第七章 串行通信与外设

1 概论

计算机与外界所进行的信息交换被人们称为“数据通信”。

1.1 串行通信的分类

1.1.1按基本方式分，可以分为串行通信和并行通信。

所谓串行通信，就是数据一位一位的传输，只需一位或二位数据线，线路简单，传输距离较长，但传输速度略慢。

并行通信：数据多位一起传输（如 8 位），硬件复杂，传输速度快，但仅限短距离传输。

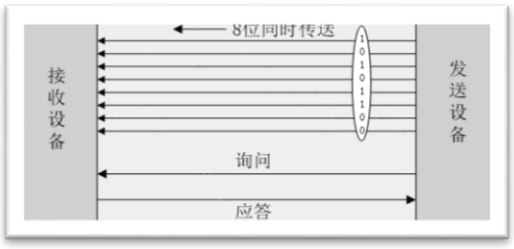


图 7.1 并行通信示意图

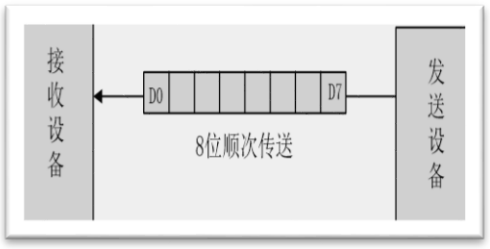
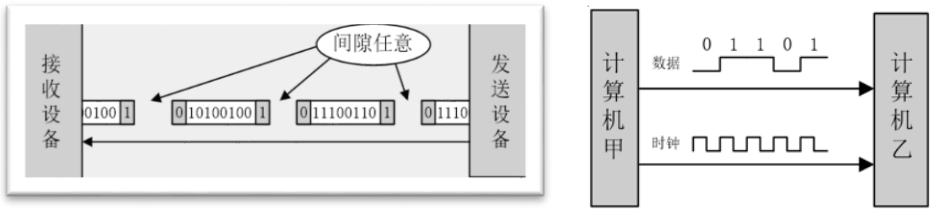


图 7.2 串行通信示意图

鉴于此，现在多使用串行通信。

1.1.2 根据有无同步信号，可以分为异步通信和同步通信。

异步通信如图 7.3（a）所示和同步通信如图 7.3（b）所示。异步通信常见的为 Uart，同步通信常用的有 SSI 通信和 I2C 通信，。



（a） （b）

图 7.3 异步通信和同步通信

异步通信是指通信的发送与接收设备使用各自的时钟控制数据的发送和接收过程。为使双方的收发协调，要求发送和接收设备的时钟尽可能一致。

异步通信是以字符（构成的帧）为单位进行传输，字符与字符之间的间隙（时间间隔）是任意的，但每个字符中的各位是以固定的时间传送的，即字符之间不一定有“位间隔”的整数倍的关系，但同一字符内的各位之间的距离均为“位间隔”的整数倍。

同步通信时要建立发送方时钟对接收方时钟的直接控制，使双方达到完全同步。此时，传输数据的位之间的距离均为“位间隔”的整数倍，同时传送的字符间不留间隙，即

保持位同步关系，也保持字符同步关系。

1.1.3 如果按照通信方向来分类，可以分为单工、半双工和全双工三种方式。

① 单工方式：

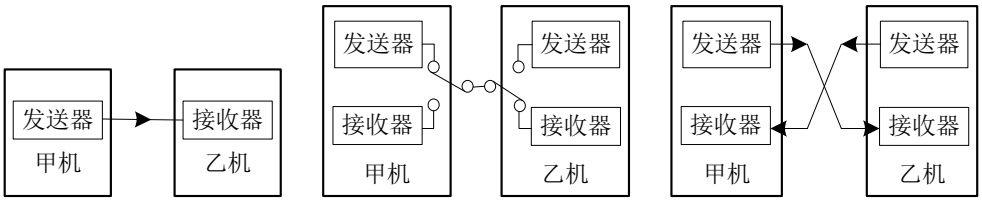
信号（不包括联络信号）在信道中只能沿一个方向传送，而不能沿相反方向传送的工作方式称为单工方式，如图 7.4（a）所示。例如一些遥控、遥测信号通信。

② 半双工方式：

通信的双方均具有发送和接收信息的能力，信道也具有双向传输性能，但是，通信的任何一方都不能同时既发送信息又接收信息，即在指定的时刻，只能沿某一个方向传送信息。这样的传送方式称为半双工方式如图 7.4（b）所示。例如 I2C 串行接口的应用。

③ 全双工方式

若信号在通信双方之间沿两个方向同时传送，任何一方在同一时刻既能发送又能接收信息，这样的方式称为全双工方式，如图 7.4（c）所示。例如 SPI 串行接口的应用。

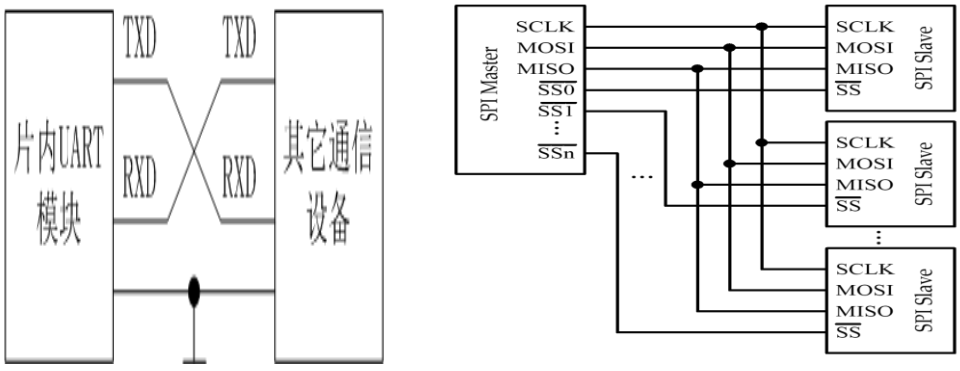


（a） （b） （c）

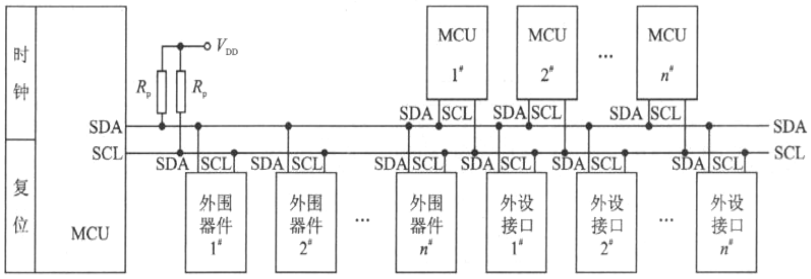
图 7.4 通信方向示意图

1.1.4 如果按参与通讯的机器数量来分，可以分为两机通信和多机通讯

两机通信是两个机器之间的通信，如图 7.5（a）所示，为点对点通信模式，常用的有 UART 通信；多机通信通常为单主机与多从机进行通信，如图 7.5（b）、（c）所示，常用的有 SSI 通信和 I2C 通信模式，为点对多通信模式。



（a）UART 通信模式 （b）SSI 通信模式



（c）I2C 通信模式

图 7.5 两机通信和多机通信示意图

1.2 几种常用的串行通信方式及特性

串行通信的实现在制式、种类、形式、规范、标准、编码、检错、纠错、帧结构、组网方式、调制方式、主要用途等许多方面，存在着多种类型、变化、选择和解决方案。比如，Philips 公司发明的 I2C 总线，Intel 等公司提出的 SMBus 总线，MOTOROLA 公司首先应用的 SPI 接口，国家半导体（NSC）公司首先应用的 MicroWire 接口，DALLAS 公司推出的 1—Wire 总线，美国电子工业协会推荐标准 RS—232、RS—422、RS—485 接口，Intel 公司提出 USB 总线，Apple 等公司提出的 IEEE—1394（俗称“火线”），BOSCH 公司提出的 CAN 总线等，都是用来实现与串行通信功能相关的技术和规范。

三种最基本的串行通信方式是：UART、SSI(SPI)、IIC(I2C) 。UART (Universal Asynchronous Receiver/Transmitter) 通用异步收发器，主要用于嵌入系统与 PC 或两个嵌入系统之间的串行通信；SSI 接口（Synchronous Serial Interface ），意为串行外围接口(高速,最高可到 2M，甚至更高)；I2C(Inter-Integrated Circuit)总线是用于连接微控制器及其外围设备（低速，常用于标准速度 100Kbps 和高速 400Kbps)。这三种虽然同属串行通信范畴，但适应范围和场合不一样，三种串行通信方式的比较如表 7.1 所示。

表 7.1 三种串行通信方式的比较

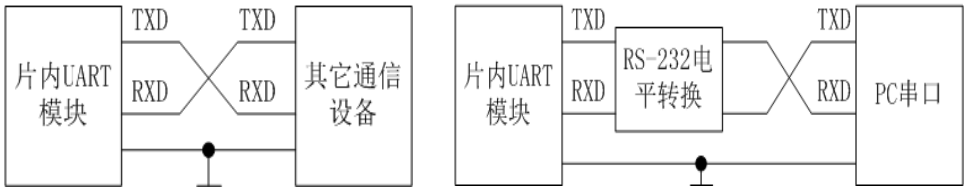
通信方式	作用对象	同步信号	连接线	通信方向	主要用途
UART	系统层或芯片层	无（用波特率实现同步）	RXD、TXD、GND	全双工	适用于点对点的串行通信
SSI(SPI)	芯片层	有同步时钟信号	MOSI、MISO、SCLK、SS#	全双工	适用于系统内数据的传输
IIC(I2C)	芯片层	有同步时钟信号	SDA、SCL	半双工	适用于系统内数据的传输

2 异步串行通信 UART

2.1 UART 的基本原理

UART，英文全称为 Universal Asynchronous Receiver/Transmitter,译成“通用异步串行接收/发送器”，人们常常称为串口。UART 模块将微处理器内部的并行数据转换为串行数据，通过串行总线 UnTX 以异步通信的方式发送出去；另一方面它也可以接收 UnRX 总线上的串行数据，转换为并行数据后返回给微处理器进行处理。UART 总线采用双向通信，可以实现

全双工的发送和接收。在嵌入式设计中，UART 用来与计算机或其他设备进行通信。使用时的基本连接图如图 7.6（a）和图 7.6（b）所示。



（a） UART模块之间通信连接图 （b）UART模块与PC之间通信连接图

图7.6 UART通信示意图

UART 通信的数据帧格式如下图 7.7 所示。

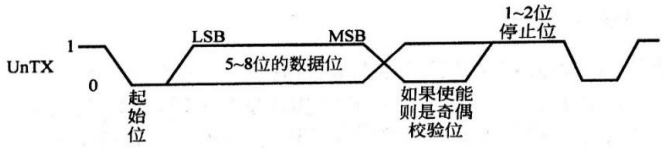


图 7.7 UART 通信的数据帧格式

LSB 为最低有效位，MSB 为最高有效位，数据从 LSB 开始传输。

数据线上空闲时为高电平，起始位为低电平。传输时，首先要约定通信协议，包括每一帧的位数—5~8 位数据位、波特率、校验位、停止位（1 或者 2 位停止位--高电平）等，协议确定后每一帧即是固定的。

在通信过程中往往要对数据传送的正确与否进行校验。校验是保证准确数据无误传输的关键。常用的校验方法有奇偶校验、代码和校验及循环冗余码校验。

奇偶校验就是双方对比数据中“1”的个数的奇偶性。在发送数据时，数据位尾随的 1 位为奇偶校验位（1 或 0）。若为奇校验，数据中“1”的个数与校验位“1”的个数之和应为奇数；若为偶数，则为偶校验。接收字符时，对“1”的个数进行校验，若发现不一致，则说明传输数据过程中出现了差错。

和校验就是双方对比数据之和，还有循环冗余码校验等。

几乎所有的 MCU 都把 UART 作为一个最基本的通信接口，用来实现与其他嵌入式设备或 PC 的数据通信（点对点）。有一些比较高档的 UART 还提供输入输出数据的缓冲区。

2.2 TM4C123GH6PM 的 UART 配置

TM4C123GH6PM 微控制器配备了**8 路UART**，而且拥有以下特征：

- 配备可编程波特率发生器，能够达到5Mbps（16分频）的常规速度以及10Mbps的最高速度（8分频）
- 每路UART都具有16*8的发送和接收缓冲区FIFO来降低CPU的中断服务程序加载
- FIFO 长度可编程，提供常规的双缓冲接口来实现1 字节的深操作

- FIFO 的触发电平可以为1/8,1/4,1/2,3/4,7/8
- 含开始，停止，校验的标准异步通信位
- 断线的产生和检测
- 完全可编程的串行接口参数

  —5，6，7，8 数据位

  —支持奇校验、偶校验、空格或者无校验位

  —1 或者2 位的停止位
- 提供IrDA SIR编码器/解码器

  —可编程使用的串行红外或者UART 输入/输出

  —半双工的115.2Kbps 数据率的IrDA 串行红外编码器/解码器功能

  —对正常情况3/16 和低功耗比特持续时间（1.41-2.23us）的支持

  —可编程内部时钟发生器，根据低功耗模式位持续时间分频参考时钟，分频系数1-256
- 对ISO7816 智能卡通讯的支持
- 调制解调器流控制（仅UART1）
- 支持9位的EIA-485
- 标准FIFO电平和传送结束中断
- 通过uDMA支持高效传输

  —发送和接收独立

  —数据在FIFO时接收单个请求断言，在可编程FIFO 电平时处理断言

  —当FIFO有空间时发送单个请求断言，在可编程FIFO 电平时处理断言

2.3 内部结构

2.3.1内部结构框图

每个UART模块主要包括三部分，即寄存器、FIFO以及收发装置。其结构框图如图7.8所示。其中粗线表示通信数据信号的传输，细线表示控制/状态信号的传输。

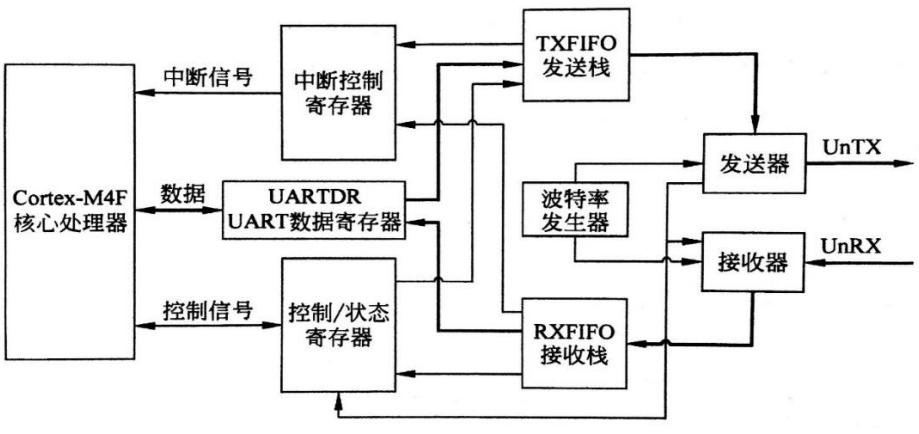


图 7.8 UART 结构框图

图中的寄存器主要有三种：UART数据寄存器、中断控制寄存器以及控制/状态寄存器。其中数据寄存器用于暂存具体的数据信号。中断控制寄存器包括多个寄存器，跟GPIO、GPTM模块一样，为UARTIM（中断屏蔽寄存器）、UARTMIS（原始中断状态寄存器）、UARTRIS（中断屏蔽状态寄存器）、UARTICR（中断清除寄存器），主要用于UART通信过程中的中断控制。而控制/状态寄存器则用于对UART通信的设置以及状态的监控。此外，除了这3种寄存器外，还有诸如标识寄存器、DMA控制寄存器等其他功能寄存器。

中间部分是栈FIFO，即数据缓冲区，主要用于存储待处理的并行数据。共有2个16×8位的FIFO，其中TXFIFO用于存储待发送的数据，RXFIFO用于存储已接收的数据。

发送器和接收器主要起到一个“并-串”数据转换的作用。发送器将TXFIFO中待发送的并行数据转换为串行数据通过总线UnTX发送出去，而接收器则将总线UnRX上的串行数据转换为并行数据存入RXFIFO中。

TM4C123GH6PM 的 UART 具体寄存器结构如图 7.9 所示。

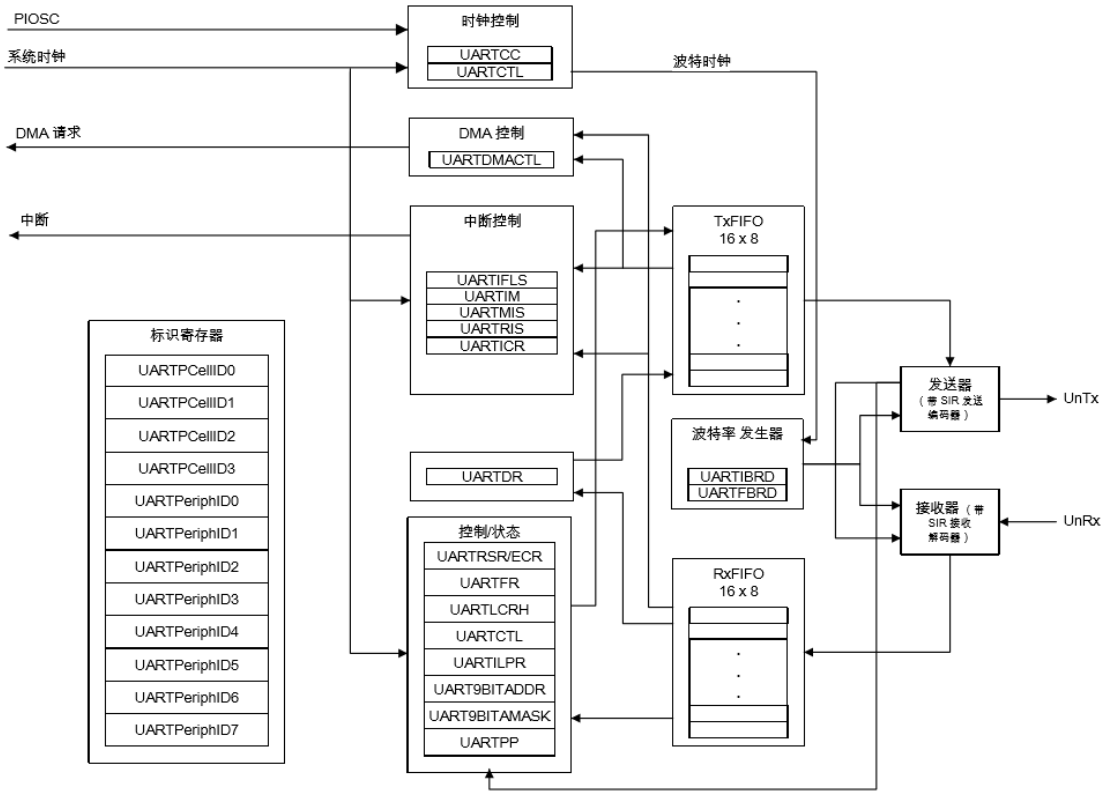


图 7.9 TM4C123GH6PM 的 UART 寄存器结构图

2.3.2引脚复用

表7.2中列举了UART 模块的外部信号，并描述了各自的功能。UART 信号是一些GPIO 信号的复用功能，除了UORx 和UOTx 引脚在默认情况下是UART 功能，其它的在复位之后默认都是GPIO 功能。如果要要做UART功能，需要配置GPIOAFSEL寄存器。配置GPIO 请参考“通用输入/输出端口”章节。

表7.2 UART 信号（64LQFP）

引脚名称	引脚序号	引脚复用/引脚分配	引脚类型	缓冲区类型a	描述
U0Rx	17	PA0(1)	I	TTL	UART 模块0 接收端
U0Tx	18	PA1(1)	0	TTL	UART 模块0 发送端
U1CTS	15 29	PC5(8) PF1(1)	I	TTL	UART 模块1 清除发送硬件
U1RTS	16 28	PC4(8) PF0(1)	0	TTL	UART 模块1 请求发送硬件流控制输出端
U1Rx	16 45	PC4(2) PB0(1)	I	TTL	UART 模块1 接收端
U1Tx	15 46	PC5(2) PB1(1)	0	TTL	UART 模块1 发送端
U2Rx	53	PD6(1)	I	TTL	UART 模块2 接收端
U2Tx	10	PD7(1)	0	TTL	UART 模块2 发送端
U3Rx	14	PC6(1)	I	TTL	UART 模块3 接收端
U3Tx	13	PC7(1)	0	TTL	UART 模块3 发送端
U4Rx	16	PC4(1)	I	TTL	UART 模块4 接收端
U4Tx	15	PC5(1)	0	TTL	UART 模块4 发送端
U5Rx	59	PE4(1)	I	TTL	UART 模块5 接收端
U5Tx	60	PE5(1)	0	TTL	UART 模块5 发送端
U6Rx	43	PD4(1)	I	TTL	UART 模块6 接收端
U6Tx	44	PD5(1)	0	TTL	UART 模块6 发送端
U7Rx	9	PE0(1)	I	TTL	UART 模块7 接收端
U7Tx	8	PE1(1)	0	TTL	UART 模块7 发送端

2.4 Tiva-C 的 UART 数据传送过程

数据接收和发送都保存在两个 16×8 位的 FIFO 缓冲区中。

当数据写到发送缓冲区时 UART 标志寄存器的 BUSY 位会立即被置为有效，并且在数据发送过程中保持有效状态。

只有当发送缓冲区没有数据,并且包括发送位的最后一个字节也已经从数据位移寄存器发送出去，BUSY 位才会取消有效状态。

当接收器处于空闲状态，并且数据输入变成低电平，那么接收计数器开始运行而且数据会在一定周期后采样。如果仍然是低电平，那么开始位就是有效和可识别的，否则就会被忽略。

如果启用校验模式那么还会检查校验位。

当 UnRx 信号变成高电平时候，会确认一个有效的停止位，否则会发生帧错误。

缓冲区可供选择的配置包含 1/8,1/4,1/2,3/4,7/8，例如 1/2 会在发送或接收到 8 个数

据（1/2×16=8）时产生中断。

缓冲区大小用函数设置：UARTFIFOLevelSet(ui32Base, ui32TxLevel, ui32RxLevel);

2.5 波特率

UART数据的收发速度是由波特率决定的，波特率的单位为位/秒（bit/s），即表示每秒钟发送多少位数据。通常选取115200bit/s。

波特率除数是一个22 位的数字，由16 位的整数部分和6 位的小数部分组成。波特率发生器可以使用这两个部分组成的数字决定位周期。

波特率除数（BRD）和系统时钟有如下关系：

BRD = BRDI + BRDF =UARTSysClk / (ClkDiv * Baud Rate)

可以直接调用函数实现：

UARTConfigSetExpClk(ui32Base, ui32UARTClk, ui32Baud, ui32Config);

其中UARTSysClk 是连接到UART 的系统时钟，ClkDiv 要么是16（如果UARTCTL 寄存器中HSE位清零）要么是8（如果HSE 置1）。它表示一个信号“1”或“0”持续16个UART系统时钟周期；还是在高速模式下ClkDiv=8，持续8个UART系统时钟周期。设置好比特率后，系统会自动计算好并赋值该寄存器。

2.6 Tiva-C 的 UART 中断

当以下条件发生时 UART 产生中断：

- 溢出错误：当 FIFO 已满时又收到新的数据，导致数据丢失；
- 间隔错误：接收端被拉低的时间过长，超过一个完整字的传输时间（指包含起始位、数据位、奇偶校验位、停止位的完整一帧时间）；
- 校验错误：接收到的奇偶校验位错；
- 帧错误：在应该接收到停止位的时候，接收到逻辑的“0”；
- 接收超时：当接收方 FIFO 不为空时，且在一个 32 位周期内（HSE 清零时）或在一个 64 位周期内（HSE 置位时）不再接收数据，接收超时中断有效；
- 发送：发送 FIFO 到达设置的触发深度（EOT=0）；

  数据全部发完（EOT=1）；
- 接收：接收 FIFO 到达设置的触发深度。

UART 的中断与 GPIO 模块一样，有 4 个主要的中断寄存器，包括：中断屏蔽寄存器(UARTIM)；原始中断状态寄存器（UARTRIS）；屏蔽中断状态寄存器 (UARTMIS) ；中断清除寄存器（UARTICR），作用原理也相同，可参阅相关章节。

所有的中断会在送入中断控制器之前进行或操作，所以 UART 在任意给定的时间只能产生一个中断请求。软件通过读 UART 屏蔽中断状态寄存器的值，可以在单个的中断服务程序处理多个中断事件。

通过写 1 到 UART 中断清除寄存器（UARTICR）总是可以清除所有 UART 中断（UARTMIS 和 UARTRIS 寄存器中的）。

2.7 回环模式 Loopback

回环模式是一个用于诊断和调试 UART 的运行模式。顾名思义，回环就是把 UnTX 和 UnRX 这两根线在内部连接起来，这样自己发送出去的数据就可以被自己接收，以进行调试和诊断。进入 Loopback 需要将寄存器 UARTCTL 中的 LBE 位置为 1。

2.8 初始化配置

要启用和初始化 UART，需要按照以下步骤进行：

1. 使用 RCGCUART 寄存器使能 UART 模块。（通过调用 SysCtlPeripheralEnable（）函数实现）

2. 使用 RCGCGPIO 寄存器使能到 GPIO 模块的时钟。

3. 设置 GPIO 相应引脚的 AFSEL 位。

4. 根据选择模式将 GPIO 配置成指定的电压水平和/或转换速率。（通过调用 GPIOPinConfigure（）函数实现）

5. 配置 GPIOPCTL 的 PMCn 域将 UART 信号分配到合适的引脚上。（通过调用 GPIOPinTypeUART（）函数实现）

6.配置波特率、校验位等串口参数。（通过调用 UARTConfigSetExpClk（）函数实现）

2.9 实验设计

本章将使用 LaunchPad 的 UART 与 PC 进行数据通信：使用 PC 向 LaunchPad 的 UART 发送数据，而后 Lauchpad 将此数据回传给 PC 端，程序十分简单。

2.9.1 程序设计

设备初始化步骤:

1 设备时钟设置

2 USART 外设使能，GPIO 外设使能

3 GPIO 端口模式设置

4 串口参数初始化

5 接收发送字符

实验流程如图 7.10 所示：

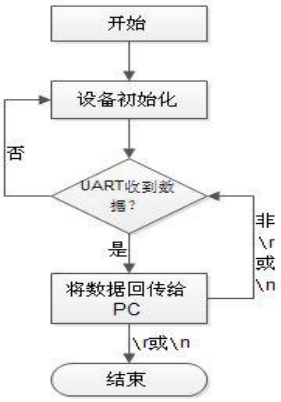


图 7.10 串口通信实验流程图

2.9.2 程序代码

```
int main(void)
{
    char cThisChar;
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
    SYSCTL_XTAL_16MHZ); //配置设备时钟为 16MHz，时钟源为外部晶振
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0); //外设使能
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    GPIOPinConfigure(GPIO_PA0_U0RX); //GPIO 引脚配置
    GPIOPinConfigure(GPIO_PA1_U0TX);
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
    UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
            (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
    UART_CONFIG_PAR_NONE)); //配置 UART 参数
    UARTCharPut(UART0_BASE, '!'); //发送‘!’字符表示设备初始化成功，等待接收字符
    do{
        cThisChar = UARTCharGet(UART0_BASE);
//将接收到的字符发送；判断是否为‘\r’或者‘\n’，是则结束循环，否则进入下一循环
        UARTCharPut(UART0_BASE, cThisChar);
    }
    while((cThisChar != '\n') && (cThisChar != '\r'));
    While（1）{}
}
```

2.9.3使用的库函数

1、GPIOPinTypeUART();//配置引脚作为 UART 外设

原型：voidGPIOPinTypeUART(uint32_t ui32Port, uint8_t ui8Pins)

参数： ui32Port 是 GPIO 的基地址； ui8Pins 是特定的引脚

如：GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

2、UARTConfigSetExpClk();//设置 UART 的参数，如端口、波特率等

UARTConfigSetExpClk(uint32_t ui32Base, uint32_t ui32UARTClk, uint32_t ui32Baud, uint32_t ui32Config)

参数： ui32Base 为 UART 端口基地址； ui32UARTClk 为提供给 UART 模块的时钟频率；ui32Baud 为波特率； ui32Config 为端口数据格式，包含配置信息（数据位、停止位、校验位）

如：UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,(UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));

3、UARTCharPut();//发送字符到指定的端口的发送缓冲区

原型：void UARTCharPut(uint32_t ui32Base,unsigned char ucData)

参数：ui32Base 为 UART 端口的基地址；ucData 为待发送的字符

如：UARTCharPut(UART0_BASE, 'E');

4、UARTCharGet();//从指定端口接收字符

如：UARTCharPut(UART0_BASE, UARTCharGet(UART0_BASE))

5、UARTCharsAvail();//确认在接收 FIFO 里是否存在字符

如：UARTCharsAvail(UART0_BASE)

6、UARTSpaceAvail();//确认在发送 FIFO 里是否有可利用的空间

2.9.4常用的 UART 寄存器

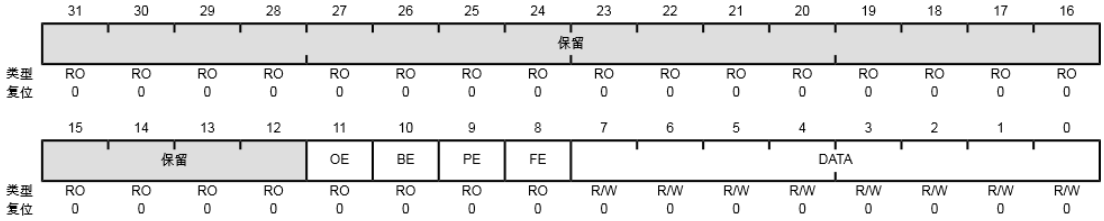
特殊功能寄存器的介绍详见TM4C123GH6PM中文数据手册P770~816。可以了解一下特殊功能寄存器：

1、UART 数据寄存器 (UARTDR)：

当接收数据时，若 FIFO 已使能，则收到的数据字节以及 4 个状态位（线中止错误、帧错误、奇偶校验错误、溢出错误）将推入 12 位宽的接收 FIFO 中。如果接收 FIFO 被禁用，则数据字节和状态位保存在接收保持寄存器（即接收 FIFO 的最底部单元）中。对 UARTDR 寄存器的读操作即可获取数据字节。

UART 数据寄存器 (UARTDR)

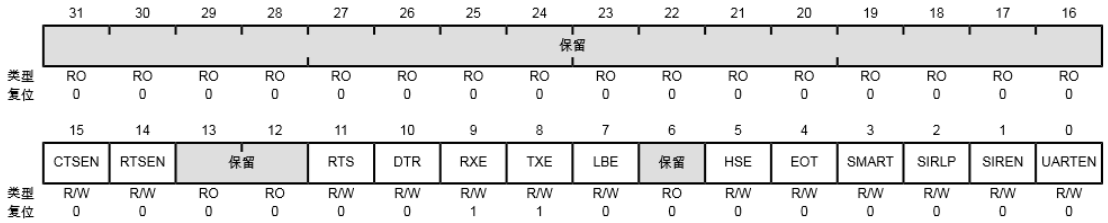
UART0 基址: 0x4000.C000
UART1 基址: 0x4000.D000
UART2 基址: 0x4000.E000
UART3 基址: 0x4000.F000
UART4 基址: 0x4001.0000
UART5 基址: 0x4001.1000
UART6 基址: 0x4001.2000
UART7 基址: 0x4001.3000
偏移量 0x000
类型 R/W, 复位 0x0000.0000



位/域	名称	类型	复位	描述
31:12	保留	RO	0x0000.0	软件不应该依赖保留位的值。为了兼容未来的器件，保留位的值在读 - 修改 - 写操作过程中应当保持不变。
11	OE	RO	0	UART溢出错误 值 描述 0 未发生数据溢出丢失 1 当FIFO已满时又收到新的数据，导致数据丢失

BE， UART 线中止错误；PE， UART 奇偶校验错误；FE， UART 帧错误。

2、UARTCTL(UART 控制寄存器)：UARTCTL 寄存器称为控制寄存器。复位后，除了启用发送位（TXE）和启用接收位（RXE）处于置位状态，其他所有位都清零。



要使能 UART 模块，必须将 UARTEN 置位。假如软件准备对 UART 模块的配置进行更改，则必须先将 UARTEN 清零，之后才能写入更改的配置内容。假如在接收或发送期间禁用 UART，则当前数据会话结束后，UART模块才会停止运行。

注意：启用 UART 后不得修改 UARTCTL 寄存器，否则会产生无法预料的后果。推荐按照下面的步骤来更改 UARTCTL 寄存器的内容。

1. 禁用UART模块；
2. 等待当前数据会话（传输的字符）结束；
3. 通过将线控寄存器 UARTLCRH 的第 4 位（FEN）清 0，从而清空发送 FIFO。
4. 修改控制寄存器的内容；
5. 重新使能UART模块。

3、UARTLCRH(UART 线控寄存器)：UARTLCRH 寄存器称为线控寄存器。数据长度、奇偶校验位、停止位等串行通讯参数都是通过本寄存器设置的。

2.9.5实验现象

建立并设置好工程，编辑好代码后进行编译，将所有错误警告排除后进行烧写与仿真，载入完毕后让 MCU 全速运行。此时使用串口通信软件向 Launchpad 发送字符串“Hello LaunchPad”，可以看到串口通信软件的显示框也接收到显示“Hello LaunchPad”字样，如图 7.11，表明串口通信是成功的。

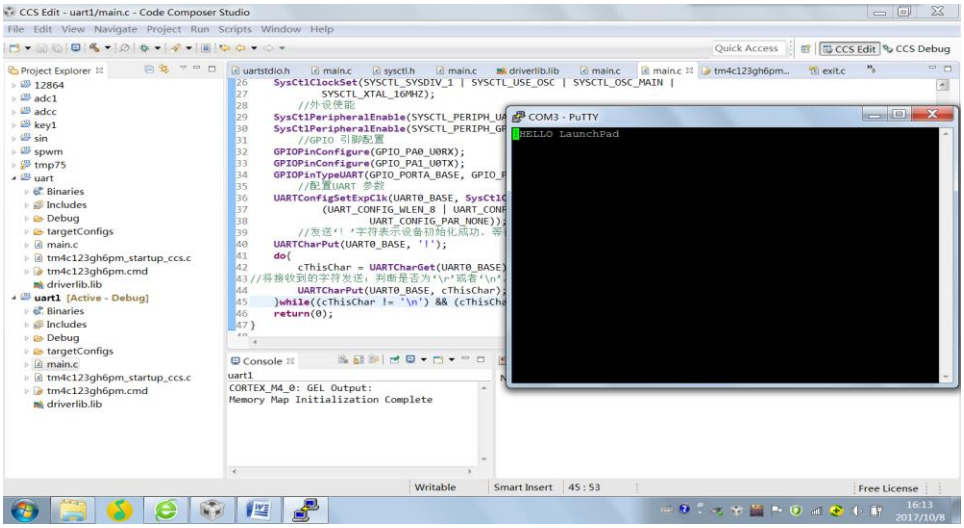


图 7.11 UART 串口通信结果

3 同步串行通信 SSI

同步串行接口SSI（Synchronous Serial Interface）是由TI公司定义的一种高速、全双工、同步串行外围接口协议，它兼容Motorola公司定义的SPI（Serial Peripheral Interface，串行外设总线）接口协议。SSI总线常用在短距离、高速串行传输中，例如高速ADC、SD卡、WIFI数据传输等。SSI的速度可以高达50Mbps，而I2C的速度一般为1 Mbps和400 kbps 两种（半双工），UART一般为115200bps。

3.1 SSI 基本工作原理

每一个 SSI 模块都能以主机或从机方式与外围设备进行数据交换，需要设置主从（Master/Slave）工作模式，其时钟由主机控制。在数据传输时，主机通过拉低一个设备的 SS 引脚来选通该设备进行数据传输。

3.2 基本结构

典型的结构如图 7.12 所示，为 1 个主 SSI 和 3 个从 SSI 构成。有 4 根传输线构成：MOSI（Master Output/Slave Input）为主机输出/从机输入，亦作 SSITX；MISO（Master Input/Slave Output）为主机输入/从机输出，亦作 SSIRX；SCLK（Serial Clock）为时钟信号，由主机产生；SS（Slave Select）为从机使能信号，相当于片选信号。

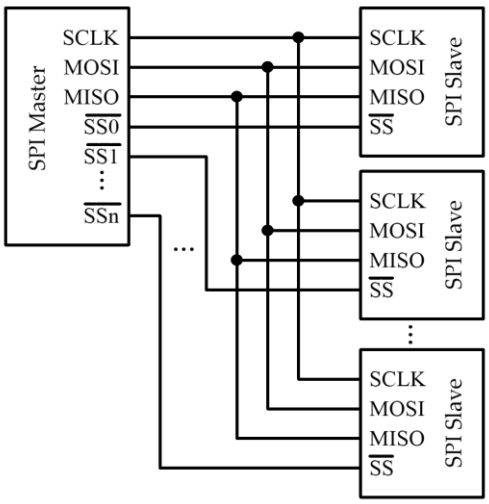


图 7.12 SSI(SPI)典型结构：一个主 SSI 与三个从 SSI

SSI 标配为 4 线,但也可以根据需要接成 3 线或 2 线连接方式,3 线为主机只输出(SCLK、MOSI、SS)、主机只输入（SCLK、MISO、SS）方式，例如 DY 实验板上的 TF Card 连接（所接为 SCLK、SSITX、SSIRX 三线),地线接地,片选始终有效。2 线也可以接成主机只输出(SCLK、MOSI)、主机只输入（SCLK、MISO）方式，例如 DY 实验板上 12864 液晶显示模块的连线方式-主机只输出。同学们今后在设计嵌入系统的时候可以借鉴。

通常情况下，SSI 数据传输速率介于 1~25MHz 之间，字节长度可以为 4 ~16 位，或这些数值的整数倍数位，因此可以在 SSI 总线上进行较高速的数据传输。

3.3 TM4C123GH6PM 的 SSI 特性

TM4C123GH6PM 总共有四个同步串行接口(SSI) ，可工作在主/从模式，支持包括 Freescale SPI，MICROWIRE 以及Texas Instruments（TI） 的SSI 同步串行接口。

TM4C123GH6PM 的SSI 接口有如下特点：

- 可编程接口，支持包括Freescale SPI，MICROWIRE 以及Texas Instruments 的同步串行接口
- 主/从操作模式
- 可编程的时钟速率/预分频器
- 独立的发送和接收FIFO ， 16 位x 8
- 可编程数据帧大小：4 至16 位，
- 内部环回测试，可用于诊断/调试
- 标准FIFO 中断、传输完成中断
- μDMA 直接内存访问控制器

3.4 内部结构框图

每个SSI模块对外部的信号包括4个，即发送总线SSITX、接收总线SSIRX、同步时钟信

号SSIClk以及帧信号SSIFss。每个SSI模块包括3个部分，即寄存器、栈以及收发装置。其结构框图如图7.13所示，其中粗线表示通信数据信号的传输，细线表示控制/状态信号的传输。

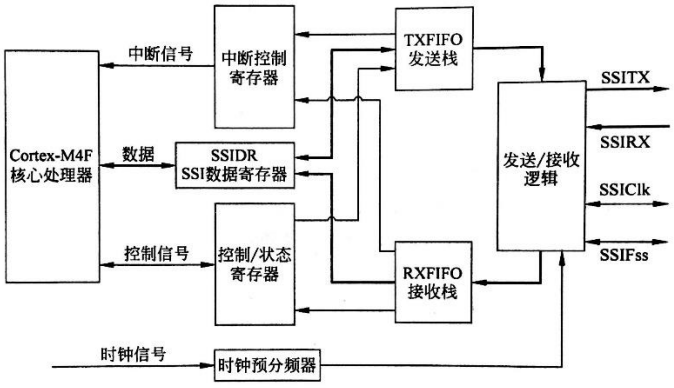


图7.13 SSI结构框图

其左侧的寄存器主要有3种：SSI数据寄存器SSIDR，中断控制寄存器以及控制/状态寄存器。其中数据寄存器SSIDR用于暂存具体的数据信号。中断控制寄存器包括多个寄存器，主要用于SSI通信过程中的中断控制。而控制/状态寄存器则用于对SSI通信的设置以及状态的监控。除了这3种寄存器，还有诸如标识寄存器、DMA控制寄存器、时钟控制寄存器等其他功能的寄存器。

中间部分是栈区，即数据缓冲区FIFO，主要用于存储待处理的并行数据。跟UART模式类似，FIFO共有两个，其中TXFIFO用于存储待发送的数据，RXFIFO用于存储已接收的数据。

右侧的发送/接收逻辑主要起到在并行数据和标准SSI格式数据之间的转换。发送部分将TXFIFO中的并行数据按照配置的SSI的标准格式发送出去。接收部分的作用则是将接收到的SSI格式的串行数据转换为并行数据存入RXFIFO中。收发的同时还有时钟信号SSIClk和帧信号SSIFss的配合。

左下角的时钟预分频用于产生不同频率的SSIClk时钟信号。

TM4C123GH6PM 的 SSI 模块具体寄存器结构如图 7.14 所示。

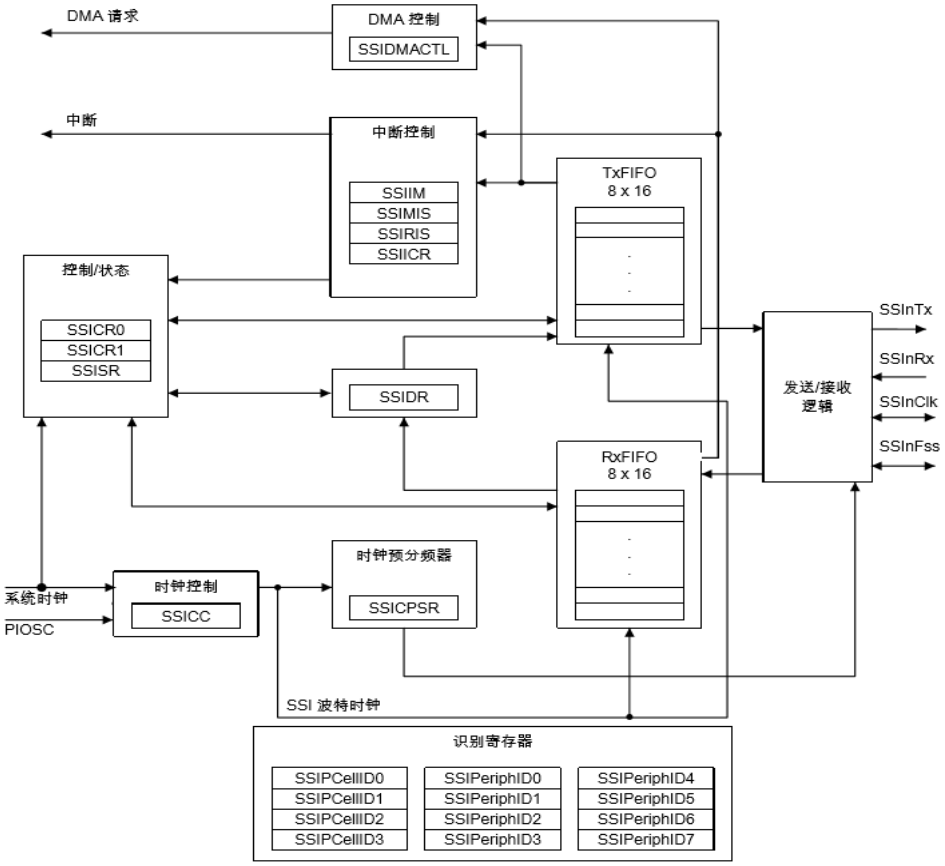


图 7.14 SSI 模块寄存器结构图

寄存器的介绍详见 TM4C123GH6PM 中文数据手册 P830~859。GPIOAFSEL 寄存器的 AFSEL 位确定了 SSI 的功能。常用寄存器为：SSIDR、SSICR0、SSICR1、SSISR、SSICPSR 等。

SSIDR: 发送数据或接收数据

SSICR0:数据帧格式(包括极性和相位)、数据长度、SSI 串行时钟速率（SCR）

SSICR1:主/从、环回模式，SSI 使能，传输结束 EOT

SSISR:忙标志、接收 FIFO 满、接收 FIFO 不空、发送 FIFO 未满、发送 FIFO 空

SSICC、SSICPSR 为比特率设置寄存器。

3.5 比特率生成器

SSI 由一个可编程时钟分频器以及一个预分频器来生成串行时钟。连接外围设备的比特率最大可达 2 MHz。

串行比特率由系统时钟控制，首先除以 CPSDVSR（2～254） （由 SSICPSR 寄存器设定)；随后除以 1 + SCR（0～255）(由 SSICR0 寄存器设定）

输出时钟频率（SSIClk）可由以下公式计算：

$$SSIClk = SysClk / (CPSDVSR * (1 + SCR))$$

注意：SSIClk 可由系统时钟或 PIOSC 驱动。

当 SSICC 时钟配置寄存器的 CS 位被配置为 0x5 时，使用 PIOSC 为时钟源。为 0 时选

用系统时钟。（只有两种状态，其余保留）

在主模式下，系统时钟/PIOSC 必须是 SSIClk 速度的两倍，并且此时 SSIClk 不能高于 25 MHz。

在从模式下，系统时钟/PIOSC 必须是 SSIClk 速度的 12 倍，并且此时 SSIClk 不能高于 6.67 MHz。

3.6 Freescale 极性、相位和帧格式

一个 SSI 的主机与一个从机通信时，除了要保证两者之间的时钟 SCLK 要一致外，还要确保 SSI 数据传输的极性和相位一致。

Freescale 公司为 SSI 定义了时钟极性(CPOL)(定义 SSICLK 信号空闲时的状态,CPOL=0，空闲为低电平，CPOL=1，空闲为高电平）和时钟相位（CPHA）（定义上升沿还是下降沿读取数据），使其成为了业界标准。SSI 的模式编号、极性、相位及对应关系如表 7.3 所列。

表 7.3 SSI 时钟极性列表

模式编号	CPOL（极性）	CPHA（相位）	对应的 Tiva 库函数定义
0	0（空闲为低）	0（上升沿读数）	SSI_FRF_MOTO_MODE_0
1	0（空闲为低）	1（下降沿读数）	SSI_FRF_MOTO_MODE_1
2	1（空闲为高）	0（下降沿读数）	SSI_FRF_MOTO_MODE_2
3	1（空闲为高）	1（上升沿读数）	SSI_FRF_MOTO_MODE_3

这里的 CPOL 极性和 CPHA 相位与 Tiva 库函数的 ui32Protocol 参数相对应；在配置 SSI 时钟时应根据从机的硬件模式选择相应的 SSI 模式，以保证两者可以正常实现 SSI 通信。

帧格式：SSI 数据帧的位长在 4~16 之间，并以最高有效位（MSB）格式存储。

需要注意的是，对于 Freescale SPI、Microwire、TI 的帧格式，当 SSI 空闲时，SSI 串行时钟（SSICLK）不会产生时钟信号；只有在发送或接收数据时，SSICLK 才在设置好的频率下工作。

Freescale 的 SSI 有 4 种模式可供选择，如图 7.15 所示。依据不同的时钟相位和极性，Freescale 的 SSI 有如下四种模式：

（1）CPOL=0，时钟在低电平时为空闲状态：当 SSI 空闲时，SSIClk 无效

① 如果 CPHA=0，则数据会在 SSIClk 的上升沿读取，在下降沿变化。

② 如果 CPHA=1，则数据会在 SSIClk 的下降沿读取，在上升沿变化。

（2）CPOL=1，时钟在逻辑高电平时为空闲状态：

① 如果 CPHA=0，则数据会在 SSIClk 的下降沿读取，在上升沿变化。

② 如果 CPHA=1，则数据会在 SSIClk 的上升沿读取，在下降沿变化。

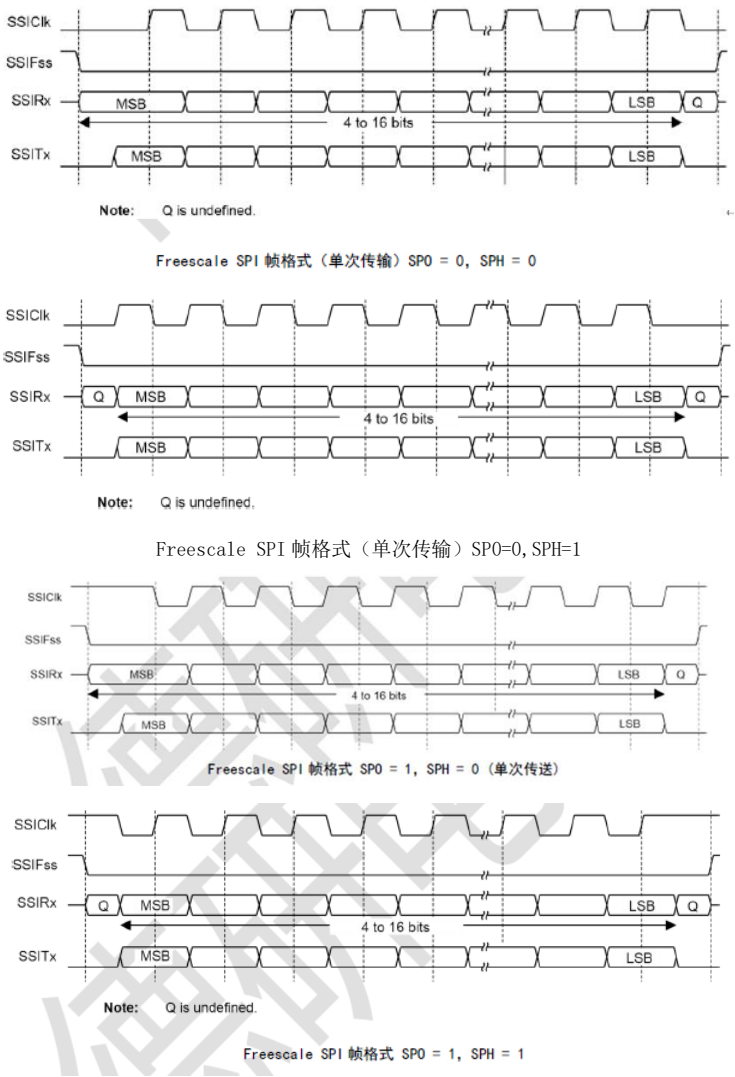


图 7.15 时钟极性和相位时序图

Freescale 公司的 SSI 主要特点：通过修改 SSICR0 控制寄存器的 SPO 和 SPH 位，可对 SSICLK 的状态和相位进行编程控制。SPO 对应 CPOL 控制，SPH 对应 CPHA 控制。

当 SPO 被清除时，SSIClk 引脚将被拉低；当 SPO 被置位，却没有传输数据时，SSIClk 引脚将被拉高。SPH 位如果选择时钟边沿捕获数据，并允许改变状态，那么它的状态决定了传输的第一位数据是否产生一次时钟跳变。当 SPH 位被清除时，数据在第一个时钟边沿即可被收到；如果该位被置位，则数据在第二个时钟边沿被收到。

3.7 FIFO 操作

3.7.1 发送 FIFO 缓冲区

发送 FIFO 缓冲区为 16 位宽，个数为 8 个。

发送数据时，CPU 将数据写入 SSIDR 寄存器。

SSI 配置为主/从模式时，写入 SSITx 引脚的数据将存储到 FIFO 缓冲区中。

在从模式下， SSI 的传输节拍受与其连接的主模式设备控制。

若主设备发送 FIFO 为空，则主设备请求从设备将其 FIFO 中的前 8 个数据发送至主设备；若从设备 FIFO 缓冲区的数据少于 8 个，则未填入数据的 FIFO 缓冲区将发送 0。因此，应确保有 FIFO 里有有效数据。对于 DMA 操作，当 FIFO 缓冲区为空时，可产生 μDMA 请求中断。

3.7.2接收 FIFO 缓冲区

接收 FIFO 缓冲区为 16 位宽，个数为 8 个。

接收数据时，CPU 从 SSIDR 寄存器中读取数据。

3.8 中断处理

SSI 在下列条件发送时可产生中断：

- 发送 FIFO 工作时（当发送 FIFO 中的数据少于或等于 4 个（EOT =0），数据已发完（ EOT =1））
- 接收 FIFO 工作中（当接收 FIFO 达到一半时）
- 接收 FIFO 超时（接收 FIFO 从空状态变为非空状态后 32 个 SSIClk 周期内没有再次变为空状态就会触发，（需手工清除标志））
- 接收 FIFO 溢出
- 传输结束

所有的中断在进行位或操作后被送到 SSI 中断控制器，随后中断控制器根据中断掩码输出一个中断请求。

设置 SSI 中断屏蔽寄存器（ SSIIM ）的相应位可使能相应的可屏蔽中断。

SSI 的发送和接收中断与 SSI 状态中断是相互独立的，因此，可在特定的 FIFO 水位线触发中断。可从 SSI 原始中断状态寄存器（SSIRIS）和 SSI 屏蔽中断状态寄存器（SSIMIS）中读取各个中断源的状态。

传输结束中断（EOT）表示数据已传输完成（仅在主模式下有效）。该中断可用判断何时关闭 SSI 模块时钟或进入睡眠模式。此外，该中断也表示数据已完成发送或数据已完成接收，通常该中断比 FIFO 超时中断来得快。

注意：在 FreescaleSPI 模式下，当 FIFO 已满时，可在每个字节传输时产生一个 EOT 中断。

3.9 初始化及配置

3.9.1启用、初始化 SSI 的步骤如下：

1. 通过 RCGCSSI 寄存器使能 SSI 模块（通过调用 SysCtlPeripheralEnable（）函数实现）

2. 通过 RCGCGPIO 寄存器启用相应的 GPIO 模块时钟（通过调用 SSIClockSourseSet

（）函数初始化 SSI 时钟）

3. 设置 GPIO AFSEL 位的相应引脚（通过调用 GPIOPinConfigure（）函数实现）

4. 配置 GPIOPCTL 寄存器的 PMCn 字段的 SSI 信号，并将其分配到相应的引脚。（通过调用 GPIOPinTypeSSI（）函数实现）

3.9.2配置 SSI 帧格式的步骤如下：

1. 在任何配置和更改之前，首先要将寄存器 SSICR1 中的 SSE 位置 0，即禁用 SSI 模块。

2. 根据 SSI 是作为主机还是从机选择。

- 主模式：设置 SSICR1 寄存器为 0x0000.0000
- 从模式（输出使能）：0x0000.0004
- 从模式（输出禁用）：0x0000.000C

3. 设置 SSICR0 、SSICPSR 寄存器，对 SSI 时钟进行配置（通过调用 SSIClockSourceSet（）函数配置 SSI 时钟源、时钟分频）

①串行时钟率（SCR）：即前面提到的进一步配置位速率需要的 SCR；

②如果使用的是 Freescale SPI 模式，那么通过 SPH 和 SPO 这两位配置所需时钟信号的相位/极性；

③协议类型：选择 Freescale SPI、TI SSF、MICROWIRE 这三种帧格式中的一种。通过 FRF 位配置；

④数据帧的长度：通过 DSS 位配置。

4. 对 SSICR1 寄存器的 SSE 位置 1，使能 SSI 模块（通过调用 SSIEnable（）函数使能 SSI）

例如：要求配置一个 SSI 传输，有如下需求：（通讯之前双方需约定好）

- 主模式
- Freescale SPI 模式(SPO=1, SPH=1)
- 1 Mbps bit rate
- 8 data bits

假设系统时钟为 20 MHz ，比特率应以如下方式计算：

SSIClk = SysClk / (CPSDVSR * (1 + SCR))

$1x10^6 = 20x10^6 / (CPSDVSR * (1 + SCR))$

CPSDVSR 范围为 2～254 的偶数，SCR 为 0～255。若 CPSDVSR 选择 2，SCR 为 9。

该例程的寄存器配置如下：

1. SSICR1 寄存器的 SSE 位被清除

2. 向 SSICR1 寄存器写入 0x0000.0000

3. 向 SSICPSR 寄存器写入 0x0000.0002

4. 向 SSICR0 寄存器写入 0x0000.09C7

5. 将 SICR1 寄存器的 SSE bit 置位，启用 SSI

3.10 SSI 基础实验

本小节将展示 SSI 外设的数据传输功能与方法。

Tiva 开发板提供了相关的 API 函数，方便用户可以使用休眠模块的各种功能。

3.10.1 使用的库函数

1. SSIConfigSetExpClk( )

功能：SSI 配置（需要提供明确的时钟速度）

原型： void SSIConfigSetExpClk(unsigned long ulBase ,unsigned long ulSSIClk ,unsigned long ulProtocol ,unsigned long ulMode ,unsigned long ulBitRate ,unsigned long ulDataWidth)

参数： ulBase：SSI 模块的基址；

ulSSIClk：提供给 SSI 模块的时钟速度

ulProtocol：数据传输的协议

ulMode：SSI 模块的工作模式

ulBitRate：SSI 的位速率，这个位速率必须满足下面的时钟比率标准：其中 FSSI 是提供给 SSI 模块的时钟速率

ulDataWidth：数据宽度，取值 4～16

2. SSIEnable( )

功能：使能 SSI 发送和接收

原型：void SSIEnable(unsigned long ulBase)

参数：ulBase：SSI 模块的基址，取值 SSI_BASE、SSI0_BASE 或 SSI1_BASE

3. SSIDisable( )

功能：禁止 SSI 发送和接收

原型：void SSIDisable(unsigned long ulBase)

参数：ulBase：SSI 模块的基址，取值 SSI_BASE、SSI0_BASE 或 SSI1_BASE

4. SSIDataPutNonBlocking( )

功能：将一个数据单元放入 SSI 的发送 FIFO 里（不等待）

原型：long SSIDataPutNonBlocking(unsigned long ulBase , unsigned long ulData)

参数：ulBase：SSI 模块的基址，取值 SSI_BASE、SSI0_BASE 或 SSI1_BASE

ulData：要发送的数据单元（4～16 个有效位）

返回：返回写入发送 FIFO 的数据单元数量（如果发送 FIFO 里没有可用的空间则返回 0）

5. SSIDataGetNonBlocking( )

功能：从 SSI 的接收 FIFO 里读取一个数据单元（不等待）

原型：long SSIDataGetNonBlocking(unsigned long ulBase , unsigned long *pulData)

参数：ulBase：SSI 模块的基址，取值 SSI_BASE、SSI0_BASE 或 SSI1_BASE

pulData：指针，指向保存读取到的数据单元地址

返回：返回从接收 FIFO 里读取到的数据单元数量（如果接收 FIFO 为空，则返回 0）

6. SSIDataNonBlockingPut( )

功能：将一个数据单元放入 SSI 的发送 FIFO 里（不等待）

原型：#define SSIDataNonBlockingPut(a , b) SSIDataPutNonBlocking(a , b)

参数：参见函数 SSIDataPutNonBlocking( )的描述

7. SSIDataNonBlockingGet( )

功能：从 SSI 的接收 FIFO 里读取一个数据单元（不等待）

原型：#define SSIDataNonBlockingGet(a , b) SSIDataGetNonBlocking(a , b)

参数：参见函数 SSIDataGetNonBlocking( )的描述

8. SSIDataPut( )

功能：将一个数据单元放入 SSI 的发送 FIFO 里

原型：void SSIDataPut(unsigned long ulBase , unsigned long ulData)

参数：ulBase：SSI 模块的基址，取值 SSI_BASE、SSI0_BASE 或 SSI1_BASE

ucData：要发送数据单元（4～16 个有效位）

3.10.2 实验代码设计

```
char *pcChars = "SSI Master send data.";
int32_t i32Idx;
// Configure the SSI.
SSIConfigSetExpClk(SSI_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE0,
    SSI_MODE_MASTER, 2000000, 8);

// Enable the SSI module.
SSIEnable(SSI_BASE);

// Send some data.
i32Idx = 0;
while(pcChars[i32Idx])
{
```

```
SSIDataPut(SSI_BASE, pcChars[i32Idx]);
    i32Idx++;
}
```

4 I2C 接口

内部集成电路（I2C）总线通过一个两线设计（串行数据线SDA 和串行时钟线SCL）来提供双向数据传输，线路连接简单，PCB布线方便，因此得到了较广泛的应用。但其传输速率不高，较常用于低速传感器数据传输，例如：温湿度传感器的应用。I2C 总线也可用于产品开发和制造的系统测试和诊断等应用场合。

4.1 I2C 基本原理

4.1.1 I2C 连接图

一个典型的 I2C 电路连接图如图 7.16 所示。系统只包含两根连接线，SDA（串行数据线）和 SCL（串行时钟线），数据可以双向传输，是一个典型的半双工通信方式。其中的每一个器件都可以配置成主机和从机，由唯一的地址进行识别。主机产生 SCL 时钟信号。通信原理是通过对 SCL 和 SDA 线高低电平时序的控制，来产生 I2C 总线协议所需要的信号进行数据的传递。在通常的应用中，我们把 CPU 带 I2C 总线接口的模块作为主设备，把挂接在总线上的其他设备都作为从设备。

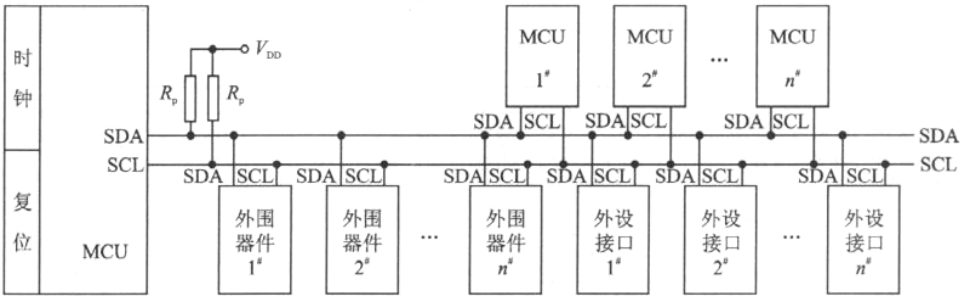


图 7.16 I2C 连接示意图

对于正确的操作，SDA 引脚必须被配置为开漏信号。由于支持高速运行的内部电路，虽然内部电路使其表现得好像它是一个漏极开路信号，但是 SCL 引脚不能被配置为漏极开路信号，这是因为 SCL 信号内部已经连接了上拉电阻。SDA 信号必须外接上拉电阻，连接到正向电源电压。请参考 I2C 总线规范和用户手册来确定正常运行所需的上拉的大小。

由于引脚开漏的连接方式，数据线和时钟线空闲状态都是高电平。当第一个字节的地址匹配以后，从机会发出应答信号（低电平），任一时刻只能有一个从机应答。只有当没有数据需要传输，SDA 和 SCL 才会被释放成高电平，即空闲状态。这就是所谓的“线与”特性。

4.1.2 I2C 通信协议

4.1.2.1 数据有效格式

I2C 的两根总线，即 SCL 时钟信号线和 SDA 数据信号线，两者之间需要相互配合，才能生成有效信号，具体规则如图 7.17 所示。

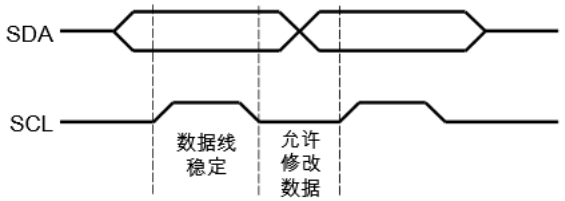


图 7.17 I2C 总线位传输期间数据的有效性

1、SDA 上的有效数据信号在 SCL 为高电平时产生，并且需要覆盖整个 SCL 为高电平的区间，期间不能复生变化，否则无效；

2、SDA 总线上的信号变化只能发生在 SCL 为低电平的区间内；

3、在 SCL 为高电平时 SDA 的信号变化是有特殊含义的，即表示起始和终止信号。

4.1.2.2 起始位和停止位

I2C 协议规定，总线上数据的传输必须以一个起始信号作为开始条件，以一个结束信号作为传输的停止条件。起始和结束信号总是由主设备产生。总线在空闲状态 时，SCL 和 SDA 都保持着高电平，当 SCL 为高电平而 SDA 由高到低的跳变，表示产生一个起始条件；当 SCL 为高而 SDA 由低到高的跳变，表示产生一个 停止条件。在起始条件产生后，总线处于忙状态，由本次数据传输的主从设备独占，其他 I2C 器件无法访问总线；而在停止条件产生后，本次数据传输的主从设备 将释放总线，总线再次处于空闲状态。如图 7.18 所示为起始条件和停止条件。

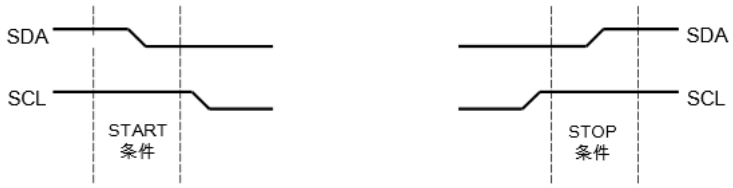


图 7.18 起始和停止条件

4.1.2.3 数据帧

图 7.19 为一帧 I2C 的典型帧结构。

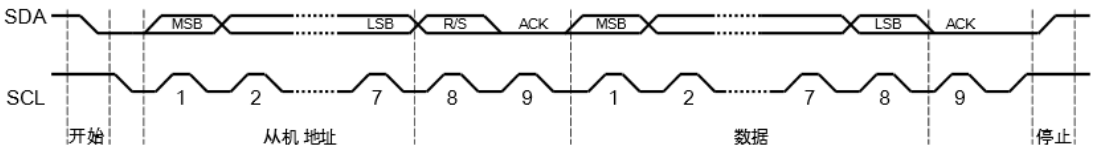


图 7.19 I2C 的数据帧

I2C 总线每次传输的数据长度为九位，其中包括八位数据位和一位应答位（ACK）传送时高位在前，低位在后。

每次传输都由一个起始位开始，从机地址将被发送。地址 7 位，紧跟着的第 8 位是数据传输方向位(I2CMSA 寄存器的 R/S 位)， R/S 位清零表示传输操作（发送），此位置 1 位表示数据请求（接收）。

应答由接收器产生。如果接收器遇到问题无法接收，则会产生一个高电平的 ACK，此时发送器就会产生 STOP 条件结束这一次的数据传输。

数据传输总是由主机生成一个停止条件终止的，然而主机可以在没有产生停止信号的时候，通过再产生一个开始信号和总线上另一个设备的地址，来与另一个设备通信。因此，在一次传输过程中可能会存在各种不同组合的接收/发送格式。

4.1.3 I2C 总线操作

对 I2C 总线的操作实际就是主从设备之间的读写操作。大致可分为以下三种操作情况：

第一，主设备写数据到从设备。数据传输格式如图 7.20 所示：

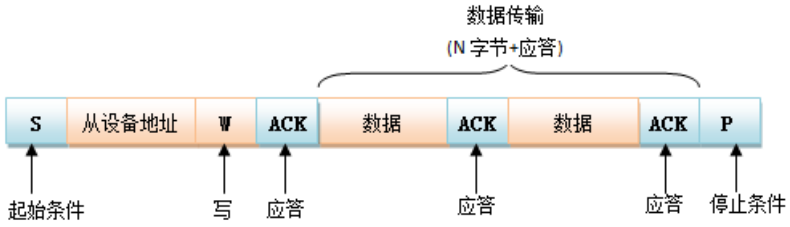


图 7.20 主设备往从设备写数据格式

第二，主设备从从设备中读取数据。数据传输格式如图 7.21 所示：

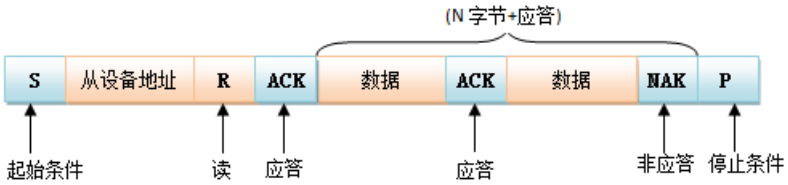


图 7.21 主设备从从设备读数据格式

第三，主设备往从设备中写数据，然后重启起始条件，紧接着从从设备中读取数据；或者是主设备从从设备中读数据，然后重启起始条件，紧接着主设备往从设备中写数据。数据传输格式如图 7.22 所示：

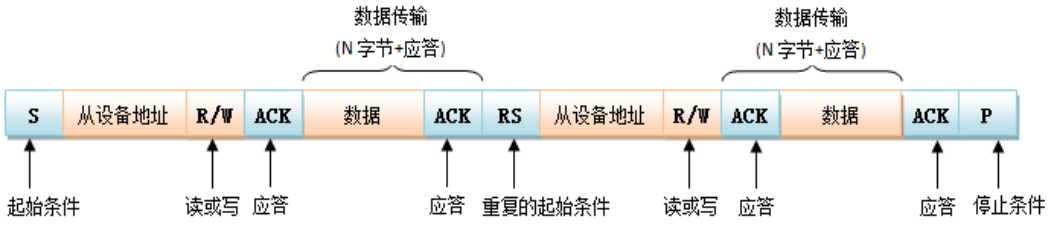


图 7.22 重启起始条件数据格式

第三种操作在单个主设备系统中，重复的开启起始条件机制要比用 STOP 终止传输后又

再次开启总线更有效率。我们的 TIVA 板就只有一个 CPU，适用这种通信模式。

4.2 TM4C123GH6PM 微控制器 I2C 的特点及结构图

TM4C123GH6PM 控制器的I2C 模块具有以下特点：

- I2C 总线上的设备可被配置为主机或从机
  - 支持一个主机或从机发送和接收数据
  - 同时支持主机和从机操作
- 四个I2C 模式：
  - 主机发送模式
  - 主机接收模式
  - 从机发送模式
  - 从机接收模式
- 四个发送速率：
  - 标准模式（100 Kbps）
  - 快速模式（400 Kbps）
  - 超快速模式（1Mbps）
  - 高速模式（3.33Mbps）
- 时钟低超时中断的
- 双从地址能力
- 抗干扰
- 主机和从机中断的产生
  - 当主机发送或接收操作完成时（或因错误终止时），产生中断
  - 当从机发送数据或主机需要数据或检测到起始或停止条件时，产生中断
- 主机由仲裁和时钟同步，支持多主机，以及7 位寻址模式

I2C结构图如图7.23所示。

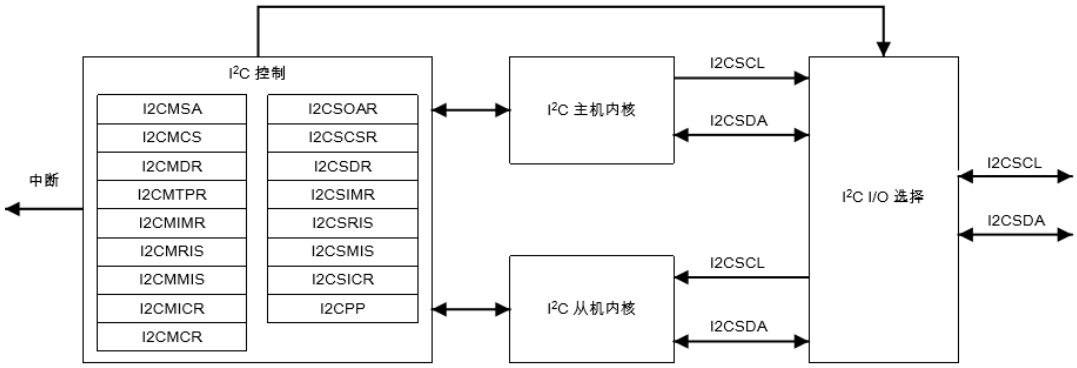


图 7.23 I2C 结构图

寄存器的介绍详见 TM4C123GH6PM 中文数据手册 P879~908。

4.3 引脚复用

表 7.4 列出了部分 I2C 接口的外部信号，并描述每组 I2C 接口信号的功能。I2C 接口信号是一些 GPIO 信号的复用功能，并在复位时默认为 GPIO 信号；I2C0SCL 和 I2C0SDA 引脚例外，该两个引脚默认为 I2C 功能。下表的“引脚复用/分配”列出 I2C 信号 GPIO 引脚的配置。GPIO 复用功能选择（GPIOAFSEL）寄存器的 AFSEL 位应该被设置选择 I2C 功能。括号中的数字是该编码必须编程到 GPIO 端口控制（GPIOPCTL）寄存器 PMCn 位组中，以便给 I2C 信号分配特定的 GPIO 端口引脚。注意，I2CSDA 引脚应使用 GPIO 开漏选择（GPIOODR）寄存器设置为开漏，使用时需要外接上拉电阻。而 I2CSCL 信号不能设置为开漏输出，这是由于该款芯片已经内接了上拉电阻。关于 GPIO 配置的更多信息，请参阅“通用输入/输出端口（GPIO）”章节。

表 7.4 I2C 信号（64LQFP）

管脚名称	管脚编号	管脚复用/管脚赋值	管脚类型	缓冲区类型[a]	描述
I2C0SCL	47	PB2 (3)	I/O	OD	I²C 模块 0 时钟。请注意该信号具有有源上拉。不应将相应的端口管脚配置为开漏。
I2C0SDA	48	PB3 (3)	I/O	OD	I²C 模块 0 数据。
I2C1SCL	23 33	PA6 (3) PG4 (3)	I/O	OD	I²C 模块 1 时钟。请注意该信号具有有源上拉。不应将相应的端口管脚配置为开漏。
I2C1SDA	24 32	PA7 (3) PG5 (3)	I/O	OD	I²C 模块 1 数据。
I2C2SCL	59	PE4 (3)	I/O	OD	I²C 模块 2 时钟。请注意该信号具有有源上拉。不应将相应的端口管脚配置为开漏。
I2C2SDA	60	PE5 (3)	I/O	OD	I²C 模块 2 数据。
I2C3SCL	37 61	PG0 (3) PD0 (3)	I/O	OD	I²C 模块 3 时钟。请注意该信号具有有源上拉。不应将相应的端口管脚配置为开漏。
I2C3SDA	36 62	PG1 (3) PD1 (3)	I/O	OD	I²C 模块 3 数据。
I2C4SCL	35	PG2 (3)	I/O	OD	I²C 模块 4 时钟。请注意该信号具有有源上拉。不应将相应的端口管脚配置为开漏。
I2C4SDA	34	PG3 (3)	I/O	OD	I²C 模块 4 数据。
I2C5SCL	1	PB6 (3)	I/O	OD	I²C 模块 5 时钟。请注意该信号具有有源上拉。不应将相应的端口管脚配置为开漏。

4.4 传输速率控制

I2C 总线可以运行在标准模式（100 kbps），快速模式（400 kbps），超快速模式（1 Mbps）和高速模式（3.33 Mbps）。所选择的模式应符合总线上其他 I2C 器件的速度。

标准，快速和超快速模式的选择，是通过使用在 I2C 主机定时器周期（I2CMTPR）寄存器的值，I2CMTPR 寄存器中的 7 位 TPR 值[6:0]决定了 I2C 的运行速率，可以为 100 kbps 的标准模式，400 kbps 的快速模式，和 1 Mbps 的超快速模式。

I2C 时钟速率由的参数 CLK_PRD、TIMER_PRD、SCL_LP 和 SCL_HP 决定：

CLK_PRD 是系统时钟周期；

SCL_LP 是 SCL 的低相位；

SCL_HP 是 SCL 的高相位；

TIMER_PRD 是 I2CMTPR 寄存器的编程值。通过取代下列方程的已知变量来求解 TIMER_PRD 值。

传输时钟周期 SCL_PERIOD = 2 × (1 + TIMER_PRD) × (SCL_LP + SCL_HP) × CLK_PRD

例如：

CLK_PRD = 50 ns

TIMER_PRD = 2

SCL_LP=6

SCL_HP=4

产生的 SCL 传输速率频率=1/SCL_PERIOD = 333 KHz

表 7.6 给出定时器周期的例子，这应用于基于不同的时钟周期频率来产生标准模式、快速模式、和超快速模式的 SCL 频率。

表 7.6 I2C 主机定时器周期与速度模式的例子

系统时钟	定时器周期	标准模式	定时器周期	快速模式	定时器周期	超快模式
4 MHz	0x01	100 Kbps	-	-	-	-
6 MHz	0x02	100 Kbps	-	-	-	-
12.5 MHz	0x06	89 Kbps	0x01	312 Kbps	-	-
16.7 MHz	0x08	93 Kbps	0x02	278 Kbps	-	-
20 MHz	0x09	100 Kbps	0x02	333 Kbps	-	-
25 MHz	0x0C	96.2 Kbps	0x03	312 Kbps	-	-
33 MHz	0x10	97.1 Kbps	0x04	330 Kbps	-	-
40 MHz	0x13	100 Kbps	0x04	400 Kbps	0x01	1000 Kbps
50 MHz	0x18	100 Kbps	0x06	357 Kbps	0x02	833 Kbps
80 MHz	0x27	100 Kbps	0x09	400 Kbps	0x03	1000 Kbps

4.5 中断

I2C 遵循到以下条件时，可以产生中断：

- 完成主机事务
- 主机仲裁丢失
- 主机传输错误
- 收到从机事务
- 请求从机传输
- 检测到总线停止条件
- 检测到总线开始条件

I2C 主机和从机模式都有独立的中断信号。尽管这两个模块可以产生多种条件的中断，但是由于所有中断事件在发送到中断控制器前会进行一个“或”操作，所以任意时刻只有一个单一的中断信号被发送到中断控制器。

4.6 其他控制

4.6.1环路模式 Loopback

置位 I2C 主机配置（I2CMCR）寄存器中的 LPBK 位可以将 I2C 模块设置为环路模式，Loopback 模式用于诊断和调试 I2C 的运行模式。在环路模式下，来自主机的 SDA 和 SCL 信号和从机模块的 SDA 和 SCL 信号在内部被连接起来。

4.6.2时钟信号低电平超时

当 I2C 的从机较忙时，它可以通过将时钟信号 SCL 拉低从而暂停数据的传输，等到空闲后拉高 SCL，以继续数据传输。这样做可以降低传输速率，从而照顾一些处理能力较低的从机。但是，暂停时间不能无限长，I2C 模块中有一个计时器，用来统计低电平的时长（实际是从 START 开始的总时长），如果超时则会产生一个中断提醒主机。这时主机会认为该传输超时无效，因此会在从机释放 SCL 信号线的下一时刻就立即产生一个终止信号 STOP 终止当前的数据传输。

当然，如果从机一直不释放 SCL 信号线，那么就只能在更高层的协议上重启主机和从机，以终止这样的状态。

4.6.3双重地址

对于从机来说，I2C 接口支持两个地址。其中一个是正常地址，在寄存器 I2CSOAR 的 OAR 位定义。另一个地址在寄存器 I2CSOAR2 的 OAR2 位定义。当双重地址功能被使能时（通过将 I2CSOAR2 的 OAR2 位置位），这两个地址都代表该从机，即当数据帧中的地址位与其中任何一个地址匹配时，该从机都会产生一个 ACK 信号。

当然，为了避免冲突，I2CSOAR 的 OAR 位定义的地址有着较高的优先级，不会被禁用；而 I2CSOAR2 的 OAR2 位定义的地址是可以被禁用的。另外，寄存器 I2CSCSR 中的 OAR2SEL 位可以指定匹配的地址是 OAR 还是 OAR2。

4.7 初始化与配置

以主机发送单字节（假定系统时钟为 20 MHz）为例说明使能以及初始化 I2C 的步骤。

1. 在系统控制模块使用 RCGCI2C 寄存器使能 I2C 模块。（通过调用 SysCtlPeripheralEnable（）函数实现）

2. 通过在系统控制模块的 RCGCGPIO 寄存器为相应的 GPIO 模块使能时钟。要了解使能哪些 GPIO 端口。（通过调用 I2CMasterInitExpClk（）函数实现）

3. 在 GPIO 模块，通过 GPIOAFSEL 寄存器位它们的复用功能使能相应的引脚，以确定配置哪个 GPIO。（通过调用 GPIOPinConfigure（）函数实现）

4. 使能 I2CSDA 引脚来配置开漏操作。

5. 在 GPIOPCTL 寄存器配置 PMCn 位组为相应的引脚配置 I2C 信号。（通过调用

GPIOPinTypeI2C（）函数实现）

6. 向 I2CMCR 寄存器写入 0x00000010 值来初始化 I2C 主机。（通过调用 I2CSlaveInit（）函数实现）

7. 通过写入 I2CMTPR 寄存器正确的值来设置所需的 100 Kbps 的 SCL 时钟速度。写入 I2CMTPR 寄存器的值代表在一个 SCL 时钟周期中系统时钟周期数。TPR 值由以下等式确定：

TPR = {System Clock/【2*(SCL_LP + SCL_HP)*SCL_CLK】} -1

TPR = {20MHz/(2*(6+4)*100000)} -1

TPR = 9

向 I2CMTPR 寄存器写入 0x00000009。（前面调用函数 I2CMasterInitExpClk（）函数已经实现）

8. 规定主机的从机地址，下一个操作是一个发送，该发送通过向 I2CMSA 寄存器值写入 0x00000076 实现。该操作设置了从机地址为 0x3B。（通过调用 I2CMasterSlaveAddrSet（）函数实现）

9. 通过向 I2CMDR 寄存器写入所需数据将数据（位）传输到数据寄存器。（通过调用 I2CMasterDataPut（）函数实现）

10. 启动从主机到从机的数据单字节的传输是通过向 I2CMCS 寄存器写入 0x00000007 (STOP, START, RUN)实现的。（通过调用 I2CMasterControl（）函数实现）

11. 等待直到传输完成，通过轮询 I2CMCS 寄存器的 BUSBSTY 位，直到该位被清除。（通过调用 I2CMasterBusy（）函数实现）

12. 检测 I2CMCS 寄存器的 ERROR 位以确保传输被应答。

4.8 实验设计

本小节将对 launchpad 的 I2C 的主从发送、接收进行设计。

4.8.1 程序设计

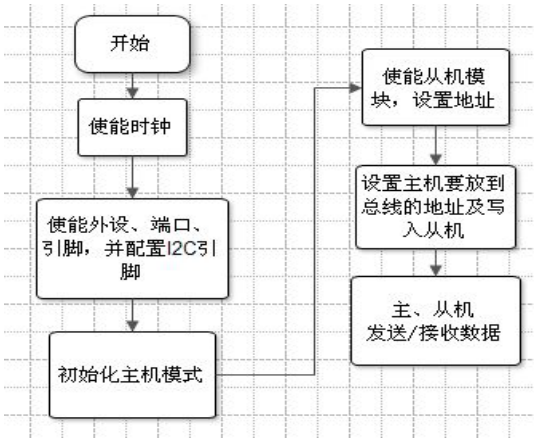


图 7.24 配置流程图

4.8.2 程序代码

```
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdarg.h>
#include <string.h>


#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_timer.h"
#include "inc/hw_ints.h"
#include "inc/hw_gpio.h"
#include "inc/hw_i2c.h"
#include "inc/hw_sysctl.h"
#include "inc/hw_adc.h"
#include "inc/hw_ssi.h"
#include "inc/hw_uart.h"
#include "driverlib/adc.h"
#include "driverlib/rom.h"
#include "driverlib/timer.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "driverlib/systick.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/ssi.h"
#include "driverlib/udma.h"
#include "driverlib/fpu.h"
#include "driverlib/rom.h"
#include "driverlib/uart.h"
#include "driverlib/i2c.h"

#include "utils/uartstdio.h"
#include "I2C.h"
/*****************************************************************
*函数名：main
*描述：无
*输入：无
*****************************************************************/
#define NUM_I2C_DATA 3 // 定义要通过I2C发送的数据包的数目
// 设定slave（从）模块的地址，是一个7-bit的地址加上RS位，具体形式如下：
```

```
[A6:A5:A4:A3:A2:A1:A0:RS]
// RS位是一个指示位，如果RS=0，则说明是主发送数据，从接收数据；RS=1说明是主接收数据，从发送数据
#define SLAVE_ADDRESS 0x3C //0111100
void initConsole(void);

 int main(void)
{
uint32_t pui32DataTx[NUM_I2C_DATA]; //发送数据缓冲区
uint32_t pui32DataRx[NUM_I2C_DATA]; //接收数据缓冲区
uint32_t ui32Index; //要发送数据在缓冲区的位置
//设置时钟源、系统时钟分频数、系统时钟频率
SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);
// 使能I2C0 外设
SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);
// 使能PB 端口
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
// 配置引脚PB2、PB3 复用功能
GPIOPinConfigure(GPIO_PB2_I2C0SCL);
GPIOPinConfigure(GPIO_PB3_I2C0SDA);
// 配置I2C 引脚
GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_2 | GPIO_PIN_3);
// 使能环回模式，用于调试。
HWREG(I2C0_BASE + I2C_O_MCR) |= 0x01;
// 初始化并使能主机模式，使用系统时钟为I2C0 模块提供时钟频率，主机模块传输速//率为100Kbps。
I2CMasterInitExpClk(I2C0_BASE, SysCtlClockGet(), false);
// 使能从机模块
I2CSlaveEnable(I2C0_BASE);
// 设置从机地址
I2CSlaveInit(I2C0_BASE, SLAVE_ADDRESS);
// 设置主机放到总线上的地址，写入从机。
I2CMasterSlaveAddrSet(I2C0_BASE, SLAVE_ADDRESS, false);
initConsole();
UARTprintf("I2C Loopback Example ->");
UARTprintf("\n Module = I2C0");
UARTprintf("\n Mode = Single Send/Receive");
UARTprintf("\n Rate = 100kbps\n\n");
// 初始化要发送的数据
pui32DataTx[0] = 'I';
pui32DataTx[1] = '2';
pui32DataTx[2] = 'C';
// 初始化接收缓冲区
```

```
for(ui32Index = 0; ui32Index < NUM_I2C_DATA; ui32Index++)
{
pui32DataRx[ui32Index] = 0;
}
//发送3 个I2C 数据
for(ui32Index = 0; ui32Index < NUM_I2C_DATA; ui32Index++)
{
// 显示I2C 正在传输的数据
UARTprintf(" Sending: '%c' . . . ", pui32DataTx[ui32Index]);
// 将要发送的数据放到主机数据寄存器中
I2CMasterDataPut(I2C0_BASE, pui32DataTx[ui32Index]);
// 初始化I2C 主机模块状态为单端发送
I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_SINGLE_SEND);
// 等待从机接收数据并确认
while(!(I2CSlaveStatus(I2C0_BASE) & I2C_SLAVE_ACT_RREQ))
{
}
// 从从机数据寄存器读取数据
pui32DataRx[ui32Index] = I2CSlaveDataGet(I2C0_BASE);
// 等待主机传输完毕
while(I2CMasterBusy(I2C0_BASE))
{
}
// 显示从机接收的数据
UARTprintf("Received: '%c'\n", pui32DataRx[ui32Index]);
}
// 重置接收缓冲区
for(ui32Index = 0; ui32Index < NUM_I2C_DATA; ui32Index++)
{
pui32DataRx[ui32Index] = 0;
}
UARTprintf("\n\nTranferring from: Slave -> Master\n");
// 主机从该地址读取数据
I2CMasterSlaveAddrSet(I2C0_BASE, SLAVE_ADDRESS, true);
// 初始化I2C 主机模块状态为单端接收
I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);
// 等待主机请求从机发送数据
while(!(I2CSlaveStatus(I2C0_BASE) & I2C_SLAVE_ACT_TREQ))
{
}
for(ui32Index = 0; ui32Index < NUM_I2C_DATA; ui32Index++)
{
UARTprintf(" Sending: '%c' . . . ", pui32DataTx[ui32Index]);
// 将要发送的数据放到从机数据寄存器中
```

```
I2CSlaveDataPut(I2C0_BASE, pui32DataTx[ui32Index]);
// 初始化I2C 主机模块状态为单端接收
I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);
// 等待从机发送完毕
while(!(I2CSlaveStatus(I2C0_BASE) & I2C_SLAVE_ACT_TREQ))
{
}
// 从主机数据寄存器读取数据
pui32DataRx[ui32Index] = I2CMasterDataGet(I2C0_BASE);
// 显示接收的数据
UARTprintf("Received: '%c'\n", pui32DataRx[ui32Index]);
}
UARTprintf("\nDone.\n\n");
return(0);
}
/*************************************************************************
*函数名: initConsole
*描述: 无
*输入: 无
*************************************************************************/
void initConsole(void)
{
//使能用于UART0 的端口
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
// Enable so that we can configure the clock.
//配置UART0 时钟并使能
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
//使用内部16MHz 振荡器作为UART 时钟源
UARTClockSourceSet (UART0_BASE, UART_CLOCK_PIOSC);
//选择引脚的复用功能为UART
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
//初始化UART
UARTStdioConfig(0, 115200, 16000000);
}
```

4.8.3 使用到的库函数

1. 函数 I2CMasterInitExpClk

功能：初始化 I2C 的主机模块。

原型：void I2CMasterInitExpClk(uint32_t ui32Base,uint32_t ui32I2CClk,
bool bFast);

参数：ui32Base 是 I2C 主机模块的基地址。

ui32I2CClk 是为 I2C 提供的时钟频率。

bFast 建立快速数据传输。

描述：该函数通过为主机配置总线速度并使能 I2C 主机模块，来初始化 I2C 主机模块的操作。如果 bFast 为 true，那么主机模块的传输速率为 400 Kbps；否则，主机模块的传输速率为 100 Kbps。如果需要超快速模式(1 Mbps)，则软件在调用该函数之后，需要手动写入 I2CMTPR 寄存器。对于高速传输模式(3.4 Mbps)，在与从机的初始通信可在 100 Kbps 或 400 Kbps 速率下完成后，一个特定的命令用于切换到快速模式。外设时钟与系统时钟相同。该值由 SysCtlClockGet()函数返回，如果它是常数且已知，那么它可以显示地编程。

返回值：无。

2. 函数 I2CSlaveEnable

功能：使能 I2C 从机模块。

原型：void I2CSlaveEnable(uint32_t ui32Base);

参数：ui32Base 为 I2C 从机模块基地址。

描述：该函数使能 I2C 从机操作。

返回值：无。

3. 函数 I2CSlaveInit

功能：初始化 I2C 从机模块。

原型：void I2CSlaveInit(uint32_t ui32Base,uint8_t ui8SlaveAddr);

参数：ui32Base 为 I2C 从机模块基地址。ui8SlaveAddr 为 7 位从机地址。

描述：该函数通过配置从机地址和使能 I2C 从机模块，来初始化 I2C 从机模块的操作。

返回值：无。

4. 函数 I2CMasterSlaveAddrSet

功能：设置主机放到总线上的地址。

原型：void I2CMasterSlaveAddrSet(uint32_t ui32Base,uint8_t ui8SlaveAddr,

bool bReceive);

参数：ui32Base 为 I2C 从机模块基地址。

ui8SlaveAddr 为 7 位从机地址。

bReceive 标志表示与从机通信的类型。

描述：在开始通信时，该函数配置 I2C 主机放到总线上的地址。当 bReceive 参数被设置为 true 时，表示主机正读取从机；否则，表示主机正写入从机。

返回值：无。

5. 函数 IntMasterEnable

功能：使能处理器中断。

原型：bool IntMasterEnable(void)；

描述：该函数允许处理器响应中断。该函数不影响中断控制器中已使能的中断设置；它仅门控从控制器到处理器的单个中断。

返回值：当函数被调用时，中断失能则返回 true；若中断使能，则返回 false。

6. 函数 I2CMasterDataPut

功能：从 I2C 主机传输一个字节。

原型：void I2CMasterDataPut(uint32_t ui32Base,uint8_t ui8Data)；

参数：ui32Base 为 I2C 主机模块基地址。

ui8Data 为主机传输的数据。

描述：该函数将提供的数据放入 I2C 主机数据寄存器。

返回值：无。

7. 函数 I2CMasterControl

功能：控制 I2C 主机模块状态。

原型：void I2CMasterControl(uint32_t ui32Base,uint32_t ui32Cmd)；

参数：ui32Base 为 I2C 主机模块基地址。

ui32Cmd 为向主机下达的指令。

描述：该函数为控制主机模块状态的发送和接收状态。参数 ui32Cmd 为以下值之一：

I2C_MASTER_CMD_SINGLE_SEND

I2C_MASTER_CMD_SINGLE_RECEIVE

I2C_MASTER_CMD_BURST_SEND_START...

返回值：无。

8. 函数 I2CSlaveStatus

功能：获取 I2C 从机模块的状态。

原型：uint32_t I2CSlaveStatus(uint32_t ui32Base)；

参数：ui32Base 为 I2C 主机模块基地址。

描述：如果有的话，该函数返回来自主机的活动请求。可能的取值为：

I2C_SLAVE_ACT_NONE

I2C_SLAVE_ACT_RREQ

I2C_SLAVE_ACT_TREQ...

返回值：返回 I2C_SLAVE_ACT_NONE 表示没有 I2C 从机模块的请求，I2C_SLAVE_ACT_RREQ 表示主机已经给从机模块发送数据，I2C_SLAVE_ACT_TREQ 表示主机请求从机发送数据，I2C_SLAVE_ACT_RREQ_FBR 表示 I2C 主机已经向从机发送了数据并且紧跟从机地址后的第一个字节已被接收，I2C_SLAVE_ACT_OWN2SEL 表示与第二个 I2C 从机地址匹配，I2C_SLAVE_ACT_QCMD 表示接收了一个快速指令， I2C_SLAVE_ACT_QCMD_DATA 表示当接收

到快速指令时，设置数据位。

9. 函数 I2CSlaveDataGet

功能：接收已经发给从机的一个字节。

原型：uint32_t I2CSlaveDataGet(uint32_t ui32Base);

参数：ui32Base 为 I2C 从机模块基地址。

描述：该函数从 I2C 从机数据寄存器读取数据的一个字节。

返回值：返回接收从机的字节，并强制转换为 uint32_t 类型。

10. 函数 I2CSlaveDataPut

功能：从 I2C 从机传输一字节数据。

原型：void I2CSlaveDataPut (uint32_t ui32Base,uint8_t ui8Data);

参数：ui32Base 为 I2C 从机模块基地址。

ui8Data 为从机传输的数据。

描述：该函数将提供的数据放入 I2C 从机数据寄存器。

返回值：无。

11. 函数 I2CMasterBusy

功能：表示主机是否忙。

原型：bool I2CMasterBusy(uint32_t ui32Base);

参数：ui32Base 为 I2C 从机模块基地址。

描述：该函数返回主机是否忙于传输或接收数据的标识。

返回值：如果主机忙，返回 true；否则返回 false。

4.8.4 实验现象

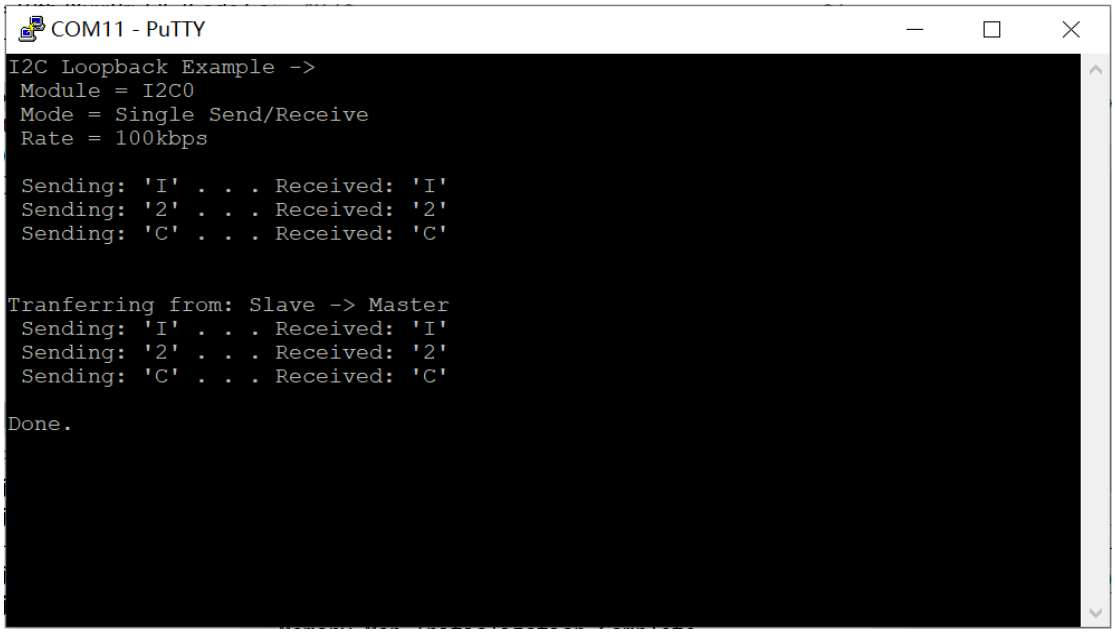


图 7.25　实验现象

【第九周课堂练习 1】

1、在 LaunchPad 上编程实现 UART 模块和 PC 的串行通信，并考虑 MCU 接收使用中断处理：

（1） Tiva MCU 每隔 1s 发送一个字符串“Hello!”，在 PC 上显示出来。

（2） Tiva MCU 接收到 PC 串口发送的一个字符“A”后，MCU 再发送一个字符串“Hello!”，在 PC 上显示出来。

（3） Tiva MCU 接收到 PC 串口发送的一个字符“send”后，MCU 再发送一个字符串“Hello!”，在 PC 上显示出来。

2、编程实现 Tiva 实验板上 SPI 接口 LCD 显示屏的通信操作，并把自己的姓名、学号显示在 LCD 上，再实现慢速滚屏显示（上下、左右），并用示波器观察 SSI 传输数据的时序。

【第九周课堂练习 2】

用 I/O 口模拟 I2C 数据总线时序，读出 TMP75 测量的温度数据。

【第十周课堂练习 2】

结合各自的课题，完成一种通信方式的串行数据传输任务。

【课前预习】

1、 什么是异步通信和同步通信？

2、 什么是串行通信和并行通信？

3、 Tiva 串行同步通信 SSI 的最大传输速率是多少？

4、 SSI/SPI 总线最少需要几根线才能工作？为什么？

5、 简述 Tiva I2C 的基本结构和通信原理。

6、 I2C 有哪几种有效的通信速率模式？

【课后作业】

1、 请画出在 UART 传输线上传输字符“X”的波形图。假设通信帧格式为 1 个起始位、8 个数据位、偶校验、1 个停止位，波特率为 115200bps。

2、 请画出多个 SSI 接口设备应用连接的典型结构图。

3、 请画出多个 I2C 接口设备应用连接的典型结构图。

4、 当使用 I2C 数据通信时，是否可以忽略外围设备的 ACK？

5、 试总结归纳 UART、SSI、I2C 三种通信方式的特点各有哪些。

6、 编程实现 Tiva 实验板上 I2C 总线温度传感器的通信操作，并把读到的温度保留 2 位小数显示出来。直接提交运行结果及图片。