主设备时钟频率=PCLK/预分频器值，LRCK频率=主设备时钟频率/CODECLK，串行位时钟频率类型=CODECLK的采用类型/串行数据位数，CODECLK即过采样率

通常情况下，UART设备以中断或者查询的形式来传输或者接收数据，是通过MCU控制UART的内部FIFO来实现的。但不论是以中断还是以查询的形式，传输过程中总是会占用到MCU的时间。这样，在传输大数据量时，如音频数据、升级固件数据时，串口数据包较大，UART设备的发送过程会触发大量中断，MCU需要从来源把每一片段的资料复制到暂存器，然后把它们再次写回到新的地方。在这个时间中，MCU对于其他的工作来说就无法使用，其中大多数时间可能是在等待数据包传输完成从而影响系统的运行速率，并且还可能出现数据丢失或者错误，从而导致数据传输的效率较低。

在常规的UART串行数据通信的基础上，结合DMA控制器，允许不同速度的硬件装置来沟通，而不需要依赖于 CPU 的大量中断负载，实现将UART串口数据包通过DMA通道搬运。

通过设计DMA链表缓存，将发送数据进行链路层、应用层两次打包，通过查询方式检测DMA传输数据量，并对链表缓存数据进行完整数据包解析，提高了数据传输的效率以及准确率，提升了MCU的利用率。

初始化过程一般包括以下具体操作：

使能UART时钟，设置UART波特率，设置UART的触发深度，设置UART的发送接收为DMA方式；

配置DMA相关参数，包括DMA传输的数据存储速度，数据存储地址等；

/\*配置UART使用DMA模式，然后DMA使用链表，向0x70480000起始的BUF传输数据\*//\*DMA配置使用CHANNEL1，因为CHANNEL0被SPI FLASH使用\*//\*创建一个循环链表，每个链表传输4KB字节数据\*//\*起始地址为0x70480000 \*/在DMA设备中定义链表缓存，将缓存分成M个大小相等的链表缓存块，并将这些链表缓存块组成一个环形缓存链，初始化链表缓存块；将环形缓存链中的数据源地址和目标地址作为DMA传输的数据源地址、数据目标地址，并将第一个链表缓存块链接的下一个链表缓存块作为其后续模块，以此类推设置所有链表缓存块；

数据接收过程中，数据处理进程每10ms检测缓存区数据更新状态，通过读取DMA寄存器数据和DMA中断触发次数，可以计算出目前缓存区数据大小，如果数据量比上次检测有增加，则进行数据解析。

(方向是wifi->mcu)

for(i = 0; i < 19; i++)

{

dma\_lli[i].SrcAddr = 0x64000000; //UART1 DMA接口地址

dma\_lli[i].DestAddr = 0x70480000 + i\*4\*1024;

dma\_lli[i].NextLLI = (unsigned int)&dma\_lli[i+1];

dma\_lli[i].Control = lli\_control\_word;

}

/\*循环链表\*/

dma\_lli[19].SrcAddr = 0x64000000;

dma\_lli[19].DestAddr = 0x70480000 + 19\*4\*1024;

dma\_lli[19].NextLLI = (unsigned int)&dma\_lli[0];

dma\_lli[19].Control = lli\_control\_word;

………………….

DMAC\_ChannelLLIControl(DMACChannel1,dma\_lli[0].Control);

DMAC\_ChannelSoureAddr(DMACChannel1,dma\_lli[0].SrcAddr);

DMAC\_ChannelDestAddr(DMACChannel1,dma\_lli[0].DestAddr);

DMAC\_ChannelLLI(DMACChannel1,(unsigned int)dma\_lli[0].NextLLI,DMAC\_AHBMaster2);

设置DMA传输中每传输K 大小的数据触发一次中断，根据中断触发次数可以计算缓存块存储的数据大小;(4x1024)

DMAC\_ChannelSourceConfig(DMACChannel1,NOINCREMENT,DMAC\_AHBMaster2,TRANSFERWIDTH\_32b,BURSTSIZE4); DMAC\_ChannelDestConfig(DMACChannel1,INCREMENT,DMAC\_AHBMaster2,TRANSFERWIDTH\_32b,BURSTSIZE4);

DMAC\_ChannelTransferSize(DMACChannel1,1024);

所述K可以整除L并小于L ，所述L为链表缓存块的存储空间大小。这里每次中断触发表示完成一次搬运,根据中断次数可以计算缓存块存储的数据大小。

每搬运L/M次数据切换一个链表缓存块。

在mcu和WIFI模块UART 端口之间设定DMA传输通道，两个模块的UART都配置为DMA工作模式，为适应单声道语音采样，采用波特率为961200bps，理论传输速度为112.5K/S,足够实时传输16K采样率单声道语音数据。将UART接口接收的数据存入链表缓存，链表缓存总大小为80K，由20个大小为4K的数据缓存块链接组成。设置DMA搬运每1K数据触发一次中断，设定每传输4K数据切换一个4K大小的数据缓存块。

通过读取DMA寄存器数据和DMA中断触发次数，可以计算出目前缓存区数据大小，如果数据量比上次检测有增加，则进行数据解析。其中DMA寄存器是用于临时存储中断之前的存储数据,每中断一次,数据都会存入缓存块中进行数据解码时，分别使用两个指针记录缓存区数据起始和结束位置，每当缓存区数据增加时，计算起始和结束位置之间的数据量，可以将数据从起点位置开始进行链路层数据解析，其中起点位置会每次向后移动一个字节，直到解析出一个完整数据包或者到达结束位置时停止后移，当解析出完整数据包后，还原数据中的字符，然后提取数据，并更新释放缓存区。

\* @功能:DMA中断服务程序

\* @注意:无

\* @参数:无

\* @返回值:无

\*/

void DMA\_IRQHandler(void)

{

DMAC->DMACIntErrClr = 0xff;

//g\_dma\_translate\_ok = 1;

if(DMAC\_IntTCStatus(DMACChannel0))

{

g\_dma\_translate\_ok = 1;

}else if(DMAC\_IntTCStatus(DMACChannel1))//wifi->mcu

{

g\_dma\_translate\_ok1 = 1;

dma\_recv\_interrupt ++;

}else if(DMAC\_IntTCStatus(DMACChannel2))//mcu->wifi

{

g\_dma\_translate\_ok2 = 0;

}

DMAC->DMACIntTCClear = 0xff;

}

接收函数：

/\*获取DMA已经传输的UART数据大小\*/

trans\_byte\_count = \*(volatile unsigned int \*)(0x40011000 + 0x12c); /\*通道1\*/

trans\_byte\_count = 1024 - (trans\_byte\_count & 0xfff);

/\*有几次DMA中断，即表示完成几次\*/

current\_size = 1024 \* dma\_recv\_interrupt + trans\_byte\_count;

if(current\_size > last\_size) /\*如果当前数据比上次大，说明接收到新数据\*/

{

delta\_size = current\_size - last\_size;

last\_size = current\_size;

delta\_size \*= 4; /\*计算出这次接收的数据长度\*/

if(delta\_size <= (80\*1024 - buf\_count))

{

memcpy(pbuf\_write,(void \*)(UART\_BUF\_START\_ADDR + buf\_count),delta\_size); /\*注意此pbuf\_write buf最大到80KB\*/

buf\_count += delta\_size;

pbuf\_write += delta\_size;

}

else

{

memcpy(pbuf\_write,(void \*)(UART\_BUF\_START\_ADDR + buf\_count),80\*1024 - buf\_count);

pbuf\_write += (80\*1024 - buf\_count);

memcpy(pbuf\_write,(void \*)UART\_BUF\_START\_ADDR,delta\_size - (80\*1024 - buf\_count));

buf\_count = delta\_size - (80\*1024 - buf\_count);

pbuf\_write += buf\_count;

dma\_recv\_interrupt -= 20;

last\_size -= 20\*1024;

}

}

loop\_size = pbuf\_write - pbuf\_read;

if((loop\_size >= 0) && (loop\_size <= 3072))

{

loop\_count = loop\_size;

}

else

{

loop\_count = 3072;

}

链路层打包完成后，传给UART传输接口，随后系统检测DMA是否处于空闲状态，若未处于空闲状态则定时查询状态，如果处于传输状态则等待正在传输的数据传输完成，检测到数据传输完成,DMA通道为空闲状态后，由传输进程配置UART发送接口，并根据前述返回打包后的数据长度设置本次数据发送的大小以及数据所在地址，发出DMA使能信号,开始发送数据。数据发送，将打包好的数据交付于数据传输进程，传输进程检测当前DMA工作状态，如果处于正在传输状态则等待正在传输的数据传输完成，检测到DMA为空闲状态后，由传输进程配置UART发送接口，设置本次数据发送的大小以及数据所在地址，使能DMA发送数据。

….

while (1)

{

if (g\_dma\_translate\_ok2 == 0)

{

break; /\*dma传输处于空闲状态\*/

}

else

{

vTaskDelay(pdMS\_TO\_TICKS(1)); /\*延迟20ms,该时间可能需要根据实际情况调整\*/

}

}

g\_dma\_translate\_ok2 = 1;

……………….

**名词解释：**

<http://www.openedv.com/forum.php?mod=viewthread&tid=285474&extra=page=1>

如果是增量突发传输，就是一次性就传输4、8、16个数据，其间不被中断  
因为要求不被中断也就会产生一个问题，总线给你占用了，其他组件就没得用了

**lli\_ctrl.DestTransferWidth = TRANSFERWIDTH\_32b;**

**lli\_ctrl.SrcTransferWidth = TRANSFERWIDTH\_32b;**

**lli\_ctrl.DestBurstSize = BURSTSIZE4;**

**lli\_ctrl.SrcBurstSize = BURSTSIZE4;**

**lli\_ctrl.TransferSize = 1024;**

**TransferWidth:是指传输位数，这里指32位既4个字节；**

**TransferSize:是指一次传几个tansferwidth,这里指1024个width既4k字节；**

**Burstsize：数据分几次传输，这里4k字节分4次传输，每次传输1k就产生一个中断。**

**uart中断模式配置：**

1、NVIC配置uart\_irq

NVIC\_InitStruct.NVIC\_IRQChannel = (UARTx == UART0)?UART0\_IRQn:UART1\_IRQn;

NVIC\_InitStruct.NVIC\_IRQChannelPreemptionPriority = 1;

NVIC\_InitStruct.NVIC\_IRQChannelSubPriority = 0;

NVIC\_InitStruct.NVIC\_IRQChannelCmd = ENABLE;

NVIC\_Init(&NVIC\_InitStruct);

2、设置Uart时钟开关、配置管脚复用

2、uart使能关闭

3、uart信号发送和接收使能控制

4、uart波特率配置

5、设置uart发送FIFO数据位宽（以字节或者word方式，这里是字节）

6、UART数据位、停止位、奇偶校验位设置

7、UART接收和发送FIFO触发深度配置

**uart 查询模式配置：**

1、设置Uart时钟开关、配置管脚复用

2、uart使能关闭

3、uart信号发送和接收使能控制

4、uart波特率配置

10、清除uart中断标志

11、uart中断屏蔽使能

5、设置uart发送FIFO数据位宽（以字节或者word方式，这里是字节）

6、UART数据位、停止位、奇偶校验位设置

7、UART接收和发送FIFO触发深度配置

10、uart fifo清除

11、uart nced信号使能

12、uart中断发送位屏蔽使能、接收位不屏蔽

12、uart使能

**UART DMA模式配置：**

1、NVIC配置为DMA\_IRQ

2、使能DMA时钟和Uart时钟开关

3、配置管脚复用

4、uart使能关闭

5、uart信号发送和接收使能控制

UART\_CRConfig(UARTx,UART\_TXE,ENABLE);

UART\_CRConfig(UARTx,UART\_RXE,ENABLE);

6、uart波特率配置（921600）

7、设置uart发送FIFO数据位宽（以字节或者word方式，这里是字）

8、UART数据位、停止位、奇偶校验位设置

9、UART接收和发送FIFO触发深度配置

10、清除uart中断标志

11、uart中断屏蔽使能

12、uart fifo清除

13、uart nced信号使能

14、uart 接收发送dma控制使能

15、uart使能

**DMA配置：**

1、NVIC配置为DMA\_IRQ

NVIC\_InitStruct.NVIC\_IRQChannel = DMA\_IRQn;

NVIC\_InitStruct.NVIC\_IRQChannelPreemptionPriority = 6;

NVIC\_InitStruct.NVIC\_IRQChannelSubPriority = 0;

NVIC\_InitStruct.NVIC\_IRQChannelCmd = ENABLE;

NVIC\_Init(&NVIC\_InitStruct);

2、使能DMA时钟开关

3、禁止DMA通道传输并且打开DMA通道传输Power

4、清除DMA通道传输完成中断状态

5、清除DMA通道传输错误中断状态

6、DMA控制器AHB MASTER字节端序的配置（大端或小端）

7、DMA控制器使能

8、DMA通道源地址和目标地址设置

9、通道保护位设置（用户模式or特权模式；缓冲是否使能；缓存是否使能）

10、通道源的相关配置（源地址不增长or增长，源搬运使用的DMA接口；元传输数据的位宽、源突发传输的大小）

11、通道目标的相关配置（源地址不增长or增长，源搬运使用的DMA接口；元传输数据的位宽、源突发传输的大小）

12、DMA通道传输大小配置

13、关闭DMA通道传输中的halt

14、DMA通道中断屏蔽配置（传输完成中断不屏蔽，传输错误中断屏蔽）

15、其他设置（流控类型选择：是从内存到外设还是外设到外设还是内存到内存，目标外设、元外设）

16、DMA传输不锁

17、DMA通道传输使能

**i2s DMA配置：**

1、

NVIC\_InitTypeDef NVIC\_InitStruct = {0};

NVIC\_InitStruct.NVIC\_IRQChannel = IIS\_DMA\_IRQn;

NVIC\_InitStruct.NVIC\_IRQChannelPreemptionPriority = 2;

NVIC\_InitStruct.NVIC\_IRQChannelSubPriority = 0;

NVIC\_InitStruct.NVIC\_IRQChannelCmd = ENABLE;

NVIC\_Init(&NVIC\_InitStruct);

2、配置外设复位，配合Scu\_Setdevice\_ResetRelease 使用，先reset,然后release，外设复位完成

Scu\_Setdevice\_Reset((unsigned int)IIS\_DMA\_BASE);

Scu\_Setdevice\_ResetRelease((unsigned int)IIS\_DMA\_BASE);

IIS\_DMA\_RXInit\_Typedef IISDMARX\_Init\_Struct = {0};

Scu\_Setdevice\_Reset((unsigned int)IIS0);

Scu\_Setdevice\_ResetRelease((unsigned int)IIS0);

3、设置时钟开关

Scu\_SetDeviceGate((unsigned int)IIS\_DMA\_BASE,ENABLE);

Scu\_SetDeviceGate((unsigned int)IIS0,ENABLE);

4、:配置管脚复用对应功能

Scu\_SetIOReuse(I2S0\_SCLK\_PAD,FIRST\_FUNCTION);

Scu\_SetIOReuse(I2S0\_LRCLK\_PAD,FIRST\_FUNCTION);

Scu\_SetIOReuse(I2S\_MCLK\_PAD,FIRST\_FUNCTION);

Scu\_SetIOReuse(I2S0\_SDI\_PAD,FIRST\_FUNCTION);

5、配置i2sDMA接配置

IISDMARX\_Init\_Struct.rxaddr = 0x704a0000;//0x704a0000;//0x20020000;

IISDMARX\_Init\_Struct.rxinterruptsize = IISDMA\_RX32Interrupt;//IISDMA\_RX32Interrupt;

IISDMARX\_Init\_Struct.rollbackaddrsize = IISDMA\_RXTX1024RollbackADDR;//IISDMA\_RXTX1024RollbackADDR;

IISDMARX\_Init\_Struct.rxsinglesize = IISDMA\_TXRXSINGALESIZE32bytes;

IISDMARX\_Init\_Struct.rxdatabitwide = IIS\_TXRXDATAWIDE16BIT;

IISDMARX\_Init\_Struct.sck\_lrck = IIS\_BUSSCK\_LRCK64;

IISDMARX\_Init\_Struct.rxdatafmt = IIS\_RXDATAFMT\_LEFT\_JUSTIFIED;//左对齐

// IISDMARX\_Init\_Struct.rxdatafmt = IIS\_RXDATAFMT\_IIS;//标准模式

IISDMARX\_Init\_Struct.oversample = IIS\_OverSample256Fs;

IIS\_RXMODEConfig(IIS0,IIS\_MERGE\_NONE,IIS\_LR\_LEFT\_HIGH\_RIGHT\_LOW,IIS\_RXMODE\_MONO);

IISx\_RXInit(IIS0,&IISDMARX\_Init\_Struct);

IISDMA\_ADDRRollBackINT(IIS0DMA,IISxDMA\_RX\_EN,ENABLE);

\*(volatile unsigned int \*)(0x40015000 + 0x0) |= ((0x1 << 0) | (0x1 << 6));

第二代，详细看代码：

I2S1

//!block size（发送多少字节的数据来一次中断，在中断中需填写下一帧发送的数据的起始地址）

int32\_t block\_size;=1024

//!左通道是原始数据，右通道是处理之后的结果

SCK和LRCK的比值为64

\*IIS1输出前处理结果的block数目\*/=10

不调试送入FE计算的PCM数据

/\*\*

\* @功能:设置外设时钟分频

\* 更改分频系数的时候：

\* 关闭外设gate；

\* 需要先将divider复位；

\* 分频参数配置好；

\* 然后使能配置的分频参数有效；

\* 解除divider的复位。

\* @参数:device\_base，需要设置的外设基址 div\_num，分频输

\* @返回值: PARA\_ERROR: 参数错误 RETURN\_OK, 配置完成

\*/

//!IISDMA传输地址回卷中断产生，需要IISDMA搬运64次

//!单次搬运16byte的数据

i2s3

//!每帧数据大小block\_size为256\*2=512

@brief CODEC作为ADC的模式选择，//!数据窗口宽度为32bit

//!输入为单端输入模式

//!MIC增益为13dBINNER\_CODEC\_MIC\_AMP\_13dB = 2,

//!不使能ALC的情况下，PGA R 的固定增益=5

\* @brief 使能CODEC中的ADC:

\* 1.使用此函数之前先初始化CODEC\_ADC\_Config\_TypeDef结构体并填写参数；

\* 2.ALCL\_Gain和ALCR\_Gain的上下限分别是28.5dB和-18dB，步长1.5。

//!ALC噪声门限45dB

197 + 14; //数字增益

//!ALC的判断信号来自高通滤波器之后（同时也是数字增益之后）

//为接收语音的buf分配空间,不能使用malloc的时候，blocksize只能固定，用户不能设置（大小为byte）单通道512双通道1024，8个blocksize

?????//!IISDMA传输地址回卷中断产生，需要IISDMA搬运32次

?????//!IISDMA搬运4次之后，产生传输完成中断

//!单次搬运128byte的数据

16k\*16bit\*1=>1ms 传输32byte数据=>传输512byte需要16ms