目 录

第1章　通用异步收发器(UART)

1.1 UART 总体特性

1． UART 简介

计算机与外部设备的连接，基本上使用了两类接口：串行接口与并行接口。并行接口是指数据的各个位同时进行传送，其特点是传输速度块，但当传输距离远、位数又多时，通信线路变复杂且成本提高。串行通信是指数据一位位地顺序传送，其特点是适合于远距离通信，通信线路简单，只要一对传输线就可以实现双向通信，从而大大降低了成本。

串行通信又分为异步与同步两类。UART（Universal Asynchronous Receiver/Transmitter，通用异步收发器）正是设备间进行异步通信的关键模块。它的重要作用如下所示：

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

若加入一个合适的电平转换器，如 SP3232E、SP3485，UART 还能用于 RS-232、RS-485 通信，或与计算机的端口连接。UART 应用非常广泛，手机、工业控制、PC 等应用中都要用到 UART。

2． Stellaris 系列 ARM 的 UART 特性

Stellaris（群星）系列 ARM 的 UART 具有完全可编程、16C550 型串行接口的特性（但是并不兼容）。Stellaris 系列 ARM 含有 1 至 3 个 UART 模块。每个 UART 都具有以下特性：独立的发送 FIFO 和接收 FIFO（First-In First-Out，先进先出）

- FIFO 长度可编程，包括提供传统双缓冲接口的 1 字节深的操作
- FIFO 触发深度为：1/8、1/4、1/2、3/4、7/8
- 可编程的波特率发生器，允许速率高达 3.125Mbps（兆位每秒）
- 标准的异步通信：起始位、停止位和奇偶校验位
- 检测错误的起始位
- 线中止（Line-break）的产生和检测
- 完全可编程的串行接口特性：
  - 5、6、7 或 8 个数据位
  - 偶校验、奇校验、粘着或无奇偶校验位的产生/检测
  - 产生 1 或 2 个停止位（使用 2 个停止位可以降低误码率）
- 某些型号集成 IrDA 串行红外（SIR）编码器/解码器，具有以下特性：
  - 用户可以根据需要对 IrDA 串行红外（SIR）或 UART 输入/输出端进行编程
  - IrDA SIR 编码器/解码器功能模块在半双工时其数据速率可高达 115.2Kbps
  - 位持续时间（bit duration）为 3/16（正常）或 1.41～2.23μs（低功耗）

如图 1.1 所示，为 Stellaris 系列 ARM 芯片 UART 与电脑 COM 端口连接的典型应用电路。CZ1 和 CZ2 是电脑 DB9 形式的 COM 接口，U1 是 Exar（原 Sipex）公司的 UART 转 RS-232 的接口芯片 SP3232E，可在 3.3V 下工作。

注意：接在 UART 端口的上拉电阻请不要省略，否则可能会影响到通信的可靠性。

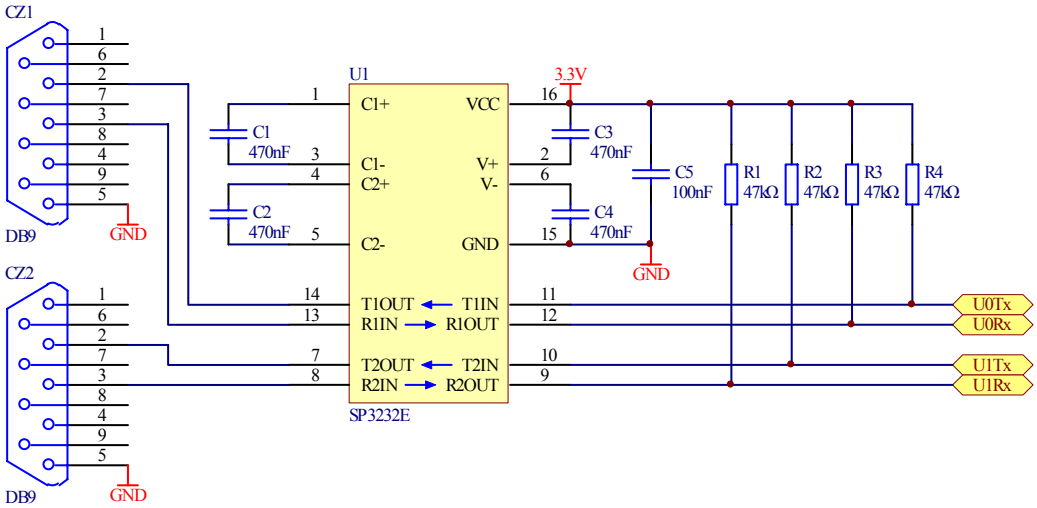

图 1.1　UART 与电脑连接的典型应用电路

1.2　UART 功能概述

1．　发送/接收逻辑

发送逻辑对从发送 FIFO 读取的数据执行“ 并→串 ”转换。控制逻辑输出起始位在先的串行位流，并且根据控制寄存器中已编程的配置，会面紧跟着数据位（注意：最低位 LSB 先输出）、奇偶校验位和停止位。参见图 1.2 的描述。

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

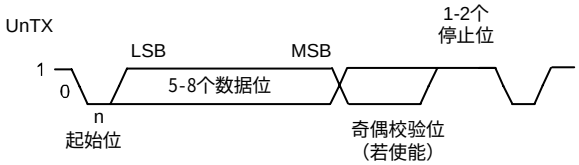

图 1.2　UART 字符帧（LSB 在前）

2．　波特率的产生

波特率除数（baud-rate divisor）是一个 22 位数，它由 16 位整数和 6 位小数组成。波特率发生器使用这两个值组成的数字来决定位周期。通过带有小数波特率的除法器，在足够高的系统时钟速率下，UART 可以产生所有标准的波特率，而误差很小。

波特率除数公式：

$$BRD = BRDI.BRDF = SystemClock / (16 \times BaudRate)$$

其中：

BRD 是 22 位的波特率除数，由 16 位整数和 6 位小数组成

BRDI 是 BRD 的整数部分

BRDF 是 BRD 的小数部分

SystemClock 是系统时钟（UART 模块的时钟直接来自 SystemClock）

BaudRate 是波特率（9600，38400，115200 等）

以 6MHz 晶振作为系统时钟、波特率取 115200 为例，误差仅 0.16%，完全符合要求。

利用《Stellaris 外设驱动库》配置 UART 的方法是采用函数 UARTConfigSet( )。以 UART0 为例，设置波特率为 9600、数据位 8、停止位 1、无校验的方法如下：

```
UARTConfigSet(UART0_BASE,                    //  配置 UART0
              9600,                          //  波特率：9600
              UART_CONFIG_WLEN_8 |           //  数据位：8 位
              UART_CONFIG_STOP_ONE |         //  停止位：1 位
              UART_CONFIG_PAR_NONE);         //  校验位：无
```

3． 数据收发

发送时，数据被写入发送 FIFO。如果 UART 被使能，则会按照预先设置好的参数（波特率、数据位、停止位、校验位等）开始发送数据，一直到发送 FIFO 中没有数据。一旦向发送 FIFO 写数据（如果 FIFO 未空），UART 的忙标志位 BUSY 就有效，并且在发送数据期间一直保持有效。BUSY 位仅在发送 FIFO 为空，且已从移位寄存器发送最后一个字符，包括停止位时才变无效。即 UART 不再使能，它也可以指示忙状态。BUSY 位的相关库函数是 UARTBusy( )，参见表 1.23 的描述。

在 UART 接收器空闲时，如果数据输入变成“ 低电平 ”，即接收到了起始位，则接收计数器开始运行，并且数据在 Baud16 的第 8 个周期被采样。如果 Rx 在 Baud16 的第 8 周期仍然为低电平，则起始位有效，否则会被认为是错误的起始位并将其忽略。

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

最后，如果 Rx 为高电平，则有效的停止位被确认，否则发生帧错误。当接收到一个完整的字符时，将数据存放在接收 FIFO 中。

4． 中断控制

出现以下情况时，可使 UART 产生中断：

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

由于所有中断事件在发送到中断控制器之前会一起进行“ 或运算 ”操作，所以任意时刻 UART 只能向中断产生一个中断请求。通过查询中断状态函数 UARTIntStatus( )，软件可以在同一个中断服务函数里处理多个中断事件（多个并列的 if 语句）。

5． FIFO 操作

FIFO 是“ First-In First-Out ”的缩写，意为“ 先进先出 ”，是一种常见的队列操作。

Stellaris 系列 ARM 的 UART 模块包含有 2 个 16 字节的 FIFO：一个用于发送，另一个

用于接收。可以将两个 FIFO 分别配置为以不同深度触发中断。可供选择的配置包括：1/8、1/4、1/2、3/4 和 7/8 深度。例如，如果接收 FIFO 选择 1/4，则在 UART 接收到 4 个数据时产生接收中断。

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

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

收发 FIFO 主要是为了解决 UART 收发中断过于频繁而导致 CPU 效率不高的问题而引入的。在进行 UART 通信时，中断方式比轮询方式要简便且效率高。但是，如果没有收发 FIFO，则每收发一个数据都要中断处理一次，效率仍然不够高。如果有了收发 FIFO，则可以在连续收发若干个数据（可多至 14 个）后才产生一次中断然后一并处理，这就大大提高了收发效率。

完全不必要担心 FIFO 机制可能带来的数据丢失或得不到及时处理的问题，因为它已经帮你想到了收发过程中存在的任何问题，只要在初始化配置 UART 后，就可以放心收发了，FIFO 和中断例程会自动搞定一切。

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

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

需要注意的是，在使能接收中断的同时一般都还要使能接收超时中断。如果没有接收超时功能，则在接收 FIFO 未填充到预设深度而对方已经发送完毕的情况下并不会触发中断，结果造成最后接收的有效数据得不到处理的问题。另一种情况是对方发送过程中出现间隔，也不会触发中断，已在接收 FIFO 里的数据同样得不到及时处理。如果使能了接收超时中断，则在对方发送过程中出现 3 个数据的传输时间间隔时内就会触发超时中断，从而确保数据能够得到及时的处理。

6． 回环操作

UART 可以进入一个内部回环（Loopback）模式，用于诊断或调试。在回环模式下，从

Tx 上发送的数据将被 Rx 输入端接收。

7． 串行红外协议

在某些 Stellaris 系列 ARM 芯片里，UART 还包含一个 IrDA 串行红外（SIR）编码器/解码器模块。IrDA SIR 模块的功能是在异步 UART 数据流和半双工串行 SIR 接口之间进行转换。片上不会执行任何模拟处理操作。SIR 模块的任务就是要给 UART 提供一个数字编码输出和一个解码输入。UART 信号管脚可以与一个红外收发器连接以实现 IrDA SIR 物理层连接。

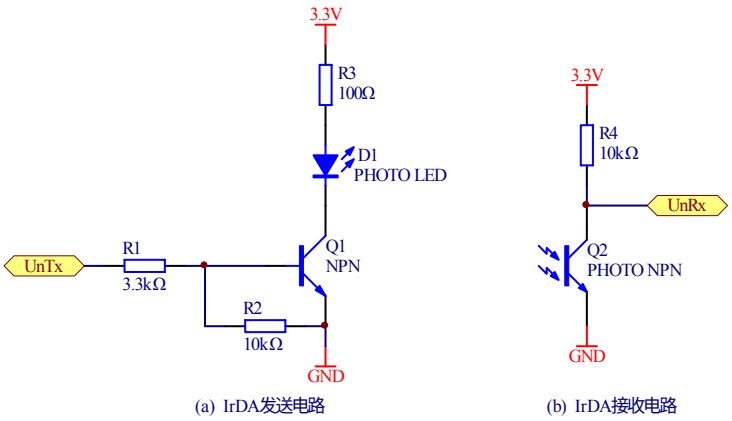

图 1.3 UART 的 IrDA SIR 模块基本应用电路

如图 1.3 所示，为 UART 的 IrDA SIR 模块基本应用电路。D1 为红外发射管，Q2 为红外接收管。

SIR 模块具有两种工作模式：

- 正常的 IrDA 模式

输出管脚上的逻辑 0 电平被当作 3/16 所选波特率位周期的高脉冲发送，而逻辑 1 电平被当作静态低信号发送。这些电平控制红外发送器的驱动器，为每个 0 发送光脉冲。在接收端，接收到的光脉冲给接收器的光敏晶体管基极加电，将其输出拉至低电平。并将 UART 输入管脚变为低电平。

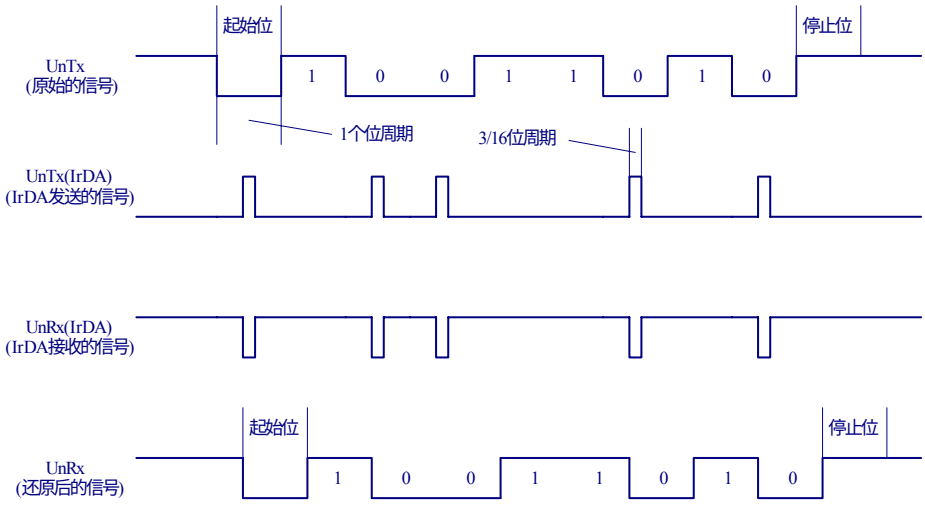

图 1.4 数据 0x59 经 IrDA 发送和接收的信号示意图（LSB 在前）

以发送数据 0x59（字母 Y）为例，含有 IrDA 调制的 UART 发送和接收信号如图 1.4 所示。在 IrDA 模块执行数据发送时，原始的 UnTx 逻辑 0 信号被转换成 3/16 位周期的高脉冲通过开启发射管发送出去，而逻辑 1 不会使发射管产生任何动作（参见图 1.3）。从总体上看，开启发射管的时间仅占很小的比例，因而能节省相当多的功耗。同样，在 UnRx 的接收端，开启接收光敏晶体管的时间也占很小的比例，也起到降低功耗的目的。

- 低功耗 IrDA 模式

在该模式中，可以将发射的红外脉冲宽度设置为内部产生的 IrLPBaud16 信号周期的 3 倍（1.63μs，假设额定频率为 1.8432MHz）。

在正常的 IrDA 模式下，假设以 9600 的波特率发送数据，则发射脉冲宽度为 19.53μs。当改用低功耗模式时，发射脉冲仅持续不到 2μs 的时间，这大大降低了功耗。

IrDA SIR 物理层指定了一个半双工通信链接，发送和接收之间的延迟最小为 10ms。这个延迟要由软件产生，因为 UART 不会自动提供。之所以需要这个延迟，是因为红外接收器电子设备可能会出现偏移，有时从相邻的发送器 LED 耦合而产生的光强甚至会将它变成饱和。这个延迟称作“等待时间”，或接收器“建立时间”。

1.3 UART 库函数

1． 配置与控制

函数 UARTConfigSetExpClk( )用来对 UART 端口的波特率、数据格式进行配置。函数 UARTConfigGetExpClk( )用来获取当前的配置情况。参见表 1.1 和表 1.2 的描述。

在实际编程时，往往用两个形式更简单的宏函数 UARTConfigSet( )和 UARTConfigGet( )来代替上述两个库函数。参见表 1.3 和表 1.4 的描述。

函数 UARTParityModeSet( )用来设置校验位的类型，但在实际编程时一般不会用到它，因为在 UARTConfigSet( )的参数里已经包含了对校验位的配置。函数 UARTParityModeGet( )用来获取校验位的设置情况。参见表 1.5 和表 1.6 的描述。

函数 UARTFIFOLevelSet( )和 UARTFIFOLevelGet( )用来设置和获取收发 FIFO 触发中断时的深度级别。参见表 1.7 和表 1.8 描述。

表 1.1 函数 UARTConfigSetExpClk( )

功能	UART 配置（要求提供明确的时钟速率）
原型	void UARTConfigSetExpClk(unsigned long ulBase, unsigned long ulUARTClk, unsigned long ulBaud, unsigned long ulConfig)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE ulUARTClk：提供给 UART 模块的时钟速率，即系统时钟频率 ulBaud：期望设定的波特率 ulConfig：UART 端口的数据格式，取下列各组数值之间的“或运算”组合形式： ● 数据字长度 UART_CONFIG_WLEN_8 // 8 位数据 UART_CONFIG_WLEN_7 // 7 位数据 UART_CONFIG_WLEN_6 // 6 位数据 UART_CONFIG_WLEN_5 // 5 位数据

	● 停止位 UART_CONFIG_STOP_ONE // 1 个停止位 UART_CONFIG_STOP_TWO // 2 个停止位（可降低误码率） ● 校验位 UART_CONFIG_PAR_NONE // 无校验 UART_CONFIG_PAR_EVEN // 偶校验 UART_CONFIG_PAR_ODD // 奇校验 UART_CONFIG_PAR_ONE // 校验位恒为 1 UART_CONFIG_PAR_ZERO // 校验位恒为 0
返回	无

表 1.2　函数 UARTConfigGetExpClk( )

功能	获取 UART 的配置（要求提供明确的时钟速率）
原型	void UARTConfigGetExpClk(unsigned long ulBase, unsigned long ulUARTClk, unsigned long *pulBaud, unsigned long *pulConfig)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE ulUARTClk：提供给 UART 模块的时钟速率，即系统时钟频率 pulBaud：指针，指向保存获取的波特率的缓冲区 pulConfig：指针，指向保存 UART 端口的数据格式的缓冲区，参见表 1.1 参数 ulConfig 的描述
返回	无

表 1.3　宏函数 UARTConfigSet( )

功能	UART 配置（自动获取时钟速率）
原型	#define UARTConfigSet(a, b, c)　　UARTConfigSetExpClk(a, SysCtlClockGet( ), b, c)
参数	详见表 1.1 的描述
返回	无
说明	本宏函数常常用来代替函数 UARTConfigSetExpClk( )，在调用之前应当先调用 SysCtlClockSet( )函数设置系统时钟（不要使用误差很大的内部振荡器 IOSC、IOSC/4、INT30 等）

示例	// 配置 UART0：波特率 9600，8 个数据位，1 个停止位，无校验
	UARTConfigSet(UART0_BASE, 9600, UART_CONFIG_WLEN_8 |
	UART_CONFIG_STOP_ONE |
	UART_CONFIG_PAR_NONE);
	
	// 配置 UART1：波特率最大，5 个数据位，1 个停止位，无校验
	UARTConfigSet(UART1_BASE, SysCtlClockGet( ) / 16, UART_CONFIG_WLEN_5 |
	UART_CONFIG_STOP_ONE |
	UART_CONFIG_PAR_NONE);
	
	// 配置 UART2：波特率 2400，8 个数据位，2 个停止位，偶校验

	UARTConfigSet(UART2_BASE, 2400, UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_TWO | UART_CONFIG_PAR_EVEN);

表 1.4　宏函数 UARTConfigGet( )

功能	获取 UART 的配置（自动获取时钟速率）
原型	#define UARTConfigGet(a, b, c)　　UARTConfigGetExpClk(a, SysCtlClockGet( ), b, c)
参数	详见表 1.1 和表 1.2 的描述
返回	无

表 1.5　函数 UARTParityModeSet( )

功能	设置指定 UART 端口的校验类型
原型	void UARTParityModeSet(unsigned long ulBase, unsigned long ulParity)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE ulParity：指定使用的校验类型，取下列值之一： UART_CONFIG_PAR_NONE　　// 无校验 UART_CONFIG_PAR_EVEN　　// 偶校验 UART_CONFIG_PAR_ODD　　// 奇校验 UART_CONFIG_PAR_ONE　　// 校验位恒为 1 UART_CONFIG_PAR_ZERO　　// 校验位恒为 0
返回	无

表 1.6　函数 UARTParityModeGet( )

功能	获取指定 UART 端口正在使用的校验类型
原型	unsigned long UARTParityModeGet(unsigned long ulBase)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE
返回	校验类型，与表 1.5 当中参数 ulParity 的取值相同

表 1.7　函数 UARTFIFOLevelSet( )

功能	设置使指定 UART 端口产生中断的收发 FIFO 深度级别
原型	void UARTFIFOLevelSet(unsigned long ulBase, unsigned long ulTxLevel, unsigned long ulRxLevel)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE ulTxLevel：发送中断 FIFO 的深度级别，取下列值之一： UART_FIFO_TX1_8　　// 在 1/8 深度时产生发送中断 UART_FIFO_TX2_8　　// 在 1/4 深度时产生发送中断 UART_FIFO_TX4_8　　// 在 1/2 深度时产生发送中断 UART_FIFO_TX6_8　　// 在 3/4 深度时产生发送中断

	UART_FIFO_TX7_8 // 在 7/8 深度时产生发送中断 注：当发送 FIFO 里剩余的数据减少到预设的深度时触发中断，而非填充到预设深度时触发中断。因此在需要发送大量数据的应用场合，为了减少中断次数提高发送效率，发送 FIFO 中断触发深度级别设置的越浅越好，如设置为 UART_FIFO_TX1_8。 ulRxLevel：接收中断 FIFO 的深度级别，取下列值之一： UART_FIFO_RX1_8 // 在 1/8 深度时产生接收中断 UART_FIFO_RX2_8 // 在 1/4 深度时产生接收中断 UART_FIFO_RX4_8 // 在 1/2 深度时产生接收中断 UART_FIFO_RX6_8 // 在 3/4 深度时产生接收中断 UART_FIFO_RX7_8 // 在 7/8 深度时产生接收中断 注：当接收 FIFO 里已有的数据累积到预设的深度时触发中断，因此在需要接收大量数据的应用场合，为了减少中断次数提高接收效率，接收 FIFO 中断触发深度级别设置的越深越好，如设置为 UART_FIFO_RX7_8。
返回	无

表 1.8　函数 UARTFIFOLevelGet( )

功能	获取使指定 UART 端口产生中断的收发 FIFO 深度级别
原型	void UARTFIFOLevelGet(unsigned long ulBase, unsigned long *pulTxLevel, unsigned long *pulRxLevel)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE pulTxLevel：指针，指向保存发送中断 FIFO 的深度级别的缓冲区 pulRxLevel：指针，指向保存接收中断 FIFO 的深度级别的缓冲区
返回	无

2．　使能与禁止

函数 UARTEnable( )和 UARTDisable( )用来使能和禁止 UART 端口的收发功能。一般是先配置 UART，最后使能收发。当需要修改 UART 配置时，应当先禁止，配置完成后再使能。参见表 1.9 和表 1.10 的描述。

函数 UARTEnableSIR( )和 UARTDisableSIR( )用来使能和禁止 UART 端口串行红外收发功能（IrDA SIR）。参见表 1.11 和表 1.12 的描述。

函数 UARTDMAEnable( )和 UARTDMADisable( )用来使能和禁止 UART 端口的 DMA（Direct Memory Access，直接存储器访问）操作。在 2008 年新推出的 DustDevil 家族里，新增了一个 μDMA 控制器。UART 端口也支持 DMA 传输，能够提高大批量传输数据的效率。参见表 1.13 和表 1.14 的描述。

表 1.9　函数 UARTEnable( )

功能	使能指定 UART 端口的发送和接收操作
原型	void UARTEnable(unsigned long ulBase)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE
返回	无

表 1.10　函数 UARTDisable( )

功能	禁止指定 UART 端口的发送和接收操作
原型	void UARTDisable(unsigned long ulBase)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE
返回	无

表 1.11　函数 UARTEnableSIR( )

功能	使能指定 UART 端口的串行红外功能（IrDA SIR），并选择是否采用低功耗模式
原型	void UARTEnableSIR(unsigned long ulBase, tBoolean bLowPower)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE bLowPower：取值 false，正常的 IrDA 模式 取值 True，选择低功耗 IrDA 模式
返回	无

表 1.12　函数 UARTDisableSIR( )

功能	禁止指定 UART 端口的串行红外功能（IrDA SIR）
原型	void UARTDisableSIR(unsigned long ulBase)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE
返回	无

表 1.13　函数 UARTDMAEnable( )

功能	使能指定 UART 端口 UART 的 DMA 操作
原型	void UARTDMAEnable(unsigned long ulBase, unsigned long ulDMAFlags)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE ulDMAFlags：DMA 特性的位屏蔽，请当取下列值之一或者它们之间的任意“或运算”组合形式： UART_DMA_TX　　// 使能 DMA 发送 UART_DMA_RX　　// 使能 DMA 接收 UART_DMA_ERR_RXSTOP　　// 当 UART 出现错误时停止 DMA 接收
返回	无

表 1.14　函数 UARTDMADisable( )

功能	禁止指定 UART 端口 UART 的 DMA 操作
原型	void UARTDMADisable(unsigned long ulBase, unsigned long ulDMAFlags)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE ulDMAFlags：DMA 特性的位屏蔽，参见表 1.13 参数 ulDMAFlags 的描述。
返回	无

3． 数据收发

函数 UARTCharPut( )以轮询的方式发送数据，如果发送 FIFO 有空位则填充要发送的数据，如果没有空位则一直等待。参见表 1.15 的描述。

函数 UARTCharGet( )以轮询的方式接收数据，如果接收 FIFO 里有数据则读出数据并返回，如果没有数据则一直等待。参见表 1.16 的描述。

函数 UARTSpaceAvail( )用来探测发送 FIFO 里是否有可用的空位。该函数一般用在正式发送之前，以避免长时间的等待。参见表 1.17 的描述。

函数 UARTCharsAvail( )用来探测接收 FIFO 里是否有接收到的数据。该函数一般用在正式接收之前，以避免长时间的等待。参见表 1.18 的描述。

函数 UARTCharPutNonBlocking( )以“无阻塞”的形式发送数据，即不去探测发送 FIFO 里是否有可用空位。如果有空位则放入数据并立即返回，否则立即返回 false 表示发送失败。因此调用该函数时不会出现任何等待。UARTCharNonBlockingPut( )是其等价的宏形式。参见表 1.19 和表 1.21 的描述。

函数 UARTCharGetNonBlocking( )以“无阻塞”的形式接收数据，即不去探测接收 FIFO 里是否有接收到的数据。如果有数据则读取并立即返回，否则立即返回－1 表示接收失败。因此调用该函数时不会出现任何等待。UARTCharNonBlockingGet( )是其等价的宏形式。参见表 1.21 和表 1.22 的描述。

不管是函数 UARTCharPut( )还是 UARTCharPutNonBlocking( )，在发送数据时实际上都是把数据往发送 FIFO 一丢然后就退出，而并非在 UnTx 管脚意义上的真正发送完毕。函数 UARTBusy( )是判断 UART 发送操作是否忙，可用来判定在发送 FIFO 里的数据是否真正发送完毕，这包括最后一个数据的最后停止位。在 UART 转半双工的 RS-485 通信里，需要在发送完一批数据后将传输方向切换为接收，如果此时发送 FIFO 里还有数据未被真正发送出去，则过早的方向切换会破坏发送过程。因此运用函数 UARTBusy( )进行判定是必要的。参见表 1.23 的描述。

函数 UARTBreakCtl( )用来控制线中止（line-break）的产生或撤销。线中止是指 UART 的接收信号 UnRx 一直为 0 的状态（包括校验位和停止位在内）。调用该函数，则会在 UnTx 管脚输出一个连续的 0 电平状态，使对方的 Rx 产生一个线中止条件，并可以触发中断。线中止是个特殊的状态，在某些情况下有特别的用途，例如可以利用它来激活串口 ISP 下载服务程序、智能化自动握手通信等。参见表 1.24 的描述。

表 1.15 函数 UARTCharPut( )

功能	发送 1 个字符到指定的 UART 端口（等待）
原型	void UARTCharPut(unsigned long ulBase, unsigned char ucData)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE ulData：要发送的字符
返回	无（在未发送完毕前不会返回）

表 1.16 函数 UARTCharGet( )

功能	从指定的 UART 端口接收 1 个字符（等待）
原型	long UARTCharGet(unsigned long ulBase)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE
返回	读取到的字符，并自动转换为 long 型（在未收到字符之前会一直等待）

表 1.17　函数 UARTSpaceAvail( )

功能	确认在指定 UART 端口的发送 FIFO 里是否有可用的空间
原型	tBoolean UARTSpaceAvail(unsigned long ulBase)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE
返回	true：在发送 FIFO 里有可用空间 false：在发送 FIFO 里没有可用空间（发送 FIFO 已满）
说明	通常，本函数需要跟函数 UARTCharPutNonBlocking( )配合使用

表 1.18　函数 UARTCharsAvail( )

功能	确认在指定 UART 端口的接收 FIFO 里是否有字符
原型	tBoolean UARTCharsAvail(unsigned long ulBase)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE
返回	true：在接收 FIFO 里有字符 false：在接收 FIFO 里没有字符（接收 FIFO 为空）
说明	通常，本函数需要跟函数 UARTCharGetNonBlocking( )配合使用

表 1.19　函数 UARTCharPutNonBlocking( )

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

表 1.20　函数 UARTCharGetNonBlocking( )

功能	从指定的 UART 端口接收 1 个字符（不等待）
原型	long UARTCharGetNonBlocking(unsigned long ulBase)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE
返回	如果接收 FIFO 里有字符，则立即返回接收到的字符（自动转换为 long 型） 如果接收 FIFO 里没有字符，则立即返回－1（接收失败）
说明	通常，在调用本函数之前应当先调用 UARTCharsAvail( )来确认接收 FIFO 里有字符

表 1.21　宏函数 UARTCharNonBlockingPut( )

功能	发送 1 个字符到指定的 UART 端口（不等待）
原型	#define UARTCharNonBlockingPut(a, b)　　UARTCharPutNonBlocking(a, b)
参数	参见表 1.19 的描述

返回	参见表 1.19 的描述

表 1.22　宏函数 UARTCharNonBlockingGet( )

功能	从指定的 UART 端口接收 1 个字符（不等待）
原型	#define UARTCharNonBlockingGet(a)　　UARTCharGetNonBlocking(a)
参数	参见表 1.20 的描述
返回	参见表 1.20 的描述

表 1.23　函数 UARTBusy( )

功能	确认指定 UART 端口的发送操作忙不忙
原型	tBoolean UARTBusy(unsigned long ulBase)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE
返回	无
说明	本函数是通过探测发送 FIFO 是否为空来确认在发送 FIFO 里的全部字符是否真正发送完毕，该判定在半双工 UART 转 RS-485 通信里可能比较重要

表 1.24　函数 UARTBreakCtl( )

功能	控制指定 UART 端口的线中止（line-break）条件发送或删除
原型	void UARTBreakCtl(unsigned long ulBase, tBoolean bBreakState)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE bBreakState：取值 true，发送线中止条件到 Tx（使 Tx 一直为低电平） 取值 false，删除线中止状态（使 Tx 恢复到高电平）
返回	无

4．　中断控制

UART 端口在收发过程中可产生多种中断，处理起来比较灵活。函数 UARTIntEnable( ) 和 UARTIntDisable( )用来使能和禁止 UART 端口的一个或多个中断。参见表 1.25 和表 1.26 的描述。

函数 UARTIntClear( )用来清除 UART 的中断状态，函数 UARTIntStatus( )用来获取 UART 的中断状态。参见表 1.27 和表 1.28 的描述。

函数 UARTIntRegister( )和 UARTIntUnregister( )用来注册或注销 UART 中断服务函数。参见表 1.29 和表 1.30 的描述。

表 1.25　函数 UARTIntEnable( )

功能	使能指定 UART 端口的一个或多个中断
原型	void UARTIntEnable(unsigned long ulBase, unsigned long ulIntFlags)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE ulIntFlags：指定的中断源，应当取下列值之一或者它们之间的任意“或运算”组合形式： UART_INT_OE　　//　FIFO 溢出错误中断

	UART_INT_BE // BREAK 错误中断 UART_INT_PE // 奇偶校验错误中断 UART_INT_FE // 帧错误中断 UART_INT_RT // 接收超时中断 UART_INT_TX // 发送中断 UART_INT_RX // 接收中断 注：接收中断和接收超时中断通常要配合使用，即 UART_INT_RX | UART_INT_RT
返回	无

表 1.26 函数 UARTIntDisable( )

功能	禁止指定 UART 端口的一个或多个中断
原型	void UARTIntDisable(unsigned long ulBase, unsigned long ulIntFlags)
参数	参见表 1.25 的描述
返回	无

表 1.27 函数 UARTIntClear( )

功能	清除指定 UART 端口的一个或多个中断
原型	void UARTIntClear(unsigned long ulBase, unsigned long ulIntFlags)
参数	参见表 1.25 的描述
返回	无

表 1.28 函数 UARTIntStatus( )

功能	获取指定 UART 端口当前的中断状态
原型	unsigned long UARTIntStatus(unsigned long ulBase, tBoolean bMasked)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE bMasked：如果需要获取原始的中断状态，则取值 false 如果需要获取屏蔽的中断状态，则取值 true
返回	原始的或屏蔽的中断状态

表 1.29 函数 UARTIntRegister( )

功能	注册一个指定 UART 端口的中断服务函数
原型	void UARTIntRegister(unsigned long ulBase, void(*pfnHandler)(void))
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE pfnHandler：函数指针，指向 UART 中断出现时被调用的函数
返回	无

表 1.30 函数 UARTIntUnregister( )

功能	注册指定 UART 端口的中断服务函数

原型	void UARTIntUnregister(unsigned long ulBase)
参数	ulBase：UART 端口的基址，取值 UART0_BASE、UART1_BASE 或 UART2_BASE
返回	无

1.4 UART 例程

1． 简单收发

程序清单 1.1 是 UART 端口简单收发的例子，演示了基本的 UART 的基本配置方法，以及库函数 UARTCharPut( )和 UARTCharGet( )的用法。在主循环里用 UARTCharGet( )等待接收一个字符，然后用 UARTCharPut( )回显，如果遇到回车<CR>则多回显一个换行<LF>。

程序清单 1.1 UART 例程：简单收发

```
#include   "systemInit.h"
#include   <uart.h>

//  UART 初始化
void uartInit(void)
{
    SysCtlPeriEnable(SYSCTL_PERIPH_UART2);                  //  使能 UART 模块
    SysCtlPeriEnable(SYSCTL_PERIPH_GPIOG);                  //  使能 RX/TX 所在的 GPIO 端口

    GPIOPinTypeUART(GPIO_PORTG_BASE,                        //  配置 RX/TX 所在管脚为
                    GPIO_PIN_0 | GPIO_PIN_1);               //  UART 收发功能

    UARTConfigSet(UART2_BASE,                               //  配置 UART 端口
                  9600,                                     //  波特率：9600
                  UART_CONFIG_WLEN_8 |                      //  数据位：8
                  UART_CONFIG_STOP_ONE |                    //  停止位：1
                  UART_CONFIG_PAR_NONE);                    //  校验位：无

    UARTEnable(UART2_BASE);                                 //  使能 UART 端口
}

//  通过 UART 发送字符串
void uartPuts(const char *s)
{
    while (*s != '\0')
    {
        UARTCharPut(UART2_BASE, *(s++));
    }
}

//  主函数（程序入口）
int main(void)
```

```
{
    char c;

    jtagWait( );                                    //  防止 JTAG 失效，重要！
    clockInit( );                                   //  时钟初始化：晶振，6MHz
    uartInit( );                                    //  UART 初始化

    uartPuts("hello, please input a string:\r\n");

    for (;;)
    {
        c = UARTCharGet(UART2_BASE);                //  等待接收字符
        UARTCharPut(UART2_BASE, c);                 //  回显

        if (c == '\r')                              //  如果遇到回车<CR>
        {
            UARTCharPut(UART2_BASE, '\n');          //  多回显一个换行<LF>
        }
    }
}
```

2． 发送 FIFO 工作原理

用库函数 UARTCharPut( )或 UARTCharPutNonBlocking( )发送字符，实质上是将字符扔到发送 FIFO 里就返回，而不是在等待硬件上真正发送完毕后才返回。由于填充 FIFO 极快而真正的硬件发送过程很慢（两者在时间上可相差千倍），因此当发送一个较长字符串时发送 FIFO 很快就会被填满，而最早一个填充的字符可能还在硬件上发送当中。

程序清单 1.2 演示了这一工作原理。在程序里，利用系统节拍定时器 SysTick 先后记录填充完发送 FIFO 的时刻和真正发送完毕的时刻，计算出差值并显示。设置的波特率是 9600，即发送一个字符需要 1041.7μs，程序最后运行的结果显示为 17812μs，理论值是 17×1042＝17708μs，基本吻合。

程序清单 1.2　UART 例程：演示发送 FIFO 工作原理

```
#include   "systemInit.h"
#include   <uart.h>
#include   <systick.h>
#include   <stdio.h>

//  UART 初始化
void uartInit(void)
{
    SysCtlPeriEnable(SYSCTL_PERIPH_UART2);          //  使能 UART 模块
    SysCtlPeriEnable(SYSCTL_PERIPH_GPIOG);          //  使能 RX/TX 所在的 GPIO 端口
```

```
    GPIOPinTypeUART(GPIO_PORTG_BASE,                    //  配置 RX/TX 所在管脚为
                    GPIO_PIN_0 | GPIO_PIN_1);           //  UART 收发功能

    UARTConfigSet(UART2_BASE,                           //  配置 UART 端口
                  9600,                                 //  波特率：9600
                  UART_CONFIG_WLEN_8 |                  //  数据位：8
                  UART_CONFIG_STOP_ONE |                //  停止位：1
                  UART_CONFIG_PAR_NONE);                //  校验位：无

    UARTEnable(UART2_BASE);                             //  使能 UART 端口
}

//  通过 UART 发送字符串
void uartPuts(const char *s)
{
    while (*s != '\0')
    {
        UARTCharPut(UART2_BASE, *s++);
    }
}

//  主函数（程序入口）
int main(void)
{
    int t1, t2;
    char s[40];

    jtagWait( );                                        //  防止 JTAG 失效，重要！
    clockInit( );                                       //  时钟初始化：晶振，6MHz
    uartInit( );                                        //  UART 初始化
    SysTickPeriodSet(256 * 65536);                      //  设置 SysTick 初值
    SysTickEnable( );                                   //  使能 SysTick 计数

    uartPuts("0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ\r\n");   //  发送字符串，隐含使用 FIFO
    t1 = SysTickValueGet( );                            //  记录填充完 FIFO 的时刻
    while (UARTBusy(UART2_BASE));                       //  如果发送忙则等待
    t2 = SysTickValueGet( );                            //  记录真正发送完毕的时刻

    sprintf(s, "%d(us)\r\n", (t1 - t2) / 6);            //  显示间隔时间，单位：us
    uartPuts(s);

    for (;;)
    {
    }
```

```
}
```

3． 发送 FIFO 中断原理

发送数据时，当发送 FIFO 里剩余的数据减少到预设的深度时会触发中断（发送 FIFO 快空了，请赶紧填充），而不是填充到预设的深度时触发中断。为了减少中断次数提高发送效率，发送 FIFO 中断触发深度级别越浅越好。

程序清单 1.3 演示了发送 FIFO 触发中断的原理。在函数 uartInit( )里设置发送 FIFO 触发中断的深度是 2/8，即 4 个字节。在主循环里连续地发送 16 个字符，中间有一定的延迟。在调试例程时，需要在“ TxIntFlag = false ”一行设置观察断点，如果程序停在这里就表明发生了中断，否则表明没有中断。

以 9600 波特率计算，发送 12 个字符所需要的时间是 12ms 多，因此在延迟函数 SysCtlDelay( )里，设置的延迟时间如果在 11ms 以内则不会触发中断，因为不等发送 FIFO 里剩余的字符达到 4 个，FIFO 又开始被填充了。如果延迟时间设置在 12ms 以上时，就会触发中断，因为发送 FIFO 里剩余的字符会达到 4 个而触发中断。

程序清单 1.3　UART 例程：演示发送 FIFO 中断原理

```
#include   "systemInit.h"
#include   <uart.h>

//   UART 初始化
void uartInit(void)
{
      SysCtlPeriEnable(SYSCTL_PERIPH_UART2);                  //   使能 UART 模块
      SysCtlPeriEnable(SYSCTL_PERIPH_GPIOG);                  //   使能 RX/TX 所在的 GPIO 端口

      GPIOPinTypeUART(GPIO_PORTG_BASE,                        //   配置 RX/TX 所在管脚为
                        GPIO_PIN_0 | GPIO_PIN_1);             //   UART 收发功能

      UARTConfigSet(UART2_BASE,                               //   配置 UART 端口
                  9600,                                       //   波特率：9600
                  UART_CONFIG_WLEN_8 |                        //   数据位：8
                  UART_CONFIG_STOP_ONE |                      //   停止位：1
                  UART_CONFIG_PAR_NONE);                      //   校验位：无

      UARTFIFOLevelSet(UART2_BASE,                            //   设置收发 FIFO 中断触发深度
                        UART_FIFO_TX2_8,                      //   发送 FIFO 为 2/8 深度（4B）
                        UART_FIFO_RX6_8);                     //   接收 FIFO 为 6/8 深度（12B）

      UARTIntEnable(UART2_BASE, UART_INT_TX);                 //   使能发送中断
      IntEnable(INT_UART2);                                   //   使能 UART 总中断
      IntMasterEnable( );                                     //   使能处理器中断

      UARTEnable(UART2_BASE);                                 //   使能 UART 端口
```

```
}

//  通过 UART 发送字符串
void uartPuts(const char *s)
{
    while (*s != '\0')
    {
        UARTCharPut(UART2_BASE, *(s++));
    }
}

//  定义发送中断标志，如果出现发送中断，则置位 TxIntFlag
volatile tBoolean TxIntFlag = false;

//  主函数（程序入口）
int main(void)
{
    jtagWait( );                                    //  防止 JTAG 失效，重要！
    clockInit( );                                   //  时钟初始化：晶振，6MHz
    uartInit( );                                    //  UART 初始化

    for (;;)
    {
        uartPuts("0123456789ABCDEF");               //  一次性填满 FIFO
        SysCtlDelay(11 * (TheSysClock / 3000));     //  延迟 11ms 以内不会产生中断
                                                    //  延迟 12ms 以上就会产生中断

        if (TxIntFlag)
        {
            TxIntFlag = false;                      //  在这里设置观察断点
        }
    }
}

//  UART2 中断服务函数
void UART2_ISR(void)
{
    unsigned long ulStatus;

    ulStatus = UARTIntStatus(UART2_BASE, true);     //  读取当前中断状态
    UARTIntClear(UART2_BASE, ulStatus);             //  清除中断状态

    if (ulStatus & UART_INT_TX)                     //  若是发送中断
    {
```

```
            TxIntFlag = true;
        }
}
```

4． 以 FIFO 中断方式发送

我们知道，为了减少中断次数提高发送效率，发送 FIFO 中断触发深度级别越浅越好。程序清单 1.4 演示了以 FIFO 中断方式发送批量数据的方法。在程序里定义有一个全局变量 TxIntNum，能够记录中断产生的次数，次数越少表明发送效率越高。在函数 uartInit( )里设置的发送 FIFO 中断触发深度是 2/8，即 4 个字节。程序的运行结果是，连续发送 38 个字节的数据（数组 TxData[ ]）仅中断处理了 2 次，效率较高。如果把触发深度设置为 6/8，即 12 字节，运行的结果是中断处理了 6 次，效率明显下降。

程序清单 1.4　UART 例程：以 FIFO 中断方式发送数据

```
#include   "systemInit.h"
#include   <uart.h>
#include   <stdio.h>

#define   UARTCharPutNB        UARTCharPutNonBlocking

//   UART 初始化
void uartInit(void)
{
      SysCtlPeriEnable(SYSCTL_PERIPH_UART2);                  //   使能 UART 模块
      SysCtlPeriEnable(SYSCTL_PERIPH_GPIOG);                  //   使能 RX/TX 所在的 GPIO 端口

      GPIOPinTypeUART(GPIO_PORTG_BASE,                        //   配置 RX/TX 所在管脚为
                        GPIO_PIN_0 | GPIO_PIN_1);             //   UART 收发功能

      UARTConfigSet(UART2_BASE,                               //   配置 UART 端口
                  9600,                                       //   波特率：9600
                  UART_CONFIG_WLEN_8 |                        //   数据位：8
                  UART_CONFIG_STOP_ONE |                      //   停止位：1
                  UART_CONFIG_PAR_NONE);                      //   校验位：无

      UARTFIFOLevelSet(UART2_BASE,                            //   设置收发 FIFO 中断触发深度
                        UART_FIFO_TX2_8,                      //   发送 FIFO 为 2/8 深度（4B）
                        UART_FIFO_RX6_8);                     //   接收 FIFO 为 6/8 深度（12B）

      UARTIntEnable(UART2_BASE, UART_INT_TX);                 //   使能发送中断
      IntEnable(INT_UART2);                                   //   使能 UART 总中断
      IntMasterEnable( );                                     //   使能处理器中断

      UARTEnable(UART2_BASE);                                 //   使能 UART 端口
```

```
}

//  定义待发送的数据
const char TxData[ ] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ\r\n";
volatile int TxIndex = 0;                                   //  数组下标变量
volatile tBoolean TxEndFlag = false;                        //  发送结束的标志
volatile int TxIntNum = 0;                                  //  统计发送中断产生的次数

//  填充发送 FIFO（填满 FIFO 之后就退出，不会等待）
void TxFIFOFill(void)
{
    char c;

    for (;;)
    {
        c = TxData[TxIndex];

        if (c == '\0')                                      //  若填充完毕
        {
            TxEndFlag = true;                               //  发送结束标志置位，并跳出
            break;
        }

        if (UARTSpaceAvail(UART2_BASE))                     //  若发送 FIFO 里有可用空间
        {
            UARTCharPutNB(UART2_BASE, c);                   //  填充发送 FIFO
            TxIndex++;
        }
        else                                                //  若没有空间则跳出，不必等待
        {
            break;
        }
    }
}

//  通过 UART 发送字符串
void uartPuts(const char *s)
{
    while (*s != '\0')
    {
        UARTCharPut(UART2_BASE, *s++);
    }
}
```

```
//  主函数（程序入口）
int main(void)
{
    char s[40];

    jtagWait( );                                    //  防止 JTAG 失效，重要！
    clockInit( );                                   //  时钟初始化：晶振，6MHz
    uartInit( );                                    //  UART 初始化

    TxFIFOFill( );                                  //  启动发送过程

    for (;;)
    {
        if (TxEndFlag)                              //  若发送结束则跳出
        {
            TxEndFlag = false;
            break;
        }

        //  对实际的应用，在等待发送的同时，还可以做很多其它事情
    }

    sprintf(s, "TxIntNum = %d\r\n", TxIntNum);      //  显示中断产生的次数
    uartPuts(s);

    for (;;)
    {
    }
}

//  UART2 中断服务函数
void UART2_ISR(void)
{
    unsigned long ulStatus;

    ulStatus = UARTIntStatus(UART2_BASE, true);     //  读取当前中断状态
    UARTIntClear(UART2_BASE, ulStatus);             //  清除中断状态

    if (ulStatus & UART_INT_TX)                     //  若是发送中断
    {
        TxFIFOFill( );                              //  填充发送 FIFO
        TxIntNum++;
    }
}
```

5． 以 FIFO 中断方式接收

程序清单 1.5 演示了以 FIFO 中断方式接收数据的方法。接收数据时，触发 FIFO 中断的条件是当接收 FIFO 里累积的数据增加到预设的深度时触发中断（接收 FIFO 快满了，请赶紧取走）。为了减少中断次数提高接收效率，接收 FIFO 中断触发深度级别越深越好。另外要注意接收超时中断的用法，在使能接收中断的同时一般都还要使能接收超时中断。如果没有使能接收超时中断，则在接收 FIFO 里的部分数据就有可能得不到及时处理。

程序清单 1.5　以 FIFO 中断方式接收数据

```
#include   "systemInit.h"
#include   <uart.h>
#include   <ctype.h>
#include   <string.h>

#define   UARTCharGetNB        UARTCharGetNonBlocking

//   UART 初始化
void uartInit(void)
{
      SysCtlPeriEnable(SYSCTL_PERIPH_UART2);                    //   使能 UART 模块
      SysCtlPeriEnable(SYSCTL_PERIPH_GPIOG);                    //   使能 RX/TX 所在的 GPIO 端口

      GPIOPinTypeUART(GPIO_PORTG_BASE,                          //   配置 RX/TX 所在管脚为
                        GPIO_PIN_0 | GPIO_PIN_1);               //   UART 收发功能

      UARTConfigSet(UART2_BASE,                                 //   配置 UART 端口
                  9600,                                         //   波特率：9600
                  UART_CONFIG_WLEN_8 |                          //   数据位：8
                  UART_CONFIG_STOP_ONE |                        //   停止位：1
                  UART_CONFIG_PAR_NONE);                        //   校验位：无

      UARTFIFOLevelSet(UART2_BASE,                              //   设置发送和接收 FIFO 深度
                        UART_FIFO_TX4_8,                        //   发送 FIFO 为 2/8 深度（4B）
                        UART_FIFO_RX6_8);                       //   接收 FIFO 为 6/8 深度（12B）

      UARTIntEnable(UART2_BASE, UART_INT_RX | UART_INT_RT);     //   使能接收和接收超时中断
      IntEnable(INT_UART2);                                     //   使能 UART 总中断
      IntMasterEnable( );                                       //   使能处理器中断

      UARTEnable(UART2_BASE);                                   //   使能 UART 端口
}

//   通过 UART 发送字符串
```

```
void uartPuts(const char *s)
{
        while (*s != '\0')
        {
                UARTCharPut(UART2_BASE, *s++);
        }
}

//   定义接收缓冲区
#define   MAX_SIZE 40                                   //   缓冲区最大限制长度
char RxBuf[1 + MAX_SIZE];                               //   接收缓冲区
int BufP = 0;                                           //   缓冲区位置变量
tBoolean RxEndFlag = false;                             //   接收结束标志

//   以 FIFO 中断方式接收一个字符串，不回显，返回实际接收到的有效字符数
int uartFIFOGets(char *s, int size)
{
        int n;

        while (!RxEndFlag);
        n = BufP;
        BufP = 0;
        RxEndFlag = false;
        strncpy(s, RxBuf, size);
        s[MAX_SIZE] = '\0';

        return(n);
}

//   主函数（程序入口）
int main(void)
{
        char s[1 + MAX_SIZE];

        jtagWait( );                                    //   防止 JTAG 失效，重要！
        clockInit( );                                   //   时钟初始化：晶振，6MHz
        uartInit( );                                    //   UART 初始化

        for (;;)
        {
                if (uartFIFOGets(s, MAX_SIZE) > 0)
                {
                        uartPuts(s);
                        uartPuts("\r\n");
```

```
            }
        }
}

//  UART2 中断服务函数
void UART2_ISR(void)
{
        char c;
        unsigned long ulStatus;

        ulStatus = UARTIntStatus(UART2_BASE, true);                    //  读取当前中断状态
        UARTIntClear(UART2_BASE, ulStatus);                            //  清除中断状态

        if ((ulStatus & UART_INT_RX) || (ulStatus & UART_INT_RT))      //  若是接收中断或者
        {                                                              //        接收超时中断
                for (;;)
                {
                        if (!UARTCharsAvail(UART2_BASE)) break;        //  若接收 FIFO 里无字符则跳出
                        c = UARTCharGetNB(UART2_BASE);                 //  从接收 FIFO 里读取字符

                        if (c == '\r')
                        {
                                UARTCharPut(UART2_BASE, '\r');         //  回显回车换行<CR><LF>
                                UARTCharPut(UART2_BASE, '\n');
                                RxEndFlag = true;                      //  接收结束标志置位
                                break;
                        }

                        if (isprint(c))                                //  若是可打印字符
                        {
                                if (BufP < MAX_SIZE)
                                {
                                        UARTCharPut(UART2_BASE, c);    //  回显
                                        RxBuf[BufP++] = c;
                                        RxBuf[BufP] = '\0';
                                }
                        }
                }
        }
}
```