表 4.13所示。

表 4.12 低速 GPIO 端口 0 的方向控制寄存器（IO0DIR-0xE002 8008）

位	符号	描述	复位值
31:0	P0nDIR	低速 GPIO 方向控制位，位 0～位 31 分别对应 P0.0～P0.31 0：控制引脚为输入 1：控制引脚为输出	0

表 4.13 高速 GPIO 端口 0 方向控制寄存器（FIO0DIR-0x3FFF C000）

位	符号	描述	复位值
31:0	FP0nDIR	高速 GPIO 方向控制位，位 0～位 31 分别对应 P0.0～P0.31 0：控制引脚为输入 1：控制引脚为输出	0

高速GPIO除了一个 32 位的方向控制寄存器之外，还有 4 个 8 位字节方向控制寄存器和 2 个 16 位半字方向控制寄存器。它们的功能同 32 位方向控制寄存器一样，它们的描述如表 4.14所示，这些寄存器可以使高速GPIO端口的访问速度更快。

表 4.14 高速 GPIO 端口 0 方向控制字节和半字寄存器描述

寄存器名称	寄存器长度（位）	地址	描述	复位值
FIO0DIR0	8（字节）	0x3FFFC000	高速 GPIO 方向控制位，位 0～位 7 分别对应 P0.0～P0.7 0：控制引脚为输入 1：控制引脚为输出	0
FIO0DIR1	8（字节）	0x3FFFC001	高速 GPIO 方向控制位，位 0～位 7 分别对应 P0.8～P0.15 0：控制引脚为输入 1：控制引脚为输出	0
FIO0DIR2	8（字节）	0x3FFFC002	高速 GPIO 方向控制位，位 0～位 7 分别对应 P0.16～P0.23 0：控制引脚为输入 1：控制引脚为输出	0

续上表

寄存器名称	寄存器长度（位）	地址	描述	复位值
FIO0DIR3	8（字节）	0x3FFFC003	高速 GPIO 方向控制位，位 0～位 7 分别对应 P0.24～P0.31 0：控制引脚为输入 1：控制引脚为输出	0
FIO0DIRL	16（半字）	0x3FFFC000	高速 GPIO 方向控制位，位 0～位 15 分别对应 P0.0～P0.15 0：控制引脚为输入 1：控制引脚为输出	0
FIO0DIRU	16（半字）	0x3FFFC002	高速 GPIO 方向控制位，位 0～位 15 分别对应 P0.16～P0.31 0：控制引脚为输入 1：控制引脚为输出	0

4. GPIO输出清零寄存器IOCLR

当引脚配置为 GPIO 输出模式时，该寄存器可使引脚输出低电平。向寄存器相应位写 1，则对应的引脚将输出低电平并清零 IOSET 寄存器中相应的位；写 0，无效。当引脚配置为 GPIO 输入或其它功能时，写 IOCLR 无效。

低速GPIO的输出清零寄存器为IO0CLR，如表 4.15所示。高速GPIO的输出寄存器为FIO0CLR，如表 4.16所示。当通过FIOCLR寄存器来操作高速GPIO时，需要配合FIOMASK寄存器。

表 4.15 低速 GPIO 端口 0 输出清零寄存器（IO0CLR-0xE002 800C）

位	符号	描述	复位值
31:0	P0nCLR	写 1，对应位的引脚输出低电平，位 0～位 31 分别对应 P0.0～P0.31	0x0000 0000

表 4.16 高速 GPIO 端口 0 输出清零寄存器（FIO0CLR-0x3FFF C01C）

位	符号	描述	复位值
31:0	FP0nCLR	写 1，对应位的引脚输出低电平，位 0～位 31 分别对应 P0.0～P0.31	0x0000 0000

高速GPIO除了一个 32 位的输出清零寄存器之外，还有 4 个 8 位字节输出清零寄存器和 2 个 16 位半字输出清零寄存器。它们的功能同 32 位输出清零寄存器一样，它们的描述如表 4.17所示，这些寄存器可以使高速GPIO端口的访问速度更快。

表 4.17 高速 GPIO 端口 0 输出清零字节和半字寄存器描述

寄存器名称	寄存器长度（位）	地址	描述	复位值
FIO0CLR0	8（字节）	0x3FFF C01C	写 1，对应位的引脚输出低电平，位 0～位 7 分别对应 P0.0～P0.7	0x00
FIO0CLR1	8（字节）	0x3FFF C01D	写 1，对应位的引脚输出低电平，位 0～位 7 分别对应 P0.8～P0.15	0x00
FIO0CLR2	8（字节）	0x3FFF C01E	写 1，对应位的引脚输出低电平，位 0～位 7 分别对应 P0.16～P0.23	0x00

续上表

寄存器名称	寄存器长度（位）	地址	描述	复位值
FIO0CLR3	8（字节）	0x3FFF C01F	写 1，对应位的引脚输出低电平，位 0～位 7 分别对应 P0.24～P0.31	0x00
FIO0CLRL	16（半字）	0x3FFF C01C	写 1，对应位的引脚输出低电平，位 0～位 15 分别对应 P0.0～P0.15	0x0000
FIO0CLRU	16（半字）	0x3FFF C01E	写 1，对应位的引脚输出低电平，位 0～位 15 分别对应 P0.16～P0.31	0x0000

5. 高速GPIO端口 0 屏蔽寄存器FIOMASK

FIOMASK寄存器只有在高速GPIO的情况下才有，此寄存器可以选择端口的引脚是否受寄存器FIOPIN、FIOSET或FIOCLR写操作的影响。当寄存器写 0 时，对应位的引脚可以通过相应的寄存器来访问；当寄存器写 1 时，对应位的引脚将不响应通过寄存器FIOPIN、FIOSET或FIOCLR的写操作，对FIOPIN的读操作将不会获取对应位的引脚最新状态。该寄存器的描述如表 4.18所示。

表 4.18 高速 GPIO 端口 0 屏蔽寄存器（FIO0MASK-0x3FFF C010）

位	符号	描述	复位值
31:0	FP0nMASK	高速 GPIO 屏蔽寄存器位，位 0～位 31 对应 P0.0～P0.31； 0：允许通过写 FIOSET、FIOCLR 和 FIOPIN 寄存器来改变引脚状态，也可通过读 FIOPIN 寄存器来获取最新的引脚状态。 1：写 FIOSET、FIOCLR 和 FIOPIN 寄存器不改变引脚状态，读 FIOPIN 寄存器不能获取最新的引脚状态。	0x0000 0000

高速GPIO端口 0 除了一个 32 位的屏蔽寄存器之外，还有 4 个 8 位的字节屏蔽寄存器和 2 个 16 的半字屏蔽寄存器，它们有跟 32 位的屏蔽寄存器一样的功能，这些寄存器的描述如表 4.19所示。

表 4.19 高速 GPIO 端口 0 的屏蔽字节和半字寄存器

寄存器名称	寄存器长度（位）	地址	描述	复位值
FIO0MASK0	8（字节）	0x3FFF C010	高速 GPIO 屏蔽寄存器位，位 0～位 7 对应 P0.0～P0.7	0x00
FIO0MASK1	8（字节）	0x3FFF C011	高速 GPIO 屏蔽寄存器位，位 0～位 7 对应 P0.8～P0.15	0x00
FIO0MASK2	8（字节）	0x3FFF C012	高速 GPIO 屏蔽寄存器位，位 0～位 7 对应 P0.16～P0.23	0x00
FIO0MASK3	8（字节）	0x3FFF C013	高速 GPIO 屏蔽寄存器位，位 0～位 7 对应 P0.24～P0.31	0x00
FIO0MASKL	16（半字）	0x3FFF C010	高速 GPIO 屏蔽寄存器位，位 0～位 15 对应 P0.0～P0.15	0x0000
FIO0MASKU	16（半字）	0x3FFF C012	高速 GPIO 屏蔽寄存器位，位 0～位 15 对应 P0.16～P0.31	0x0000

4.2.3 GPIO使用注意事项

1. 顺序访问IOSET和IOCLR寄存器对同一个GPIO引脚/位的影响

GPIO 引脚配置的输出状态由写入 IOSET 和 IOCLR 寄存器的值决定。IOSET 和 IOCLR 两者中，后访问的寄存器将决定引脚的最终输出状态。见下列所示：

```
IO0DIR = 0x00000080;                    /*  设置引脚 P0.7 为输出        */
IO0CLR = 0x00000080;                    /* P0.7 输出低电平              */
IO0SET = 0x00000080;                    /* P0.7 输出高电平              */
IO0CLR = 0x00000080;                    /* P0.7 输出低电平              */
```

先将P0.7 设置为输出（写IO0DIR寄存器）；然后将P0.7 输出设置为低电平（先写IO0CLR寄存器）；接着设置P0.7 输出高电平（写IO0SET）；最后，再一次设置P0.7 输出低电平。由此可知引脚的最终输出为低电平。运行上述代码可得到一个宽度约为 360ns的高电平脉冲，如图 4.4所示。

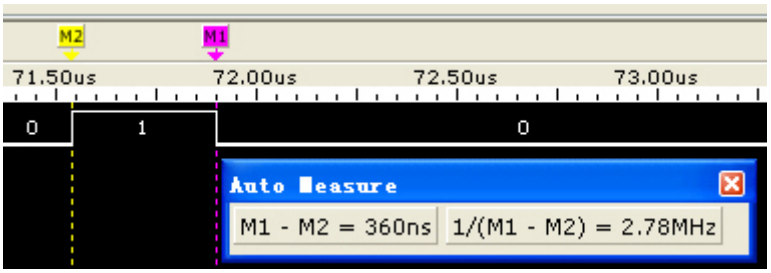


图 4.4 P0.7 引脚测试到的高电平

2. GPIO引脚输出瞬时 0 和 1

先写 IOSET 再写 IOCLR 寄存器可使管脚先输出 1 再输出 0。有的系统允许有效输出的这段延时时间。但某些应用要求一个 GPIO 口的一组管脚同时输出一个二进制数（0 和 1 混合）。这可通过写端口的 IOPIN 寄存器来实现。

下面的代码所实现的功能是：P0.[31：16]和 P0.[7：0]输出保持不变的同时将 P0.[15：8]设置成 0xA5，不管 P0.[15：8]之前是何值：

```
IO0PIN = (IO0PIN && 0xFFFF00FF) || 0x0000A500;      /* P0.7 输出低电平      */
```

使用高速 GPIO 端口可以获取同样的结果。

使用 32 位（字）寄存器访问高速 GPIO，将 P0.[15：8]的输出设置成 0xA5。

```
FIO0MASK      = 0xFFFF00FF;             /* P0.[15:7]访问使能，其它屏蔽   */
FIO0PIN       = 0x0000A500;             /* P0.[15:7]输出为 0xA5          */
```

使用 16 位（半字）寄存器访问高速 GPIO，将 P0.[15：8]的输出设置成 0xA5。

```
FIO0MASKL = 0x00FF;                     /* P0.[15:7]访问使能，其它屏蔽   */
FIO0PINL  = 0xA500;                     /* P0.[15:7]输出为 0xA5          */
```

使用 8 位（字节）寄存器访问高速 GPIO，将 P0.[15：8]的输出设置成 0xA5。下面代码的第一句可以省略，应为 FIO0MASK1 寄存器的复位值就是 0x00。

```
FIO0MASK1 = 0x00;                       /* P0.[15:7]访问使能             */
FIO0PIN1  = 0xA5;                       /* P0.[15:7]输出为 0xA5          */
```

3. IOSET/IOCLR和IOPIN的写操作

向 IOSET/IOCLR 寄存器写 1 很容易实现改变选择引脚的高/低电平输出状态，但是 IOSET/IOCLR 的写操作不可以使 GPIO 端口同时输出任意的二进制数据（混合 0 和 1）。

然而向 IOPIN 的写操作，可以使 GPIO 端口输出一个期望的值，向此寄存器写 0 和 1 都有效，写 0 时，相应引脚输出低电平；写 1 时，相应引脚输出高电平。当在设置引脚的输出时，为了不改变其它引脚的输出，应该采取“读取－修改－回写”的方式进行操作。如要求 P0.[15：8]输出 0xA5 时，可参考如下设置：

```
IO0PIN = (IO0PIN && 0xFFFF00FF) || 0x0000A500;          /* P0.7 输出低电平                    */
```

4. 高速GPIO与低速GPIO的信号输出频率

LPC2103 上的高速 GPIO 特性能够使 GPIO 引脚更好地响应控制它们的代码。而且，软件通过高速 GPIO 寄存器访问 GPIO 脚的时间比软件使用低速 GPIO 寄存器访问的时间快了近 3.5 倍。由于访问速度的增加，数字管脚的输出频率也增加了近 3.5 倍。需要说明的是，在使用 C 代码编写时，输出频率的这种极大增加有时是体现不出来的，因此，在处理高速端口输出的应用部分时，需要用汇编代码来写且在 ARM 模式中执行。

4.2.4 应用示例

1. 低速GPIO——LED灯闪烁控制实验

选择低速GPIO，控制LED灯闪烁，示例程序如程序清单 4.1所示。此示例操作需要短接 JP4 的P0.17，输出控制LED1。

程序清单 4.1 GPIO 控制 LED 闪烁

```
#include "config.h"
# define    LED1      1 << 17                                   /*  P0.17 控制 LED1                  */
/*********************************************************************************************
** 函数名称：DelayNS
** 功能描述：延时函数
** 入口参数：uiDly    值越大，延时时间越长
** 出口参数：无
*********************************************************************************************/
void DelayNS (uint32 uiDly)
{
    uint32 i;
    for (; uiDly > 0; uiDly--){
    for(i = 0; i < 50000; i++);
    }
}
/*********************************************************************************************
** 函数名称：main
** 功能描述：跳线 JP4 短接，LED1 闪烁
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
```

```
int main (void)
{
    PINSEL1 = PINSEL1 & (~(0x03 << 2));                 /*  将 P0.17 设置为 GPIO          */
    IO0DIR  = LED1;                                     /*  设置 LED 控制口为输出         */
    IO0SET  = LED1;                                     /*  LED1 熄灭                     */

    while (1) {
        IO0SET = LED1;                                  /*  LED1 熄灭                     */
        DelayNS(50);                                    /*  延时                          */
        IO0CLR = LED1;                                  /*  LED1 点亮                     */
        DelayNS(50);                                    /*  延时                          */
    }
    return 0;
}
```

2. 高速GPIO——LED灯闪烁控制实验

选择高速GPIO，控制LED灯闪烁，示例程序如程序清单 4.2所示。此示例操作需要用短接JP4 的P0.17、P0.18、P0.19、P0.20 端口，输出控制LED1、LED2、LED3 和LED4。

程序清单 4.2 高速 GPIO 控制 LED 闪烁

```
#include "config.h"

# define    LED1    1 << 17                           /*  P0.17 控制 LED1               */
# define    LED2    1 << 18                           /*  P0.18 控制 LED2               */
# define    LED3    1 << 19                           /*  P0.19 控制 LED3               */
# define    LED4    1 << 20                           /*  P0.20 控制 LED4               */
/*********************************************************************************************
** 函数名称：DelayNS
** 功能描述：延时函数
** 入口参数：uiDly   值越大，延时时间越长
** 出口参数：无
*********************************************************************************************/
void DelayNS (uint32 uiDly)
{
    uint32 i;
    for (; uiDly > 0; uiDly--){
        for(i = 0; i < 50000; i++);
    }
}
/*********************************************************************************************
** 函数名称：main
** 功能描述：跳线 JP4 短接，LED 闪烁
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
```

```
int main (void)
{
    PINSEL1  = PINSEL1 & (~(0xFF << 2));          /*  将 P0.17-P0.20 设置为 GPIO      */
    SCS      = 0x01;                              /*  设定为高速 GPIO 模式            */
    FIO0DIR  = LED1 | LED2 | LED3 | LED4;         /*  设置 LED 控制口为输出           */
    FIO0SET  = LED1 | LED2 | LED3 | LED4;         /*  LED1 熄灭                       */

    while (1) {
        FIO0SET = LED1 | LED3;                    /*  LED1、LED3 熄灭                 */
        FIO0CLR = LED2 | LED4;                    /*  LED2、LED4 点亮                 */
        DelayNS(50);                              /*  延时                            */
        FIO0CLR = LED1 | LED3;                    /*  LED1、LED3 点亮                 */
        FIO0SET = LED2 | LED4;                    /*  LED2、LED4 熄灭                 */
        DelayNS(50);                              /*  延时                            */
    }
    return 0;
}
```

3. 高速GPIO与低速GPIO比较。

高速 GPIO 的访问速度要比低速 GPIO 的访问速度快了近 3.5 倍（Fcclk=Fpclk 的条件下）。示例程序将对此进行验证，设置 P0.0 为低速 GPIO，P0.1 为高速 GPIO。

- 首先设置Fpclk等于Fcclk（约 44MHz），如图 4.5所示。

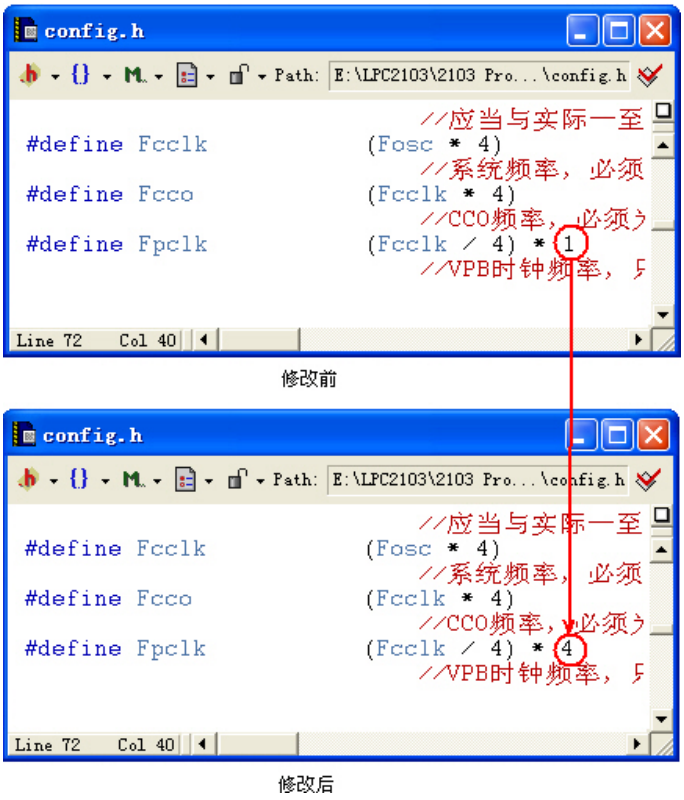


图 4.5 修改配置文件 config.h

- 设置引脚输出，如程序清单 4.3所示。

程序清单 4.3 高速 GPIO 与低速 GPIO 的比较程序

```
/*********************************************************************************************
** 函数名称：main
** 功能描述：快速 GPIO 与低速 GPIO 的比较
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
int main (void)
{
    while(1) {
        SCS      = SCS & 0xFFFFFFFE;          /* 选择低速 GPIO 模式          */
        PINSEL0  = PINSEL0 & 0xFFFFFFF0;      /* 选择 P0.[1:0]为 GPIO 功能   */
        IO0DIR   = 0x01;                      /* 设置 P0.0 为输出            */
        IO0SET   = 0x01;                      /* 使 P0.0 输出高电平          */
        IO0CLR   = 0x01;                      /* 使 P0.0 输出低电平          */
        IO0SET   = 0x01;                      /* 使 P0.0 输出高电平          */
        IO0CLR   = 0x01;                      /* 使 P0.0 输出低电平          */
        SCS      = SCS | 0x01;                /* 选择高速 GPIO 模式          */
        PINSEL0  = PINSEL0 & 0xFFFFFFF0;      /* 选择 P0.[1:0]为 GPIO 功能   */
        FIO0MASK = 0x00;                      /* 使能 P0.1 可以被相应寄存器修改 */
        FIO0DIR  = 0x02;                      /* 设置 P0.1 为输出            */
        FIO0SET  = 0x02;                      /* 使 P0.1 输出高电平          */
        FIO0CLR  = 0x02;                      /* 使 P0.1 输出低电平          */
        FIO0SET  = 0x02;                      /* 使 P0.1 输出高电平          */
        FIO0CLR  = 0x02;                      /* 使 P0.1 输出低电平          */
    }
    return 0;
}
```

- 上述程序运行后，逻辑分析仪采集的波形如图 4.6所示，在Fpclk等于Fcclk的条件下，高速GPIO的P0.1 输出频率是低速GPIO P0.0 的 3.5 倍；

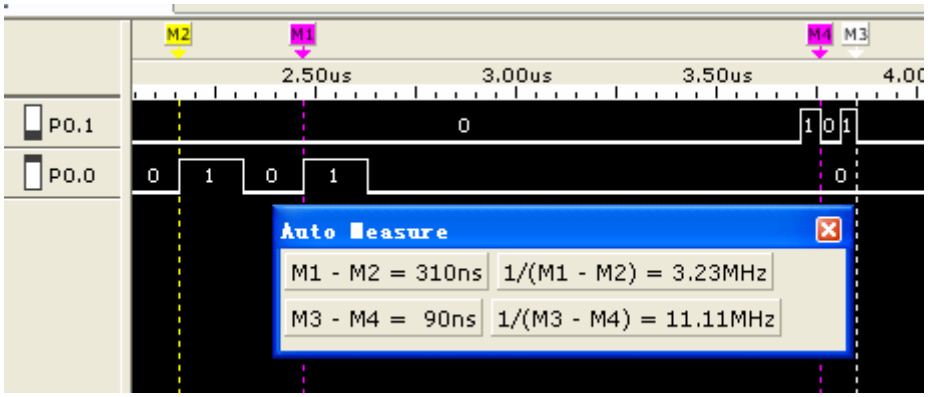


图 4.6 高速 GPIO 与低速 GPIO 的比较结果 1

- 在Fpclk=Fcclk/4 的条件下，比较的结果如图 4.7所示。

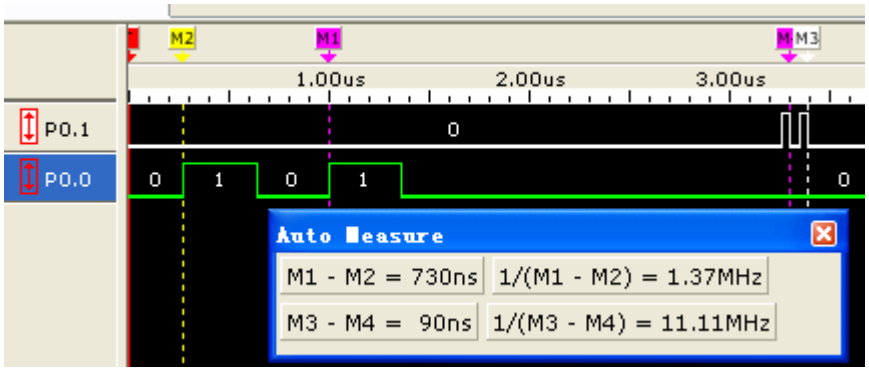


图 4.7 高速 GPIO 与低速 GPIO 的比较结果 2

4.3 向量中断控制器

4.3.1 概述

向量中断控制器 VIC（Vectored Interrupt Controller）具有 32 个中断请求输入。可将这些中断编程分为 3 类：FIQ、向量 IRQ、非向量 IRQ。

快速中断请求 FIQ（Fast Interrupt reQuest）具有最高的优先级。如果分配给 FIQ 的请求超过 1 个，VIC 将中断请求相“或”后向 ARM 处理器产生 FIQ 信号。当只有 1 个中断被分配为 FIQ 时，可实现最短的 FIQ 等待时间，因为 FIQ 服务程序只要简单地启动器件的处理即可。但分配给 FIQ 级的中断多于 1 个时，FIQ 服务程序需要读取 FIQ 状态寄存器来识别产生中断请求的 FIQ 中断源。

向量 IRQ（Vectored IRQ）具有中等优先级。该级别可分配 32 个中断请求中的 16 个。32 个请求中的任意一个都可分配到 16 个向量 IRQ slot 中的任意一个，其中 slot0 具有最高优先级，而 slot15 则为最低优先级。

非向量 IRQ（Non-vectored IRQ）的优先级最低。VIC 将所有向量 IRQ 和非向量 IRQ 相“或”，向 ARM 处理器产生 IRQ 信号。IRQ 服务程序可通过读取 VIC 的一个寄存器来启动并跳转到相应的地址。如果有任意一个向量 IRQ 发出请求，VIC 则提供最高优先级请求 IRQ 服务程序的地址；否则提供所默认程序的地址。该默认程序由所有非向量 IRQ 共用。默认程序可读取 IRQ 状态寄存器，来确定哪个 IRQ 被激活。

VIC 中所有的寄存器都为字寄存器，不支持字节和半字的读写操作。

4.3.2 特性

向量中断控制器的特性如下：

- ARM PrimeCell™ 向量中断控制器；
- 32 个中断请求输入；
- 16 个向量 IRQ 中断；
- 16 个优先级，可动态分配给中断请求；
- 可产生软件中断。

4.3.3 寄存器描述

VIC所包含的寄存器如表 4.20所列。

表 4.20 VIC 控制寄存器

名称	描述	访问	复位值	地址
VICIRQStatus	IRQ 状态寄存器，该寄存器读出定义为 IRQ 且使能的中断的状态	RO	0	0xFFFF F000
VICFIQStatus	FIQ 状态寄存器，该寄存器读出定义为 FIQ 并使能的中断的状态	RO	0	0xFFFF F004
VICRawIntr	所有中断的状态寄存器。该寄存器读出 32 个中断请求/软件中断的状态，不管中断是否使能或分类	RO	0	0xFFFF F008
VICIntSelect	中断选择寄存器。该寄存器将 32 个中断请求分配为 FIQ 或 IRQ	R/W	0	0xFFFF F00C
VICIntEnable	中断使能寄存器。该寄存器控制 32 个中断和软件中断的使能	R/W	0	0xFFFF F010
VICIntEnClear	中断使能清零寄存器。该寄存器允许将中断使能寄存器的一个或多个位清零	W	0	0xFFFF F014
VICSoftInt	软件中断寄存器。该寄存器的内容与 32 个不同外设的中断请求相“或”	R/W	0	0xFFFF F018
VICSoftIntClear	软件中断清零寄存器。该寄存器允许将软件中断寄存器中的一个或多个位清零	W	0	0xFFFF F01C
VICProtection	保护使能寄存器。该寄存器允许特权模式下运行的软件对 VIC 寄存器进行有限的访问	R/W	0	0xFFFF F020
VICVectAddr	向量地址寄存器。当发生一个 IRQ 中断时，IRQ 服务程序可读出该寄存器并跳转到读出的地址	R/W	0	0xFFFF F030
VICDefVecAddr	默认向量地址寄存器。该寄存器保存了非向量 IRQ 的中断服务程序（ISR）地址	R/W	0	0xFFFF F034
VICVectAddr0	向量地址 0 寄存器。向量地址寄存器 0～15 保存了 16 个向量 IRQ slot 的中断服务地址	R/W	0	0xFFFF F100
VICVectAddr1	向量地址 1 寄存器	R/W	0	0xFFFF F104
VICVectAddr2	向量地址 2 寄存器	R/W	0	0xFFFF F108
VICVectAddr3	向量地址 3 寄存器	R/W	0	0xFFFF F10C
VICVectAddr4	向量地址 4 寄存器	R/W	0	0xFFFF F110
VICVectAddr5	向量地址 5 寄存器	R/W	0	0xFFFF F114
VICVectAddr6	向量地址 6 寄存器	R/W	0	0xFFFF F118
VICVectAddr7	向量地址 7 寄存器	R/W	0	0xFFFF F11C
VICVectAddr8	向量地址 8 寄存器	R/W	0	0xFFFF F120
VICVectAddr9	向量地址 9 寄存器	R/W	0	0xFFFF F124
VICVectAddr10	向量地址 10 寄存器	R/W	0	0xFFFF F128
VICVectAddr11	向量地址 11 寄存器	R/W	0	0xFFFF F12C
VICVectAddr12	向量地址 12 寄存器	R/W	0	0xFFFF F130
VICVectAddr13	向量地址 13 寄存器	R/W	0	0xFFFF F134
VICVectAddr14	向量地址 14 寄存器	R/W	0	0xFFFF F138
VICVectAddr15	向量地址 15 寄存器	R/W	0	0xFFFF F13C

续上表

名称	描述	访问	复位值	地址
VICVectCntl0	向量控制 0 寄存器。向量控制寄存器 0～15 分别控制 16 个向量 IRQ slot 中的一个。slot0 优先级最高，而 slot15 优先级最低	R/W	0	0xFFFF F200
VICVectCntl1	向量控制 1 寄存器	R/W	0	0xFFFF F204
VICVectCntl2	向量控制 2 寄存器	R/W	0	0xFFFF F208
VICVectCntl3	向量控制 3 寄存器	R/W	0	0xFFFF F20C
VICVectCntl4	向量控制 4 寄存器	R/W	0	0xFFFF F210
VICVectCntl5	向量控制 5 寄存器	R/W	0	0xFFFF F214
VICVectCntl6	向量控制 6 寄存器	R/W	0	0xFFFF F218
VICVectCntl7	向量控制 7 寄存器	R/W	0	0xFFFF F21C
VICVectCntl8	向量控制 8 寄存器	R/W	0	0xFFFF F220
VICVectCntl9	向量控制 9 寄存器	R/W	0	0xFFFF F224
VICVectCntl10	向量控制 10 寄存器	R/W	0	0xFFFF F228
VICVectCntl11	向量控制 11 寄存器	R/W	0	0xFFFF F22C
VICVectCntl12	向量控制 12 寄存器	R/W	0	0xFFFF F230
VICVectCntl13	向量控制 13 寄存器	R/W	0	0xFFFF F234
VICVectCntl14	向量控制 14 寄存器	R/W	0	0xFFFF F238
VICVectCntl15	向量控制 15 寄存器	R/W	0	0xFFFF F23C

1. 软件中间寄存器（VICSoftInt-0xFFFF F018）

在执行任何逻辑之前，将该寄存器的内容与 32 个不同外设的中断请求相“或”。软件中断寄存器的描述如表 4.21所列。

表 4.21 软件中断寄存器描述

位	符号	描述	复位值
31:0	VICSoftInt	0：不强制产生对应该位中断请求，向此寄存器写 0 无效 1：强制产生与该位相关的中断请求	0

软件中断寄存器中的位与外设中断的分配关系如表 4.22所列。

表 4.22 软件中断寄存器位分配

位	31	30	29	28	27	26	25	24
对应外设	-	-	-	-	TIMER3	TIMER2	-	-
位	23	22	21	20	19	18	17	16
对应外设	-	-	-	-	I^2C1	AD0	-	EINT2
位	15	14	13	12	12	10	9	8
对应外设	EINT1	EINT0	RTC	PLL	SSP/SPI1	SPI0	I^2C0	-
位	7	6	5	4	3	2	1	0
对应外设	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT

2. 软件中断清零寄存器（VICSoftIntClear-0xFFFF F01C）

该寄存器可以不读取软件中断寄存器的值，而对软件中断寄存器的一个或多个位进行清

零。软件中断清零寄存器的描述如表 4.23所示，其中该寄存器的位与功能外设的分配关系跟软件中断寄存器一样，如表 4.22所列。

表 4.23 软件中断清零寄存器

位	符号	描述	复位值
31:0	VICSoftIntClear	0：写 0，不改变 VICSoftInt 中的相应位 1：写 1，清零软件中断寄存器的相应位，并解除强制的中断请求	0

3. 所有中断状态寄存器（VICRawIntr-0xFFFF F008）

这是一个只读寄存器，读取所有 32 个中断请求和软件中断的状态，而不管中断是否使能或分类。软件中断清零寄存器的描述如表 4.24所示，寄存器的位分配关系如表 4.22所列。

表 4.24 软件中断清零寄存器描述

位	符号	描述	复位值
31:0	VICSoftIntClear	0：写 0，不改变 VICSoftInt 中的相应位 1：写 1，清零软件中断寄存器的相应位，并解除强制的中断请求	0

4. 中断使能寄存器（VICIntEnable-0xFFFF F010）

该寄存器对分配为FIQ或IRQ的中断请求或软件中断进行使能。中断使能寄存器的描述如表 4.25所示，其中寄存器的位分配关系如表 4.22所列。

表 4.25 中断使能寄存器描述

位	符号	描述	复位值
31:0	VICIntEnable	读该寄存器时，相应位为 1 表示中断请求使能为 FIQ 或 IRQ 0：写 0，对寄存器的位没有影响 ； 1：写 1，使能被分配为 FIQ 或 IRQ 的中断请求或软件中断	0

5. 中断使能清零寄存器（VICIntEnClear-0xFFFF F014）

该寄存器可以在不读中断使能寄存器情况下，允许软件清除一个或多个中断使能寄存器的位，中断使能清零寄存器的描述如表 4.26所示，其寄存器位的分配关系如表 4.22所列。

表 4.26 中断使能清零寄存器描述

位	符号	描述	复位值
31:0	VICIntEnClear	0：写 0，对寄存器的位没有影响 ； 1：写 1，清零中断使能寄存器中的相应位，同时禁止该中断请求	0

6. 中断选择寄存器（VICIntSelect-0xFFFF F00C）

该寄存器把 32 个中断请求分配为FIQ或IRQ。中断选择寄存器的描述如表 4.27所示，寄存器位的分配关系如表 4.22所列。

表 4.27 中断选择寄存器

位	符号	描述	复位值
31:0	VICIntSelect	0：写 0，对应位的中断请求被分配为 IRQ 1：写 1，对应位的中断请求被分配为 FIQ	0

7. IRQ状态寄存器（VICIRQStatus -0xFFFF F000）

该寄存器读取使能并分配IRQ的中断请求状态，它不对向量IRQ和非向量IRQ进行区分，IRQ中断状态寄存器的描述如表 4.28所示，寄存器位的分配关系如表 4.22所列。

表 4.28 IRQ 中断状态寄存器

位	符号	描述	复位值
31:0	VICIRQStatus	1：对应位的中断请求使能，并被分配为 IRQ	0

8. FIQ状态寄存器（VICFIQStatus-0xFFFF F004）

该寄存器读出使能并被分配为FIQ的中断请求的状态，如果有超过一个请求分配为FIQ，FIQ服务程序可读取该寄存器来确定哪一个（几个）请求被激活。FIQ状态寄存器的描述如表 4.29所列。

表 4.29 FIQ 状态寄存器

位	符号	描述	复位值
31:0	VICFIQStatus	1：对应位的中断请求使能，并被分配为 FIQ	0

9. 向量控制器（VICVectCntl0-15-0xFFFF F200-23C）

每个寄存器控制 16 个向量IRQ slot中的一个，这 16 个寄存器仅适用于向量IRQ中断，必须和向量地址寄存器 0-15 配对使用，用于确定每个向量IRQ中断通道的优先级。向量IRQ通道 0 优先级最高，向量IRQ通道 15 优先级最低。在VICIntCntl寄存器中，禁止一个向量IRQ slot不会禁止中断本身，中断只是变为非向量的形式。向量控制寄存器的描述如表 4.30所列。

表 4.30 向量控制寄存器 0～15

位	描述	复位值
4:0	写入分配给此向量 IRQ slot 的中断请求或软件中断的编号。不要将相同的中断编号分配给多于一个使能的向量 IRQ slot，但是如果这样做了，当中断请求或软件中断使能且被分配为 IRQ 时，会使用最低编号的 slot	0
5	写 1：向量 IRQ 使能，当分配的中断请求或软件中断使能，且被分配为 IRQ 时，可产生一个唯一的 ISR 地址	0
31:6	保留，用户不应该向这些位进行写操作	NA

10. 向量地址寄存器（VICVectAddr0-15-0xFFFF F100-13C）

这些寄存器保存 16 个向量IRQ slot中断服务程序的地址。向量地址寄存器的描述如表 4.31所列。

表 4.31 向量地址寄存器 0～15 的描述

位	描述	复位值
31:0	分配为该向量 IRQ slot 的中断服务程序地址（由程序设置），当中断请求或软件中断使能并被分配为 IRQ 时，读取向量地址寄存器（VICVectAddr）时会得到最高优先级 slot 寄存器的值	0

11. 默认向量地址寄存器（VICDefVectAddr-0xFFFF F034）

该寄存器保存非向量IRQ中断服务程序（ISR）的地址，该寄存器的描述如表 4.32所列。当非向量IRQ中断发生时，该寄存器的值将会被复制到VICVectAddr中，程序根据VICVectAddr中的地址进行跳转。

表 4.32 默认向量地址寄存器描述

位	描述	复位值
31:0	非向量 IRQ 中断服务程序地址，当一个 IRQ 服务程序读取向量地址寄存器（VICVectAddr）但没有 IRQ slot 响应时，则返回此寄存器的值	0

12. 向量地址寄存器（VICVectAddr-0xFFFF F030）

当发生IRQ中断时，IRQ服务程序可读取该寄存器并跳转到读出的地址处。向量寄存器的描述如表 4.33所列。

表 4.33 向量地址寄存器描述

位	描述	复位值
31:0	IRQ 中断服务程序地址（来自 VICDefVectAddr 或者 VICVectAddr0-15），由于该寄存器的设置并不能在以后读出，因此，在 ISR 快结束时，应该对此寄存器执行写操作（通常写入 0），以便更新硬件优先级	0

13. 基本使用方法

VIC 的寄存器比较多，但在一般应用中，没有必要用到全部寄存器，掌握一些常用、基本的寄存器即可。VIC 的基本使用方法：

- 确定该中断分配为 FIQ 还是 IRQ 中断；
- 若分配为 FIQ，则进行相关初始化；若分配为 IRQ，继续分配是向量 IRQ 还是非向量 IRQ 中断，然后分别进行相关设置；
- 清除相应中断标志，并使能相应中断；
- 编写中断服务程序。

以上步骤以流程图方式归纳如图 4.8所示。

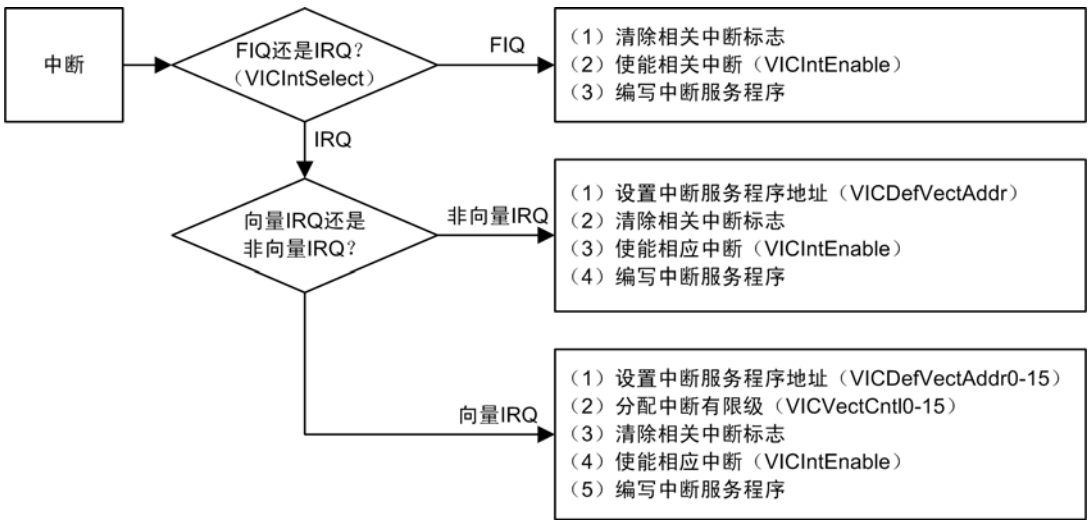


图 4.8 VIC 基本使用流程图

4.3.4 中断源

VIC可以管理 32 路中断请求，但在LPC2103 中，并没有使用完所有中断请求通道，没有使用的中断通道保留供后续芯片使用。LPC2103 的大部分外设都能产生中断，有的还能产生多个中断，LPC2103 的可用中断源如表 4.34。

表 4.34 LPC2103 中断源

模块	标志	VIC 通道号
WDT	看门狗中断（WDINT）	0
-	保留给软件中断	1
ARM 内核	EmbeddedICE, DbgCommRx	2
ARM 内核	EmbeddedICE, DbgCommTx	3
定时器 0	匹配 0-2（MR0, MR1, MR2），捕获 0-（CR0, CR1, CR2）	4
定时器 1	匹配 0-3（MR0, MR1, MR2, MR3），捕获 0-3（CR0, CR1, CR2, CR3）	5
UART0	Rx 线状态（RLS），发送保持寄存器空（THRE）；Rx 数据可用（RDA），字符超时指示（CTI）	6
UART1	Rx 线状态（RLS），发送保持寄存器空（THRE）；Rx 数据可用（RDA），字符超时指示（CTI），Modem 状态中断（MSI）	7
-	保留	8
I^2C0	SI（状态改变）	9
SPI0	SPI0 中断标志（SPI0F），模式错误（MODF）	10
SPI1（SSP）	Tx FIFO 至少一半为空（TXRIS），Rx FIFO 至少一半为满（RXRIS），接收超时条件（RTRIS），接收溢出（RORRIS）	11
PLL	PLL 锁定（PLOCK）	12
RTC	计数器增加（RTCCIF），报警（RTCALF）	13
系统控制	外部中断 0（EINT0）	14
	外部中断 1（EINT1）	15
	外部中断 2（EINT2）	16
	保留	17

续上表

模块	标志	VIC 通道号
A/D0	A/D 转换器 0，转换结束	18
I^2C1	SI（状态改变）	19
-	保留	20-25
TIMER2	匹配 0-2（MR0，MR1，MR2），捕获 0-2（CR0，CR1，CR2）	26
TIMER3	匹配 0-（MR0，MR1，MR2，MR3）	27

4.3.5 中断处理

1. ARM对异常中断的响应过程

当发生异常中断时，ARM 处理器硬件会自动响应如下过程：（假设中断未被屏蔽）

- 将寄存器 LR 设置为返回地址；
- 保存当前程序状态寄存器。方法：将前一模式的 CPSR 内容保存到异常中断对应的 SPSR 寄存器中；
- 关中断——在 CPSR 中禁止 IRQ 中断，或同时禁止 IRQ 与 FIQ 中断，防止 CPU 响应同类型的中断；
- 将 PC 值设置成该异常中断的入口地址。发生 IRQ 中断时，PC = 0x0000 0018；发生 FIQ 中断时，PC = 0x0000 001C。

再次强调一下，上述过程是 ARM 处理器硬件自动完成的，无需用户干预。

2. 从异常中断处理程序中返回

ARM 处理器从异常中断中返回时，要执行两个操作：

- 恢复程序状态寄存器。方法：将寄存器 SPSR 的值复制到 CPSR 中即可；
- 恢复用户程序的运行。

ARM 处理器从中断返回时执行的两个操作是由用户软件实现的。

4.3.6 FIQ中断

1. FIQ中断描述

FIQ优先级最高，中断响应最迅速，常用于处理系统中最重要最紧急的中断事件。一旦发生FIQ中断，ARM处理器进入FIQ模式，PC指向FIQ异常入口 0x1C，开始处理FIQ中断，如图 4.9所示。ARM处理器为FIQ模式多设计了R8~R12 这 5 个私有寄存器，加速FIQ的处理。设计FIQ模式的目的是为了获得最快的中断处理，减小中断延迟。

图 4.9 FIQ 示意图

建议只分配一个中断为 FIQ，保证 FIQ 能以最快的速度处理 FIQ 中断。如果将多个中断分配为 FIQ，只能在软件中通过 VICFIQStatus 来查询到底是哪个中断已经发生，这势必增加中断延迟，这有悖于 FIQ 的设计初衷。

正常情况下，用户程序工作在系统模式，发生FIQ中断后，处理器硬件会执行一些固定的操作，然后执行用户软件，最后返回到中断处。整个处理过程如图 4.10所示。

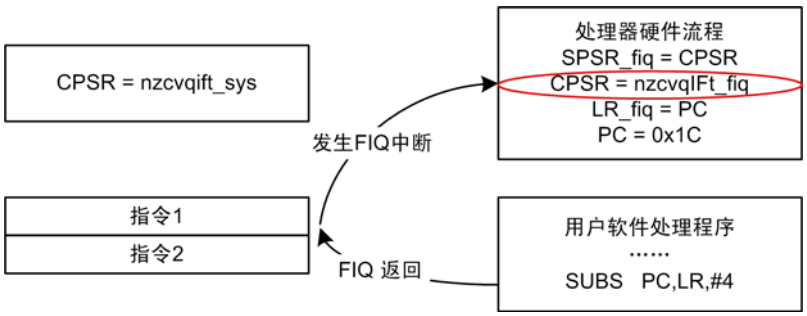


图 4.10 FIQ 操作示例

FIQ 中断返回仅使用一条指令“SUBS PC, LR, #4”，由于 SUB 指令结尾含有“S”，并且目的寄存器是 PC，因此 CPSR 将自动从 SPSR 寄存器中恢复。这样就同时达到了恢复程序状态寄存器的目的。当然，实现这种目的的方法也不是唯一的。

需要注意一点：产生 FIQ 中断时，处理器硬件会同时禁止 FIQ 和 IRQ 中断。

2. 应用示例

将一个中断分配为 FIQ 后，只需在主程序中使能该中断，然后编写 FIQ 服务处理程序即可。下面给出一个简单示例，将外部中断 0 分配为 FIQ，并编写 FIQ 服务程序。

- 在该示例的主程序中，首先选择P0.0 为外部中断功能，其后是FIQ初始化，具体的设置如程序清单 4.4所示。

程序清单 4.4 FIQ 中断主程序

```
/*********************************************************************************************
** 函数名称：main
** 函数功能：FIQ 中断主程序
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
int main (void)
{
    PINSEL1     = (PINSEL1 & 0xFFFFFFFC) | 0x01;     /* 设置 P0.16 为外部中断 0 管脚     */
    EXTMODE     = 0x00;                              /* 跟下条语句一起决定低电平触发     */
    EXTPOLAR    = 0x00;
    PINSEL1     = PINSEL1 & 0xFFFFFFF3;              /* 设置 P0.17 为 GPIO 功能          */
    IO0DIR      = IO0DIR | LED;                      /* 设置 P0.17 为输出                */
    IO0SET      = IO0SET | LED;                      /* 设置输出为高电平                 */
    VICIntSelect    = 1 << 14;                       /* 选择 EINT0 为 FIQ 中断           */
    EXTINT      = 0x01;                              /* 清除 EINT0 中断标志              */
    VICIntEnable    = 1 << 14;                       /* 使能 EINT0 中断                  */
    FIQEnable();                                     /* FIQ 中断使能                     */
    while(1);
    return 0;
```

```
}
```

- FIQ中断的服务程序如程序清单 4.5所示，该函数已经在模板的target.c文件中定义，所以只要在该函数下添加相应的代码就可以了。示例的结果现象是按键按下一次，发光二极管将会点亮或熄灭。

程序清单 4.5 FIQ 中断服务程序

```
uint32   FIQ_Count = 0;                          /* 定义变量，对 FIQ 中断计数      */
void FIQ_Exception(void)
{
    FIQ_Count++;                                 /* 进入中断加 1                   */
    if (( FIQ_Count % 3) == 0) {
    IO0SET   = IO0SET | (1 << 17);               /* 熄灭发光二极管                 */
    }
    else {
    IO0CLR   = IO0CLR | (1 << 17);               /* 点亮发光二极管                 */
    }
   while((IO0PIN & (1 << 16)) == 0) ;            /* 等待按键松开                   */
   EXTINT = 0x01;                                /* 清中断标志                     */
}
```

4.3.7 向量IRQ中断

1. 向量IRQ中断描述

向量IRQ具有中等优先级，处理中断比较迅速。一旦发生向量IRQ中断，ARM处理器进入IRQ模式，PC指向IRQ异常入口 0x18，同时向量IRQ服务程序的地址从相应通道的向量地址寄存器(VICVectAddr0-15)中复制到VIC的向量地址寄存器(VICVectAddr)，PC根据VICVectAddr内的地址进行跳转，执行相应的服务程序，如图 4.11所示。这整个过程都是由VIC硬件自动完成的，无需用户的软件干预。

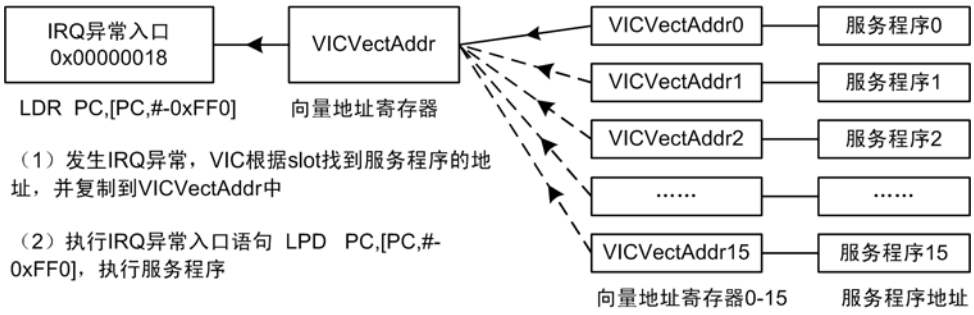


图 4.11 向量 IRQ 示意图

在一个具体应用中，向量 IRQ 中断使用往往是最多的，一个系统往往都会有多个向量 IRQ 中断。将一个中断分配为向量 IRQ 中断后，需要在程序中设置该中断的优先级、服务程序地址，接着清除相关中断标识再使能相应中断，最后编写 IRQ 中断服务程序即可。

IRQ的中断处理过程与FIQ比较类似，如图 4.12所示。在系统模式下，IRQ与FIQ中断允许，当发生IRQ中断后，处理器硬件会禁止IRQ中断，但是，FIQ中断仍然是允许的，这一点与FIQ的处理是不同的。

发生 IRQ 中断时，处理器硬件会获取向量 IRQ 的服务地址，即，将对应的向量地址寄存器内容复制到 VICVectAddr 寄存器中。

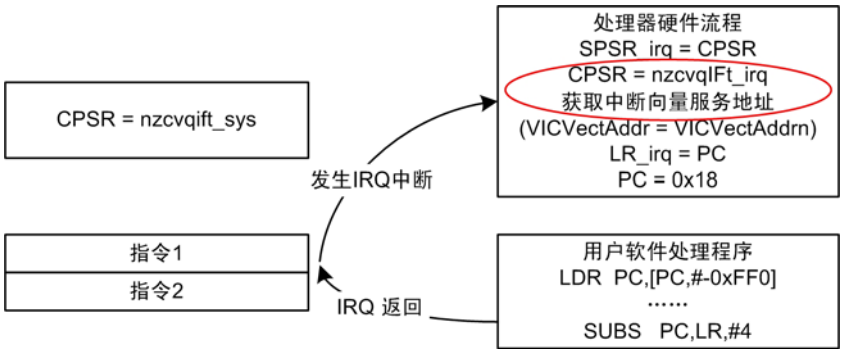


图 4.12 IRQ 操作示例

2. 应用示例

还是对外部中断 0 进行操作，将外部中断 0 分配为 IRQ。

- 这个示例程序的主程序如程序清单 4.6所示，包含外部中断 0 的功能选择和向量IRQ的初始化。

程序清单 4.6 IRQ 中断主程序

```
/*********************************************************************************************
** 函数名称：main
** 函数功能：FIQ 中断主程序
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
int main (void)
{
    PINSEL1         = (PINSEL1 & 0xFFFFFFFC) | 0x01;        /* 设置 P0.16 为外部中断 0 管脚      */
    EXTMODE         = 0x00;                                 /* 跟下条语句一起决定低电平触发      */
    EXTPOLAR        = 0x00;

    PINSEL1    = PINSEL1 & 0xFFFFFFF3;                      /* 设置 P0.17 为 GPIO 功能           */
    IO0DIR     = IO0DIR | LED;                              /* 设置 P0.17 为输出                 */
    IO0SET     = IO0SET | LED;                              /* 设置输出为高电平                  */

    VICIntSelect    = 0 << 14;                              /* 选择 EINT0 为 FIQ 中断            */
    VICVectCntl0    = 0x20 | 14;                            /* 将外部中断 0 分配给向量中断 0     */
    VICVectAddr0    = (uint32)IRQ_Server;                   /* 设置中断服务程序地址              */
    VICIntEnable    = 1 << 14;                              /* 使能 EINT0 中断                   */
    EXTINT          = 0x01;                                 /* 清除 EINT0 中断标志               */
    IRQEnable();                                            /* IRQ 中断使能                      */
    while(1);
    return 0;
```

```
}
```

- IRQ向量中断的服务程序如程序清单 4.7所示。注意与FIQ中断的不同，多了最后一条中断向量结束语句。

程序清单 4.7 IRQ 中断服务程序

```
/*********************************************************************************************
** 函数名称：IRQ_Server
** 函数功能：IRQ 中断服务程序，控制发光二极管的亮灭
** 输入参数：无
** 输出参数：无*
*********************************************************************************************/
uint32  IRQ_Count = 0;                                  /* 定义变量，对 FIQ 中断计数        */
void  __irq IRQ_Server(void)
{
    IRQ_Count++;                                        /* 进入中断加 1                     */
    if (( IRQ_Count % 3) == 0)  {
        IO0SET  = IO0SET | (1 << 17);                   /* 熄灭发光二极管                   */
    }
    else {
        IO0CLR  = IO0CLR | (1 << 17);                   /* 点亮发光二极管                   */
    }
    while((IO0PIN & (1 << 16)) == 0) ;                  /* 等待按键松开                     */
    EXTINT = 0x01;                                      /* 清中断标志                       */
    VICVectAddr = 0x00;                                 /* 向量中断结束                     */
}
```

4.3.8 非向量中断

非向量中断优先级最低，在将多个中断分配为非向量IRQ中断的情况下，中断延迟时间也较长，一般仅用于处理一般中断事件。在向量IRQ够用的情况下，最好不要使用非向量IRQ中断。一旦发生非向量IRQ中断，ARM处理器进入IRQ模式，同时中断服务程序的地址从默认向量地址寄存器VICDefVectAddr复制到向量地址寄存器VICVectAddr中，PC根据VICVectAddr中的地址进行跳转，执行相应的中断服务程序。如果将多个中断分配为非向量中断，需要在程序中查询VICIRQStatus寄存器，确定到底哪个中断已经发生，执行哪段服务程序，如图 4.13所示。

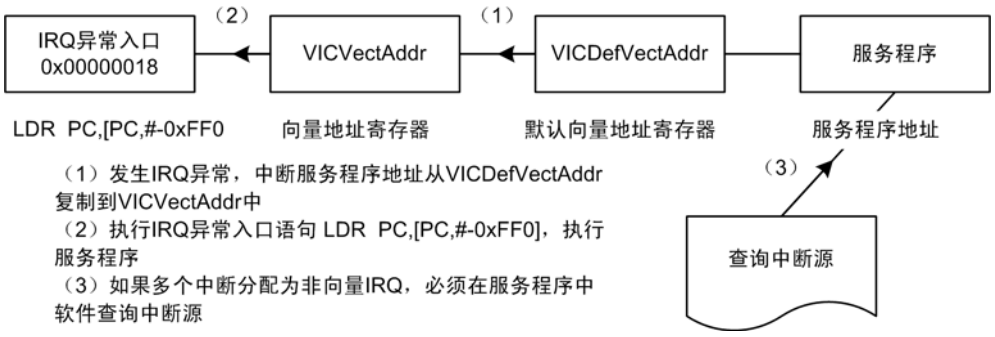


图 4.13 非向量中断示意图

将一个中断分配为非向量 IRQ 中断后，只需设置服务程序的地址，接着清除相关中断标志后使能该中断，最后编写中断服务程序即可。

1. 应用示例

把外部中断 0 设置为非向量中断，由此可对比较非向量 IRQ、向量 IRQ 和 FIQ 中断程序的不同。

- 首先，区别非向量中断的主程序与其它中断的不同，非向量中断的主程序如程序清单 4.8所示。跟向量中断相比，它没有通道号的分配。

程序清单 4.8 非向量中断主程序

```
/*********************************************************************************************
** 函数名称：main
** 函数功能：非向量中断主程序
** 输入参数：无
** 输出参数：无
*********************************************************************************************/
int main (void)
{
    PINSEL1         = (PINSEL1 & 0xFFFFFFFC) | 0x01;        /* 设置 P0.16 为外部中断 0 管脚      */
    EXTMODE         = 0x00;                                 /* 跟下条语句一起决定低电平触发      */
    EXTPOLAR        = 0x00;
    PINSEL1         = PINSEL1 & 0xFFFFFFF3;                 /* 设置 P0.17 为 GPIO 功能           */
    IO0DIR          = IO0DIR | LED;                         /* 设置 P0.17 为输出                 */
    IO0SET          = IO0SET | LED;                         /* 设置输出为高电平                  */
    VICIntSelect    = 0x00;                                 /* 选择 EINT0 为 IRQ 中断            */
    VICDefVectAddr = (uint32)DEF_Server;                    /* 给非向量中断分配地址              */
    VICIntEnable = 1 << 14;                                 /* 使能 EINT0 中断                   */
    EXTINT = 0x01;                                          /* 清除 EINT0 中断标志               */
    IRQEnable();                                            /* IRQ 中断使能                      */
    while(1);
    return 0;
}
```

- 再看非向量中断的服务程序，如程序清单 4.9所示，跟向量中断的服务程序基本一样。

程序清单 4.9　非向量中断服务程序

```
/*********************************************************************************************
** 函数名称：DEF_Server
** 函数功能：非向量中断服务程序
** 输入参数：无
** 输出参数：无
*********************************************************************************************/
uint32   DEF_Count = 0;                                  /* 定义变量，对非向量中断计数      */
void   __irq DEF_Server(void)
{
    DEF_Count++;                                         /* 进入中断加 1                    */
    if (( DEF_Count % 3) == 0) {
        IO0SET   = IO0SET | (1 << 17);                   /* 熄灭发光二极管                  */
    }
    else {
        IO0CLR   = IO0CLR | (1 << 17);                   /* 点亮发光二极管                  */
    }
    while((IO0PIN & (1 << 16)) == 0);                    /* 等待按键松开                    */
    EXTINT = 0x01;                                       /* 清中断标志                      */
    VICVectAddr = 0x00;                                  /* 中断结束                        */
}
```

4.4　外部中断

4.4.1 概述

早期的计算机系统是没有中断概念的，CPU 对于外设状态的获取只能使用查询方式，但我们知道外设的速度相对于 CPU 而言是很慢的，所以查询方式会浪费 CPU 大量的时间。后来中断系统出现了，它可以让 CPU 一边干着其它的事情，一边监听等待的事件是否发生，从而大大地提高了工作效率。

中断的触发方式有 2 种类型：边沿触发和电平触发。其中边沿触发分为上升沿触发和下降沿触发，电平触发分为高电平触发和低电平触发。图 4.14所示为下降沿触发的中断，在中断信号由高电平变为低电平的那一时刻向CPU发出中断请求。

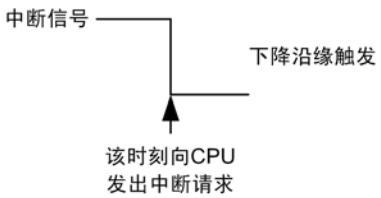


图 4.14　边沿触发中断示意图

图 4.15所示为低电平触发中断，在中断信号变由高电平变为低电平的之后，中断控制器向CPU发出中断请求，若中断信号一直为低电平，中断控制器将不停的向CPU发出该中断请求。

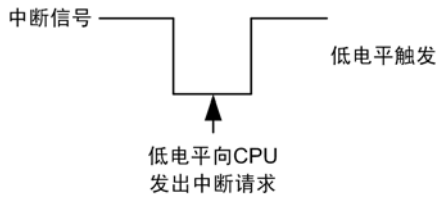


图 4.15 电平触发中断示意图

LPC2103 可以产生丰富的中断信息，甚至可以说几乎所有的外设部件都可以产生中断。其中“外部中断”就是很重要的中断源，LPC2103 有 3 个外部中断输入，也就是说外部中断有 3 个中断源，如图 4.16所示。

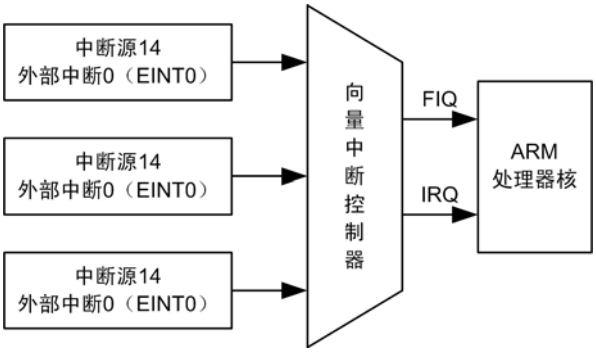


图 4.16 外部中断源

此外 LPC2103 的 10 个捕获输入也可作为外部中断输入，但是跟外部中断不同的是不能把掉电模式下的处理器唤醒。

4.4.2 寄存器描述

图 4.17是外部中断的简化结构图，从图中我们可以了解到各个部件的控制寄存器名称。

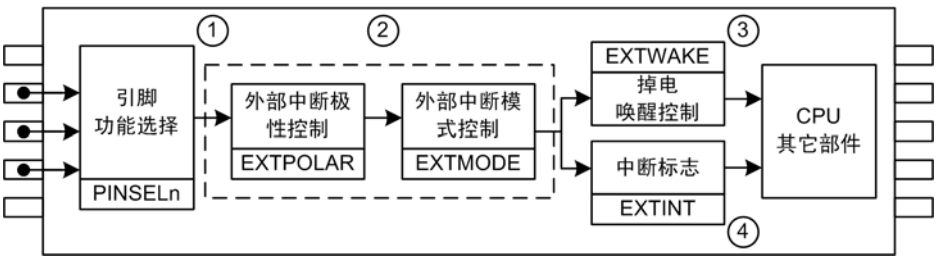


图 4.17 外部中断系统结构

LPC2103 允许一个或多个芯片引脚为外部输入信号端，信号首先经过 PINSELn 寄存器控制的引脚连接模块，然后判别输入信号的极性和方式是否符合预设要求，如果都通过了，将作为有效中断信号设置中断标志，还可以把 CPU 从掉电模式唤醒。

外部中断具有 4 个相关的寄存器，如表 4.35所示。其中EXTINT寄存器包含中断标志；EXTWAKEUP寄存器包含使能唤醒位，可使能独立的外部中断输入将处理器从掉电模式唤醒；EXTMODE和EXTPOLAR寄存器用来指定引脚使用电平或边沿触发方式。

表 4.35 外部中断寄存器

地址	名称	描述	访问
0xE01F C140	EXTINT	外部中断标志寄存器包含 ENIT0，EINT1 和 EINT2 的中断标志	R/W

续上表

地址	名称	描述	访问
0xE01F C144	EXTWAKE	外部中断唤醒寄存器包含 3 个用于控制外部中断是否将处理器从掉电模式唤醒的使能位	R/W
0xE01F C148	EXTMODE	外部中断模式寄存器控制每个引脚的边沿或电平触发中断	R/W
0xE01F C14C	EXTPOLAR	外部中断极性寄存器控制由每个引脚的哪种电平或边沿来触发中断	R/W

1. 外部中断标志寄存器（EXTINT-0xE01F C140）

当一个引脚选择使用外部中断功能时（通过设置 PINSELn 寄存器实现），若引脚上出现了对应于 EXTPOLAR 和 EXTMODE 寄存器设置的电平或边沿信号，EXTINT 寄存器中的中断标志将被置位，然后向 VIC 提出中断请求，如果这个外部中断已使能，则产生中断。

在标志位置“1”后，通过向EXTINT寄存器的EINT0～EINT2 位写入“1”来将其清零，在电平触发方式下，只有在引脚处于无效状态时才有可能将标志位清零，比如设置为低电平中断，则只有在中断引脚恢复为高电平后才能清除中断标志。EXTINT寄存器描述如表 4.36 所示。

表 4.36 外部中断标志寄存器

EXTINT	功能	描述	复位值
0	EINT0	如果引脚的外部中断输入功能被选用，并且在该引脚上出现了符合预定要求的中断信号时，该位置位。该位通过写入 1 清除，但电平触发方式下引脚处于有效状态的情况除外	0
1	EINT1	同上	0
2	EINT2	同上	0
7:3	保留	保留，用户软件不要向其写入 1，从保留位读出的值未被定义	NA

操作示例

```
while((EXTINT & 0x01) == 0);                /* 等待 EINT0 出现中断          */
EXTINT = 0x01;                              /* 清除 EINT0 中断标志          */
```

2. 外部中断唤醒寄存器（EXTWAKE-0xE01F C144）

EXTWAKE寄存器中的使能位允许相应的外部中断将处理器从掉电模式唤醒，相关的EINTn功能必须连接到引脚才能实现掉电唤醒功能。实现掉电唤醒不需要（在向量中断控制器中）使能相应的中断，这样做的好处是允许外部中断输入将处理器从掉电模式唤醒，但不产生中断（只是简单地恢复操作）。EXTWAKE寄存器描述如表 4.37所示。

表 4.37 外部中断唤醒寄存器

EXTWAKE	功能	描述	复位值
0	EXTWAKE0	该位为 1 时，使能 $\overline{\text{EINT0}}$ 将处理器从掉电模式唤醒	0
1	EXTWAKE1	该位为 1 时，使能 $\overline{\text{EINT1}}$ 将处理器从掉电模式唤醒	0
2	EXTWAKE2	该位为 1 时，使能 $\overline{\text{EINT2}}$ 将处理器从掉电模式唤醒	0
14:3	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义	NA
15	RTCWAKE	该位为 1 时，使能 RTC 中断将处理器从掉电模式唤醒	0

操作示例

```
EXTWAKE = 0x01;                             /* 使能 EINT0 将处理器从掉电模式唤醒  */
```

3. 外部中断模式寄存器（EXTMODE–0xE01F C148）

EXTMODE寄存器中的位用来选择每个EINT脚是电平还是边沿触发。只有选择用作EINT功能的引脚，并已通过VICIntEnable使能相应中断，才能产生外部中断。EXTMODE寄存器描述如表 4.38所示。

表 4.38 外部中断模式寄存器

EXTMODE	功能	描述	复位值
0	EXTMODE0	设置为 0 时，该外部中断为电平触发 设置为 1 时，该外部中断为边沿触发	0
1	EXTMODE1	同上	0
2	EXTMODE2	同上	0
7:3	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义	NA

操作示例

```
EXTMODE = (1 << 1) | (1 << 2);          /* EINT0 设置为电平触发          */
                                        /* EINT1 和 EINT2 设置为边沿触发  */
```

4. 外部中断极性寄存器（EXTPOLAR–0xE01F C14C）

在电平触发方式中，EXTPOLAR寄存器用来选择相应引脚是高电平或低电平有效。在边沿触发方式中，EXTPOLAR寄存器用来选择引脚上升沿或下降沿有效。只有选择用作EINT功能的引脚，并已通过VICIntEnable使能相应中断，才能产生外部中断。EXTPOLAR寄存器描述如表 4.39所示。

表 4.39 外部中断极性寄存器

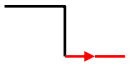
EXTPOLAR	功能	描述	复位值
0	EXTPOLAR0	该位为 0 时，该外部中断由低电平或下降沿触发（由 EXTMODE 的对应位决定） 该位为 1 时，该外部中断由高电平或上升沿有效（由 EXTMODE 的对应位决定）	0
1	EXTPOLAR1	同上	0
2	EXTPOLAR2	同上	0
7:3	保留	保留，用户软件不要向其写入 1。	NA

操作示例

```
EXTPOLAR = (1 << 1) | (1 << 2);         /* EINT0 设置为低电平或下降沿触发
                                           EINT1 和 EINT2 设置为高电平或上升沿触发  */
```

外部中断方式和外部中断极性两个寄存器组合设置后可以准确描述中断信号的波形，所有可能的波形与设置组合如表 4.40所示。

表 4.40 外部中断触发设置

设置说明	相应位设置值		信号波形
	极性控制寄存器（EXPOLAR）	方式控制寄存（EXTMODE）	
低电平触发	0（低）	0（电平）	

续上表

设置说明	相应位设置值		信号波形
	极性控制寄存器（EXPOLAR）	方式控制寄存（EXTMODE）	
高电平触发	1（高）	0（电平）	
下降沿触发	0（下降）	1（边沿）	
上升沿触发	1（上升）	1（边沿）	

4.4.3 外部中断引脚设置

在实际使用外部中断功能时还应注意以下几点：

- 如果要产生外部中断，除了引脚连接模块的设置，还需设置 VIC 模块，否则外部中断只能反映在 EXTINT 寄存器中；
- 若使器件进入掉电模式并通过外部中断唤醒，软件应正确设置引脚外部中断功能。

4.4.4 中断设置

LPC2103 含有 3 个外部中断源，每个中断源可以产生 2 种类型的中断：电平中断和边沿中断。外部中断与向量中断控制器（VIC）的关系如图 4.18所示。

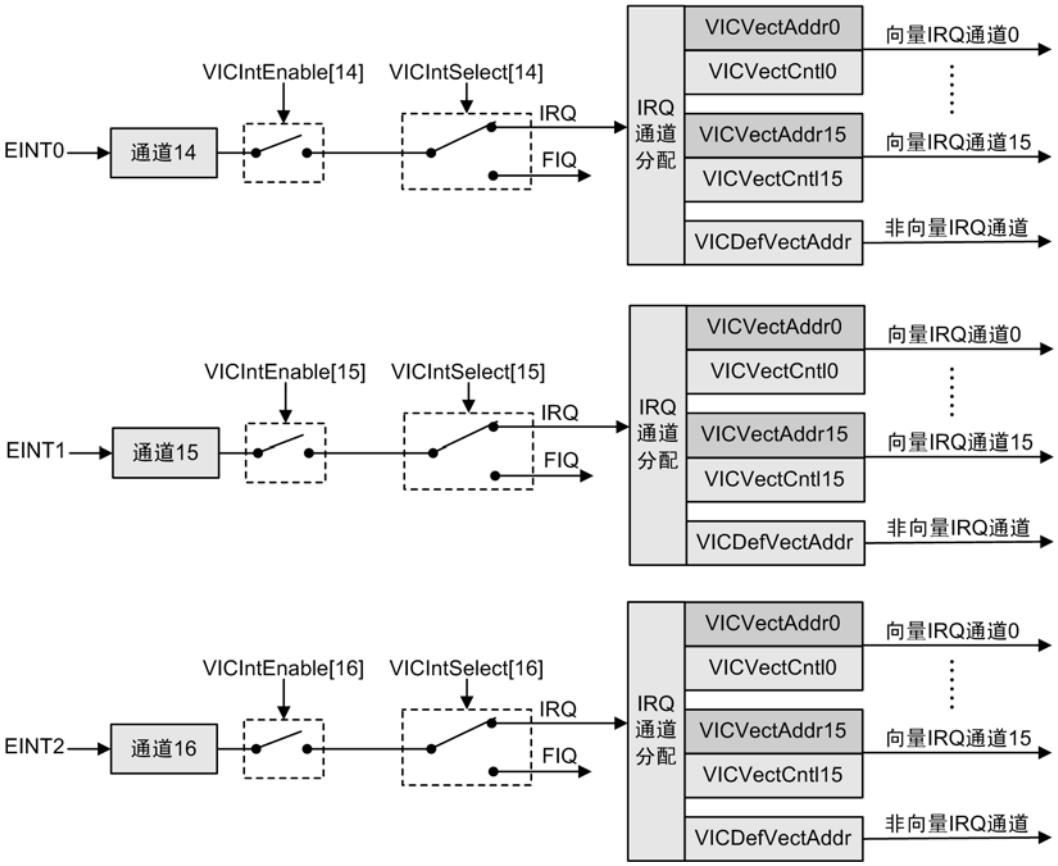


图 4.18 外部中断 0、1、2 与 VIC 的关系

外部中断 0~2 分别处于 VIC 的通道 14~16，中断使能寄存器 VICIntEnable 用来控制 VIC

通道的中断使能。

- 当 VICIntEnable[14] = 1 时，通道 14 中断使能，即：外部中断 0 中断使能；
- 当 VICIntEnable[15] = 1 时，通道 15 中断使能，即：外部中断 1 中断使能；
- 当 VICIntEnable[16] = 1 时，通道 16 中断使能，即：外部中断 2 中断使能。

中断选择寄存器 VICIntSelect 用来分配 VIC 通道的中断。当某一位为“1”时，对应的通道中断分配为 FIQ；当某一位为“0”时，对应的通道中断分配为 IRQ。VICIntSelect[14] ~ VICIntSelect[16]分别用来控制外部中断 0～2。

- 当 VICIntSelect[14] = 1 时，EINT0 中断分配为 FIQ 中断；
- 当 VICIntSelect[14] = 0 时，EINT0 中断分配为 IRQ 中断；
- 当 VICIntSelect[15] = 1 时，EINT1 中断分配为 FIQ 中断；
- 当 VICIntSelect[15] = 0 时，EINT1 中断分配为 IRQ 中断；
- 当 VICIntSelect[16] = 1 时，EINT2 中断分配为 FIQ 中断；
- 当 VICIntSelect[16] = 0 时，EINT2 中断分配为 IRQ 中断。

当分配为 IRQ 时，还需要设置对应的通道控制寄存器和地址寄存器。有关寄存器 VICVectCntln 和 VICVectAddrn 的说明，请参考“向量中断控制器（VIC）”一节。

1. 电平中断

LPC2103 外部中断可以设置为电平触发中断：高电平触发、低电平触发，我们以EINT0 为例来介绍如何设置电平触发中断。如图 4.19所示，模式寄存器（EXTMODE）用来在“边沿中断”和“电平中断”之间进行选择，极性寄存器（EXTPOLAR）用来选择中断的极性：“低电平中断”、“高电平中断”、“上升沿中断”、“下降沿中断”。

对于 EINT0 来说，EXTMODE[0] = 0 时，EINT0 为电平中断：

- EXTPOLAR[0] = 0 时，EINT0 为低电平中断；
- EXTPOLAR[0] = 1 时，EINT0 为高电平中断。

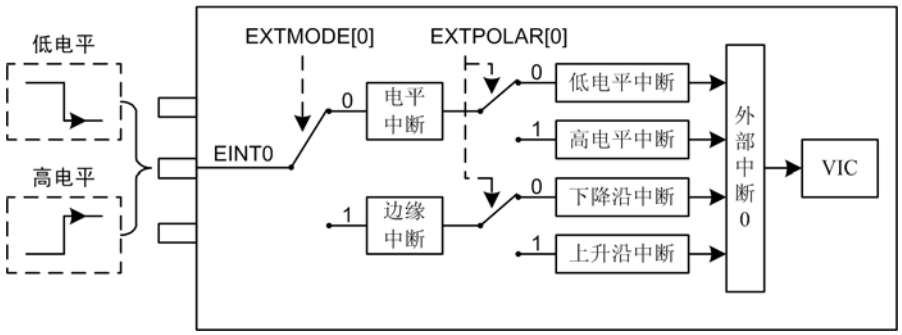


图 4.19 EINT0 电平中断示意图

2. 边沿中断

LPC2103 外部中断除了可以电平触发外，还可以设置为边沿触发：下降沿触发、上升沿触发，我们仍然以EINT0 为例来介绍如何设置边沿触发中断，如图 4.20所示。当 EXTMODE[0] = 1 时，EINT0 为边沿中断：

- EXTPOLAR[0] = 0 时，EINT0 为下降沿中断；
- EXTPOLAR[0] = 1 时，EINT0 为上升沿中断。

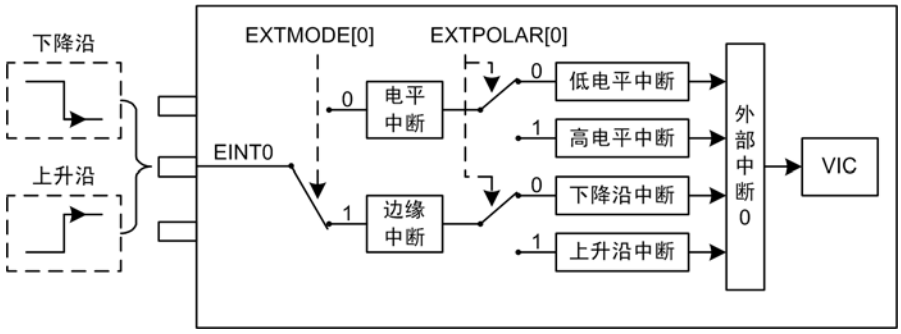


图 4.20　EINT0 边沿中断示意图

3.　中断标志

LPC2103 具有 3 路外部中断，当外部中断触发时，会置位对应的中断标志位，如图 4.21 所示，之后向中断标志位写“1”将清零该标志位。

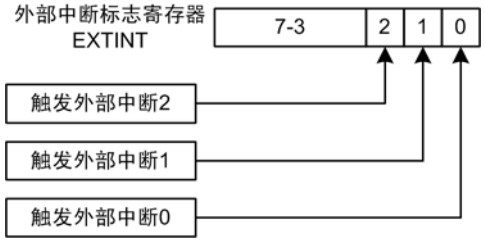


图 4.21　外部中断标志示意图

4.4.5 应用示例

本示例应用P0.16 低电平触发外部中断 0，短接JP3 的P0.16 端口，当按键KEY1 按下后，P0.16 输入低电平触发外部中断，取反LED，示例代码如程序清单 4.10所示。

注意，当相应引脚设置为外部中断功能时，引脚为输入模式，由于没有内部上拉电阻，用户需要外接一个上拉电阻，确保引脚不会悬空。

程序清单 4.10　P0.16 低电平触发外部中断 0

```
#include "config.h"
#define     LED       1 << 17
/*********************************************************************************************
**  函数名称：Eint0IRQ
**  功能描述：外部中断 0 服务程序
**  入口参数：无
**  出口参数：无
*********************************************************************************************/
void __irq Eint0IRQ(void)
{
      if ((IO0PIN & (1 << 17)) == 0) {
      IO0SET = 1 << 17;                                    /*  熄灭发光二极管                  */
      }
      else {
            IO0CLR = 1 << 17;                              /*  点亮发光二极管                  */
```

```
    }
    while((IO0PIN & (1 << 16)) == 0);                    /* 等待按键松开                */
    EXTINT = 0x01;                                       /* 清中断标志                  */
    VICVectAddr = 0x00;                                  /* 通知 VIC 中断处理结束       */
}
/*********************************************************************************************
** 函数名称：main
** 功能描述：P0.16 低电平触发外部中断主函数
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
int main (void)
{
    PINSEL1 = PINSEL1 & (~0x03);
    PINSEL1 = PINSEL1 | 0x01;                            /* 设置 P0.16 为外部中断 0 管脚 */

    PINSEL1 = PINSEL1 & (~(0x03 << 2));                  /* 设置 P0.17 为 GPIO 功能     */
    IO0DIR  = LED;                                       /* 设置 P0.17 为输出           */
    IO0SET  = LED;                                       /* 设置输出为高电平            */

    IRQEnable();                                         /* IRQ 中断使能                */
    EXTMODE  = 0x00;                                     /* 设置外部中断为低电平触发    */

    VICIntSelect = 0 << 14;                              /* 选择 EINT0 为 FIQ 中断      */
    VICVectCntl0 = 0x20 | 14;                            /* 将外部中断 0 分配给向量中断 0 */
    VICVectAddr0 = (uint32)Eint0IRQ;                     /* 设置中断服务程序地址        */
    VICIntEnable = 1 << 14;                              /* 使能 EINT0 中断             */

    EXTINT = 0x01;                                       /* 清除 EINT0 中断标志         */
    while(1);
    return 0;
}
```

4.5 定时器 0 和定时器 1

4.5.1 概述

LPC2103 含有两个 32 位定时器：定时器 0 和定时器 1，这两个定时器除了外设基地址不同外，其它都相同。定时器/计数器对外设时钟（P_{CLK}）或外部提供的时钟周期进行计数，可选择产生中断或根据 4 个匹配寄存器的设定，在到达指定的定时值时执行其它动作。它还包括 4 个捕获输入，用于在输入信号发生跳变时捕获定时器值，并可选择产生中断。

由于 LPC2101/02/03 器件的管脚数目有限，定时器 0 只有 3 个捕获输入和 3 个匹配输出被连接到器件的管脚。2 个匹配寄存器可以在 MATn.2～0 管脚上提供单边沿控制的 PWM 输出。由于 MAT0.3 寄存器在定时器 0 上没有管脚输出，因此建议使用 MRn.3 寄存器来控制 PWM 周期长度。需要用另一个匹配寄存器来控制 PWM 边沿位置。剩余的两个寄存器可用来创建其它的 PWMn 输出，该输出具有由 MRn.3 决定的 PWM 周期率。

4.5.2 特性

- 两个 32 位定时器/计数器各含有一个可编程 32 位预分频器；
- 计数器或定时器操作；
- 定时器 0 有 3 路、定时器 1 有 4 路捕获通道。当输入信号跳变时可取得定时器的瞬时值。也可选择使捕获事件产生中断；
- 每个定时器共有 4 个 32 位匹配寄存器，匹配时的动作有如下 3 种：
  - 匹配时定时器继续工作，可选择产生中断；
  - 匹配时停止定时器，可选择产生中断；
  - 匹配时复位定时器，可选择产生中断。
- 定时器 0 有 3 个、定时器 1 有 4 个对应于匹配寄存器的外部输出，匹配时的输出有如下 4 种：
  - 匹配时设置为低电平；
  - 匹配时设置为高电平；
  - 匹配时翻转；
  - 匹配时无动作。
- 对于每个定时器，多达 4 个匹配寄存器可配置为 PWM，允许使用多达 3 个匹配输出作为单边沿控制的 PWM 输出。

4.5.3 引脚描述

定时器/计数器相关管脚的简要描述如表 4.41所列。

表 4.41 定时器/计数器管脚描述

管脚名称	管脚方向	管脚描述
CAP0.2～0 CAP1.3～0	输入	**捕获信号**：捕获管脚的跳变可配置为将定时器值装入一个捕获寄存器，并可选择产生一个中断。 下面是所有捕获信号及其选择的管脚的列表： CAP0.0：P0.2 CAP0.1：P0.4 CAP0.2：P0.6 CAP1.0：P0.10 CAP1.1：P0.11 CAP1.2：P0.17 CAP1.3：P0.18 计数器/定时器可选择一个捕获信号作为时钟源来代替 P_{CLK} 分频时钟。

续上表

管脚名称	管脚方向	管脚描述
MAT0.2～0 MAT1.3～0	输出	**外部匹配输出 0/1**：当匹配寄存器 0/1（MR3：0）等于定时器计数器（TC）时，该输出可翻转、变为低电平、变为高电平或不变。外部匹配寄存器（EMR）和 PWM 控制寄存器（PWMCON）控制该输出的功能。 下面是所有匹配信号及其选择的管脚的列表： MAT0.0：P0.3 MAT0.1：P0.5 MAT0.2：P0.16 MAT1.0：P0.12 MAT1.1：P0.13 MAT1.2：P0.19 MAT1.3：P0.20

4.5.4 寄存器描述

1. 寄存器汇总

定时器/计数器 0、1 所包含的寄存器如表 4.42所列。

表 4.42 定时器 0、定时器 1 寄存器映射

通用名称	描述	访问	复位值	定时器/计数器 0 地址&名称	定时器/计数器 1 地址&名称
IR	中断寄存器。可以写 IR 来清除中断。可读取 IR 来识别哪个中断源（8 个可能中断源中）被挂起	R/W	0	0xE0004000 T0IR	0xE0008000 T1IR
TCR	定时器控制寄存器。TCR 用于控制定时器计数器功能，定时器计数器可通过 TCR 禁止或复位	R/W	0	0xE0004004 T0TCR	0xE0008004 T1TCR
TC	定时器计数器。32 位 TC 每经过 PR+1 个 P_{CLK} 周期加 1，TC 通过 TCR 进行控制	R/W	0	0xE0004008 T0TC	0xE0008008 T1TC
PR	预分频寄存器。预分频计数器（下面）等于该值时，下个时钟增加 TC 并清除 PC	R/W	0	0xE000400C T0PR	0xE000800C T1PR
PC	预分频计数器。32 位 PC 是加 1 到 PR 内的存储值的计数器。当到达 PR 中保存的值时，TC 加 1 且 PC 被清除。通过总线接口可观察和控制 PC	R/W	0	0xE0004010 T0PC	0xE0008010 T1PC
MCR	匹配控制寄存器。MCR 用于控制在匹配时是否产生中断或复位 TC	R/W	0	0xE0004014 T0MCR	0xE0008014 T1MCR
MR0	匹配寄存器 0。MR0 可通过 MCR 设定为在每次 MR0 匹配 TC 时复位 TC、停止 TC 和 PC，选择产生中断	R/W	0	0xE0004018 T0MR0	0xE0008018 T1MR0
MR1	匹配寄存器 1。见 MR0 描述	R/W	0	0xE000401C T0MR1	0xE000801C T1MR1
MR2	匹配寄存器 2。见 MR0 描述	R/W	0	0xE0004020 T0MR2	0xE0008020 T1MR2
MR3	匹配寄存器 3。见 MR0 描述	R/W	0	0xE0004024 T0MR3	0xE0008024 T1MR3
CCR	捕获控制寄存器。CCR 控制用于装载捕获寄存器的捕获输入边沿以及在发生捕获时是否产生中断	R/W	0	0xE0004028 T0CCR	0xE0008028 T1CCR
CR0	捕获寄存器 0。当在 CAPn.0（CAP0.0、CAP1.0）上产生捕获事件时，CR0 装载 TC 的值	RO	0	0xE000402C T0CR0	0xE000802C T1CR0

续上表

通用名称	描述	访问	复位值	定时器/计数器 0 地址&名称	定时器/计数器 1 地址&名称
CR1	捕获寄存器 1。见 CR0 描述	RO	0	0xE0004030 T0CR1	0xE0008030 T1CR1
CR2	捕获寄存器 2。见 CR0 描述	RO	0	0xE0004034 T0CR2	0xE0008034 T1CR2
CR3	捕获寄存器 3。见 CR0 描述 注：CAP0.3 在定时器 0 上不可用	RO	0	0xE0004038 T0CR3 不可用	0xE0008038 T1CR3
EMR	外部匹配寄存器。EMR 控制外部匹配管脚 MAT0.2..0 和 MAT1.3..0 注：MAT0.3 不连接到 LPC2101/02/03 的管脚	R/W	0	0xE000403C T0EMR	0xE000803C T1EMR
CTCR	计数控制寄存器。CTCR 选择定时器或计数器模式，且在计数器模式下选择计数的信号和边沿	R/W	0	0xE0004070 T0CTCR	0xE0008070 T1CTCR
PWM CON	PWM 控制寄存器。PWMCON 使能外部匹配管脚 MAT0.3..0 和 MAT1.3..0 的 PWM 模式	R/W	0	0xE0004074 PWM0CON	0xE0008074 PWM1CON

2. 基本寄存器组

基本寄存器组主要针对基本计数器功能，包括：中断标志寄存器、定时器控制寄存器、定时器计数器、预分频寄存器和预分频计数器。

（1）中断标志寄存器（T0IR：0xE00040000；T1IR：0xE0008000）

中断标志寄存器包含 4 个用于匹配中断的标志位，4 个用于捕获中断的标志位，如表 4.43 所列。如果有中断产生，寄存器对应位会置位，否则为 0。向对应的位写入“1”会清除中断标志位，写入“0”不影响。

表 4.43 中断标志寄存器描述

IR	功能	描述	复位值
0	MR0 中断	匹配通道 0 的中断标志	0
1	MR1 中断	匹配通道 1 的中断标志	0
2	MR2 中断	匹配通道 2 的中断标志	0
3	MR3 中断	匹配通道 3 的中断标志	0
4	CR0 中断	捕获通道 0 事件的中断标志	0
5	CR1 中断	捕获通道 1 事件的中断标志	0
6	CR2 中断	捕获通道 2 事件的中断标志	0
7	CR3 中断	捕获通道 3 事件的中断标志（不用于定时器 0）	0

操作示例

```
T0IR = 0xFF;                                    /*  清除定时器 0 的全部中断标志*/
```

（2）定时器控制寄存器（T0CR：0xE0004004；T1CR：0xE0008004）

定时器控制寄存器用于控制定时器计数器的操作，该寄存器的描述如表 4.44所列。

表 4.44 定时器控制寄存器描述

TCR	功能	描述	复位值
0	计数器使能	为 1 时，定时器计数器和预分频计数器使能计数。为 0 时，计数器被禁止	0
1	计数器复位	为 1 时，定时器计数器和预分频计数器在 P_{CLK} 的下一个上升沿同步复位。计数器在 TCR 的 bit1 恢复为 0 之前保持复位状态	0
7:2	-	保留，用户不应该向这些位写 1，从这些位读出的值也没意义	NA

操作示例

```
T0TCR = 0x01                                    /*  启动定时器 0                */
```

（3）定时器计数器（T0TC：0xE0004008；T1TC：0xE0008008）

当预分频计数器到达计数的上限时，32 位定时器计数器TC加 1，如图 4.22所示。如果TC在到达计数上限之前没有被复位，它将一直计数到 0xFFFFFFFF然后翻转到 0x00000000，该事件不会产生中断。如果需要，可用匹配寄存器检测溢出。

（4）预分频寄存器（T0PR：0xE000400C；T1PR：0xE000800C）

32 位预分频寄存器指定了预分频计数器的最大值。

（5）预分频计数器寄存器（T0PC：0xE0004010；T1PC：0xE0008010）

预分频计数器使用某个常量来控制P_{CLK}的分频，这样可实现控制定时器分辨率和定时器溢出时间之间的关系。预分频计数器每个P_{CLK}周期加 1，当其到达预分频寄存器中保存的值时，定时器计数器加 1，预分频计数器在下个P_{CLK}周期复位。当PR=0 时，定时器计数器每个P_{CLK}周期加 1；当PR=1 时，定时器计数器每 2 个P_{CLK}周期加 1，如图 4.22所示。

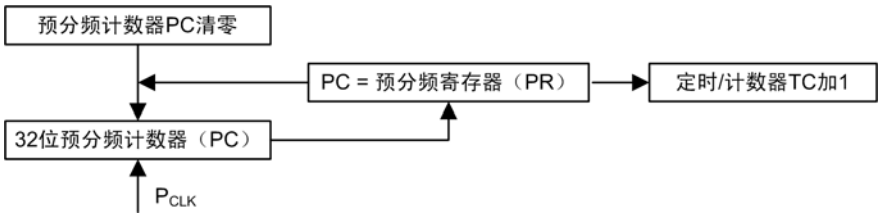


图 4.22 定时器计数示意图

相关的计算公式如下所示：

$$\text{定时时器的计数} = \frac{F_{pclk}}{PR+1}$$

3. 匹配功能寄存器组

匹配功能寄存器组主要针对定时器的匹配功能，包括：匹配寄存器、匹配控制寄存器以及外部匹配寄存器。其中，匹配寄存器用来设置定时器的匹配值，发生匹配事件时，匹配控制寄存器用来设置定时器的动作，外部匹配寄存器用来设置匹配输出引脚的动作。

（1）匹配寄存器（MR0 - MR3）

匹配寄存器值连续与定时器计数值（TC）相比较，当 2 个值相等时，则自动触发产生中断，复位定时器计数器或停止定时器，所执行的动作由 MCR 寄存器控制。

（2）匹配控制寄存器（T0MCR：0xE0004014；T1MCR：0xE0008014）

匹配控制寄存器用于控制在发生匹配时所执行的操作，相关位的描述如表 4.45所列。

表 4.45 匹配控制寄存器描述

MCR	功能	描述	复位值
0	中断（MR0I）	为 1 时，MR0 与 TC 值的匹配将产生中断。为 0 时，中断被禁止	0
1	复位（MR0R）	为 1 时，MR0 与 TC 值的匹配将使 TC 复位。为 0 时，该特性被禁止	0
2	停止（MR0S）	为 1 时，MR0 与 TC 值的匹配将使 TC 和 PC 停止，TCR 的 bit0 清零。为 0 时，该特性被禁止	0
3	中断（MR1I）	为 1 时，MR1 与 TC 值的匹配将产生中断。为 0 时，中断被禁止	0
4	复位（MR1R）	为 1 时，MR1 与 TC 值的匹配将使 TC 复位。为 0 时，该特性被禁止	0
5	停止（MR1S）	为 1 时，MR1 与 TC 值的匹配将使 TC 和 PC 停止，TCR 的 bit0 清零。为 0 时，该特性被禁止	0
6	中断（MR2I）	为 1 时，MR2 与 TC 值的匹配将产生中断。为 0 时，中断被禁止	0
7	复位（MR2R）	为 1 时，MR2 与 TC 值的匹配将使 TC 复位。为 0 时，该特性被禁止	0
8	停止（MR2S）	为 1 时，MR2 与 TC 值的匹配将使 TC 和 PC 停止，TCR 的 bit0 清零。为 0 时，该特性被禁止	0
9	中断（MR3I）	为 1 时，MR3 与 TC 值的匹配将产生中断。为 0 时，中断被禁止	0
10	复位（MR3R）	为 1 时，MR3 与 TC 值的匹配将使 TC 复位。为 0 时，该特性被禁止	0
11	停止（MR3S）	为 1 时，MR3 与 TC 值的匹配将使 TC 和 PC 停止，TCR 的 bit0 清零。为 0 时，该特性被禁止	0

操作示例

```
T0MR0    = 100;                    /* 设置匹配寄存器                */
T0MCR    = 0x03;                   /* 匹配时复位，并产生中断        */
```

（3）外部匹配寄存器（T0EMR：0xE000403C；T1EMR：0xE0008003C）

外部匹配寄存器提供外部匹配管脚MATn.0～MATn.3（n为 0 或 1）的控制和状态，EMR寄存器描述如表 4.46所列，EMR外部匹配控制如表 4.47所列。

当匹配输出配置为 PWM 输出时，外部匹配寄存器的功能由 PWM 决定。

表 4.46 外部匹配寄存器描述

EMR	功能	描述	复位值
0	外部匹配 0	不管 MAT0.0/MAT1.0 是否连接到管脚，该位都会反映 MAT0.0/MAT1.0 的状态。当 MR0 发生匹配时，该输出可翻转，变为低电平，变为高电平或不执行任何动作。位 EMR[4:5]控制该输出的功能	0
1	外部匹配 1	不管 MAT0.1/MAT1.1 是否连接到管脚，该位都会反映 MAT0.1/MAT1.1 的状态。当 MR1 发生匹配时，该输出可翻转，变为低电平，变为高电平或不执行任何动作。位 EMR[6:7]控制该输出的功能	0
2	外部匹配 2	不管 MAT0.2/MAT1.2 是否连接到管脚，该位都会反映 MAT0.2/MAT1.2 的状态。当 MR2 发生匹配时，该输出可翻转，变为低电平，变为高电平或不执行任何动作。位 EMR[8:9]控制该输出的功能	0
3	外部匹配 3	不管 MAT0.3/MAT1.3 是否连接到管脚，该位都会反映 MAT0.3/MAT1.3 的状态。当 MR3 发生匹配时，该输出可翻转，变为低电平，变为高电平或不执行任何动作。位 EMR[10:11]控制该输出的功能	0
5:4	外部匹配控制 0	决定外部匹配 0 的功能，如表 4.47所列	0
7:6	外部匹配控制 1	决定外部匹配 1 的功能，如表 4.47所列	0

续上表

EMR	功能	描述	复位值
9:8	外部匹配控制 2	决定外部匹配 2 的功能，如表 4.47所列	0
11:10	外部匹配控制 3	决定外部匹配 3 的功能，如表 4.47所列	0

表 4.47 EMR 外部匹配控制

EMR[11:10]、EMR[9:8] EMR[7:6]、 EMR[5:4]	功能
00	不执行任何动作
01	将对应的外部匹配输出设置为 0（如果连接到管脚，则输出低电平）
10	将对应的外部匹配输出设置为 1（如果连接到管脚，则输出高电平）
11	使对应的外部匹配输出翻转

操作示例

```
T0EMR = 0x30;                    /* 定时器 0 发生匹配时，MAT0.0 引脚输出翻转*/
```

4. 捕获功能寄存器组

捕获功能寄存器组针对定时器的捕获功能，包括：捕获寄存器和捕获控制寄存器。其中，捕获控制寄存器用来设置捕获信号，发生捕获事件时，定时器的计数值保存到捕获寄存器中。

（1）捕获寄存器（CR0 - CR3）

每个捕获寄存器都与一个器件管脚相关联。当管脚发生特定的事件时，可将定时器计数值装入该寄存器，捕获控制寄存器的设定决定捕获功能是否使能以及捕获事件在管脚的上升沿、下降沿或是双边沿发生。

（2）捕获控制寄存器（T0CCR：0xE0004028；T1CCR：0xE0008028）

捕获控制寄存器的功能：

- 设置捕获事件发生的位置：上升沿、下降沿还是上升沿+下降沿；
- 捕获事件发生时，是否产生中断。

CCR寄存器描述如表 4.48。在下面的描述中，“n”代表定时器的编号 0 或 1。每路捕获功能，都是由 3 个位控制的。

表 4.48 捕获控制寄存器描述

CCR	功能	描述	复位值
0	CAP0RE 上升沿捕获	为 1 时，CAPn.0 上 0 到 1 的跳变将导致 TC 的内容装入 CR0。 为 0 时，该特性被禁止	0
1	CAP0FE 下降沿捕获	为 1 时，CAPn.0 上 1 到 0 的跳变将导致 TC 的内容装入 CR0。 为 0 时，该特性被禁止	0
2	CAP0I 事件中断	为 1 时，CAPn.0 的捕获事件所导致的 CR0 装载将产生一个中断。 为 0 时，该特性被禁止	0
3	CAP1RE 上升沿捕获	为 1 时，CAPn.1 上 0 到 1 的跳变将导致 TC 的内容装入 CR1。 为 0 时，该特性被禁止	0
4	CAP1FE 下降沿捕获	为 1 时，CAPn.1 上 1 到 0 的跳变将导致 TC 的内容装入 CR1。 为 0 时，该特性被禁止	0

续上表

CCR	功能	描述	复位值
5	CAP1I 事件中断	为 1 时，CAPn.1 的捕获事件所导致的 CR1 装载将产生一个中断。 为 0 时，该特性被禁止	0
6	CAP2RE 上升沿捕获	为 1 时，CAPn.2 上 0 到 1 的跳变将导致 TC 的内容装入 CR2。 为 0 时，该特性被禁止	0
7	CAP2FE 下降沿捕获	为 1 时，CAPn.2 上 1 到 0 的跳变将导致 TC 的内容装入 CR2。 为 0 时，该特性被禁止	0
8	CAP2I 事件中断	为 1 时，CAPn.2 的捕获事件所导致的 CR2 装载将产生一个中断。 为 0 时，该特性被禁止	0
9	CAP3RE 上升沿捕获	为 1 时，CAPn.3 上 0 到 1 的跳变将导致 TC 的内容装入 CR3。 为 0 时，该特性被禁止	0
10	CAP3FE 下降沿捕获	为 1 时，CAPn.3 上 1 到 0 的跳变将导致 TC 的内容装入 CR3。 为 0 时，该特性被禁止	0
11	CAP3I 事件中断	为 1 时，CAPn.3 的捕获事件所导致的 CR3 装载将产生一个中断。 为 0 时，该特性被禁止	0

操作示例

```
T0CCR = 0x05;                    /* 当 CAP0.0 引脚出现上升沿时
                                    发生捕获事件，并产生中断      */
```

5. 其它寄存器

（1）计数控制寄存器（T0CTCR：0xE0004070；T1CTCR：0xE0008070）

计数控制寄存器用来选择定时模式还是计数模式，同时在计数模式下，用来选择引脚和选择边沿计数（上升沿还是下降沿）。

当选择计数模式时，捕获输入引脚（由 CTCR[3:2]来选择）在每个 P_{CLK} 的上升沿采样。在比较两个连续的捕获输入引脚采样值之后，将会识别为上升沿、下降沿、边沿的任一种或捕获输入引脚的电平没有变化。只有 CTCR[1:0]选择的事件被识别后，定时计数寄存器才会增加。

给计数器提供的外部时钟有些限制。因为需要连续两个 P_{CLK} 的上升沿采样才能识别一个捕获输入引脚的变化，捕获输入引脚的输入频率不能超过 P_{CLK} 的 1/2。在这种情况下，高/低电平的持续时间必须不小于 $1/P_{CLK}$。

表 4.49 计数控制寄存器描述

CTCR	功能	描述	复位值
1:0	计数/定时模式	选择 P_{CLK} 的边沿来使 PC 加 1，或清零 PC 和 TC 加 1。 00：定时模式，每个 P_{CLK} 的上升沿 01：计数模式，位 3:2 选择的捕获输入的上升沿使 TC 加 1 10：计数模式，位 3:2 选择的捕获输入的下降沿使 TC 加 1 11：计数模式，位 3:2 选择的捕获输入的上升沿和下降沿都使 TC 加 1	00

续上表

CTCR	功能	描述	复位值
3:2	计数输入选择	当寄存器位 1:0 的值不为 00 时，这些位用来选择哪个捕获输入引脚被采样。 00：CAPn.0（CAP0.0 对应定时器 0 和 CAP1.0 对应定时器 1） 01：CAPn.1（CAP0.1 对应定时器 0 和 CAP1.1 对应定时器 1） 10：CAPn.2（CAP0.2 对应定时器 0 和 CAP1.2 对应定时器 1） 11：CAP1.3 对应定时器 1 注意：如果一个捕获输入的引脚由 TnCTCR 选择为计数模式，则对应该输入引脚的 TnCCR 寄存器相应的 3 位应该写 0，对于其它 3 个捕获输入可选择捕获或中断 注意：定时器 0 中没有 CAP0.3	00
7:4	-	保留	NA

（2）PWM 控制寄存器（PWM0CON-0xE0004074，PWM1CON-0xE0008074）

PWM控制寄存器用来控制配置匹配输出为PWM输出。每个匹配输出可独立的设置为PWM输出。对于每个定时器，最多可选择 3 个单边沿PWM输出在MATn.2:0 引脚上。另外一个匹配寄存器用来决定PWM输出的周期，当其它任何一个匹配寄存器发生匹配时，PWM输出将会置为高电平。定时器可被选择作为PWM周期的匹配寄存器复位。当定时器复位为零时，所有PWM的输出将会置为低电平。PWM控制寄存器描述如表 4.50所列。

表 4.50 PWM 控制寄存器描述（定时器 0、1）

PWMCON	功能	描述	复位值
0	PWM 使能	写 1：MATn.0 使能为 PWM 模式 写 0：MATn.0 由 EM0 控制	0
1	PWM 使能	写 1：MATn.0 使能为 PWM 模式 写 0：MATn.0 由 EM0 控制	0
2	PWM 使能	写 1：MATn.0 使能为 PWM 模式 写 0：MATn.0 由 EM0 控制	0
3	PWM 使能	写 1：MATn.0 使能为 PWM 模式 写 0：MATn.0 由 EM0 控制 注意：推荐用 MATn.3 来设置 PWM 的输出周期，因为定时器 0 的 MAT0.3 没有连接引脚	0
32:4	-	保留	NA

单边沿控制 PWM 输出的规则如下：

- 除了在匹配值为 0 的情况之外，每个 PWM 周期的开始，所有控制的 PWM 输出为低（此时定时器设置为 0）；
- 匹配寄存器发生匹配时，相关的 PWM 输出将会置高。如果没有匹配发生（例如，匹配值大于 PWM 周期），PWM 输出将会一直输出低电平；
- 如果匹配寄存器的值大于 PWM 的输出周期时，且 PWM 的输出为高电平，则在定时器复位时，PWM 输出将会被清零；
- 如果有一个匹配寄存器的值跟 PWM 周期值一样，则在下一个时钟 PWM 周期计数时钟之后将会复位，因此，一个 PWM 将由一个时钟宽度的高电平组成，宽度由 PWM 的计数时钟决定；

- 如果一个匹配寄存器的值为 0，则第一次的 PWM 输出为高电平，同时在定时器复位以后还将一直保持高电平。

4.5.5 定时器中断

LPC2103 含有两个 32 位定时器，每个定时器可以产生 8 种类型的中断：4 路匹配中断、4 路捕获中断，可以通过读取中断标志寄存器（TnIR）来区分中断类型。定时器中断与向量中断控制器（VIC）的关系如图 4.23所示。

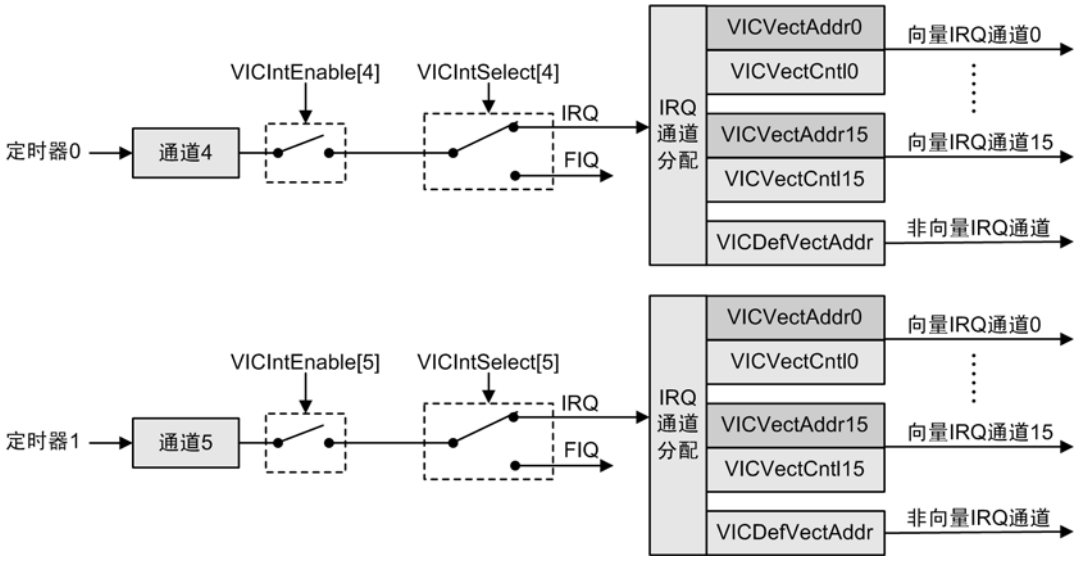


图 4.23 定时器 0、1 与 VIC 的关系

定时器 0 和定时器 1 分别处于 VIC 的通道 4 和通道 5，中断使能寄存器 VICIntEnable 用来控制 VIC 通道的中断使能。

- 当 VICIntEnable[4] = 1 时，通道 4 中断使能，即：定时器 0 中断使能；
- 当 VICIntEnable[5] = 1 时，通道 5 中断使能，即：定时器 1 中断使能。

中断选择寄存器 VICIntSelect 用来分配 VIC 通道的中断。当某一位为“1”时，对应的通道中断分配为 FIQ；当某一位为“0”时，对应的通道中断分配为 IRQ。VICIntSelect[4]和 VICIntSelect[5]分别用来控制通道 4 和通道 5，即：

- 当 VICIntSelect[4] = 1 时，定时器 0 中断分配为 FIQ 中断；
- 当 VICIntSelect[4] = 0 时，定时器 0 中断分配为 IRQ 中断；
- 当 VICIntSelect[5] = 1 时，定时器 1 中断分配为 FIQ 中断；
- 当 VICIntSelect[5] = 0 时，定时器 1 中断分配为 IRQ 中断。

分配为 IRQ 之后，还需要设置对应的通道控制寄存器和地址寄存器。有关寄存器 VICVectCntln 和 VICVectAddrn 的说明，请参考“向量中断控制器（VIC）”一节。

1. 匹配中断

LPC2103 定时器计数溢出时不会产生中断，但是匹配时可以产生中断。每个定时器都具有 4 个匹配寄存器（MR0~MR3），可以用来存放匹配值，当定时器的当前计数值 TC 等于匹配值 MR 时，就可以产生中断。

寄存器TnMCR控制匹配中断的使能，以定时器 0 为例，定时器匹配控制寄存器TnMCR用来使能定时器的匹配中断，如图 4.24所示。

- 当 T0TC = T0MR0 时，发生匹配事件 0，若 T0MCR[0] = 1，则 T0IR[0]置位；
- 当 T0TC = T0MR1 时，发生匹配事件 1，若 T0MCR[3] = 1，则 T0IR[1]置位；
- 当 T0TC = T0MR2 时，发生匹配事件 2，若 T0MCR[6] = 1，则 T0IR[2]置位；
- 当 T0TC = T0MR3 时，发生匹配事件 3，若 T0MCR[7] = 1，则 T0IR[3]置位。

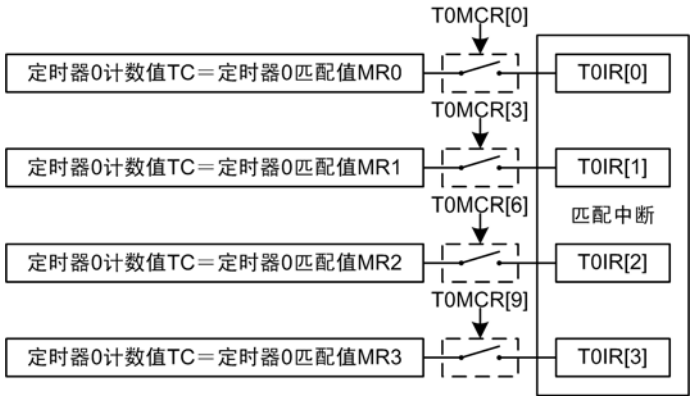


图 4.24 匹配中断示意图

2. 捕获中断

当定时器的捕获引脚CAP上出现特定的捕获信号时，可以产生中断。以CAP0.0 为例，如图 4.25所示。

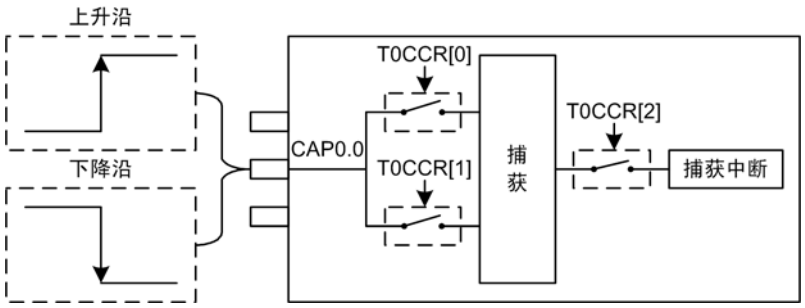


图 4.25 捕获中断示意图

捕获控制寄存器TnCCR用来设置定时器的捕获功能，包括捕获信号和中断使能等，如图 4.25所示。

- 若 T0CCR[0] = 1，捕获引脚 CAP0.0 上出现“上升沿”信号时，发生捕获事件；
- 若 T0CCR[1] = 1，捕获引脚 CAP0.0 上出现“下降沿”信号时，发生捕获事件。

发生捕获事件时，若 T0CCR[2] = 1，则捕获中断使能。

4.5.6 应用示例

LPC2103 两个 32 位定时器，分别具有 4 路捕获、4 路比较匹配并输出电路，定时器是增量计数的，但上溢时不会产生中断标志，而只能通过比较匹配或捕获输入产生中断标志。两个定时器具有同样的寄存器，只是地址不同而已。

如图 4.26所示：

- 32 位定时器 TC 的计数频率由 Fpclk 经过预分频器分频得到；
- 定时器的启动/停止、计数复位由 TCR 控制；

- 定时器溢出时不会产生中断，定时器的中断是由捕获事件或匹配事件引发的，所以上图采用虚线连接。

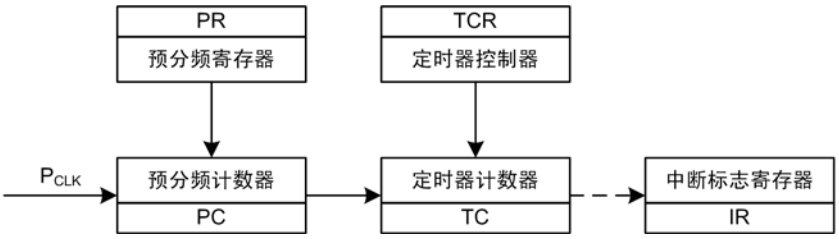


图 4.26 定时器的寄存器功能框图

如图 4.27所示：

- 定时器的比较匹配功能由寄存器 MCR 进行控制；
- MR0～3 寄存器为 4 路比较匹配通道的比较值；
- 当发生匹配时，按照 MCR 设置的方法产生中断或复位 TC 等；
- 当发生匹配时，EMR 控制匹配引脚输出——高电平、低电平、引脚电平翻转等。

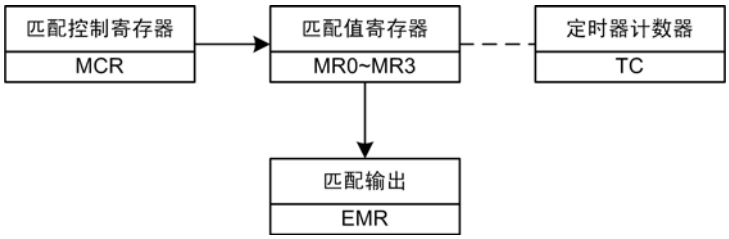


图 4.27 定时器的比较匹配寄存器功能框图

如图 4.28所示：

- 定时器的捕获功能由寄存器 CCR 进行控制；
- 通过 CCR 寄存器，捕获事件可以设定为上升沿触发、下降沿触发、双边沿触发；
- 通过 CCR 寄存器，可以设定，当捕获事件发生时，是否产生中断；
- CR0～3 寄存器为 4 路捕获寄存器，用来保存对应的捕获值；
- 捕获事件发生时，捕获电路将会立即把当时的定时器值 TC 保存到对应的捕获寄存器中。

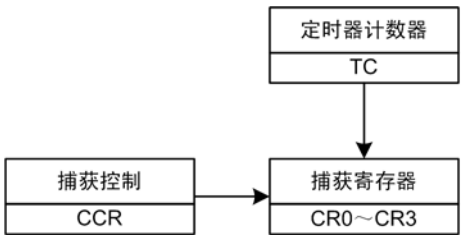


图 4.28 定时器的捕获寄存器功能框图

定时器基本操作方法：

- 计算定时器的时钟频率，设置 PR 寄存器进行分频操作；
- 若使用匹配功能，则设置匹配通道的初值及其工作模式；若使用捕获功能，则设置捕获方式；
- 若使用定时器的相关中断，则设置 VIC，使能中断；

- 设置 TCR，启动定时器。

定时器计数时钟频率计算如下：

$$计数时钟频率=\frac{F_{pclk}}{N+1}$$

其中，N 为 PR 的值。

1. 定时中断

定时器 0 匹配产生中断，示例应用如程序清单 4.11所示。用户需要短接JP4 的P0.17 端口，输出控制 LED 闪烁。

程序清单 4.11 定时器 0 定时中断

```
#include "config.h"
#define   LED   (1 << 17)

/*********************************************************************************************
** 函数名称：Timer0_ISR
** 函数功能：定时器 0 中断主程序
** 输入参数：无
** 输出参数：无
*********************************************************************************************/
void  __irq Timer0ISR (void)
{
    if ((IO0PIN & LED) == 0) {
        IO0SET = LED;                                   /* 点亮发光二极管              */
    }
    else  IO0CLR   = LED;                               /* 熄灭发光二极管              */
    T0IR           =    0x01;                           /* 清除中断标志                */
    VICVectAddr    =    0x00;                           /* 中断向量结束                */
}
/*********************************************************************************************
** 函数名称：Timer0Init
** 函数功能：定时器 0 初始化
** 输入参数：无
** 输出参数：无
*********************************************************************************************/
void  Timer0Init(void)
{
    T0TCR    = 0x02;                                    /* 定时器 0 复位               */
    T0PR           = 0;                                 /* 不设时钟分频                */
    T0MCR    = 0x03;                                    /* 匹配后复位 TC，并产生中断 */
    T0MR0    = Fpclk / 2;                               /* 设置 0.5 秒匹配值           */
    T0IR     = 0x01;                                    /* 清除中断标志                */
    T0TCR    = 0x01;                                    /* 启动定时器 0                */
```

```
}
/*********************************************************************************************
** 函数名称：main
** 函数功能：定时器 0 匹配产生中断
** 输入函数：无
** 输出参数：无
*********************************************************************************************/
int main (void)
{
    PINSEL1   = PINSEL1 & 0xFFFFFFF3;                    /* 选择 P0.17 的 GPIO 功能          */
    IO0DIR    = LED;                                     /* 设置 P0.17 为输出                */
    IO0SET    = LED;                                     /* 设置输出高电平                   */

    Timer0Init();                                        /* 定时器 0 初始化                  */
    IRQEnable();                                         /* IRQ 中断使能                     */
    /* 设置向量中断控制器          */
    VICIntSelect      = VICIntSelect & (~(1 << 4));      /* 定时器 0 分配为 IRQ 中断         */
    VICVectCntl0      = 0x20 | 4;                        /* 定时器 0 分配为向量 IRQ 通道 0   */
    VICVectAddr0      = (uint32) Timer0ISR;              /* 分配中断服务程序地址             */
    VICIntEnable      = 1 << 4;                          /* 定时器 0 中断使能                */

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

2. 匹配输出

由于匹配寄存器可以匹配输出翻转，定时器可以匹配输出占空比为 50%的方波，高低电平持续时间为均定时的时长。

定时器 1 匹配输出方波，示例应用如程序清单 4.12所示，定时时间为 0.2 秒。用户需要短接 JP4 的 P0.19 端口，控制 LED 闪烁。

程序清单 4.12 定时器 1 匹配输出

```
#include "config.h"
#define      LED              (1 << 19)
/*********************************************************************************************
** 函数名称：Timer1Init
** 函数功能：定时器 1 初始化
** 输入参数：无
** 输出参数：无
*********************************************************************************************/
void   Timer1Init(void)
{
    T1TCR     = 0x02;                                    /* 定时器 1 复位                    */
    T1PR            = 0;                                 /* 不设时钟分频                     */
```

```
    T1MCR    = 0x02;                                    /* 设置 T1MR1 匹配后复位 T1TC    */

    T1EMR    = 0x03 << 8;                               /* 匹配翻转                      */
    T1MR0    = Fpclk / 5;                               /* 设置 0.2 秒匹配值             */
    T1IR     = 0x01;                                    /* 清除中断标志                  */
    T1TCR    = 0x01;                                    /*  启动定时器 1                 */
}
/*********************************************************************************************
** 函数名称：main
** 函数功能：定时器 1 匹配输出翻转主程序
** 输入参数：无
** 输出参数：无
*********************************************************************************************/
int main (void)
{
    PINSEL1  = PINSEL1 & (0x03 << 6) ;                  /* 选择 MAT1.2 输出              */
    PINSEL1  = PINSEL1 | (0x02 << 6);                   /* 选择 MAT1.2 输出              */

    Timer1Init();                                       /* 定时器 1 初始化               */
    IRQEnable();                                        /* IRQ 中断使能                  */

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

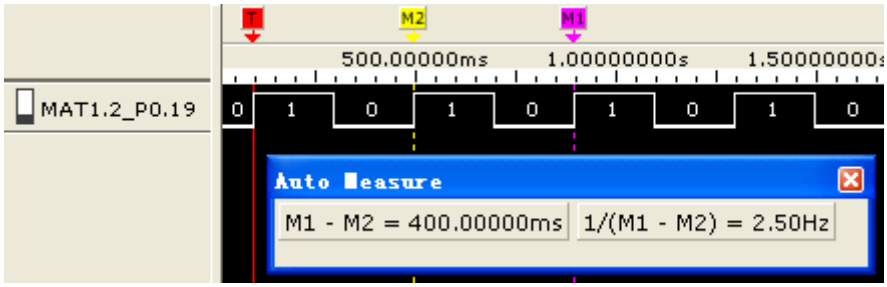


图 4.29 匹配输出波形

3. PWM输出

定时器 0、1 可匹配输出 PWM 波形，波形的占空比自行设定。在本应用示例中，P0.19 输出的波形占空比为 50%，P0.20 输出的占空比为 25%。

定时器PWM输出应用，示例如程序清单 4.13所示。产生波形如图 4.30所示。

程序清单 4.13 定时器 1 PWM 输出初始化

```
#include "config.h"

/*********************************************************************************************
** 函数名称：Timer1Init
```

```
** 函数功能：定时器 1 初始化
** 输入参数：无
** 输出参数：无
*********************************************************************************************/
void   Timer1Init(void)
{
    T1TCR    =    0x02;                          /* 定时器 0 复位                    */
    T1PR     =    0;                             /* 不设时钟分频                     */
    PWM1CON=     0x0C;                          /* 使能 PWM 输出                    */
    T1MCR    = 0x02;                            /* 设置 T0MR0 匹配后复位 T0TC       */
    T1MR0    = Fpclk / 2000;                    /* 设置 PWM 输出的周期              */
    T1MR2    = (Fpclk / 2000) / 2;              /* 设置 PWM1.2 输出占空比为 50%     */
    T1MR3    = ((Fpclk / 2000) / 4) * 3;        /* 设置 PWM1.3 输出占空比为 25%     */
    T1TCR    = 0x01;                            /* 启动定时器 0                     */
}
/*********************************************************************************************
** 函数名称：main
** 函数功能：定时器 1PWM 输出
** 输入参数：无
** 输出参数：无
*********************************************************************************************/
int main (void)
{
    PINSEL1  = (PINSEL1 & (~(0x03 << 6))) | (0x02 << 6);    /* 选择 MAT1.2 输出     */
    PINSEL1  = (PINSEL1 & (~(0x03 << 8))) | (0x02 << 8);    /* 选择 MAT1.3 输出     */

    Timer1Init();                                           /* 定时器 1 初始化      */
    while(1);
    return 0;
}
```

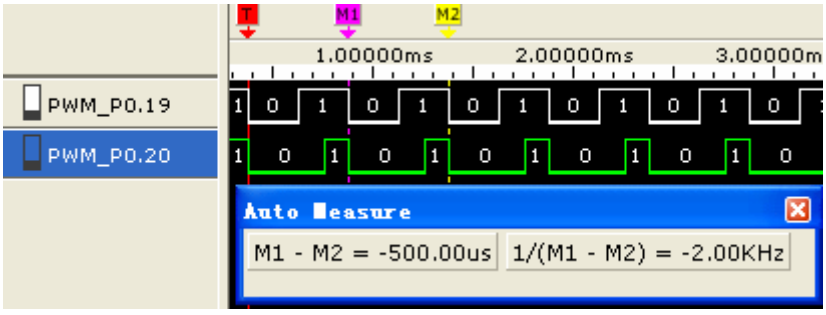


图 4.30　PWM 输出波形

4.6 定时器 2 和定时器 3

4.6.1 概述

定时器/计数器对外设时钟（P_{CLK}）或外部提供的时钟周期进行计数，可选择产生中断或根据 4 个匹配寄存器的设定，在到达指定的定时值时执行其它动作。它还包括 4 个捕获输入，用于在输入信号发生跳变时捕获定时器值，并可选择产生中断。

由于 LPC2101/02/03 器件的管脚数目有限，因此定时器 3 没有捕获输入，只有定时器 2 的 3 个捕获输入和 3 个匹配输出，以及定时器 3 的 4 个匹配输出被连接到器件的管脚。

2 个匹配寄存器可以在 MATn.2..0 管脚上提供单边沿控制的 PWM 输出。由于 MAT2.3 寄存器在定时器 2 上没有管脚输出，因此建议使用 MRn.3 寄存器来控制 PWM 周期长度，需要用另一个匹配寄存器来控制 PWM 边沿位置，剩余的两个匹配寄存器可用来创建其它的 PWM 输出，该输出具有 MRn.3 决定的 PWM 周期率

4.6.2 特性

- 带可编程 16 位预分频器的 16 位定时器/计数器。
- 计数器或定时器操作。
- 具有 3 路（定时器 2）16 位的捕获通道。当输入信号跳变时可取得定时器的瞬时值，也可选择使捕获事件产生中断。
- 4 个 16 位匹配寄存器：
  - 匹配时定时器继续工作，可选择产生中断；
  - 匹配时停止定时器，可选择产生中断；
  - 匹配时复位定时器，可选择产生中断。
- 多达 4 个（定时器 3）和 3 个（定时器 2）对应于匹配寄存器的外部输出，具有下列特性：
  - 匹配时设置为低电平；
  - 匹配时设置为高电平；
  - 匹配时翻转；
  - 匹配时无动作。
- 对于每个定时器，多达 4 个匹配寄存器可配置为 PWM，允许使用多达 3 个匹配输出作为单边沿控制的 PWM 输出。

4.6.3 管脚描述

定时器/计数器相关管脚的简要描述如表 4.51所列。

表 4.51 定时器/计数器 2、3 管脚描述

管脚名称	管脚方向	管脚描述
CAP2.2..0	输入	**捕获信号** 捕获管脚的跳变可配置为将定时器值装入一个捕获寄存器，并可选择产生一个中断 下面是所有捕获信号及其选择的管脚的列表： CAP2.0：P0.27 CAP2.1：P0.28 CAP2.2：P0.29 计数器/定时器可选择一个捕获信号作为时钟源来代替 P_{CLK} 分频时钟

续上表

管脚名称	管脚方向	管脚描述
MAT2.2..0 MAT3.3..0	输出	**外部匹配输出 0/1** 当匹配寄存器 0/1（MR3：0）等于定时器计数器（TC）时，该输出可翻转，变为低电平、变为高电平或不变。外部匹配寄存器（EMR）和 PWM 控制寄存器（PWMCON）控制该输出的功能 下面是所有匹配信号及其选择的管脚的列表： MAT2.0：P0.7 MAT2.1：P0.8 MAT2.2：P0.9 MAT3.0：P0.21 MAT3.1：P0.0 MAT3.2：P0.1 MAT3.3：P0.30

4.6.4 寄存器描述

1. 寄存器汇总

每个定时器/计数器所包含的寄存器如表 4.52所列。

表 4.52 定时器/计数器 2、3 寄存器映射

通用名称	描述	访问	复位值	定时器/计数器 2 地址&名称	定时器/计数器 3 地址&名称
IR	中断寄存器。可以写 IR 来清除中断。可读取 IR 来识别中断源（8 个可能中断源中）被挂起	R/W	0	0xE0070000 T2IR	0xE0074000 T3IR
TCR	定时器控制寄存器。TCR 用于控制定时器计数器功能。定时器计数器可通过 TCR 禁止或复位	R/W	0	0xE0070004 T2TCR	0xE0074004 T3TCR
TC	定时器计数器。16 位 TC 每经过 PR+1 个 P_{CLK} 周期加 1。TC 通过 TCR 进行控制	R/W	0	0xE0070008 T2TC	0xE0074008 T3TC
PR	预分频寄存器。预分频计数器（下面）等于该值时，下个时钟增加 TC 并清除 PC	R/W	0	0xE007000C T2PR	0xE007400C T3PR
PC	预分频计数器。16 位 PC 是加 1 到 PR 内的存储值的计数器。当到达 PR 中保存的值时，TC 加 1 且 PC 被清除。通过总线接口可观察和控制 PC	R/W	0	0xE0070010 T2PC	0xE0074010 T3PC
MCR	匹配控制寄存器。MCR 用于控制在匹配时是否产生中断或复位 TC	R/W	0	0xE0070014 T2MCR	0xE0074014 T3MCR
MR0	匹配寄存器 0。MR0 可通过 MCR 设定为在每次 MR0 匹配 TC 时复位 TC，停止 TC 和 PC，和/或产生中断	R/W	0	0xE0070018 T2MR0	0xE0074018 T3MR0
MR1	匹配寄存器 1。见 MR0 描述	R/W	0	0xE007001C T2MR1	0xE007401C T3MR1
MR2	匹配寄存器 2。见 MR0 描述	R/W	0	0xE0070020 T2MR2	0xE0074020 T3MR2
MR3	匹配寄存器 3。见 MR0 描述	R/W	0	0xE0070024 T2MR3	0xE0074024 T3MR3
CCR	捕获控制寄存器。CCR 控制用于装载捕获寄存器的捕获输入边沿以及在发生捕获时是否产生中断	R/W	0	0xE0070028 T2CCR	0xE0074028 T3CCR
CR0	捕获寄存器 0。当在 CAP2.0 上产生捕获事件时，CR0 装载 TC 的值 注：CAP3.0 在定时器 3 上不可用	RO	0	0xE007002C T2CR0	0xE007402C T3CR0

续上表

通用名称	描述	访问	复位值	定时器/计数器 2 地址&名称	定时器/计数器 3 地址&名称
CR1	捕获寄存器 1。见 CR0 描述 注：CAP3.1 在定时器 3 上不可用	RO	0	0xE0070030 T2CR1	0xE0074030 T3CR1
CR2	捕获寄存器 2。见 CR0 描述 注：CAP3.2 在定时器 3 上不可用	RO	0	0xE0070034 T2CR2	0xE0074034 T3CR2
EMR	外部匹配寄存器。EMR 控制外部匹配管脚 MAT2.2..0 和 MAT3.3..0 注：MAT2.3 不连接到 LPC2101/02/03 的管脚	R/W	0	0xE007003C T2EMR	0xE007403C T3EMR
CTCR	计数控制寄存器。CTCR 选择定时器或计数器模式，且在计数器模式下选择计数的信号和边沿	R/W	0	0xE0070070 T2CTCR	0xE0074070 T3CTCR
PWM CON	PWM 控制寄存器。PWMCON 使能外部匹配管脚 MAT2.3..0 和 MAT3.3..0 的 PWM 模式	R/W	0	0xE0070074 PWM2CON	0xE0074074 PWM3CON

2. 基本寄存器组

基本寄存器组主要针对基本计数器功能，包括：中断标志寄存器、定时器控制寄存器、定时器计数器、预分频寄存器和预分频计数器。

（1）中断标志寄存器（T2IR：0xE0070000；T3IR：0xE0074000）

中断标志寄存器包含 4 个用于匹配中断的标志位，4 个用于捕获中断的标志位，如表 4.53 所列。如果有中断产生，寄存器对应位会置位，否则为 0。向对应的位写入“1”会清除中断标志位，写入“0”不影响。

表 4.53 中断标志寄存器描述

IR	功能	描述	复位值
0	MR0 中断	匹配通道 0 的中断标志	0
1	MR1 中断	匹配通道 1 的中断标志	0
2	MR2 中断	匹配通道 2 的中断标志	0
3	MR3 中断	匹配通道 3 的中断标志	0
4	CR0 中断	捕获通道 0 事件的中断标志	0
5	CR1 中断	捕获通道 1 事件的中断标志	0
6	CR2 中断	捕获通道 2 事件的中断标志	0
7	CR3 中断	捕获通道 3 事件的中断标志（不用于定时器 2、3）	0

操作示例

```
T2IR = 0xFF;                                    /*  清除定时器 2 的全部中断标志*/
```

（2）定时器控制寄存器（T2CR：0xE0070004；T3CR：0xE0074004）

定时器控制寄存器用于控制定时器计数器的操作，该寄存器的描述如表 4.54所列。

表 4.54 定时器控制寄存器描述

TCR	功能	描述	复位值
0	计数器使能	为 1 时，定时器计数器和预分频计数器使能计数。为 0 时，计数器被禁止	0

续上表

TCR	功能	描述	复位值
1	计数器复位	为 1 时，定时器计数器和预分频计数器在 P_{CLK} 的下一个上升沿同步复位。计数器在 TCR 的 bit1 恢复为 0 之前保持复位状态	0
7:2	-	保留，用户不应该向这些位写 1，从这些位读出的值也没意义	NA

操作示例

```
T2TCR = 0x01                                    /*  启动定时器 0                */
```

（3）定时器计数器（T2TC：0xE0070008；T3TC：0xE0074008）

当预分频计数器到达计数的上限时，16 位定时器计数器 TC 加 1。如果 TC 在到达计数上限之前没有被复位，它将一直计数到 0xFFFFFFFF 然后翻转到 0xE0000000，该事件不会产生中断。如果需要，可用匹配寄存器检测溢出。

（4）预分频寄存器（T2PR：0xE007000C；T1PR：0xE007400C）

16 位预分频寄存器指定了预分频计数器的最大值。

（5）预分频计数器寄存器（T2PC：0xE0070010；T3PC：0xE0074010）

这个 16 位的预分频计数器使用某个常量来控制P_{CLK}的分频，这样可实现控制定时器分辨率和定时器溢出时间之间的关系。预分频计数器每个P_{CLK}周期加 1，当其到达预分频寄存器中保存的值时，定时器计数器加 1，预分频计数器在下个P_{CLK}周期复位。当PR=0 时，定时器计数器每个P_{CLK}周期加 1；当PR=1 时，定时器计数器每 2 个P_{CLK}周期加 1，如图 4.22所示。

3. 匹配功能寄存器组

匹配功能寄存器组主要针对定时器的匹配功能，包括：匹配寄存器、匹配控制寄存器以及外部匹配寄存器。其中，匹配寄存器用来设置定时器的匹配值，发生匹配事件时，匹配控制寄存器用来设置定时器的动作，外部匹配寄存器用来设置匹配输出引脚的动作。

（1）匹配寄存器（MR0 - MR3）

匹配寄存器值连续与定时器计数值（TC）相比较，当 2 个值相等时，则自动触发产生中断，复位定时器计数器或停止定时器，所执行的动作由 MCR 寄存器控制。

（2）匹配控制寄存器（T2MCR：0xE0070014；T3MCR：0xE00074014）

匹配控制寄存器用于控制在发生匹配时所执行的操作，相关位的描述如表 4.45所列。

表 4.55 匹配寄存器描述（定时器 2、3）

MCR	功能	描述	复位值
0	中断（MR0I）	为 1 时，MR0 与 TC 值的匹配将产生中断。为 0 时，中断被禁止	0
1	复位（MR0R）	为 1 时，MR0 与 TC 值的匹配将使 TC 复位。为 0 时，该特性被禁止	0
2	停止（MR0S）	为 1 时，MR0 与 TC 值的匹配将使 TC 和 PC 停止，TCR 的 bit0 清零。为 0 时，该特性被禁止	0
3	中断（MR1I）	为 1 时，MR1 与 TC 值的匹配将产生中断。为 0 时，中断被禁止	0
4	复位（MR1R）	为 1 时，MR1 与 TC 值的匹配将使 TC 复位。为 0 时，该特性被禁止	0
5	停止（MR1S）	为 1 时，MR1 与 TC 值的匹配将使 TC 和 PC 停止，TCR 的 bit0 清零。为 0 时，该特性被禁止	0
6	中断（MR2I）	为 1 时，MR2 与 TC 值的匹配将产生中断。为 0 时，中断被禁止	0

续上表

MCR	功能	描述	复位值
7	复位（MR2R）	为 1 时，MR2 与 TC 值的匹配将使 TC 复位。为 0 时，该特性被禁止	0
8	停止（MR2S）	为 1 时，MR2 与 TC 值的匹配将使 TC 和 PC 停止，TCR 的 bit0 清零。为 0 时，该特性被禁止	0
9	中断（MR3I）	为 1 时，MR3 与 TC 值的匹配将产生中断。为 0 时，中断被禁止	0
10	复位（MR3R）	为 1 时，MR3 与 TC 值的匹配将使 TC 复位。为 0 时，该特性被禁止	0
11	停止（MR3S）	为 1 时，MR3 与 TC 值的匹配将使 TC 和 PC 停止，TCR 的 bit0 清零。为 0 时，该特性被禁止	0

操作示例

```
T2MR0    = 100;                    /* 设置匹配寄存器                    */
T2MCR    = 0x03;                   /* 当定时器 2 与 MR0 匹配时
                                      定时器 2 复位，并产生中断          */
```

（3）外部匹配寄存器（T2EMR：0xE007003C；T3EMR：0xE007403C）

外部匹配寄存器提供外部匹配管脚MATn.0～MATn.3（n为 0 或 1）的控制和状态，EMR寄存器描述如表 4.56所列，EMR外部匹配控制如表 4.57所列。

当匹配输出配置为 PWM 输出时，外部匹配寄存器的功能由 PWM 决定。

表 4.56 外部匹配寄存器描述

EMR	功能	描述	复位值
0	外部匹配 0	不管 MAT0.0/MAT1.0 是否连接到管脚，该位都会反映 MAT0.0/MAT1.0 的状态。当 MR0 发生匹配时，该输出可翻转，变为低电平，变为高电平或不执行任何动作。位 EMR[4:5]控制该输出的功能	0
1	外部匹配 1	不管 MAT0.1/MAT1.1 是否连接到管脚，该位都会反映 MAT0.1/MAT1.1 的状态。当 MR1 发生匹配时，该输出可翻转，变为低电平，变为高电平或不执行任何动作。位 EMR[6:7]控制该输出的功能	0
2	外部匹配 2	不管 MAT0.2/MAT1.2 是否连接到管脚，该位都会反映 MAT0.2/MAT1.2 的状态。当 MR2 发生匹配时，该输出可翻转，变为低电平，变为高电平或不执行任何动作。位 EMR[8:9]控制该输出的功能	0
3	外部匹配 3	不管 MAT0.3/MAT1.3 是否连接到管脚，该位都会反映 MAT0.3/MAT1.3 的状态。当 MR3 发生匹配时，该输出可翻转，变为低电平，变为高电平或不执行任何动作。位 EMR[10:11]控制该输出的功能	0
5:4	外部匹配控制 0	决定外部匹配 0 的功能，如表 4.47所列	0
7:6	外部匹配控制 1	决定外部匹配 1 的功能，如表 4.47所列	0
9:8	外部匹配控制 2	决定外部匹配 2 的功能，如表 4.47所列	0
11:10	外部匹配控制 3	决定外部匹配 3 的功能，如表 4.47所列	0

表 4.57 EMR 外部匹配控制

EMR[11:10]，EMR[9:8] EMR[7:6]或 EMR[5:4]	功能
00	不执行任何动作
01	将对应的外部匹配输出设置为 0（如果连接到管脚，则输出低电平）
10	将对应的外部匹配输出设置为 1（如果连接到管脚，则输出高电平）
11	使对应的外部匹配输出翻转

操作示例

```
T2EMR = 0x30;                    /* 定时器 2 发生匹配时，MAT2.0 引脚输出翻转*/
```

4. 捕获功能寄存器组

捕获功能寄存器组针对定时器的捕获功能，包括：捕获寄存器和捕获控制寄存器。其中，捕获控制寄存器用来设置捕获信号，发生捕获事件时，定时器的计数值保存到捕获寄存器中。

（1）捕获寄存器（CR0 - CR3）

每个捕获寄存器都与一个器件管脚相关联。当管脚发生特定的事件时，可将定时器计数值装入该寄存器，捕获控制寄存器的设定决定捕获功能是否使能以及捕获事件在管脚的上升沿、下降沿或是双边沿发生。

（2）捕获控制寄存器（T2CCR：0xE0070028；T3CCR：0xE0074028）

捕获控制寄存器的功能：

- 设置捕获事件发生的位置：上升沿、下降沿还是上升沿+下降沿；
- 捕获事件发生时，是否产生中断。

CCR寄存器描述如表 4.58所列。在下面的描述中，“n”代表定时器的编号 0 或 1。每路捕获功能，都是由 3 个位控制的。

表 4.58 捕获控制寄存器描述

CCR	功能	描述	复位值
0	CAP0RE 上升沿捕获	为 1 时，CAPn.0 上 0 到 1 的跳变将导致 TC 的内容装入 CR0 为 0 时，该特性被禁止	0
1	CAP0FE 下降沿捕获	为 1 时，CAPn.0 上 1 到 0 的跳变将导致 TC 的内容装入 CR0 为 0 时，该特性被禁止	0
2	CAP0I 事件中断	为 1 时，CAPn.0 的捕获事件所导致的 CR0 装载将产生一个中断 为 0 时，该特性被禁止	0
3	CAP1RE 上升沿捕获	为 1 时，CAPn.1 上 0 到 1 的跳变将导致 TC 的内容装入 CR1 为 0 时，该特性被禁止	0
4	CAP1FE 下降沿捕获	为 1 时，CAPn.1 上 1 到 0 的跳变将导致 TC 的内容装入 CR1 为 0 时，该特性被禁止	0
5	CAP1I 事件中断	为 1 时，CAPn.1 的捕获事件所导致的 CR1 装载将产生一个中断 为 0 时，该特性被禁止	0
6	CAP2RE 上升沿捕获	为 1 时，CAPn.2 上 0 到 1 的跳变将导致 TC 的内容装入 CR2 为 0 时，该特性被禁止	0
7	CAP2FE 下降沿捕获	为 1 时，CAPn.2 上 1 到 0 的跳变将导致 TC 的内容装入 CR2 为 0 时，该特性被禁止	0

续上表

CCR	功能	描述	复位值
8	CAP2I 事件中断	为 1 时，CAPn.2 的捕获事件所导致的 CR2 装载将产生一个中断 为 0 时，该特性被禁止	0
9	CAP3RE 上升沿捕获	为 1 时，CAPn.3 上 0 到 1 的跳变将导致 TC 的内容装入 CR3 为 0 时，该特性被禁止	0
10	CAP3FE 下降沿捕获	为 1 时，CAPn.3 上 1 到 0 的跳变将导致 TC 的内容装入 CR3 为 0 时，该特性被禁止	0
11	CAP3I 事件中断	为 1 时，CAPn.3 的捕获事件所导致的 CR3 装载将产生一个中断 为 0 时，该特性被禁止	0

操作示例

```
T2CCR = 0x05;                          /* 当 CAP2.0 引脚出现上升沿时，
                                          发生捕获事件，并产生中断      */
```

5. 其它寄存器

（1）计数控制寄存器（T2CTCR：0xE0070070；T3CTCR：0xE0074070）

计数控制寄存器用来选择定时模式还是计数模式，同时在计数模式下，用来选择引脚和选择边沿计数（上升沿还是下降沿）。定时器 2、3 的计数控制寄存器描述如表 4.59所列。

当选择计数模式时，捕获输入引脚（由 CTCR[3:2]来选择）在每个 P_{CLK} 的上升沿采样。在比较两个连续的捕获输入引脚采样值之后，将会识别为上升沿、下降沿、边沿的任一种或捕获输入引脚的电平没有变化。只有 CTCR[1:0]选择的事件被识别后，定时计数寄存器才会增加。

给计数器提供的外部时钟有些限制。因为需要连续两个 P_{CLK} 的上升沿采样才能识别一个捕获输入引脚的变化，捕获输入引脚的输入频率不能超过 P_{CLK} 的 1/2。在这种情况下，高/低电平的持续时间必须不小于 $1/P_{CLK}$。

表 4.59 计数控制寄存器描述

CTCR	功能	描述	复位值
1:0	计数/定时模式	选择 P_{CLK} 的边沿来使 PC 加 1，或清零 PC 和 TC 加 1。 00：定时模式，每个 P_{CLK} 的上升沿 01：计数模式，位 3:2 选择的捕获输入的上升沿使 TC 加 1 10：计数模式，位 3:2 选择的捕获输入的下降沿使 TC 加 1 11：计数模式，位 3:2 选择的捕获输入的上升沿和下降沿都使 TC 加 1	00
3:2	计数输入选择	当寄存器位 1:0 的值不为 00 时，这些位用来选择哪个捕获输入引脚被采样。 00：CAP2.0 01：CAP2.1 10：CAP2.2 **注意**：如果一个捕获输入的引脚由 TnCTCR 选择为计数模式，则对应该输入引脚的 TnCCR 寄存器相应的 3 位应该写 0，对于其它 3 个捕获输入可选择捕获或中断。 **注意**：定时器 2、3 中没有 CAPn.3	00
7:4	-	保留	NA

（2）PWM 控制寄存器（PWM2CON-0xE0070074，PWM3CON-0xE0074074）

PWM控制寄存器用来控制配置匹配输出为PWM输出。每个匹配输出可独立的设置为PWM输出。对于每个定时器，最多可选择 3 个单边沿PWM输出在MATn.2:0 引脚上。另外一个匹配寄存器用来决定PWM输出的周期，当其它任何一个匹配寄存器发生匹配时，PWM输出将会置为高电平。定时器可被选择作为PWM周期的匹配寄存器复位。当定时器复位为零时，所有PWM的输出将会置为低电平。PWM控制寄存器描述如表 4.60所列。

表 4.60 PWM 控制寄存器描述（定时器 2、3）

PWMCON	功能	描述	复位值
0	PWM 使能	写 1：MATn.0 使能为 PWM 模式 写 0：MATn.0 由 EM0 控制	0
1	PWM 使能	写 1：MATn.1 使能为 PWM 模式 写 0：MATn.1 由 EM0 控制	0
2	PWM 使能	写 1：MATn.2 使能为 PWM 模式 写 0：MATn.2 由 EM0 控制	0
3	PWM 使能	写 1：MATn.3 使能为 PWM 模式 写 0：MATn.3 由 EM0 控制 **注意**：推荐用 MATn.3 来设置 PWM 的输出周期，因为定时器 2 的 MAT2.3 没有连接引脚	0
32:4	-	保留	NA

单边沿控制 PWM 输出的规则如下：

- 除了在匹配值为 0 的情况之外，每个 PWM 周期的开始，所有控制的 PWM 输出为低（此时定时器设置为 0）；
- 匹配寄存器发生匹配时，相关的 PWM 输出将会置高。如果没有匹配发生（例如，匹配值大于 PWM 周期），PWM 输出将会一直输出低电平；
- 如果匹配寄存器的值大于 PWM 的输出周期时，且 PWM 的输出为高电平，则在定时器复位时，PWM 输出将会被清零。
- 如果有一个匹配寄存器的值跟 PWM 周期值一样，则在下一个时钟 PWM 周期计数时钟之后将会复位，因此，一个 PWM 将由一个时钟宽度的高电平组成，宽度由 PWM 的计数时钟决定；
- 如果一个匹配寄存器的值为 0，则第一次的 PWM 输出为高电平，同时在定时器复位以后还将一直保持高电平。

4.6.5 定时器中断

LPC2103 含有两个 16 位定时器，每个定时器可以产生多种类型的中断，可以通过读取中断标志寄存器（TnIR）来区分中断类型。定时器中断与向量中断控制器（VIC）的关系如图 4.31所示。

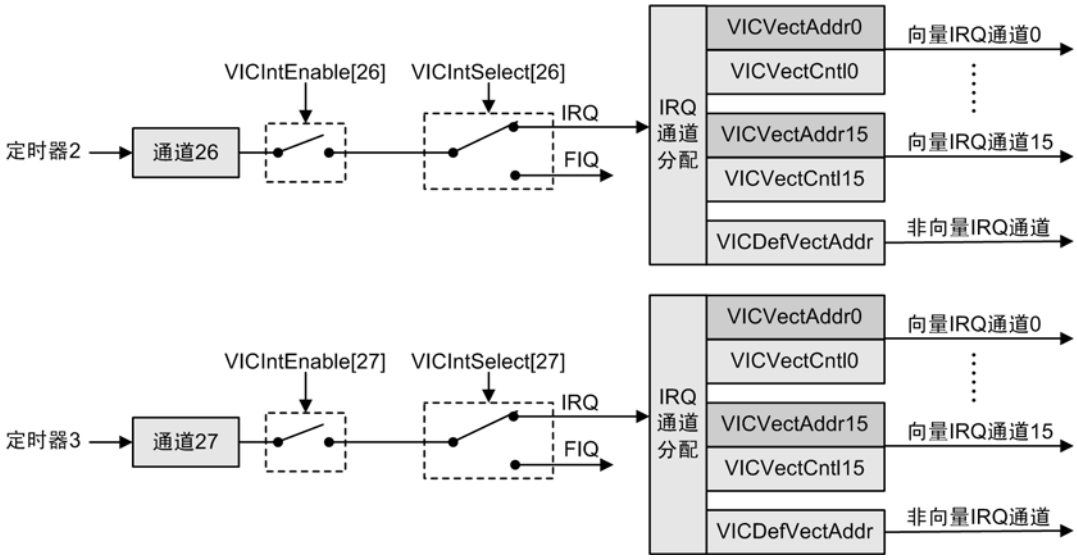


图 4.31 定时器 2、3 与 VIC 的关系

定时器 2 和定时器 3 分别处于 VIC 的通道 26 和通道 27，中断使能寄存器 VICIntEnable 用来控制 VIC 通道的中断使能。

- 当 VICIntEnable[26] = 1 时，通道 26 中断使能，即：定时器 26 中断使能；
- 当 VICIntEnable[27] = 1 时，通道 27 中断使能，即：定时器 27 中断使能。

中断选择寄存器 VICIntSelect 用来分配 VIC 通道的中断。当某一位为“1”时，对应的通道中断分配为 FIQ；当某一位为“0”时，对应的通道中断分配为 IRQ。VICIntSelect[26]和 VICIntSelect[27]分别用来控制通道 26 和通道 27，即：

- 当 VICIntSelect[26] = 1 时，定时器 2 中断分配为 FIQ 中断；
- 当 VICIntSelect[26] = 0 时，定时器 2 中断分配为 IRQ 中断；
- 当 VICIntSelect[27] = 1 时，定时器 3 中断分配为 FIQ 中断；
- 当 VICIntSelect[27] = 0 时，定时器 3 中断分配为 IRQ 中断。

分配为 IRQ 之后，还需要设置对应的通道控制寄存器和地址寄存器。有关寄存器 VICVectCntln 和 VICVectAddrn 的说明，请参考“向量中断控制器（VIC）”一节。

定时器 2、3 的匹配中断和捕获中断可参考定时器 0、1 的设置。

4.6.6 应用示例

示例应用定时器 2 捕获外部输入的下降沿信号，产生中断，控制LED闪烁。示例代码如程序清单 4.14所示。定时器 2、3 的定时、匹配等功能的示例可参考定时器 1、2。

在示例中捕获 CAP2.0 外部输入引脚是 P0.27，需要跳接到按键 KEYn 输入端，当按键 KEYn 按下时，有下降沿信号，触发捕获产生中断。

程序清单 4.14 定时器 2 捕获中断

```
#include "config.h"


#define   LED   1 << 17
/*********************************************************************************************
** 函数名称：Timer2ISR
```

```
** 函数功能：中断服务函数
** 输入参数：无
** 输出参数：无
*********************************************************************************************/
void __irq Timer2ISR (void)
{
    if ((IO0PIN & LED) == 0) {
        IO0SET = LED;                                   /* 熄灭发光二极管                   */
    }
    else    IO0CLR  = LED;
    T2IR            = 0x10;                             /* 清除中断标志                     */
    VICVectAddr     = 0x00;                             /* 中断向量结束                     */
}
/*********************************************************************************************
** 函数名称：Timer2Init
** 函数功能：定时器 2 初始化
** 输入参数：无
** 输出参数：无
*********************************************************************************************/
void   Timer2Init(void)
{
    PINSEL1   = PINSEL1 & (~(0x03<<22)) | (0x02<<22);   /* 选择 CAP2.0 功能                 */
    T2TCR     = 0x02;                                   /* 复位定时器 2                     */
    T2IR      = 0x10;                                   /* 清除中断标志                     */
    T2CCR     = 0x06;                                   /* 设置上升沿捕获，并产生中断       */
    T2TCR     = 0x01;                                   /* 启动定时器                       */
}

/*********************************************************************************************
** 函数名称：main
** 函数功能：定时器捕获中断主程序
** 输入参数：无
** 输出参数：无
*********************************************************************************************/
int main (void)
{
    PINSEL1   = PINSEL1 & 0xFFFFFFF3;                   /* 选择 P0.17 的 GPIO 功能          */
    IO0DIR    = LED;                                    /* 设置 P0.17 为输出                */
    IO0SET    = LED;                                    /* 设置输出高电平                   */

    Timer2Init();                                       /* 定时器 2 初始化                  */
    IRQEnable();                                        /*  IRQ 中断使能                    */

    /* 设置向量中断控制器           */
```

```
    VICIntSelect    = VICIntSelect & (~(1 << 26));          /* 定时器 2 分配为 IRQ 中断          */
    VICVectCntl0    = 0x20 | 26;                            /* 定时器 2 分配为向量 IRQ 通道 0    */
    VICVectAddr0    = (uint32) Timer2ISR;                   /* 分配中断服务程序地址              */
    VICIntEnable    = 1 << 26;                              /* 定时器 2 中断使能                 */

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

4.7 SPI控制器

4.7.1 特性

- 单个完整和独立的 SPI 控制器；
- 遵循串行外设接口(SPI)规范；
- 同步、串行、全双工通信；
- 组合的 SPI 主机和从机；
- 最大数据位速率为输入时钟速率的 1/8；
- 每次传输 8～16 位数据。

4.7.2 引脚描述

表 4.61 SPI 管脚描述

引脚名称	CPU 管脚	类型	描述
SCK0	P0.4	输入 输出	**串行时钟** 用于同步 SPI 接口间数据传输的时钟信号。该时钟总是由主机驱动并且从机接收。时钟可编程为高有效或低有效。它只在数据传输时才被激活，其它任何时候都处于非激活状态或三态。
SSEL0	P0.7	输入	**从机选择** SPI 从机选择信号是一个低有效信号，用于指示被选择参与数据传输的从机。每个从机都有各自特定的从机选择输入信号。在数据处理之前，SSEL 必须为低电平并在整个处理过程中保持低电平。如果在数据传输中 SSEL 信号变为高电平，传输将被中止。这种情况下，从机返回到空闲状态并将接收到的数据丢弃。对于这样的异常没有其它的指示。该信号不直接由主机驱动。可通过软件使用一个通用 I/O 口来驱动。
MISO0	P0.5	输入 输出	**主入从出** MISO 信号是一个单向的信号，它将数据从从机传输到主机。当器件为从机时，串行数据从该端口输出。当器件为主机时，串行数据从该端口输入。当从机没有被选择时，将该信号驱动为高阻态。
MOSI0	P0.6	输入 输出	**主出从入** MOSI 信号是一个单向的信号，它将数据从主机传输到从机。当器件为主机时，串行数据从该端口输出。当器件为从机时，串行数据从该端口输入。

注：如果将 SPI 设置为主机模式，那么可将 SSEL（P0.7）端单独设置为 GPIO 口使用。

4.7.3 SPI总线规范

1. SPI总线概述

SPI（Serial Peripheral Interface——串行外设接口）总线系统是一种同步串行外设接口，允许 MCU 与各种外围设备以串行方式进行通信、数据交换。外围设备包 Flash、RAM、A/D 转换器、网络控制器、MCU 等。SPI 系统可直接与各个厂家生产的多种标准外围器件直接接口，一般使用 4 条线：串行时钟线 SCK、主机输入/从机输出数据线 MISO、主机输出/从机输入数据线 MOSI 和低电平有效的从机选择线 SSEL。

如图 4.32所示为基于LPC2103 主机的SPI总线配置。一个SPI总线可以连接多个主机和多个从机，但是在同一时刻只允许有一个主机操作总线。在数据传输过程中，总线上只能有一个主机和一个从机通信。在一次数据传输中，主机总是向从机发送一个字节数据(主机通过MOSI输出数据)，而从机也总是向主机发送一个字节数据(主机通过MISO接收数据)。**SPI总线时钟是由主机产生的。**

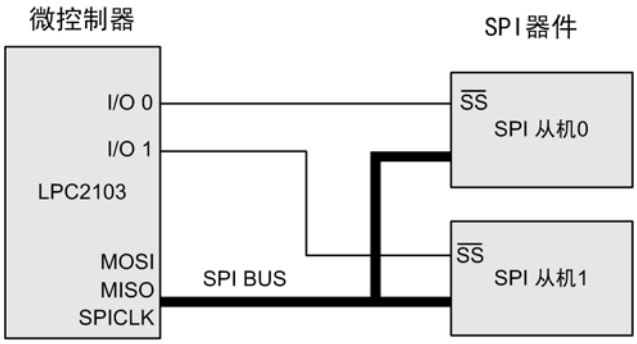


图 4.32 SPI 总线配置

2. SPI数据传输

在讨论 SPI 数据传输时，有两个位是必须要说明的：

- CPOL： 时钟极性控制位。该位决定了 SPI 总线空闲时，SCK 时钟线的电平状态。

> CPOL = 0，当 SPI 总线空闲时，SCK 时钟线为 低 电平；
> CPOL = 1，当 SPI 总线空闲时，SCK 时钟线为 高 电平。

- CPHA：时钟相位控制位。该位决定了 SPI 总线上数据的采样位置。

> CPHA = 0：SPI 总线在时钟线的第 1 个跳变沿处采样数据；
> CPHA = 1：SPI 总线在时钟线的第 2 个跳变沿处采样数据。

注：CPOL 和 CPHA 位在 SPI 控制寄存器中进行设置

图 4.33所示为SPI的 4 种不同数据传输格式的时序，该时序图描述的是 8 位数据的传输。需要注意的是，该时序图按水平方向可分成了 3 个部分，第一部分描述SCK和SSEL信号；第二部分描述了CPHA=0 时的MOSI和MISO信号；第三部分描述了CPHA＝1 时的MOSI和MISO信号。

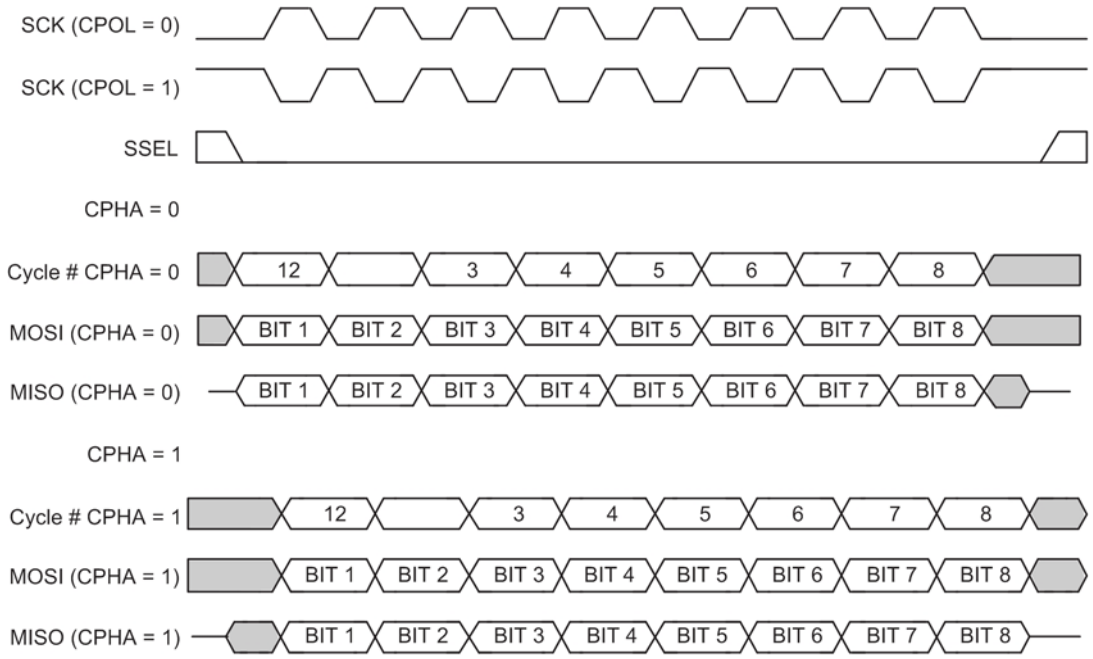


图 4.33 SPI 数据传输格式（CPHA=0 和 CPHA=1）

在时序图的第一部分需要注意以下 2 点：

- 时序图包含了 CPOL 设置为 0 和 1 的情况；
- SSEL 信号，作为 SPI 从机时用作器件的片选信号。

数据和时钟的相位关系在表 4.62中描述，该表汇集了CPOL和CPHA的每一种设定。其中“第一个数据的输出”和“其它位数据的输出”栏是表示数据在什么时刻更新输出，这是由硬件SPI接口自动操作，用户一般不需理会。**用户需要注意的是“数据的采样”这一栏，这代表数据是SCK上升沿有效，还是下降沿有效。**

表 4.62 SPI 数据和时钟的相位关系

CPOL 和 CPHA 的设定	第一位数据的输出	其它位数据的输出	数据的采样
CPOL=0, CPHA=0	在第一个 SCK 上升沿之前	SCK 下降沿	**SCK 上升沿**
CPOL=0, CPHA=1	在第一个 SCK 上升沿	SCK 上升沿	**SCK 下降沿**
CPOL=1, CPHA=0	在第一个 SCK 下降沿之前	SCK 上升沿	**SCK 下降沿**
CPOL=1, CPHA=1	在第一个 SCK 下降沿	SCK 下降沿	**SCK 上升沿**

思考：在表 4.62中，“数据的采样”栏中“SCK上升沿”和“SCK下降沿”都分别有两种设置，使用时，该如何选择？

当器件为主机时，传输的起始由主机发送数据来启动，此时，主机可激活时钟并开始传输。当传输的最后一个时钟周期结束时，传输结束。

当器件为从机并且 CPHA=0 时，传输在 SSEL 信号被激活时开始，并在 SSEL 变为高电平时结束。当器件为从机且 CPHA=1 时，如果该器件被选择，传输从第一个时钟沿开始，并在数据采样的最后一个时钟沿结束。

说明：对于 16 位的数据发送，发送完第 1 个字节数据之后，接着再发送第 2 字节数据即可，在 2 字节发送过程中，从机片选要保持有效。

3. SPI功能模块描述

有 5 个寄存器控制SPI功能模块，这里只作如下简单的描述，在第4.7.4节中将详细讲述。

- SPCR——SPI 控制寄存器。包含一些可编程位来控制 SPI 功能模块。该寄存器必须在数据传输之前进行设定；
- SPSR——SPI 状态寄存器。只读的寄存器，用于监视 SPI 功能模块的状态，包括一般性功能和异常状况。该寄存器的主要用途是检测数据传输的完成，这可以通过判断 SPIF 位来实现，其它位用于指示异常状况；
- SPDR——SPI数据寄存器。用于发送和接收数据，在发送时向SPI数据寄存器写入数据。串行数据的发送和接收通过内部移位寄存器来实现。写数据时，数据寄存器和内部移位寄存器之间没有缓冲区，写SPDR会使数据直接进入内部移位寄存器，因此数据只能在上一次数据发送完成之后写入该寄存器。读数据是带有缓冲区的，当传输结束时，接收到的数据转移到一个单字节的数据缓冲区，读SPI数据寄存器将返回读缓冲区的值。SPI数据传输方向如图 4.32所示；
- SPCCR——SPI 时钟计数器寄存器，用于设置 SPI 时钟分频值。当 SPI 功能模块处于主模式时，SPCCR 寄存器用于控制时钟速率，即 SPI 总线速率。该寄存器必须在数据传输之前设定。**当 SPI 功能模块处于从模式时，该寄存器无效；**
- SPINT——SPI 中断标志寄存器，该寄存器包含了 SPI 的中断标志位。

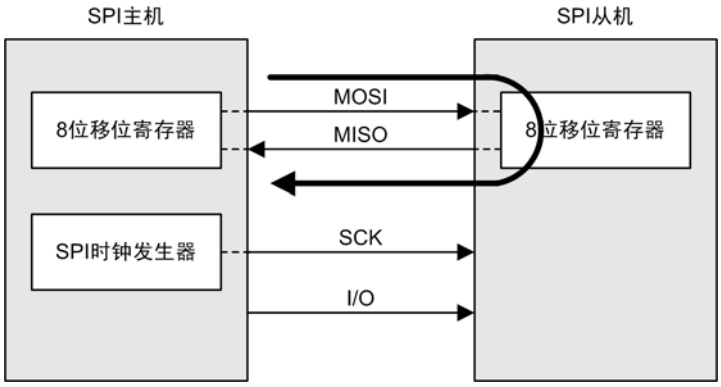


图 4.34 SPI 数据传输方向

4.7.4 寄存器描述

SPI包含 5 个寄存器，见表 4.63，所有寄存器都可以字节、半字和字的形式访问。

表 4.63 SPI 寄存器映射

名称	描述	访问	复位值	地址
S0SPCR	SPI 控制寄存器。该寄存器控制 SPI 的操作。	R/W	0x00	0xE0020000
S0SPSR	SPI 状态寄存器。该寄存器显示 SPI 的状态。	RO	0x00	0xE0020004
S0SPDR	SPI 数据寄存器。该双向寄存器为 SPI 提供发送和接收的数据。发送数据通过写该寄存器提供。SPI0 接收的数据可从该寄存器读出。	R/W	0x00	0xE0020008
S0SPCCR	SPI 时钟计数寄存器。 该寄存器控制主机 SCK0 的频率。	R/W	0x00	0xE002000C
S0SPINT	SPI 中断标志寄存器。该寄存器包含 SPI 接口的中断标志。	R/W	0x00	0xE002001C

注意：复位值仅指已使用位中保存的数据，不包括保留位的内容。

1. SPI控制寄存器（S0SPCR - 0xE0020000）

S0SPCR寄存器根据每个配置位的设定来控制SPI的操作，见表 4.64。

表 4.64 SPI 控制寄存器

S0SPCR	功能	描述	复位值
1：0	保留	保留，用户软件不要向其写入 1，从保留位读出的值未被定义	NA
2	BitEnable	SPI 控制器每次传输发送和接收 8 位数据	
3	CPHA	时钟相位控制位，决定 SPI 传输时数据和时钟的关系并控制从机传输的。**起始和结束**：(1)CPHA = 1 时，数据在 SCK 的第二个时钟沿采样。当 SSEL 信号激活时，传输从第一个时钟沿开始并在最后一个采样时钟沿结束；(2)CPHA = 0 时，数据在 SCK 的第一个时钟沿采样。传输从 SSEL 信号激活时开始，并在 SSEL 信号无效时结束	0
4	CPOL	**时钟极性控制**：(1)CPOL = 1 时，SCK 为低有效。在总线空闲状态，SCK 为高电平；(2)CPOL = 0 时，SCK 为高有效。在总线空闲状态，SCK 为低电平	0

续上表

S0SPCR	功能	描述	复位值
5	MSTR	**主模式选择**：(1)MSTR = 1 时，SPI 处于主模式；(2)MSTR = 0 时，SPI 处于从模式	0
6	LSBF	**LSBF 用来控制传输的每个字节的移动方向**：(1)LSBF = 1 时，SPI 数据传输 LSB (位 0)在先；(2)LSBF = 0 时，SPI 数据传输 MSB (位 7)在先	0
7	SPIE	**SPI 中断使能**：(1)SPIE = 1 时，每次 SPIF 或 MODF 置位时都会产生硬件中断；(2)SPIE = 0 时，SPI 中断被禁止	0
11：8	BITS	在位 2 为 1 时，该字段控制每次传输的数据的位数 1000 每次传输 8 位 1001 每次传输 9 位 1010 每次传输 10 位 1011 每次传输 11 位 1100 每次传输 12 位 1101 每次传输 13 位 1110 每次传输 14 位 1111 每次传输 15 位 0000 每次传输 16 位	0000
15：12	保留	保留，用户软件不要向其写入 1，从保留位读出的值未被定义	NA

操作示例

```
SPI_SPCR =  (0<<2) |        /* 每次传输 8 位数据                    */
            (0<<3) |        /* CPHA=0,数据在 SCK 的第一个跳变沿采样  */
            (1<<4) |        /* CPOL=1,时钟为低有效                   */
            (1<<5) |        /* 设置为主机                           */
            (0<<6) |        /* LSBF=0,数据传输 MSB 在先              */
            (1<<7);         /* SPI 中断使能                         */
```

2. SPI状态寄存器（S0SPSR - 0xE0020004）

S0SPSR寄存器根据配置位的设定来控制SPI的操作，见表 4.65。

表 4.65 SPI 状态寄存器

S0SPSR	功能	描述	复位值
2:0	保留	保留，用户软件不要向其写入 1，从保留位读出的值未被定义	NA
3	ABRT	**从机中止**：该位为 1 时表示发生了从机中止。当读取该寄存器时，该位清零	0
4	MODF	**模式错误**：为 1 时表示发生了模式错误，先通过读取该寄存器清零 MODF 位，再写 SPI 控制寄存器	0
5	ROVR	**读溢出**：为 1 时表示发生了读溢出。当读取该寄存器时，该位清零	0
6	WCOL	**写冲突**：为 1 时表示发生了写冲突。先通过读取该寄存器清零 WCOL 位，再访问 SPI 数据寄存器	0
7	SPIF	**SPI 传输完成标志**：为 1 时表示一次 SPI 数据传输完成。在主模式下，该位在传输的最后一个周期置位。在从机模式下，该位在 SCK 的最后一个数据采样边沿置位。当读取该寄存器时，该位清零。然后才能访问 SPI 数据寄存器。 注：SPIF 不是 SPI 中断标志。中断标志位于 SPINT 寄存器中	0

注：在访问 SPI 数据寄存器之前，必须要先读取 SPSR 寄存器（清除 SPIF 位）。

3. SPI数据寄存器（S0SPDR - 0xE0020008）

双向数据寄存器为SPI提供数据的发送和接收，见表 4.66。发送数据通过将数据写入该寄存器来实现，SPI接收的数据可从该寄存器中读出。处于主模式时，写该寄存器将启动SPI数据传输，由于在发送数据时，没有缓冲，所以在发送数据期间**（包括SPIF置位，但是还没有读取状态寄存器）**，不能再对该寄存器进行写操作。

表 4.66 SPI 数据寄存器

S0SPDR	功能	描述	复位值
7:0	DataLow	SPI 双向数据	0
15：8	DataHigh	当 SPCR 寄存器位 2 值为 1 时，按照设定位数传输数据，当设定少于 16 位时，高位置 0	0

操作示例

```
SPI_SPDR = data;                          /* 发送数据                          */
while((SPI_SPSR & 0x80) ==0);             /* 等待 SPIF 置位，即等待数据发送完毕  */
```

4. SPI时钟计数寄存器（S0SPCCR - 0xE002000C）

该寄存器控制主机SCK的频率，见表 4.67。该寄存器的值必须为偶数，因此bit0 必须为 0，该寄存器的值还必须大于等于 8。如果寄存器的值不符合上述条件，可能导致产生不可预测的动作。

$F_{SPI} = F_{PCLK}/SPCCR$
其中，pclk 速率为 CCLK/VPB 除数，由 VPBDIV 寄存器的内容决定。

SPI 的最大通信速率为 $F_{PCLK}/8$。如果将 SPI 设置为从机模式，则该寄存器是无效的。

表 4.67 SPI 时钟计数寄存器

S0SPCCR	功能	描述	复位值
7:0	计数值	SPI 时钟计数值设定	0

操作示例

```
SPI_SPCCR = Fpclk / F_spi;                /* 设置 SPI 时钟分频                 */
```

5. SPI中断寄存器（S0SPINT - 0xE002001C）

该寄存器包含SPI接口的中断标志，见表 4.68。

表 4.68 SPI 中断寄存器

S0SPINT	功能	描述	复位值
0	SPI 中断	**SPI 中断标志**：由 SPI 接口置位以产生中断。向该位写入 1 清零。 注：当 SPIE=1 并且 SPIF 和 MODF 位中至少有一位为 1 时该位置位。只有当 SPI 中断位置位并且 SPI 中断在 VIC 中被使能，SPI 中断才能由中断处理软件处理	0
7:1	保留	保留，用户软件不要向其写入 1，从保留位读出的值未被定义	NA

操作示例

```
    SPI_SPINT = 0x01;                                   /* 清除 SPI 中断标志位                  */
```

4.7.5 操作模式

1. 主机操作

下面的步骤描述了 SPI 设置为主机时如何处理数据传输，该处理假设上一次的数据传输已经结束。

- 设置 SPCCR 寄存器，得到相应的 SPI 时钟；
- 设置 SPCR 寄存器，控制 SPI 为主机；
- 控制片选信号，选择从机；
- 将要发送的数据写入 SPDR 寄存器，即启动 SPI 数据传输；
- 读取 SPSR 寄存器，等待 SPIF 位置位。SPIF 位将在数据传输的最后一个周期之后由硬件自动置位；
- 从 SPI 数据寄存器中读出接收到的数据（可选）；
- 如果有更多数据需要发送，则跳到第（3）步，否则取消对从机的选择。

主机初始化示例程序见程序清单 4.15，程序首先判断需要设置的SPI时钟分频值是否合法，如果设置值小于 8 则强行设置为 8。由于SPCCR寄存器的值必须为偶数，所以将分频值和 0xFE进行“与”操作。

程序清单 4.15 SPI 主机初始化示例

```
#define   MSTR              (1<<5)
#define   CPOL              (1<<4)
#define   CPHA              (1<<3)
#define   LSBF              (1<<6)
#define   SPI_MODE          (MSTR | CPOL)   /* SPI 接口模式，MSTR=1，CPOL=1，CPHA=0，LSBF=0 */
/*********************************************************************************************
** 函数名称：MSpiIni
** 函数功能：初始化 SPI 接口，设置为主机
** 入口参数：fdiv    SPI 时钟分频值，大于 8 的偶数
** 出口参数：无
*********************************************************************************************/
void   MSpiIni(uint8 fdiv)
{
    if(fdiv < 8) fdiv = 8;
    SPI_SPCCR = fdiv & 0xFE;                            /* 设置 SPI 时钟分频                    */
    SPI_SPCR = SPI_MODE;
}
```

SPI主机数据发送和接收示例程序见程序清单 4.16，向S0PDR寄存器(即SPI0 的SPDR)写入一字节数据后，即可启动数据发送。SPI功能模块在发送数据时同时接收一字节数据，并将接收到的数据返回。

程序清单 4.16 SPI 主机数据发送和接收程序

```
/*********************************************************************************************
** 函数名称：MSendData
```

```
** 函数功能：向 SPI 总线发送数据，并接收从机发回的数据
** 入口参数：data  待发送的数据
** 出口参数：返回值为接收到的数据
*********************************************************************************************/
uint8  MSendData (uint8 data)
{
    IO0CLR = HC595_CS;                          /* 片选                                  */
    SPI_SPDR = data;
    while((SPI_SPSR&0x80) == 0 );               /* 等待 SPIF 置位，即等待数据发送完毕  */
    IO0SET = HC595_CS;
    return(SPI_SPDR);
}
```

2. 从机操作

下面的步骤描述了 SPI 设置为从机时如何处理数据传输。该处理假设上一次的数据传输已经结束。要求驱动 SPI 逻辑的系统时钟速度至少 8 倍于 SPI。

- 设置 SPCR 寄存器，控制 SPI 为从机；
- 将要发送的数据写入 SPI 数据寄存器（可选）。注意，这只能在 SPI 总线空闲（即主机还没有启动 SPI 传输）时执行；
- 读取 SPSR 寄存器，等待 SPIF 位置位。SPIF 位将在 SPI 数据传输的最后一个采样时钟沿后由硬件自动置位；
- 从 SPI 数据寄存器中读出接收到的数据（可选）；
- 如果有更多数据需要发送，则跳到第（2）步。

从机初始化示例程序见程序清单 4.17，程序只对S0PCR进行设置，控制SPI为从机。为了能够上SPI主机进行通讯，需要正确设置CPOL、CPHA、LSBF控制位。

注意：SPI 时钟脉冲是由主机产生，所以从机无需初始化 S0PCCR 寄存器。

程序清单 4.17 SPI 从机初始化示例

```
#define  CPOL            (1<<4)
#define  CPHA            (1<<3)
#define  LSBF            (1<<6)
#define  SPI_MODE        (CPOL)          /* SPI 接口模式，MSTR=0，CPOL=1，CPHA=0，LSBF=0 */
/*********************************************************************************************
** 函数名称：SSpiIni
** 函数功能：初始化 SPI 接口，设置为从机
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
void  SSpiIni(void)
{
    SPI_SPCR = SPI_MODE;
}
```

SPI从机数据发送示例程序见程序清单 4.18，向S0PDR寄存器(即SPI0 的SPDR)写入一字节数据，然后等待主机读数据操作。当然，用户可以使用中断形式进行数据的发送，这样有

助于提高整个系统程序的效率。

程序清单 4.18　SPI 从机数据发送程序

```
/*********************************************************************************************
** 函数名称：SSendData
** 函数功能：SPI 从机发送数据
** 入口参数：data　待发送的数据
** 出口参数：无
*********************************************************************************************/
void　SSendData(uint8 data)
{
    SPI_SPDR = data;
    while(( SPI_SPSR & 0x80) == 0 );                    /* 等待 SPIF 置位，即等待数据发送完毕　*/
}
```

SPI从机数据接收示例程序见程序清单 4.19，程序首先等待S0PSR寄存器的SPIF位为 1，然后读取S0PDR寄存器内的数据（即是接收到的数据）。当然，用户可以使用中断形式进行数据的接收，这样有助于提高整个系统程序的效率。

程序清单 4.19　SPI 从机数据接收程序

```
/*********************************************************************************************
** 函数名称：SRcvData
** 函数功能：SPI 从机接收数据
** 入口参数：无
** 出口参数：返回值为读取到的数据
*********************************************************************************************/
uint8　SRcvData(void)
{
    while(( SPI_SPSR&0x80) == 0 ) ;
    return(SPI_SPDR);
}
```

3.　异常状况

读溢出——当 SPI 功能模块内部读缓冲区满时，又接收到新的数据，就会发生读溢出。SPI 模块内部的读缓冲区大小为 1 个字节，SPIF = 1 表示读缓冲区满。当一次传输结束时，SPI 功能模块将接收到的数据保存到读缓冲区中。如果 SPIF 置位（读缓冲区已满），新接收到的数据将会丢失，而状态寄存器的读溢出（ROVR）位将置位。

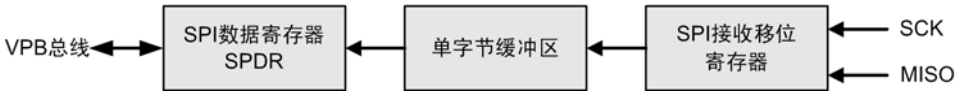


图 4.35　SPI 主机模式接收示意图

写冲突——我们在前面提到过，在 SPI 总线接口与内部移位寄存器之间没有写缓冲区。这就要求，只能在 SPI 总线空闲期间向 SPI 数据寄存器写入数据。从启动传输到 SPIF 置位（包括读取状态寄存器），在此期间不能向 SPI 数据寄存器写入数据。如果在这段时间内写

SPI 数据寄存器，写入的数据将会丢失，状态寄存器中的写冲突位（WCOL）置位，写冲突错误不会产生中断。

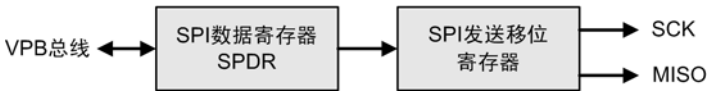


图 4.36 SPI 主机模式发送示意图

模式错误——SSEL 信号在 SPI 功能模块为主机时不需要配置，可以用作 GPIO。当 SPI 功能模块为主机时，如果 SSEL 信号配置有效，表示有另外一个主机将该器件选择为从机。这种状态称为模式错误。当检测到一个模式错误时，状态寄存器的模式错误位（MODF）位置位，SPI 时钟信号驱动器关闭，而 SPI 模式转换为从模式，如果中断使能，将触发中断。如果要清除模式错误位（MODF），必须要先读取 SPI 状态寄存器，然后再重新初始化 SPI 控制寄存器。

从机中止——在从模式下，如果 SSEL 信号在传输结束之前变为高电平，从模式传输将被认为中止。此时，正在处理的发送或接收数据都将丢失，状态寄存器的从机中止（ABRT）位置位。

4.7.6 SPI接口中断

LPC2103 SPI接口具有中断功能，当传输完成或者发生模式错误时，SPI接口就会触发中断，SPI接口中断与向量中断控制器（VIC）的关系如图 4.37所示。

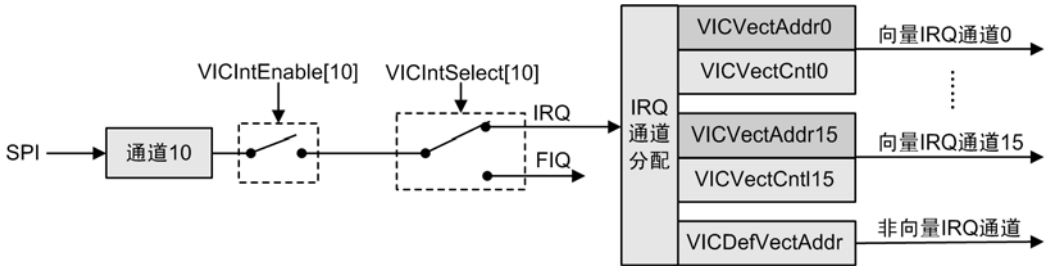


图 4.37 SPI 接口与 VIC 的关系

SPI 接口处于 VIC 的通道 10，中断使能寄存器 VICIntEnable 用来控制 VIC 通道的中断使能。 当 VICIntEnable[10] = 1 时，通道 10 中断使能，即：SPI 中断使能。

中断选择寄存器 VICIntSelect 用来分配 VIC 通道的中断。当某一位为“1”时，对应的通道中断分配为 FIQ；当某一位为“0”时，对应的通道中断分配为 IRQ。VICIntSelect[10] 用来控制通道 10，即：

- 当 VICIntSelect[10] = 1 时，SPI 中断分配为 FIQ 中断；
- 当 VICIntSelect[10] = 0 时，SPI 中断分配为 IRQ 中断。

当分配为 IRQ 时，还需要设置对应的通道控制寄存器和地址寄存器。有关寄存器 VICVectCntln 和 VICVectAddrn 的说明，请参考“向量中断控制器（VIC）”一节。

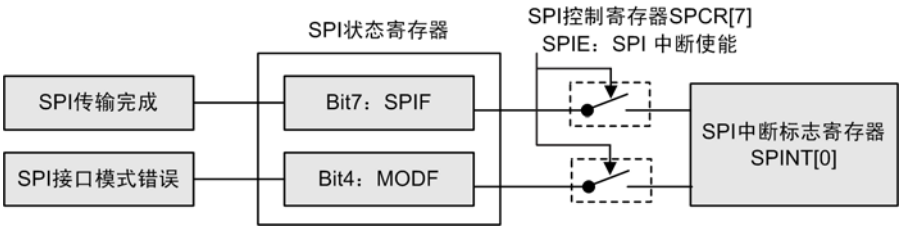


图 4.38 SPI 中断示意图

由图 4.38所示为SPI接口的中断示意图，SPI中断使能位仅由“SPIE”位控制，当SPIE = 1 时，只要传输完成或者发生模式错误，就会触发中断，置位SPI中断标志寄存器SPINT的Bit0。对SPI中断标志寄存器Bit0 执行写“1”操作，可以清除SPI中断标志。

4.7.7 应用示例

EasyARM2103 开发板支持 SPI 与 SSP 模块进行主从机通信，通过串口将总线传输的数据发送到上位机进行显示。

EasyARM2103 开发板配套提供串口调试软件“EasyARM-C.exe”，本示例应用此软件的模拟数码管发光功能，将SPI与SSP通信的数据发送到上位机，“驱动”数码管进行显示。示例应用如程序清单 4.21所示。

注意：**当 SPI 作为主机时，不需要初始化 SSEL0，可以将此引脚作为 GPIO。**串口显示需要调用软件包“UART.h”、“UART.c”。

程序清单 4.20 SPI 与 SSP 主从机通信

```
# include "config.h"
# include "UART.h"                                          /* 用户需要添加串口软件包          */

# define SLAVE_CS          1 << 13                          /* P0.13 口作为 SSP 从机的片选     */
uint8 const uiBuf[8] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07};
/*********************************************************************************************
** 函数名称：DelayNS
** 功能描述：延时函数
** 入口参数：uiDly    值越大，延时时间越长
** 出口参数：无
*********************************************************************************************/
void DelayNS (uint32  uiDly)
{
    uint32  i;
    for(; uiDly > 0; uiDly--){
        for(i = 0; i < 50000; i++);
    }
}
/*********************************************************************************************
** 函数名称：MSPIInit
** 功能描述：初始化 SPI 接口，设置为主机
** 入口参数：无
```

```
** 出口参数：无
*********************************************************************************************/
void MSPIInit(void)
{
    PINSEL0 = (PINSEL0 & 0xCFFF00FF) | 0x20001500;      /*  设置 SPI 引脚连接              */
    SPI_SPCCR    = 0x52;                                /*  设置 SPI 时钟分频              */
    SPI_SPCR     = 0 << 3 |                             /*  CPHA = 0 第一个时钟采样        */
                   1 << 4 |                             /*  CPOL = 1，SCK 低有效           */
                   1 << 5 |                             /*  MSTR = 1，设置为主模式         */
                   0 << 6 |                             /*  LSBF = 0，SPI 传输 MSB 在先    */
                   0 << 7 ;                             /*  SPIE = 0，SPI 中断禁止         */
}
/*********************************************************************************************
** 函数名称：SSPIInit
** 功能描述：初始化 SSP 接口，设置为从机
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
void SSPIInit(void)
{
    PINSEL1 = (PINSEL1 & 0xFFFFF00F) | 0x00000540;      /*  设置 SSP 引脚连接              */

    SSPCR0   =   0x07 << 0 |                            /*  DSS  数据长度为 8 位           */
                 0x00 << 4 |                            /*  FRF  格式:SPI                  */
                 0x01 << 6 |                            /*  CPOL = 1，SCK 低有效           */
                 0x00 << 7 |                            /*  CPHA = 0, 第一个时钟采样       */
                 0x05 << 8 ;                            /*  SCR  设置 SPI 时钟分频         */

    SSPCR1    =  0x00 << 0 |                            /*  LBM  回写模式                  */
                 0x01 << 1 |                            /*  SSE SSP 使能                   */
                 0x01 << 2 |                            /*  MSTR = 1，设置为从机           */
                 0x00 << 3 ;                            /*  SOD   = 0, 从机输出允许        */

    SSPCPSR  = 0x52;                                    /*  设置 SSP 时钟分频              */
    SSPIMSC  = 0x02;                                    /*  使能接收超时中断               */
    SSPICR   = 0x03;                                    /*  清除中断标志                   */
}
/*********************************************************************************************
** 函数名称：MSPISendData
** 功能描述：SPI 主模式发送数据
** 入口参数：uiData：     将要发送的数据
** 出口参数：S0SPDR：  返回值为读取的数据无
*********************************************************************************************/
uint8 MSPISendData(uint8 uiData)
```

```
{
    IOCLR       = SLAVE_CS;                          /*  片选 SSP                    */
    SPI_SPDR = uiData;
    while ((SPI_SPSR & 0x80) == 0);                  /*  等待数据发送完毕            */
    IOSET       = SLAVE_CS;
    return(SPI_SPDR);
}
/*********************************************************************************************
** 函数名称：MSPIRcvByte
** 功能描述：从 SPI 总线接收 1 字节数据
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
uint8 MSPIRcvByte (void)
{
    while ((SPI_SPSR & 0x80) == 0);                  /*  等待数据接收完成            */
    return (SPI_SPDR);
}
/*********************************************************************************************
** 函数名称：SSPSendByte
** 功能描述：SSP 接口向总线发送 1 字节数据
** 入口参数：uiDat  待发送的数据
** 出口参数：接收到的数据
*********************************************************************************************/
uint8 SSPSendByte(uint8    uiDat)
{
    IOCLR     = SLAVE_CS;
    SSPDR     = uiDat;
    while((SSPSR & 0x01) == 0);                      /* 等待 TFE 置位，即发送 FIFO 空   */
    IOSET     = SLAVE_CS;
    return(SSPDR);
}
/*********************************************************************************************
** 函数名称：SSPRcvByte
** 功能描述：SSP 接口从总线接收 1 字节数据
** 入口参数：无
** 出口参数：接收到的数据
*********************************************************************************************/
uint8 SSPRcvByte(void)
{
    while((SSPSR & 0x04) == 0);                      /* 等待 RFF 置位，即接收 FIFO 不为空*/
    return(SSPDR);
}
/*********************************************************************************************
```

```
** 函数名称：main
** 功能描述：SPI 做主机、SSP 做从机进行数据通信，并将 SPI 发送的数据通过串口发送到 PC 机显示
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
int main(void)
{
    uint8 i;
    uint8 j;
    uint8 uiRcvData;

    PINSEL0 = PINSEL0 & (~(0x03 << 26));                /*  设置 GPIO 引脚连接              */

    IO0DIR  = SLAVE_CS;                                 /*  设置 SSP 片选为输出             */
    IOSET   = SLAVE_CS;

    UARTInit();
    MSPIInit();                                         /*  初始化 SPI 接口                 */
    SSPIInit();                                         /*  初始化 SSP 接口                 */

    for(;;){
        for ( i = 0; i < 8; i++){                       /*  SPI 发送数据                    */
            MSPISendData(uiBuf[i]);
            DelayNS(10);
            uiRcvData = SSPRcvByte();                   /*  SSP 接收数据                    */
            DelayNS(10);

            for ( j = 0; j < 8; j++){
                PCDispChar (j, uiRcvData);              /*  将 SSP 接收到的数据发送到 PC 机*/
            }
            DelayNS(50);
        }
    }
    return 0;
}
```

4.8 SSP控制器

4.8.1 概述

SSP 是一个同步串行端口控制器，可控制 SPI、4 线 SSI 或 Microwire 总线的操作。它可处理总线上多个主机和从机的相互作用。在数据传输过程中，总线上只能有一个主机和一个从机进行通信。数据传输原则上是全双工的，4～16 位帧的数据由主机发送到从机或由从机发送到主机。但实际上，大多数情况下只有一个方向上的数据流包含有意义的数据。

4.8.2 特性

- 兼容 SPI、4 线 TI SSI 和 Microwire 总线；
- 同步串行通信；
- 主机或从机操作；
- 8 帧收发 FIFO；
- 每帧 4～16 位。

4.8.3 引脚描述

LPC2103 芯片SSP引脚介绍如表 4.69所列。

表 4.69 LPC2103 SSP 接口引脚介绍

接口引脚名称/功能			CPU	类	引脚描述
SPI	SSI	Microwire	引脚	型	
SCK	CLK	SK	P0.14	I/O	串行时钟。SCK/CLK/SK 是用来同步数据传输的时钟信号。它由主机驱动，从机接收。当使用 SPI 接口时，时钟可编程为高有效或低有效，否则，时钟总是高有效。SCK1 的状态只能在数据传输过程中改变，在其它时间里，SSP 使其保持无效状态或不驱动它（使其处于高阻态）
SSEL	FS	CS	P0.21	I/O	帧同步/从机选择。当 SSP 是总线主机时，它在串行数据启动前，串行数据传输终止后都会立刻驱动该信号，酌情指示所选总线和模式的数据传输状况。当 SSP 是总线从机时，该信号根据使用的通信协议来指示总线上的数据有效状况。 当只有一个总线主机和一个总线从机时，主机的帧同步或从机选择信号直接与从机相应的输入相连。当总线上接有多个从机时，需要管理好这些从机的帧选择/从机选择输入，以免一次传输有多个从机响应
MISO	DR(M) DX(S)	SI(M) SO(S)	P0.19	I/O	主机输入从机输出。MISO 信号使串行数据从从机传输到主机。当 SSP 是从机时，串行数据从该信号输出。当 SSP 是主机时，该引脚用于接收从机的数据输入。当 SSP 是从机且未被 FS/SSEL 选择时，它将不驱动该信号（使其处于高阻态）
MOSI	DX(M) DR(S)	SO(M) SI(S)	P0.20	I/O	主机输出从机输入。MOSI 信号使串行数据从主机传输到从机。当 SSP 是主机时，串行数据从该信号输出。当 SSP 是从机时，该引脚接收主机的数据输入

4.8.4 总线描述

1. SPI帧格式

SPI 接口是一个 4 线接口，它的 SSEL 信号用作从机选择。SPI 格式的主要特性是 SCK 信号的无效状态和相位可通过 SSPCR0 控制寄存器的 CPOL 位和 CPHA 位来编程设定。

- CPOL = 0，则 SPI 总线空闲时，使 SCK 引脚产生一个稳定的低电平；
- CPOL = 1，则 SPI 总线空闲时，使 SCK 引脚产生一个稳定的高电平。

CPHA 控制位用来选择捕获数据的时钟沿模式。它将对通讯时传输的第一个位产生重要影响。当 CPHA 相位控制位为 0 时，数据在第 1 个时钟沿（SSEL 引脚由无效变为有效以后的第一个 SCK 跳变）被捕获。如果 CPHA 时钟相位控制位为 1，那么数据在第 2 个时钟沿被捕获。

（1）CPOL=0, CPHA=0 时的 SPI 帧格式

CPOL=0，CPHA=0 时SPI格式的单帧传输和连续帧传输信号时序如图 4.39所示。

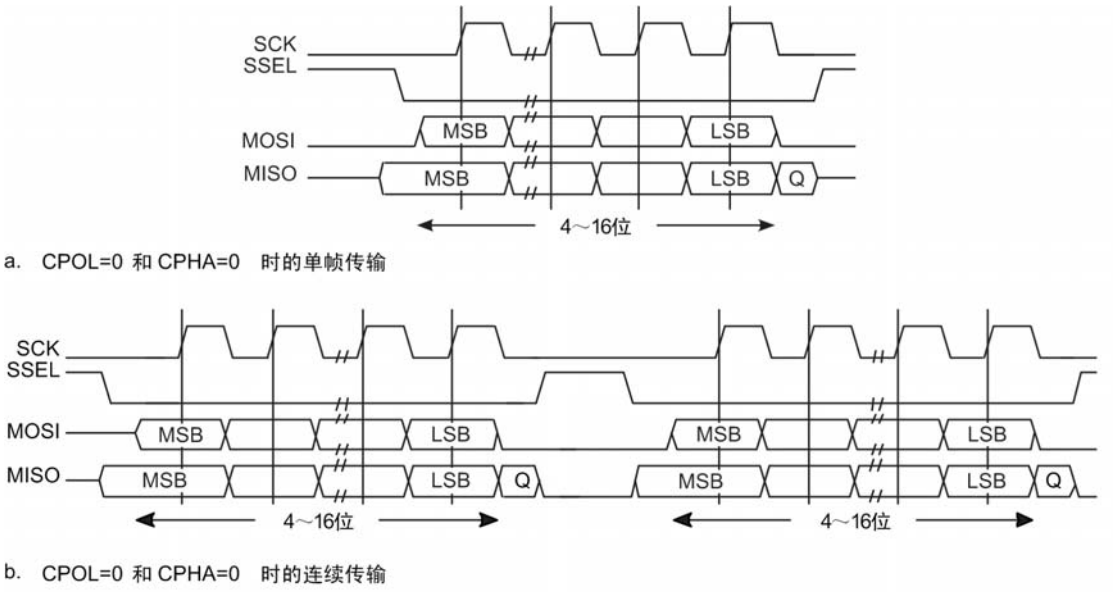


图 4.39 CPOL＝0 和 CPHA＝0 时的 SPI 帧格式 （单帧和连续传输）

这种配置下，总线在空闲期间：

- SCK 信号强制为低；
- SSEL 强制为高；
- MOSI/MISO 引脚处于高阻态。

如果 SSP 使能并且发送 FIFO 中装载了有效数据，则 SSEL 信号驱动为低，指示数据发送开始。这就使得主机的 MOSI 引脚被使能，从机的数据也出现在 MISO 引脚上。1/2 个 SCK 周期后，主机发送的数据传输到 MOSI 引脚，从机发送的数据传输到 MISO 引脚。由于主机和从机数据都被设置，再过 1/2 个 SCK 周期，SCK 引脚电平变为高。此时，数据在 SCK 信号的上升沿被捕获，保持到 SCK 的下降沿。

发送单个帧时，当数据帧的所有位发送完，最后一个数据位被捕获的一个 SCK 周期后，SSEL 返回至高电平状态。但是，在连续帧的发送过程中，每个数据帧传输之间 SSEL 信号必须为高。这是因为当 CPHA 位为 0 时，从机选择引脚冻结了串行外围寄存器中的数据，不允许改变。因此，在每次数据帧传输之间主器件必须拉高从器件的 SSEL 引脚来使能串行外设数据的写操作。当连续帧传输结束，最后一位被捕获一个 SCK 周期后，SSEL 返回到空闲状态。

（2）CPOL=0，CPHA=1 时的 SPI 帧格式

CPOL=0，CPHA=1 时SPI格式的传输信号时序如图 4.40所示，它包括单帧传输和连续传输两种方式。

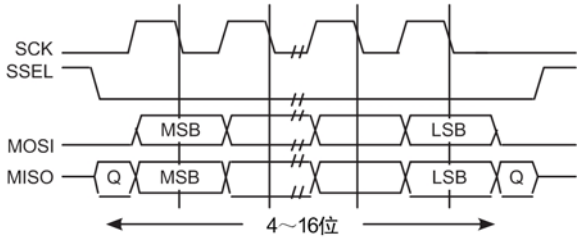


图 4.40 CPOL＝0 和 CPHA=1 时的 SPI 帧格式

这种配置在空闲期间：

- CLK 信号强制为低；
- SSEL 强制为高；
- 发送 MOSI/MISO 引脚处于高阻态。

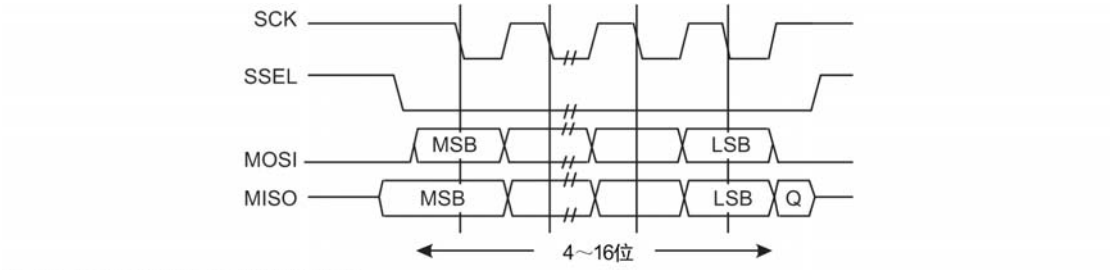
如果 SSP 使能并且发送 FIFO 中含有效数据，则 SSEL 被驱动为低表示开始发送数据。主机的 MOSI 引脚使能。再过 1/2 个 SCK 周期，主机和从机的有效数据都被使能输出到各自的发送线上。同时，SCK 出现上升沿跳变。然后，数据在 SCK 信号的下降沿被捕获并保持到 SCK 信号的上升沿。

在单个字的传输过程中，当所有位传输结束后，最后一位被捕获后一个 SCK 周期内，SSEL 引脚返回到空闲的高电平状态。

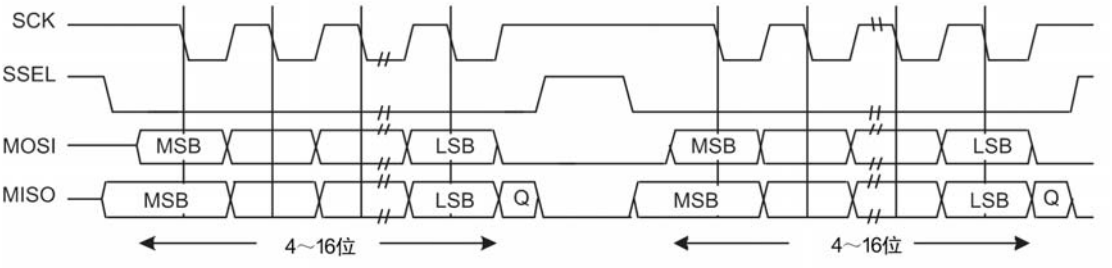
对于连续帧的顺序传输，SSEL 在两个连续的数据字传输之间保持低电平，终止传输的方法与单个字传输时相同。

（3）CPOL=1, CPHA=0 时的 SPI 帧格式

CPOL=1，CPHA=0 时SPI格式的单帧和连续传输信号时序如图 4.41所示。



a. CPOL=1和CPHA=0时的单帧传输



b. CPOL=1和CPHA=0时的连续传输

图 4.41 CPOL=1 和 CPHA=0 时的 SPI 帧格式（a）单帧和(b)连续传输

这种配置在空闲期间：

- CLK 信号强制为高；
- SSEL 强制为高；
- 发送 MOSI/MISO 引脚处于高阻态。

如果 SSP 使能并且发送 FIFO 中含有效数据，则 SSEL 被驱动为低表示开始发送数据，这使得从机数据立即传输到主机的 MISO 线上。主机的 MOSI 引脚被使能。1/2 个 SCK 周期后，有效的主机数据被传输到 MOSI 线。由于主机和从机数据都被设置，再过 1/2 个 SCK 周期后 SCK 引脚将变低。数据在 SCK 信号的下降沿被捕获，并保持到 SCK 的上升沿。

在发送单个字时，当数据字的所有位发送完，最后一位被捕获的一个 SCK 周期后，SSEL 引脚返回到高电平状态。但是，在连续帧的发送过程中，在每个数据字传输之间 SSEL 信号必须为高。这是因为当 CPHA 位为逻辑 0 时，从机选择引脚冻结了串行外围寄存器中的数据，不允许改变。因此，在每次数据传输之间主器件必须拉高从器件的 SSEL 引脚来使能对串行外设数据的写操作。当连续传输结束，最后一位被捕获的一个 SCK 周期后，SSEL 引脚返回到空闲状态。

（4）CPOL=1，CPHA=1 时的 SPI 帧格式

CPOL=1，CPHA=1 时SPI格式的传输信号时序如图 4.42所示，它包含单帧传输和连续传输两种方式。

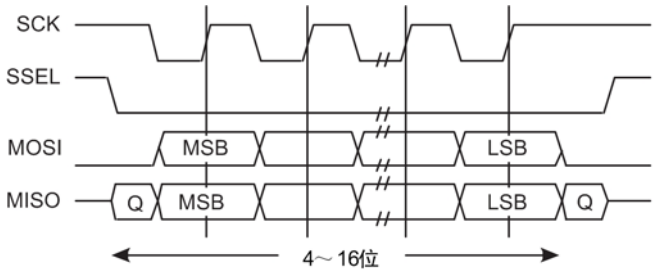


图 4.42 CPOL=1 和 CPHA=1 时的 SPI 帧格式

这种配置在空闲期间：

- CLK 信号强制为高；
- SSEL 强制为高；
- 发送 MOSI/MISO 引脚处于高阻态。

如果 SSP 使能并且发送 FIFO 中含有效数据，则 SSEL 主机信号被驱动为低，指示数据发送开始。主机的 MOSI 引脚使能。再过 1/2 个 SCK 周期，主机和从机的有效数据都被使能输出到各自的发送线上。同时，SCK 出现上升沿跳变。然后，数据在 SCK 信号的上升沿被捕获并保持到 SCK 信号的下降沿。

在单个字的传输过程中，当所有位传输结束，最后一位被捕获的一个 SCK 周期后，SSEL 返回到高电平状态。对于连续帧的顺序传输，SSEL 引脚仍保持有效的低电平状态，直到最后一个字的最后一位捕获，它再返回到空闲状态。总的说来，SSEL 引脚在两个连续的数据字传输之间保持低电平，终止传输的方法与单个字传输相同。

2. Texas仪器同步串行（SSI）数据帧格式

图 4.43给出了SSP模块支持的 4 线Texas仪器同步串行数据帧（SSI）的格式。

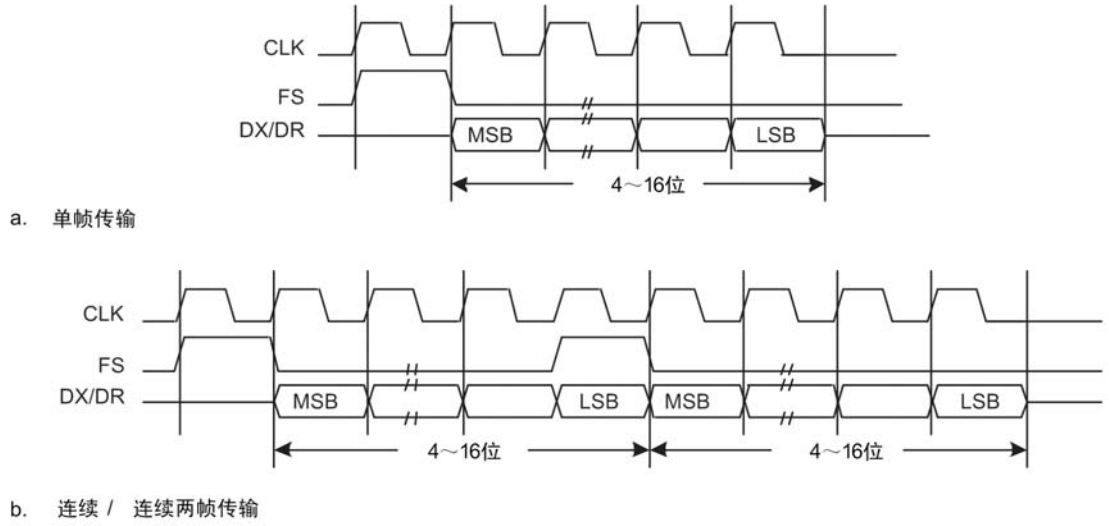


图 4.43 Texas 仪器同步串行数据帧（SSI）的格式：

时序分析：

假设器件在 SSI 模式下配置为主机，将 SSI 通讯时序介绍如下：

- SSI 主机空闲

当 SSI 主机空闲时，CLK 和 FS 强制为低，发送数据线 DX 为三态。

- 帧传输启动

一旦发送 FIFO 的底部装入数据，FS 立即变为高电平，并维持一个 CLK 周期，指示通讯启动。此时，需发送的数据位从发送 FIFO 转移到串行移位寄存器。

- 数据位的发送

在 CLK 上升沿，帧的一个数据位从主机的串行移位寄存器输出到 DX 引脚。此时，该数据位也移到了 DR 脚。

- 数据位的接收

在 CLK 下降沿，SSI 主机和 SSI 从机将每一数据位送入相应的串行移位寄存器。对于 SSI 从机而言，LSB 被锁存后，接收到的数据在 CLK 的首个上升沿从串行移位寄存器传递到接收 FIFO。

3. 半导体Microwire帧格式

图 4.44所示为Microwire帧格式的单帧传输。图 4.45所示为Microwire帧格式的连续帧传输。

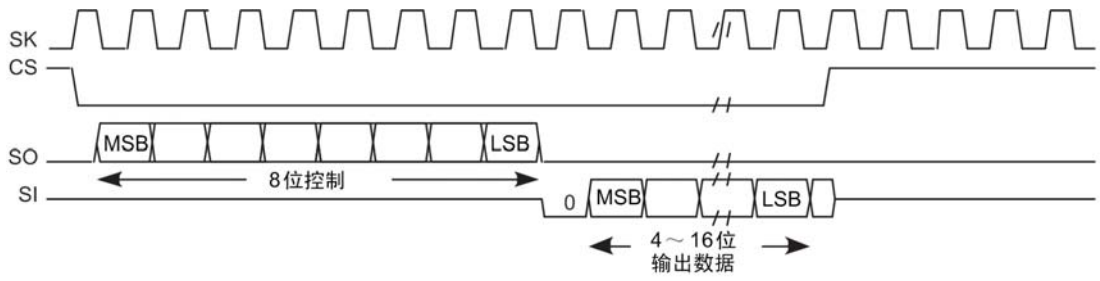


图 4.44 Microwire 帧格式（单帧传输）

Microwire 格式与 SPI 格式类似，但它的发送是半双工而非全双工模式，数据从主机传输到从机。每次串行发送以 SSP 传输到片外从器件的一个 8 位控制字开始。在发送控制字的过程中，SSP 不接收数据。控制字发送结束后，片外从器件对其进行译码，在 8 位控制信

息的最后一位发送结束的一个串行时钟后，才返回主机所需的数据。返回的数据长度为 4～16 位，使得总的数据帧长度在 13～25 位之间。

这种配置在空闲期间：

- SK 信号强制为低；
- CS 强制为高；
- 发送数据线 SO 可强制为低。

发送过程由写入到发送 FIFO 的一个控制字节来触发。CS 的下降沿使发送 FIFO 底端的数据（即 8 位控制帧）传输到串行移位寄存器，8 位控制帧的 MSB 移到 SO 引脚。CS 在帧发送过程中保持低电平。SI 在帧发送过程中保持三态。片外串行从器件在每个 SK 的上升沿将每个控制位锁存到串行移位器。当从器件完成最后一位的锁存后，再用一个 SK 周期对控制字节进行译码，然后将所需数据发回给 SSP。

每一个数据位在 SK 的下降沿驱动到 SI 上，主机则在 SK 的上升沿将其锁存。帧传输结束后，对于单帧传输，在最后一位被锁存到接收串行移位器的一个 SK 周期后，CS 信号被拉高，使数据传输到接收 FIFO。

注意：在 LSB 被接收移位器锁存后，或当 CS 变为高电平时，片外从器件的接收线在任何一个 SK 下降沿时刻都呈现三态。

连续传输过程的数据发送开始和结束的方法都与单帧传输相同。所不同的是，在连续传输过程中，CS 持续有效（保持低电平），数据连续发送。当前数据帧的 LSB 被接收后，下一帧的控制字节立刻直接发送。在一帧数据的 LSB 被锁存后，每个接收到的数据在 SK 的下降沿传送到接收移位器。

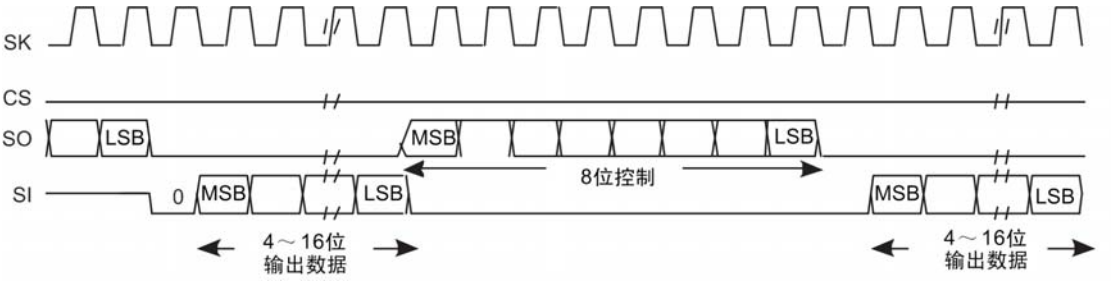


图 4.45 Microwire 帧格式（连续传输）

Microwire 模式中，当 CS 变低后，SSP 从机在 SK 上升沿对接收数据的首位进行采样。因此，主机驱动 SK 自由运行时必须确保 CS 信号相对 SK 的上升沿有足够的建立和保持时间。

图4.46给出了对CS的建立和保持时间的要求。相对SSP从机采样接收数据的首个位的SK上升沿，CS的建立时间至少为SSP的SK周期的 2 倍。相对于之前的SK上升沿，CS的保持时间至少为一个SK周期。

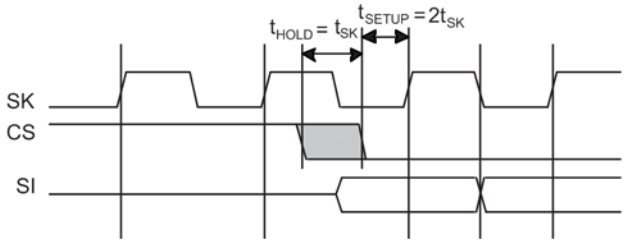


图 4.46 Microwire 帧格式的建立和保持时间

4.8.5 寄存器描述

SSP包括表 4.70中给出的 9 个寄存器。

表 4.70 SSP 寄存器映射

名称	描述	访问	复位值	地址
SSPCR0	控制寄存器 0。选择串行时钟速率、总线类型和数据长度	R/W	0x0000	0xE006 8000
SSPCR1	控制寄存器 1。选择主机/从机和其它模式	R/W	0x00	0xE006 8004
SSPDR	数据寄存器。写满发送 FIFO 和读空接收 FIFO	R/W	0x0000	0xE006 8008
SSPSR	状态寄存器	RO	0x03	0xE006 800C
SSPCPSR	时钟预分频寄存器	R/W	0x00	0xE006 8010
SSPIMSC	中断屏蔽设置和清零寄存器	R/W	0x00	0xE006 8014
SSPRIS	原始中断状态寄存器	R/W	0x04	0xE006 8018
SSPMIS	屏蔽中断状态寄存器	RO	0x00	0xE006 801C
SSPICR	SSPICR 中断清零寄存器	WO	NA	0xE006 8020

注意：复位值仅指已使用位中保存的数据，不包括保留位的内容。

1. SSP控制寄存器(SSPCR0 - 0xE006 80000)

该寄存器控制着SSP控制器的基本操作。该寄存器的位功能描述如表 4.71所列。

表 4.71 SSP 控制寄存器

位	符号	值	描述	复位值
3：0	DSS		数据长度选择。该字段控制着每帧传输的位数目。不支持且不使用值 0000-0010。	0000
		0011	4 位传输	
		0100	5 位传输	
		0101	6 位传输	
		0110	7 位传输	
		0111	8 位传输	
		1000	9 位传输	
		1001	10 位传输	
		1010	11 位传输	
		1011	12 位传输	
		1100	13 位传输	
		1101	14 位传输	
		1110	15 位传输	
		1111	16 位传输	
5：4	FRF		帧格式。	00
		00	SPI	
		01	SSI	
		10	Microwire	
		11	不支持且不应使用这个组合	
6	SPO		时钟输出极性。该位只用在 SPI 模式	0
		0	SSP 控制器使总线时钟在每帧传输之间保持低电平	
		1	SSP 控制器使总线时钟在每帧传输之间保持高电平	

续上表

位	符号	值	描述	复位值
7	SPH	0 1	时钟输出相位。该位只用在 SPI 模式。 SSP 控制器在帧传输的第一个时钟跳变沿捕获串行数据 SSP 控制器在帧传输的第二个时钟跳变沿捕获串行数据	0
15：8	SCR		控制 SSP 通讯的位速率。SCR 之值为总线上传输的每一个数据位对应的 SSP 时钟数减 1。假设 CPSDVR 为预分频器分频值，P_{CLK} 为 SSP 时钟预分频器的输入时钟，则位速率为 P_{CLK} / (CPSDVSR×[SCR + 1])	0x00

操作示例：

```
SSPCR0  =   (0x07 << 0) |          /* 每帧数据长度为 8 位                */
            (0x00 << 4 |           /* 设置帧格式为 SPI                  */
            (0x01 << 6 |           /* 设置传输时 SCK 为高电平有效       */
            (0x00 << 7 |           /* 数据在 SCK 的第 1 个时钟沿采样    */
            (0x00 << 8);           /* 设置 SSP 通讯位速率为默认值       */
```

2. SSP控制寄存器 1

本寄存器名称及映射如表 4.72所列。

表 4.72 SSP 控制寄存器 1 映射

寄存器名称	寄存器地址
SSPCR1	0xE006 8004

该寄存器控制着SSP控制器的工作方式，其位功能描述如表 4.73所列。

表 4.73 SSP 控制寄存器 1（SSPCR0 - 0xE006 80004）位描述

位	符号	值	描述	复位值
0	LBM	0 1	回写模式 正常工作模式 串行输入脚可用作串行输出脚（MOSI 或 MISO），而不是仅用作串行输入脚（MISO 或 MOSI 分别起作用）	0
1	SSE	0 1	SSP 使能 SSP 控制器禁能 SSP 控制器可与串行总线上的其它器件相互通信。在置位该位前，软件应将合适的控制信息写入其它 SSP 寄存器和中断控制器寄存器	0
2	MS	0 1	主机/从机模式。该位只能在 SSE 位为 0 时写入 SSP 控制器用作总线主机，驱动 SCLK、MOSI 和 SSEL 并接收 MISO 线 SSP 控制器用作总线从机，驱动 MISO、SCLK、MOSI 和 SSEL 引脚	0
3	SOD		从机输出禁能。该位只与从机模式有关（MS=1）。如果该位为 1，禁止 SSP 控制器驱动发送数据线（MISO）	0
7：4	-		保留，用户软件不要向保留位写入 1。从保留位读出的值未被定义	NA

操作示例：

```
SSPCR1  = (0x00 << 0) |            /* 不使用回写模式                    */
          (0x01 << 1) |            /* 设置 SSP 使能                    */
          (0x00 << 2) |            /* SSP 用作主机模式                 */
```

```
(0x00 << 3);                                    /* 从机输出禁能          */
```

3. SSP数据寄存器

本寄存器名称及映射如表 4.74所列。

表 4.74 SSP 数据寄存器映射

寄存器名称	寄存器地址
SSPDR	0xE006 8008

软件可将要发送的数据写入该寄存器，或从该寄存器读出接收的数据，如表 4.75所列。

表 4.75 SSP 数据寄存器（SSPDR - 0xE006 80008）位描述

位	符号	描述	复位值
15：0	DATA	**写：**当状态寄存器的 TNF 位为 1 指示 Tx FIFO 未满时，软件可将要发送的帧数据写入该寄存器。如果 Tx FIFO 以前为空且 SSP 控制器空闲，则立刻开始发送数据。否则，写入该寄存器的数据要等到所有数据发送（或接收）完后才能发送。如果数据长度小于 16 位，软件必须对数据进行调整[1]后再写入该寄存器 **读：**当状态寄存器的 RNE 位为 1 指示 Rx FIFO 不为空时，软件可读取该寄存器。软件读取该寄存器时，SSP 控制器将返回 Rx FIFO 中的最早收到的一帧数据。如果数据长度小于 16 位，该字段的数据必须进行合适的调整，低位对齐，高位补零	0x0000

[1] 软件调整是指：对于送入发送 FIFO 的数据，低地址的数据位先被 SSP 发送。用户需将预期先发送的数据位放到数据里的较低地址位。

操作示例：

```
SSPDR = data;                                   /* 发送数据 data         */
```

4. SSP状态寄存器

本寄存器名称及映射如表 4.76所列。

表 4.76 SSP 状态寄存器映射

寄存器名称	寄存器地址
SSPSR	0xE006 800C

该只读寄存器反映了SSP控制器的当前状态，位功能描述如表 4.77所列。

表 4.77 SSP 状态寄存器（SSPSR - 0xE006 8000C）位描述

位	符号	描述	复位值
0	TFE	发送 FIFO 空。发送 FIFO 为空时该位为 1，反之为 0	1
1	TNF	发送 FIFO 未满。Tx FIFO 满时该位为 0，反之为 1	1
2	RNE	接收 FIFO 不为空。接收 FIFO 为空时该位为 0，反之为 1	0
3	RFF	接收 FIFO 满。接收 FIFO 满时该位为 1，反之为 0	0
4	BSY	忙。SSP 控制器空闲时该位为 0，或当前正在发送/接收一帧数据和/或 Tx FIFO 不为空时该位为 1	0
7：5	-	保留，用户软件不要向保留位写入 1。从保留位读出的值未被定义	NA

操作示例：

```
while ( (SSPSR & 0x01) == 0 );                    /* 等待 TFE 置位,即发送 FIFO 空*/
```

5. SSP时钟预分频寄存器

本寄存器名称及映射如表 4.78所列。

表 4.78 SSP 时钟预分频寄存器映射

寄存器名称	寄存器地址
SSPCPSR	0xE006 8010

该寄存器控制着 P_{CLK} 到 SSP 预分频器时钟的分频系数，反过来，SSP 预分频器时钟被 SSPCR0 中的 SCR 分频后得到位时钟。

表 4.79 SSP 时钟预分频寄存器（SSPCPSR - 0xE006 80010）位描述

位	符号	描述	复位值
7：0	CPSDVSR	这是 2～254 中的一个偶数。它是 P_{CLK} 的分频因子，P_{CLK} 通过分频后得到预分频器输出时钟。位 0 读出时总是为 0	0

要点：必须适当地初始化 SSPCPSR 值，否则 SSP 控制器将不能正确发送数据。如果 SSP 在主控器模式下操作则 $CPSDVSR_{min}$＝2，若 SSP 工作在 SPI 从机模式下，该寄存器内的分频值无效。

操作示例：

```
SSPCPSR = 0x02;                    /* 设置 SSP 对 PCLK 的分频值,主模式下有效, 最小值为 0x02*/
```

SSP 中存在着三种时钟：P_{CLK}、SSP_{CLK}、BIT_{CLK}。P_{CLK} 是芯片外设的时钟周期，由处理器时钟周期分频获得；SSP_{CLK} 是 SSP 通讯中时钟引脚输出的时钟周期；BIT_{CLK} 是 SSP 通讯里传输一个位所用的时间即 SSP 位周期。这三种时钟之间的关系如下所列：

$$\mathbf{SSP_{CLK} = P_{CLK} / CPSDVSR}$$
$$\mathbf{BIT_{CLK} = SSP_{CLK} \times (SCR + 1)}$$

6. SSP中断使能设置/清除寄存器

本寄存器名称及映射如表 4.80所列。

表 4.80 SSP 中断使能设置/清除寄存器映射

寄存器名称	寄存器地址
SSPIMSC	0xE006 8014

该寄存器控制着 SSP 控制器 4 个中断条件的使能。

表 4.81 SSP 中断使能设置/清除寄存器(SSPIMSC - 0xE006 80014)位描述

位	符号	描述	复位值
0	RORIM	软件置位该位来使能接收溢出中断。当 Rx FIFO 满时又完成另一个帧的接收时该位置位。ARM 特别指出，发生接收溢出时新数据帧会将前面的数据帧覆盖	0

续上表

位	符号	描述	复位值
1	RTIM	软件置位该位来使能接收超时中断。当 Rx FIFO 不为空且在 32 个位时间内既未接收新数据又未从 FIFO 中读出数据时，产生接收超时	0
2	RXIM	软件置位该位，使得当 Rx FIFO 至少有一半为满时触发中断	0
3	TXIM	软件置位该位，使得当 Tx FIFO 至少有一半为空时触发中断	0
7：4	-	保留，用户软件不要向保留位写入 1。从保留位读出的值未被定义	NA

操作示例：

```
SSPIMSC = 0x01;                                    /* 设置 SSP 使能接收溢出中断      */
```

7. SSP原始中断状态寄存器

本寄存器名称及映射如表 4.82所列。

表 4.82 SSP 原始中断状态寄存器映射

寄存器名称	寄存器地址
SSPRIS	0xE006 8018

当一个中断条件出现时，该只读寄存器中对应的位置位，与中断是否通过 SSPIMSC 使能无关。

表 4.83 SSP 原始中断状态寄存器（SSPRIS - 0xE006 80018）位描述

位	符号	描述	复位值
0	RORRIS	当 Rx FIFO 满时又接收到另一帧数据时该位置位。ARM 特别指出，此时接收到的新数据帧会将前面的数据帧覆盖	0
1	RTRIS	如果 Rx FIFO 不为空，且在“超时周期”中没有被读出时该位置位	0
2	RXRIS	当 Rx FIFO 至少一半为满时该位置位	0
3	TXRIS	当 Tx FIFO 至少一半为空时该位置位	1
7：4	-	保留，用户软件不要向保留位写入 1。从保留位读出的值未被定义	NA

操作示例：

```
if ((SSPRIS & 0x01) == 0x01) {                          /* 当发生接收溢出时          */
    RcvBuf = SSPDR;     /* 读取 SSP 数据寄存器里的新数值（旧数据已被覆盖，因而丢失）  */
}
```

8. SSP使能中断（Masked Interrupt）状态寄存器

表 4.84 SSP 使能中断状态寄存器映射

寄存器名称	寄存器地址
SSPMIS	0xE006 801C

SSP中断使能（Masked Interrupt）状态寄存器及映射如表 4.84所列。 当一个中断条件出现且相应的中断在SSPIMSC中被使能时，该只读寄存器的对应位置位。当产生SSP中断时，中断服务程序可通过读该寄存器来判断中断源。该寄存器位功能描述如表 4.85所列。

表 4.85 SSP 使能中断状态寄存器（SSPMIS - 0xE006 8001C）位描述

位	符号	描述	复位值
0	RORMIS	当 Rx FIFO 满时又接收到另一帧数据，并且中断被使能时该位置位	0
1	RTMIS	当接收超时条件出现且中断被使能时该位置位。**注意：如果要接收更多的数据，接收超时可撤除**	0
2	RXMIS	当 Rx FIFO 至少一半为满且该中断被使能时该位置位	0
3	TXMIS	当 Tx FIFO 至少一半为空且该中断被使能时该位置位	0
7：4	-	保留，用户软件不要向保留位写入 1。从保留位读出的值未被定义	NA

操作示例：

```
void __irq SSP_ISR(void)                                    /* SSP 中断服务子程序          */
{
    if ((SSPMIS & 0x01) == 0x01) {                          /* 当发生接收溢出中断时        */
        RcvBuf = SSPDR;          /* 读取 SSP 数据寄存器里的新数值（旧数据已被覆盖）    */
        ......
    }
}
```

9. SSP中断清除寄存器

本寄存器名称及映射如表 4.86所列。

表 4.86 SSP 中断清除寄存器映射

寄存器名称	寄存器地址
SSPICR	0xE006 8020

软件可通过向该寄存器写入 1 来清除 SSP 控制器的相关中断，即清除相应的中断标志位并取消该中断的使能。

注意：另外 2 个中断可通过写或读相应的 FIFO 来清除，或通过清除 SSPIMSC 对应的位来禁能。

本寄存器的位功能描述如表 4.87所列。

表 4.87 SSP 中断清除寄存器（SSPICR - 0xE006 80020）位描述

位	名称	描述	复位值
0	RORIC	向该位写 1 来清除接收溢出中断	NA
1	RTIC	向该位写 1 来清除超时中断	NA
7：2	-	保留，用户软件不要向保留位写入 1。从保留位读出的值未被定义	NA

操作示例：

```
SSPICR |= 0x01;                                            /* 清除接收溢出中断            */
```

4.8.6 操作模式

1. 主机操作

下面的步骤描述了 SSP 设置为主机时如何处理数据传输。该处理假设上一次的数据传输已经结束。

- 设置 SSPCR0 寄存器，确定通讯格式和参数；

- 设置 SSPCR1 寄存器，确定主从模式和启动通讯；
- 将要发送的数据写入 SSPDR 寄存器，即启动 SSP 数据传输；
- 读取 SSPSR 寄存器，等待 TFE 位置位；
- 从 SSP 数据寄存器中读出接收到的数据（可选）；
- 如果有更多数据需要发送，则继续将数据写入 SSPDR。

主机初始化示例程序见程序清单 4.21。

程序清单 4.21 SSP 主机初始化示例

```
SSPCR0 =  (0x00 << 8) |                 /*  设置位速率为默认值                  */
          (0x00 << 7) |                 /* CPHA   时钟输出相位，仅 SPI 模式有效*/
          (0x01 << 6) |                 /* CPOL   时钟输出时钟，仅 SPI 模式有效*/
          (0x00 << 4) |                 /* FRF    帧格式：00=SPI               */
          (0x07 << 0);                  /* DSS    数据长度：0111=8 位          */

SSPCR1 =  (0x00 << 0) |                 /*  不使用回写模式                      */
          (0x01 << 1) |                 /*  设置 SSP 使能                       */
          (0x00 << 2) |                 /* SSP 用作主机模式                     */
          (0x00 << 3) |                 /*  从机输出禁能                        */
SSPCPSR = 0x02;                         /*  设置 SSP 从 PCLK 获得的分频值       */
```

SSP主机数据发送和接收示例程序见程序清单 4.22，向SSPDR寄存器写入一字节数据后，即可启动数据发送。SSP功能模块在发送数据时同时接收一字节数据，并将接收的数据返回。

程序清单 4.22 SSP 主机数据发送和接收程序

```
SSPDR = data;
while ( (SSPSR & 0x01) == 0 );          /*  等待 TFE 置位，即发送 FIFO 空       */
return (SSPDR);
```

2. 从机操作

从机初始化示例程序见程序清单 4.23，程序只对SSPCR0 和SSPCR1 进行设置，控制SSP为从机。为了能够连上SSP主机进行通讯，需要正确设置CPOL、CPHA、LBM控制位。注意，SSP时钟脉冲是由主机产生，所以从机无需初始化SSPCPSR寄存器。

程序清单 4.23 SSP 从机初始化示例

```
SSPCR0 =  (0x05 << 8) |                 /* SCR    设置 SPI 时钟分频            */
          (0x00 << 7) |                 /* CPHA   时钟输出相位，仅 SPI 模式有效*/
          (0x01 << 6) |                 /* CPOL   时钟输出极性，仅 SPI 模式有效*/
          (0x00 << 4) |                 /* FRF    帧格式：00=SPI               */
          (0x07 << 0);                  /* DSS    数据长度 0111=8 位           */
SSPCR1 =  (0x00 << 3) |                 /* SOD    从机输出禁能：1=禁止，0=允许*/
          (0x01 << 2) |                 /* MS     主从选择：0=主机，1=从机     */
          (0x01 << 1) |                 /* SSE    SSP 使能                     */
          (0x00 << 0);                  /* LBM    为 1 时是回写模式            */
```

4.8.7 SSP接口中断设置

LPC2103 芯片SSP接口具有中断功能，当发生接收溢出、接收超时、接收FIFO一半为满或者发送FIFO一半为空时，SSP接口就能触发中断，SSP接口中断与向量中断控制器（VIC）的关系如图 4.47所示。

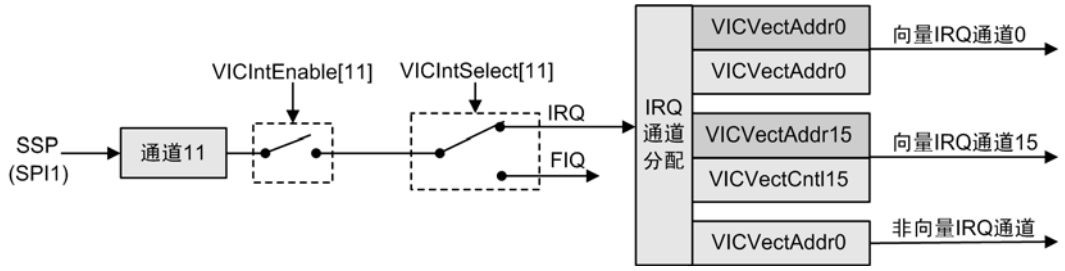


图 4.47 SSP 接口与 VIC 的关系

SSP 接口与 SPI 接口共用 VIC 的通道 11，中断使能寄存器 VICIntEnable 用来控制 VIC 通道的中断使能。

- 当 VICIntEnable[11] = 1 时，通道 11 中断使能，即：SPI 和 SSP 中断使能；

中断选择寄存器 VICIntSelect 用来分配 VIC 通道的中断。当某一位为“1”时，对应的通道中断分配为 FIQ；当某一位为“0”时，对应的通道中断分配为 IRQ。VICIntSelect[11] 用来控制通道 11，即：

- 当 VICIntSelect[11] = 1 时，SSP、SPI1 中断分配为 FIQ 中断；
- 当 VICIntSelect[11] = 0 时，SSP、SPI1 中断分配为 IRQ 中断；

当分配为 IRQ 时，还需要设置对应的中断优先级寄存器和地址寄存器。

LPC2103 的SSP中断有四类：接收溢出中断、接收超时中断、接收FIFO至少一半为满中断和发送FIFO至少一半为空中断，如图 4.48所示。

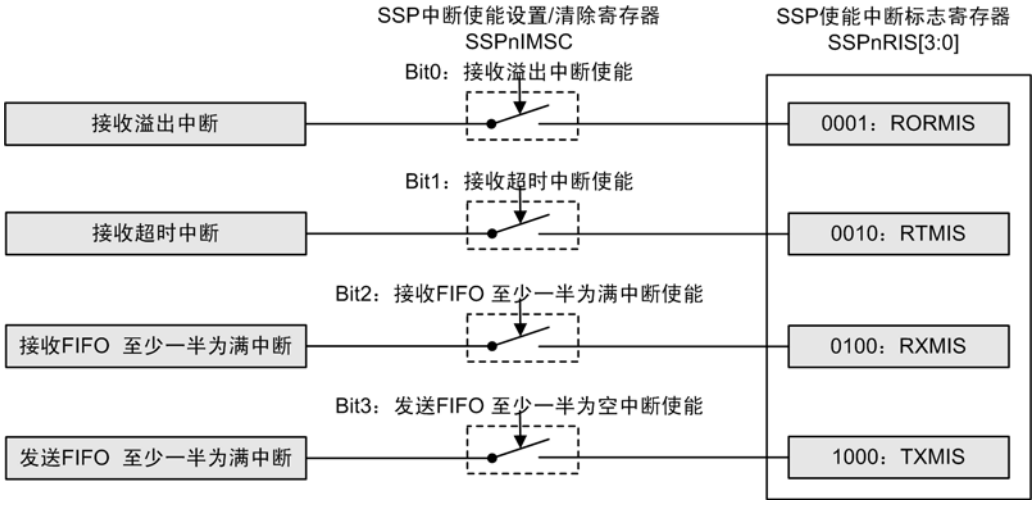


图 4.48 SSP 中断示意图

（1）接收溢出中断

LPC2103 有 8 帧的接收 FIFO，当接收 FIFO 满时又完成另一帧数据的接收时，就会触发“接收溢出中断”。

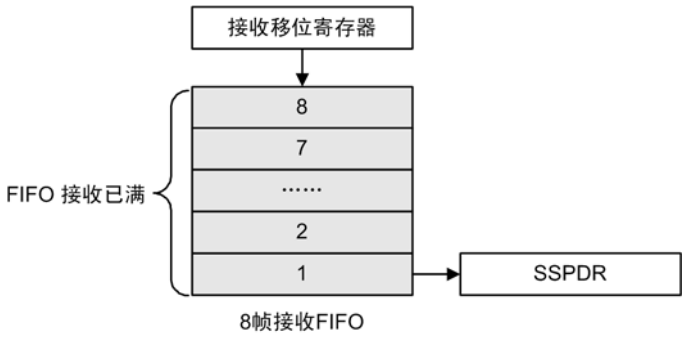


图 4.49 接收溢出中断示意图

如图 4.49所示，SSP接收数据时，先将数据送入“接收FIFO”。现已经接收了 8 帧数据，即接收FIFO已满。此时又接收到一帧数据，就会触发接收溢出中断，新的数据会将旧数据覆盖。

发生“接收溢出中断”后，向中断清除寄存器(SSPnICR)的 Bit0 写入“1”即可清除“接收溢出中断”标志位。

（2）接收超时中断

LPC2103 具有“接收超时中断”，当接收FIFO不为空且在 32 个位时间内既没有接收新数据又没有从FIFO中读出数据时，就会触发“接收超时中断”，如图 4.50所示。

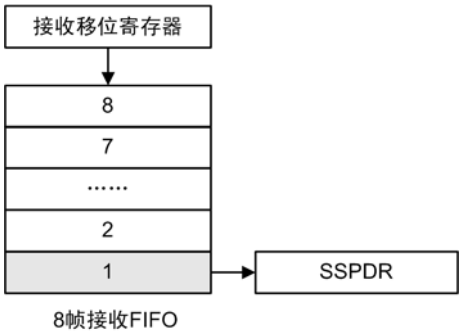


图 4.50 接收超时中断示意图

发生“接收超时中断”后，向中断清除寄存器(SSPnICR)的 Bit1 写入“1”即可清除“接收超时中断”标志位。

（3）接收 FIFO 至少一半为满中断

LPC2103 具有“接收FIFO至少一半为满中断”，当 8 帧的接收FIFO中至少含有 4 帧数据就会触发“接收FIFO至少一半为满中断”，如图 4.51所示。

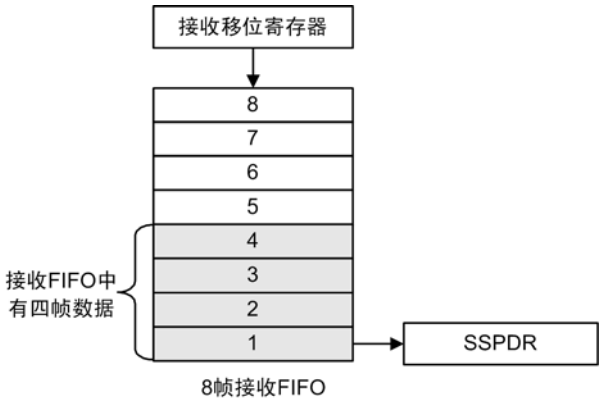


图 4.51 接收 FIFO 至少一半为满中断示意图

发生“接收 FIFO 至少一半为满中断”后，读取数据寄存器即可清除中断标志位；或者清除中断使能设置/清除寄存器(SSPnIMSC)中相应的位来禁止中断。

（4）发送 FIFO 至少一半为空中断

LPC2103 具有“发送FIFO至少一半为空中断”，当 8 帧的发送FIFO中最多含有 4 帧数据就会触发“发送FIFO至少一半为空中断”，如图 4.52所示。

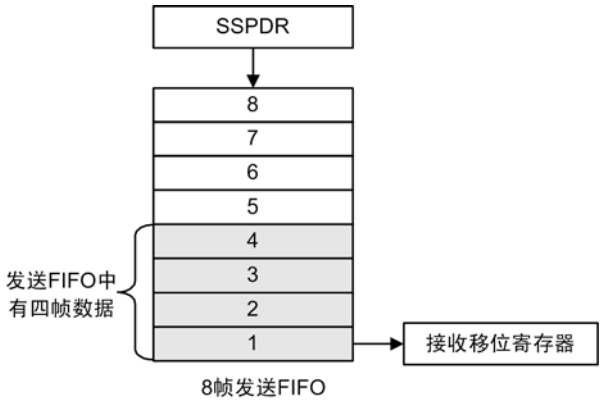


图 4.52 发送 FIFO 至少一半为空中断示意图

发生“发送 FIFO 至少一半为空中断”后，继续向数据寄存器中写入数据即可清除中断标志位；或者清除中断使能设置/清除寄存器(SSPnIMSC)中相应的位来禁止中断。

4.8.8 应用示例

本实验以 SSP 主机与 SPI 从机通信为示例，讲述 SSP 与 SPI 进行通信。在示例程序中，将 SSP 初始化为主机，SPI 初始化为从机。当 SPI 初始化为从机时，SPI 时钟发生器就会关闭。而当 SSP 作主机且只和一个从机通信时，直接用 SSEL1 作为片选即可。

程序中，SSP 接口循环发送数据，SPI 接收到数据后，通过 UART0 将数据发送到 PC 机并在软件 EasyARM-C.exe 的终端 LDE 上显示数据“0-7”；

注意：串口显示需要调用软件包“UART.h”、“UART.c”。

程序清单 4.24 SSP 与 SPI 主从机通信

```
# include "config.h"
# include "UART.h"
```

```
uint8 const uiBuf[8] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07};
/*********************************************************************************************
** 函数名称：DelayNS
** 功能描述：延时函数
** 入口参数：uiDly    值越大，延时时间越长
** 出口参数：无
*********************************************************************************************/
void DelayNS (uint32   uiDly)
{
    uint32   i;
    for(; uiDly > 0; uiDly--){
        for(i = 0; i < 50000; i++);
    }
}
/*********************************************************************************************
** 函数名称：SPIInit
** 功能描述：初始化 SPI 接口，设置为从机
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
void SPIInit(void)
{
    PINSEL0 = (PINSEL0 & 0xCFFF00FF) | 0x20005500;      /* 设置 SPI 引脚连接                */
    SPI_SPCCR     = 0x52;                               /* 设置 SPI 时钟分频                */
    SPI_SPCR      =    0 << 3 |                         /* CPHA = 0 第一个时钟采样          */
                       1 << 4 |                         /* CPOL = 1，SCK 低有效             */
                       0 << 5 |                         /* MSTR = 0，设置为从模式           */
                       0 << 6 |                         /* LSBF = 0，SPI 传输 MSB 在先      */
                       0 << 7 ;                         /* SPIE = 0，SPI 中断禁止           */
}
/*********************************************************************************************
** 函数名称：SSPIInit
** 功能描述：初始化 SSP 接口，设置为主机
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
void SSPIInit(void)
{
    PINSEL1 = (PINSEL1 & 0xFFFFF00F) | 0x00000540;      /* 设置 SSP 引脚连接                */
    SSPCR0   =    0x07 << 0 |                           /* DSS  数据长度为 8 位             */
                  0x00 << 4 |                           /* FRF  格式:SPI                    */
                  0x01 << 6 |                           /* CPOL = 1，SCK 低有效             */
                  0x00 << 7 |                           /* CPHA = 0, 第一个时钟采样         */
```

```
                    0x01 << 8 ;                         /* SCR  设置 SPI 时钟分频          */
    SSPCR1   =      0 << 0 |                            /* LBM  回写模式                   */
                    1 << 1 |                            /* SSE SSP 使能                    */
                    0 << 2 |                            /* MSTR = 0，设置为主机            */
                    0 << 3 ;                            /* SOD   = 0, 从机输出允许         */
    SSPCPSR  =      0x52;                               /*  设置 SSP 时钟分频              */
    SSPIMSC  =      0x02;                               /*  使能接收超时中断               */
    SSPICR   =      0x03;                               /*  清除中断标志                   */
}
/*********************************************************************************************
**  函数名称：SPIRcvByte
**  功能描述：从 SPI 总线接收 1 字节数据
**  入口参数：无
**  出口参数：无
*********************************************************************************************/
uint8 SPIRcvByte (void)
{
    while ((SPI_SPSR & 0x80) == 0);                     /*  等待数据接收完成               */
    return (SPI_SPDR);
}
/*********************************************************************************************
**  函数名称：SSPSendByte
**  功能描述：SSP 接口向总线发送 1 字节数据
**  入口参数：uiDat   待发送的数据
**  出口参数：接收到的数据
*********************************************************************************************/
uint8 SSPSendByte(uint8    uiDat)
{
    SSPDR = uiDat;
    while((SSPSR & 0x01) == 0);                         /*  等待 TFE 置位，即发送 FIFO 空  */
    return(SSPDR);
}
/*********************************************************************************************
**  函数名称：main
**  功能描述：SSP 做主机、SPI 做从机进行数据通信，并将 SSP 发送的数据通过串口发送到 PC 机显示
**  入口参数：无
**  出口参数：无
*********************************************************************************************/
int main(void)
{
    uint8 i;
    uint8 j;
    uint8 uiRcvData;
```

```
    UARTInit();                                     /*  串口初始化                          */
    SPIInit();                                      /*  初始化 SPI 接口                     */
    SSPIInit();                                     /*  初始化 SSP 接口                     */

    for(;;){
        for ( i = 0; i < 8; i++){                   /* SPI 发送数据                         */
            SSPSendByte(uiBuf[i]);
            DelayNS(20);

            uiRcvData = SPIRcvByte();               /* SSP 接收数据                         */
            DelayNS(10);
            for ( j = 0; j < 8; j++){
                PCDispChar (j, uiRcvData);          /*  将 SSP 接收到的数据发送到 PC 机  */
            }
            DelayNS(50);
        }
    }
    return 0;
}
```

4.9 UART接口

4.9.1 概述

LPC2103 微控制器包含有两个符合 16C550 工业标准的异步串行口（UART）UART0 和 UART1。其中，UART0 仅提供 TXD 和 RXD 信号引脚，UART1 增加了一个调制解调器（Modem）接口。UART1 仅仅比 UART0 多了一个 Modem 接口，其余方面两者都是完全相同的。

4.9.2 特性

- 16 字节收发 FIFO；
- 寄存器位置符合 16C550 工业标准；
- 接收器 FIFO 触发点可为 1、4、8 和 14 字节；
- 内置带波特率功能的波特率发生器；
- 包含使能实现软件和硬件流控制的机制；
- UART1 含有标准调制解调器接口信号，且硬件完全支持流控制（auto-CTS/RTS）。

小知识：16C550 是一种工业标准的 UART，此类 UART 芯片内部集成了可编程的波特率发生器、发送/接收 FIFO、处理器中断系统和各种状态错误检测电路等等，并具有完全的 Modem 控制能力。工作模式为全双工模式，支持 5～8 位数据长度，1/2 位停止位，可选奇偶校验位。

注意：UART0 没有完整的 Modem 接口信号，仅提供 TXD、RXD 信号引脚，因此，严格来说，UART0 不符合 16C550 工业标准。

4.9.3 引脚描述

UART管脚描述见表 4.88所列。

表 4.88 UART 管脚描述

管脚名称	UART 管脚	类型	描述
P0.1	RxD0	输入	UART0 串行输入 串行接收数据
P0.0	TxD0	输出	UART0 串行输出 串行发送数据
P0.9	RxD1	输入	UART1 串行输入 串行接收数据
P0.8	TxD1	输出	UART1 串行输出 串行发送数据
P0.14	DCD1	输入	数据载波检测： 低有效信号指示外部 modem 是否已经与 UART1 建立了通信连接，可以进行数据交换
P0.11	CTS1	输入	清零发送： 低电平有效信号指示外部 modem 的接收是否已经准备就绪，UART1 数据可通过 TxD1 发送
P0.12	DSR1	输入	数据设备就绪： 低电平有效指示外部 modem 是否准备建立与 UART1 的通信连接
P0.13	DTR1	输出	数据终端就绪： 低电平有效指示 UART1 准备建立与外部 modem 的连接
P0.15	RI1	输入	铃响指示： 有效低电平指示 modem 检测到电话的响铃信号
P0.10	RTS1	输出	请求发送： 低电平有效指示 UART1 打算向外部 modem 发送数据

4.9.4 典型应用

- 使用UART与其它控制器进行数据交换，如图 4.53所示。由于LPC2103 的I/O电压

为 3.3V（但I/O口可承受 5V电压），所以连接时注意电平的匹配；

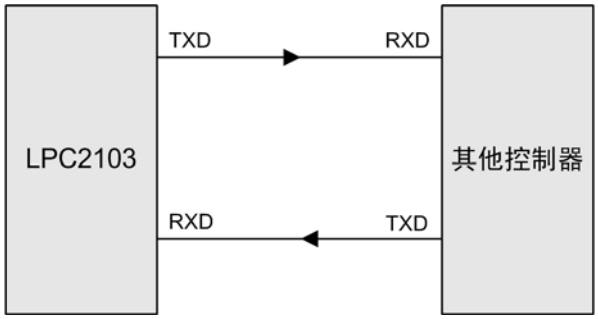


图 4.53 使用串口进行数据交换

- 使用UART与PC机通讯，如图 4.54所示。由于PC机串口是RS-232 电平，所以连接时需要使用RS-232 转换器，LPC2103 就是通过UART0 进行ISP操作的；

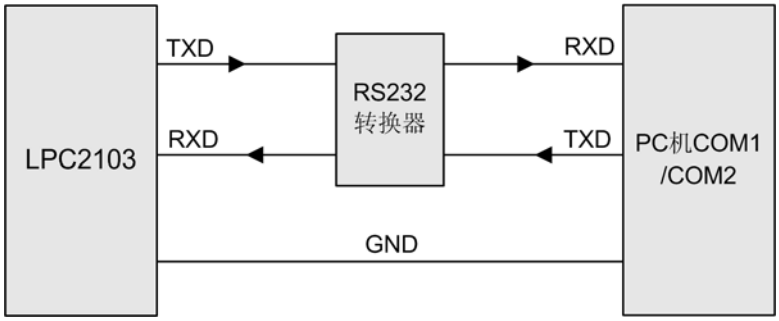


图 4.54 使用串口与 PC 机通讯

- 当使用Modem接口时，需要一个RS-232 转换器将信号转换为RS-232 电平后，才能与Modem连接，如图 4.55所示。

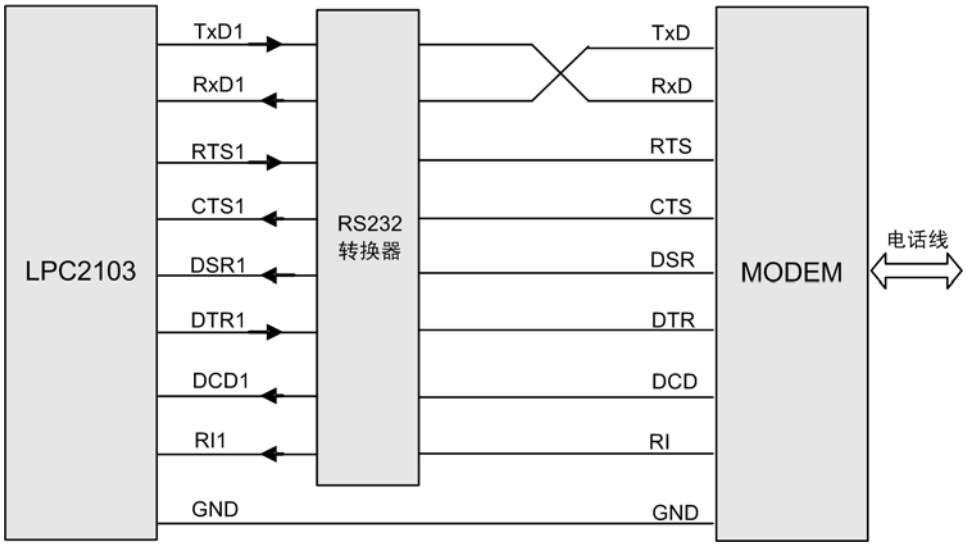


图 4.55 UART1 与 Modem 接口电路

4.9.5 寄存器描述

UART所包含的寄存器如表 4.89所列。除数锁存访问位（DLAB）位于UnLCR的bit7，它使能对除数锁存寄存器的访问。

表 4.89 UART 寄存器映射

名称	描述	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	访问	复位值*
UnRBR	接收缓冲	MSB	读数据						LSB	RO	未定义
UnTHR	发送保持	MSB	写数据						LSB	WO	NA
UnIER	中断使能	0	0	0	0	使能 Modem 状态中断	使能 Rx 状态中断	使能 THRE 中断	使能 Rx 数据可用中断	R/W	0
UnIIR	中断 ID	FIFO 使能		0	0	IIR3	IIR2	IIR1	IIR0	RO	0x01
UnFCR	FIFO 控制	Rx 触发		保留		—	Tx FIFO 复位	Rx FIFO 复位	FIFO 使能	WO	0
UnLCR	控制	DLAB	设置间隔	奇偶固定	偶选择	奇偶使能	停止位个数	字长度选择		R/W	0
UnMCR	Modem 控制	0	0	0	回送	0	0	RTS	DTR	R/W	0
UnLSR	状态	Rx FIFO 错误	TEMT	THRE	BI	FE	PE	OE	DR	RO	0x60
UnMSR	Modem 控制	DCD	RI	DSR	CTS	Delta DCD	后沿 RI	Delta DSR	Delta CTS	RO	0
UnDLL	除数锁存 LSB	MSB							LSB	R/W	0x01
UnDLM	除数锁存 MSB	MSB							LSB	R/W	0
UnSCR	高速缓存	MSB							LSB	R/W	0x00
UnTER	发送使能	TXEN	—	—	—	—	—	—	—	R/W	0x80
UnACR	自动波特率控制	—	—	—	—	—	—	ABTOInt Clr	ABEOInt Clr	R/W	0x00
		—	—	—	—	—	AutoRestart	Mode	Start		
UnFDR	小数分频	MULVAL				DIVADDVAL				R/W	0x10

*：复位值仅指已使用位中保存的数据，不包括保留位的内容。

注：寄存器 UnMCR 和 UnMSR 是 Modem 寄存器，只有 UART1 含有。

其中寄存器UnRBR、UnTHR和UnDLL的地址是相同的，寄存器UnIER和UnDLM的地址是相同的。对于这些寄存器的访问是通过DLAB位和读写方式来确定的，如表 4.90所列。

表 4.90 对相同地址寄存器的访问方法

UART	寄存器	地址	访问方式

UART0	U0RBR	0xE000 C000	DLAB=0，对地址：0xE000 C000 进行读访问
	U0THR		DLAB=0，对地址：0xE000 C000 进行写访问
	U0DLL		DLAB=1，对地址：0xE000 C000 进行访问
	U0IER	0xE000 C004	DLAB=0，对地址：0xE000 C004 进行访问
	U0DLM		DLAB=1，对地址：0xE000 C004 进行访问
UART1	U1RBR	0xE001 0000	DLAB=0，对地址：0xE001 0000 进行读访问
	U1THR		DLAB=0，对地址：0xE001 0000 进行写访问
	U1DLL		DLAB=1，对地址：0xE001 0000 进行访问
	U1IER	0xE001 0004	DLAB=0，对地址：0xE001 0004 进行访问
	U1DLM		DLAB=1，对地址：0xE001 0004 进行访问

1. UART接收器缓存寄存器（U0RBR - 0xE000 C000，U1RBR - 0xE001 0000，DLAB=0，只读）

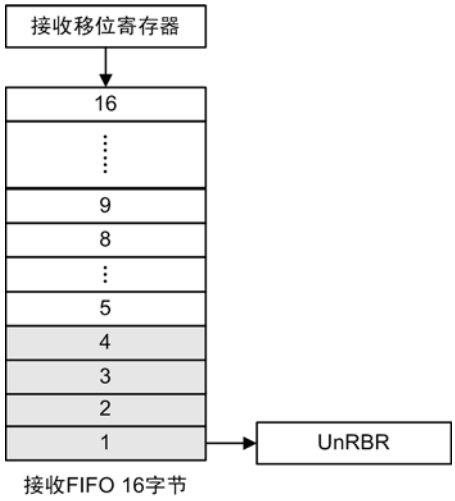


图 4.56 接收 FIFO 示意图

LPC2103 的UART0 和UART1 各含有 16 字节的接收FIFO，UnRBR是UARTn接收FIFO的出口，如图 4.56所示，它包含了最早接收到的字符，可通过总线接口读出。串口接收数据时低位在先，即LSB(bit0)为最早接收到的数据位。如果接收到的数据小于 8 位，未使用的MSB填充为 0。UnRBR寄存器描述见表 4.91。

表 4.91 UART 接收器缓存寄存器

UnRBR	功能	描述	复位值
7:0	接收器缓存	接收器缓存寄存器包含 UARTn Rx FIFO 当中最早接收到的字节	未定义

注：如果要访问 UnRBR，UnLCR 的除数锁存访问位（DLAB）必须为 0。UnRBR 为只读寄存器。

操作示例

```
while((U0LSR & 0x01) == 0);                    /* 等待接收标志置位          */
data_buf = U0RBR;                              /* 保存接收到的数据          */
```

2. UART发送器保持寄存器（U0THR - 0xE000 C000，U1THR - 0xE001 0000，DLAB=0，只写）

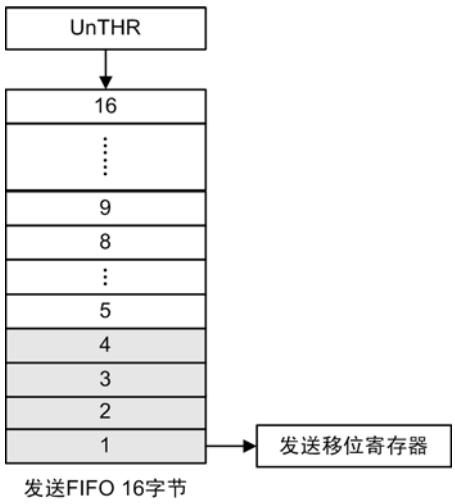


图 4.57 发送 FIFO 示意图

LPC2103 的UART0 和UART1 各含有 16 字节的发送FIFO，UnTHR是UARTn发送FIFO的入口，如图 4.57所示。它包含了发送FIFO中最新的字符，可通过总线接口写入。串口发送数据时低位在先，LSB（bit0）代表最先发送的位。U0THR寄存器描述见表 4.92。

表 4.92 UART 发送器保持寄存器

UnTHR	功能	描述	复位值
7:0	发送器保持	写 UARTn 发送器保持寄存器，使数据保存到 UARTn 发送 FIFO 当中。当字节到达 FIFO 的最底部并且发送器就绪时，该字节将被发送	N/A

注：如果要访问 UnTHR，UnLCR 的除数锁存访问位（DLAB）必须为 0。UnTHR 为只写寄存器。

操作示例

```
U0THR = data;                          /* 发送数据          */
while((U0LSR & 0x40)==0);              /* 等待数据发送完毕  */
```

3. UART除数锁存寄存器（U0DLL - 0xE000 C000，U0DLM - 0xE000 C004，DLAB=1）（U1DLL - 0xE001 0000，U1DLM - 0xE001 0004，DLAB=1）

UART0 和 UART1 各含有一个独立的波特率发生器，除数锁存是波特率发生器的一部分，它保存了用于产生波特率时钟的 VPB 时钟（pclk）分频值，波特率时钟是波特率的 16 倍。UnDLL 和 UnDLM 寄存器一起构成一个 16 位除数，UnDLL 包含除数的低 8 位，UnDLM 包含除数的高 8 位。值 0x0000 被看作是 0x0001，因为除数是不允许为 0 的。由于 UnDLL 与 UnRBR/UnTHR 共用同一地址，UnDLM 与 UnIER 共用同一地址，所以访问 UART 除数锁存寄存器时，除数锁存访问位（DLAB）必须为 1，以确保寄存器的正确访问。

UnDLL寄存器描述见表 4.93，UnDLM寄存器描述见表 4.94。

表 4.93 UART 除数锁存 LSB 寄存器

UnDLL	功能	描述	复位值
7:0	除数锁存 LSB 寄存器	UARTn 除数锁存 LSB 寄存器与 UnDLM 寄存器一起决定 UARTn 的波特率	0

表 4.94 UART 除数锁存 MSB 寄存器

UnDLM	功能	描述	复位值
7:0	除数锁存 MSB 寄存器	UARTn 除数锁存 MSB 寄存器与 UnDLL 寄存器一起决定 UARTn 的波特率	0

波特率的计算： $\mathrm{Baud} = \dfrac{\mathrm{Fpclk}}{16 \times (\mathrm{UnDLM} : \mathrm{UnDLL})}$

其中，（UnDLM：UnDLL）是由 UnDLL 和 UnDLM 一起构成的 16 位除数。

操作示例

```
U0LCR    = 0x80;                                    /* DLAB=1                    */
U0DLM    = ((Fpclk/16)/baud) / 256;
U0DLL    = ((Fpclk/16)/baud) % 256;
```

4. UART中断使能寄存器（U0IER - 0xE000 C004，U1IER - 0xE001 0004，DLAB=0）

UnIER可以使能 4 个UARTn中断源，UnIER寄存器描述见表 4.95。

表 4.95 UART 中断使能寄存器

UnIER	功能	描述	复位值
0	RBR 中断使能[1]	0：禁止 RDA 中断，1：使能 RDA 中断 UnIER[0]使能 UARTn 接收数据可用中断，它还控制字符接收超时中断	0
1	THRE 中断使能	0：禁止 THRE 中断，1：使能 THRE 中断 UnIER[1]使能 UARTn THRE 中断，该中断的状态可从 UnLSR[5]读出	0
2	Rx 状态中断使能	0：禁止 Rx 状态中断，1：使能 Rx 状态中断 UnIER[2]使能 UARTn Rx 状态中断，该中断的状态可从 UnLSR[4:1]读出	0
3	Modem 状态中断使能[2]	0：禁止 Modem 中断，1：使能 Modem 中断 U1IER[3]使能 Modem 中断，中断的状态可从 U1MSR[3:0]读取	0
6:4	保留	保留，用户软件不要向其写入 1，从保留位读出的值未被定义	NA
7	CTS 中断使能[1]	如果自动 CTS 模式被使能，那么该位使能/禁能 CTS1 信号跳变上产生的 modem 状态中断。如果自动 CTS 模式被禁止，若 modem 状态中断使能（U1IER[3]）置位，那么 CTS1 跳变将产生一个中断 在正常操作下，CTS1 信号跳变将产生 modem 状态中断，除非中断已被禁止（通过清零 U1IER 寄存器中的位 U1IER[3]）。在自动 CTS 模式中，只要 U1IER[3]和 U1IER[7]位置位，则 CTS1 位上的跳变将触发一个中断 0：禁止 CTS 中断 1：使能 CTS 中断	0

续上表

UnIER	功能	描述	复位值
8	ABTOIntEn 自动波特率超时中断使能	U0IER8 使能自动波特率超时中断 0：禁止自动波特率超时中断 1：使能自动波特率超时中断	0
9	ABEOIntEn 自动波特率完成中断使能	U0IER9 使能自动波特率中断结束 0：禁止自动波特率中断结束 1：使能自动波特率中断结束	0
31：10	—	保留，用户软件不要向保留位写入 1。从保留位读出的值未被定义	NA

说明：

[1]：RBR 中断包含了两个中断源，一是接收数据可用(RDA)中断，即正确接收到数据；二是接收超时中断(CTI)；

[2]：UART1 含有 Modem 接口，所以，只有 U1IER 寄存器含有该位，U0IER 寄存器没有。

操作示例

```
U0IER = 0x01;                                    /* 使能 RBR 中断，即，接收中断    */
```

5. UART中断标识寄存器（U0IIR - 0xE000 C008，U1IIR - 0xE001 0008，只读）

UnIIR提供状态代码用于指示一个挂起中断的中断源和优先级，在访问UnIIR过程中，中断被冻结。如果在访问UnIIR时产生了中断，该中断被记录，下次访问UnIIR时可读出该中断。UART中断标识寄存器描述如表 4.96所列。

表 4.96 UART 中断标识寄存器

UnIIR	功能	描述	复位值
0	中断挂起	**0**：至少有 1 个中断被挂起，**1**：没有挂起的中断 UnIIR[0]为低有效，挂起的中断可通过 UnIIR[3:1]确定	1
3:1	中断标识	**011**：1－接收状态（RLS），**010**：2a－接收数据可用（RDA）[1] **110**：2b－字符超时指示（CTI）[2]，**001**：3－THRE 中断[3] **000**：4－Modem 中断，只有 UART1 才含有 Modem 中断 UnIIR 的 bit3 指示对应于 UARTn 接收 FIFO 的中断，上面未列出的 UnIIR[3:1]的其它组合都为保留值（100, 101, 111）	0
5:4	保留	保留，用户软件不要向其写入 1，从保留位读出的值未被定义	NA
7:6	FIFO 使能	这些位等效于 UnFCR[0]	0
8	ABEOInt 自动波特率完成中断	自动波特率中断的结束。如果成功完成波特率且中断被使能则 ABEOInt 为 1	0
9	ABTOInt 自动波特率超时中断	自动波特率超时中断。如果波特率超时且中断被使能，则 ABTOInt 为 1	0
31：10	-	保留，用户软件不要向保留位写入 1。从保留位读出的值未被定义	NA

说明：

[1]、[2]：RDA 中断和 CTI 中断的优先级是相同的，均是第二级；

[3]：UART0 没有 Modem 接口，因此，只有 U1IIR[3:1]才会存在 000 的组合。

UART0 中断源和中断使能关系如图 4.58所示。

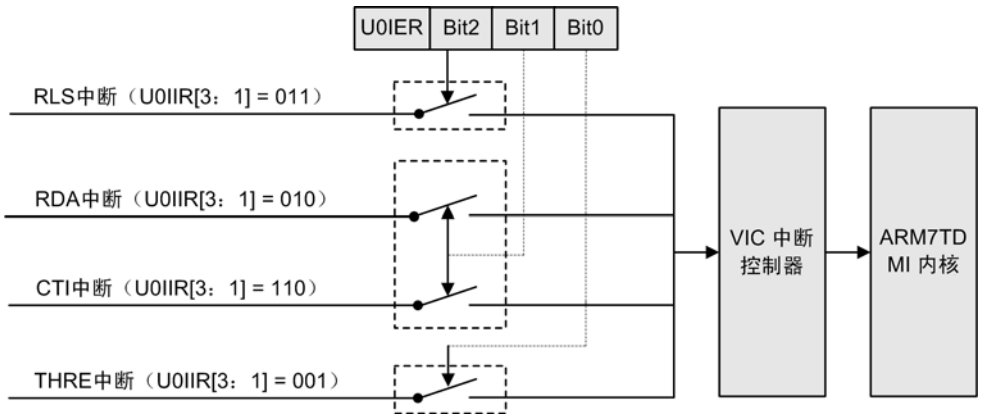


图 4.58 UART0 中断源和中断使能关系图

UART1 中断源和中断使能关系如图 4.59所示。

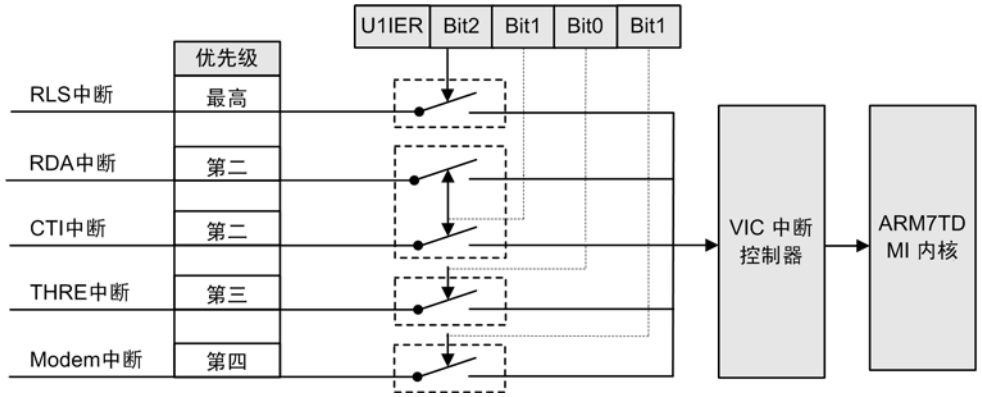


图 4.59 UART1 中断源和中断使能关系图

中断的处理见表 4.97，给定了UnIIR[3:0]的状态，中断处理程序就能确定中断源以及如何清除激活的中断。

表 4.97 UART 中断处理

UnIIR[3:0]	优先级	中断类型	中断源	中断复位
0001	—	无	无	—
0110	最高	Rx 状态/错误	OE, PE, FE, 或 BI	UnLSR 读操作
0100	第二	Rx 数据可用	Rx 数据可用或 FIFO 模式下（U0FCR[0] = 1）到达触发点	UnRBR 读或 UARTn FIFO 低于触发值
1100	第二	字符超时指示	接收 FIFO 包含至少 1 个字符并且在一段时间内无字符输入或移出，该时间的长短取决于 FIFO 中的字符数以及在 3.5 到 4.5 字符的时间内的触发值，实际的时间为：[(字长度)×7-2]×8+[(触发值-字符数)×8+1] P_{CLK}	UnRBR 读操作
0010	第三	THRE	THRE	UnIIR 读或 UnTHR 写操作
0000	第四	Modem 状态	CTS, DSR, RI 或 DCD	MSR 读操作
注： “0011”, “0101”, “0111”, “1000”, “1001”, “1010”, “1011”, “1101”, “1110”, “1111”为保留值				

- **UART RLS 中断**

RLS（UnIIR[3:1]=011）是最高优先级的中断，只要 UARTn Rx 输入产生下面 4 个错误中的任意一个，该中断标志将置位。即溢出错误（OE）、奇偶错误（PE）、帧错误（FE）、间隔中断（BI）。通过查看 UnLSR[4:1] 可以得到错误标志，当读取 UnLSR 寄存器时，清除该中断标志。

- **UART RDA 中断**

RDA（UnIIR[3:1]=010）与 CTI 中断（UnIIR[3:1]=110）共用第二优先级。当 UARTn Rx FIFO 达到 UnFCR[7:6]所定义的触发点时，RDA 被激活。当 UARTn Rx FIFO 的深度低于触发点时，RDA 复位。当 RDA 中断激活时，CPU 可读出由触发点所定义的数据块。

- **UART CTI 中断**

CTI（UnIIR[3:1]=110）为第二优先级中断。当接收 FIFO 中的有效数据个数少于触发个数时（至少有一个），如果长时间没有数据到达，将触发 CTI 中断。这个触发时间为：接收 3.5 到 4.5 个字符的时间。

“3.5～4.5 个字符的时间”，其意思是在串口当前的波特率下，发送 3.5～4.5 个字节所需要的时间。产生 CTI 中断后，对接收 FIFO 的任何操作都会清除该中断标志：

- 从接收 FIFO 中读取数据，即，读取 UnRBR 寄存器；
- 有新的数据送入接收 FIFO，即，接收到新数据。

需要注意的是：当接收 FIFO 中存在多个数据，从 UnRBR 读取数据，但是没有读完所有数据，那么在经过 3.5～4.5 个字节的时间后将再次触发 CTI 中断；

例如，一个外设向 LPC2103 发送 105 个字符，而 LPC2103 的接收触发值为 10 个字符，那么前 100 个字符将使 CPU 接收 10 个 RDA 中断，而剩下的 5 个字符使 CPU 接收 1～5 个 CTI 中断（取决于服务程序）。

- **UART THRE 中断**

THRE(UnIIR[3:1]=001)为第三优先级中断。这个中断称为发送 FIFO 为空中断。但是，并非只要 FIFO 为空便激活中断。THRE 中断有如下特性：

- 系统启动时，虽然发送 FIFO 为空，但不会产生 THRE 中断；
- 在上一次发生 THRE 中断后，仅向发送 FIFO 中写入 1 个字节数据，将在延时一个字节加上一个停止位的时间后发生 THRE 中断；
- 如果在发送 FIFO 中有过两个字节以上的数据，但是现在发送 FIFO 为空时，将立即触发 THRE 中断。

发送FIFO的结构示意图如图 4.60所示。当FIFO为空时，向其中写入 1 个字节的数据，该数据会直接传送到发送移位寄存器（UnTSR）中，这时发送FIFO为空。

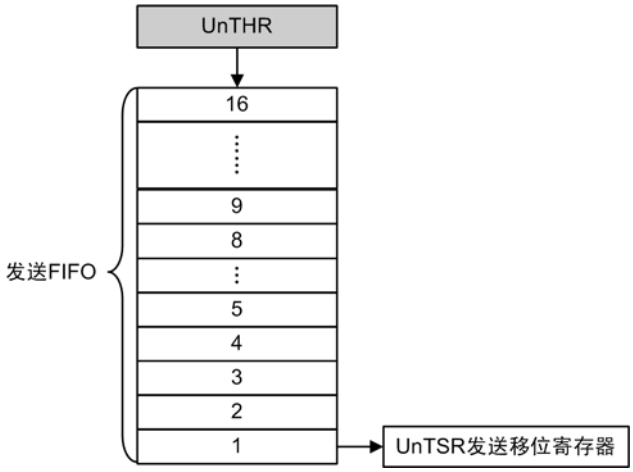


图 4.60 发送 FIFO 结构示意图

如果此时产生“发送 FIFO 为空中断”，那么会影响紧接着写入发送 FIFO 的数据，因此，要等到将该字节数据以及停止位发送完毕后才能产生中断。如果发送 FIFO 中含有 2 个字节以上的数据，那么当发送 FIFO 为空后，便会产生 THRE 中断。

当 THRE 中断为当前有效的最高优先级中断时，向 UnTHR 寄存器写数据，或者对 UnIIR 的读操作，都会清除 THRE 中断标志。

- **Modem 中断**

Modem 中断（U1IIR[3:1]=000）是最低优先级中断，只要在 Modem 输入引脚 DCD、DSR 或 CTS 上发生任何状态变化，该中断就会激活。此外，Modem 输入口 RI 上低到高电平的跳变也会产生一个 Modem 中断。Modem 中断源可通过检查 U1MSR[3:0]得到，读取 U1MSR 可以清除 Modem 中断标志。

注意：只有 UART1 接口才具有 Modem 中断。

6. UART FIFO控制寄存器（U0FCR - 0xE000 C008，U1FCR - 0xE001 0008）

UnFCR控制UARTn收发FIFO的操作，UnFCR寄存器描述见表 4.98。

表 4.98 UART FIFO 控制寄存器

UnFCR	功能	描述	复位值
0	FIFO 使能	为 1 时使能对 UARTn 收发 FIFO 以及 UnFCR[7:1]的访问，该位必须置位以实现正确的 UARTn 操作，该位的任何变化都将使 UARTn 的收发 FIFO 清空	0
1	Rx FIFO 复位	该位置位会清零 UARTn 接收 FIFO 中的所有字节并复位指针逻辑，操作完成后，该位会自动清零	0
2	Tx FIFO 复位	该位置位会清零 UARTn 发送 FIFO 中的所有字节并复位指针逻辑，操作完成后，该位会自动清零	0
5:3	保留	保留，用户软件不要向其写入 1，从保留位读出的值未被定义	NA
7:6	Rx 触发选择	**00**：触发点 0（默认 1 字节），**01**：触发点 1（默认 4 字节） **10**：触发点 2（默认 8 字节），**11**：触发点 3（默认 14 字节） 这两个位决定在激活中断之前，UARTn 接收 FIFO 必须写入多少个字节。4 个触发点由用户在编程时定义，可以选择所需要的触发深度	0

操作示例

```
U0FCR = 0x81;                                    /* UART0 接收缓冲区的触发点为 8 字节   */
```

7. UART控制寄存器（U0LCR - 0xE000 C00C, U1LCR - 0xE001 000C）

UnLCR决定发送和接收数据字符的格式，UnLCR寄存器描述见表 4.99。

表 4.99 UART 控制寄存器

UnLCR	功能	描述	复位值
1:0	字长度选择	**00**：5 位字符长度，**01**：6 位字符长度，**10**：7 位字符长度，**11**：8 位字符长度	0
2	停止位选择	**0**：1 个停止位，**1**：2 个停止位（如果 UnLCR[1:0]=00 则为 1.5）	0
3	奇偶使能	**0**：禁止奇偶产生和校验，**1**：使能奇偶产生和校验	0
5:4	奇偶选择	**00**：奇校验，**01**：偶校验，**10**：强制为 1，**11**：强制为 0	0
6	间隔控制[1]	**0**：禁止间隔发送，**1**：使能间隔发送 当 UnLCR 的 bit6 为 1 时，输出引脚 UARTn TxD 强制为逻辑 0	0
7	除数锁存访问位	**0**：禁止访问除数锁存寄存器，**1**：使能访问除数锁存寄存器	0

[1]：当该位为 1 时，输出引脚（TxD0）强制为逻辑 0，可以引起通信对方(LPC2000)产生间隔中断。在某些通信方式中，使用间隔中断作为通信的起始信号（如：LIN Bus）。

操作示例

```
U0LCR = 0x03;                    /* UART 的工作模式为：8 位字符长度，1 个停止位，无奇偶校验位   */
```

8. UART1 Modem控制寄存器（U1MCR - 0xE0010010）

U1MCR使能Modem的回写模式并控制Modem的输出信号，见表 4.100。

表 4.100 U1RT1 Modem 控制寄存器

U1MCR	功能	描述	复位值
0	DTR 控制	选择 modem 输出引脚 DTR，该位在回写模式激活时读出为 0	0
1	RTS 控制	选择 modem 输出引脚 RTS，该位在回写模式激活时读出为 0	
2	保留	保留，用户软件不要向其写入 1，从保留位读出的值未被定义	NA
3	保留	保留，用户软件不要向其写入 1，从保留位读出的值未被定义	NA
4	回写模式选择	**0**：禁止 modem 回写模式，**1**：使能 modem 回写模式 modem 回写模式提供了一个执行回写测试的诊断机制。发送器输出的串行数据在内部连接到接收器的串行输入端。输入脚 RxD1 对回写模式无影响，输出脚 TxD1 保持总为 1 的状态。4 个 modem 输入（CTS, DSR, RI 和 DCD）与外部断开。从外部来看，modem 的输出端（RTS, DTR）无效。在内部，4 个 modem 输出连接到 4 个 modem 输入。这样连接的结果是 U1MSR 的高 4 位由 U1MCR 的低 4 位驱动，而不是在正常模式下由 4 个 modem 输入驱动。这样在回写模式下，写 U1MCR 的低 4 位就可产生 modem 状态中断	0
7:5	保留	保留，用户软件不要向其写入 1，从保留位读出的值未被定义	NA

9. UART状态寄存器（U0LSR - 0xE000 C014，U1LSR - 0xE001 0014，只读）

状态寄存器（UnLSR）为只读寄存器，它提供UARTn Tx和Rx模块的状态信息，UnLSR

寄存器的描述见表 4.101。

表 4.101 状态寄存器

UnLSR	功能	描述	复位值
0	接收数据就绪.（RDR）	**0**：UnRBR 为空，**1**：Un0RBR 包含有效数据 当 UnRBR 包含未读取的字符时，RDR 位置位；当 UARTn RBR FIFO 为空时，RDR 位清零	0
1	溢出错误（OE）	**0**：溢出错误状态未激活，**1**：溢出错误状态激活 溢出错误条件在错误发生后立即设置，UnLSR 读操作清零 OE 位。当 UARTn RSR 已经有新的字符就绪而 UARTn RBR FIFO 已满时，OE 位置位。此时 UARTn RBR FIFO 不会被覆盖，UARTn RSR 中的字符将丢失	0
2	奇偶错误（PE）	**0**：奇偶错误状态未激活，**1**：奇偶错误状态激活 当接收字符的奇偶位处于错误状态时产生一个奇偶错误，UnLSR 读操作清零 PE 位。奇偶错误检测时间取决于 UnFCR 的 bit0，奇偶错误与 UARTn RBR FIFO 中读出的字符相关	0
3	帧错误（FE）	**0**：帧错误状态未激活，**1**：帧错误状态激活 当接收字符的停止位为 0 时，产生帧错误。UnLSR 读操作清零 FE 位，帧错误检测时间取决于 UnFCR 的 bit0，帧错误与 UARTn RBR FIFO 中读出的字符相关。当检测到一个帧错误时，Rx 将尝试与数据重新同步并假设错误的停止位实际是一个超前的起始位。但即使没有出现帧错误，它也不能假设下一个接收到的字节是正确的	0
4	间隔中断（BI）	**0**：间隔中断状态未激活，**1**：间隔中断状态激活 在发送整个字符（起始位、数据、奇偶位和停止位）过程中 RxDn 如果都保持逻辑 0，则产生间隔中断。当检测到中断条件时，接收器立即进入空闲状态直到 RxDn 变为全 1 状态。UnLSR 读操作清零该状态位，间隔检测的时间取决于 UnFCR 的 bit0，间隔中断与 UARTn RBR FIFO 中读出的字符相关	0
5	发送保持寄存器空（THRE）	**0**：UnTHR 包含有效数据，**1**：UnTHR 空 当检测到 UARTn THR 空时，THRE 置位，UnTHR 写操作清零该位	1
6	发送器空（TEMT）	**0**：UnTHR 和/或 UnTSR 包含有效数据，**1**：UnTHR 和 UnTSR 空 当 UnTHR 和 UnTSR 都为空时，TEMT 置位；当 UnTSR 或 UnTHR 包含有效数据时，TEMT 清零	1
7	Rx FIFO 错误（RXFE）	**0**：UnRBR 中没有 UARTn Rx 错误，或 UnFCR 的 bit0 为 0；**1**：UnRBR 包含至少一个 UARTn Rx 错误 当一个带有 Rx 错误（例如帧错误、奇偶错误或间隔中断）的字符装入 UnRBR 时，RXFE 位置位。当读取 UnLSR 寄存器并且 UARTn FIFO 中不再有错误时，RXFE 位清零	0

10. UART1 Modem状态寄存器（U1MSR - 0x0E0010018）

U1MSR是一个只读寄存器，它提供Modem输入信号的状态信息，见表 4.102，U1MSR[3:0]在读取U1MSR时清零。需要注意的是，Modem信号对UART1 的操作没有直接影响，Modem信号的操作是通过软件来实现的。

表 4.102 UART1 Modem 状态寄存器

U1MSR	功能	描述	复位值
0	Delta CTS	0：没有检测到 Modem 输入引脚 CTS 上的状态变化 1：检测到 Modem 输入引脚 CTS 上的状态变化 当输入引脚 CTS 状态发生变化时，该位置位。读取 U1MSR 时清零	0
1	Delta DSR	0：没有检测到 Modem 输入引脚 DSR 上的状态变化 1：检测到 Modem 输入引脚 DSR 上的状态变化 当输入引脚 DSR 状态发生变化时，该位置位。读取 U1MSR 时清零	0
2	后沿 RI	0：没有检测到 Modem 输入引脚 RI 上的状态变化 1：检测到 Modem 输入引脚 RI 上的状态变化 当输入引脚 RI 状态发生变化时，该位置位。读取 U1MSR 时清零	0
3	Delta DCD	0：没有检测到 Modem 输入引脚 DCD 上的状态变化 1：检测到 Modem 输入引脚 DCD 上的状态变化 当输入引脚 DCD 状态发生变化时，该位置位。读取 U1MSR 时清零	0
4	CTS	清零发送状态：输入信号 CTS 的补码。在回写模式下，该位连接到 U1MCR 的 bit1	0
5	DSR	数据设置就绪状态：输入信号 DSR 的补码。在回写模式下，该位连接到 U1MCR 的 bit0	0
6	RI	响铃指示状态：输入信号 RI 的补码。在回写模式下， 该位连接到 U1MCR 的 bit2	0
7	DCD	数据载波检测状态：输入信号 DCD 的补码。在回写模式下，该位连接到 U1MCR 的 bit3	0

11. UART高速缓存寄存器（U0SCR – 0xE000 C01C，U1SCR - 0xE001 001C）

在操作UARTn时，UnSCR无效，用户可自由对该寄存器进行读或写，不提供中断接口向主机指示UnSCR所发生的读或写操作，UnSCR寄存器描述见表 4.103。

表 4.103 UARTn 高速缓存寄存器

UnSCR	功能	描述	复位值
7:0	—	一个可读可写的字节	0

12. UART小数分频寄存器（U0FDR – 0xE000 C028，U1FDR – 0xE001 0028）

表 4.104 UART 小数分频寄存器位描述

UnFDR	功能	描述	复位值
3：0	DIVADDVAL	波特率生成预分频除数值： 如果该字段为 0，小数波特率发生器将不会影响 UART0 波特率	0
7：4	MULVAL	波特率预分频乘数值： 不管小数波特率发生器是否使用，该字段必须大于或等于 1 以使 UART0 正常操作	1
31：8	—	保留，用户软件不要向保留位写入 1。从保留位读出的值未被定义	NA

该寄存器控制生成波特率的时钟预分频器。该寄存器的复位值保持小数波特率发生器功能禁止，以确保 UART 的软件和硬件与没有该特性的 UART 完全兼容。

可用下面的等式计算 UART 波特率：

$$\text{Baud} = \frac{\text{PCLK}}{16(\text{UnDLM} : \text{UnDLL})(1 + \text{DivAddVal} / \text{MulVal})}$$

其中，P_{CLK} 为外围时钟频率，UnDLM 和 UnDLL 为标准的 UART 波特率除数寄存器，DIVADDVAL 和 MULVAL 为 UART 小数波特率发生器特定的参数。

MULVAL 和 DIVADDVAL 的值应遵循以下的条件：

- $0 < \text{MULVAL} \le 15$；
- $0 \le \text{DIVADDVAL} \le 15$。

如果 UnFDR 寄存器值不遵循这两个请求，那么小数分频输出将出错。如果 DIVADDVAL 为 0，那么小数分频禁止且时钟将不会被分频。

当正在发送/接收数据或数据可能丢失或被破坏时，不应修改 U0FDR 的值。

在实际使用中，UART 波特率公式可用下面的形式表示：

$$\text{Baud} = \frac{\text{PCLK}}{16(\text{UxDLM}：\ \text{UxDLL})} \times \frac{\text{MulVal}}{\text{MulVal} + \text{DivAddVal}}$$

根据这种表示，小数波特率发生器也可描述为具有 MULVAL/（MULVAL+DIVADDVAL）系数的预分频。

13. UART波特率计算

例 1：使用上面的 $UART_{baudrate}$ 公式，假设系统的 P_{CLK}＝20MHz，如果 UnDL=130（UnDLM=0x00 和 UnDLL=0x82）、DIVADDVAL＝0 和 MULVAL＝1，则可得出 UART 的 $UART_{audrate}$=9615 bauds。

例 2：使用上面的 $UART_{baudrate}$ 公式，假设系统的 P_{CLK}＝20MHz，如果 UnDL=93（UnDLM=0x00 和 UnDLL=0x5D）、DIVADDVAL＝2 和 MULVAL＝5，则可得出 UART 的 $UART_{baudrate}$=9600 bauds。

表 4.105 外围时钟为 20MHz 时的波特率（P_{CLK}=20MHz）

期望的波特率	MULVAL=0 DIVADDVAL=0			选用 MULVAL & DIVADDVAL		
	UnDLM：UnDLL		%error[1]	UnDLM：UnDLL 十进制	小数的预分频值 MULDIV / (MULDIV + DIVADDVAL)	%error[1]
	十六进制	十进制				
50	61A8	25000	0.0000	25000	1/（1+0）	0.0000
75	411B	16667	0.0020	12500	3/（3+1）	0.0000
110	2C64	11364	0.0032	6250	11/（11+9）	0.0000
134.5	244E	9294	0.0034	3983	3/（3+4）	0.0001
150	208D	8333	0.0040	6250	3/（3+1）	0.0000
300	1047	4167	0.0080	3125	3/（3+1）	0.0000
600	0823	2083	0.0160	1250	3/（3+2）	0.0000
1200	0412	1042	0.0320	625	3/（3+2）	0.0000
1800	02B6	694	0.0640	625	9/（9+1）	0.0000
2000	0271	625	0.0000	625	1/（1+0）	0.0000
2400	0209	521	0.0320	250	12/（12+13）	0.0000

续上表

期望的波特率	MULVAL=0 DIVADDVAL=0			选用 MULVAL & DIVADDVAL		
	UnDLM：UnDLL		%error[1]	UnDLM：UnDLL 十进制	小数的预分频值 MULDIV / (MULDIV + DIVADDVAL)	%error[1]
	十六进制	十进制				
3600	015B	347	0.0640	248	5/（5+2）	0.0064
4800	0104	260	0.1600	125	12/（12+13）	0.0000
7200	00AE	174	0.2240	124	5/（5+2）	0.0064
9600	0082	130	0.1600	93	5/（5+2）	0.0064
19200	0041	65	0.1600	31	10/（10+11）	0.0064
38400	0021	33	1.3760	12	7/（7+12）	0.0594
56000	0021	22	1.4400	13	7/（7+5）	0.0160
112000	000B	11	1.4400	6	7/（7+6）	0.1600
115200	000B	11	1.3760	4	7/（7+12）	0.0594
224000	0006	6	7.5200	3	7/（7+6）	0.1600
448000	0003	3	7.5200	2	5/（5+2）	0.3520

[1] 期望波特率和实际波特率的百分比误差。

14. UART自动波特率控制寄存器（U0ACR – 0xE000 C020，U1ACR – 0xE001 0020）

UART 自动波特率控制寄存器（UnACR）控制测量波特率生成的输入时钟/数据率的过程，用户可自由对该寄存器进行读或写。

表 4.106 自动波特率控制寄存器描述

UnACR	功能	描述	复位值
0	Start 自动波特率启动位	自动波特率完成后该位自动清零 0：自动波特率停止（自动波特率不运行） 1：自动波特率启动（自动波特率正在运行）。自动波特率运行位。该位在自动波特率结束后自动清零	0
1	Mode 自动波特率模式选择位	0：模式 0 1：模式 1	0
2	AutoRestart 超时自动重启位	0：如果超时，不重新启动 1：如果超时则重新启动（计数器在下一个 UART Rx 下降沿重新启动）	0
7：3	—	保留，用户软件不要向保留位写入 1。从保留位读出的值未被定义	0
8	ABEOIntClr 自动波特率完成中断清零位	自动波特率完成中断清零位（仅可写访问）。写 1 将在 UnIIR 中清除相应的中断。写 0 无影响	0
9	ABTOIntClr 自动波特率超时中断清零位	自动波特率超时中断清零位（仅可写访问）。写 1 将在 UnIIR 中清除相应的中断。写 0 无影响	0
31：10	—	保留，用户软件不要向保留位写入 1。从保留位读出的值未被定义	0

15. 自动波特率

UART 自动波特率功能可用于测量基于“AT”协议（Hayes 命令）的输入波特率。如果使能，那么自动波特率特性将测量接收数据流的位时间，并根据该时间设置除数锁存寄存器 UnDLM 和 UnDLL。

自动波特率通过置位 UnACR0 来启动，通过清零 UnACR0 来停止。一旦自动波特率结束，起始位将清零，并且读该位将返回自动波特率的状态（等待/完成）。

可通过设置 UnACR 模式位来使用两种自动波特率测量模式：

模式 0 通过对 UART Rx 管脚上两个连续的下降沿进行测量（起始位的下降沿和第一位的下降沿）来得出波特率。

模式 1 通过测量 UART Rx 管脚上的下降沿和后续的上升沿之间的时间（起始位的长度）来得出波特率。

如果出现超时（速率测量计数器溢出），那么 UnACR AutoRestart 位可用于自动重新启动波特率测量。如果该位置位，新的测量将在下一个 UART Rx 下降沿开始。

自动波特率功能可产生两个中断：

- 自动波特率超时中断。如果中断使能，那么将设置 UnIIR ABTOInt 中断（UnIER ABToIntEn 置位且自动波特率测量计数器溢出）；
- 自动波特率完成中断。如果中断使能，那么将设置 UnIIR ABEOInt 中断（UnIER ABEoIntEn 置位且自动波特率成功完成）。

通过置位相应的 UnACR ABTOIntClr 和 ABEOIntEn 位来清零自动波特率中断。

在自动波特率运行期间，小数波特率发生器被禁止 （DIVADDVAL=0）。然而，如果小数波特率发生器被使能（DIVADDVAL＞0），那么它将影响 UART Rx 管脚波特率的测量，但 UnFDR 寄存器的值在速率测量后不会被修改。同时，当使用自动波特率时，任何写 UnDLM 和 UnDLL 寄存器的操作都应在写 UnACR 寄存器前完成。UART 支持的波特率最小和最大值受 P_{CLK}、数据位的个数、停止位和校验位影响。

$$\text{ratemin}=\frac{2\times \text{PCLK}}{16\times 2^{15}} \le \text{UART0}_{\text{baudrate}} \le \frac{\text{PCLK}}{16\times(2+\text{数据位}+\text{奇偶位}+\text{停止位})}=\text{ratemax}$$

16. 自动波特率模式

当软件正在期望“AT”命令时，它用期望的字符格式配置 UART 并设置 UnACR 起始位。可以不用关心除数锁存器 UnDLL 和 UnDLM 的初始值。由于“A”或“a”ASCII 编码（“A”=0x41，“a”=0x61），因此 UART Rx 管脚检测起始位且所期望字符的 LSB 由两个下降沿限定。当 UnACR 起始位置位时，自动波特率协议将执行以下过程：

- 一旦 UnACR 起始位置位，波特率测量计数器复位，同时 UART UnRSR 复位。UnRSR 波特率切换为最高的速率；
- UART Rx 管脚下降沿触发起始位的开始。速率测量计数器将开始对 P_{CLK}（可选择被小数波特率发生器预分频）进行计数；
- 在起始位的接收过程中，RSR 的波特率输入端产生 16 个具有 UART 输入时钟频率（被小数波特率发生器预分频）的脉冲（脉冲频率和 UART 输入时钟频率一致），保证起始位存储在 UnRSR 中；
- 在接收起始位（以及模式 0 下字符的 LSB）的过程中，速率计数器将随着 UART 输入时钟（P_{CLK}）的增加而增加；

- 如果 Mode=0，那么速率计数器将在 UART Rx 管脚的下一个下降沿停止。如果 Mode=1，那么速率计数器将在 UART Rx 管脚的下一个上升沿停止；
- 速率计数器的值被装入到 UnDLM/UnDLL，并且波特率将自动切换为正常操作模式。UnDLM/UnDLL 设置完成后，如果自动波特率结束中断被使能的话，UnIIR ABEOInt 将被置位。接下来 UnRSR 将继续接收“A/a”字符剩下的位。

4.9.6 UART中断

LPC2103 UART接口具有中断功能，而且由向量中断控制器（VIC）管理，UART0 中断和UART1 中断分别位于VIC中断通道 6 和通道 7，如图 4.61所示。

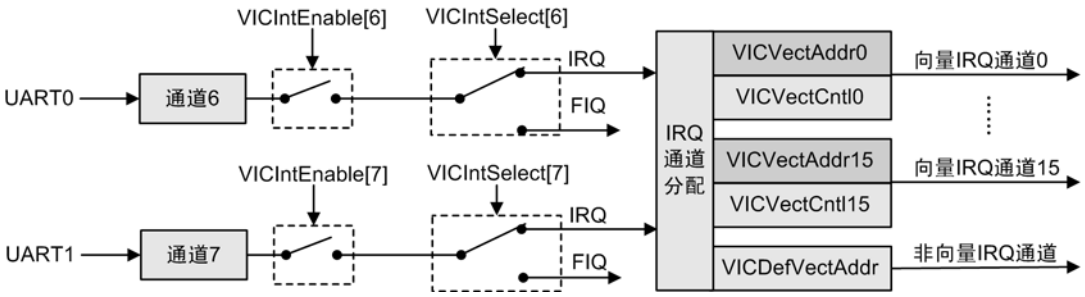


图 4.61 UART 与 VIC 的关系

UART0 和 UART 1 分别处于 VIC 的通道 6 和通道 7，中断使能寄存器 VICIntEnable 用来控制 VIC 通道的中断使能。

- 当 VICIntEnable[6] = 1 时，通道 6 中断使能，即：UART0 中断使能；
- 当 VICIntEnable[7] = 1 时，通道 7 中断使能，即：UART1 中断使能。

中断选择寄存器 VICIntSelect 用来分配 VIC 通道的中断。当某一位为“1”时，对应的通道中断分配为 FIQ；当某一位为“0”时，对应的通道中断分配为 IRQ。VICIntSelect[6]和 VICIntSelect[7]分别用来控制通道 6 和通道 7，即：

- 当 VICIntSelect[6] = 1 时，UART 0 中断分配为 FIQ 中断；
- 当 VICIntSelect[6] = 0 时，UART 0 中断分配为 IRQ 中断；
- 当 VICIntSelect[7] = 1 时，UART 1 中断分配为 FIQ 中断；
- 当 VICIntSelect[7] = 0 时，UART 1 中断分配为 IRQ 中断。

当分配为 IRQ 时，还需要设置对应的通道控制寄存器和地址寄存器。有关寄存器 VICVectCntln 和 VICVectAddrn 的说明，请参考“向量中断控制器（VIC）”一节。

LPC2103 UART中断主要分为 4 类：接收中断、发送中断、接收状态中断和Modem中断，如图 4.62所示。其中，接收状态中断指接收过程中发生了错误，即接收错误中断。只有UART1 接口具有Modem中断，UART0 接口由于没有Modem功能，所以没有Modem中断。

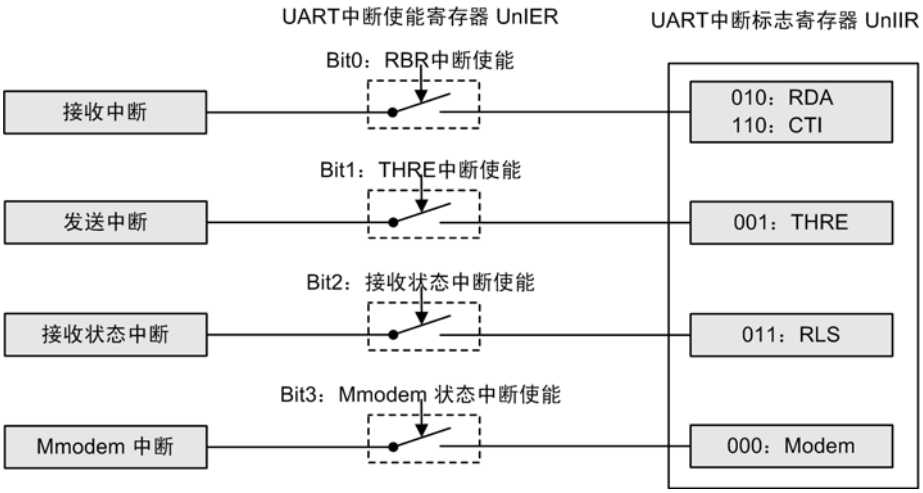


图 4.62　UART 中断示意图

1. 接收中断

对于 UART 接口来说，有两种情况可以触发 UART 接收中断：接收字节数达到接收 FIFO 的触发点（RDA）、接收超时（CTI）。

（1）接收字节数达到接收 FIFO 中的触发点（RDA）

LPC2103 UART 接口具有 16 字节的接收 FIFO，接收触发点可以设置为 1、4、8、14 字节，当接收到的字节数达到接收触发点时，便会触发中断。

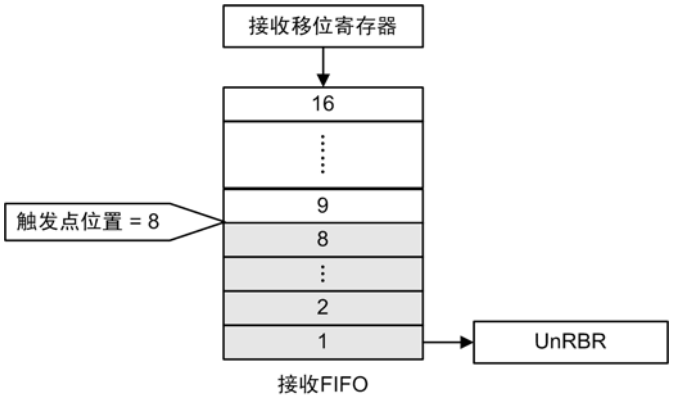


图 4.63　RDA 中断示意图

如图 4.63所示，通过UART FIFO控制寄存器UnFCR，将接收触发点设置为“8 字节触发”。那么当UART接收 8 个字节时，便会触发RDA中断（注：在接收中断使能的前提下）。

- RDA 中断初始化；

RDA中断初始化函数需要设置两个寄存器：UART中断使能寄存器UnIER和UART FIFO控制寄存器UnFCR。其中，UART FIFO控制寄存器UnFCR用来设置触发点。例如：将UART0接口设置为 8 字节触发，设置代码如程序清单 4.25所示。

程序清单 4.25　8 字节接收中断设置代码

```
……
U0IER = 0x01;                                   /* UART0 接收中断使能                  */
```

```
U0FCR = 0x81;                                   /* UART0 接收缓冲区的触发点为 8 字节    */
……
```

- RDA 中断服务程序。

如果UART接口触发RDA中断，那么中断服务程序只需要连续读取UART接收器缓存寄存器UnRBR即可。例如：已将UART0 设置为 8 字节触发，那么RDA中断服务程序代码如程序清单 4.26所示。

程序清单 4.26　8 字节 RDA 中断服务程序示例代码

```
……
switch(U0IIR & 0x0f) {
    case 0x04:                                  /*  发生 RDA 中断                       */
        for(i = 0; i < 8; i++){                 /*  连续读取 U0RBR 寄存器 8 次          */
            RcvBuf[ i ] = U0RBR;                /*  接收数据保存到接收缓冲区 RcvBuf 中  */
        }
}
……
```

（2）接收超时（CTI）

LPC2103 的接收 FIFO 有两种中断操作：一种是 RDA 触发中断，另外一种就是超时中断（CTI）。

当接收 FIFO 中的有效数据个数少于触发个数时（注：接收 FIFO 中至少有一个字节），如果长时间没有数据到达，将触发 CTI 中断。这个时间为：3.5 到 4.5 个字符的时间。

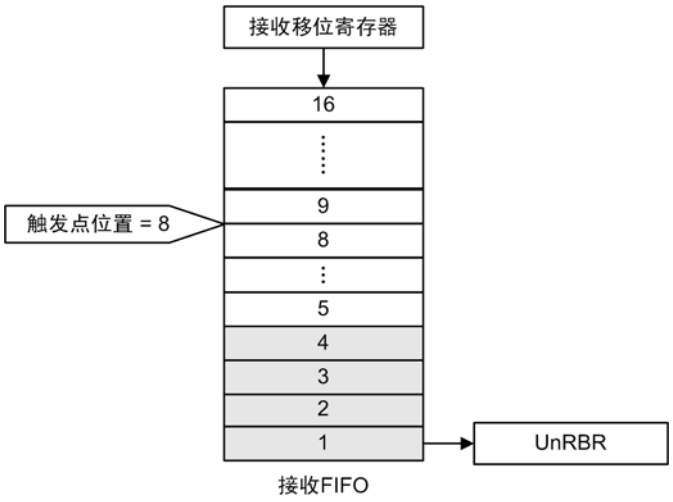


图 4.64　CTI 中断示意图

如图 4.64所示，将接收FIFO的触发点设置为 8 字节，而外部设备只发了 4 个字节的数据，则经过一段时间后（3.5 到 4.5 个字符时间），UART接口便会触发“接收超时中断”。

接收超时中断的初始化设置与RDA中断的设置完全一致，而中断服务程序却有一些区别：对于RDA中断，可以确定当前接收FIFO中的字符数，而CTI中断却不清楚当前的字符个数，只能够通过判断当前接收FIFO是否为空来实现，如程序清单 4.27所示。

程序清单 4.27　CTI 中断服务程序示例代码

```
……
switch(U0IIR & 0x0f) {
    case 0x0c:                                  /* 发生超时中断——CTI                    */
        while((U0LSR & 0x01) == 1) {/* 如果接收 FIFO 中含有数据，就读取 UnRBR 寄存器  */
            RcvBuf [ i++ ] = U0RBR;             /* 将数据保存到接收缓冲区 RcvBuf 中     */
        }
        break;
    ……
}
……
```

2. 发送中断（THRE）

LPC2103 UART 接口具有 16 字节的发送 FIFO，当发送 FIFO 由非空变为空时，便会触发“发送中断”。

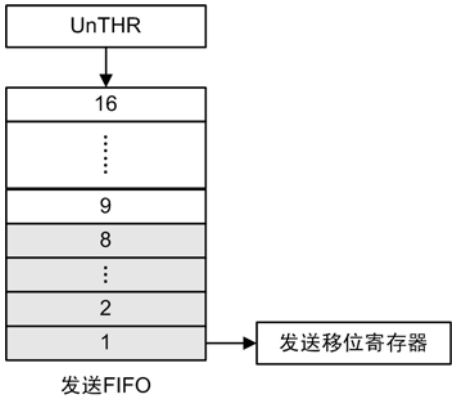


图 4.65　发送中断示意图

如图 4.65所示，UART发送数据时，首先都是将数据送入“发送FIFO”。现在连续将 8 个字节的数据送入“发送FIFO”，则当FIFO变为空时，会触发“THRE”中断。

发生“THRE”中断后，读取 UART 中断标志寄存器（UnIIR）可以清除 THRE 中断标志位。此外，对 UART 发送器保持寄存器 UnTHR 执行写操作也可以清除 THRE 中断。

3. 接收状态中断

在 UART 接收数据时，如果出现溢出错误（OE）、奇偶错误（PE）、帧错误（FE）和间隔中断（BI）中的任意一个错误时，都会触发接收状态中断。具体的错误标志可以通过读取 UART 状态寄存器 UnLSR[4:1]得到。当读取 UnLSR 寄存器时，会清除该中断标志。

4. Modem中断

UART1 接口具有 Modem 中断，当 Modem 输入引脚 DCD、DSR 或 CTS 上发生状态变化时，都会触发 Modem 中断。此外，Modem 输入引脚 RI 上低到高电平的跳变也会产生一个 Modem 中断。Modem 中断源可通过检查 U1MSR[3:0]得到，读取 U1MSR 可以清除 Modem 中断标志。

4.9.7 应用示例

1. UART通信——查询方式

实验程序才用查询方式，通过串口 0 接收上位机发送的字符串，如“Hello EasyARM2103!”，然后送回上位机终端EasyARM-C.exe的数据接收窗口进行显示，实验例程如程序清单 4.28所示。

串口 0 的通信实验需要短接 JP6 的 P0.0 和 P0.1 引脚。

注意：此示例程序定义串口接收 20 个字节数据，所以发送字符串“Hello EasyARM2103!”，需要同时发送双引号，共 20 字符。用户可根据需要自行更改接收字节个数及接收缓冲区大小。

程序清单 4.28 UART0 查询方式通信实验

```
#include "config.h"
#include "string.h"

# define      UART_BPS      115200                                  /*  串口通信波特率              */
/*********************************************************************************************
**  函数名称：DelayNS
**  函数功能：延时函数
**  输入参数：uiDly    值越大，延时时间越长
**  输出参数：无
*********************************************************************************************/
void DelayNS (uint32 uiDly)
{
    uint32 i;
    for (; uiDly > 0; uiDly--){
        for(i = 0; i < 50000; i++);
    }
}
/*********************************************************************************************
**  函数名称：UARTInit
**  函数功能：串口初始化，设置为 8 位数据位，1 位停止位，无奇偶校验，波特率为 9600
**  输入参数：uiDly    值越大，延时时间越长
**  输出参数：无
*********************************************************************************************/
void UARTInit (void)
{
    uint16 uiFdiv;
    U0LCR    = 0x83;                                    /*  允许设置波特率              */
    uiFdiv   = (Fpclk / 16) / UART_BPS;                 /*  设置波特率                  */
    U0DLM    = uiFdiv / 256;
    U0DLL    = uiFdiv % 256;
    U0LCR    = 0x03;                                    /*  锁定波特率                  */
}
/*********************************************************************************************
**  函数名称：UART0GetByte
```

```
** 函数功能：从串口接收 1 字节数据，使用查询方式接收
** 输入参数：无
** 输出参数：无
** 返回参数：uiRcvData    接收到的数据
*********************************************************************************************/
uint8 UART0GetByte (void)
{
    uint8 uiRcvData;

    while ((U0LSR & 0x01) == 0);                          /*  等待接收标志置位              */
    uiRcvData = U0RBR;                                    /*  读取数据                      */
    return (uiRcvData);
}
/*********************************************************************************************
** 函数名称：UART0GetStr
** 函数功能：串口接收字符串
** 输入参数：uiStr      指向接收数据数组的指针
**           uiNum     接收数据的个数
** 输出参数：无
*********************************************************************************************/
void UART0GetStr(uint8 *uiStr, uint32 uiNum)
{
    for (; uiNum > 0; uiNum--){
        *uiStr++ =   UART0GetByte ();
    }
}
/*********************************************************************************************
** 函数名称：UART0SendByte
** 函数功能：向串口发送子节数据，并等待数据发送完成，使用查询方式
** 输入参数：uiDat    要发送的数据
** 输出参数：无
*********************************************************************************************/
void UART0SendByte (uint8 uiDat)
{
    U0THR = uiDat;                                        /*  写入数据                      */
    while ((U0LSR & 0x40) == 0);                          /*  等待数据发送完毕              */
}
/*********************************************************************************************
** 函数名称：UART0SendStr
** 函数功能：向串口发送字符串
** 输入参数：uiStr    要发送的字符串指针
**输出参数：无
*********************************************************************************************/
void UART0SendStr(uint8   const *uiStr)
```

```
{
    while (1){
        if (*uiStr == '\0')break;                          /*  遇到结束符，退出              */
        UART0SendByte (*uiStr++);
    }
}
/*********************************************************************************************
** 函数名称：main
** 函数功能：跳线 JP4 短接，LED1 闪烁
** 输入参数：无
** 输出参数：无
*********************************************************************************************/
int main (void)
{
    uint8    uiBuf[30] = {0};

    PINSEL0 = PINSEL0 & (~0x0F);
    PINSEL0 = PINSEL0 | 0x05;                              /*  设置 I/O 连接到 UART          */

    UARTInit ();                                           /*  串口初始化                    */
    UART0GetStr(uiBuf, 20);                                /*  从串口接收字符串              */
    DelayNS(10);
    UART0SendStr (uiBuf);                                  /*  向串口发送字符串              */
    DelayNS(10);

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

2. UART通信——中断方式

本示例介绍中断方式的串口通信，若使用查询方式，CPU必须不停地查询相应得标志位，占用CPU很多时间，效率较低。而采用中断方式进行通信，则可以避免这些问题。UART中断方式通信实验程序如程序清单 4.29所示。

当中断服务处理子程序接收到 1 次中断，它仅仅能知道的是 UART 产生了中断，至于是什么中断，还需要查询中断标志寄存器 UnIIR，依据不同的中断源类型，进行不同的处理。

程序清单 4.29 UART 中断方式通信实验程序

```
# include "config.h"
# define    UART_BPS      115200                         /*  串口通信波特率                */

volatile    uint8 uiGRcvNew;                             /*  串口接收新数据的标志          */
uint8       uiGRcvBuf[30] = {0};                         /*  串口接收数据缓冲区            */
uint32      uiGNum;                                      /*  串口接收数据的个数            */
/*********************************************************************************************
```

```
** 函数名称：DelayNS
** 函数功能：延时函数
** 输入参数：uiDly    值越大，延时时间越长
** 输出参数：无
*********************************************************************************************/
void DelayNS (uint32 uiDly)
{
    uint32 i;
    for (; uiDly > 0; uiDly--){
        for(i = 0; i < 50000; i++);
    }
}
/*********************************************************************************************
** 函数名称：UART0_IRQ
** 函数功能：串口中断服务函数
** 输入参数：无
** 输出参数：无
*********************************************************************************************/
void __irq UART0_IRQ (void)
{
    uiGNum = 0;

    while ((U0IIR & 0x01) == 0){                          /* 判断是否有中断挂起          */
        switch (U0IIR & 0x0E){                            /* 判断中断标志                */

            case 0x04:                                    /* 接收数据中断                */
                uiGRcvNew = 1;                            /* 置接收新数据标志            */
                for (uiGNum = 0; uiGNum < 8; uiGNum++){   /* 连续接收 8 个字节           */
                uiGRcvBuf[uiGNum] = U0RBR;
                }
                break;

            case 0x0C:                                    /* 字符超时中断                */
                uiGRcvNew = 1;
                while ((U0LSR & 0x01) == 0x01){           /* 判断数据是否接收完毕        */
                    uiGRcvBuf[uiGNum] = U0RBR;
                    uiGNum++;
                }
                break;
            default:
                break;
        }
    }
  VICVectAddr = 0x00;
```

```
}
/*********************************************************************************************
** 函数名称：UARTInit
** 函数功能：串口初始化，设置为 8 位数据位，1 位停止位，无奇偶校验，波特率为 115200
** 输入函数：uiDly    值越大，延时时间越长
** 输出函数：无
*********************************************************************************************/
void UARTInit (void)
{
    uint16 uiFdiv;

    U0LCR     = 0x83;                                   /*  允许设置波特率              */
    uiFdiv    = (Fpclk / 16) / UART_BPS;                /*  设置波特率                  */
    U0DLM     = uiFdiv / 256;
    U0DLL     = uiFdiv % 256;
    U0LCR     = 0x03;                                   /*  锁定波特率                  */
}
/*********************************************************************************************
** 函数名称：UART0SendByte
** 函数功能：向串口发送子节数据，并等待数据发送完成，使用查询方式
** 输入参数：uiDat    要发送的数据
** 输出参数：    无
*********************************************************************************************/
void UART0SendByte (uint8 uiDat)
{
    U0THR = uiDat;                                      /*  写入数据                    */
    while ((U0LSR & 0x20) == 0);                        /*  等待数据发送完毕            */
}
/*********************************************************************************************
** 函数名称：UART0SendStr
** 函数功能：向串口发送字符串
** 输入参数：uiStr       要发送的字符串指针
**           uiNum     要发送的数据个数
** 输出参数：无
*********************************************************************************************/
void UART0SendStr(uint8 const *uiStr, uint32 uiNum)
{
    uint32 i;

    for (i = 0; i < uiNum; i++){                        /*  发送指定个字节数据          */
        UART0SendByte (*uiStr++);
    }
}
/*********************************************************************************************
```

```
** 函数名称：main
** 函数功能：跳线 JP6 短接，打开串口调试软件,串口 0 中断方式通信
** 输入参数：无
** 输出参数：无
*********************************************************************************************/
int main (void)
{

    PINSEL0    = PINSEL0 & (~0x0F);
    PINSEL0    = PINSEL0 | 0x05;                    /* 设置 I/O 连接到 UART              */

    uiGRcvNew = 0;

    UARTInit ();                                    /* 串口初始化                        */
    U0FCR      = 0x81;                              /* 使能 FIFO，设置 8 个字节触发点    */
    U0IER      = 0x01;                              /* 使能接收中断                      */

    IRQEnable ();

    VICIntSelect      = 0x00000000;                 /* 设置所有中断为向量中断            */
    VICVectCntl0      = 0x20 | 0x06;                /* 设置串口中断为最高优先级          */
    VICVectAddr0      = (uint32)UART0_IRQ;          /* 设置向量地址                      */
    VICIntEnable      = 1 << 0x06;                  /* 使能串口中断                      */

    while (1){
        if (uiGRcvNew == 1){                        /* 判断是否有新数据                  */
            uiGRcvNew = 0;                          /* 清除标志                          */
            UART0SendStr (uiGRcvBuf, uiGNum);       /* 向串口发送数据                    */
        }
    }
    return 0;
}
```

本实验例程用到了接收中断和超时中断，用户可根据发送字节数多少，对 GuiDataBuf[ ] 数据缓冲区进行相应的设置。

4.10 A/D转换器

4.10.1 概述

A/D 转换器的基本时钟由 VPB 时钟提供，可编程分频器可将时钟调整至 4.5MHz（逐步逼近转换的最大时钟），10 位精度要求的转换需要 11 个 A/D 转换时钟。

4.10.2 特性

- 10 位逐次逼近式模数转换器；
- 测量范围：0～3.3V；
- 10 位转换时间≥2.44us；
- 一路或多路输入的 Burst 转换模式；
- 转换触发信号可选择：输入引脚的跳变或定时器的匹配；
- 具有掉电模式。

4.10.3 引脚描述

A/D引脚描述见表 4.107。

表 4.107 ADC 管脚描述

AD 管脚	管脚名称	类型	管脚描述
AIN0	P0.22	输入	模拟输入。A/D 转换器单元可测量输入信号的电压。注意：这些模拟输入通常连接到管脚上，即使管脚功能选择寄存器将它们设定为端口管脚。通过将这些管脚驱动成端口输出来实现 A/D 转换器的简单自测 注：当使用 A/D 转换器时，模拟输入管脚的信号电平在任何时候都不能大于 V_{3A}，否则，读出的 A/D 值无效。如果在应用中未使用 A/D 转换器，则 A/D 输入管脚用作可承受 5V 电压的数字 I/O 口 警告：当 ADC 管脚指定为 5V 电压时，ADC 模块中的模拟复用功能不可用。3.3V(V_{DDA})+10%以上的电压不应该应用到选择作为 ADC 输入的任何管脚，否则 ADC 读操作将不正确。例如，若 AD0.0 和 AD0.1 作为 ADC0 输入，AD0.0=4.5V 而 AD0.1=2.5V，AD0.0 上过量的电压会造成 AD0.1 的错误读操作，尽管 AD0.1 输入电压在正确的范围以内
AIN1	P0.23		
AIN2	P0.24		
AIN3	P0.10		
AIN4	P0.11		
AIN5	P0.12		
AIN6	P0.25		
AIN7	P0.26		
V_{DDA}		参考电压	参考电压。该管脚为 A/D 转换器提供参考电压电平
V_{DDA},V_{SSA}		电源	模拟电源和地。它们分别与标称为 V_{DD} 和 V_{SS} 的电压相同，但为了降低噪声和出错几率，两者应当隔离

4.10.4 寄存器描述

A/D转换器寄存器如表 4.108所列。

表 4.108 ADC 寄存器

通用名称	描述	访问	复位值[1]	AD0 地址& 名称
ADCR	A/D 控制寄存器： A/D 转换开始前，必须写入 ADCR 寄存器来选择工作模式	R/W	0x0000 0001	0xE003 4000 AD0CR
ADGDR	A/D 全局数据寄存器：该寄存器包含 ADC 的 DONE 位和最当前 A/D 转换的结果	R/W	NA	0xE003 4004 AD0GDR

续上表

通用名称	描述	访问	复位值[1]	AD0 地址& 名称
ADSTAT	A/D 状态寄存器： 该寄存器包括所有 A/D 通道的 DONE 和 OVERRUN 标志，以及 A/D 中断标志	RO	0x0000 0000	0xE003 4030 AD0STAT
ADINTEN	A/D 中断使能寄存器： 该寄存器含有使能位，这些使能位允许每个 A/D 通道的 DONE 标志是否产生 A/D 中断	R/W	0x0000 0100	0xE003 400C AD0INTEN
ADDR0	A/D 通道 0 数据寄存器： 该寄存器包括在通道 0 完成的最当前的转换结果	RO	NA	0xE003 4010 AD0DR0
ADDR1	A/D 通道 1 数据寄存器： 该寄存器包括在通道 1 完成的最当前的转换结果	RO	NA	0xE003 4014 AD0DR1
ADDR2	A/D 通道 2 数据寄存器： 该寄存器包括在通道 2 完成的最当前的转换结果	RO	NA	0xE003 4018 AD0DR2
ADDR3	A/D 通道 3 数据寄存器： 该寄存器包括在通道 3 完成的最当前的转换结果	RO	NA	0xE003 401C AD0DR3
ADDR4	A/D 通道 4 数据寄存器： 该寄存器包括在通道 4 完成的最当前的转换结果	RO	NA	0xE003 4020 AD0DR4
ADDR5	A/D 通道 5 数据寄存器：该寄存器包括在通道 5 完成的最当前的转换结果	RO	NA	0xE003 4024 AD0DR5
ADDR6	A/D 通道 6 数据寄存器： 该寄存器包括在通道 6 完成的最当前的转换结果	RO	NA	0xE003 4028 AD0DR6
ADDR7	A/D 通道 7 数据寄存器： 该寄存器包括在通道 7 完成的最当前的转换结果	RO	NA	0xE003 402C AD0DR7

[1]：复位值仅指已使用位中保存的数据。它不包括保留位的内容。

1. A/D控制寄存器（ADCR – 0xE003 4000）

ADCR寄存器描述见表 4.109。

表 4.109 A/D 控制寄存器

ADCR	名称	描述	复位值
7:0	SEL	在 AIN3～AIN0(LPC2114/2124)或 AIN7～AIN0(LPC2212/2214)中选择采样引脚。SEL 段中的 bit0～bit7 分别对应 AIN0～AIN7 引脚，为 1 表示选中。在 64 脚封装的 LPC2114/2124 中只有 bit3～bit0 可置位 ● 软件控制模式下，只有一位可被置位 ● 硬件扫描模式下，SEL 可为 1～0xFF 中的任何一个值（在 64 脚封装的 LPC2114/2124 中，SEL 从 1～0x0F 中取值）。SEL 为 0 时，等效于为 0x01	0x01
15:8	CLKDIV	A/D 转换时钟是通过对 VPB 时钟进行分频得到的，A/D 转换时钟的最大值为 4. 5MHz，**A/D 转换时钟 = PCLK/（CLKDIV + 1）**	0

续上表

ADCR	名称	描述	复位值
16	BURST	如果该位为 0，转换由软件控制，需要 11 个时钟方能完成。如果该位为 1，A/D 转换器以 CLKS 字段选择的速率重复执行转换，并从 SEL 字段中为 1 的位对应的引脚开始扫描。A/D 转换器启动后，首先采样的 AIN 通道是编号低的通道（由 SEL 选中），然后是编号高的通道。例：SEL 选中了 AIN1、AIN3、AIN5，那么 A/D 转换器的采样顺序是：AIN1、AIN3、AIN5。 重复转换通过清零该位终止，但该位被清零时并不会中止正在进行的转换	0
19:17	CLKS	该字段用来选择 Burst 模式下每次转换使用的时钟数和 A/D 转换结果的有效位数。CLKS 可在 11 个时钟（10 位）～4 个时钟（3 位）之间选择： ● 000：11 个时钟，10 位 ● 001：10 个时钟，9 位 ● 010：9 个时钟，8 位 ● 011：8 个时钟，7 位 ● 100：7 个时钟，6 位 ● 101：6 个时钟，5 位 ● 110：5 个时钟，4 位 ● 111：4 个时钟，3 位	000
21	PDN	**1**：A/D 转换器处于正常工作模式，**0**：A/D 转换器处于掉电模式	0
23:22	TEST1:0	这些位用于器件测试。00＝正常模式，01＝数字测试模式，10＝DAC 测试模式，11＝一次转换测试模式	0
26:24	START	当 BURST 为 0 时，这些位控制着 A/D 转换的启动时间： ● 000：不启动（PDN 清零时使用该值） ● 001：立即启动转换 ● 010：当 ADCR 寄存器 bit27 选择的边沿出现在 P0.16/EINT0/ MAT0.2/CAP0.2 脚时启动转换 ● 011：当 ADCR 寄存器 bit27 选择的边沿出现在 P0.22/CAP0.0/ MAT0.0 脚时启动转换 注意：START 选择 100-111 时，MAT 信号不必输出到引脚上 ● 100：当 ADCR 寄存器 bit27 选择的边沿在 MAT0.1 出现时启动转换 ● 101：当 ADCR 寄存器 bit27 选择的边沿在 MAT0.3 出现时启动转换 ● 110：当 ADCR 寄存器 bit27 选择的边沿在 MAT1.0 出现时启动转换 ● 111：当 ADCR 寄存器 bit27 选择的边沿在 MAT1.1 出现时启动转换	000
27	EDGE	该位只有在 START 字段为 010～111 时有效 ● 0：在所选 CAP/MAT 信号的下降沿启动转换 ● 1：在所选 CAP/MAT 信号的上升沿启动转换	0

ADC 转换时钟分频值计算：

$$\mathrm{CLKDIV} = \frac{\mathrm{Fpclk}}{\mathrm{Fadclk}} - 1$$

注意：Fadclk 为所要设置的 ADC 时钟，其值不能大于 4.5MHz。

如程序清单 4.30所示，使用AIN0 进行 10 位ADC转换的初始化程序，转换时钟设置为1MHz。

程序清单 4.30 AIN0 初始化示例

```
PINSEL1 = 0x00400000;                    /* 设置 P0.27 为 AIN0 功能                    */
ADCR =   (1 << 0) |                      /* SEL = 1 ，选择通道 0                       */
         ((Fpclk / 1000000 - 1) << 8) |  /* CLKDIV = Fpclk / 1000000 - 1，转换时钟为 1MHz */
         (0 << 16) |                     /* BURST = 0 ，软件控制转换操作               */
         (0 << 17) |                     /* CLKS = 0 ，使用 11clock 转换               */
         (1 << 21) |                     /* PDN = 1 ， 正常工作模式(非掉电转换模式)    */
         (0 << 22) |                     /* TEST1:0 = 00 ，正常工作模式(非测试模式)    */
         (1 << 24) |                     /* START = 1 ，直接启动 ADC 转换              */
         (0 << 27);                      /* EDGE = 0 (CAP/MAT 引脚下降沿触发 ADC 转换) */
```

2. A/D全局数据寄存器（AD0GDR - 0xE003 4004）

A/D全局数据寄存器包含最近一次A/D转换的结果。其中包含数据、DONE和OVERRUN标志以及与数据相关的A/D通道的数目，该寄存器的各个位功能描述如表 4.110所列。

表 4.110 A/D 全局数据寄存器位的描述

位	符号	描述	复位值
5：0	未使用	这些位读出时为 0。专门用于未来的扩展和分辨率更高的 A/D 转换器	NA
15：6	V/V_{REF}	当 DONE 为 1 时，该字段为 ADC 转换结果（二进制形式），表示 SEL 字段选中的 A_{IN} 脚的电压。该字段为 0 表明 A_{IN} 脚的电压小于，等于或接近于 V_{SSA}；该字段为 0x3FF 表明 A_{IN} 脚的电压接近于，等于或大于 V_{REF} 上的电压	NA
23：16	未使用	这些位读出时为 0	NA
26：24	CHN	这些位包含的是 A/D 转换通道	NA
29：27	-	这些位读出为 0。它们用于未来 CHN 字段的扩展，使之兼容可以转换更多通道的 A/D 转换器。现在未使用	NA
30	OVERRUN	Burst 模式下，如果产生 LS 位结果的转换前的一个或多个转换结果丢失或覆盖，该位置位。在非 FIFO 操作中，该位通过读 ADDR 寄存器清零	0
31	DONE	A/D 转换结束时该位置位。该位在 ADDR 被读出和 ADCR 被写入时清零。如果 AD0CR 在转换过程中被写入，该位置位，启动一次新的转换	0

操作示例：

读取全局数据寄存器的值，并提取 10 位的 A/D 转换结果：

```
uint32   ADC_Data;
ADC_Data = AD0GDR;                       /* 读取全局数据寄存器的值                     */
ADC_Data = (ADC_Data >> 6) & 0x3ff;      /* 提取 10 位的 A/D 转换结果                  */
```

3. A/D状态寄存器（AD0STAT - 0xE003 40008）

A/D状态寄存器允许同时检查所有A/D通道的状态。每个A/D通道的ADDRn寄存器的DONE和OVERRUN标志都反映在AD0STAT中。在AD0STAT中同样可以找到中断标志（所有DONE标志逻辑或的结果）。该寄存器的各个位功能如表 4.111所列。

表 4.111 A/D 状态寄存器位的描述

位	符号	描述	复位值
7：0	Done7：0	这些位反映了每个 A/D 通道的结果寄存器中的 DONE 状态标志	0
15：8	OVERRUN7:0	这些位反映了每个 A/D 通道的结果寄存器中的 OVERRUN 状态标志。对 ADSTAT 进行读操作可以同时检查所有 A/D 通道的状态	0
16	ADINT	该位是 A/D 中断标志。当使能 A/D 产生中断时（通过 ADINTEN 寄存器设置）任何一条 A/D 通道的 Done 标记被置 1，该位为 1	0
31：17	-	未使用，始终为 0	0

一般启动了一次 A/D 转换后，需要使用状态寄存器的低 8 位来判断是否转换结束，比如通道 0 转换结束，则 AD0STAT 的第 0 位 Done0 会置 1；而第 5 通道转换结束，则 AD0STAT 的第 5 位 Done5 会置 1。并且读取了相应的结果寄存器 ADDRx 后会把 AD0STAT 低八位中相应的 Done 位清零。以下示例是对第五通道 AIN5 进行转换，并等待转换结束的代码。

```
AD0CR |= 1 << 24;                          /*  进行第一次转换          */
while ((AD0STAT & (1<<5)) == 0);           /*  等待 AIN5 转换结束      */
……                                         /*  接下来读取有效转换结果  */
```

4. A/D中断使能寄存器（AD0INTEN - 0xE003 400C）

该寄存器用来控制转换完成时哪个A/D通道产生中断。其各位功能如表 4.112所列。

表 4.112 A/D 中断使能寄存器位的描述

位	符号	描述	复位值
7：0	ADINTEN 7:0	转换完成时，这些位用来控制哪个 A/D 通道在转换结束时产生中断。如果 bit0 为 1，则当 A/D 通道 0 的转换结束时产生中断，如果 bit1 等于 1，则当 A/D 通道 1 的转换结束时产生中断，如此类推	0x00
8	ADGINTEN	为 1 时，使能 ADDR 的全局 DONE 标志产生中断。为 0 时，只有个别由 ADINTEN7:0 使能的 A/D 通道才产生中断	1
31：9	-	未使用，始终为 0	0

然而，在实际应用中 A/D 转换一般都不使用中断方式。比如，当需要通过对 A/D 通道转换来监控传感器时，A/D 通道转换结束时就无需产生中断，只须通过查询方式即可随时通过程序读取转换结果。

5. A/D数据寄存器（ADDR0～ADDR7 - 0xE003 4010～0xE003 402C）

A/D数据寄存器保存着A/D转换的结果，同时包含指示转换结束以及转换溢出的标志。此寄存器的位功能如表 4. 113所列。

表 4.113 A/D 数据寄存器位的描述

位	符号	描述	复位值
5：0	-	未使用，始终为 0。用于未来的扩展和分辨率更高的 A/D 转换器	0
15：6	V/V_{REF}	当 DONE 为 1 时，该字段为 ADC 的转换结果（二进制形式），表示 SEL 字段选中的 A_{IN} 脚的电压。该字段为 0 表明 A_{IN} 脚的电压小于，等于或接近于 V_{REF}；该字段为 0x3FF 表明 A_{IN} 脚的电压接近于，等于或大于 V_{REF} 上的电压	NA

续上表

位	符号	描述	复位值
29：16	-	未使用。这些位读出时总是为 0。这些位暂时没有使用到	0
30	OVERRUN	Burst 模式下，如果产生 LS 位结果的转换前一个或多个转换结果丢失或覆盖，该位置位。通过读该寄存器，该位可清零	0
31	DONE	A/D 转换结束时该位置位。此位在该寄存器被读出时清零	0

操作示例：

读取数据寄存器的值，并提取 10 位的 A/D 转换结果。

```
uint32 ADC_Data;
while ((AD0STAT & (1<<5)) == 0);          /* 等待 AIN5 转换结束          */
ADC_Data = ADDR5;                         /* 读取第五通道的数据寄存器的值 */
ADC_Data = (ADC_Data >> 6) & 0x3ff;       /* 提取 10 位的 A/D 转换结果    */
```

4.10.5 操作

1. 硬件触发转换

如果 ADCR 的 BURST 位为 0 且 START 字段的值包含在 010-111 之内，当所选引脚或定时器匹配信号发生跳变时，A/D 转换器启动一次转换。

2. 时钟产生

用于产生 A/D 转换时钟的分频器在 A/D 转换器空闲时保持复位状态，在下面场合会立即启动采样时钟：

- 当 ADCR 的 START 字段被写入 001；
- 所选引脚或匹配信号发生触发。

这个特性可以节省功率，尤其适用于 A/D 转换不频繁的场合。

3. 中断

当 DONE 位为 1 时，将对向量中断控制器（VIC）发送中断请求。通过软件设置 VIC 中 A/D 转换器的中断使能位来控制是否产生中断。DONE 在读 ADDR 操作时清零。

4. 精度和引脚设置

当A/D转换器用来测量Ain脚的电压时，并不理会引脚在引脚选择寄存器中的设置，但是通过选择Ain功能(即禁能引脚的数字功能)，可以提高转换精度。如图 4.66所示，以引脚 P0.22 为例，即使P0.27 引脚设置为GPIO功能，可是，P0.22 引脚还是连接到模拟输入引脚 AIN0 上的。但是，此时P0.22 引脚是数字引脚，因此，输入到A/D引脚的电压只有高低电平，即，3.3V和 0V。

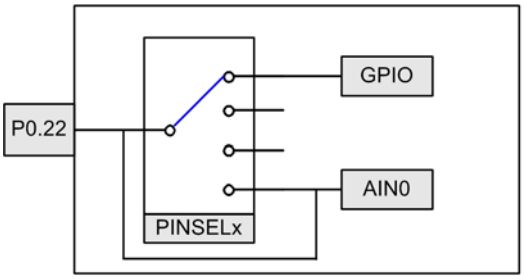


图 4.66 AIN0 引脚示意图

5. ADC使用方法

使用 ADC 模块时，先要将测量通道引脚设置为 AINx 功能，然后通过 ADCR 寄存器设置 ADC 的工作模式、ADC 转换通道、转换时钟(CLKDIV 时钟分频值)、并启动 ADC 转换。可以通过查询或中断的方式等待 ADC 转换完毕，转换数据保存在 ADDR 寄存器中。

A/D 没有独立的的参考电压引脚，A/D 的参考电压与供电电压连接在一起——3.3V。假定从 ADDR 寄存器中读取到的 10 位 A/D 转换结果为 VALUE，则对应的实际电压为：

$$U = \frac{VALUE}{1024} \times 3.3V$$

4.10.6 ADC中断

ADC中断与向量中断控制器（VIC）的关系如图 4.67所示。

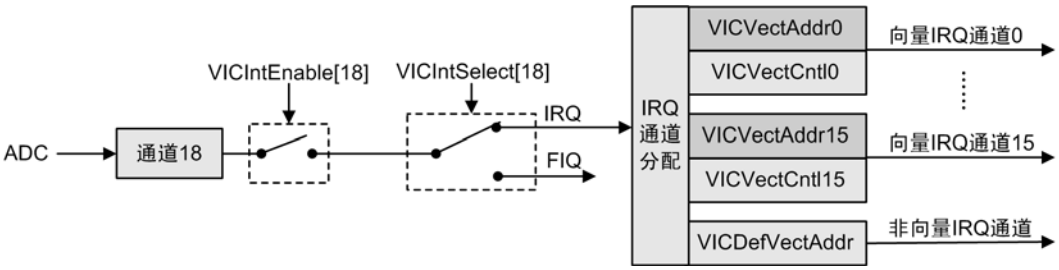


图 4.67 ADC 接口与 VIC 的关系

当ADC转换结束后会触发中断，ADC处于VIC的通道18，中断使能寄存器VICIntEnable用来控制 VIC 通道的中断使能。当 VICIntEnable[18] = 1 时，通道 18 中断使能，即：ADC 中断使能。

中断选择寄存器 VICIntSelect 用来分配 VIC 通道的中断。当某一位为“1”时，对应的通道中断分配为 FIQ；当某一位为“0”时，对应的通道中断分配为 IRQ。VICIntSelect[18] 用来控制通道 18，即：

- 当 VICIntSelect[18] = 1 时，ADC 中断分配为 FIQ 中断；
- 当 VICIntSelect[18] = 0 时，ADC 中断分配为 IRQ 中断。

当分配为 IRQ 时，还需要设置对应的通道控制寄存器和地址寄存器。有关寄存器 VICVectCntln 和 VICVectAddrn 的说明，请参考“向量中断控制器（VIC）”一节。

图 4.68 ADC 中断示意图

如图 4.68所示，无论是通过软件还是硬件启动ADC，ADC转换结束以后，都会置位DONE位，此时ADC会向VIC发送中断有效信号。

4.10.7 应用示例

1. 软件触发ADC转换

LPC2103 内部 A/D 最简单的使用方法是软件启动工作方式，当需要采样时才通过软件

启动 A/D 采样或者通过定时器定时的方式启动 A/D。用户可以通过使能 A/D 转换结束中断或者查询中断标志位的方式来得知 A/D 结束转换，从而读取转换结果。

为了能够在PC机上显示出A/D采样值，则需要用串口延长线将目标板的串口 0 和PC的串口相连，并且用短路器将目标板JP2 跳线器上的P0.0、P0.1 分别与TXD0、RXD0 短接。目标板和PC机通信的示意图如图 4.69所示。

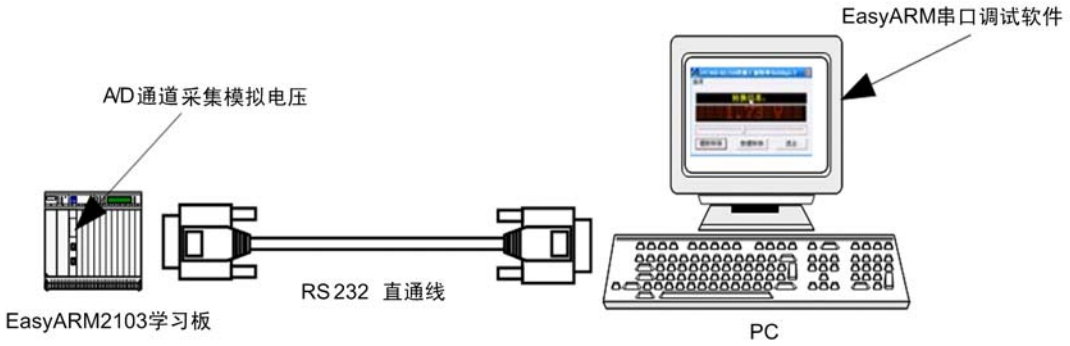


图 4.69 目标板与 PC 机通信示意图

（1）A/D 转换器的初始化

A/D 转换功能用到的寄存器有 AD0CR、AD0DR0、AD0STAT，AD0CR 控制 A/D 转换通道的选择、A/D 转换速度的设定、转换精度的设定、A/D 模式的选择和 A/D 启动方式的设置，其中第 24～26 位控制 ADC 的启动方式。如果使用软件直接启动 A/D，则这 3 个位设为 001，就立即启动一次 A/D 转换。

初始化A/D时，对AD0CR寄存器的设置见程序清单 4.31。

程序清单 4.31 AD0CR 寄存器设置

```
AD0CR =  (1 << 0) |                          /*  选择通道 0                          */
         (((Fpclk / 1000000) - 1) << 8) |    /*  转换时钟为 1MHz                     */
         (0 << 16) |                         /*  BURST = 0,软件控制转换操作          */
         (0 << 17) |                         /*  CLKS = 0,使用 11clock 转            */
         (1 << 21) |                         /*  PDN = 1，正常工作模式               */
         (0 << 22) |                         /*  TEST = 0，正常工作模式              */
         (1 << 24) |                         /*  设置直接启动模式                    */
         (0 << 27 );                         /*  设置模式，直接启动模式下无效        */
```

目标板使用A/D通道 0 进行采样，所以A/D转换的结果存放在AD0DR0 寄存器中。启动了ADC后，通过查询状态寄存器AD0STAT的低 8 位来判断转换是否结束。查询AD0STAT寄存器和读取AD0DR0 的代码见程序清单 4.32。

程序清单 4.32 AD0STAT 查询和读取 ADDR5 寄存器

```
while ((AD0STAT & (1<<0)) == 0)      /* 等待第 0 通道 AIN0 转换结束，当转换结束时 AD0STAT
                                        的第 0 位置 1，程序往下执行；否则一直等      */
ADC_Data = AD0DR0                    /* 读取通道 0 的 ADC 的值                       */
```

A/D转换的初始化代码见程序清单 4.33。

程序清单 4.33 A/D 转换初始化

```
PINSEL1    |= (0x03) << 12;                    /* 设置 P0.22 为 AIN0 功能          */
/* 进行 ADC 模块设置 */
AD0CR = (1 << 0) |                            /*  选择通道 0                      */
        (((Fpclk / 1000000) - 1) << 8) |      /*  转换时钟为 1MHz                 */
        (0 << 16) |                           /*  BURST = 0,软件控制转换操作      */
        (0 << 17) |                           /*  CLKS = 0,使用 11clock 转        */
        (1 << 21) |                           /*  PDN = 1，正常工作模式           */
        (0 << 22) |                           /*  TEST = 0，正常工作模式          */
        (1 << 24) |                           /*  设置直接启动模式                */
        (0 << 27 );                           /*  设置模式，直接启动模式下无效    */
```

在初始化 A/D 功能模块时需要注意以下 3 点：

- 使用 ADC 前必须将功率控制寄存器 PCONP 中的 ADC 功率控制位 bit12 置 1，否则 ADC 是不能正常工作的，并且不要忘了是“或”操作。因为如果没有加上“或”，则会将其它本身已经打开的功率控制位关闭，导致其它的功能部件不能工作，这是很多初学者易犯的错误；
- 转换时间是用户比较关心的参数。如果要保证 10bit 的转换精度，则转换的时间就需要 11 个 A/D 转换时钟（clock），并且一个 clock 最快不能大于 4.5MHz，这样要完成一次 10bit 的 A/D 转换至少需要 11/4.5MHz，即 2.44us。当然如果用户不需要这么高的精度，则可以通过减少转换的 clock 的个数来达到提高转换速度的目的。

（2）等待转换结束和数据处理

查询A/D转换结束和计算A/D转换结果的代码如程序清单 4.34。

程序清单 4.34 等待转换结束和计算结果

```
AD0CR |= 1 << 24;                             /* 进行第一次转换                   */
while ((AD0STAT & (1<<0)) == 0);              /* 等待第 0 通道 AIN0 转换结束      */
AD0CR |= 1 << 24;                             /* 再次启动转换                     */
while ((AD0STAT & (1<<0)) == 0);              /* 等待第 0 通道 AIN0 转换结束      */
DelayNS(10);                                  /*  至少等待 11 个 AD 转换时钟后才读取数据 */

ADC_Data = ADDR0;                             /* 读取 ADC 结果                    */
ADC_Data = (ADC_Data >> 6) & 0x3ff;
ADC_Data = ADC_Data * 3300;                   /* 参考电压由精密恒压源提供的 3.3V  */
ADC_Data = ADC_Data / 1024;
```

需要说明的 3 点：

- 使用查询方式时，一般查询状态寄存器 AD0STAT 的低 8 个比特来判断转换是否结束，而不采用查询 ADDRx 的 DONE 标志位来判断;
- A/D 转换值计算，LPC2103 内部的 ADC 转换精度是 10bit，因此当通道 0 的结果寄存器 ADDR0 中保存的转换值为 $VALUE_{ADDR0}$ 时，则转换结果 $VALUE_{ADC}$ 的计算方法如下：

$$VALUE_{ADC} = \frac{V_{REF}}{2^{10}} \times VALUE_{ADDR5} = \frac{3300}{1024} \times VALUE_{ADDR5}(mv)$$

如果使用 3.3v 作为基准参考电压源，则 LPC2103 的 A/D 转换器的最小分辨率 LSB

的计算如下：

$$LSB=\frac{V_{REF}}{2^{10}}=\frac{3300}{1024}(mv)=3.22mv$$

即在一次转换中，A/D 转换器能够区分的最小电压为 3.22mv。

- 关于A/D电压计算的问题，编程时要注意使用整数运算，即“先乘后除”，而不采用“先除后乘”。因为程序中定义的ADC_Data是个整型变量，如果采用先除 1024，则得到的结果将会把小数部分去掉，其结果必然和实际的数值有很大的偏差。因此，正确的做法应该是将ADC_Data先乘 3300 后再除 1024，这样得到的结果才和实际电压接近，参见程序清单 4.34。

（3）通过串口 0 发送 A/D 转换结果到 PC 机显示

首先，将得到的 A/D 转换结果转换成字符串格式，通过函数 sprintf 完成，此函数是标准的 C 函数，因此一定要在 config.h 中包含 C 标准输入输出函数 stdio.h，否则，编译出来需要的 ROM（即 flash）代码空间会额外增加很多（约 8KB），这一点对代码空间紧张的使用者来说必须要注意。

然后将字符串通过UART0 发送给PC机，在EasyARM串口调试软件上显示。要在EasyARM串口调试软件上显示数据，需要一个简单的通信协议，此协议请参考EasyARM串口调试软件使用说明。实现此协议的API函数如程序清单 4.35。

程序清单 4.35 EasyARM 串口调试软件通信协议 API 函数

```
/*********************************************************************************************
** 函数名称：PC_DispChar
** 函数功能：向 PC 机的 EasyARM 串口调试软件发送显示字符
** 入口参数：x          显示字符的横坐标
**           y          显示字符的纵坐标
**           chr        显示的字符，不能为 ff
**           color      显示的状态，包括前景色、背景色、闪烁位。
**                      与 DOS 字符显示一样：0～3,前景色，4～6，背景色，7，闪烁位
** 出口参数：无
*********************************************************************************************/
void PC_DispChar (uint8 x, uint8 y, uint8 chr, uint8 color)
{
    UART0_SendByte(0xff);                          /* 起始标志                          */
    UART0_SendByte(x);
    UART0_SendByte(y);
    UART0_SendByte(chr);
    UART0_SendByte(color);
}
```

（4）软件触发 AD 转换示例应用

本示例是软件触发的单通道AD转换实验，首先对通道 0 的电压进行采样，进行AD转换，然后将转换后的电压通过串口发送到上位机终端EasyARM-C.exe的全DOS仿真窗口进行显示。示例代码如程序清单 4.36所示。

注意：串口显示需要调用软件包“UART.h”、“UART.c”。由于程序使用了 sprintf()函数，因而需要添加 stdio.h 头文件。

程序清单 4.36 软件触发 AD 转换

```
# include "config.h"
# include <stdio.h>
# include "UART.H"
/*********************************************************************************************
** 函数名称：DelayNS
** 功能描述：延时函数
** 入口参数：uiDly    值越大，延时时间越长
** 出口参数：无
*********************************************************************************************/
void DelayNS(uint32 uiDly)
{
    uint32 i;
    for(; uiDly > 0; uiDly--){
        for(i=0; i<50000; i++);
    }
}
/*********************************************************************************************
** 函数名称：main
** 功能描述：利用 ADC 对通道 0 的电压进行采样，并把值发送到 PC 机显示出来
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
int main(void)
{
    uint32 uiVal;
    char    cStr[20];

    PINSEL1 = PINSEL1 & (~(0x03 << 12));                    /*  将 P0.22 设置为 AD 功能          */
    PINSEL1 = PINSEL1 | (0x03 << 12);
    UARTInit();                                             /*  串口初始化                       */

   /*  进行 ADC 模块设置*/
    AD0CR = (1 << 0) |                                      /*  选择通道 0                       */
            (((Fpclk / 1000000) - 1) << 8) |                /*  转换时钟为 1MHz                  */
            (0 << 16) |                                     /*  BURST = 0,软件控制转换操作       */
            (0 << 17) |                                     /*  CLKS = 0,使用 11clock 转         */
            (1 << 21) |                                     /*  PDN = 1，正常工作模式            */
            (0 << 22) |                                     /*  TEST = 0，正常工作模式           */
            (1 << 24) |                                     /*  设置直接启动模式                 */
            (0 << 27 );                                     /*  设置模式，直接启动模式下无效     */

    while(1) {
        /*  启动 AD 转换            */
```

```
            AD0CR |= (1 <<24);
            while ((AD0STAT & 0x01) == 0);                  /*  读取 AD0STAT 的通道 0 的 Done */
            AD0CR |= (1 <<24);
            while ((AD0STAT & 0x01) == 0);                  /*  读取 AD0STAT 的通道 0 的 Done */
            DelayNS(10);                                    /*  至少等待 11 个 AD 转换时钟后才
                                                                读取数据否则会造成死等待       */
            uiVal = AD0DR0;
            uiVal = (uiVal >> 6) & 0x3FF;
            uiVal = (uiVal * 3300)/1024;                    /*  ADC 的参考电压为 3300mV       */

            sprintf(cStr,"%4d mV VIN0", uiVal);             /*  格式化显示数据                */
            ISendStr (0,0,0x30,cStr);                       /*  将数据发送到 PC 机显示        */
            DelayNS(50);
        }
        return (0);
}
```

在PC机上打开EasyARM串口调试软件，设置COM口（默认是COM1）、波特率等，如图4.70所示。

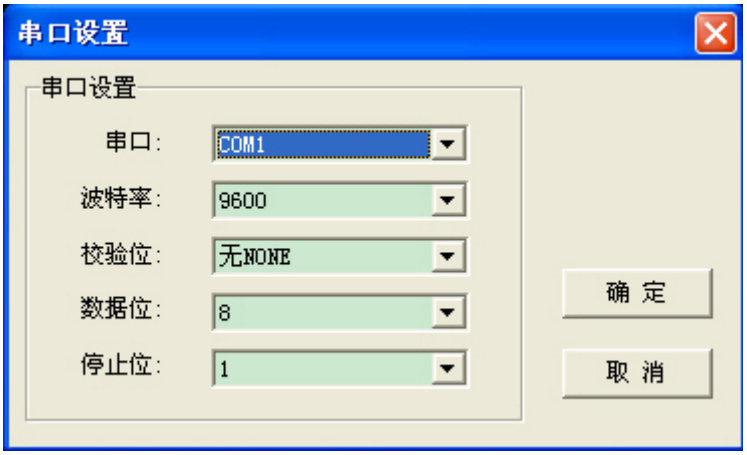


图 4.70　EasyARM 串口调试软件设置

连接好目标板的串口和PC的COM1 口后，编译程序启动AXD调试，全速运行程序，在上位机的EasyARM串口调试软件即可得到如图 4.71所示结果。

图 4.71 单通道 ADC 转换结果

2. P0.16 触发ADC转换

本节介绍 A/D 功能模块外部管脚触发 ADC 转换的方法以及如何使用中断方式读取 A/D 转换结果。LPC2103 可以设置外部管脚 P0.22 的边沿信号来触发 A/D 转换。

当一次 A/D 转换结束，A/D 中断服务函数将通道 0 的结果寄存器 ADDR0 的值读到全局变量 addr0_data 中，并将标志（定义的一个全局变量 ADC_Send_Enable）置 1。CPU 退出 A/D 中断服务函数后，main 函数通过查询标志 ADC_Send_Enable 变量，即可得知 ADC 转换结束，进而进行数据处理和发送数据给上位机。

（1）A/D 转换器的初始化

A/D控制寄存器（AD0CR）中第 24～26 位控制着ADC的启动方式，若要使用P0.22 脚上的边沿信号触发ADC转换，则这三个位设为 010，并且在第 27 位中设置触发信号的极性。初始化ADC的代码见程序清单 4.37。

程序清单 4.37 触发 ADC 转换初始化

```
PCONP   |= 1 <<12;                          /*  必须打开 ADC 的功率控制位 ADC 才可以正常工作*/
PINSEL1 |= (0x03) << 12;                    /*  管脚连接设置 P0.22 为 AIN0 功能              */
AD0CR = (1 << 0) |                          /*  选择通道 0*/
        ((Fpclk / 1000000 - 1) << 8) |      /* CLKDIV=Fpclk/1000000-1,设置转换时钟为 1MHz   */
        (0 << 16) |                         /* BURST=0,软件控制转换操作                     */
        (0 << 17) |                         /* CLKS=0, 使用 11clock 转换                    */
        (1 << 21) |                         /* PDN=1,正常工作模式                           */
        (2 << 24) |       /* START=2,当 EDGE 选择的边沿出现在 P0.22 管脚时启动 ADC 转换 */
        (1 << 27);                          /* P0.22 下降沿触发                             */
```

（2）设置 ADC 中断

采用中断方式处理 ADC 转换结果，需要作如下处理：

- 设置 A/D 中断使能寄存器（AD0INTEN）。AD0INTEN 寄存器的低 8 位控制着哪个 A/D 通道在转换结束时产生中断，bit0 为 1，则 A/D 通道 0 转换结束产生中断，bit1 设为 1，则 A/D 通道 1 转换结束产生中断，以此类推。当使用了 A/D 通道 0（AIN0）转换，采用中断方式，则 AD0INTEN 的 bit0 设为 1，即：

```
AD0INTEN = 1 << 0;                          /*  使能 A/D 通道 0 转换结束后产生中断   */
```

- 进行 VIC 初始化；
- 使用 IRQEnable()打开 IRQ 中断；
- 编写中断服务函数。

完整的中断设置代码如程序清单 4.38所示。

程序清单 4.38 中断设置部分的程序代码

```
AD0INTEN = 1 << 0;                          /*  使能 A/D 通道 0 转换结束后产生中断   */
IRQEnable();                                /*  打开 IRQ 中断                        */

/*  进行 VIC 设置  */
VICIntSelect        = 0x00;                 /*  所有中断通道设置为 IRQ 中断          */
VICVectCntl0        = 0x20 | 18;            /*  设置 ADC 中断最高优先级              */
```

```
VICVectAddr18       = (uint32)IRQ_ADC;      /* 设置中断服务程序地址                    */
VICIntEnable        = 1 << 18;              /* 使能 ADC 中断                           */
```

ADC的中断服务函数代码如程序清单 4.39所示。

程序清单 4.39 ADC 中断服务程序代码

```
void __irq IRQ_ADC(void)
{
    addr0_data      = ADDR0;            /* 读取通道 0 的结果寄存器 ADDR0 数据到全局标量 */
    ADC_Send_Enable = 1;                /* 置位串口发送 ADC 数据的标志              */
    VICVectAddr     = 0x00;             /* 通知 VIC 中断处理结束                    */
}
```

3. MAT触发ADC转换

ADC 转换的硬件触发模式除了外部管脚触发外，还能够通过定时器的定时翻转功能触发 ADC 转换，这样就可以实现无需 CPU 干预就能精确定时触发 ADC 转换，并且无需占用定时器中断。

（1）初始化 ADC 为 MAT 触发转换

若要使用定时器 1 匹配通道 0（MAT1.0）产生的下降沿来启动ADC，则AD0CR的第 24～26 位应设置为 110，第 27 位设为 1，如程序清单 4.40。

程序清单 4.40 MAT1.0 触发 ADC 对 AD0CR 寄存器的设置

```
PCONP   |=1 << 12;                      /* 必须打开 ADC 的功率控制位 ADC 才可以正常工作*/
PINSEL1 |= (0x03) << 12;                /* 管脚连接设置 P0.22 为 AIN0 功能          */
AD0CR   = (1 << 0) |                    /* 选择通道 0                               */
          ((Fpclk / 1000000 - 1) << 8) | /* CLKDIV=Fpclk/1000000-1,设置转换时钟为 1MHz */
          (0 << 16) |                   /* BURST=0,软件控制转换操作                 */
          (0 << 17) |                   /* CLKS=0, 使用 11clock 转换                */
          (1 << 21) |                   /* PDN=1,正常工作模式                       */
          (6 << 24) |   /* START=6,当 EDGE 选择的边沿出现在 MAT1.0 时启动 ADC 转换 */
          (1 << 27);                    /* MAT1.0 下降沿触发                        */
```

（2）设置定时器 1 的 MAT1.0 脚定时翻转

初始化定时器 1，使MAT1.0 每隔 1s输出翻转，并开启定时器。对定时器 1 的设置代码如程序清单 4.41所示。

程序清单 4.41 定时器 1 初始化代码

```
T1MCR = 0x02;                   /* 设置 T1MR0 匹配后将 T1TC 复位                */
T1EMR = 3 << 4;                 /* T1MR0 匹配后 MAT1.0 输出翻转                 */
T1MR0 = Fpclk;                  /* 输出频率周期控制,定时 1s 钟 MAT1.0 翻转一次  */
T1TCR = 0x03;                   /* 复位并启动 T1TC                              */
```

以上代码初始化定时器需要说明两点。

- 定时器 MAT1.0 的下降沿触发启动 A/D 转换不一定需要将 MAT1.0 连接到芯片的管脚；
- 设置定时器 1 每隔 1s 钟 MAT1.0 输出翻转，则 MAT1.0 每隔 2 秒钟产生一个下降

沿，则 A/D 每隔 2 秒钟转换一次，这点注意观察。

4. 突发模式（Burst）ADC转换

ADC 转换的硬件触发模式还有一种 Burst 模式，此模式下，ADC 按照用户设定的采样通道（一路或多路）自动重复地转换。并且，可按照对转换速度的要求设定完成一次 A/D 所需的时钟个数；若设置转换一次所需的时钟个数越少，转换速度就越快，但精度会相应变低。例如，转换一次使用 11 个 clock 会得到一个 10bit 精度的 A/D 转换值，用时 2.44us；而如果转换一次使用 5 个 clock 会得到一个 4bit 精度的转换值，用时只有 1.11μs，可见，转换速度的提高是会相应地降低转换的精度。

要设置ADC转换模式为Burst模式，需要把AD0CR的第 16 位设置为 1，并且AD0CR的第 24～26 位一定要设置为 000，否则ADC是无法启动的，如程序清单 4.42所示。

程序清单 4.42 Burst 模式对 AD0CR 寄存器的设置

```
PCONP |=1 << 12;                          /*  必须打开 ADC 的功率控制位 ADC 才可以正常工作*/
PINSEL1 |= (0x03) << 12;                  /*  管脚连接设置 P0.22 为 AIN0 功能              */
AD0CR = (1 << 0) |                        /*  选择通道 0                                   */
        ((Fpclk / 1000000 - 1) << 8) |    /* CLKDIV=Fpclk/1000000-1,设置转换时钟为 1MHz   */
        (0 << 16) |                       /* BURST=0,软件控制转换操作                     */
        (0 << 17) |                       /* CLKS=0, 使用 11clock 转换                    */
        (1 << 21) |                       /* PDN=1,正常工作模式                           */
        (6 << 24) |      /* START=6,当 EDGE 选择的边沿出现在 MAT1.0 时启动 ADC 转换       */
        (1 << 27);                        /* MAT1.0 下降沿触发                            */
```

程序的其它部分和**软件触发 ADC 转换**的代码都一样，请阅读之前的描述。

4.11 I²C接口

4.11.1 特性

- 标准的 I²C 总线接口，可配置为主机，从机或主/从机；
- 同时发送的主机之间进行仲裁，避免了总线数据的冲突；
- 可编程时钟可实现 I²C 传输速率控制；
- 主机从机之间双向数据传输；
- 串行时钟同步使器件在一条串行总线上实现不同位速率的通信；
- 串行时钟同步可作为握手机制使串行传输挂起和恢复；
- I²C 总线可用于测试和诊断。

4.11.2 引脚描述

LPC2103 中I²C的引脚排列如表 4.114所示。

表 4.114 I²C 的引脚

管脚名称	I²C 管脚	描述
P0.2	SCL0	I²C0 时钟线
P0.3	SDA0	I²C0 数据线
P0.17	SCL1	I²C1 时钟线
P0.18	SDA1	I²C1 数据线

4.11.3 I²C总线规范

1. I²C总线规范简介

I²C BUS（Inter IC BUS）是 NXP 半导体公司推出的芯片间串行传输总线，它以 2 根连线实现了完善的全双工同步数据传送，可以极方便地构成多机系统和外围器件扩展系统。I²C 总线采用了器件地址的硬件设置方法，通过软件寻址完全避免了器件的片选线寻址方法，从而使硬件系统具有最简单而灵活的扩展方法。

I²C 总线的 2 根线（串行数据——SDA，串行时钟——SCL）连接到总线上的任何一个器件，每个器件都应有一个唯一的地址，而且都可以作为一个发送器或接收器。此外，器件在执行数据传输时也可以被看作是主机或从机。

- **发送器：**本次传送中发送数据（不包括地址和命令）到总线的器件；
- **接收器：**本次传送中从总线接收数据（不包括地址和命令）的器件；
- **主　机：**初始化发送、产生时钟信号和终止发送的器件，它可以是发送器或接收器主机通常是微控制器；
- **从　机：**被主机寻址的器件，它可以是发送器或接收器。

I²C总线应用系统的典型结构如图 4.72所示。在该结构中，微控制器A可以做为该总线上的唯一主机，其它的器件全部是从机。而另一种方式是微控制器A和微控制器B都作为总线上的主机。

I²C 总线是一个多主机的总线，即，总线上可以连接多个能控制总线的器件。当 2 个以上控制器件同时发动传输时，只能有一个控制器件能真正控制总线而成为主机，并使报文不被破坏，这个过程叫仲裁。与此同时，能同步多个控制器件所产生的时钟信号。

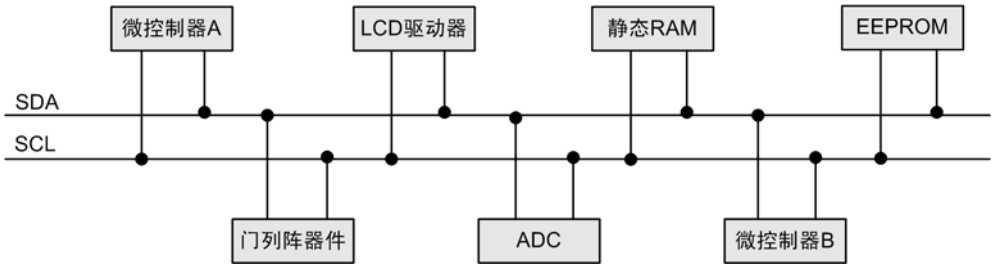


图 4.72 I^2C 总线应用系统典型结构

SDA 和 SCL 都是双向线路。连接到总线的器件的输出端必须是漏极开路或集电极开路，都通过一个电流源或上拉电阻连接到正的电源电压，这样才能够实现“线与”功能。当总线空闲时，这 2 条线路都是高电平。

在标准模式下，总线数据传输的速度为 0～100Kbit/s，在高速模式下，可达 0～400Kbit/s。总线速率与总线上拉电阻的关系：总线速率越高，总线上拉电阻要越小。100Kbit/s 总线速率，通常使用 5.1KΩ的上拉电阻。

2. I^2C总线上的位传输

I^2C 总线上每传输一个数据位必须产生一个时钟脉冲。

（1）数据的有效性

SDA线上的数据必须在时钟线SCL的高电平期间保持稳定，数据线的电平状态只有在SCL线的时钟信号为低电平时才能改变，如图 4.73所示。在标准模式下，高低电平宽度必须不小于 4.7μS。

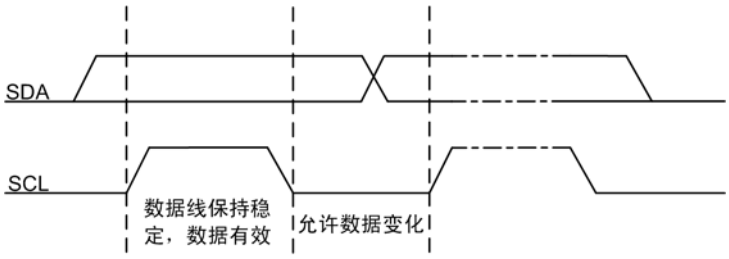


图 4.73 I^2C 总线的位传输

（2）起始和停止信号

在I^2C总线中，唯一违反上述数据有效性的是起始（S）和停止（P）信号，如图 4.74所示。

- **起始信号（重复起始信号）**：在 SCL 为高电平时，SDA 从高电平向低电平切换；
- **停止信号**：在 SCL 为高电平时，SDA 由低电平向高电平切换。

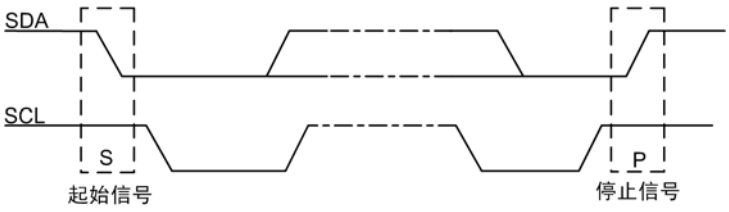


图 4.74 I^2C 总线的起始信号和停止信号

起始和停止信号一般由主机产生。起始信号作为一次传送的开始，在起始信号后总线被

认为处于忙的状态。停止信号作为一次传送的结束，在停止信号的某段时间后，总线被认为再次处于空闲状态。重复起始信号既作为上次传送的结束，也作为下次传送的开始。

3. 数据传输

（1）字节格式

发送到SDA线上的每个字节必须为 8 位。每次传输可以发送的字节数量不受限制。每个字节后必须跟一个应答位。首先传输的是数据的最高位（MSB）（见图 4.75所示）。

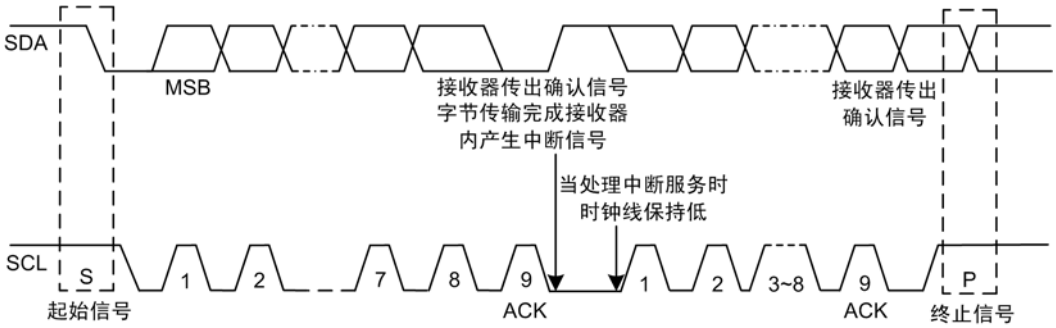


图 4.75 I^2C 总线的数据传输

（2）应答

相应的应答时钟脉冲由主机产生。在应答的时钟脉冲期间，发送器释放SDA线（高）。在应答的时钟脉冲期间，接收器必须将SDA线拉低，使它在这个时钟脉冲的高电平期间保持稳定的低电平。如图 4.75中时钟信号SCL的第 9 位。

一般说来，被寻址匹配的从机（可继续接收下一个字节的接收器）将产生一个应答位。如果作为发送器的主机在发送完一个字节后，没有收到应答位（或收到一个非应答位），或者作为接收器的主机没有发送应答位（或发送一个非应答位），那么主机必须产生一个停止信号或重复起始信号来结束本次传输。

若从机（接收器）不能接收更多的数据字节，将不产生这个应答位；主机（接收器）在接收完最后一个字节后不产生应答，通知从机（发送器）数据传输结束。

4. 仲裁与时钟发生

（1）同步

时钟同步是通过各个能产生时钟的器件线连接到 SCL 线上来实现的，上述的各个器件可能都有自己独立的时钟，各个时钟信号的频率、周期、相位和占空比可能都不相同，由于“线与”的结果，在 SCL 线上产生的实际时钟的低电平宽度由低电平持续时间最长的器件决定，而高电平宽度由高电平持续时间最短的器件决定。

（2）仲裁

当总线空闲时，多个主机同时启动传输，可能会有不止一个主机检测到满足起始信号，而同时获得主机权，这样就要进行仲裁。当 SCL 线是高电平时，仲裁在 SDA 线发生，当其他主机发送低电平时，发送高电平的主机将丢失仲裁，因为总线上的电平与它自己的电平不同。

仲裁可以持续多位，它的第一个阶段是比较地址位，如果每个主机都尝试寻址相同的器件，仲裁会继续比较数据位，或者比较响应位。因为 I^2C 总线的地址和数据信息由赢得仲裁的主机决定，在仲裁过程中不会丢失信息。

（3）用时钟同步机制作为握手

器件可以快速接收数据字节，但可能需要更多时间保存接收到的字节或准备一个要发送的字节。此时，这个器件可以使 SCL 线保持低电平，迫使与之交换数据的器件进入等待状态，直到准备好下一字节的发送或接收。

5. 传输协议

（1）寻址字节

主机产生起始信号后，发送的第一个字节为寻址字节，该字节的头 7 位（高 7 位）为从机地址，最低位（LSB）决定了报文的方向，“0”表示主机写信息到从机，“1”表示主机读从机中的信息，如图 4.76所示。当发送了一个地址后，总线上的每个器件都将头 7 位与它自己的地址比较。如果一样，器件就会应答主机的寻址，至于是从机接收器还是从机发送器都由R/$\overline{W}$位决定。

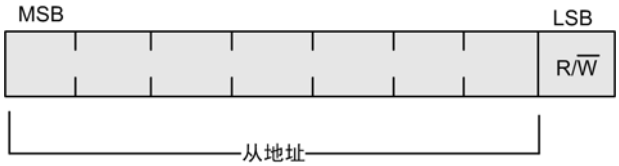


图 4.76 起始信号后的第一个字节

从机地址由一个固定的和一个可编程的部分构成。例如，某些器件有 4 个固定的位（高 4 位）和 3 个可编程的地址位（低 3 位），因此同一总线上共可以连接 8 个相同的器件。I^2C 总线委员会协调 I^2C 地址的分配，保留了 2 组 8 位地址（0000××××和 1111××××），这 2 组地址的用途可查阅相关资料。

（2）传输格式

主机产生起始信号后，发送一个寻址字节，收到应答后紧跟着的就是数据传输，数据传输一般由主机产生的停止位终止。但是，如果主机仍希望在总线上通讯，它可以产生重复起始信号（Sr）和寻址另一个从机，而不是首先产生一个停止信号。在这种传输中，可能有不同的读/写格式结合。可能的数据传输格式有：

- **主机发送数据到从机**：寻址字节的“R/$\overline{W}$”位为 0，数据传输的方向不改变，如图 4.77所示。

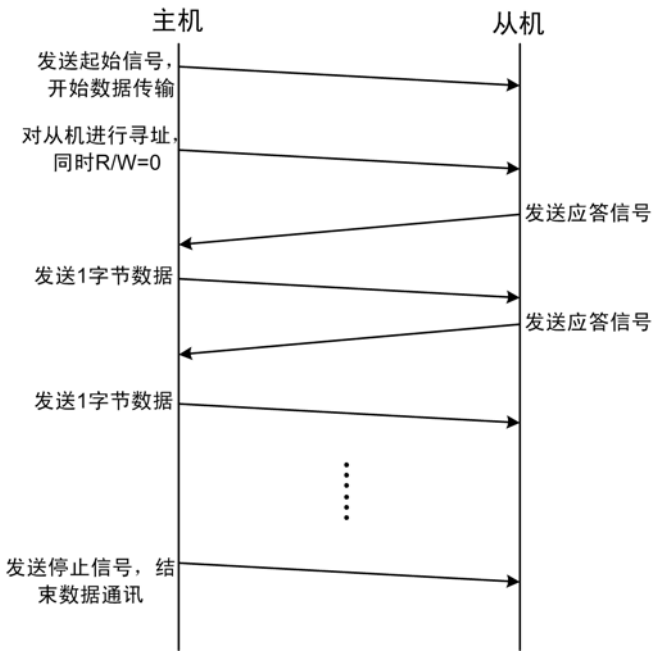


图 4.77 主机发送数据到从机

- **主机读取从机中的数据：**主机发送完寻址字节后，主机立即读取从机中的数据。如图 4.78所示，寻址字节的“R/$\overline{W}$”位为 1，在第一次从机产生的响应后，主机发送器变成主机接收器，从机接收器变成从机发送器。之后，数据由从机发送，主机接收，每个应答由主机产生，时钟信号CLK仍由主机产生。若主机要终止本次传输，则发送一个非应答信号（$\overline{A}$），接着主机产生停止信号。

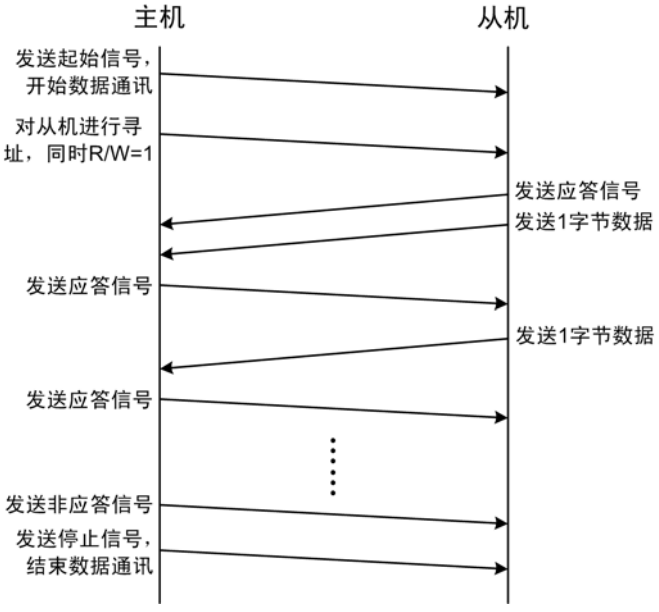


图 4.78 主机读取从机中的数据

- **复合格式：**复合格式是上面两种格式的混合。如图 4.79所示为一个复合格式传输示例，由主机负责开始数据通信，并向从机发送数据，R/$\overline{W}$ =0。主机数据传输完毕后，没有发送停止信号，而是再次发送起始信号，接下来主机读取从机中的数据，R/$\overline{W}$ =1，当主机读取完毕后，先发一个非应答信号，然后发送

停止信号，结束数据通信。

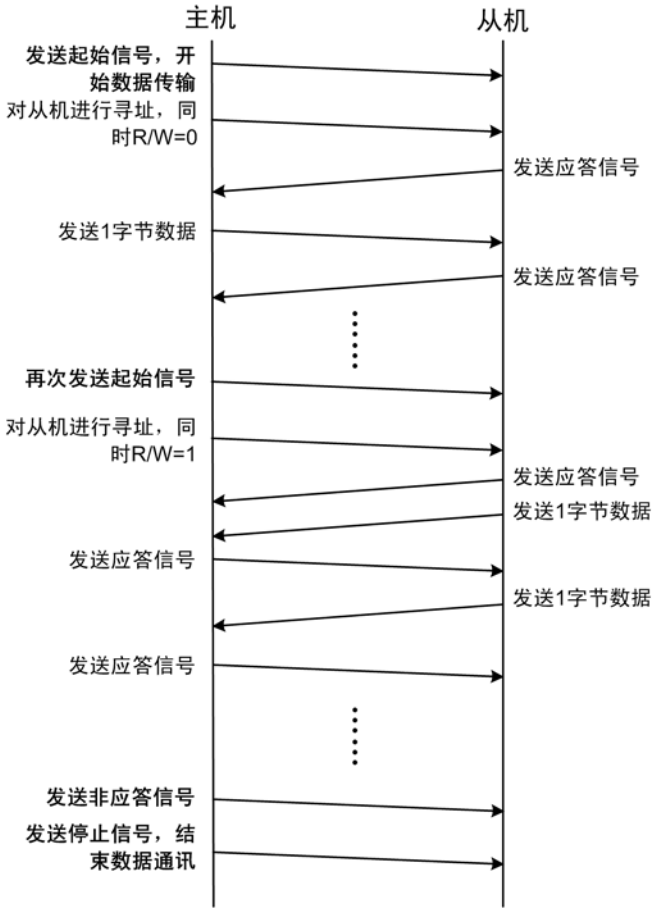


图 4.79 复合格式传输示例

4.11.4 寄存器描述

每个I^2C接口包含 7 个寄存器，如表 4.115所示。

表 4.115 I2C 寄存器映射

名称	描述	访问	复位值[1]	I^2C0 地址& 名称	I^2C1 地址& 名称
I2CONSET	I^2C 控制置位寄存器。当向该寄存器写入 1 时，I2C 控制寄存器中相应位置位。写 0 到 I2C 控制寄存器的相应位没有影响	R/W	0x00	0xE001C000 I2C0CONSET	0xE005C000 I2C1CONSET
I2STAT	I^2C 状态寄存器。在 I^2C 操作中，该寄存器提供详细的状态码使软件确定所需的下一步操作	RO	0xF8	0xE001C004 I2C0STAT	0xE005C004 I2C1STAT
I2DAT	I^2C 数据寄存器。在主/从机发送模式下，要发送的数据被写入该寄存器。在主/从机接收模式下，接收到的数据可从该寄存器中读取	R/W	0x00	0xE001C008 I2C0DAT	0xE005C008 I2C1DAT

续上表

名称	描述	访问	复位值[1]	I²C0 地址& 名称	I²C1 地址& 名称
I2ADR	I²C 从地址寄存器。包含从机模式下 I²C 接口操作的 7 位从地址，不在主机模式下使用。最低位决定从机是否响应通用调用地址	R/W	0x00	0xE001C00C I2C0ADR	0xE005C00C I2C1ADR
I2SCLH	SCH 占空比寄存器高半字。确定 I²C 时钟的高时间	R/W	0x04	0xE001C010 I2C0SCLH	0xE005C010 I2C1SCLH
I2SCLL	SCL 占空比寄存器低半字。确定 I²C 时钟的低时间。12nSCLL 和 12nSCLH 一起确定 I²C 主机产生的时钟频率和从机模式下使用的特定时间	R/W	0x04	0xE001C014 I2C0SCLL	0xE005C014 I2C1SCLL
I2CONCLR	I²C 控制清零寄存器。当向该寄存器中的位写入 1 时，I²C 控制寄存器中相应位被清零。写 0 到 I²C 控制寄存器的相应位没有影响	WO	NA	0xE001C018 I2C0CONCLR	0xE005C018 I2C1CONCLR

[1] 复位值仅指已使用位中保存的数据。它不包括保留位的内容。

4.11.5 操作模式

1. 主模式I²C

在该模式中，LPC2103 作为主机，向从机发送数据(即，主发送模式)及接收从机的数据(即，主接收模式)。在进入主模式 I2C，I2CONSET 必须按照进行初始化。

I2EN 置 1 操作是通过向 I2CONSET 写入 0x40 实现；AA，STA 和 SI 置 0 操作是通过向 I2CONCLR 写入 0x2C 实现；当总线产生了一个停止条件时，STO 位由硬件自动置 0。

图 4.80 主模式配置

说明：I2EN = 1，使能 I²C 接口；
AA = 0，不产生应答信号，即不允许进入从机模式；
SI = 0，I²C 中断标志为 0；
STO=0，停止标志为 0；
STA=0，起始标志为 0。

（1）主模式 I2C 的初始化

使用主模式I²C时，先设置I/O口功能选择，然后设置总线的速率，再使能主I²C，接下来便可以开始发送/接收数据，主模式I²C初始化示例如程序清单 4.43所示。实际应用中，通常会使用中断方式进行I²C的操作，所以初始化程序中加入了中断的初始化。

程序清单 4.43 主模式 I²C 初始化示例

```
/*********************************************************************************************
** 函数名称： I2C_Init
** 函数功能： I2C 初始化，包括初始化其中断为向量 IRQ 中断
** 入口参数： fi2c        初始化 I2C 总线速率，最大值为 400K
** 出口参数： 无
*********************************************************************************************/
void  I2C_Init(uint32 fi2c)
{
    if(fi2c>400000)  fi2c = 400000;
    PINSEL0       = (PINSEL0&0xFFFFFF0F) | 0x50;    /* 设置 I2C 控制口有效                  */
    I2SCLH        = (Fpclk/fi2c + 1) / 2;           /* 设置 I2C 时钟为 fi2c                 */
    I2SCLL        = (Fpclk/fi2c) / 2;
    I2CONCLR      = 0x2C;
    I2CONSET      = 0x40;                           /* 使能主 I2C                           */

    /* 设置 I2C 中断允许 */
    VICIntSelect  = 0x00000000;                     /* 设置所有通道为 IRQ 中断              */
    VICVectCntl0  = 0x29;                           /* I2C 通道分配到 IRQ slot 0 即优先级最高 */
    VICVectAddr0  = (int)IRQ_I2C;                   /* 设置 I2C 中断向量地址                */
    VICIntEnable  = 0x0200;                         /* 使能 I2C 中断                        */
}
```

（2）主模式 I²C 的数据发送

主模式I²C的数据发送格式见图 4.81，起始和停止条件用于指示串行传输的起始和结束，第一个发送的数据包含接收器件的从地址（7 位）和读写操作位。在此模式下，读写操作位（R/W）应该为 0，表示执行写操作。数据的发送每次为 8 位，即一字节，每发送完一个字节，主机都接收到一个应答位(是由从机回发的)。

主模式I²C的数据发送波形图如图 4.82所示。

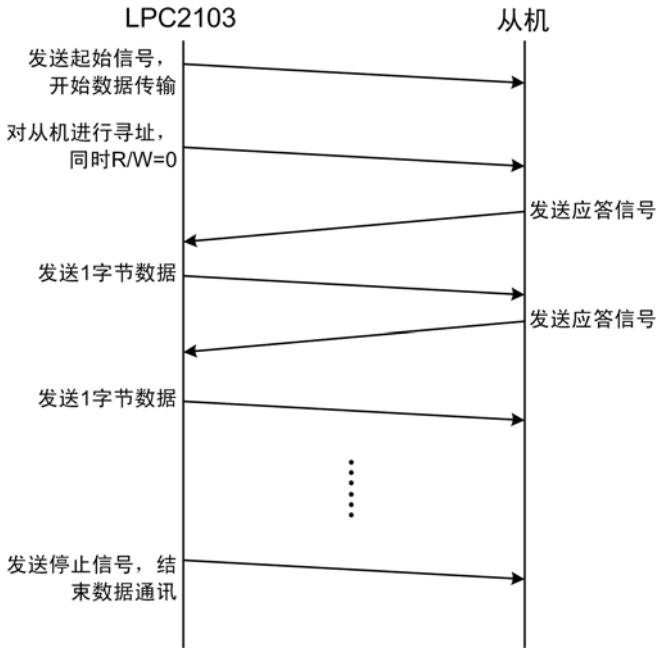


图 4.81 LPC2103 主发送模式

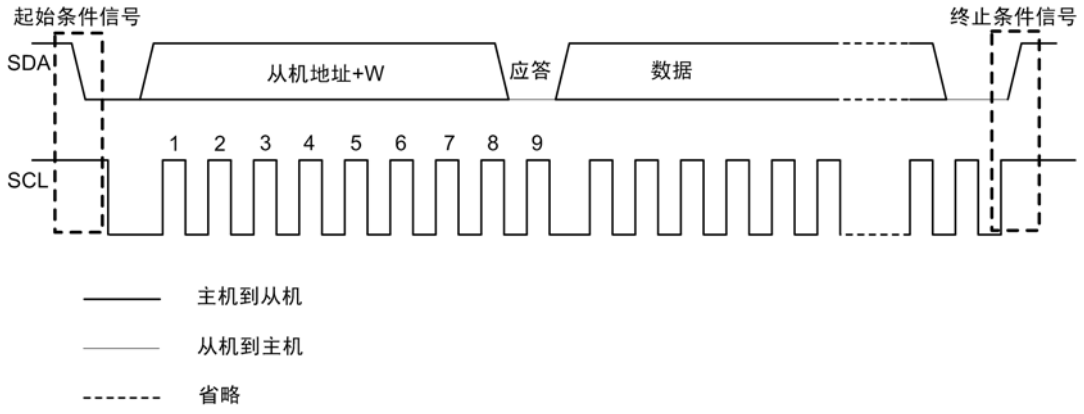


图 4.82 主模式 I^2C 的数据发送波形

主模式 I^2C 的数据发送操作步骤如下：

- 通过软件置位 STA 进入 I^2C 主发送模式，I^2C 逻辑在总线空闲后立即发送一个起始条件；
- 当发送完起始条件后，SI 会置位，此时 I2STAT 中的状态代码为 08H，该状态代码用于中断服务程序的处理；
- 把从地址和读写操作位装入 I2DAT（数据寄存器），然后清零 SI 位，开始发送从地址和 W 位；
- 当从地址和W位已发送且接收到应答位之后，SI位再次置位，可能的状态代码为 18H，20H或 38H，每个状态代码及其对应的执行动作见表 4.116；
- 若状态码为 18H，表明从机已应答，则可以将数据装入 I2DAT，然后清零 SI 位，开始发送数据；
- 当正确发送数据，SI位再次置位，可能的状态代码为 28H或 30H，此时可以再次发送数据，或者置位STO结束总线，每个状态代码及其对应的执行动作见表 4.116。

表 4.116 主发送模式状态

状态代码 (I2STAT)	I^2C 总线硬件状态	应用软件的响应					I^2C 硬件执行的下一个动作
		读/写 I2DAT	写 I2CON				
			STA	STO	SI	AA	
08H	已发送起始条件	装入 SLA+W	×	0	0	×	将发送 SLA+W，接收 ACK 位
10H	已发送重复起始条件	装入 SLA+W	×	0	0	×	同上
		装入 SLA+R	×	0	0	×	将发送 SLA+W，I^2C 将切换到主接收模式
18H	已发送 SLA+W；已接收 ACK	装入数据字节	0	0	0	×	将发送数据字节，接收 ACK 位
		无 I2DAT 动作	1	0	0	×	将发送重复起始条件
		无 I2DAT 动作	0	1	0	×	将发送停止条件；STO 标志将复位
		无 I2DAT 动作	1	1	0	×	将发送停止条件，然后发送起始条件；STO 标志将复位
20H	已发送 SLA+W；已接收非 ACK	装入数据字节	0	0	0	×	将发送数据字节，接收 ACK 位
		无 I2DAT 动作	1	0	0	×	将发送重复起始条件
		无 I2DAT 动作	0	1	0	×	将发送停止条件；STO 标志将复位
		无 I2DAT 动作	1	1	0	×	将发送停止条件，然后发送起始条件；STO 标志将复位
28H	已发送 I2DAT 中的数据字节；已接收 ACK	装入数据字节	0	0	0	×	将发送数据字节，接收 ACK 位
		无 I2DAT 动作	1	0	0	×	将发送重复起始条件
		无 I2DAT 动作	0	1	0	×	将发送停止条件；STO 标志将复位
		无 I2DAT 动作	1	1	0	×	将发送停止条件，然后发送起始条件；STO 标志将复位
30H	已发送 I2DAT 中的数据字节；已接收非 ACK	装入数据字节	0	0	0	×	将发送数据字节，接收 ACK 位
		无 I2DAT 动作	1	0	0	×	将发送重复起始条件
		无 I2DAT 动作	0	1	0	×	将发送停止条件；STO 标志将复位
		无 I2DAT 动作	1	1	0	×	将发送停止条件，然后发送起始条件；STO 标志将复位
38H	在 SLA+R/W 或数据字节中丢失仲裁	无 I2DAT 动作	0	0	0	×	I^2C 总线将被释放；进入不可寻址从模式
		无 I2DAT 动作	1	0	0	×	当总线变为空闲时发送起始条件

主模式I^2C的数据发送(中断方式)程序原理示意图如图 4.83所示。

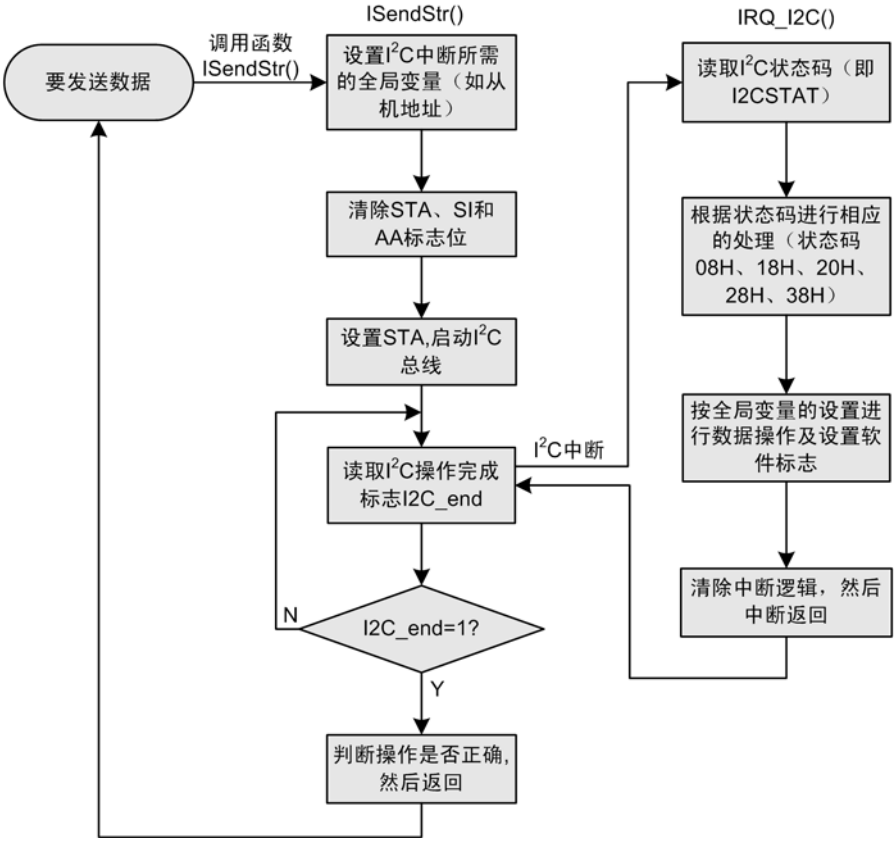


图 4.83　主模式 I²C 的数据发送程序原理示意图

（3）主模式 I²C 的数据接收

在主接收模式中，主机所接收的数据字节来自从发送器(即从机) ，主模式I²C的数据接收格式见图 4.84。起始和停止条件用于指示串行传输的起始和结束，第一个发送的数据包含接收器件的从地址（7 位）和读写操作位。在此模式下，读写操作位（R/W）应该为 1，表示执行读操作。

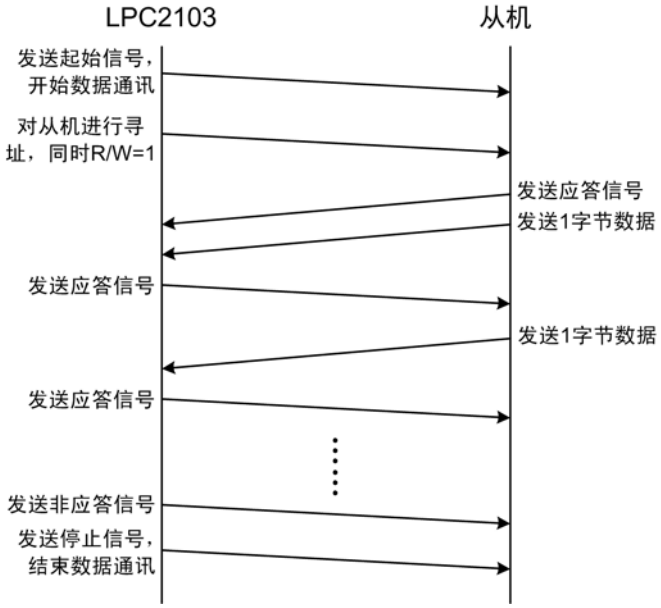


图 4.84 LPC2103 主接收模式

主模式I²C的数据接收波形图如图 4.85所示。

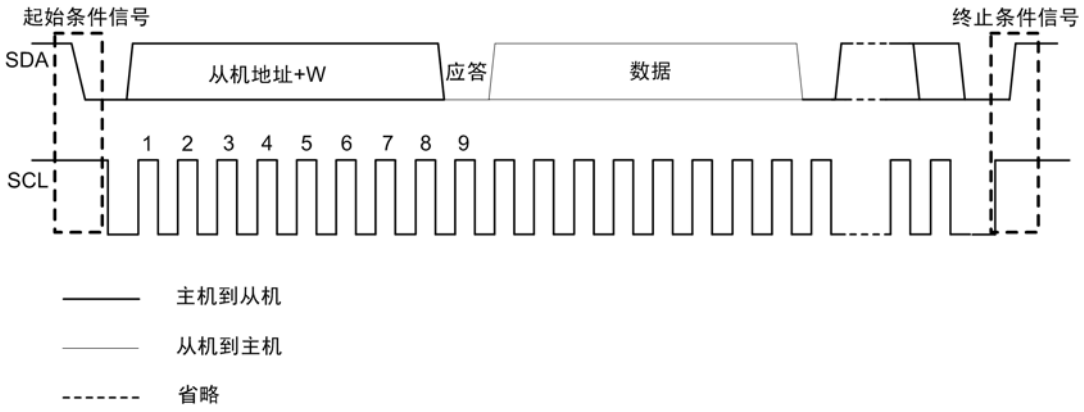


图 4.85 主模式 I²C 的数据接收波形

主模式 I²C 的数据发送操作步骤如下：

- 通过软件置位 STA 进入 I²C 主发送模式，I²C 逻辑在总线空闲后立即发送一个起始条件；
- 当发送完起始条件后，SI 会置位，此时 I2STAT 中的状态代码为 08H，该状态代码用于中断服务程序的处理；
- 把从地址和读写操作位装入 I2DAT（数据寄存器），然后清零 SI 位，开始发送从地址和 R 位；
- 当从地址和R位已发送且接收到应答位之后，SI位再次置位，可能的状态代码为 38H，40H或 48H，每个状态代码及其对应的执行动作见表 4.117；
- 若状态码为 40H，表明从机已应答。设置 AA 位，用来控制接收到数据后是产生应答信号，还是产生非应答信号，然后清零 SI 位，开始接收数据；
- 当正确接收到一字节数据后，SI位再次置位，可能的状态代码为 50H或 58H，此时可以再次接收数据，或者置位STO结束总线。每个状态代码及其对应的执行动作见表 4.117。

表 4.117 主接收模式状态

状态代码 (I2STAT)	I²C 总线硬件状态	应用软件的响应					I²C 硬件执行的下一个动作
		读/写 I2DAT	写 I2CON				
			STA	STO	SI	AA	
08H	已发送起始条件	装入 SLA+R	×	0	0	×	将发送 SLA+R，接收 ACK 位
10H	已发送重复起始条件	装入 SLA+R	×	0	0	×	同上
		装入 SLA+W	×	0	0	×	将发送 SLA+W，I²C 将切换到主发送模式
38H	在发送 SLA+R 时丢失仲裁	无 I2DAT 动作	0	0	0	×	I²C 总线将被释放；I²C 将进入从模式
		无 I2DAT 动作	1	0	0	×	当总线恢复空闲后发送起始条件
40H	已发送 SLA+R；已接收 ACK	无 I2DAT 动作	0	0	0	0	将接收数据字节；返回非 ACK 位
		无 I2DAT 动作	0	0	0	1	将接收数据字节；返回 ACK 位

续上表

状态代码 (I2STAT)	I²C 总线硬件状态	应用软件的响应					I²C 硬件执行的下一个动作
		读/写 I2DAT	写 I2CON				
			STA	STO	SI	AA	
48H	已发送 SLA+R；已接收非 ACK	无 I2DAT 动作	1	0	0	×	将发送重复起始条件
		无 I2DAT 动作	0	1	0	×	将发送停止条件；STO 标志将复位
		无 I2DAT 动作	1	1	0	×	将发送停止条件，然后发送起始条件；STO 标志将复位
50H	已接收数据字节；已返回 ACK	读数据字节	0	0	0	0	将接收数据字节，返回非 ACK 位
		读数据字节	0	0	0	1	将接收数据字节；返回 ACK 位
58H	已接收数据字节；已返回非 ACK	读数据字节	1	0	0	×	将发送重复起始条件
		读数据字节	0	1	0	×	将发送停止条件；STO 标志将复位
		读数据字节	1	1	0	×	将发送停止条件，然后发送起始条件；STO 标志将复位

主模式I²C的数据接收（中断方式）程序原理示意图如图 4.86所示。

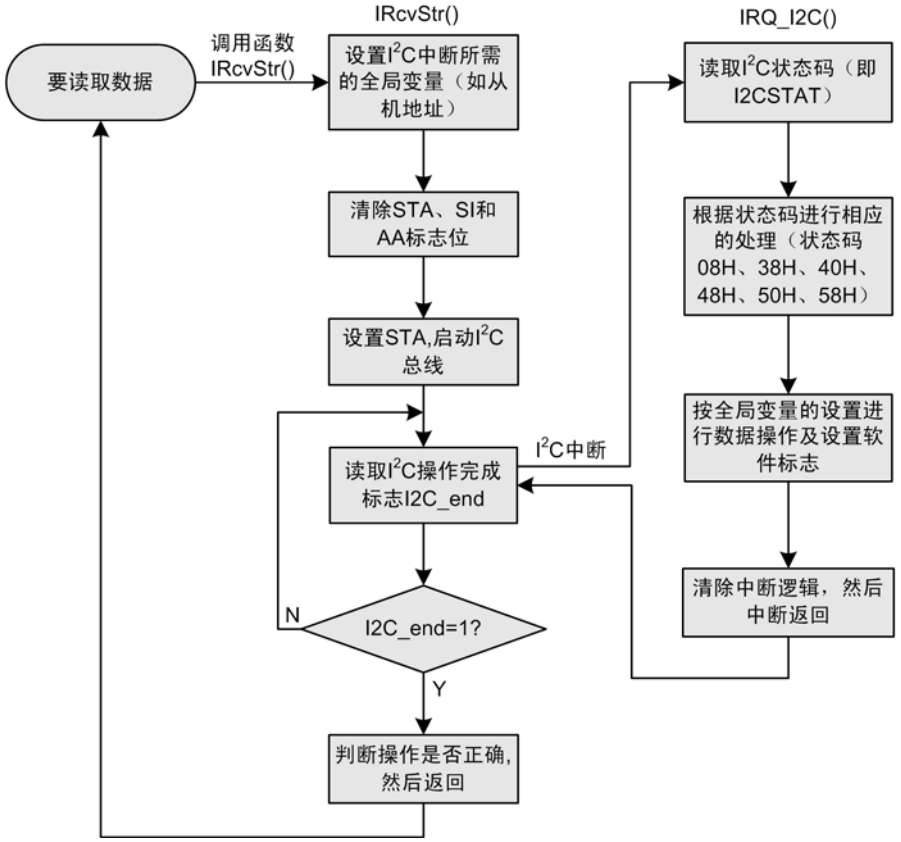


图 4.86 主模式 I²C 的数据接收程序原理示意图

2. 从模式I²C

LPC2103 配置为I²C从机时，I²C主机可以对它进行读/写操作，此时从机处于从发送/接收模式。要初始化从接收模式，用户必须将从地址写入从地址寄存器（I2ADR）并按照图 4.87配置I²C控制置位寄存器（I2CONSET），I2CONSET寄存器的详细说明见表 4.115。

I2EN 和 AA 置 1 操作是通过向 I2CONSET 写入 0x44 实现；STA 和 SI 置 0 操作是通过向 I2CONCLR 写入 0x28 实现；当总线产生了一个停止条件时，STO 位由硬件自动置 0。

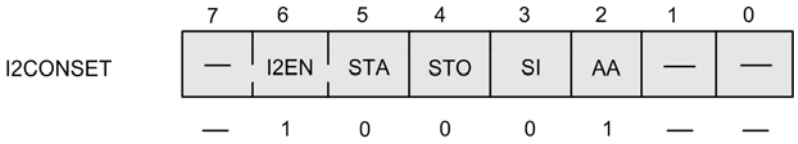


图 4.87 从模式配置

说明：I2EN = 1，使能 I^2C 接口；

AA = 1，应答主机对本从机地址的访问；

SI = 0，I^2C 中断标志为 0；

STO=0，停止标志为 0；

STA=0，起始标志为 0。

（1）从模式 I^2C 的初始化

使用从模式I^2C时，先设置I/O口功能选择，再设置从机地址，然后使能I^2C(配置为从模式)，即可等待主机访问，从模式I^2C初始化示例如程序清单 4.44所示。实际应用中，通常是使用中断方式进行I^2C的操作，所以初始化程序中加入了中断的初始化。

因为 I^2C 总线时钟信号是由主机产生，所以从机不用初始化 I2SCLH 和 I2SCLL 寄存器。

程序清单 4.44 从模式 I^2C 初始化示例

```
/*********************************************************************************************
** 函数名称： I2C_SlaveInit
** 函数功能： 从模式 I2C 初始化，包括初始化其中断为向量 IRQ 中断。
** 入口参数： adr        本从机地址
** 出口参数： 无
*********************************************************************************************/
void  I2C_SlavInit(uint8 adr)
{
    PINSEL0 = (PINSEL0&0xFFFFFF0F) | 0x50;          /* 设置 I2C 控制口有效                  */
    I2ADR       =   adr&0xFE;                       /* 设置从机地址                         */
    I2CONCLR    =   0x28;
   I2CONSET     =   0x44;                           /* I2C 配置为从机模式                   */

   /* 设置 I2C 中断允许*/
    VICIntSelect    =   0x00000000;                 /* 设置所有通道为 IRQ 中断              */
    VICVectCntl0    =   0x29;                       /* I2C 通道分配到 IRQ slot 0，优先级最高 */
    VICVectAddr0    =   (int)IRQ_I2C;               /* 设置 I2C 中断向量地址                */
    VICIntEnable    =   0x0200;                     /* 使能 I2C 中断                        */
}
```

（2）从模式 I^2C 的数据接收

当主机访问从机时，若读写操作位为 0（W），则从机进入从接收模式，接收主机发送过来的数据，并产生应答信号。从模式I^2C的数据接收过程见图 4.88，从接收模式中，总线时钟、起始条件、从机地址、停止条件仍由主机产生。

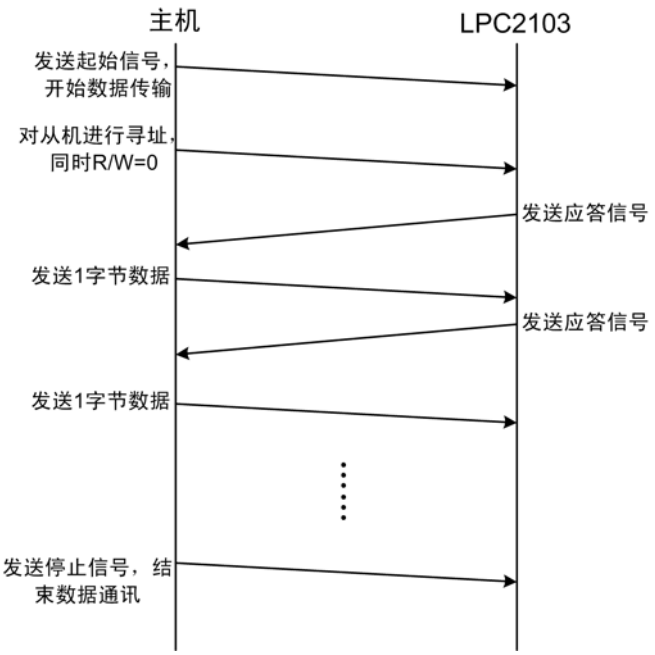


图 4.88 LPC2103 从接收模式

使用从模式I^2C时，用户程序只需要在I^2C中断服务程序完成各种数据操作，即是根据各种状态码作出相应的操作，从接收模式的每个状态代码及其对应的执行动作见表 4.118。

表 4.118 从接收模式状态

状态代码 (I2STAT)	I^2C 总线硬件状态	应用软件的响应					I^2C 硬件执行的下一个动作
		读/写 I2DAT	写 I2CON				
			STA	STO	SI	AA	
60H	已接收自身 SLA+W; 已返回 ACK	无 I2DAT 动作	×	0	0	0	将接收数据字节并返回非 ACK 位
		无 I2DAT 动作	×	0	0	1	将接收数据字节并返回 ACK 位
68H	主控器时在 SLA+W 中丢失仲裁; 已接收自身 SLA+W, 已返回 ACK	无 I2DAT 动作	×	0	0	0	将接收数据字节并返回非 ACK 位
		无 I2DAT 动作	×	0	0	1	将接收数据字节并返回 ACK 位
70H	已接收通用调用地址 (00H); 已返回 ACK	无 I2DAT 动作	×	0	0	0	将接收数据字节并返回非 ACK 位
		无 I2DAT 动作	×	0	0	1	将接收数据字节并返回 ACK 位
78H	主控器时在 SLA+R/W 中丢失仲裁; 已接收通用调用地址;已返回 ACK	无 I2DAT 动作	×	0	0	0	将接收数据字节并返回非 ACK 位
		无 I2DAT 动作	×	0	0	1	将接收数据字节并返回 ACK 位

续上表

状态代码 (I2STAT)	I^2C 总线硬件状态	应用软件的响应					I^2C 硬件执行的下一个动作
		读/写 I2DAT	写 I2CON				
			STA	STO	SI	AA	
80H	前一次寻址使用自身从地址；已接收数据字节；已返回 ACK	读数据字节	×	0	0	0	将接收数据字节并返回非 ACK 位
		读数据字节	×	0	0	1	将接收数据字节并返回 ACK 位
88H	前一次寻址使用自身从地址；已接收数据字节；已返回非 ACK	读数据字节	0	0	0	0	切换到不可寻址SLV模式；不识别自身SLA或通用调用地址
		读数据字节	0	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址
		读数据字节	1	0	0	0	切换到不可寻址SLV模式；不识别自身SLA或通用调用地址；当总线空闲后发送起始条件
		读数据字节	1	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址；当总线空闲后发送起始条件
90H	前一次寻址使用通用调用；已接收数据字节；已返回 ACK	读数据字节	×	0	0	0	将接收数据字节并返回非 ACK 位
		读数据字节	×	0	0	1	将接收数据字节并返回 ACK 位
98H	前一次寻址使用通用调用；已接收数据字节；已返回非 ACK	读数据字节	0	0	0	0	切换到不可寻址SLV模式；不识别自身SLA或通用调用地址
		读数据字节或	0	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址
		读数据字节或	1	0	0	0	切换到不可寻址SLV模式；不识别自身SLA或通用调用地址；当总线空闲后发送起始条件
		读数据字节	1	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址；当总线空闲后发送起始条件
A0H	当使用 SLV/REC 或 SLV/TRX 静态寻址时,接收到停止条件或重复的起始条件	无 I2DAT 动作	0	0	0	0	切换到不可寻址SLV模式；不识别自身SLA或通用调用地址
		无 I2DAT 动作	0	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址
		无 I2DAT 动作	1	0	0	0	切换到不可寻址SLV模式；不识别自身SLA或通用调用地址；当总线空闲后发送起始条件
		无 I2DAT 动作	1	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址；当总线空闲后发送起始条件

（3）从模式 I²C 的数据发送

当主机访问从机时，若读写操作位为 1（R），则从机进入从发送模式，向主机发送数据，并等待主机的应答信号。从模式I²C的数据发送过程见图 4.89，从发送模式中，总线时钟、起始条件、从机地址、停止条件仍由主机产生。

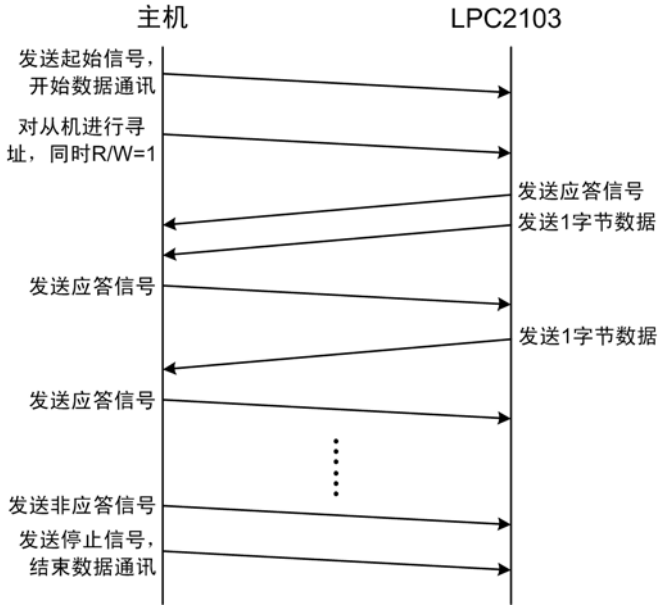


图 4.89 从发送模式的过程

使用从模式I²C时，用户程序只需要在I²C中断服务程序完成各种数据操作，即是根据各种状态码作出相应的操作，从发送模式的每个状态代码及其对应的执行动作见表 4.119。

表 4.119 从发送模式状态

状态代码 (I2STAT)	I²C 总线硬件状态	应用软件的响应					I²C 硬件执行的下一个动作
		读/写 I2DAT	写 I2CON				
			STA	STO	SI	AA	
A8H	已接收自身 SLA+R；已返回 ACK	装入数据字节 或装入数据字节	× ×	0 0	0 0	0 1	将发送最后的数据字节并接收 ACK 位将发送数据字节并接收 ACK 位
B0H	主控器时在 SLA+R/W 中丢失仲裁；已接收自身 SLA+R，已返回 ACK	装入数据字节 或装入数据字节	× ×	0 0	0 0	0 1	将发送最后的数据字节并接收 ACK 位将发送数据字节并接收 ACK 位
B8H	已发送 I2DAT 中数据字节；已返回 ACK	装入数据字节 或装入数据字节	× ×	0 0	0 0	0 1	将发送最后的数据字节并接收 ACK 位将发送数据字节并接收 ACK 位
00H	在 MST 或选择的从模式中，由于非法起始或停止条件，使总线发生错误。当干扰导致 I²C 进入一个未定义状态时，也可产生状态 00H	无 I2DAT 动作	0	1	0	×	在 MST 或寻址 SLV 模式中只有内部硬件受影响。在所有情况下，总线被释放，而 I²C 切换到不可寻址 SLV 模式。STO 复位

续上表

状态代码 (I2STAT)	I^2C 总线硬件状态	应用软件的响应					I^2C 硬件执行的下一个动作
		读/写 I2DAT	写 I2CON				
			STA	STO	SI	AA	
C0H	已发送 I2DAT 中数据字节；已返回非 ACK	无 I2DAT 动作或 无 I2DAT 动作或 无 I2DAT 动作或 无 I2DAT 动作	0 0 1 1	0 0 0 0	0 0 0 0	0 1 0 1	切换到不可寻址 SLV 模式；不识别自身 SLA 或通用调用地址切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址 切换到不可寻址 SLV 模式；不识别自身 SLA 或通用调用地址；当总线空闲后发送起始条件切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址；当总线空闲后发送起始条件
C8H	已发送 I2DAT 中最后的数据字节(AA=0)；已返回 ACK	无 I2DAT 动作或 无 I2DAT 动作或 无 I2DAT 动作或 无 I2DAT 动作	0 0 1 1	0 0 0 0	0 0 0 0	0 1 0 1	切换到不可寻址 SLV 模式；不识别自身 SLA 或通用调用地址 切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址 切换到不可寻址 SLV 模式；不识别自身 SLA 或通用调用地址；当总线空闲后发送起始条件 切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址；当总线空闲后发送起始条件
F8H	无可用相关信息；SI=0	无 I2DAT 动作	无 I2DAT 动作				等待或进行当前的传输

4.11.6 I^2C中断

从前面的描述可以看出，对于硬件 I^2C 接口，通常都使用中断的方式进行操作。当 I^2C 的状态发生变化时，就会产生中断，因此，发生 I^2C 中断时，必须要读取 I^2C 状态寄存器，根据当前的状态采取相应的措施，I^2C 接口的状态与其所处的模式有关，每种模式所对应的状态在表 4.116、表 4.117、表 4.118和表 4.119中有介绍，这里不再重复。I^2C中断与向量中断控制器（VIC）的关系如图 4.90所示。

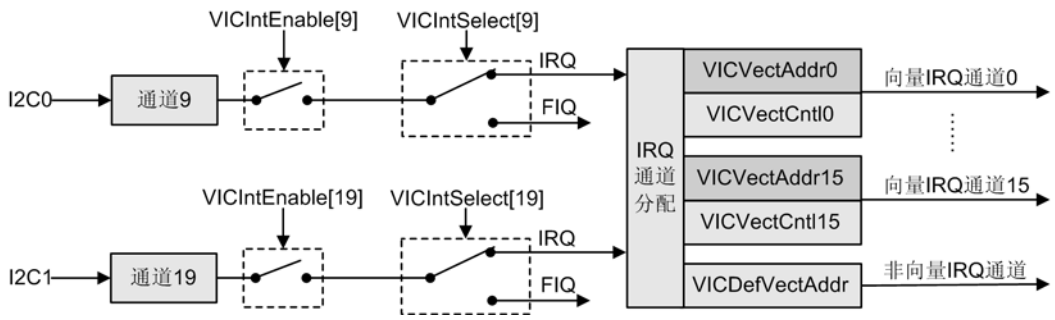


图 4.90 I^2C 与 VIC 的关系

I^2C0 和 I^2C1 分别处于 VIC 的通道 9 和通道 19，中断使能寄存器 VICIntEnable 用来控制 VIC 通道的中断使能。

- 当 VICIntEnable[9] = 1 时，通道 9 中断使能，即：I^2C0 中断使能；
- 当 VICIntEnable[19] = 1 时，通道 19 中断使能，即，I^2C1 中断使能。

中断选择寄存器 VICIntSelect 用来分配 VIC 通道的中断。当某一位为“1”时，对应的通道中断分配为 FIQ；当某一位为“0”时，对应的通道中断分配为 IRQ。VICIntSelect[9] 和 VICIntSelect[19]分别用来控制通道 9 和通道 19，即：

- 当 VICIntSelect[9] = 1 时，I^2C0 中断分配为 FIQ 中断；
- 当 VICIntSelect[9] = 0 时，I^2C0 中断分配为 IRQ 中断；
- 当 VICIntSelect[19] = 1 时，I^2C1 中断分配为 FIQ 中断；
- 当 VICIntSelect[19] = 0 时，I^2C1 中断分配为 IRQ 中断。

当分配为 IRQ 时，还需要设置对应的通道控制寄存器和地址寄存器。有关寄存器 VICVectCntln 和 VICVectAddrn 的说明，请参考“向量中断控制器（VIC）”一节。

4.11.7 应用示例

本示例使用硬件I^2C软件包，往CAT1025 内部的EEPROM中写入 10 字节数据，然后读出校验。如果写入正确，则LED闪烁一次，否则LED一直点亮。示例程序代码如程序清单 4.45所示。

用户需要短接 JP4 的 P0.17、P0.18 端口，输出控制 LED。

注意：CAT1025 内部 EEPROM 是 16 字节页存储，在写入数据时注意不要页溢出，否则会重新向写入页的首地址空间写入内容，覆盖原有数据。

程序清单 4.45 I^2C 读取 CAT1025 实验

```
#include "config.h"
#include "I2CINT.h"

#define    LED1          1 <<  17                    /*  控制 LED1 闪烁          */
#define    LED2          1 <<  18                    /*  控制 LED2 闪烁          */
#define    CAT1025       0xA0                        /*  CAT1025 器件从地址      */

/*********************************************************************************************
** 函数名称：DelayNS
** 功能描述：延时函数
```

```
** 入口参数：uiDly    值越大，延时时间越长
** 出口参数：无
*********************************************************************************************/
void DelayNS(uint32 uiDly)
{
    uint32 i;
    for(; uiDly > 0; uiDly--){
        for(i=0; i<50000; i++);
    }
}
/*********************************************************************************************
** 函数名称：I2cInit
** 功能描述：I2C 初始化
** 入口参数：uiFi2c I2C 总线频率(最大 400K)
** 出口参数：无
*********************************************************************************************/
void I2cInit(uint32 uiFi2c)
{
    if (uiFi2c > 400000){
        uiFi2c = 400000;
    }

    PINSEL0   = (PINSEL0 & (~0xF0)) | 0x50;          /* 设置 I2C 引脚连接          */
    I2SCLH    = (Fpclk / uiFi2c + 1)   / 2;          /* 设定 I2C 时钟              */
    I2SCLL    = (Fpclk / uiFi2c) / 2;
    I2CONCLR = 0x2C;
    I2CONSET = 0x40;                                 /* 使能主 I2C                 */

    /*   设置 I2C 中断            */
    VICIntSelect = 0x00000000;                       /* 设置所有通道为 IRQ 中断    */
    VICVectCntl0 = (0x20 | 0x09);                    /* I2C 通道分配最高优先级     */
    VICVectAddr0 = (int32)IRQ_I2C;                   /* 设置 I2C 中断向量          */
    VICIntEnable = 1 << 9;                           /* 使能 I2C 中断              */
}
/*********************************************************************************************
** 函数名称：Main
** 功能描述：I2C 总线对 CAT1025 进行读写操作
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
int main (void)
{
    uint8 i;
    uint8 uiDataBuf[32];
```

```
    PINSEL0 = 0x00000000;                                   /* 设置管脚连接 GPIO                    */
    PINSEL1 = 0x00000000;
    IO0DIR  = LED1 | LED2;                                  /* 设置 LED 控制口输出                  */
    IO0SET  = LED1 | LED2;

    IRQEnable();                                            /* 使能中断                             */
    I2cInit(400000);                                        /* I2C 初始化                           */

    for (i=0; i<10; i++){
        uiDataBuf[i] = i + '0';                             /* 数据 0~9，转换成 ASCII 码            */
    }
    /* 从起始地址 0x00 写入 10 个数据          */
    I2C_WriteNByte(CAT1025, ONE_BYTE_SUBA, 0x00, uiDataBuf, 10);
    DelayNS(10);
    for (i=0; i<10; i++){                                   /* 清零数据缓冲区，防止出错             */
        uiDataBuf[i] = 0;
    }

    /*  读回刚才写入的数据                  */
    I2C_ReadNByte(CAT1025, ONE_BYTE_SUBA, 0x00, uiDataBuf, 10);

    /*  判断读回的数据是否正确              */
    for (i=0; i<10; i++){
        if (uiDataBuf[i] != (i + '0')){                     /* 若读出数据错误，LED 一直点亮         */
            while (1){
                IO0CLR = LED1 | LED2;
            }
        }
        else{                                               /* 若读出数据正确，LED 闪烁、熄灭       */
            IO0CLR = LED1 | LED2;
            DelayNS(50);
            IO0SET = LED1 | LED2;
            DelayNS(50);
        }
    }
    while (1);
    return 0;
}
```

4.12 实时时钟

4.12.1 概述

实时时钟（RTC）提供一套计数器在系统上电和关闭操作时对时间进行测量。RTC 在掉电模式下消耗的功率非常低。LPC2101/02/03 的 RTC 时钟可由独立的 32.768kHz 振荡器或基于 VPB 时钟的可编程预分频器来提供。另外，RTC 还具有专用的电源管脚 V_{BAT}，可连接到电池或其它器件使用的相同的 3.3V 电压上。

4.12.2 特性

- 测量保持日历和时钟的时间通路；
- 超低功耗设计，支持电池供电系统；
- 提供秒、分、小时、日、月、年和星期；
- 指定的 32kHz 振荡器或可编程 VPB 时钟预分频器；
- 专用电源管脚可与电池或 3.3V 的电压相连。

4.12.3 寄存器描述

RTC包含了许多寄存器。地址空间按照功能分成 4 个部分。前 8 个地址为混合寄存器组。第二部分的 8 个地址为定时器计数器组，第三部分的 8 个地址为报警寄存器组。剩余的寄存器控制基准时钟分频器。如表 4.120所列。

表 4.120 实时时钟(RTC)寄存器映射

组别	名称	规格	描述	访问	复位值[1]	地址
混合寄存器	ILR	2	中断位置寄存器	R/W	*	0xE0024000
	CTC	15	时钟节拍计数器	RO	*	0xE0024004
	CCR	4	时钟控制寄存器	R/W	*	0xE0024008
	CIIR	8	计数器递增中断寄存器	R/W	*	0xE002400C
	AMR	8	报警屏蔽寄存器	R/W	*	0xE0024010
时间寄存器	CTIME0	32	完整时间寄存器 0	RO	*	0xE0024014
	CTIME1	32	完整时间寄存器 1	RO	*	0xE0024018
	CTIME2	32	完整时间寄存器 2	RO	*	0xE002401C
时间计数器	SEC	6	秒计数器	R/W	*	0xE0024020
	MIN	6	分寄存器	R/W	*	0xE0024024
	HOUR	5	小时寄存器	R/W	*	0xE0024028
	DOM	5	日期（月）寄存器	R/W	*	0xE002402C
	DOW	3	星期寄存器	R/W	*	0xE0024030
	DOY	9	日期（年）寄存器	R/W	*	0xE0024034
	MONTH	4	月寄存器	R/W	*	0xE0024038
	YEAR	12	年寄存器	R/W	*	0xE002403C
报警寄存器	ALSEC	6	秒报警值	R/W	*	0xE0024060
	ALMIN	6	分报警值	R/W	*	0xE0024064
	ALHOUR	5	小时报警值	R/W	*	0xE0024068
	ALDOM	5	日期（月）报警值	R/W	*	0xE002406C
	ALDOW	3	星期报警值	R/W	*	0xE0024070

续上表

组别	名称	规格	描述	访问	复位值[1]	地址
	ALDOY	9	日期（年）报警值	R/W	*	0xE0024074
	ALMON	4	月报警值	R/W	*	0xE0024078
	ALYEAR	12	年报警值	R/W	*	0xE002407C
预分频器	PREINT	13	预分频值，整数部分	R/W	0	0xE0024080
	PREFRAC	15	预分频值，小数部分	R/W	0	0xE0024084

[1] RTC 中除预分频器部分之外的其它寄存器都不受器件复位的影响。如果 RTC 使能，这些寄存器必须通过软件来初始化。复位值仅指已使用位中保存的数据。它不包括保留位的内容。

4.12.4 闰年计算

RTC 执行一个简单的位比较，看年计数器的最低两位是否为 0。如果为 0，那么 RTC 认为这一年为闰年。RTC 认为所有能被 4 整除的年份都为闰年。这个算法从 1901 年到 2099 年都是准确的，但在 2100 年出错，2100 年并不是闰年。闰年对 RTC 的影响只是改变 2 月份的长度、日期（月）和年的计数值。

4.12.5 RTC使用注意事项

V_{BAT} 必须连接到 $V_{DD(3.3)}$脚或一个独立的电源（外部电池）。由于 RTC 有两个可用的时钟（VPB 时钟(P_{CLK})或来自 RTCX1-2 管脚的 32KHz 的信号），所选择时钟的任何中断都会导致时间值的偏移。如果 RTC 初始化成这个时间值或从 RTC 激活后运行的一段时间内出现了一个错误，它们带来的变化都将影响真实的时钟时间。

RTCX1-2 管脚的信号可随时为 RTC 提供时钟，选择 P_{CLK} 作为 RTC 时钟和进入掉电模式会使时间的更新出现误差。而且，在系统操作过程中（重新配置 PLL、VPB 分频器或 RTC 预分频器）改变 RTC 的时间基准会使累加时间出现错误。当 RTC 时钟由 P_{CLK} 转变为 RTCX 管脚信号时也会出现累加时间误差。

一旦 RTCX1-2 管脚的 32KHz 信号被选择用作 RTC 的时钟源，RTC 可完全独立工作，与 VPB 时钟(P_{CLK})无关。因此，在要用到 RTC 且对功耗敏感的应用中（如电池供电设备）可通过使用 RTCX1-2 管脚的信号和清除 PCONP 功率控制寄存器的 PCRTC 位来降低功耗。

4.12.6 RTC中断

实时时钟含有两类中断——计数器增量中断（CIIR）、报警中断。寄存器 ILR 实际上就是中断标志寄存器，通过读取该寄存器的值，就可以判断中断类型。

- **计数器增量中断**

在RTC中，含有 8 个时间计数器（如秒计数器、分计数器等）。计数器增量中断寄存器（CIIR）中的每个位都对应一个时间计数器，如果使能其中的某一位，那么该位所对应的时间计数器每增加一次，就产生一次中断。例如：CIIR寄存器bit0 对应RTC中的秒计数器，如果CIIR[0] = 1，那么RTC每秒就会引发一次中断。增量中断控制原理示意图如图 4.91所示。

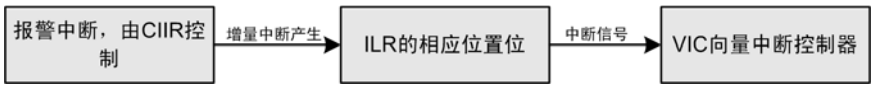


图 4.91 增量中断控制原理示意图

- **报警中断**

报警寄存器允许用户设定产生中断的时间，当RTC的当前时间与报警时间相匹配时，就会引发中断。在RTC中，设置了 8 个报警时间寄存器，分别用来存储报警时间值，当前时间是否与对应的报警时间进行比较，是由报警屏蔽寄存器（AMR）进行设定的。AMR寄存器中的每一位都对应一个报警时间寄存器，如果AMR寄存器中的某位为“1”，则对应的报警时间寄存器就被屏蔽了。例：当位AMRYEAR = 1 时，RTC的年值就不再和报警时间寄存器中的年值比较，年报警寄存器被屏蔽。如图 4.92所示。

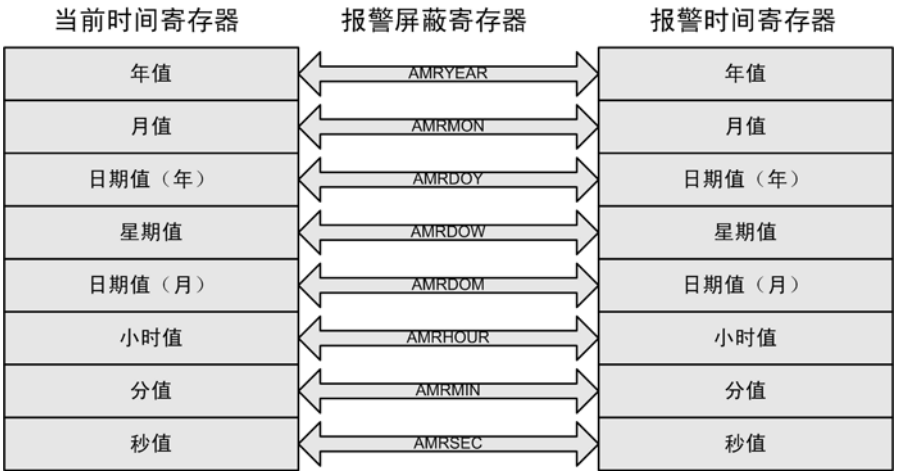


图 4.92 RTC 报警寄存器

如果所有未屏蔽的报警时间寄存器的值与它们对应的当前时间寄存器的值相匹配时，则会产生中断。报警中断控制原理示意图如图 4.93所示。

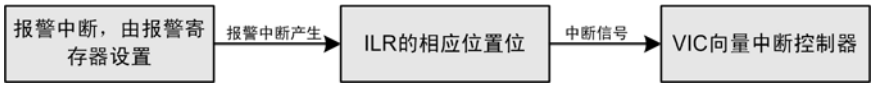


图 4.93 报警中断控制原理示意图

4.12.7 应用示例

1. 万年历显示实验

本实验需要使用开发套件提供的 EasyARM-C.exe 软件，程序读取 RTC 内部的时间寄存器，通过串口将数据送到上位机并在 EasyARM-C.exe 的万年历终端上显示。

本示例需要短接JP6 的P0.0、P0.1 端口，示例代码见程序清单 4.46所示。

程序清单 4.46 RTC 秒定时中断实验程序

```
#include "config.h"

/*  定义串口模式设置的数据结构          */
typedef struct UartMode{
    uint8 datab;                                /*  字长度 5/6/7/8          */
    uint8 stopb;                                /*  停止位 1/2              */
    uint8 parity;                               /*  奇偶校验                */
}UARTMODE;

# define   UART_BPS     9600                   /*  串口通信波特率          */
```

```
/*********************************************************************************************
** 函数名称：UARTInit
** 功能描述：串口初始化，设置为 8 位数据位，1 位停止位，无奇偶校验，波特率为 9600
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
void UARTInit (void)
{
    uint16 uiFdiv;
    PINSEL0 = (PINSEL0 & 0xFFFFFFF0) | 0x00000005;          /*  串口引脚设置                */

    U0LCR  = 0x83;                                          /*  允许设置波特率              */
    uiFdiv = (Fpclk / 16) / UART_BPS;                       /*  设置波特率                  */
    U0DLM  = uiFdiv / 256;
    U0DLL  = uiFdiv % 256;
    U0LCR  = 0x03;                                          /*  锁定波特率                  */
    U0FCR  = 0x01;                                          /*  FIFO 使能                   */
}
/*********************************************************************************************
** 函数名称：UART0SendByte
** 功能描述：向串口发送子节数据，并等待数据发送完成，使用查询方式
** 入口参数：uiDat    要发送的数据
** 出口参数：无
*********************************************************************************************/
void UART0SendByte (uint8 uiDat)
{
    U0THR = uiDat;                                          /*  写入数据                    */
    while ((U0LSR & 0x40) == 0);                            /*  等待数据发送完毕            */
}
/*********************************************************************************************
** 函数名称：PC_DispChar
** 功能描述：向 PC 机发送显示字符
** 入口参数：x      显示字符的横坐标
**           y      显示字符的纵坐标
**           chr    显示的字符，不能为 ff
**           color 显示的状态，包括前景色、背景色、闪烁位
**           与 DOS 字符显示一样：0～3,前景色，4～6，背景色，7，闪烁位
** 出口参数：无
*********************************************************************************************/
void PCDispChar (uint8 uiX, uint8 uiChr)
{
    UART0SendByte(0xFF);                                    /* 起始字符                     */
    UART0SendByte(0x81);
    UART0SendByte(uiX);
```

```
    UART0SendByte(uiChr);
    UART0SendByte(0x00);
}

uint8 const uiSHOWTABLE[10] = {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f};
/*********************************************************************************************
** 函数名称：SendTimeRtc
** 功能描述：将 RTC 时间值发送到串口显示
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
void SendTimeRtc (void)
{
    uint32 uiDatas;
    uint32 uiTimes;
    uint32 bak;

    uiTimes = CTIME0;                                   /* 读取完整的时钟寄存器      */
    uiDatas = CTIME1;

    bak = (uiDatas >> 16) & 0xfff;                      /* 获取 年                   */
    PCDispChar (0, uiSHOWTABLE [bak / 1000]);
    bak = bak % 1000;
    PCDispChar(1, uiSHOWTABLE   [bak / 100]);
    bak = bak % 100;
    PCDispChar (2, uiSHOWTABLE [bak / 10]);
    PCDispChar (3, uiSHOWTABLE [bak % 10]);

    bak = (uiDatas >> 8) & 0x0f;                        /* 获取 月                   */
    PCDispChar (4, uiSHOWTABLE [bak / 10]);
    PCDispChar (5, uiSHOWTABLE [bak % 10]);

    bak = uiDatas & 0x1f;                               /* 获取 日                   */
    PCDispChar (6, uiSHOWTABLE [bak / 10]);
    PCDispChar (7, uiSHOWTABLE [bak % 10]);

    bak = (uiTimes >> 24) & 0x07;                       /* 获取 星期                 */
    PCDispChar(8, uiSHOWTABLE [bak]);

    bak = (uiTimes >> 16) & 0x1f;                       /* 获取 小时                 */
    PCDispChar (9, uiSHOWTABLE [bak / 10]);
    PCDispChar (10,uiSHOWTABLE [bak % 10]);

    bak = (uiTimes >> 8) & 0x3f;                        /* 获取 分钟                 */
```

```
    PCDispChar (11, uiSHOWTABLE [bak / 10]);
    PCDispChar (12, uiSHOWTABLE [bak % 10]);

    bak = uiTimes & 0x3f;                                   /* 获取 秒钟                */
    PCDispChar (13, uiSHOWTABLE [bak / 10]);
    PCDispChar (14, uiSHOWTABLE [bak % 10]);
}
/*****************************************************************************************
** 函数名称：RTCInit
** 功能描述：初始化实时时钟
** 入口参数：无
** 出口参数：无
*****************************************************************************************/
void RTCInit (void)
{
    PREINT  = Fpclk / 32768 - 1;                            /* 设置基准时钟分频器       */
    PREFRAC = Fpclk - (Fpclk / 32768) * 32768;

    CCR     = 0x00;                                         /* 禁止时间计数器           */
    YEAR    = 2008;
    MONTH   = 04;
    DOM     = 07;
    DOW     = 4;
    HOUR    = 15;
    MIN     = 52;
    SEC     = 59;
    CIIR    = 0x01;                                         /* 设置秒值的增量产生 1 次中断*/
    CCR     = 0x01;                                         /* 启动 RTC                 */
}
/*****************************************************************************************
** 函数名称：main
** 功能描述：主函数，通过串口发送到 PC 机显示当前 RTC 时间
** 入口参数：无
** 出口参数：无
*****************************************************************************************/
int main (void)
{
    UARTMODE uart0_set;

    uart0_set.datab  = 8;
    uart0_set.stopb  = 1;
    uart0_set.parity = 0;

    UARTInit();                                             /* 串口初始化               */
```

```
    RTCInit();                                          /* RTC 初始化                */
    while (1) {
        while (0 == (ILR & 0x01));                      /* 等待 RTC 增量中断         */
        ILR = 0x01;                                     /* 清除中断标志              */
        SendTimeRtc();                                  /* 发送到串口显示            */
    }
    return 0;
}
```

2. 使用RTC唤醒掉电的CPU

RTC 中断可以唤醒掉电的 CPU，本实验使用 RTC 产生的秒增量中断，唤醒处于掉电状态的 CPU。CPU 唤醒后取反 LED，然后重新进入掉电模式。

为了能够将 CPU 从掉电模式唤醒，需要注意以下两点：

- RTC 使用独立的振荡器；
- 在中断唤醒寄存器（INTWAKE）中设置允许 RTC 唤醒 CPU。

进行该实验前必须用跳线帽短接JP4 的P0.17 端口，代码见程序清单 4.47所示。

程序清单 4.47 RTC 唤醒掉电 CPU 实验程序

```
#include "config.h"
#define   LED       (1 << 17)                       /*  P0.17 控制 LED           */
/*********************************************************************************************
** 函数名称：RTCInit
** 功能描述：初始化实时时钟
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
void RTCInit (void)
{
    CCR       = 0x12;                               /*  使用独立振荡器           */
    CIIR      = 0x01;                               /*  设置秒值的增量产生一次中断  */
    ILR       = 0x03;                               /*  清除增量中断标志         */
    CCR       = 0x11;                               /*  启动 RTC                 */
    INTWAKE= 1 << 15;                               /*  允许 RTC 唤醒 CPU        */
}
/*********************************************************************************************
** 函数名称：main
** 功能描述：RTC 中断唤醒掉电 CPU 实验
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
int main (void)
{
    PINSEL1 = PINSEL1 & (~(0x03 << 2));
    IO0DIR = LED;
```

```
    IO0SET = LED;

    RTCInit ();                                          /*  RTC 初始化                  */
while(1) {
    PCON = 0x02;                                         /*  CPU 进入掉电模式            */
        ILR = 0x01;                                      /*  清除增量中断标志            */

        /*  唤醒 CPU 后，取反 LED          */
        if((IO0PIN & LED) == 0) {
            IO0SET = LED;
        }
        else IO0CLR = LED;
    }
    return 0;
}
```

4.13 看门狗

4.13.1 概述

看门狗包括一个 4 分频的预分频器和一个 32 位计数器，时钟通过预分频器输入定时器，定时器递减。定时器递减的最小值为 0xFF。如果设置一个小于 0xFF 的值，系统会将 0xFF 装入计数器。因此最小看门狗间隔为(T_{PCLK}×256×4)，最大间隔为(T_{PCLK}×2^{32}×4)，两者都是 T_{PCLK}×4 的倍数。看门狗应当根据下面的方法来使用：

- 在 WDTC 寄存器中设置看门狗定时器的固定装载值；
- 在 WDMOD 寄存器中设置模式；
- 通过向 WDFEED 寄存器顺序写入 0xAA 和 0x55 启动看门狗；
- 在看门狗计数器向下溢出之前应当再次喂狗以防止复位/中断。

当看门狗计数器向下溢出时，程序计数器将从 0x0000 0000 开始，和外部复位一样。可以检查看门狗超时标志（WDTOF）来确定看门狗是否产生复位条件。WDTOF标志必须由软件清零。看门狗结构方框图如图 4. 94所示。

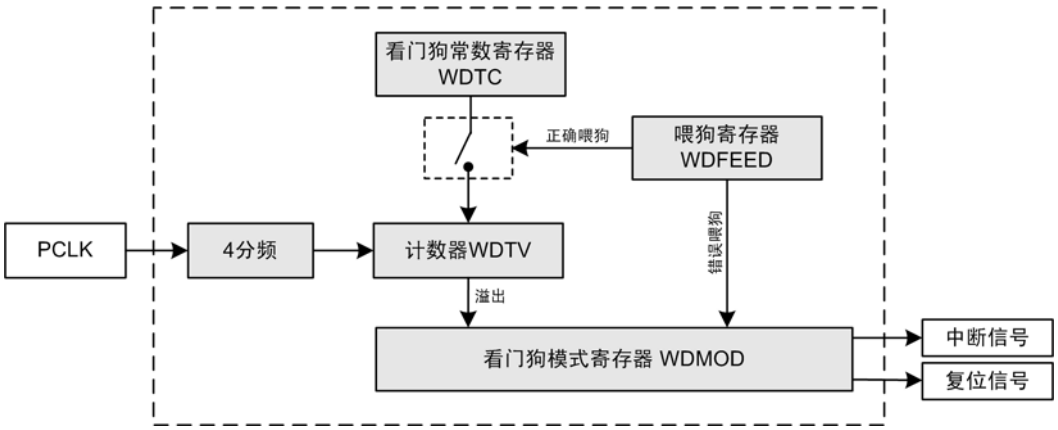


图 4.94 看门狗结构方框图

4.13.2 特性

- 如果没有周期性重装（即喂狗），则产生片内复位；
- 支持调试模式；
- 由软件使能看门狗定时器，但要求禁止硬件复位或看门狗复位/中断；
- 错误/不完整的喂狗时序会导致复位/中断（如果使能）；
- 指示看门狗复位的标志；
- 带内部预分频器的可编程 32 位定时器；
- 可选择 T_{PCLK}×4 倍数的时间周期：从(T_{PCLK}×256×4) 到 (T_{PCLK}×2^{32}×4)。

4.13.3 寄存器描述

WDMOD寄存器描述见表 4.121。

表 4.121 看门狗模式寄存器

WDMOD	功能	描述	复位值
0	WDEN	看门狗中断使能位（只能置位）	0
1	WDRESET	看门狗复位使能位（只能置位）	0

续上表

WDMOD	功能	描述	复位值
2	WDTOF	看门狗超时标志	0（外部复位）
3	WDINT	看门狗中断标志（只读）	0
7:4	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义	NA

操作示例

```
WDMOD = 0x02;                                   /* 看门狗溢出时，产生复位          */
```

看门狗的操作是通过WDEN位和WDRESET位来控制的，如表 4.122所示。

表 4.122 看门狗的操作控制

WDEN	WDRESET	操作
0	×	看门狗关闭时的调试/操作
1	0	带看门狗中断的调试，但没有 WDRESET
1	1	带看门狗中断和 WDRESET 的操作

注意：没有只复位无中断的组合。

一旦 WDEN、WDRESET 位置位，就无法使用软件将其清零。这两个标志由外部复位或看门狗定时器溢出清零。

注意：将 WDEN 设置为 1 只是使能 WDT，但没有启动 WDT，当第一次喂狗操作时才启动 WDT。

WDTOF 若看门狗发生超时，看门狗超时标志置位。该标志由软件清零。

WDINT 若看门狗发生超时，看门狗中断标志置位。任何复位都会使该位清零，无法使用软件将其清零。由于看门狗的中断标志位不能够通过软件清零，因此，发生看门狗中断时，只能够通过禁止看门狗中断的方式返回，如程序清单 4.48所示。

程序清单 4.48 看门狗中断返回

```
VICIntEnClr = 0x01;          /*  看门狗溢出中断只能通过禁止看门狗中断的方式返回  */
VICVectAddr = 0x00;          /*  通知 VIC 中断结束                              */
```

1. 看门狗定时器常数寄存器

WDTC寄存器决定看门狗超时值，见表 4.123。当喂狗时序产生时，WDTC的内容重新装入看门狗定时器。它是一个 32 位寄存器，最小值为 0xFF，若写入一个小于 0xFF的值，会使 0xFF装入WDTC。因此，WDT的最小时间间隔为$t_{pclk}\times256\times4$，最大时间间隔为$t_{pclk}\times2^{32}\times4$。WDTC寄存器描述见表 4.123。WDT初始化，如程序清单 4.49所示。

表 4.123 看门狗定时器常数寄存器

WDTC	功能	描述	复位值
31:0	计数值	看门狗超时间隔	0xFF

程序清单 4.49 WDT 初始化

```
WDTC = 0x10000;              /*  设置 WDT 定时值                */
WDMOD = 0x03;                /*  设置 WDT 工作模式，使能 WDT    */
```

2. 看门狗喂狗寄存器（WDFEED - 0xE0000008）

向该寄存器写入 0xAA，然后写入 0x55 会使WDTC的值重新装入看门狗定时器，WDFEED寄存器描述见表 4.124。如果看门狗通过WDMOD寄存器使能，该操作还将启动看门狗运行。在看门狗溢出之前，必须完成一次正确的喂狗时序。向WDFEED寄存器写入0xAA，然后向WDFEED寄存器写入 0x55，如程序清单 4.50所示。不正确喂狗时序之后的第二个pclk周期，看门狗复位/中断被触发。

表 4.124 看门狗喂狗寄存器

WDFEED	功能	描述	复位值
7:0	喂狗	喂狗值应当为 0xAA，然后是 0x55	未定义

程序清单 4.50 WDT 喂狗操作

```
WDFFED  =  0xAA;
WDFFED  =  0x55;
```

3. 看门狗定时器值寄存器（WDTV - 0xE000000C）

WDTV寄存器用于读取看门狗定时器的当前值，见表 4.125。

表 4.125 看门狗定时器值寄存器

WDTV	功能	描述	复位值
31:0	计数	当前定时器值	0xFF

4.13.4 WDT中断

从前面的描述可以看出，只要启动WDT，那么WDT就不能停止，而且，WDT溢出后便会触发中断，WDT中断与向量中断控制器（VIC）的关系如图 4.95所示。

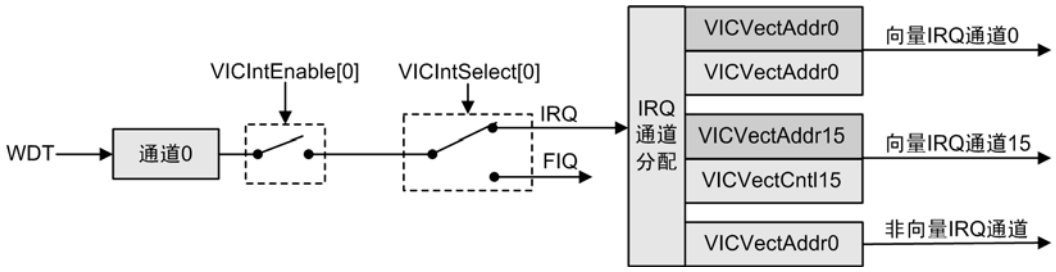


图 4.95 WDT 与 VIC 的关系

WDT 处于 VIC 的通道 0，中断使能寄存器 VICIntEnable 用来控制 VIC 通道的中断使能。当 VICIntEnable[0] = 1 时，通道 0 中断使能，即：WDT 中断使能。

中断选择寄存器 VICIntSelect 用来分配 VIC 通道的中断。当某一位为“1”时，对应的通道中断分配为 FIQ；当某一位为“0”时，对应的通道中断分配为 IRQ。VICIntSelect[0] 用来控制通道 0，即：

- 当 VICIntSelect[0] = 1 时，WDT 中断分配为 FIQ 中断；
- 当 VICIntSelect[0] = 0 时，WDT 中断分配为 IRQ 中断。

当分配为 IRQ 时，还需要设置对应的通道控制寄存器和地址寄存器。

此外，还需要说明一点：WDT 的中断标志位无法通过软件清零，只能通过硬件复位清零。因此，当发生 WDT 中断时，只能通过禁止 WDT 中断的方式返回，即，VICIntEnable[0] = 0。

4.13.5 应用示例

本实验演示看门狗溢出后复位现象，由于 LPC2103 复位后执行 Flash 内的代码，因此该实验需要将程序写入到 Flash 中，即在 ADS 中选择 DebugInFlash 或者 RelInFlash 进行调试。

本示例实验现象：在正常情况下，上电后LED1 闪烁一次熄灭，程序周期性喂狗，防止看门狗溢出；用按键KEY1 模拟导致无法周期性喂狗的意外情况，当意外情况发生，即KEY1 按下后，喂狗周期被打断，超过WDTC设定的时间，看门狗将会溢出并复位。复位后LED1 闪烁一次后熄灭，程序周期性喂狗……，程序回到正常情况。示例代码如程序清单 4.51所示。用户需短接跳线JP4 的P0.17 端口、JP3 的P0.16 端口。

程序清单 4.51 看门狗溢出复位实验程序

```
#include   "config.h"

#define    LED1      1 << 17                          /* P0.17 控制 LED1                */
#define    KEY1      1 << 16                          /* P0.16 产生看门狗溢出按键       */
/*********************************************************************************************
** 函数名称：DelayNS
** 功能描述：延时函数
** 入口参数：uiDly    值越大，延时时间越长
** 出口参数：无
*********************************************************************************************/
void DelayNS(uint32 uiDly)
{
    uint32 i;
    for(; uiDly > 0; uiDly--){
        for(i=0; i<50000; i++);
    }
}
/*********************************************************************************************
** 函数名称：WDTInit
** 功能描述：看门狗初始化
** 入口参数：uiTime    设置看门狗定时器参数
** 出口参数：无
*********************************************************************************************/
void WDTInit(uint32 uiTime)
{
    WDTC    = uiTime;                                /* 设置看门狗定时器参数           */
    WDMOD   = 0x03;                                  /* 设置看门狗模式：溢出后复位     */

    WDFEED  = 0xAA;                                  /* 第一次喂狗启动 WDT             */
    WDFEED  = 0x55;                                  /* 喂狗序列                       */
```

```
}

/*********************************************************************************************
** 函数名称：FeedDog
** 功能描述：喂狗函数
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
void FeedDog(void)
{
    IRQDisable ();                                  /* 禁止中断                    */
    WDFEED  = 0xAA;                                 /* 喂狗序列                    */
    WDFEED  = 0x55;
    IRQEnable ();                                   /* 喂狗序列                    */
}
/*********************************************************************************************
** 函数名称：main
** 功能描述：演示看门狗溢出复位，按键 KEY1 被按下后使得看门狗溢出复位，LED 闪烁一次
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
int main (void)
{
    uint32 i;
    PINSEL0  =    0x00000000;                       /* 管脚连接 GPIO               */
    IO0DIR   =    LED1;                             /* LED1 控制口输出             */
    IO0CLR   =    LED1;                             /* 关闭蜂鸣器                  */
    DelayNS(100);
    IO0SET   =    LED1;                             /* 关闭蜂鸣器                  */

    WDTInit(0x200000);                              /* 看门狗初始化                */
    while(1){
        /* 用按键模拟导致无法周期性喂狗的意外情况  */
        while((IO0PIN & KEY1) == 0);                /* 如果按键 1 按下，就停止喂狗  */
        for(i = 0; i < 0xFF; i++);                  /* LED1 闪烁周期               */
        FeedDog();                                  /* 喂狗                        */
    }
    return 0;
}
```

4.14 PLL

4.14.1 概述

PLL接受的输入时钟频率范围仅为 10MHz~25MHz。输入频率通过一个电流控制振荡器（CCO）倍增到范围 10MHz~70MHz。倍频器可以是从 1 到 32 的整数（实际倍频值受 CPU 最高频率的限制，LPC2103 最高频率可达到 70MHz）。CCO的操作频率为 156MHz~320MHz，因此在环中有一个额外的分频器在PLL提供所需要的输出频率时使CCO保持在其频率范围内。输出分频器可设置为 2、4、8 或 16 分频来产生输出时钟。由于输出分频器的最小值为 2，它保证了PLL输出有 50%的占空比。对于PLL在时钟系统中所处的位置，以及PLL内部逻辑原理如图 4.96所示。

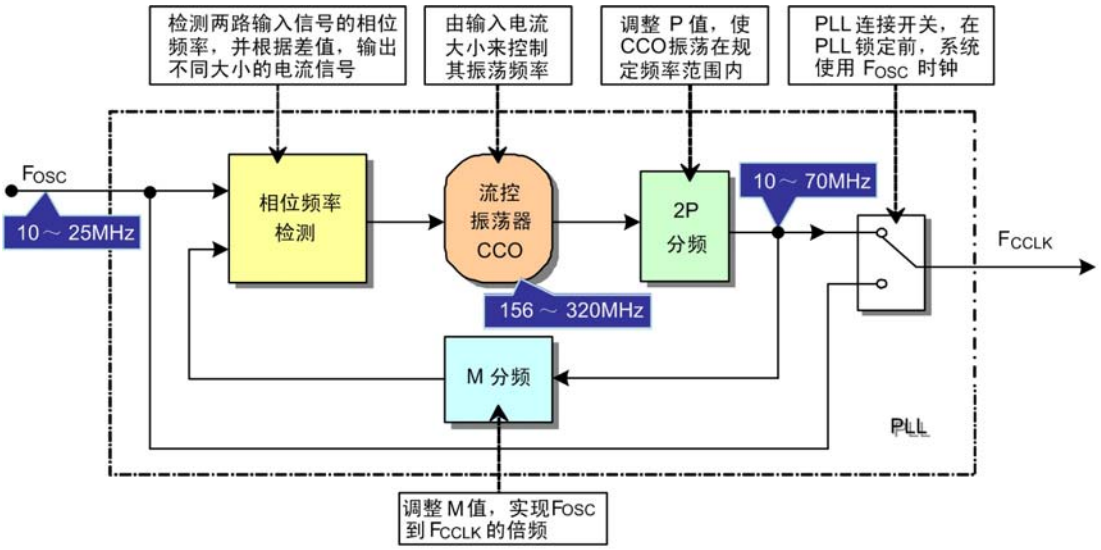


图 4.96 PLL 频率输入输出范围

PLL 的激活由 PLLCON 寄存器控制。PLL 倍频器和分频器的值由 PLLCFG 寄存器控制。为了防止 PLL 参数发生意外改变或 PLL 失效，对这两个寄存器进行了保护。当提供芯片时钟时，由于芯片的所有操作（包括看门狗定时器）都依赖于 PLL，因此 PLL 设置的意外改变将导致微控制器执行不期望的动作。对它们的保护由一个类似于操作看门狗定时器的馈送(feed)序列来实现。PLL 在芯片复位和进入掉电模式时被关闭并旁路。PLL 只能通过软件使能。程序必须在配置并激活 PLL 后等待其锁定，然后作为时钟源连接到 PLL。

4.14.2 寄存器描述

PLL由表 4.126所示的寄存器进行控制。

警告：PLL 值的不正确设定会导致器件的错误操作！

表 4.126 PLL 寄存器

通用名称	描述	访问	复位值[1]	地址
PLLCON	PLL 控制寄存器。保持寄存器用于更新 PLL 控制位。写入该寄存器的值在有效的 PLL 馈送序列执行之前不起作用	R/W	0	0xE01F C080
PLLCFG	PLL 配置寄存器。保持寄存器用于更新 PLL 配置值。写入该寄存器的值在有效的 PLL 馈送序列执行之前不起作用	R/W	0	0xE01F C084

续上表

通用名称	描述	访问	复位值[1]	地址
PLLSTAT	PLL 状态寄存器。PLL 控制和配置信息的读回寄存器。如果曾对 PLLCON 或 PLLCFG 执行了写操作，但没有产生 PLL 馈送序列，这些值将不会反映 PLL 的当前状态。读取该寄存器提供了控制 PLL 和 PLL 状态的实际值	RO	0	0xE01F C088
PLLFEED	PLL 馈送寄存器。该寄存器使能装载 PLL 控制和配置信息，该配置信息从 PLLCON 和 PLLCFG 寄存器装入实际影响 PLL 操作的映像寄存器中	WO	NA	0xE01F C08C

[1] 复位值仅反映已使用位中保存的数据，它不包括保留位的内容。

4.14.3 PLL配置过程

如果一个特定的应用使用 PLL，它的配置必须依照下面的原则：

- 选择需要的处理器操作频率（CCLK）。这可以根据处理器的整体要求、UART 波特率特定设置的支持等因素来决定。记住外围器件的时钟频率可以低于处理器频率；
- 选择振荡器频率（F_{OSC}）。CCLK 一定是 F_{OSC} 的整数（非小数）倍；
- 计算M值以配置MSEL位。M = CCLK/F_{OSC}，M的取值范围为 1~32。在PLLCFG中，写入MSEL位的值为M-1(见表 4.128)；
- 选择P值以配置PSEL位，使F_{CCO}在定义的频率限制范围内，F_{CCO}可通过前面的等式计算。P必须是 1, 2, 4 或 8 其中的一个。写入PLLCFG中PSEL位的值为：00 表示P=1；01 表示P=2；10 表示P=4；11 表示P=8 (见表 4.127)。

表 4.127 PLL 分频器值

PSEL 位（PLLCFG 位[6:5]）	P 值
00	1
01	2
10	4
11	8

表 4.128 PLL 倍频器值

MSEL 位（PLLCFG 位[4:0]）	M 值
00000	1
00001	2
00010	3
00011	4
……	……
11110	31
11111	32

4.14.4 PLL操作

1. 查询方式

- PLLCON＝1——设定 PLL 之前，必须使能 PLL，但不能连接 PLL；
- 设定 P 和 M 的值（PLLCFG）；
- 发送 PLL 馈送序列；
- 等待 PLL 锁定——PLLSTAT. 10 ＝ 1；
- PLLCON＝3——设定 P 和 M 之后，连接 PLL；
- 发送 PLL 馈送序列，把 P 和 M 的值写入硬件。

2. 中断方式（假定已经使能和正确设置PLL中断）

- PLLCON＝1——使能但不能连接 PLL；
- 设定 P 和 M 的值；
- 发送 PLL 馈送序列。

PLL 中断服务程序：

- PLLCON＝3——连接 PLL；
- 发送 PLL 馈送序列，把 P 和 M 的值写入硬件；
- 禁止 PLL 中断，返回。

3. 配置 PLL 的应用程序

系统设计要求 Fosc ＝ 10MHz 和要求 CCLK ＝ 60MHz。

根据这些要求，可得出 M = CCLK / Fosc ＝ 60MHz/10MHz＝6。因此，M-1 ＝ 5 将写入 PLLCFG[4:0]。

P值可由P = Fcco/(CCLK×2)得出，Fcco必须在 156MHz~320MHz的范围内。假设Fcco取最低频率 156MHz，则P=156MHz/(2×60MHz)＝1.3。Fcco取最高频率可得出P=2.67。因此，P的唯一解决方案是同时满足Fcco最低和最高频率的值，见表 4.127中列出的P=2。所以，将使用PLLCFG[6:5]=1。最终得到PLLCFG的值为 0x25。

4. 设定PLL序列

```
PLLCON      =   1;                          /*  使能 PLL                    */
PLLCFG      =   0x25;                       /*  设置 M 为 6，P 为 2         */
PLLFEED     =   0xAA                        /*  发送 PLL 馈送序列           */
PLLFEED     =   0x55;
while((FLLSTAT & (1 << 10)) == 0);          /*  等待 PLL 锁定               */
PLLCON      =   3;                          /* PLL 使能和锁定               */
PLFEED      =   0xAA;                       /*  发送 PLL 馈送序列           */
PLFEED      =   0x55;
```

4.14.5 应用示例

1. PLL查询方式实验

采用查询方式操作PLL，在PLL成功锁定后，LED闪烁一次。实现程序代码如程序清单4.52所示。用户需短接JP4 的P0.17 端口，输出控制LED闪烁。

程序清单 4.52 PLL 查询方式实验

```
#include "config.h"

#define   LED1    1 << 17;                                    /* P0.17 控制 LED1，低电平点亮      */
/*********************************************************************************************
** 函数名称：main
** 函数功能：PLL 查询方式实验
** 入口参数：无
** 出口参数：0
*********************************************************************************************/
int main (void)
{
    PINSEL1   = 0x00000000;                                   /* 设置管脚连接 GPIO                */
    IO1DIR    = LED1;                                         /* 设置 LED1 控制口为输出           */
    IO1SET     = LED1;                                        /* 关闭 LED                         */

    PLLCON = 1;                                               /* 使能 PLL                         */
    PLLCFG = 0x23;                                            /* 设置 M 为 4，P 为 2              */
    PLLFEED = 0xaa;                                           /* 发送 PLL 馈送序列                */
    PLLFEED = 0x55;

    while((PLLSTAT & (1 << 10)) == 0);                        /* 等待 PLL 锁定                    */
    PLLCON = 3;                                               /* PLL 使能和连接                   */
    PLLFEED = 0xaa;                                           /* 发送 PLL 馈送序列                */
    PLLFEED = 0x55;

    IO1CLR = LED1;                                            /*点亮 LED                          */
    return 0;
}
```

2. PLL中断实验

采用中断方式打开PLL，在PLL中断服务函数中添加了LED闪烁用于指示PLL锁定。当PLL锁定成功后，LED闪烁一次。实现程序代码如程序清单 4.53所示。

程序清单 4.53 PLL 中断实验程序

```
#include "config.h"

#define    LED      1 << 17                                   /* P0.17 控 LED 闪烁                */
/*********************************************************************************************
** 函数名称：DelayNS
** 功能描述：延时函数
** 入口参数：uiDly    值越大，延时时间越长
** 出口参数：无
*********************************************************************************************/
```

```
void DelayNS(uint32 uiDly)
{
    uint32 i;
    for(; uiDly > 0; uiDly--){
        for(i=0; i<50000; i++);
    }
}
/*********************************************************************************************
** 函数名称：PLL_INT
** 功能描述：中断
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
void __irq PLLINT (void)
{
    PINSEL1  =  0x00000000;                    /* 设置所有管脚连接 GPIO          */
    IO0DIR   =  LED;                           /* 设置 LED 控制口输出            */

    PLLCON   =  3;                             /* PLL 使能和连接                 */
    PLLFEED  =  0xAA;                          /* 发送 PLL 馈送序列              */
    PLLFEED  =  0x55;

    IO0CLR   =  LED;                           /* LED 闪烁 1 次                  */
    DelayNS(200);
    IO0SET   =  LED;

    VICIntEnClr =  1 << 12;                    /* 禁能 PLL_INT 中断              */
    VICVectAddr =  0x00;                       /* 向量中断处理结束               */
    }
/*********************************************************************************************
** 函数名称：main
** 功能描述：在中断中打开 PLL，LED 闪烁一次
** 入口参数：无
** 出口参数：无
*********************************************************************************************/
int main (void)
{
    PINSEL0  =  0x00000000;                    /* 设置管脚连接 GPIO              */
    IO0SET   =  LED;                           /* LED 控制口输出                 */

    PLLCON   =  1;                             /* 使能 PLL                       */
    PLLCFG   =  0x25;                          /* 设置 M 为 6，P 为 2            */

    PLLFEED =  0xAA;                           /* 发送 PLL 馈送序列              */
```

```
    PLLFEED =     0x55;

    IRQEnable();                                        /*  使能 IRQ 中断                */
    /*  设置 PLL 中断          */
    VICIntSelect    =    0x00000000;                    /*  设置所有的通道为 IRQ 中断    */
    VICVectCntl0    =    0x20 | 12;                     /* PLL_INT 分配到最高优先级      */
    VICVectAddr0    =    (uint32)PLLINT;                /*  设置 PLL_INT 向量地址       */
    VICIntEnable    =    1 << 12;                       /*  使能 PLL_INT 中断           */

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