第5章 串口通信

CHAPTER 5

随着数字技术水平不断提高，串行方式传输速度也随之得到了一定的改善。如果对传输速度要求不是很高，大都采用串行方式。串口还可以用于各种各样外设之间的通信。例如，目前广泛使用的 GSM/GPRS 手机调制解调器和蓝牙调制解调器可以与微控制器 UART 接口通信。主流的微控制器都带有串口，STM32 自然也不例外。串口作为 STM32 的重要外部接口，同时也是软件开发重要的调试手段，其重要性不言而喻。串行协议提供访问各种各样设备的功能。STM32 串口资源相当丰富，功能也相当强劲。STM32 一般都有多路串口，有分数波特率发生器、支持同步单线通信和半双工单线通信、支持 LIN、支持调制解调器操作、智能卡协议和 IrDA SIR ENDEC 规范、具有 DMA 等。本章将主要介绍串口的基本概念及其工作原理，然后以 STM32F103 为例讲述串口的编程开发，最后通过一个开发案例说明如何使用 STM32F103 的串口。通过 STM32CubeMX 配置串口并生成源程序，将串口发过来的数据接收后并输出到控制台上。

学习目标

- 了解串行通信的基本概念；
- 理解 STM32 的串行接口内部结构；
- 理解并掌握串行接口的使用方法；
- 理解并掌握串口的波特率设置、结构、控制和工作方式；
- 掌握 STM32 串口的配置步骤；
- 熟练使用 STM32CubeMX 创建串口工程；
- 熟练使用 Keil 和 Proteus 对串口调试的仿真过程。

5.1 串口通信基础

通用异步收发/传输器（Universal Asynchronous Receiver/Transmitter，UART）是一种常用的对异步比特流进行编码和解码的设备。UART 是将外围设备提供的数据字节转换成一组单独的比特信息。反过来，又将这组比特信息转换成数据字节传递给外围设备。UART 也是一种计算机与计算机或与仪器仪表间相互交互数据的通信协议。在实际应用中，不仅计算机与外部设备之间常常要进行信息交换，而且计算机之间也需要交换信息，所有这些信息的交换均称为“通信”。按发送数据位数不同，可将通信过程细化为并行通

信和串行通信。UART 是一种设备间非常常用的串行通信方式，因为它简单便捷，大部分电子设备都支持该通信方式，电子工程师在调试设备时也经常使用该通信方式输出调试信息。

在计算机科学里，大部分复杂的问题都可以通过分层来简化。如芯片被分为内核层和片上外设；STM32 标准库则是在寄存器与用户代码之间的软件层。对于通信协议也以分层的方式来理解，最基本的是把它分为物理层和协议层。物理层规定通信系统中具有机械、电子功能部分的特性，确保原始数据在物理媒体的传输。协议层主要规定通信逻辑，统一收发双方的数据打包、解包标准。

5.1.1　并行通信和串行通信

并行通信是指将一个数据块各位的数据同一时刻传送出去。例如构成一个 8 位或 16 位的数据块并行传送，如图 5-1 所示。其特点是传输速度快，但当距离较远、位数又多时导致了通信线路复杂且成本高。

串行通信是一个数据块的每一个数据按位串行传送，如图 5-2 所示。其特点是通信线路简单，只要一对传输线就可以实现通信(如电话线)，从而大大降低了成本，特别适用于远距离通信。其缺点是传送速度慢。异步串行通信的最基本形式是通过连接两个设备的对称导线来实现的。通常计算机与另一台计算机间相互通信时，计算机 A 通过 Tx 数据线将数据发送到计算机 B，接收数据的计算机 B 则通过 Rx 线接收来自发送计算机 A 的比特流；如果当接收计算机 B 通过 Tx 线发送数据到发送计算机 A 的比特流时，则计算机 A 需要通过 Rx 线接收这些数据，整个数据收发过程如图 5-2 所示。这种通信模式也称为“异步”，因为主机和目标没有共享同一个时钟。

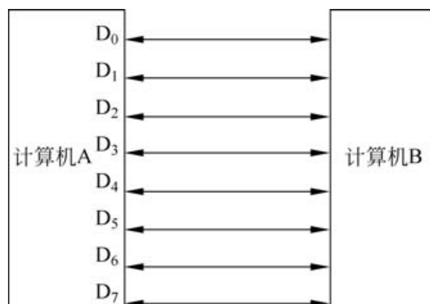

图 5-1　并行通信

图 5-2　串行通信

为进一步说明并行通信和串行通信的传输原理及其优缺点，表 5-1 比较了并行通信和串行通信两种方式的区别。

表 5-1　并行和串行通信的区别

比　较　项	并 行 通 信	串 行 通 信
传输原理	数据各个位同时传输	数据按位顺序传输
优点	速度快	占用引脚资源少
缺点	占用引脚资源多	速度相对较慢

5.1.2 单工、半双工和全双工

由于在串行通信中数据是在两机之间进行传送的，按照数据通信方向，可将发送数据过程分为单工、半双工和全双工三种方式。

1. 单工方式

单工（Simplex）方式：数据传输是由发送机向接收机的单一固定方向上传输数据，数据传输只支持数据在一个方向上传输，通信双方设备中发送机与接收机分工明确，一方固定为发送端，另一方固定为接收端。采用单工方式通信的典型发送设备如早期计算机的读卡器，典型的接收设备如打印机。计算机与打印机之间的串行通信就是单工方式，因为只能由计算机向打印机传递数据，而不可能有相反方向的数据传递。单工方式通信的示意图如图 5-3 所示。

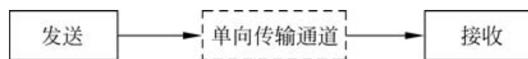

图 5-3　单工通信的示意图

2. 半双工方式

半双工（Half Duplex）方式：通信双方设备之间只有一个通信回路，接收和发送不能同时进行，只能分时接收和发送，允许数据在两个方向上传输，设备既是发送机，也是接收机，两台设备可以相互传送数据。在某一时刻，只允许数据在一个方向上传输，它实际上是一种切换方向的单工通信；它不需要独立的接收端和发送端，两者可以合并一起使用一个端口。某一时刻则只能向一个方向传送数据。例如，步话机是半双工设备，因为在一个时刻只能有一方说话。半双工通信的示意图如图 5-4 所示。

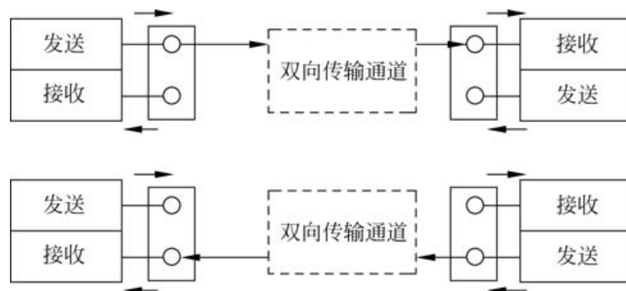

图 5-4　半双工通信的示意图

3. 全双工方式

全双工（Full Duplex）方式：通信双方设备之间的数据发送和接收可以同时进行，既是发送机，也是接收机，允许数据同时在两个方向上传输。全双工通信是两个单工通信方式的结合，需要独立的接收端和发送端。两台设备可以在两个方向上同时传送数据。例如，电话是全双工设备，因为双方可同时说话。全双工通信的示意图如图 5-5 所示。

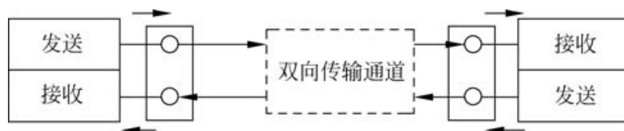

图 5-5 全双工通信的示意图

5.1.3 波特率

波特率既是指在串行通信中每秒传输数据的速度，也是指数据信号对载波的调制速率。它用单位时间内载波调制状态改变次数来表示，单位为波特。比特率指单位时间内传输的比特数，单位为 bit/s(bps)。对于 USART 波特率与比特率相等，不区分这两个概念。由于数据是按位进行传送的，传送速率往往用每秒传送的字节数来表示。在波特率选定之后，对于设计者来说，就是如何得到能满足波特率要求的发送时钟脉冲和接收时钟脉冲。收发双方对发送或接收的数据速率(即波特率)要有一定的约定。由于收发双方的信号中没有直接编码时钟信息，发送端和接收端各自独立地保持时钟，并以一个一致的频率(倍数)运行。而且发送端时钟和接收端时钟不同步，也不能保证完全相同的频率，但它们必须在频率上足够接近(优于 2%)才能正常恢复传输的数据。因此，异步通信中由于没有时钟信号，所以两个通信设备之间需要约定好波特率，即每个码元的长度，以便对信号进行解码，图 5-6 中用虚线分开的每一格就是代表一个码元。常见的波特率为 4800bps、9600bps、115 200bps 等。波特率越大，传输速率越快。

为了理解接收机如何提取编码数据，假设它有一个从空闲状态开始以波特率倍数(例如 16 倍)运行的时钟，如图 5-6 所示。接收机“采样”它的 Rx 信号，直到它检测到一个高低转换。然后等待 1.5 个周期(24 个时钟周期)，以在它估计为 0 位数据中心的位置采样 Rx 信号。然后接收机以位周期间隔(16 个时钟周期)采样 Rx，直到它读取了剩余的 7 个数据位和停止位。从那一点开始，这个过程不断重复，直到成功地从帧中提取数据需要超过 10.5 个周期。为了正确地检测到停止位，目标时钟相对于主机时钟的漂移要小于 0.5 个周期。

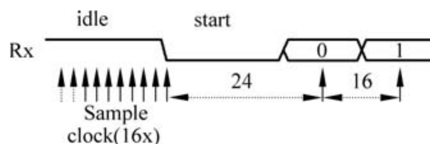

图 5-6 UART 信号编码

5.1.4 同步通信和异步通信

同步通信是一种连续串行传送数据的通信方式，每一次通信仅传送一帧信息。同步通信需要带时钟信号传输。比如 SPI，I^2C 通信接口。这里的信息帧与异步通信中的字符帧不同，通常含有若干数据字符。

采用同步通信时，将许多字符组成一个信息组，这样字符可以一个接一个地传输，但是在每组信息(通常称为帧)的开始要加上同步字符。在没有信息传输时，要填上空字符，因为同步传输不允许有间隙。换言之，一个信息帧中包含许多字符，每个信息帧用同步字符作为开始，一般将同步字符和空字符用同一个代码。在同步传输过程中，一个字符可以对应 5～8 位。当然，对同一个传输过程，所有字符对应同样的数位，比如 n 位。这样传输时，按每 n 位划分为一个时间片，发送端在一个时间片中发送一个字符，接收端则在一个时间片

中接收一个字符。在整个系统中,由一个统一的时钟控制发送端的发送和空字符用同一个代码。接收端是能识别同步字符的,当检测到有一串数位和同步字符相匹配时,就认为开始一个信息帧。于是,把此后的数位作为实际传输信息来处理。同步通信中,在数据开始传送前用同步字符来指示,同步字符通常为1～2个,数据传送由时钟系统实现发送端和接收端同步,即检测到规定的同步字符后,连续按顺序传送数据,直到通信结束。同步传送时,字符与字符之间没有间隙,不用起始位和停止位,仅在数据块开始时用同步字符SYNC(即同步字符8位)来指示,同步传送格式如图5-7所示。

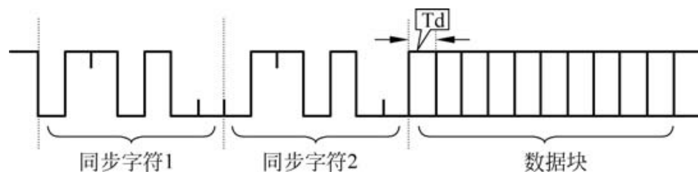

图5-7 同步串行通信格式

同步通信中数据块传送时去掉了字符开始和结束的标志,因而其速度高于异步传送,但这种通信方式对硬件的结构要求比较高。在同步通信中,收发设备上方会使用一根信号线传输信号,在时钟信号的驱动下双方进行协调,同步数据,如图5-8所示。例如,通信中通常双方会统一规定在时钟信号的上升沿或者下降沿对数据线进行采样。

异步通信是一种不带时钟同步信号的通信方式,比如UART、单总线等。异步通信在发送字符时,所发送的字符之间的时间间隔可以是任意的。当然,接收端必须时刻做好接收的准备。发送端可以在任意时刻发送字符,因此必须在每一个字符的开始和结束的地方加上标志,即加上开始位和停止位,以便使接收端能够正确地将每一个字符接收下来。异步通信的优点是通信设备简单、便宜,但传输效率较低(因为开始位和停止位的开销所占比例较大),如图5-9所示。

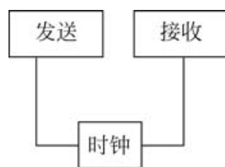

图5-8 同步串行通信

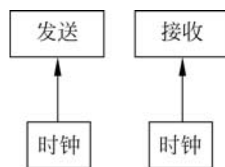

图5-9 异步串行通信

在异步通信中不使用时钟信号进行数据同步,它们直接在数据信号中穿插一些用于同步的信号位,或者将主机数据进行打包,以数据帧的格式传输数据。通信中还需要双方规约好数据的传输速率(也就是波特率)等,以便更好地同步。常用的波特率有9600bps或115 200bps等。

在异步通信中,数据或字符是一帧一帧地传送的。帧定义为一个字符的完整的通信格式,一般也称为帧格式。在帧格式中,一个字符由4部分组成:起始位、数据位、奇偶校验位和停止位。首先是一个起始位"0"表示字符的开始;然后是5～8位数据,规定低位在前,高位在后;接下来是奇偶校验位(该位可省略);最后是一个停止位"1",用以表示字符的结束,停止位可以是1位、1.5位、2位,不同的计算机规定有所不同。异步串行通信格式如

图 5-10 所示。

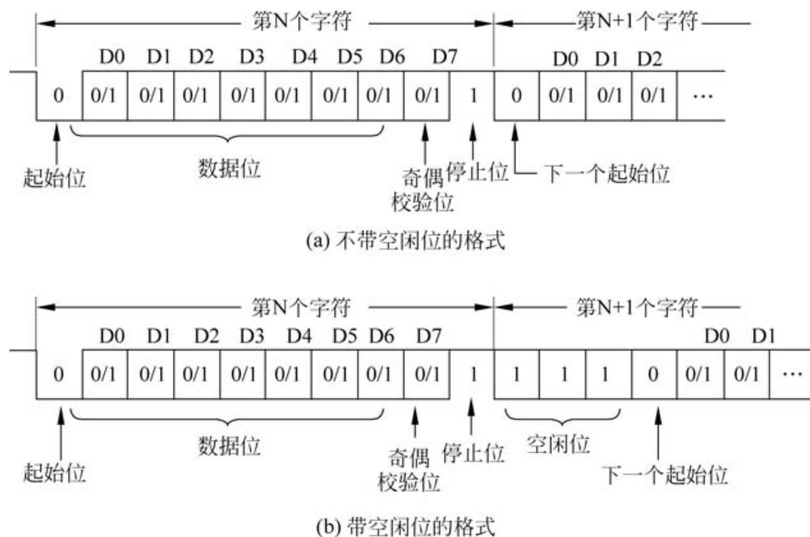

图 5-10　异步串行通信格式

由于异步通信每传送一帧有固定格式，通信双方只需按约定的帧格式发送和接收数据，所以硬件结构比较简单。此外，它还能利用奇偶校验位检测错误，因此，这种通信方式应用比较广泛。

同步通信与异步通信的区别：

（1）同步通信要求接收端时钟频率和发送端时钟频率一致，发送端发送连续的比特流；异步通信时不要求接收端时钟和发送端时钟同步，发送端发送完一个字节后，可经过任意长的时间间隔再发送下一个字节。

（2）同步通信效率高；异步通信效率较低。

（3）同步通信较复杂，双方时钟的允许误差较小；异步通信简单，双方时钟可允许一定误差。

（4）同步通信可用于点对多点；异步通信只适用于点对点。

（5）在同步通信中，数据信号所传输的内容绝大部分是有效数据，而异步通信中则会包含数据帧的各种标识符，所以同步通信效率高；但是同步通信双方的时钟允许误差小，时钟稍稍出错就可能导致数据错乱，异步通信双方的时钟允许误差较大。

5.1.5　串口引脚连接

对于两个芯片串口之间的连接，两个芯片 GND 共地，同时 TxD 和 RxD 交叉连接，如图 5-11 所示。这里的交叉连接的意思是，芯片 1 的 RxD 连接芯片 2 的 TxD，芯片 2 的 RxD 连接芯片 1 的 TxD。这样，两个芯片之间就可以进行 TTL 电平通信了。

- RxD：数据输入引脚表示接收数据；
- TxD：数据发送引脚表示发送数据。

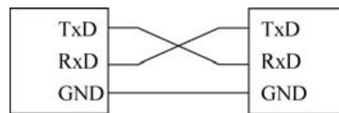

图 5-11　串口通信

如果下位机与计算机（或上位机）通过串口相连，那么上位机和下位机需要共地，但不能直接交叉连接。尽管计算机和下位机都有 TxD 和 RxD 引脚，但是计算机（或上位

机)使用的都是 RS-232 接口(通常为 DB9 封装)。RS-232 接口是 9 针(或引脚),通常是 TxD 和 RxD 经过电平转换得到的,因此不能直接交叉连接。故要想使得下位机与计算机使用 RS-232 标准传输数据信号,需要将下位机的芯片的输入/输出端口电平转换成 RS-232 类型,再交叉连接。这是因为 RS-232 电平标准的信号不能直接被微控制器芯片直接识别,所以这些信号会经过一个“电平转换芯片”转换成控制器能识别的“TTL 校准”的电平信号,才能实现通信。常见的电子电路中常使用 TTL 的电平标准,理想状态下,使用 5V 表示二进制逻辑 1,使用 0V 表示逻辑 0;而为了增加串口通信的远距离传输及抗干扰能力,RS-232 的电平标准是+15/+13V 表示 0,-15/-13V 表示 1。一般都是两个通信设备的 DB9 接口之间通过串口信号线建立起连接。

在上面的串口信号线通信方式中,由于微控制器芯片的串口和 RS-232 的电平标准是不一样的,需要经过电平转换后才可实现它们之间通信。根据通信使用的电平标准不同,串口通信可分为 TTL 标准及 RS-232 标准,见表 5-2。

表 5-2 TTL 电平标准与 RS-232 电平标准

通信标准	电平标准
RS-232	逻辑 0:+3～+15V 逻辑 1:-15～-3V
5V TTL	逻辑 0:0～0.5V 逻辑 1:2.4～5V

5.2 STM32 串口通信基础

STM32 的串口通信接口有两种,分别是通用异步收发器(Universal Asynchronous Receiver and Transmitter,UART)、通用同步异步收发器(Universal Synchronous Asynchronous Receiver and Transmitter,USART)。而对于大容量 STM32F10x 系列芯片,分别有 3 个 USART 和 2 个 UART。STM32 芯片具有多个 USART 外设用于串口通信,即通用同步异步收发器可以灵活地与外部设备进行全双工数据交换。有别于 USART,它还具有 UART 外设,它是在 USART 基础上裁剪掉了同步通信功能,只有异步通信。简单区分同步和异步就是看通信时是否需要对外提供时钟输出,在实际串口通信应用中主要使用 UART。

USART 只需两根信号线即可完成双向通信,对硬件要求低,使得很多模块都预留 USART 接口来实现与其他模块或者控制器进行数据传输,比如 GSM 模块、WiFi 模块、蓝牙模块等。在硬件设计时,注意还需要一根“共地线”。经常使用 USART 来实现控制器与计算机之间的数据传输,这使得调试程序变得非常方便,比如可以把一些变量的值、函数的返回值、寄存器标志位等通过 USART 发送到串口调试助手,这样可以非常清楚程序的运行状态,当正式发布程序时再把这些调试信息去除即可。这样不仅可以将数据发送到串口调试助手,还可以在串口调试助手发送数据给微控制器芯片,微控制器芯片的程序根据接收到的数据进行下一步操作。

对于 STM32F103 来说,为实现串行通信,在其内部都设计有串口电路,USART 通过软件编程既可以用作通用异步接收和发送器,也可以用作同步移位寄存器。USART 满足

外部设备对工业标准不归零编码(Non-Return-to-Zero,NRZ)异步串行数据格式的要求,并且使用了小数波特率发生器,可以提供多种波特率,使得它的应用更加广泛。USART 支持同步单向通信和半双工单线通信;还支持局域互联网络(LIN)、智能卡(Smart Card)协议与 lrDA(红外线数据协会)SIR ENDEC 规范。USART 支持使用 DMA,可实现高速数据通信。有关 DMA 具体应用将在 DMA 章节作具体讲解。

USART 在 STM32 的应用最多莫过于“打印”程序信息,一般在硬件设计时都会预留一个 USART 通信接口连接计算机,用于在调试程序中可以把一些调试信息“打印”在计算机的串口调试助手工具上,从而了解程序运行是否正确、指出运行出错位置等。STM32 的 USART 输出的是 TTL 电平信号,若需要 RS-232 标准的信号,则可使用 MAX3232 芯片进行转换。

5.2.1 STM32F103 芯片的 USART 引脚

USART 引脚在 STM32F103 芯片的具体分布见表 5-3。

表 5-3 STM32F103 芯片的 USART 引脚

引 脚	APB2	APB1	
	USART1	USART2	USART3
Tx	PA9/PB6	PA2/PD5	PB10/PD8/PC10
Rx	PA10/PB7	PA3/PD6	PB11/PD9/PC11
SCLK	PA8	PA4/PD7	PB12/PD10/PC12
nCTS	PA11	PA0/PD3	PB13/PD11
nRTS	PA12	PA1/PD4	PB14/PD12

观察表 5-3 可发现很多 USART 的功能引脚有多个 GPIO 引脚可选,这方便硬件设计,只要在程序编程时软件绑定 GPIO 引脚并对串口进行初始化即可。

STM32F103 内置了 3 个通用同步/异步收发器(USART1、USART2 和 USART3)和 2 个通用异步收发器(UART4 和 UART5)。其中,USART1 的时钟来源于 APB2 总线时钟,其最大频率为 72MHz,其余的时钟来源于 APB1 总线时钟。UART 只具有异步传输功能,所以没有 SCLK、nCTS 和 nRTS 功能引脚。

5.2.2 USART 功能框图

USART 功能框图如图 5-12 所示。按串口通信功能划分,将这个框图分成上、中、下三个部分,分别是数据处理、收发控制和波特率控制。

1. 数据处理

数据处理在功能框图 5-12 的上部分,数据从 Rx 进入接收移位寄存器,后进入接收数据寄存器,最终供 CPU 或者 DMA 来进行读取;数据从 CPU 或者 DMA 传递过来,进入发送数据寄存器,后进入发送移位寄存器,最终通过 Tx 发送出去。USART 支持 DMA 传输,可以实现高速数据传输。

1)功能引脚

下面对数据处理部分的功能引脚进行介绍。

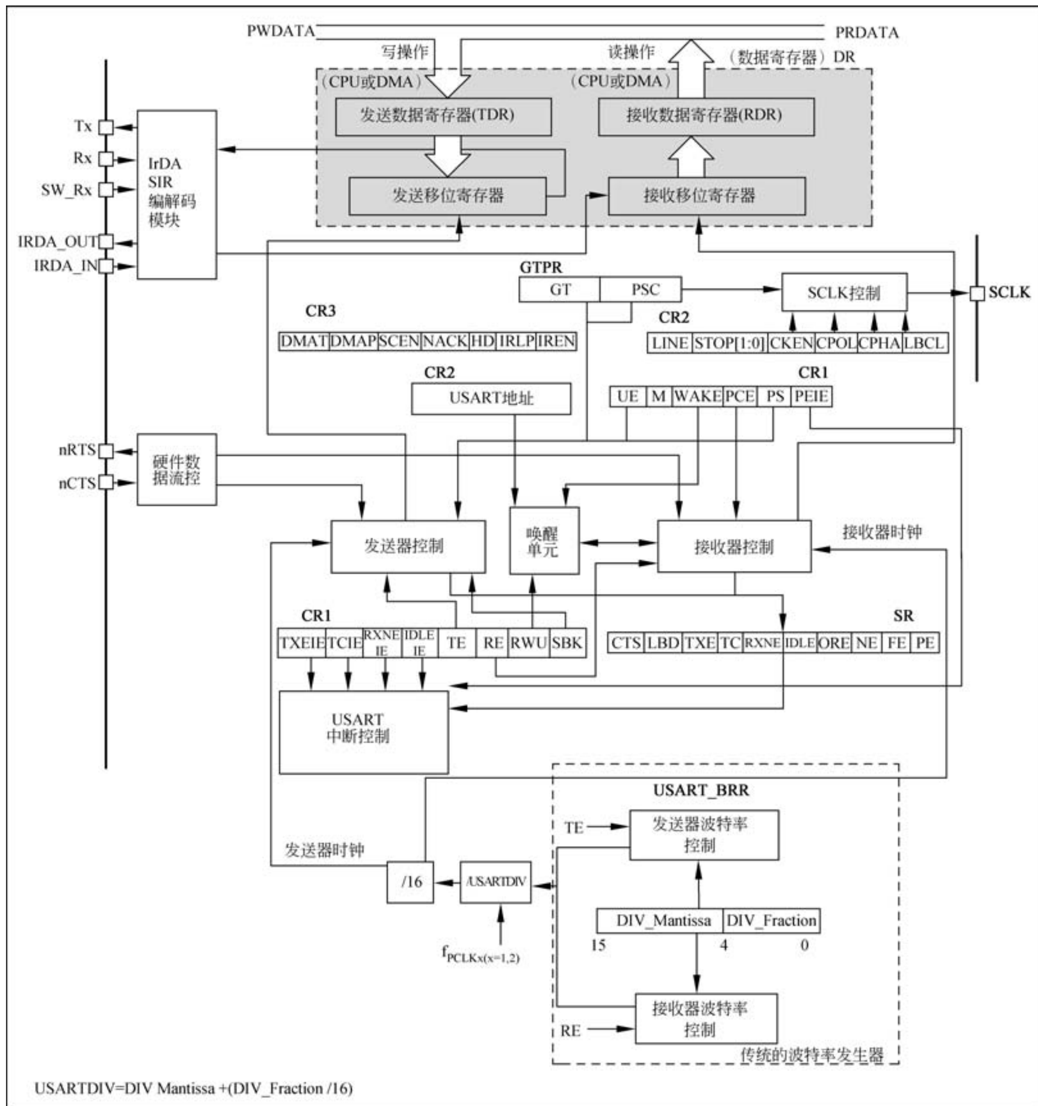

图 5-12 USART 功能框图

Tx：发送数据输出引脚。

Rx：接收数据输入引脚。

SW_Rx：数据接收引脚，只用于单线和智能卡模式，属于内部引脚，没有具体外部引脚。

nRTS：请求以发送(Request To Send)，n 表示低电平有效。如果使能 RTS 流控制，当 USART 接收器准备好接收新数据时就会将 nRTS 变成低电平；当接收寄存器已满时，nRTS 将被设置为高电平。该引脚只适用于硬件流控制。

nCTS：清除以发送(Clear To Send)，n 表示低电平有效。如果使能 CTS 流控制，发送器在发送下一帧数据之前会检测 nCTS 引脚，如果为低电平，表示可以发送数据，如果为高

电平则在发送完当前数据帧之后停止发送。该引脚只适用于硬件流控制。

SCLK：发送器时钟输出引脚。这个引脚仅适用于同步模式。

2）数据寄存器

USART数据寄存器（USART_DR）只有低9位有效，并且第9位数据是否有效要取决于USART控制寄存器1（USART_CR1）的M位设置，当M位为0时表示8位数据字长，当M位为1表示9位数据字长，一般使用8位数据字长。

USART_DR包含了已发送的数据或者接收到的数据。USART_DR实际是包含了两个寄存器，一个专门用于发送的可写TDR，另一个专门用于接收的可读RDR。当进行发送操作时，往USART_DR写入数据会自动存储在TDR内；当进行读取操作时，向USART_DR读取数据会自动提取RDR数据。

TDR和RDR都是介于系统总线和移位寄存器之间。串行通信传输的单位是一个数据位，发送时把TDR内容转移到发送移位寄存器，然后把移位寄存器数据每一位发送出去，接收时把接收到的每一位顺序保存在接收移位寄存器内然后才转移到RDR。

2. 收发控制器

收发控制器在框图5-12的中间部分。USART有专门控制发送的发送器、控制接收的接收器、中断控制等。使用USART之前需要向USART_CR1寄存器的UE位置1使能USART。发送或者接收数据字长可选8位或9位，由USART_CR1的M位控制。

1）发送器

当USART_CR1寄存器的发送使能位TE置1时，启动数据发送，发送移位寄存器的数据会在Tx引脚输出，如果是同步模式SCLK也输出时钟信号。一个字符帧发送需要三个部分：起始位、数据帧、停止位。起始位是一个位周期的低电平，位周期就是每一位占用的时间；数据帧就是要发送的8位或9位数据，数据是从最低位开始传输的；停止位是一定时间周期的高电平。停止位时间长短可以通过USART控制寄存器2（USART_CR2）的STOP[1：0]位控制，可选0.5个、1个、1.5个和2个停止位。默认使用1个停止位。2个停止位适用于正常USART模式、单线模式和调制解调器模式。0.5个和1.5个停止位用于智能卡模式。具体发送字符时序图如图5-13所示。

当发送使能位TE置1之后，发送器开始会先发送一个空闲帧（一个数据帧长度的高电平），接下来就可以往USART_DR寄存器写入要发送的数据。在写入最后一个数据后，需要等待USART状态寄存器（USART_SR）的TC位为1，表示数据传输完成，如果USART_CR1寄存器的TCIE位置1，将产生中断。在发送数据时，几个比较重要的标志位的总结如表5-4所示。

2）接收器

如果将USART_CR1寄存器的RE位置1，使能USART接收，使得接收器在Rx线开始搜索起始位。在确定到起始位后就根据Rx线电平状态把数据存放在接收移位寄存器内。接收完成后就把接收移位寄存器数据移到RDR内，并把USART_SR寄存器的RXNE位置1。如果USART_CR2寄存器的RXNEIE置1，则可以产生中断。在接收数据时，几个比较重要的标志位的总结如表5-5所示。

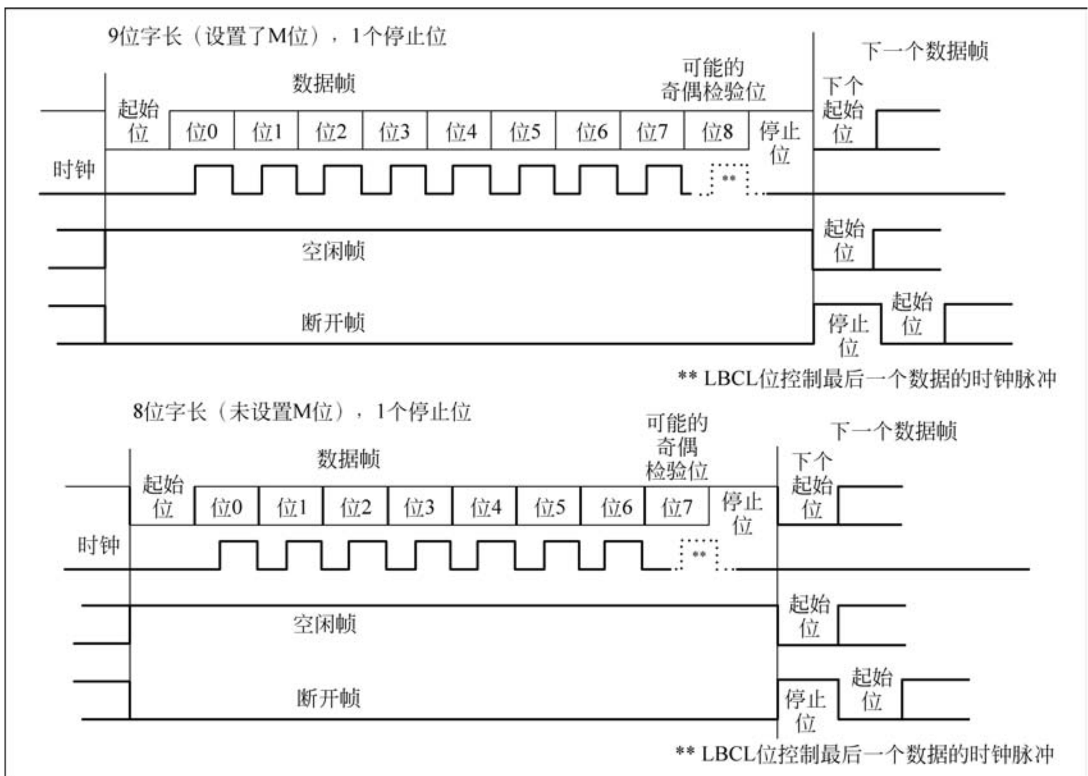

图 5-13　字符发送时序图

表 5-4　串口发送寄存器标志位

名　称	描　　述
TE	发送使能
TxE	发送寄存器为空，发送单个字节数据时使用
TC	发送完成，发送多个字节数据时使用
TxIE	发送完成中断使能

表 5-5　串口接收寄存器标志位

名　称	描　　述
RE	接收使能
RxNE	读数据寄存器非空
RxNEIE	发送完成中断使能

为得到一个信号真实情况，需要用一个比这个信号频率高的采样信号去检测，称为过采样。这个采样信号的频率大小决定最后得到源信号的准确度，一般频率越高得到的准确度越高。但为了得到越高频率采样信号也越困难，运算和功耗等也会增加，所以一般选择合适的采样频率就好。

接收器可配置为不同的采样技术，以实现从噪声中提取有效的数据。USART_CR1 寄存器的 OVER8 位用来选择不同的采样方法，如果 OVER8 位设置为 1 则采用 8 倍过采样，即用 8 个采样信号采样 1 位数据；如果 OVER8 位设置为 0 则采用 16 倍过采样，即用 16 个采样信号采样 1 位数据。

USART 的起始位检测需要用到特定序列。如果在 Rx 线识别到该特定序列就认为是检测到了起始位。起始位检测对使用 16 倍或 8 倍过采样的序列都是一样的。16 倍过采样速度虽然没有 8 倍过采样那么快，但得到的数据更加精准，其最大速度为 $f_{PCLK}/16$，f_{PCLK} 为 USART 外设的时钟。16 倍采样过程如图 5-14 所示。

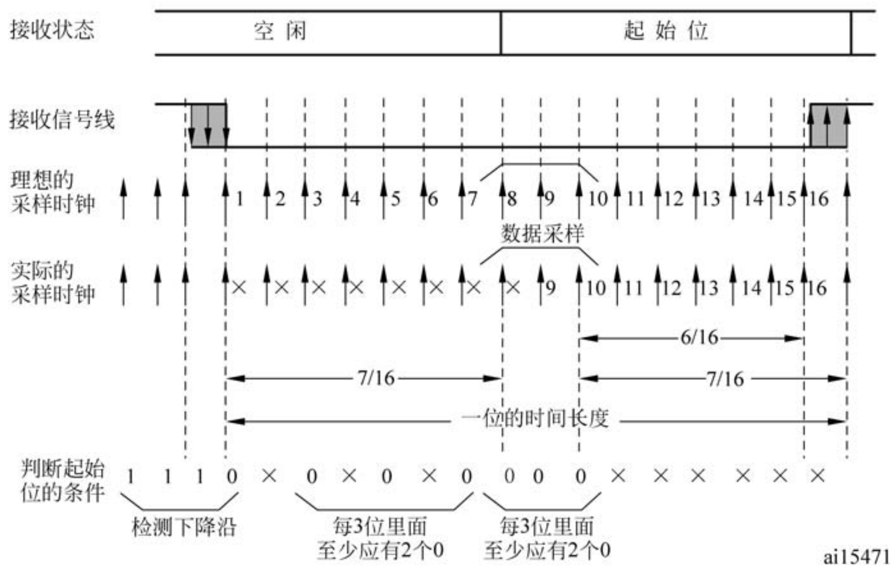

图 5-14　16 倍采样过程

图 5-14 中使用第 8、9、10 次脉冲的值决定该位的电平状态。图中的×表示电平任意，1 或 0 皆可。

3）中断控制

STM32F103 的 USART 包含多个中断请求事件，例如发送数据寄存器为空、发送完成、准备好读取接收到的数据、奇偶校验错误、检测到空闲线路等。每一个中断请求事件对应了寄存器的事件标志位。如果要想使能某个中断请求事件，需要事先中断使能控制位，见表 5-6。

表 5-6　USART 中断请求

中断事件	事件标志	使能控制位
发送数据寄存器为空	TxE	TxEIE
CTS 标志	CTS	CTSIE
发送完成	TC	TCIE
准备好读取接收到的数据	RxNE	RxNEIE
检测到上溢错误	ORE	
检测到空闲线路	IDLE	IDLEIE
奇偶校验错误	PE	PEIE
断路标志	LBD	LBDIE
多缓冲通信中的噪声标志、上溢错误和帧错误	NF/ORE/FE	EIE

3. 波特率控制

波特率控制在框图 5-12 的下部分。UART 的发送和接收都需要波特率来进行控制，在接收移位寄存器、发送移位寄存器都有一个进入的箭头，分别连接到接收器控制、发送器控制。而这两者连接的又是接收器时钟、发送器时钟。也就是说，异步通信尽管没有时钟同步信号，但是在串口内部是提供了时钟信号来进行控制的。从图 5-12 中可以看到，接收器

时钟和发送器时钟又被连接到同一个控制单元,也就是说它们共用一个波特率发生器。同时也可以看到,接收器时钟(发生器时钟)的计算方法、USARTDIV 的计算方法。

USART 的发送器和接收器使用相同的波特率。波特率的常用值有 2400bps、9600bps、19 200bps、115 200bps。接收器和发送器的波特率在 USARTDIV 的整数和小数寄存器中的值应设置成相同。计算公式如下:

$$\text{Tx/Rx 波特率}=\frac{f_{\text{PLCK}}}{(16*\text{USARTDIV})}$$

其中,f_{PLCK} 是 USART 外设的时钟;USARTDIV 是一个存放在波特率寄存器(USART_BRR)的无符号定点数。这 12 位的值设置在 USART_BRR 寄存器。

下面讲解如何通过设定寄存器值得到波特率的值。选取 USART1 作为实例讲解,当使用 16 倍过采样时,即 OVER8=0,设定 f_{PLCK}=90MHz,为得到 115 200bps 的波特率,此时,

$$115\,200=\frac{90\,000\,000}{(16*\text{USARTDIV})}$$

解得 USARTDIV=48.825 125。其中,USARTDIV=DIV_Mantissa+(DIV_Fraction/16),可算得 DIV_Fraction=0xD,DIV_Mantissa=0x30,即应该设置 USART_BRR 的值为 0x30D。在计算 DIV_Fraction 时经常出现小数情况,经过取舍得到整数,这样会导致最终输出的波特率较目标值略有偏差。下面从 USART_BRR 的值为 0x30D 开始计算得出实际输出的波特率大小。由 USART_BRR 的值为 0x30D,可得 DIV_Fraction=13,DIV_Mantissa=48,所以 USARTDIV=48+13÷16=48.8125,所以实际波特率为 115 237;这个值与目标波特率的误差为 0.03%,这么小的误差在正常通信的允许范围内。8 倍过采样时计算原理是一样的。

5.2.3 STM32 的 UART 特点

STM32 的 UART 特点如下:

(1) 全双工异步通信;

(2) 分数波特率发生器系统,提供精确的波特率。发送和接收共用的可编程波特率,最高可达 4.5Mbps;

(3) 可编程的数据字长度(8 位或者 9 位);

(4) 可配置的停止位(支持 1 或者 2 位停止位);

(5) 使用 DMA 多缓冲器通信;

(6) 单独的发送器和接收器使能位;

(7) 检测标志:①接收缓冲器标志;②发送缓冲器标志;③传输结束标志;

(8) 多个带标志的中断源,触发中断;

(9) 校验控制,四个错误检测标志。

5.2.4 STM32 中的 UART 参数

STM32 中串口异步通信需要定义的参数:起始位、数据位(8 位或者 9 位)、奇偶校验位(第 9 位)、停止位(1 位,1.5 位,2 位)、波特率设置。

UART 串口通信的数据包以帧为单位,常用的帧结构为:1 位起始位、8 位数据位、1 位

奇偶校验位(可选)、1位停止位。字长设置如图5-15所示。

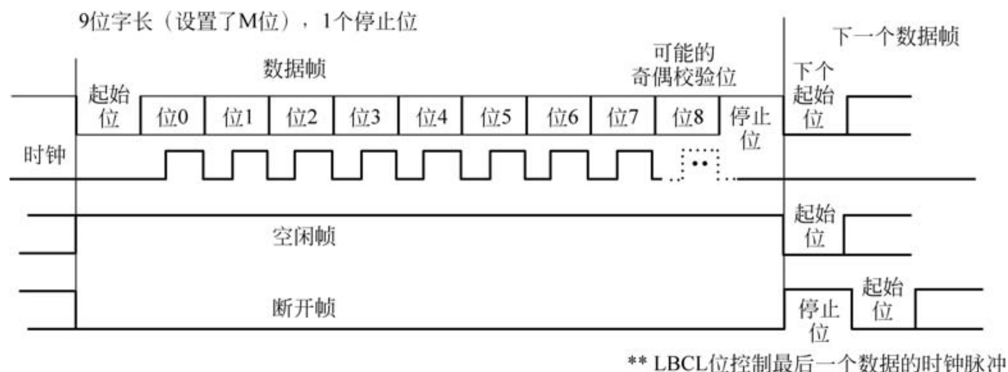

图5-15 字长设置

奇偶校验位分为奇校验和偶校验两种,是一种简单的数据误码校验方法。奇校验是指每帧数据中,包括数据位和奇偶校验位的全部9位中“1”的个数必须为奇数;偶校验是指每帧数据中,包括数据位和奇偶校验位的全部9位中“1”的个数必须为偶数。校验方法除了奇校验(odd)、偶校验(even)之外,还可以有0校验(space)、1校验(mark)以及无校验(noparity)。0/1校验:不管有效数据中的内容是什么,校验位总为0或者1。

串口通信的数据包由发送设备通过自身的TxD接口传输到接收设备的RxD接口。在串口通信的协议层中,规定了数据包的内容,它由起始位、主体数据、校验位以及停止位组成,通信双方的数据包格式要约定一致才能正常收发数据,其组成见图5-16。

图5-16 串口通信协议

5.3 STM32串口的库函数

5.3.1 串口的标准库函数

标准库函数对每个外设都建立了一个初始化结构体,比如USART_InitTypeDef。结构体成员用于设置串口外设工作参数,并由串口外设初始化配置函数,比如USART_Init()调用。这些设定参数将会设置外设相应的寄存器,达到配置串口外设工作环境的目的。

串口初始化结构体和初始化库函数配合使用是标准库的关键理解了初始化结构体每个成员的意义,有助于串口通信的配置。初始化结构体定义在stm32f1xx_usart.h文件中,初始化库函数定义在stm32f1xx_usart.c文件中,编程时可以结合这两个文件的注释,以便正确使用这些功能。

USART初始化结构体如下:

```
typedef struct {
    uint32_t USART_BaudRate;                //此成员配置 USART 通信波特率
```

```
    uint16_t USART_WordLength;              //指定在一个帧中传输或接收的数据位数
    uint16_t USART_StopBits;                //指定所传输的停止位数
    uint16_t USART_Parity;                  //指定奇偶校验模式
    uint16_t USART_Mode;                    //指定接收模式是否启用或禁用传输模式
    uint16_t USART_HardwareFlowControl;     //指定是否启用或禁用硬件流控制模式
} USART_InitTypeDef;
```

USART_BaudRate：波特率设置。一般设置为 2400bps、9600bps、19 200bps、115 200bps。标准库函数会根据设定值计算得到 USARTDIV 值，并设置 USART_BRR 寄存器值。

USART_WordLength：数据帧字长，可选 8 位或 9 位。它设定 USART_CR1 寄存器的 M 位的值。如果没有使能奇偶校验控制，一般使用 8 数据位；如果使能了奇偶校验，则一般设置为 9 数据位。

USART_StopBits：停止位设置，可选 0.5 个、1 个、1.5 个和 2 个停止位，它设定 USART_CR2 寄存器的 STOP[1：0]位的值，一般我们选择 1 个停止位。

USART_Parity：奇偶校验控制选择，可选 USART_Parity_No（无校验）、USART_Parity_Even（偶校验）以及 USART_Parity_Odd（奇校验），它设定 USART_CR1 寄存器的 PCE 位和 PS 位的值。启用奇偶校验时，将计算出的奇偶校验插入传输数据的 MSB 位置（字长度设置为 9 数据位时第 9 位；字长度设置为 8 数据位时第 8 位）。

USART_Mode：USART 模式选择，有 USART_Mode_Rx 和 USART_Mode_Tx，允许使用逻辑或运算选择，它设定 USART_CR1 寄存器的 RE 位和 TE 位。

USART_HardwareFlowControl：硬件流控制选择，只有在硬件流控制模式才有效，可有四种选择：使能 RTS、使能 CTS、同时使能 RTS 和 CTS、不使能硬件流。

当使用同步模式时需要配置 SCLK 引脚输出脉冲的属性，标准库通过一个时钟初始化结构体 USART_ClockInitTypeDef 来设置，因此该结构体内容也只有在同步模式才需要设置。

USART 时钟初始化结构体如下：

```
typedef struct {
    uint16_t USART_Clock;                   //时钟使能控制
    uint16_t USART_CPOL;                    //时钟极性
    uint16_t USART_CPHA;                    //时钟相位
    uint16_t USART_LastBit;                 //最尾位时钟脉冲
} USART_ClockInitTypeDef;
```

USART_Clock：同步模式下 SCLK 引脚上时钟输出使能控制，可选禁止时钟输出 USART_Clock_Disable 或开启时钟输出 USART_Clock_Enable；如果使用同步模式发送，一般都需要开启时钟。它设定 USART_CR2 寄存器的 CKEN 位的值。

USART_CPOL：同步模式下 SCLK 引脚上输出时钟极性设置，可设置在空闲时 SCLK 引脚为低电平（USART_CPOL_Low）或高电平（USART_CPOL_High），即设定 USART_CR2 寄存器的 CPOL 位的值。

USART_CPHA：同步模式下 SCLK 引脚上输出时钟相位设置，可设置在时钟第一个变化沿捕获数据 USART_CPHA_1Edge 或在时钟第二个变化沿捕获数据，即设定 USART_CR2 寄存器的 CPHA 位的值。USART_CPHA 与 USART_CPOL 配合使用可以获得多

种模式时钟关系。

USART_LastBit：选择在发送最后一个数据位时时钟脉冲是否在 SCLK 引脚输出，可以是不输出脉冲 USART_LastBit_Disable、输出脉冲 USART_LastBit_Enable，即设定 USART_CR2 寄存器的 LBCL 位的值。

下面列出了常用的操作串口的标准库函数：

```
void USART_DeInit(USART_TypeDef * USARTx);
/**
  * @brief  将 USARTx 外围寄存器取消初始化为其默认重置值
  * @param  USARTx:选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *       USART1,USART2,USART3,UART4 or UART5
  * @retval 无
  */
void USART_Init(USART_TypeDef * USARTx, USART_InitTypeDef * USART_InitStruct);
/**
  * @brief  根据 USART_InitStruct 中指定参数初始化 USARTx 外围设备
  *
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5.
  * @param  USART_InitStruct: 指向包含指定 USART 外围设备的配置信息的 USART_InitTypeDef
  * 结构体的指针
  * @retval 无
  */
void USART_StructInit(USART_InitTypeDef * USART_InitStruct);
/**
  * @brief  向每个 USART_InitStruct 成员填充其默认值
  * @param  USART_InitStruct: 指向将被初始化 USART_InitTypeDef 结构体的指针
  * @retval 无
  */
void USART_ClockInit(USART_TypeDef * USARTx, USART_ClockInitTypeDef * USART_ClockInitStruct);
/**
  * @brief  根据 USART_ClockInitStruct 结构体中指定的参数初始化 USARTx 外围时钟
  * @param  USARTx:用于选择 USART 外围设备,其中 x 可以为 1、2 和 3
  * @param  USART_ClockInitStruct: 指向包含指定 USART 外围设备的配置信息的 USART_
  * ClockInitTypeDef 结构体的指针
  * @note UART4 和 UART5 不能使用智能卡和同步模式
  * @retval 无
  */
void USART_ClockStructInit(USART_ClockInitTypeDef * USART_ClockInitStruct);
/**
  * @brief  向每个 USART_ClockInitStruct 结构体成员填充其默认值
  * @param  USART_ClockInitStruct:指向将被初始化 USART_ClockInitTypeDef 结构体的指针
  *
  * @retval 无
  */
void USART_Cmd(USART_TypeDef * USARTx, FunctionalState NewState);
/**
  * @brief  启用或禁用指定的 USART 外围设备
```

```
  * @param  USARTx:选择 USART 或 UART 外围设备
  *          此参数可以是以下值之一:
  *            USART1, USART2, USART3, UART4 or UART5
  * @param  NewState: USARTx 外围设备的新状态
  *          此参数可以是:启用或禁用
  * @retval 无
  */
void USART_ITConfig(USART_TypeDef * USARTx, uint16_t USART_IT, FunctionalState NewState);
/**
  * @brief  启用或禁用指定的 USART 中断
  * @param  USARTx:选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  USART_IT:指定要启用或禁用的 USART 中断源
  *   此参数可以是以下值之一:
  *     @arg USART_IT_CTS:  CTS 变更中断(UART4 和 UART5 不可用)
  *     @arg USART_IT_LBD:  LIN 断路检测中断
  *     @arg USART_IT_TXE:  传输数据寄存器为空的中断
  *     @arg USART_IT_TC:   传输完成中断
  *     @arg USART_IT_RXNE: 接收数据寄存器,而不是空的中断
  *     @arg USART_IT_IDLE: 空闲线检测中断
  *     @arg USART_IT_PE:   奇偶校验错误中断
  *     @arg USART_IT_ERR:  错误中断(帧错误、噪声错误、溢出错误)
  * @param  NewState: 指定 USARTx 中断的新状态
  *   此参数可以是:启用或禁用
  * @retval 无
  */
void USART_DMACmd(USART_TypeDef * USARTx, uint16_t USART_DMAReq, FunctionalState NewState);
/**
  * @brief  启用或禁用 USART 的 DMA 接口
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  USART_DMAReq: 指定 DMA 请求
  *   此参数可以是以下值的任意组合:
  *     @arg USART_DMAReq_Tx: USART 的 DMA 传输请求
  *     @arg USART_DMAReq_Rx: USART 的 DMA 接收请求
  * @param  NewState: DMA 请求源的新状态
  *   此参数可以是:启用或禁用
  * @note 除 STM32 高密度值设备(STM32F10X_HD_VL)外,UART5 不使用 DMA 模式
  *
  * @retval 无
  */

void USART_SetAddress(USART_TypeDef * USARTx, uint8_t USART_Address);
/**
  * @brief  设置 USART 节点的地址
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  USART_Address: 指示 USART 节点的地址
```

```
  * @retval 无
  */
void USART_WakeUpConfig(USART_TypeDef * USARTx, uint16_t USART_WakeUp);
/**
  * @brief  选择 USART 的 WakeUp 方法
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  USART_WakeUp: 指定 USART 的 WakeUp 方法
  *   此参数可以是以下值之一:
  *     @arg USART_WakeUp_IdleLine: 通过空闲线路检测进行唤醒
  *     @arg USART_WakeUp_AddressMark: 通过地址标记进行唤醒
  * @retval 无
  */
void USART_ReceiverWakeUpCmd(USART_TypeDef * USARTx, FunctionalState NewState);
/**
  * @brief  确定 USART 是否处于闲置模式
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  NewState: USART 闲置模式的新状态
  *   此参数可以是:启用或禁用
  * @retval 无
  */
void USART_LINBreakDetectLengthConfig(USART_TypeDef * USARTx, uint16_t USART_LINBreakDetectLength);
/**
  * @brief  设置 USART 的 LIN 断开线检测长度
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  USART_LINBreakDetectLength: 指定 LIN 断开线检测的长度
  *   此参数可以是以下值之一:
  *     @arg USART_LINBreakDetectLength_10b: 10 位中断检测
  *     @arg USART_LINBreakDetectLength_11b: 11 位中断检测
  * @retval 无
  */
void USART_LINCmd(USART_TypeDef * USARTx, FunctionalState NewState);
/**
  * @brief  启用或禁用 USART 的 LIN 模式
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  NewState: USART 的 LIN 模式的新状态
  *   此参数可以是:启用或禁用
  * @retval 无
  */
void USART_SendData(USART_TypeDef * USARTx, uint16_t Data);
/**
  * @brief  通过 USARTx 外围设备传输单个数据
```

```
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  Data: 要传输的数据
  * @retval 无
  */
uint16_t USART_ReceiveData(USART_TypeDef * USARTx);
/**
  * @brief  返回 USARTx 外围设备接收到的最新数据
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @retval 已接收到的数据
  */
void USART_SendBreak(USART_TypeDef * USARTx);
/**
  * @brief  传输断开符字符
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @retval 无
  */
void USART_SetGuardTime(USART_TypeDef * USARTx, uint8_t USART_GuardTime);
/**
  * @brief  设置指定的 USART 防护时间
  * @param  USARTx: 用于选择 USARTx 外围设备.其中,x 可以是 1、2 或 3
  * @param  USART_GuardTime: 指定防护时间
  * @note UART4 和 UART5 没有保护时间位
  * @retval 无
  */
void USART_SetPrescaler(USART_TypeDef * USARTx, uint8_t USART_Prescaler);
/**
  * @brief  设置系统时钟的预分频器
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  USART_Prescaler: 指定预分频器的时钟
  * @note   该功能用于具有 UART4 和 UART5 的 IrDA 模式
  * @retval 无
  */
void USART_SmartCardCmd(USART_TypeDef * USARTx, FunctionalState NewState);
/**
  * @brief  启用或禁用 USARTx 的智能卡模式
  * @param  USARTx: 用于选择 USARTx 外围设备,其中,x 可以是 1、2 或 3
  * @param  NewState: 智能卡模式的新状态
  *   此参数可以是:启用或禁用
  * @note UART4 和 UART5 不适用于智能卡模式
  * @retval 无
```

```
  */
void USART_SmartCardNACKCmd(USART_TypeDef * USARTx, FunctionalState NewState);
/**
  * @brief 启用或禁用 NACK 传输
  * @param  USARTx: 用于选择 USARTx 外围设备,其中,x 可以是 1、2 或 3
  * @param  NewState: NACK 传输器的新状态
  *   此参数可以是:启用或禁用
  * @note UART4 和 UART5 不适用于智能卡模式
  * @retval 无
  */
void USART_HalfDuplexCmd(USART_TypeDef * USARTx, FunctionalState NewState);
/**
  * @brief 启用或禁用 USARTx 的半双工路通信
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  NewState: USART 通信的新状态
  *   此参数可以是:启用或禁用
  * @retval 无
  */
void USART_OverSampling8Cmd(USART_TypeDef * USARTx, FunctionalState NewState);
/**
  * @brief  启用或禁用 USART 的 8 倍过采样模式
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  NewState: USART 一位采样方法的新状态
  *   此参数可以是:启用或禁用
  * @note
  *     在调用 USART_Init()之前,必须先调用此函数
  *     有正确的波特率分配值功能
  * @retval 无
  */
void USART_OneBitMethodCmd(USART_TypeDef *USARTx,FunctionalState NewState);
/**
  * @brief  启用或禁用 USART 的一位采样方法
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  NewState: USART 一位采样方法的新状态
  *   此参数可以是:启用或禁用
  * @retval 无
  */
void USART_IrDAConfig(USART_TypeDef * USARTx, uint16_t USART_IrDAMode);
/**
  * @brief  配置 USART 的 IrDA 接口
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
```

```
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  USART_IrDAMode: 指定 IrDA 模式
  *   此参数可以是以下值之一:
  *     @arg USART_IrDAMode_LowPower
  *     @arg USART_IrDAMode_Normal
  * @retval 无
  */
void USART_IrDACmd(USART_TypeDef * USARTx, FunctionalState NewState);
/**
  * @brief  启用或禁用 USART 的 IrDA 接口
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  NewState: IrDA 模式的新状态
  *   此参数可以是:启用或禁用
  * @retval 无
  */
FlagStatus USART_GetFlagStatus(USART_TypeDef * USARTx,uint16_t USART_FLAG);
/**
  * @brief  检查是否设置了指定的 USART 标志位
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  USART_FLAG: 指定要检查的标志位
  *   此参数可以是以下值之一:
  *     @arg USART_FLAG_CTS:  CTS 变更标志位(不适用于 UART4 和 UART5)
  *     @arg USART_FLAG_LBD:  LIN 断路检测标志位
  *     @arg USART_FLAG_TXE:  传输数据寄存器为空标志位
  *     @arg USART_FLAG_TC:   传输完成的标志位
  *     @arg USART_FLAG_RXNE: 接收数据寄存器不是空标志位
  *     @arg USART_FLAG_IDLE: 空闲线检测标志位
  *     @arg USART_FLAG_ORE:  覆盖错误标志位
  *     @arg USART_FLAG_NE:   噪声错误标志位
  *     @arg USART_FLAG_FE:   帧设置错误标志位
  *     @arg USART_FLAG_PE:   奇偶校验错误标志位
  * @retval USART_FLAG 的新状态(设置或重置)
  */
void USART_ClearFlag(USART_TypeDef * USARTx, uint16_t USART_FLAG);
/**
  * @brief  清除 USARTx 的挂起标志位
  * @param  USARTx: 选择 USART 或 UART 外围设备
  *   此参数可以是以下值之一:
  *   USART1, USART2, USART3, UART4 or UART5
  * @param  USART_FLAG: 指定要清除的标志
  *   此参数可以是以下值的任意组合:
  *     @arg USART_FLAG_CTS: CTS 变更标志位(不适用于 UART4 和 UART5)
  *     @arg USART_FLAG_LBD: LIN 断开检测标志位
  *     @arg USART_FLAG_TC: 传输完成的标志位
```

```
 *    @arg USART_FLAG_RXNE: 接收数据寄存器不是空标志位
 *
 * @note
 * - PE(奇偶校验错误)、FE(帧错误)、NE(噪声错误)、ORE(溢出错误)和 IDLE(检测到空闲行)挂起位
通过软件序列清除:对 USART_SR 寄存器执行读取操作 USART_GetITStatus(),然后对 USART_DR 寄存器
执行读取操作 USART_ReceiveData()
 *  - RXNE 挂起位也可以通过读取 USART_DR 寄存器 USART_ReceiveData()来清除
 * - TC 挂起位也可以通过软件序列清除:向 USART_SR 寄存器进行读取操作 USART_GetITStatus(),
然后向 USART_DR 寄存器进行写入操作 USART_SendData()
 *   - TXE 挂起位只通过写入 USART_DR 寄存器 USART_SendData()来清除
 * @retval 无
 */
ITStatus USART_GetITStatus(USART_TypeDef * USARTx, uint16_t USART_IT);
/**
 * @brief  指定要检查的 USART 中断源
 * @param  USARTx: 选择 USART 或 UART 外围设备
 *   此参数可以是以下值之一:
 *   USART1, USART2, USART3, UART4 or UART5
 * @param  USART_IT: 指定要检查 USART 的中断源
 *   此参数可以是以下值之一:
 *    @arg USART_IT_CTS:  CTS 变更中断(UART4 和 UART5 不可用)
 *    @arg USART_IT_LBD:  LIN 断开检测中断
 *    @arg USART_IT_TXE:  传输数据寄存器为空中断
 *    @arg USART_IT_TC:   传输完成中断
 *    @arg USART_IT_RXNE: 接收数据寄存器非空的中断
 *    @arg USART_IT_IDLE: 空闲线检测中断
 *    @arg USART_IT_ORE: 溢出错误中断
 *    @arg USART_IT_NE:  噪声错误中断
 *    @arg USART_IT_FE:  帧错误中断
 *    @arg USART_IT_PE:  奇偶校验错误中断
 * @retval USART_IT 的新状态(设置或重置)
 */
void USART_ClearITPendingBit(USART_TypeDef * USARTx, uint16_t USART_IT);
/**
 * @brief  清除 USARTx 的中断等待位
 * @param  USARTx: 选择 USART 或 UART 外围设备
 *   此参数可以是以下值之一:
 *   USART1, USART2, USART3, UART4 or UART5
 * @param  USART_IT: 指定要清除的中断等待位
 *   此参数可以是以下值之一:
 *    @arg USART_IT_CTS: CTS 变更中断(UART4 和 UART5 不可用)
 *    @arg USART_IT_LBD: LIN 断开检测中断
 *    @arg USART_IT_TC: 传输完成中断
 *    @arg USART_IT_RXNE: 接收数据寄存器非空的中断
 * @note
 *   - PE(奇偶校验错误)、FE(帧错误)、NE(噪声错误)、ORE(溢出错误)和 IDLE(检测到空闲行)挂起
位通过软件序列清除:对 USART_SR 寄存器执行读取操作(USART_GetITStatus()),然后对 USART_DR 寄
存器执行读取操作 USART_ReceiveData()
```

```
  *     - RXNE 挂起位也可以通过读取 USART_DR 寄存器 USART_ReceiveData()来清除
  *     - TC 挂起位也可以通过软件序列清除:向 USART_SR 寄存器进行读取操作(USART_GetITStatus()),
然后向 USART_DR 寄存器进行写入操作 USART_SendData()
  *     - TXE 挂起位只通过写入 USART_DR 寄存器 USART_SendData()来清除
  * @retval 无
  */
```

5.3.2 STM32 串口通信配置步骤

STM32 串口通信过程配置步骤如图 5-17 所示。

图 5-17　串口配置步骤

STM32 串口通信的具体步骤如下：

(1) 串口时钟和 GPIO 时钟使能；

(2) 串口复位(不必须)；

(3) GPIO 设置；注意 RxD 和 TxD 在设置过程中设置输入/输出方法的不同(AF_PP|IN_FLOATING)；

(4) 串口参数初始化(这里要初始化很多参数,要记住每个参数的设定值,通信双方要约定参数保持一致)；

(5) 开启中断,使能 NVIC；

(6) 实现串口的使能；

(7) 编写中断处理函数；

(8) 实现串口数据的收发；

(9) 实现串口传输状态获取。

一个 STM32 的串口初始化程序的整体结构,如以下代码所示：

```
#include <stm32f10x.h>
#include <stm32f10x_rcc.h>
#include <stm32f10x_gpio.h>
#include <misc.h>
#include"stm32f10x.h"

int main(void)
```

```
{
    /*初始化 GPIO*/
    GPIO_Config();
    AFIO_Config();
    /*串口配置*/
    USART_Config();
    while(1);
}
```

5.4 STM32 串口通信实例

5.4.1 STM32 串口通信实例的标准库函数开发

本节将学习 STM32F103 的片内外设 USART1 的数据通信。本实例通过使用 STM32F103 的 PA9 和 PA10 实现串口的发送和接收数据。最后利用 Proteus 进行仿真，依据仿真结果说明串口中断实验的通信过程。图 5-18 是 STM32 串口通信的应用实例原理图。

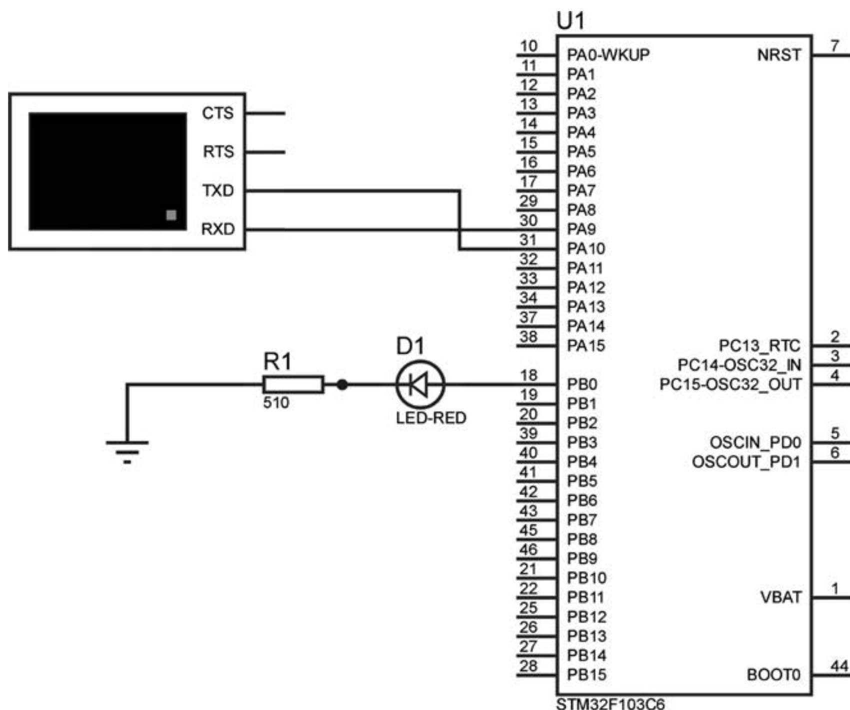

图 5-18 STM32 串口通信的应用实例原理图

图 5-19 为 STM32 串口通信的应用实例程序的流程图。

本节是通过重写 printf()函数重新定向输出到串口。由于 C 标准库函数 printf()实质是 fputc()函数的宏定义，所以需要对 fputc()函数进行重写。当运行环境检查到用户编写了与 C 库函数同名的函数时优先调用用户重写的同名函数，从而实现了对 printf()函数的

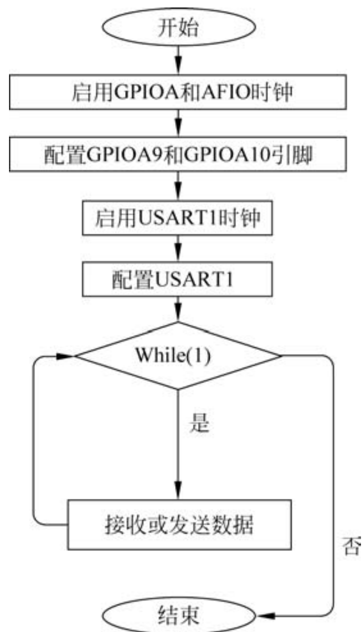

图 5-19　STM32 串口通信的应用实例程序流程

重定向输出到串口。具体代码如下：

```
#include "stm32f10x.h"
#include <stdio.h>

int fputc(int ch, FILE *f)
{
    USART_SendData(USART1, (unsigned char) ch);
    //USART1 可以换成 USART2 等
    while (!(USART1->SR & USART_FLAG_TXE));
    return (ch);
}
void delay(u16 num)
{
  u16 i,j;
  for(i=0;i<num;i++)
    for(j=0;j<0x800;j++);
}

int main(void)
{
  /* USART1 Init with 9600bps */
  GPIO_InitTypeDef GPIO_InitStructure;
  USART_InitTypeDef USART_InitStructure;

  /* Enable GPIO Alternate Function clock */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_AFIO, ENABLE);
```

```
    /* Configure USART1 Tx(PA.9) as Alternate Function Push - Pull */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* Configure USART1 Rx(PA.10) as In - Floating */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* Enable USART1 clock */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);

    /* USARTx configured as follow */
    USART_InitStructure.USART_BaudRate = 9600;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl;
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
    USART_Init(USART1, &USART_InitStructure);

    /* Clear USART1 Transmission complete flag */
    USART_ClearFlag(USART1, USART_FLAG_TC);

    /* Enable USART1 */
    USART_Cmd(USART1, ENABLE);

    /* Infinite loop */
    while (1)
    {
        printf("STM32 串口 1 测试!!!\n");
        delay(1000);
    }
}
```

运用 Keil 软件对上述代码进行编译,编译成功后开启 Debug 模式,如图 5-20 所示。观察串口 1 的配置参数和串口 1 的显示窗口。在串口 1 的显示窗口中每隔 1s 打印"STM32 串口 1 测试!!!"。

5.4.2 STM32CubeMX 基础配置

1. STM32CubeMX 工程配置

前面已经介绍了如何利用 STM32CubeMX 新建工程,本节在此基础上对串口 1 进行配置。首先要使能 GPIO 时钟,然后使能复用功能时钟,同时要把 GPIO 模式设置为复用功能

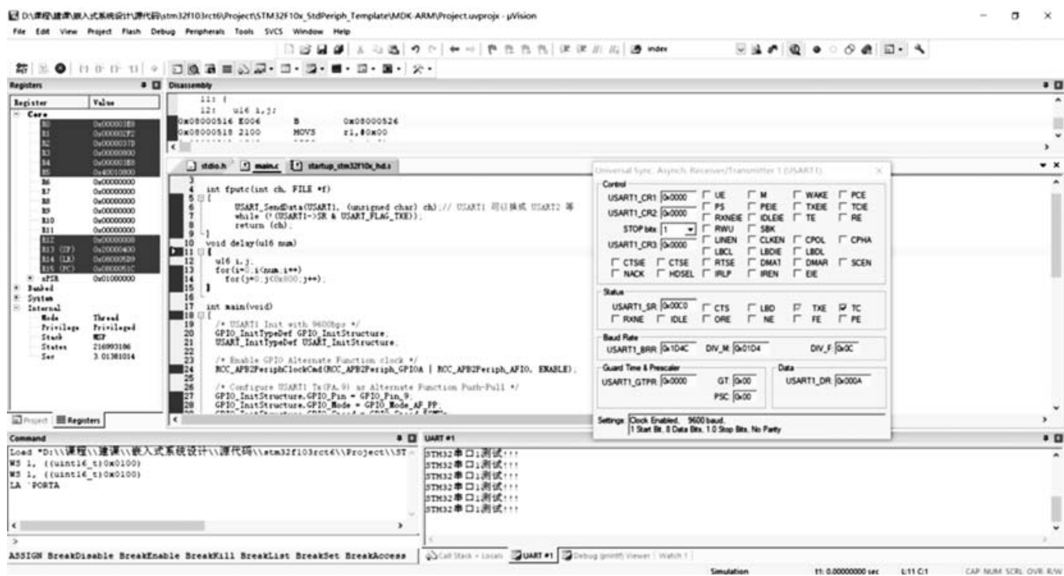

图 5-20　仿真结果

对应的模式,配置 PA9 和 PA10 引脚,如图 5-21 所示。

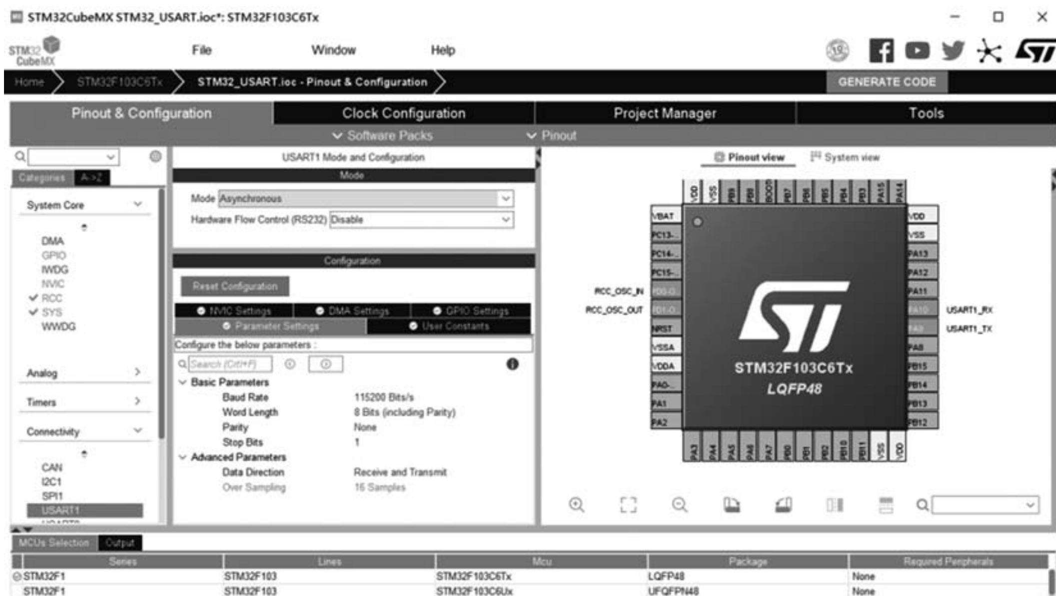

图 5-21　设置通信串口

选择 USART1 选项,然后设置串口参数,包括波特率、字长、停止位等。串口的波特率设置为默认值 115200。注意串口通信前需要将波特率设置一致,此处波特率较低是为了方便 Proteus 仿真。

设置完成后,接下来就是使能串口。这里需要用到串口中断,先要设置 NVIC 中断优先级别,再进行使能中断,由于此时只用到了串口中断,中断任务之间不存在影响,故优先级直

接采取默认设置，如图 5-22 所示。

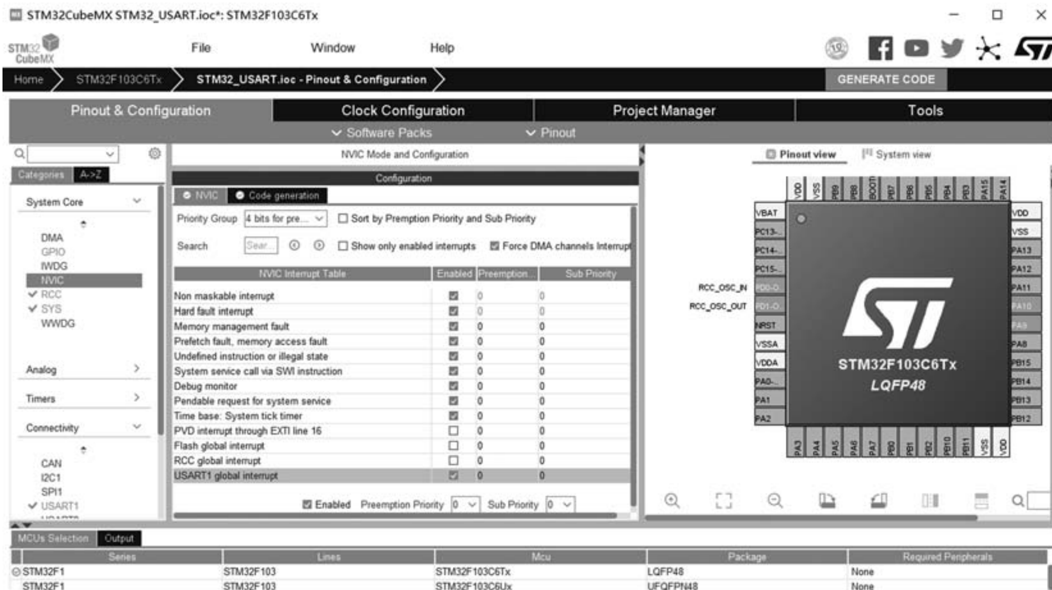

图 5-22　设置串口中断

配置完成的时钟树如图 5-23 所示。

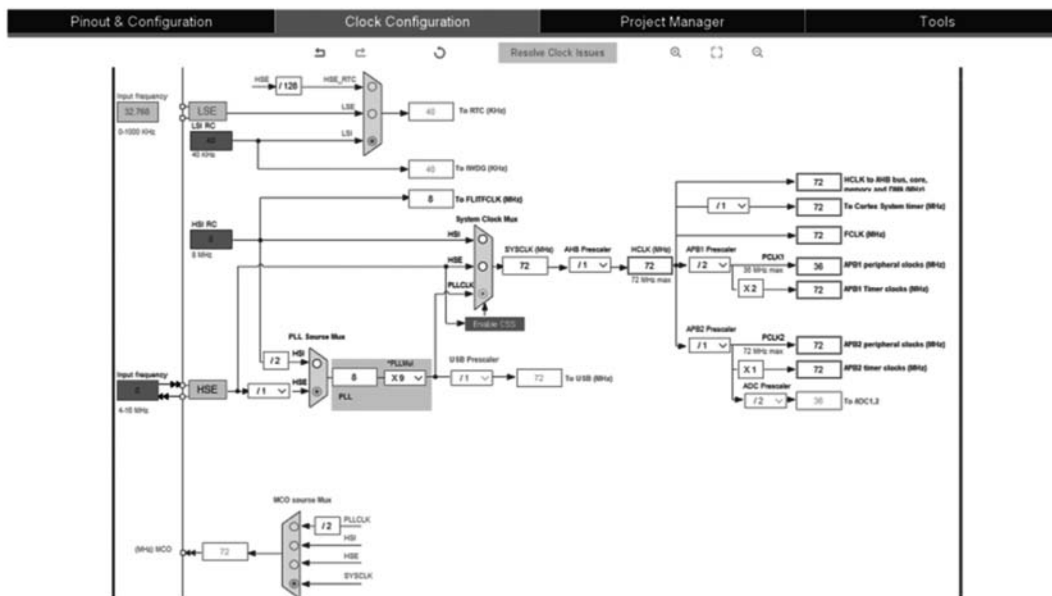

图 5-23　配置时钟树

2. Keil 软件

本节将利用串口 1 每隔 1s 发送 0xFF、0x00、0x12、0x34 和 0xDD 共 5 个数据。首先用 Keil 软件打开由 STM32CubeMX 生成的工程；然后从 Keil 左侧工程目录树的 Application/User/Core 文件夹下找到 main.c 源文件；最后打开 main.c 源文件并编写串口

发送数据代码。

在 main.c 文件中定义发送数据 pData 数组，代码如下：

```
/ * USER CODE BEGIN 1 * /
    uint8_t pData[7] = {0xFF,0x00, 0x12, 0x34, 0xDD,'\r','\n'};
/ * USER CODE END 1 * /
```

其中，\r 表示回车，英文是 linefeed，ASCII 码是 0xD。\n 表示换行，英文是 carriage return，ASCII 码是 0xA。

找到主函数，加入发送 pData 数据的代码，代码如下：

```
/ * Infinite loop * /
/ * USER CODE BEGIN WHILE * /
while (1)
{
  / * USER CODE END WHILE * /
  / * USER CODE BEGIN 3 * /
  HAL_UART_Transmit(&huart1, pData, 7100);
  HAL_Delay(1000);

}
/ * USER CODE END 3 * /
```

编译上面的代码，成功编译后开启 Debug 模式，如图 5-24 所示。

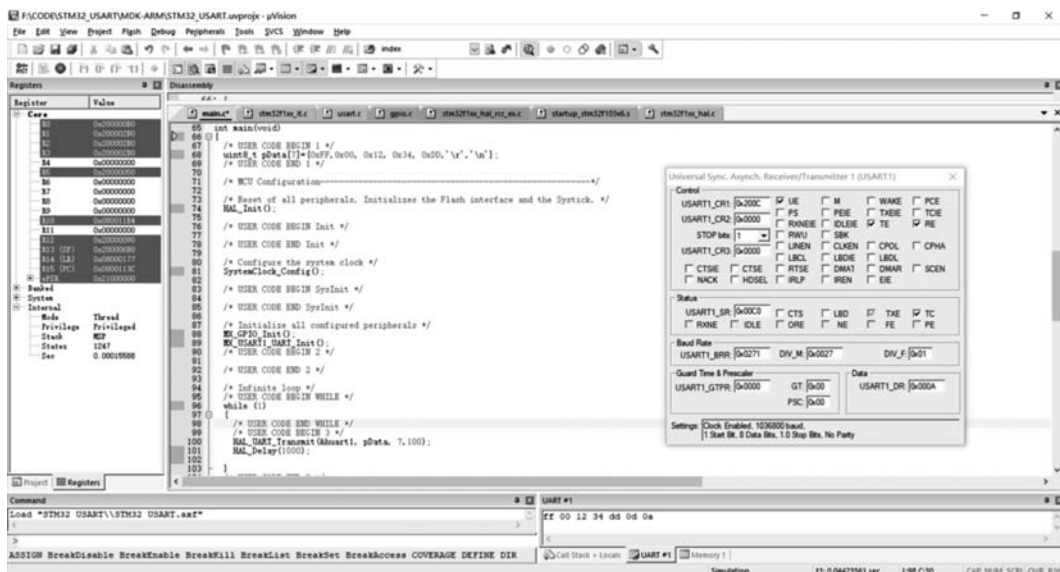

图 5-24　仿真结果

观察图中的仿真结果，可以看到串口 1 已经实现了打印 0xFF、0x00、0x12、0x34 和 0xDD 的实验结果。

3. Proteus 仿真实验

如果将编译好的程序直接烧录到芯片中，则一定要确保程序中的 GPIO 口要与实际

STM32 板的原理图相匹配。本次实验为每隔 1s 打印 0xFF、0x00、0x12、0x34 和 0xDD 数据信息。为了进一步验证上述实验的准确性，使用 Proteus 工具构建基于 STM32F103C6Tx 的串口通信仿真环境，并通过该软件模拟 STM32F103C6Tx 在物理环境的运行结果。

接下来在 Proteus 中配置 STM32F103C6，如图 5-25 所示。双击原理图上的 STM32F103C6，将上面用 Keil 编译好的串口发送程序加载到 STM32F103C6 芯片中，设置 STM32F103C6 芯片的晶振频率，其余保持默认即可，最后单击“OK”按钮，如图 5-25 所示。

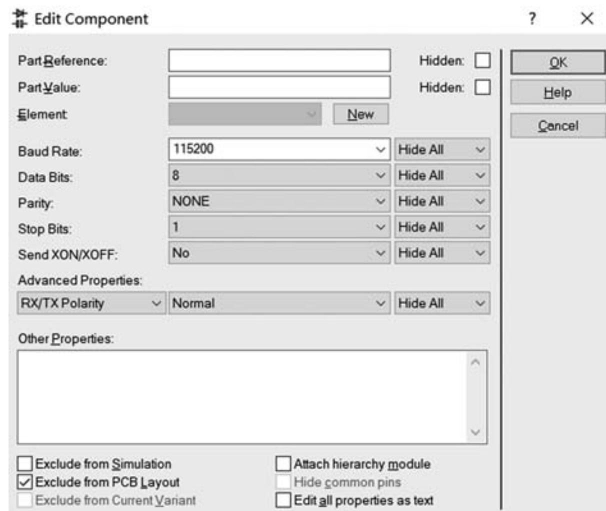

图 5-25　STM32F103C6 芯片配置

基于串口发送程序实验的原理图搭建仿真电路，并在 PA9 和 PA10 引脚上连接虚拟终端（Virtual Terminal），单击“运行仿真”或者快捷键 F12。运行仿真后能够观察到 STM32F103C6 引脚 PA9 和 PA10 的变化，仿真结果如图 5-26 所示。

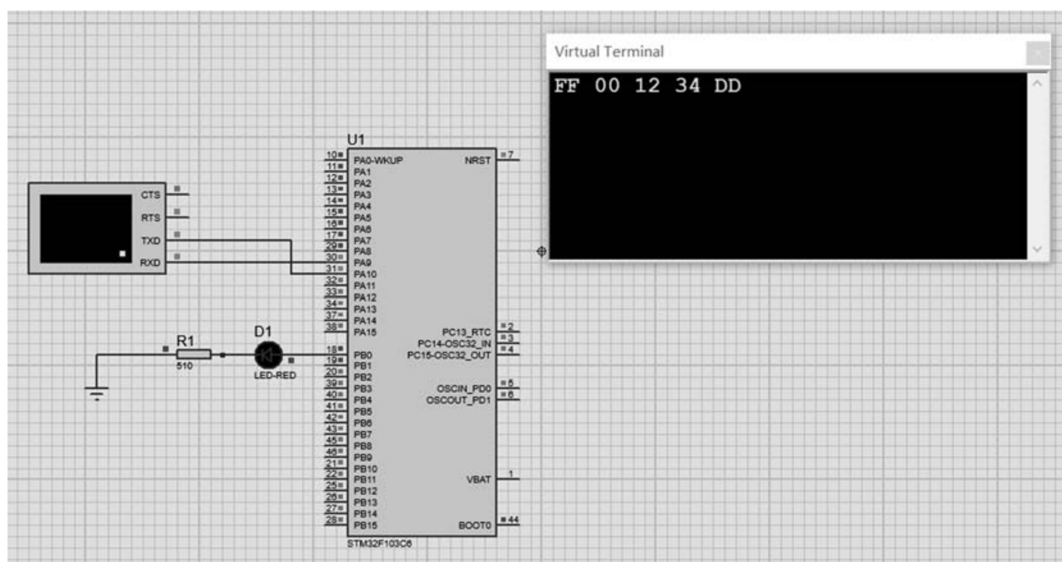

图 5-26　仿真结果

本章小结

本章主要介绍了串口的概念、通信方式、STM32 的串行接口内部结构以及如何配置 STM32 串口。通过本章的学习，要求学生掌握同步和异步串行通信，了解串行通信的数据通路形式有单工、半双工、全双工三种方式，掌握 STM32 串口的配置步骤，熟练使用 STM32CubeMX 创建串口工程，并能使用 Keil 和 Proteus 完成串口仿真调试。

习题 5

1. 填空题

(1) ________是指将一个数据块各位的数据同一时刻传送出去。

(2) ________是一个数据块的各位数据一位接一位地传送。

(3) 按照数据通信方向，可将发送数据过程分为________、________和________三种方式。

(4) ________是一种连续串行传送数据的通信方式，一次通信只传送一帧信息。

(5) STM32 的串口通信接口有________和________两种。

(6) 一个字符帧发送需要________、________和________三部分。

2. 选择题

(1) (　　)是一种常用的对异步比特流进行编码和解码的设备。

A. 中断　　B. GPIO　　C. DMA　　D. UART

(2) USART1 的时钟来源于(　　)总线时钟。

A. APB1　　B. APB2　　C. SYSCLK　　D. HCLK

(3) 下列关于波特率说法不正确的是(　　)。

A. 波特率是指在串行通信中每秒传输数据的速度

B. 波特率是指数据信号对载波的调制速率

C. 波特率指单位时间内传输的比特数

D. 异步通信中有时钟信号，所以两个通信设备之间不需要约定好波特率

(4) 下列关于同步通信描述不正确的是(　　)。

A. 同步通信需要带时钟信号传输

B. 同步通信的信息帧与异步通信中的字符帧相同

C. 采用同步通信时，将许多字符组成一个信息组

D. 每个信息帧用同步字符作为开始

(5) 下列关于 RS-232 描述不正确的是(　　)。

A. 微控制器芯片与 PC 机(或上位机)相连，除了共地之外不能直接交叉连接

B. 芯片的电平标准是+5V 表示 1，0V 表示 0

C. RS-232 的电平标准是+15/+13V 表示 0，-15/-13 表示 1

D. 微控制器芯片的串口和 RS-232 的电平标准是不一样的

(6) 串行通信传送速率的单位是波特,而波特的单位是(　　)。

A. 字节/秒　　B. 位/秒　　C. 帧/秒　　D. 字符/秒

(7) 在串行通信中,每分钟发送 60byte,其波特率为(　　)。

A. 120bps　　B. 960bps　　C. 2bps　　D. 1bps

3. 简答题

(1) 简述什么是并行通信和串行通信,及它们的优缺点。

(2) 简述什么是单工方式。

(3) 简述什么是半双工方式。

(4) 简述什么是全双工方式。

(5) 简述什么是同步通信和异步通信,及二者之间的异同点。

(6) 简述 STM32 的 UART 特点。

(7) 简述 STM32 串口通信的具体步骤。

4. 编程题

(1) 设计一个单片机的双机通信系统。工作方式 1,波特率为 1200bps,以中断方式发送和接收数据。

(2) 利用串口控制发光二极管工作,使得发光二极管每隔 0.5s 交替亮、灭,画出硬件电路图并编写程序。