title: STM32之串口DMA接收不定长数据 author: 杰杰 top: false cover: false toc: true mathjax: false date: 2019-10-05 22:11:40 img: coverImg: password: summary: tags:

- Cortex-M
- STM32
- DMA categories: STM32

STM32之串口DMA接收不定长数据

引言

在使用stm32或者其他单片机的时候，会经常使用到串口通讯，那么如何有效地接收数据呢？假如这段数据是不定长的有如何高效接收呢？

> 同学A：数据来了就会进入串口中断，在中断中读取数据就行了！

> **中断就是打断程序正常运行，怎么能保证高效呢？经常把主程序打断，主程序还要不要运行了？**

> 同学B：串口可以配置成用DMA的方式接收数据，等接收完毕就可以去读取了！

> **这个同学是对的，我们可以使用DMA去接收数据，不过DMA需要定长才能产生接收中断,如何接收不定长的数据呢？**

DMA简介

> 题外话：其实，上面的问题是很有必要思考一下的，不断思考，才能进步。

什么是DMA

DMA：全称Direct Memory Access，即直接存储器访问

DMA 传输将数据从一个地址空间复制到另外一个地址空间。CPU只需初始化DMA即可，传输动作本身是由 DMA 控制器来实现和完成。典型的例子就是移动一个外部内存的区块到芯片内部更快的内存区。这样的操作并没有让处理器参与处理，CPU可以干其他事情，当DMA传输完成的时候产生一个中断，告诉CPU我已经完成了，然后CPU知道了就可以去处理数据了，这样子提高了CPU的利用率，因为CPU是大脑，主要做数据运算的工作，而不是去搬运数据。DMA 传输对于高效能嵌入式系统算法和网络是很重要的。

在STM32的DMA资源

STM32F1系列的MCU有两个DMA控制器（DMA2只存在于大容量产品中），DMA1有7个通道，DMA2有5个通道，每个通道专门用来管理来自于一个或者多个外设对存储器的访问请求。还有一个仲裁器来协调各个DMA请求的优先权。

DMA控制器(DMA) STM32F10xxx参考手册

表59 各个通道的DMA1请求一览

外设	通道1	通道2	通道3	通道4	通道5	通道6	通道7
ADC1	ADC1						
SPI/I^2S		SPI1_RX	SPI1_TX	SPI/I2S2_RX	SPI/I2S2_TX		
USART		USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I^2C				I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1		TIM1_CH1	TIM1_CH2	TIM1_TX4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
TIM2	TIM2_CH3	TIM2_UP			TIM2_CH1		TIM2_CH2 TIM2_CH4
TIM3		TIM3_CH3	TIM3_CH4 TIM3_UP			TIM3_CH1 TIM3_TRIG	
TIM4	TIM4_CH1			TIM4_CH2	TIM4_CH3		TIM4_UP

DMA控制器(DMA) STM32F10xxx参考手册

表60 各个通道的DMA2请求一览

外设	通道1	通道2	通道3	通道4	通道5
ADC3(1)					ADC3
SPI/I2S3	SPI/I2S3_RX	SPI/I2S3_TX			
UART4			UART4_RX		UART4_TX
SDIO(1)				SDIO	
TIM5	TIM5_CH4 TIM5_TRIG	TIM5_CH3 TIM5_UP		TIM5_CH2	TIM5_CH1
TIM6/ DAC通道1			TIM6_UP/ DAC通道1		
TIM7/ DAC通道2				TIM7_UP/ DAC通道2	
TIM8(1)	TIM8_CH3 TIM8_UP	TIM8_CH4 TIM8_TRIG TIM8_COM	TIM8_CH1		TIM8_CH2

1. ADC3、SDIO和TIM8的DMA请求只在大容量的产品中存在。

而**STM32F4/F7/H7**系列的MCU有两个DMA控制器总共有16个数据流（每个DMA控制器8个），每一个DMA控制器都用于管理一个或多个外设的存储器访问请求。每个数据流总共可以有多达8个通道（或称请求）。每个通

道都有一个仲裁器，用于处理 DMA 请求间的优先级。

表 35. DMA1 请求映射

外设请求	数据流 0	数据流 1	数据流 2	数据流 3	数据流 4	数据流 5	数据流 6	数据流 7
通道 0	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
通道 1	I2C1_RX		TIM7_UP		TIM7_UP	I2C1_RX	I2C1_TX	I2C1_TX
通道 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
通道 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
通道 4	UART5_RX	USART3_RX	UART4_RX	USART3_TX	UART4_TX	USART2_RX	USART2_TX	UART5_TX
通道 5	UART8_TX(1)	UART7_TX(1)	TIM3_CH4 TIM3_UP	UART7_RX(1)	TIM3_CH1 TIM3_TRIG	TIM3_CH2	UART8_RX(1)	TIM3_CH3
通道 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2		TIM5_UP	
通道 7		TIM6_UP	I2C2_RX	I2C2_RX	USART3_TX	DAC1	DAC2	I2C2_TX

1. 这些请求仅在 STM32F42xxx 和 STM32F43xxx 上可用。

表 36. DMA2 请求映射

外设请求	数据流 0	数据流 1	数据流 2	数据流 3	数据流 4	数据流 5	数据流 6	数据流 7
通道 0	ADC1		TIM8_CH1 TIM8_CH2 TIM8_CH3		ADC1		TIM1_CH1 TIM1_CH2 TIM1_CH3	
通道 1		DCMI	ADC2	ADC2		SPI6_TX(1)	SPI6_RX(1)	DCMI
通道 2	ADC3	ADC3		SPI5_RX(1)	SPI5_TX(1)	CRYP_OUT	CRYP_IN	HASH_IN
通道 3	SPI1_RX		SPI1_RX	SPI1_TX		SPI1_TX		
通道 4	SPI4_RX(1)	SPI4_TX(1)	USART1_RX	SDIO		USART1_RX	SDIO	USART1_TX
通道 5		USART6_RX	USART6_RX	SPI4_RX(1)	SPI4_TX(1)		USART6_TX	USART6_TX
通道 6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
通道 7		TIM8_UP	TIM8_CH1	TIM8_CH2	TIM8_CH3	SPI5_RX(1)	SPI5_TX(1)	TIM8_CH4 TIM8_TRIG TIM8_COM

1. 这些请求在 STM32F42xxx 和 STM32F43xxx 上可用。

DMA接收数据

DMA在接收数据的时候，串口接收DMA在初始化的时候就处于开启状态，一直等待数据的到来，在软件上无需做任何事情，只要在初始化配置的时候设置好配置就可以了。等到接收到数据的时候，告诉CPU去处理即可。

判断数据接收完成

> 那么问题来了，怎么知道数据是否接收完成呢？

其实，有很多方法：

- 对于定长的数据，只需要判断一下数据的接收个数，就知道是否接收完成，这个很简单，暂不讨论。
- 对于不定长的数据，其实也有好几种方法，麻烦的我肯定不会介绍，有兴趣做复杂工作的同学可以在网上看看别人怎么做，下面这种方法是最简单的，充分利用了stm32的串口资源，效率也是非常之高。

DMA+串口空闲中断

这两个资源配合，简直就是天衣无缝啊，无论接收什么不定长的数据，管你数据有多少，来一个我就收一个，就像广东人吃“山竹”，来一个吃一个~（最近风好大，我好怕）。

可能很多人在学习stm32的时候，都不知道idle是啥东西，先看看stm32串口的状态寄存器：

25.6.1 状态寄存器(USART_SR)

地址偏移：0x00

复位值：0x00C0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NE	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

位4	**IDLE**：监测到总线空闲 (IDLE line detected) 当检测到总线空闲时，该位被硬件置位。如果USART_CR1中的IDLEIE为’1’，则产生中断。由软件序列清除该位(先读USART_SR，然后读USART_DR)。 0：没有检测到空闲总线； 1：检测到空闲总线。 注意：IDLE位不会再次被置高直到RXNE位被置起(即又检测到一次空闲总线)

当我们检测到触发了串口总线空闲中断的时候，我们就知道这一波数据传输完成了，然后我们就能得到这些数据，去进行处理即可。这种方法是最简单的，根本不需要我们做多的处理，只需要配置好，串口就等着数据的到来，dma也是处于工作状态的，来一个数据就自动搬运一个数据。

接收完数据时处理

串口接收完数据是要处理的，那么处理的步骤是怎么样呢？

- 暂时关闭串口接收DMA通道，有两个原因：1.防止后面又有数据接收到，产生干扰，因为此时的数据还未处理。2.DMA需要重新配置。
- 清DMA标志位。
- 从DMA寄存器中获取接收到的数据字节数（可有可无）。
- 重新设置DMA下次要接收的数据字节数，注意，数据传输数量范围为0至65535。这个寄存器只能在通道不工作(DMA_CCRx的EN=0)时写入。通道开启后该寄存器变为只读，指示剩余的待传输字节数目。寄存器内容在每次DMA传输后递减。数据传输结束后，寄存器的内容或者变为0；或者当该通道配置为自动重加载模式时，寄存器的内容将被自动重新加载为之前配置时的数值。当寄存器的内容为0时，无论通道是否开启，都不会发生任何数据传输。
- 给出信号量，发送接收到新数据标志，供前台程序查询。
- 开启DMA通道，等待下一次的数据接收，注意，对DMA的相关寄存器配置写入，如重置DMA接收数据长度，必须要在关闭DMA的条件进行，否则操作无效。

注意事项

STM32的IDLE的中断在串口无数据接收的情况下，是不会一直产生的，产生的条件是这样的，当清除IDLE标志位后，必须有接收到第一个数据后，才开始触发，一断接收的数据断流，没有接收到数据，即产生IDLE中断。

如果中断发送数据帧的速率很快，MCU来不及处理此次接收到的数据，中断又发来数据的话，这里不能开启，否则数据会被覆盖。有两种方式解决：

1. 在重新开启接收DMA通道之前，将Rx_Buf缓冲区里面的数据复制到另外一个数组中，然后再开启DMA，然后马上处理复制出来的数据。

2. 建立双缓冲，重新配置DMA_MemoryBaseAddr的缓冲区地址，那么下次接收到的数据就会保存到新的缓冲区中，不至于被覆盖。

程序实现

实验效果： 当外部给单片机发送数 据的时候，假设这帧数据长度是1000个字节，那么在单片机接收到一个字节的时候并不会产生串口中断，只是DMA在背后默默地把数据搬运到你指定的缓冲区里面。当整帧数据发送完毕之后串口才会产生一次中断，此时可以利用`DMA_GetCurrDataCounter()`函数计算出本次的数据接受长度，从而进行数据处理。

串口的配置 很简单，基本与使用串口的时候一致，只不过一般我们是打开接收缓冲区非空中断，而现在是打开空闲中断——`USART_ITConfig(DEBUG_USARTx, USART_IT_IDLE, ENABLE);` 。

```
/**
  * @brief  USART GPIO 配置,工作参数配置
  * @param  无
  * @retval 无
  */
void USART_Config(void)
{
        GPIO_InitTypeDef GPIO_InitStructure;
        USART_InitTypeDef USART_InitStructure;

        // 打开串口GPIO的时钟
        DEBUG_USART_GPIO_APBxClkCmd(DEBUG_USART_GPIO_CLK, ENABLE);

        // 打开串口外设的时钟
        DEBUG_USART_APBxClkCmd(DEBUG_USART_CLK, ENABLE);

        // 将USART Tx的GPIO配置为推挽复用模式
        GPIO_InitStructure.GPIO_Pin = DEBUG_USART_TX_GPIO_PIN;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
        GPIO_Init(DEBUG_USART_TX_GPIO_PORT, &GPIO_InitStructure);

  // 将USART Rx的GPIO配置为浮空输入模式
        GPIO_InitStructure.GPIO_Pin = DEBUG_USART_RX_GPIO_PIN;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
        GPIO_Init(DEBUG_USART_RX_GPIO_PORT, &GPIO_InitStructure);

        // 配置串口的工作参数
        // 配置波特率
        USART_InitStructure.USART_BaudRate = DEBUG_USART_BAUDRATE;
        // 配置 针数据字长
        USART_InitStructure.USART_WordLength = USART_WordLength_8b;
        // 配置停止位
```

```
        USART_InitStructure.USART_StopBits = USART_StopBits_1;
        // 配置校验位
        USART_InitStructure.USART_Parity = USART_Parity_No ;
        // 配置硬件流控制
        USART_InitStructure.USART_HardwareFlowControl =
        USART_HardwareFlowControl_None;
        // 配置工作模式，收发一起
        USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
        // 完成串口的初始化配置
        USART_Init(DEBUG_USARTx, &USART_InitStructure);
        // 串口中断优先级配置
        NVIC_Configuration();

#if USE_USART_DMA_RX
        // 开启 串口空闲IDEL 中断
        USART_ITConfig(DEBUG_USARTx, USART_IT_IDLE, ENABLE);
  // 开启串口DMA接收
        USART_DMACmd(DEBUG_USARTx, USART_DMAReq_Rx, ENABLE);
        /* 使能串口DMA */
        USARTx_DMA_Rx_Config();
#else
        // 使能串口接收中断
        USART_ITConfig(DEBUG_USARTx, USART_IT_RXNE, ENABLE);
#endif

#if USE_USART_DMA_TX
        // 开启串口DMA发送
//      USART_DMACmd(DEBUG_USARTx, USART_DMAReq_Tx, ENABLE);
        USARTx_DMA_Tx_Config();
#endif

        // 使能串口
        USART_Cmd(DEBUG_USARTx, ENABLE);
}
```

串口DMA配置

把DMA配置完成，就可以直接打开DMA了，让它处于工作状态，当有数据的时候就能直接搬运了。

```
#if USE_USART_DMA_RX

static void USARTx_DMA_Rx_Config(void)
{
        DMA_InitTypeDef DMA_InitStructure;

        // 开启DMA时钟
        RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
        // 设置DMA源地址：串口数据寄存器地址*/
        DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)USART_DR_ADDRESS;
        // 内存地址(要传输的变量的指针)
        DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)Usart_Rx_Buf;
        // 方向：从内存到外设
```

```
        DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;
        // 传输大小
        DMA_InitStructure.DMA_BufferSize = USART_RX_BUFF_SIZE;
        // 外设地址不增
        DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
        // 内存地址自增
        DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
        // 外设数据单位
        DMA_InitStructure.DMA_PeripheralDataSize =
        DMA_PeripheralDataSize_Byte;
        // 内存数据单位
        DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
        // DMA模式，一次或者循环模式
        //DMA_InitStructure.DMA_Mode = DMA_Mode_Normal ;
        DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
        // 优先级：中
        DMA_InitStructure.DMA_Priority = DMA_Priority_VeryHigh;
        // 禁止内存到内存的传输
        DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
        // 配置DMA通道
        DMA_Init(USART_RX_DMA_CHANNEL, &DMA_InitStructure);
        // 清除DMA所有标志
        DMA_ClearFlag(DMA1_FLAG_TC5);
        DMA_ITConfig(USART_RX_DMA_CHANNEL, DMA_IT_TE, ENABLE);
        // 使能DMA
        DMA_Cmd (USART_RX_DMA_CHANNEL,ENABLE);
}
#endif
```

接收完数据处理

因为接收完数据之后，会产生一个idle中断，也就是空闲中断，那么我们就可以在中断服务函数中知道已经接收完了，就可以处理数据了，但是中断服务函数的上下文环境是中断，所以，尽量是快进快出，一般在中断中将一些标志置位，供前台查询。在中断中先判断我们的产生在中断的类型是不是idle中断，如果是则进行下一步，否则就无需理会。

```
/**
  *****************************************************************
  * @brief   串口中断服务函数
  * @author  jiejie
  * @version V1.0
  * @date    2018-xx-xx
  *****************************************************************
  */
void DEBUG_USART_IRQHandler(void)
{
#if USE_USART_DMA_RX
        /* 使用串口DMA */
        if(USART_GetITStatus(DEBUG_USARTx,USART_IT_IDLE)!=RESET)
        {
                /* 接收数据 */
```

```
                Receive_DataPack();
                // 清除空闲中断标志位
                USART_ReceiveData( DEBUG_USARTx );
        }
#else
  /* 接收中断 */
        if(USART_GetITStatus(DEBUG_USARTx,USART_IT_RXNE)!=RESET)
        {
    Receive_DataPack();
        }
#endif
}
```

Receive_DataPack()

这个才是真正的接收数据处理函数，为什么我要将这个函数单独封装起来呢？因为这个函数其实是很重要的，因为我的代码兼容普通串口接收与空闲中断，不一样的接收类型其处理也不一样，所以直接封装起来更好，在源码中通过宏定义实现选择接收的方式！更考虑了兼容操作系统的，可能我会在系统中使用dma+空闲中断，所以，供前台查询的信号量就有可能不一样，可能需要修改，我就把它封装起来了。不过无所谓，都是一样的。

```
/**********************************************************
  * @brief   Uart_DMA_Rx_Data
  * @param   NULL
  * @return  NULL
  * @author  jiejie
  * @github  https://github.com/jiejieTop
  * @date    2018-xx-xx
  * @version v1.0
  * @note    使用串口 DMA 接收时调用的函数
  **********************************************************/
#if USE_USART_DMA_RX
void Receive_DataPack(void)
{
        /* 接收的数据长度 */
        uint32_t buff_length;

        /* 关闭DMA ，防止干扰 */
        DMA_Cmd(USART_RX_DMA_CHANNEL, DISABLE);  /* 暂时关闭dma，数据尚未处理 */

        /* 清DMA标志位 */
        DMA_ClearFlag( DMA1_FLAG_TC5 );

        /* 获取接收到的数据长度 单位为字节*/
        buff_length = USART_RX_BUFF_SIZE -
DMA_GetCurrDataCounter(USART_RX_DMA_CHANNEL);

    /* 获取数据长度 */
    Usart_Rx_Sta = buff_length;

        PRINT_DEBUG("buff_length = %d\n ",buff_length);
```

```
        /* 重新赋值计数值，必须大于等于最大可能接收到的数据帧数目 */
        USART_RX_DMA_CHANNEL->CNDTR = USART_RX_BUFF_SIZE;

        /* 此处应该在处理完数据再打开，如在 DataPack_Process() 打开*/
        DMA_Cmd(USART_RX_DMA_CHANNEL, ENABLE);

        /* (OS)给出信号 ，发送接收到新数据标志，供前台程序查询 */

    /* 标记接收完成，在 DataPack_Handle 处理*/
    Usart_Rx_Sta |= 0xC000;

    /*
    DMA 开启，等待数据。注意，如果中断发送数据帧的速率很快，MCU来不及处理此次接收到的数据，
    中断又发来数据的话，这里不能开启，否则数据会被覆盖。有2种方式解决：

    1. 在重新开启接收DMA通道之前，将Rx_Buf缓冲区里面的数据复制到另外一个数组中，
    然后再开启DMA，然后马上处理复制出来的数据。

    2. 建立双缓冲，重新配置DMA_MemoryBaseAddr的缓冲区地址，那么下次接收到的数据就会
    保存到新的缓冲区中，不至于被覆盖。
        */
}
```

f1使用dma是非常简单的，我在f4用dma的时候也遇到一些问题，最后看手册解决了，打算下一篇文章就写一下调试过程，没有什么是debug不能解决的，如果有，那就两次。今天台风天气，连着舍友的WiFi更新的文章~中国电信还是强，台风天气信号一点都不虚，我的移动卡一动不动-_-.

喜欢就关注我吧！

相关代码可以在公众号后台回复获取。