泰凌驱动

SDK 开发手册

AN-21010601-C5

Ver1.1.1

2023.03.17

Keyword

驱动，debug，中断，Clock，协议

Brief

本文为泰凌驱动 SDK 开发手册。介绍了泰凌驱动 SDK 的结构、机制和各模块细节。

Published by
Telink Semiconductor

Bldg 3, 1500 Zuchongzhi Rd,
Zhangjiang Hi-Tech Park, Shanghai, China

Legal Disclaimer

This document is provided as-is. Telink Semiconductor reserves the right to make improvements without further notice to this document or any products herein. This document may contain technical inaccuracies or typographical errors. Telink Semiconductor disclaims any and all liability for any errors, inaccuracies or incompleteness contained herein.

Information

For further information on the technology, product and business term, please contact Telink Semiconductor Company www.telink-semi.com

For sales or technical support, please send email to the address of:

telinksales@telink-semi.com

telinksupport@telink-semi.com

修订历史

版本	主要改动
V1.0.0	初始版本
V1.0.1	添加了 USB 内容
V1.0.2	添加了 CPU 性能测试内容
V1.1.0	更新 ADC 章节，增加 Audio 章节
V1.1.1	修改部分描述

Contents

1 驱动目录结构

Driver SDK 目录结构如下：

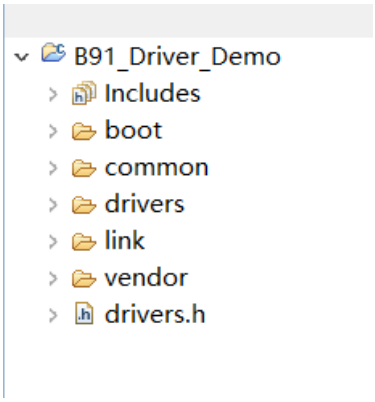


Figure 1.1: Driver SDK 目录

1.1 boot

该文件夹下是启动文件。

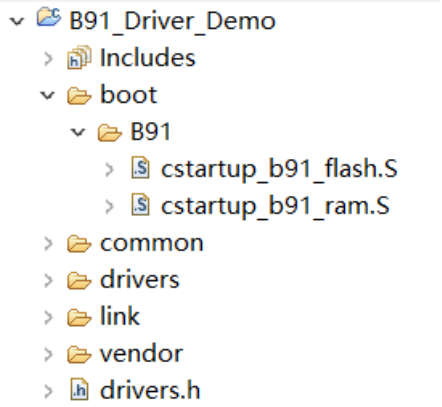


Figure 1.2: boot 文件夹

1.2 common

该文件夹下是一些和驱动无关的共用文件。其中特殊说明两个文件夹：

bt_debug：是 bt 相关模块设置 debug GPIO 的接口函数。

compatibility_pack：是为了兼容以前的各个 sdk 的驱动接口而添加的相关文件，驱动中不会使用。

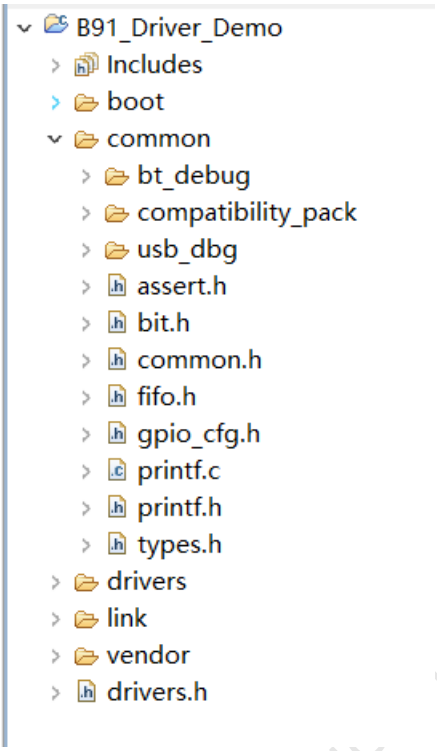


Figure 1.3: common 文件夹

1.3 drivers

该文件夹下即是相关模块的驱动。

14 link

该文件夹用于存放 link 文件，根据不同的使用需求进行选择。

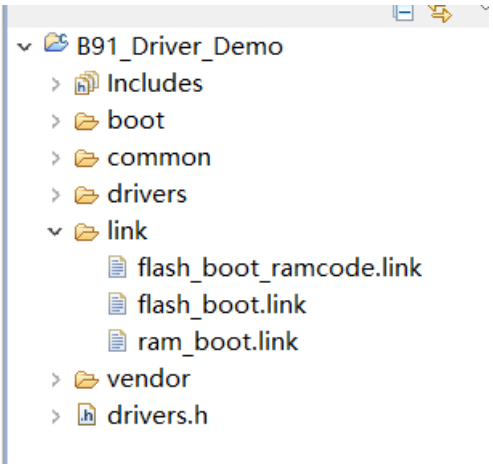


Figure 1.4: link 文件夹

1.5 vendor

该文件夹下是所有模块的 demo。

2 启动机制

2.1 Telink Platform SoC

a) 芯片上电/deep 回来：会从 Flash 中先搬一段程序到 RAM 中，然后从 RAM 启动。

b) retention 回来后直接从 RAM 启动。

可以看到，这里的启动位置是一样的，都是 RAM。

2.2 Risc-V Platform SoC

a) 芯片上电/deep 回来：不会从 Flash 中搬移代码到 RAM 中，而是跳转到 Flash 的起始地址（0x20000000）开始执行。（可以这样处理的原因是该系列的芯片支持直接从 Flash 取指执行。）

b) retention 回来会从 IRAM 启动。

注意：

- 有两个 RAM，一个是 IRAM 一个是 DRAM，IRAM 可以放程序和数据，DRAM 只能放数据。）

可以看到，这里的启动位置是不一样的，所以在 link 和 S 文件中可以看到有两个段：vectors 和 retention_reset 段。vector 段是在 flash 起始地址，retention 段是在 IRAM 的起始地，他们都是启动代码部分，代码部分会有不同的处理，后面会详细说明。

3 S 和 link 文件

3.1 使用组合

根据不同的使用场景，我们可以选择对应的组合方式。

使用场景	S 文件	link 文件
通用的从 flash 启动程序	cstartup_b91_flash.S	flash_boot.link
测试性能（coremark 和 dhrystone）时使用	cstartup_b91_flash.S	flash_boot_ramcode.link
用于 load 到 ram 中启动的程序，不能 load 到 flash 中执行	cstartup_b91_ram.S	ram_boot.link

其中 S 文件的区别如下：

S 文件	区别
cstartup_b91_flash.S	是从 flash 启动的程序需要用到的 S 文件
cstartup_b91_ram.S	是从 ram 启动的程序需要用到的 S 文件

link 文件的区别：

link 文件	区别
flash_boot.link	正常使用场景使用的
flash_boot_ramcode.link	和 flash_boot.link 文件的唯一区别是，它将除了启动需要的 vector 段，其他段都放到了 ramcode 段，所有的程序都在 ram 中跑，这样执行时间快。
ram_boot.link	该文件是用于 load 到 ram 中启动的程序，不能 load 到 flash 中执行，该 link 文件配合 cstartup_b91_ram.S 进行使用。

3.2 配置方法

3.2.1 Link

可以通过如下配置选择需要使用的 link 文件。

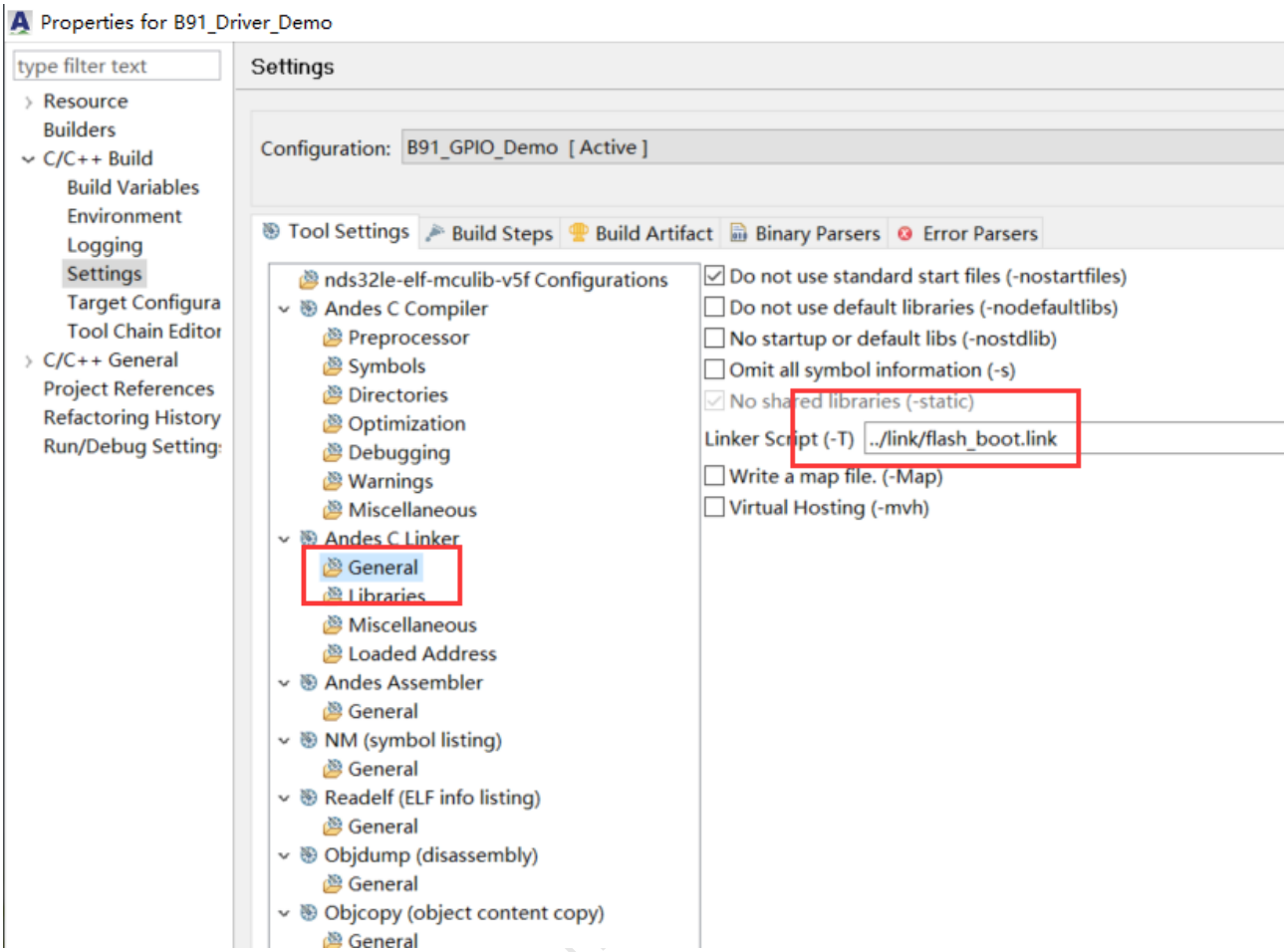


Figure 3.1: link 文件配置

3.2.2 S **文件**

S 文件是通过如下宏定义进行区分选择的。

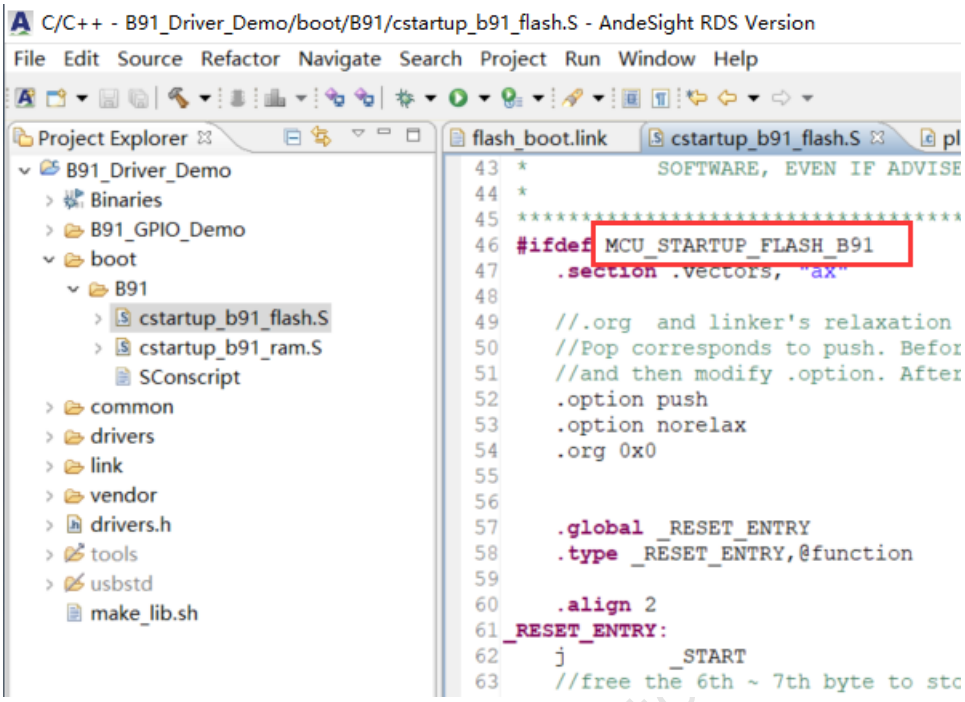


Figure 3.2: 区分 S 文件的宏定义

可以通过如下方法定义宏，决定选择使用哪个 S 文件。

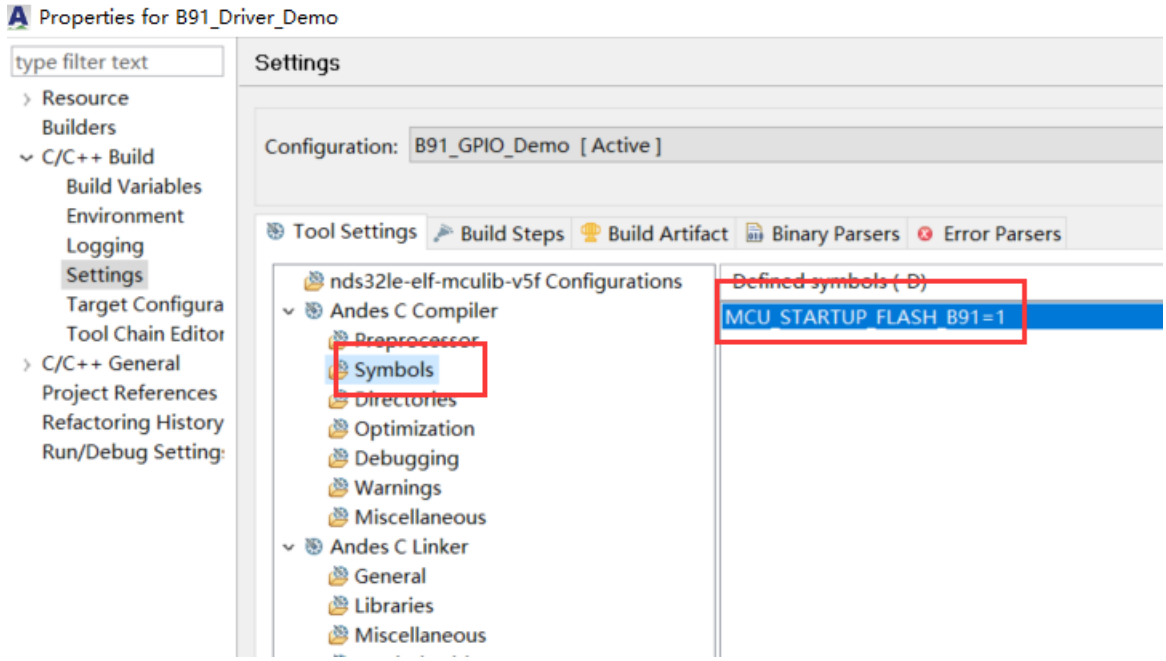


Figure 3.3: 选择 S 文件的宏定义

3.2.3 objdump.txt

可以通过生成的 objdump.txt 看到 VMA 和 LMA 的相关分布。

下图是使用 cstartup_b91_flash.S 和 flash_boot.link 编译生成的一个 objdump.txt 文件。

其中 vectors、text 的 VMA 和 LMA 地址是一样的，这两个段是从 flash 中取指执行。

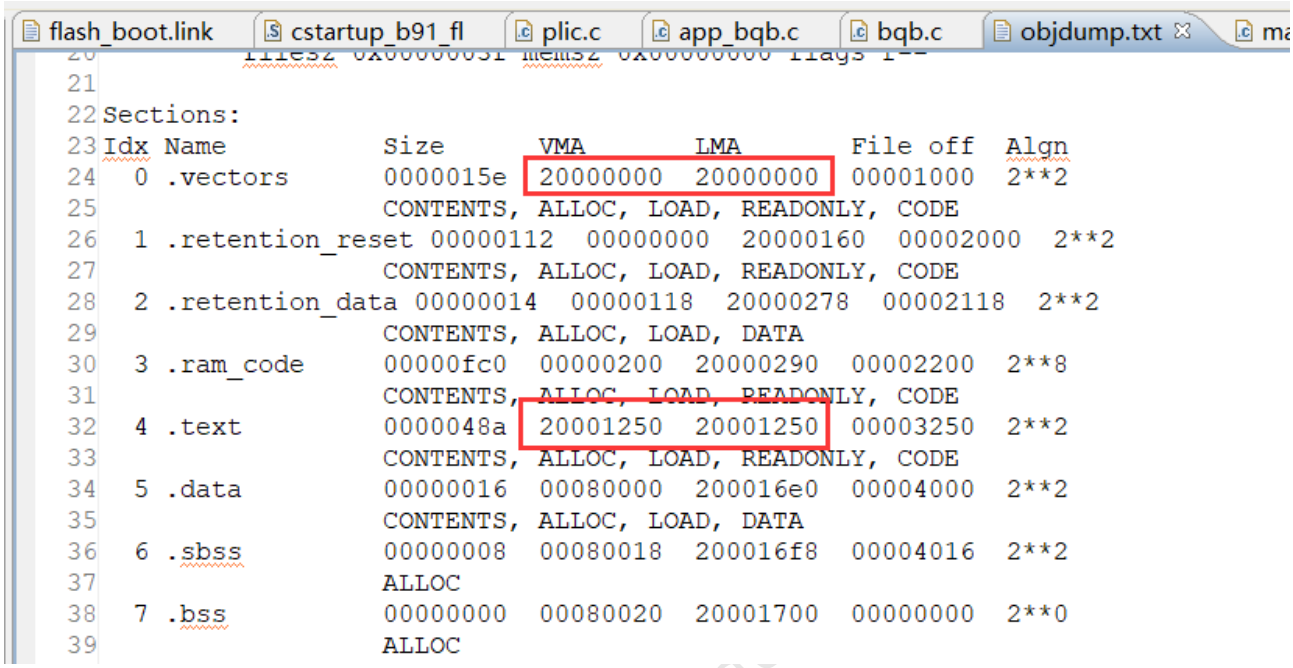


Figure 3.4: vectors、text 的 VMA 和 LMA 地址

retention_reset、retention_data、ram_code 他们的 VMA 和 LMA 的地址是不一样的，LMA 的地址是 flash 地址，VMA 的地址是 IRAM 地址，在 S 文件中，会对应的将这三个段从 flash 中搬移到 IRAM 中。

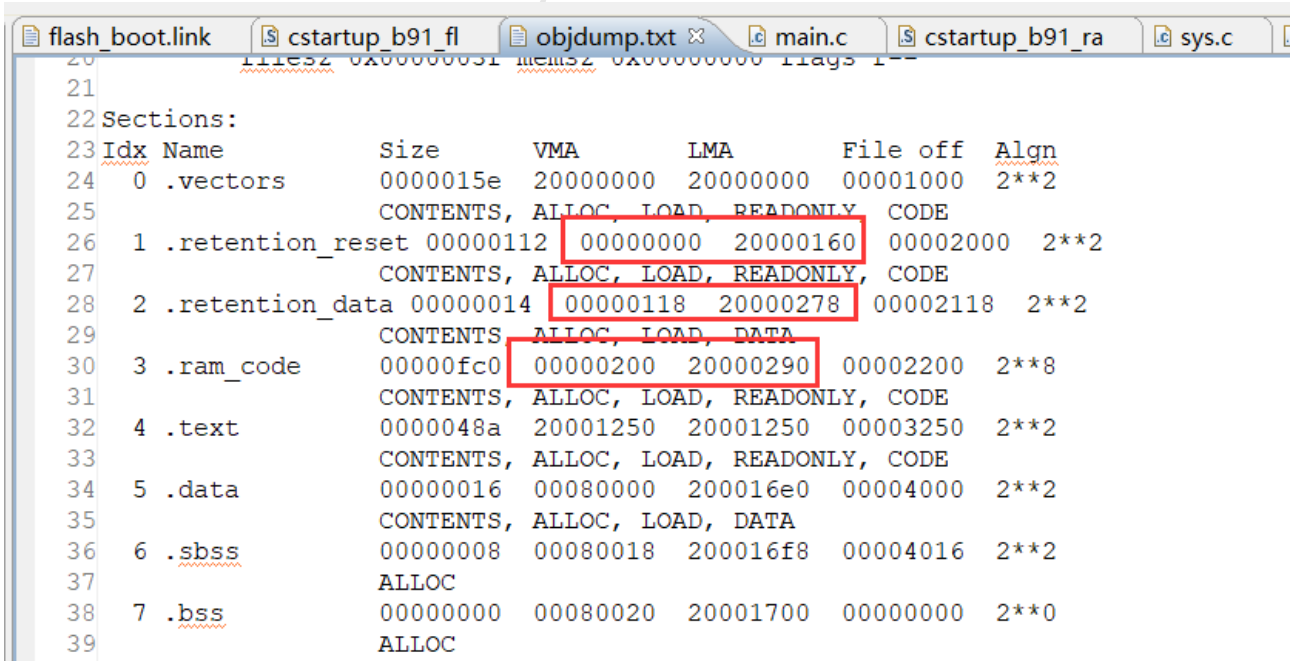


Figure 3.5: retention_reset、retention_data、ram_code 的 VMA 和 LMA 地址

data 段的 VMA 和 LMA 的地址是不一样的，LMA 的地址是在 flash 中，VMA 的地址是在 DRAM 中。

在 S 文件中，会对应的将 data 段从 flash 中搬移到 DRAM 中。

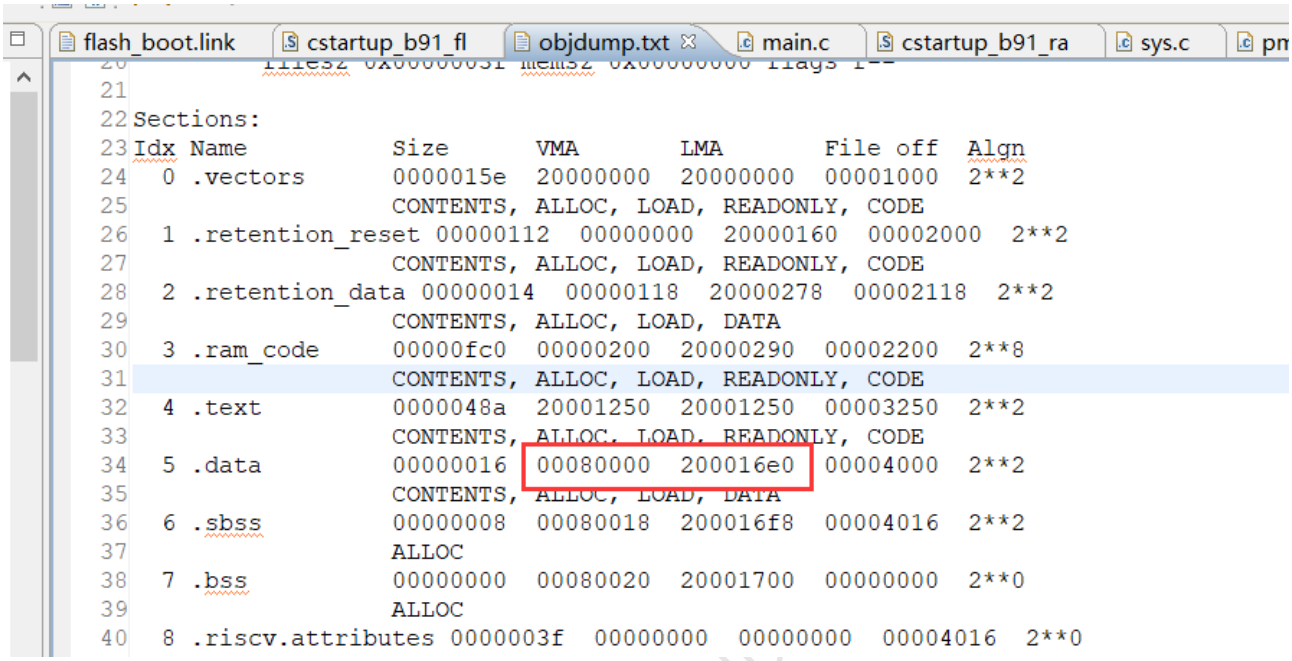


Figure 3.6: data 段的 VMA 和 LMA 的地址

3.3 Link 文件详解

3.3.1 代码详解

以 flash_boot.link 文件为例：

```
//设置代码入口为 _RESET_ENTRY
ENTRY(_RESET_ENTRY)
SECTIONS
{
//定义变量 NDS_SAG_LMA_FLASH = 0x20000000
    NDS_SAG_LMA_FLASH = 0x20000000 ;
//指定当前地址为 0x20000000（不加 AT 的涉及到的地址都是 VMA），点即是代表当前地址
//LMA(Load Memory Address)：装载地址，这里可以简单的理解为是 flash 中的地址。
    . = 0x20000000;
//定义变量 BIN_BEGIN等于当前地址
    PROVIDE (BIN_BEGIN = .);
//定义 vectors 段，VMA 地址和 LMA 的地址均为 0x2000000，所以这里没有加 AT 指令。
//即 vectors 段的装载地址以及运行地址均是在 0x20000000(这里 0x20000000 是 flash 的基地址，即
 ↪  vectors 段是存储在 flash 的 0x20000000，同时也是从这个地址取指运行的)
//keep 相当于告诉编译器，这部分不要被垃圾回收。garbage collection 就是删除不用的 section，不输出
 ↪  到输出文件中，使用--gc-sections 选项设置。但是可以使用 KEEP 来保留。如下面的 vectors 段是必须
 ↪  要保留的。
```

```
    .vectors: { KEEP(*(.vectors )) }
//指定当前位置为 0x0（即 IRAM 的起始地址）
    . = 0x00000000;
//定义 retention_reset 段，VMA 地址为 0x0
//LMA 地址 = ALIGN(LOADADDR (.vectors) + SIZEOF (.vectors),8)
//LOADADDR (section): 获取 section 的 LMA 的地址
//SIZEOF (section): 获取 section 的大小
//AT (addr)：定义本段的 LMA 的地址。
//当 VMA 和 LMA 不一致的时候，需要用 AT 指令设置 LMA
    .retention_reset  : AT( ALIGN(LOADADDR (.vectors) + SIZEOF (.vectors),8))
     { KEEP(*(.retention_reset )) }
//retention_reset 段的 VMA、LMA 的一些地址需要保存到变量中，S 文件中会用到
    PROVIDE (_RETENTION_RESET_VMA_START = ADDR(.retention_reset));
    PROVIDE (_RETENTION_RESET_LMA_START = LOADADADDR(.retention_reset));
    PROVIDE (_RETENTION_RESET_VMA_END = .) .
//aes_data 段只可以再 IRAM 的前 64K 地址，所以如果不清楚用法的情况下，请不要修改他的位置。
//设置当前地址 = . 按 8 对齐的地址
//如果没有对当前地址赋值（点），则所有的段的地址会顺次排列。
    . = ALIGN(8);
    PROVIDE (_AES_VMA_START = .) .
    .aes_data   (NOLOAD)  : { KEEP(*(.aes_data )) }
    PROVIDE (_AES_VMA_END = .) .
    . = ALIGN(8);
    .retention_data : AT( ALIGN(LOADADDR (.retention_reset) + SIZEOF (.retention_reset),8))
      { KEEP(*(.retention_data )) }
    PROVIDE (_RETENTION_DATA_VMA_START = ADDR(.retention_data));
    PROVIDE (_RETENTION_DATA_LMA_START = LOADADADDR(.retention_data));
    PROVIDE (_RETENTION_DATA_VMA_END = .) .

    . = ALIGN(8);
    .ram_code : AT( ALIGN(LOADADDR (.retention_data) + SIZEOF (.retention_data),8))
      { KEEP(*(.ram_code )) }
    PROVIDE (_RAMCODE_VMA_END = .) .
    PROVIDE (_RAMCODE_VMA_START = ADDR(.ram_code));
    PROVIDE (_RAMCODE_LMA_START = LOADADADDR(.ram_code));
    PROVIDE (_RAMCODE_SIZE = SIZEOF (.ram_code));
    . = ALIGN(LOADADADDR (.ram_code) + SIZEOF (.ram_code), 8);
    .text    : AT(ALIGN(LOADADADDR (.ram_code) + SIZEOF (.ram_code), 8))
       { *(.text .stub .text.* .gnu.linkonce.t.* ) KEEP(*(.text.*personality* )) *(.gnu.warning
↪   ) }
    . rodata   : AT(ALIGN(LOADADADDR (.text) + SIZEOF (.text), ALIGNOF(. rodata)))
       { *(. rodata . rodata.* .gnu.linkonce.r.* )}
//增加了 eh_frame/eh_frame_hdr 的段的分配，在使用 puts 函数时，如果不分配就会出现编译错误
    .eh_frame_hdr  : AT(ALIGN(LOADADADDR (. rodata) + SIZEOF (. rodata),
↪   ALIGNOF(.eh_frame_hdr)))
        { *(.eh_frame_hdr ) }
```

```
    . = ALIGN(0x20);
    .eh_frame : AT(ALIGN(LOADADDR (.eh_frame_hdr) + SIZEOF (.eh_frame_hdr), 32))
        { KEEP(*(.eh_frame )) }
//为指令压缩表分配内存空间.exec.itable
    .exec.itable : AT(ALIGN(LOADADDR (.eh_frame) + SIZEOF (.eh_frame), ALIGNOF(.exec.itable)))
        { KEEP(*(.exec.itable)) }

    . = 0x00080000;
    PROVIDE( __global_pointer$ = . + (4K / 2) );
//ALIGNOF(.data): 返回 data 的 VMA 的对齐要求。
//如果 section 已分配，返回名为 section 的对齐字节。如果段还没被分配，链接器会报错。
    .data : AT(ALIGN(LOADADADDR (.exec.itable) + SIZEOF (.exec.itable), ALIGNOF(.data)))
        { *(.data .data.* .gnu.linkonce.d.* ) KEEP(*(.gnu.linkonce.d.*personality* ))
↪   SORT(CONSTRUCTORS)
          \*(.srodata.cst16 ) \*(.srodata.cst8 ) \*(.srodata.cst4 ) \*(.srodata.cst2 ) \*(.
↪   srodata . srodata.\* ) \*(. sdata . sdata.\* .gnu.linkonce.s.\* ) \*(.sdata2 .sdata2.\*
↪   .gnu.linkonce.s.\* )
        }
    PROVIDE (_DATA_VMA_END = .) .
    PROVIDE (_DATA_VMA_START = ADDR(.data));
    PROVIDE (_DATA_LMA_START = LOADADDR(.data));
//BIN_SIZE = data 段的 LMA 地址 + data 段的大小 - BIN_BEGIN
//bin 文件其实就是所有段的 LMA 拼接在一起组成的，一般都会按照前后顺序依次排列（AT 指令中的值是按
↪   照这个规则来设置的）
    PROVIDE (BIN_SIZE = LOADADDR(.data) + SIZEOF(.data) - BIN_BEGIN);
//每个输出 section 可以有一个类型，类型是用圆括号括起来的关键字。
//NOLOAD，section 被标记为不可加载类型，程序运行时不会载入到内存。
//bss 段不需要载入，S 文件会将 bss 段的 VMA 地址空间清零。
    . = ALIGN(8);
    PROVIDE (_BSS_VMA_START = .) .
    . sbss (NOLOAD) : { \*(. dynsbss ) \*(. sbss . sbss.\* .gnu.linkonce.sb.* ) \*(. scommon .
↪   scommon.\* ) }
    . bss  (NOLOAD) : { \*(. dynbss ) \*(. bss . bss.\* .gnu.linkonce.b.\* ) \*(COMMON ) . =
↪   ALIGN(8); }
    PROVIDE (_BSS_VMA_END = .) .

    . = ALIGN(8);
//_end 是堆的起始地址，堆是向上增长的，一般我们设置在 bss 后面不用的空间
//调用了 sprintf/malloc/free 之类的函数，这些函数会调用 _sbrk 函数进行堆内存的分配，_sbrk 会通过
↪   _end 符号确定从哪里开始分配堆空间（一般都是.bss 段的末尾），否则会出现链接错误
    _end = .
    PROVIDE (end = .) .
//定义栈的起始地址，这里定义的是 DRAM 的结尾位置
    PROVIDE (_STACK_TOP = 0x00a0000);
    PROVIDE (FLASH_SIZE = 0x0100000);
}
```

```
ASSERT((BIN_SIZE)<= FLASH_SIZE, "BIN FILE OVERFLOW");
```

3.3.2 对齐

Link 文件中会涉及到一些对齐规则，如果涉及到多个段一起搬移，需要注意这几个段的对齐方式，LMA 和 VMA 起始地址不一致的时候，多个段一起搬移，很可能出现搬移出错的情况。

下面是错误的一个例子：

S 文件中的代码是想要将 sdata 和 data 段从 LMA 搬移到 VMA 中，VMA 的起始地址是 0x00080000，LMA 的起始地址和前面的代码相关，并且是按照 sdata 的 VMA 的对齐规则进行对齐的。

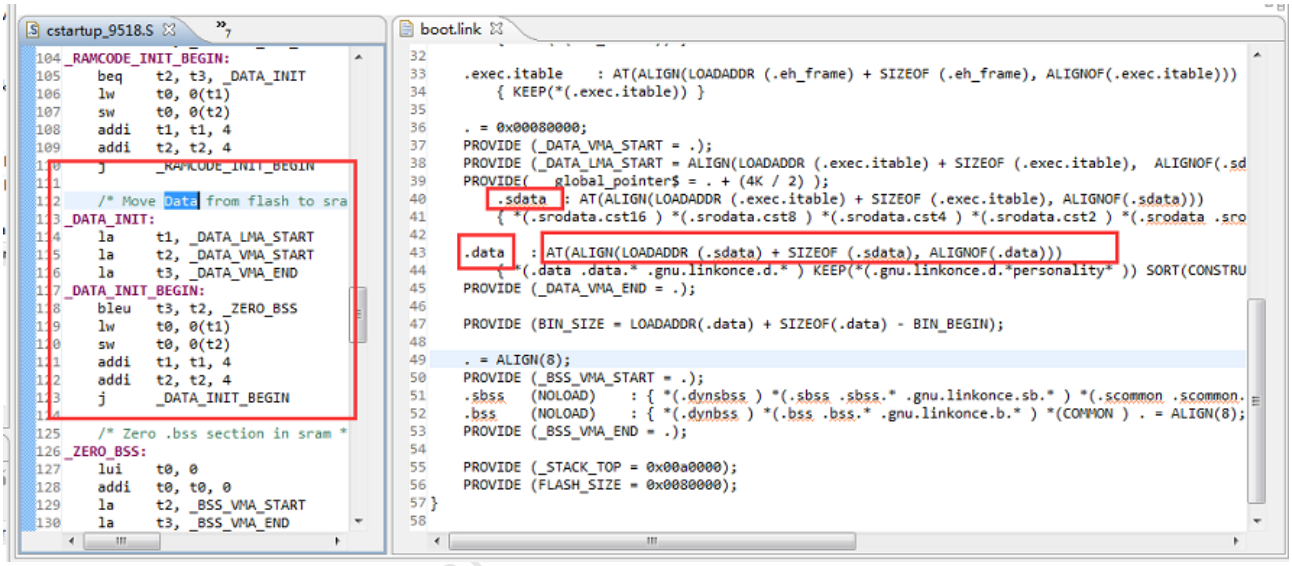


Figure 3.7: link 文件对齐规则的错误例子

下图是用上述 link 文件编译生成的一个会出错的 lst 文件。

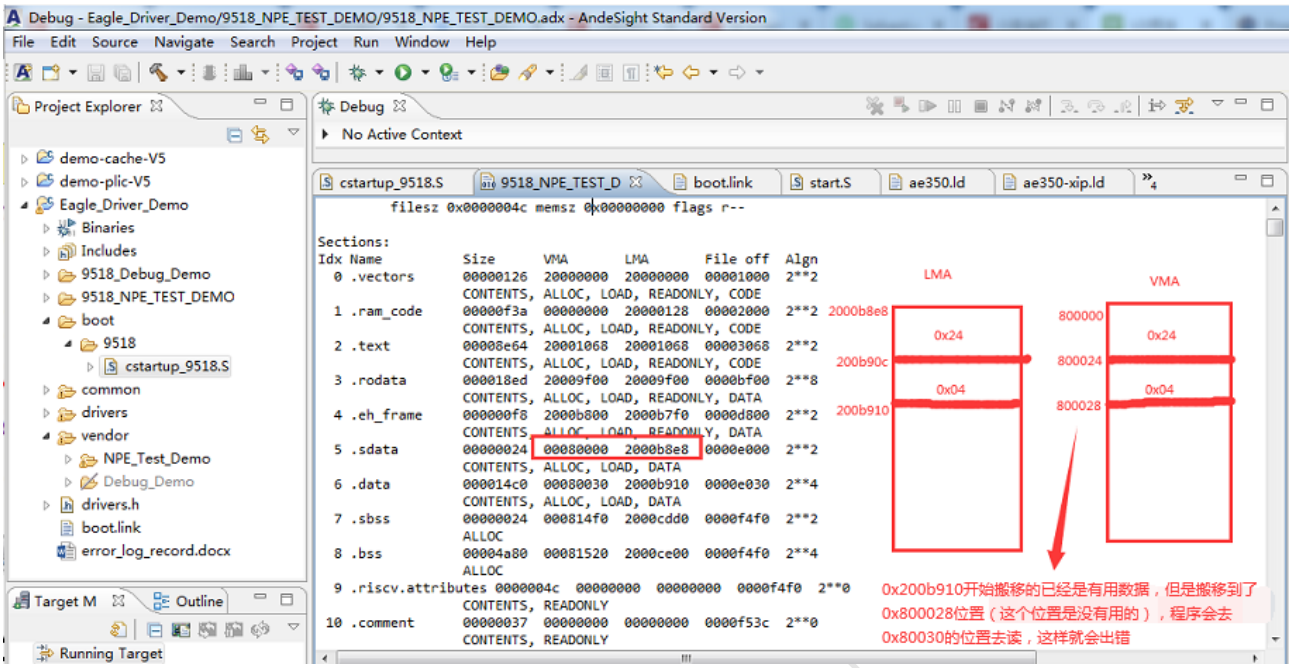


Figure 3.8: 出错的 lst 文件

修改方法有以下几种：

(1) 一个段一个段去搬移，这样就不会有上面的问题。

(2) 合并 sdata 和 data 段。

34 S 文件详解

以 cstartup_b91_flash.S 文件为例。

34.1 代码详解

```
//定义 vector 段，这句后面开始的代码属于 vectors 段，直到遇到下一个段名，或者文件结束。
//其中“ax”代表该 section 可分配并且可执行。
//`a' section is allocable `x' section is executable
.section .vectors, "ax"
.option push        //push 保存当前的.option 配置
.option norelax    //设置为 norelax
.org 0x0           //伪指令.org 是告诉编译器下一条指令的偏移地址。设置偏移地址为 0
// .global 伪指令用于定义一个全局的符号，使得链接器能够全局识别它，即一个程序文件中定义的符号能够
↪ 被所有其他程序文件可见。
.global _RESET_ENTRY
//.type 伪指令用于定义符号的类型。下面是将 _RESET_ENTRY 定义为一个函数。
.type _RESET_ENTRY,@function
//.align 伪指令用于将当前 PC 地址推进到”2 的 integer 次方个字节”对齐的位置。下面是将当前 PC 地
↪ 址推进到 4 个字节对齐的位置处。
```

```
.align 2
//标号 _RESET_ENTRY 为程序的入口地址。
_RESET_ENTRY:
//程序的入口地址必须是可执行指令，这里是一个跳转指令，跳转到标号 _START。
    j     _START
//设置偏移地址为 0x18 的位置存放 BIN_SIZE(4 个字节)BIN_SIZE 在 link 文件中有定义。
    .org 0x18
    .word (BIN_SIZE)
//设置偏移地址为 0x20 的位置存放关键字，这个位置必须是这个值，否则程序跑不起来。
    .org 0x20
    .word ('T'<<24 | 'L'<<16 | 'N'<<8 | 'K')
//设置偏移地址为 0x26 的位置存放 flash 配置，后面 flash 取指会根据这个配置决定走哪个协议。
//支持以下 6 种，可以根据 flash 型号进行选择。
     .org 0x26
     //.short (0x0003) //READ: cmd:1x, addr:1x, data:1x, dummy:0
    //.short (0x070B) //FREAD: cmd:1x, addr:1x, data:1x, dummy:8
    .short (0x173B) //DREAD: cmd:1x, addr:1x, data:2x, dummy:8
    //.short (0x53BB) //X2READ: cmd:1x, addr:2x, data:2x, dummy:4
    //.short (0x276B) //QREAD: cmd:1x, addr:1x, data:4x, dummy:8
    //.short (0x65EB) //X4READ: cmd:1x, addr:4x, data:4x, dummy:6
//使用 pop 恢复.option 配置
.option pop
//4 个字节对齐
    .align 2

_START:
//此处是为了 debug 使用的，会将 PB4 输出高，一般用来看状态的，默认关闭。
#if 0
    lui   t0,0x80140        //0x8014030a
    li    t1, 0xef
    li    t2, 0x10
    sb    t1 , 0x30a(t0)   //0x8014030a PB oen = 0xef
    sb    t2 , 0x30b(t0)   //0x8014030b PB output = 0x10
#endif
//初始化全局指针 gp 寄存器，__global_pointer$ 在 link 文件中定义。
    .option push
    .option norelax
    la gp, __global_pointer$
    .option pop
//初始化全局指针 sp 寄存器，__global_pointer$ 在 link 文件中定义。
     la t0, _STACK_TOP
    mv sp, t0

#ifdef __nds_execit
//设置指令压缩表的及地址
    la t0, _ITB_BASE_
```

```
    csrw uitb , t0
#endif
//设置 FS 为 0b11，清 fscsr（是浮点运算之前需要做的处理）
#ifdef __riscv_flen
    /* Enable FPU */
    li t0, 0x00006000
    csrrs t0, mstatus , t0
    /* Initialize FCSR */
    fscsr zero
#endif
//设置中断入口基地址（base）
    la t0, __vectors
    csrw mtvec, t0
//使能中断的 vector 模式（需要使能两个地方，这里不可以修改）
    /* Enable vectored external plic interrupt */
    csrsi mmisc_ctl, 2

    /*vector mode enable bit (VECTORED) of the Feature Enable Register */
    lui     t0, 0xe4000
    li      t1, 0x02
    sw      t1, 0x0(t0)   //(*(volatile unsigned long*)(0xe4000000))= 0x02
//使能 I/D-Cache
    csrr t0, mcache_ctl
    ori t0, t0, 1 #/I-Cache
    ori t0, t0, 2 #/D-Cache
    csrw mcache_ctl, t0
    fence.i
//将 retention_reset 段从 flash 中搬到 ram 中

_RETENTION_RESET_INIT:
    la t1, _RETENTION_RESET_LMA_START
    la t2, _RETENTION_RESET_VMA_START
    la t3, _RETENTION_RESET_VMA_END
_RETENTION_RESET_BEGIN:
    bleu t3, t2, _RETENTION_DATA_INIT
    lw t0, 0(t1)
    sw t0, 0(t2)
    addi t1, t1, 4
    addi t2, t2, 4
    j _RETENTION_RESET_BEGIN
//将 retention_data 段从 flash 中搬到 ram 中
_RETENTION_DATA_INIT:
    la t1, _RETENTION_DATA_LMA_START
    la t2, _RETENTION_DATA_VMA_START
    la t3, _RETENTION_DATA_VMA_END
_RETENTION_DATA_INIT_BEGIN:
```

```
    bleu t3, t2, _RAMCODE_INIT
    lw t0, 0(t1)
    sw t0, 0(t2)
    addi t1, t1, 4
    addi t2, t2, 4
    j _RETENTION_DATA_INIT_BEGIN
//将 ram_code 段从 flash 中搬到 ram 中
_RAMCODE_INIT:
    la t1, _RAMCODE_LMA_START
    la t2, _RAMCODE_VMA_START
    la t3, _RAMCODE_VMA_END
_RAMCODE_INIT_BEGIN:
    bleu t3, t2, _DATA_INIT
    lw t0, 0(t1)
    sw t0, 0(t2)
    addi t1, t1, 4
    addi t2, t2, 4
    j _RAMCODE_INIT_BEGIN
//将 data 段从 flash 中搬到 ram 中
_DATA_INIT:
    la t1, _DATA_LMA_START
    la t2, _DATA_VMA_START
    la t3, _DATA_VMA_END
_DATA_INIT_BEGIN:
    bleu t3, t2, _ZERO_BSS
    lw t0, 0(t1)
    sw t0, 0(t2)
    addi t1, t1, 4
    addi t2, t2, 4
    j _DATA_INIT_BEGIN
//将 bss 段清零
_ZERO_BSS:
    lui t0, 0
    la t2, _BSS_VMA_START
    la t3, _BSS_VMA_END
_ZERO_BSS_BEGIN:
    bleu t3, t2, _ZERO_AES
    sw t0, 0(t2)
    addi t2, t2, 4
    j _ZERO_BSS_BEGIN
//将 AES 段清零
_ZERO_AES:
    lui t0, 0
    la t2, _AES_VMA_START
    la t3, _AES_VMA_END
_ZERO_AES_BEGIN:
```

```
    bleu t3, t2, _FILL_STK
    sw t0, 0(t2)
    addi t2, t2, 4
    j _ZERO_AES_BEGIN
//将堆栈区域均初始化为 0x55，默认该段代码是不打开的，因为 ram 比较大，会比较费时，如果 debug 需
 ↪ 要，可以打开使用。
_FILL_STK:
#if 0
    lui t0, 0x55555
    addi t0, t0, 0x555
    la t2, _BSS_VMA_END
    la t3, _STACK_TOP
_FILL_STK_BEGIN:
    bleu t3, t2, _MAIN_FUNC
    sw t0, 0(t2)
    addi t2, t2, 4
    j _FILL_STK_BEGIN
#endif
//跳转到 main 函数
_MAIN_FUNC:
    nop
//使用 j 或者 jal 只能跳转到 [-524288,524287]，超出这个范围就只能使用 jalr 来实现，为避免出现此类
 ↪ 问题，统一使用 jalr 来进行跳转。
    la t0, main
    jalr t0

    nop
    nop
    nop
    nop
    nop
_END:
j _END

//定义一个宏，名字是 INTERRUPT，参数是 num，.endm 时这个宏的结束。
.macro INTERRUPT num
//弱定义
.weak entry_irq\num
.set entry_irq\num, default_irq_entry
.long entry_irq\num
.endm
//中断源一共有 64 个
#define VECTOR_NUMINTRS 63
.section .ram_code, "ax"
//定义 ram_code 段，所有的中断入口地址都应该在 ram_code，这样才能快速进入中断。
.global __vectors
```

```
//共有 64 个中断源，这里需要设置的对齐的计算公式为：2ceiling(log2(N))+2，N 为 64，所以这里应该设
↪ 置为 256 对齐。
.balign 256

__vectors:
    .long trap_entry
//实际这里是一个 for 循环，是在定义除了 trap 中断以外的 63 个中断入口地址
//展开之后是：
//.weak entry_irq1
//.set entry_irq1, default_irq_entry
//.long entry_irq1
//.........
//.weak entry_irq63
//.set entry_irq63, default_irq_entry
//.long entry_irq63
//我们使用 vector mode，发生中断后，pc 会指向地址 __vectors +4* 中断 ID，如果中断 ID 为 2，则会跳
↪ 转到 entry_irq2
.altmacro
.set irqno, 1
.rept VECTOR_NUMINTRS/* .rept .endr */
INTERRUPT %irqno
.set irqno, irqno+1
.endr
```

34.2 vectors 和 retention_reset 段的差异

retention_reset 是 retention 回来会跑的启动代码。

目前 retention_reset 的处理是不会从 flash 中搬这些段（retention_reset、retention_data、ram_code）到 ram，因为 retention 阶段这些段是保留的不会丢失的。这个处理的条件是，retention ram 大小是比这些段大的，如果有超过，则需要做搬移动作。

retention_reset 启动部分比 vectors 需要多加的处理包括：flash 唤醒、多地址寄存器恢复。这些是 retention 回来必须做的处理。

34.3 其他

34.3.1 .org 的使用注意事项

目前的编译环境下，.org 和优化选项中的 Link Time Optimization（-flto）不能同时使用，但是为了编译出来的 bin 文件小，（-flto）是一定会选择的，所以在 S 文件中如果想使用.org，需要按照如下方法使用：

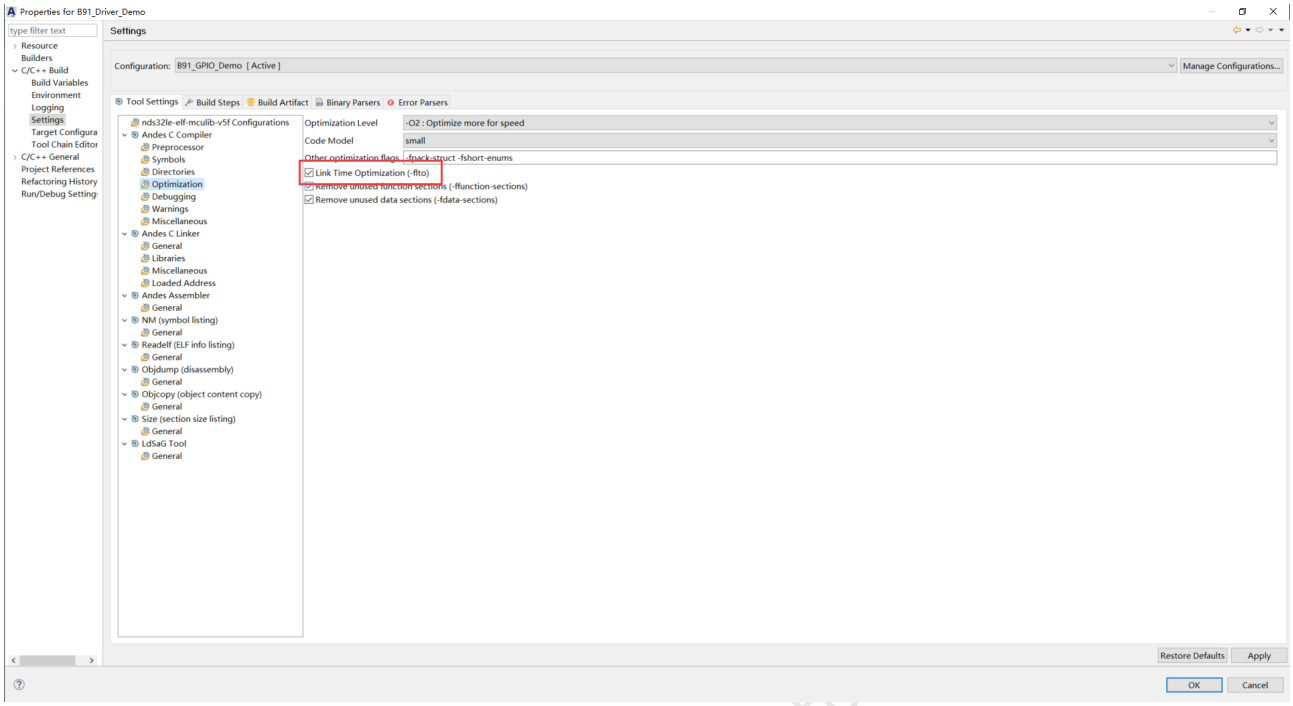


Figure 3.9: 优化选项中的 Link Time Optimization（-flto）

```
.option push       //push 保存当前的.option 配置
.option norelax   //设置为 norelax
.org 0x0          //设置.org
.........
.option pop     //使用 pop 恢复.option 配置
```

.option 说明如下：

.option 伪指令用于设定某些架构特定的选项，使得汇编器能够识别此选项并按照选项的定义采取相应的行为。

push、pop 用于临时性的保存或者恢复.option 伪指令指定的选项：

“.option push”伪指令暂时将当前的选项设置保存起来，从而允许之后使用.option 伪指令指定新的选项；而“.option pop” 伪指令将最近保存的选项设置恢复出来重新生效。

通过“.option push”和“.option pop”的组合，便可以在汇编程序中在不影响全局选项设置的情况下，为其中嵌入的某一段代码特别地设置不同的选项。

34.3.2 压缩指令

RISC-V 的 C Extension，指的是用 16bit 的指令替换 32bit 指令，而 Andes 的 CoDense（Code Dense）技术，是把 32 位的指令放到指令表里面，在原来 32 位指令出现的地方，用 16 EXEC.IT 0xxxxx 来代替。

宏 __nds_execit 在编译器中默认设置为打开状态，同时 _ITB_BASE_ 会设置为.exec.itable 的首地址。

在 S 文件中将 _ITB_BASE_ 设置到寄存器 uitb 中，作为指令表的基地址。压缩指令会放到.exec.itable 段中。

34.3.3 FPU **使能**

宏 __riscv_flen 在编译器中默认设置为打开状态。

在执行浮点指令前，需要将 mstatus<14:13>FS 字段改为非 0 的值，否则会抛指令异常，所以 S 文件中将 FS 设置为了 0b11，初始化浮点的控制状态寄存器 FCSR 为 0。

4 Debug Demo

Driver 里没有重新定义 printf 接口，直接使用了 toolchain 自带的 printf 接口，但是我们对其进行了重定向，driver 里分别实现了两种重定向，一种是将数据重定向到 GPIO（通过 GPIO 模拟串口时序），一种是重定向到 USB。可选择任意一种进行 debug info 输出。

可以在 printf.h 中选择，使用 GPIO 还是 USB 打印。

```
#define DEBUG_IO        0
#define DEBUG_USB       1
#define DEBUG_BUS       DEBUG_IO
```

4.1 GPIO 口模拟串口输出

GPIO 口相关配置在 printf.h 中配置选择，包括波特率等（只需要接 RX 和 GND）。硬件接法如下：

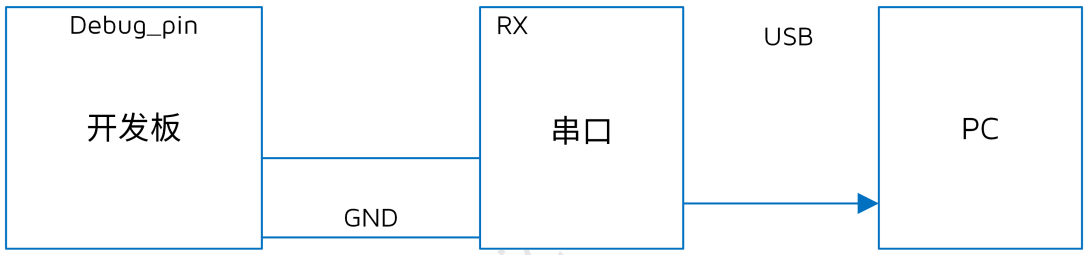


Figure 4.1: GPIO 口硬件接法

下图为串口助手接收到的打印输出信息：

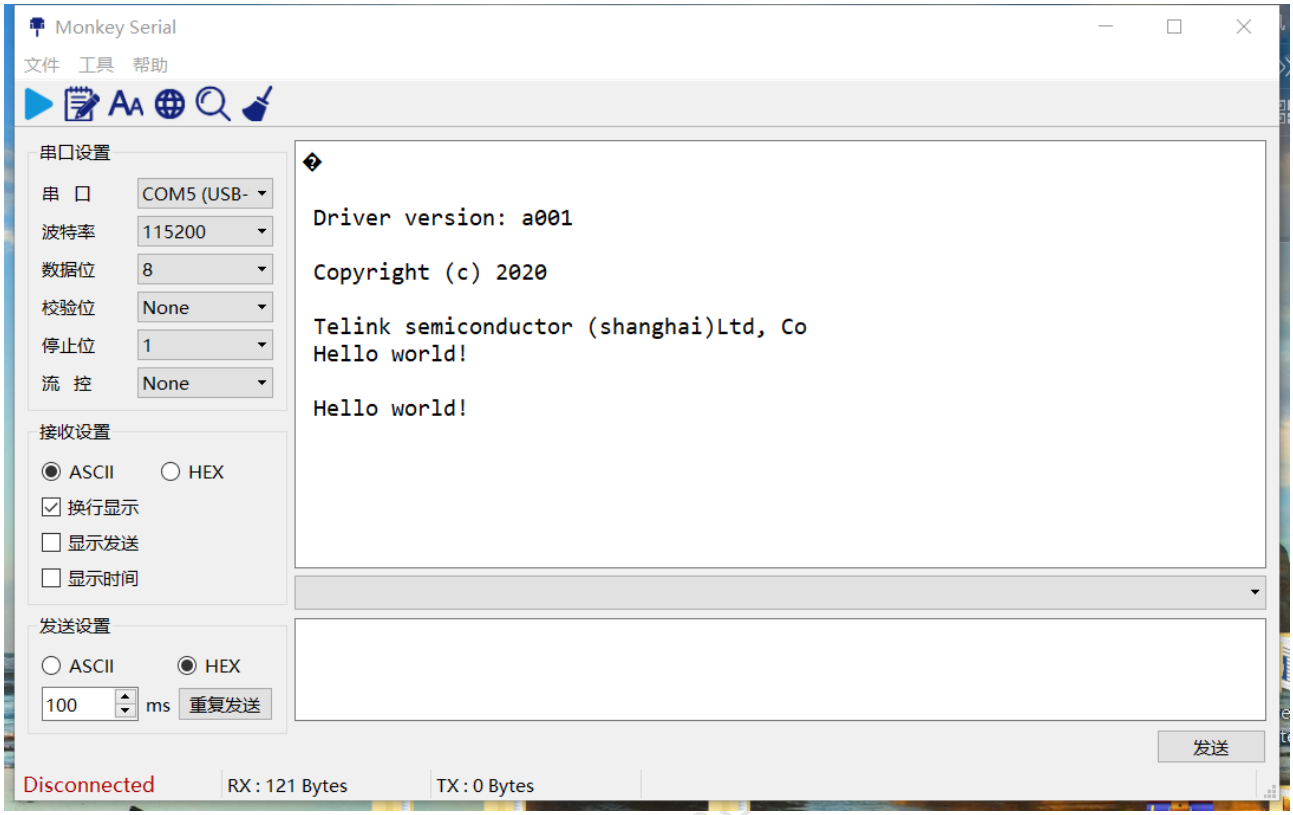


Figure 4.2: 串口助手接收到的打印输出信息

4.2 USB 打印输出

配置为 USB printf，需要使用 BDT 工具来查看输出信息，使用 USB 打印时需要注意的一点是，USB 使用的是固定 48M 时钟，在时钟初始化默认已经配置好了，但是如果使用的 PLL clock 不能整除得到 48M 的话，USB 可能不能正常使用。

使用 USB printf 时可以配置阻塞和非阻塞模式，可通过 printf 里的相关宏定义选择，默认非阻塞，如下：

```
#define BLOCK_MODE          0
```

如下为 BDT 配置 usb_log 方法和实验现象：

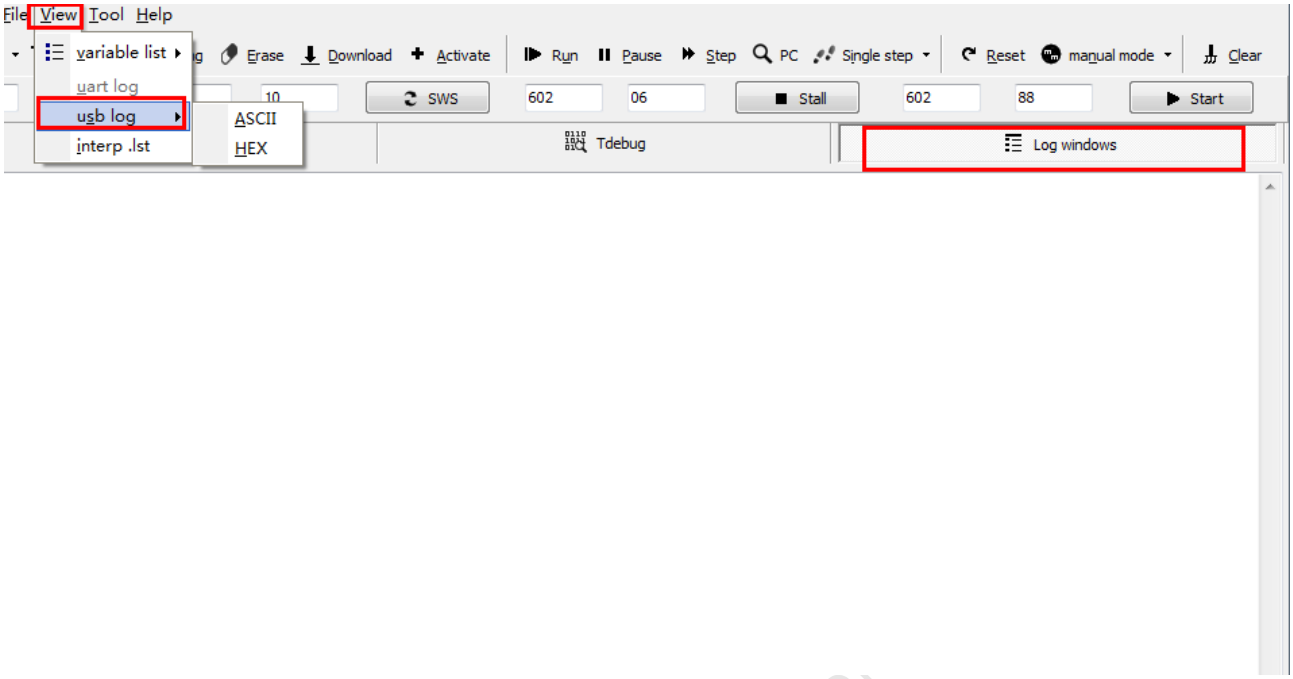


Figure 4.3: BDT 配置 usb_log 方法

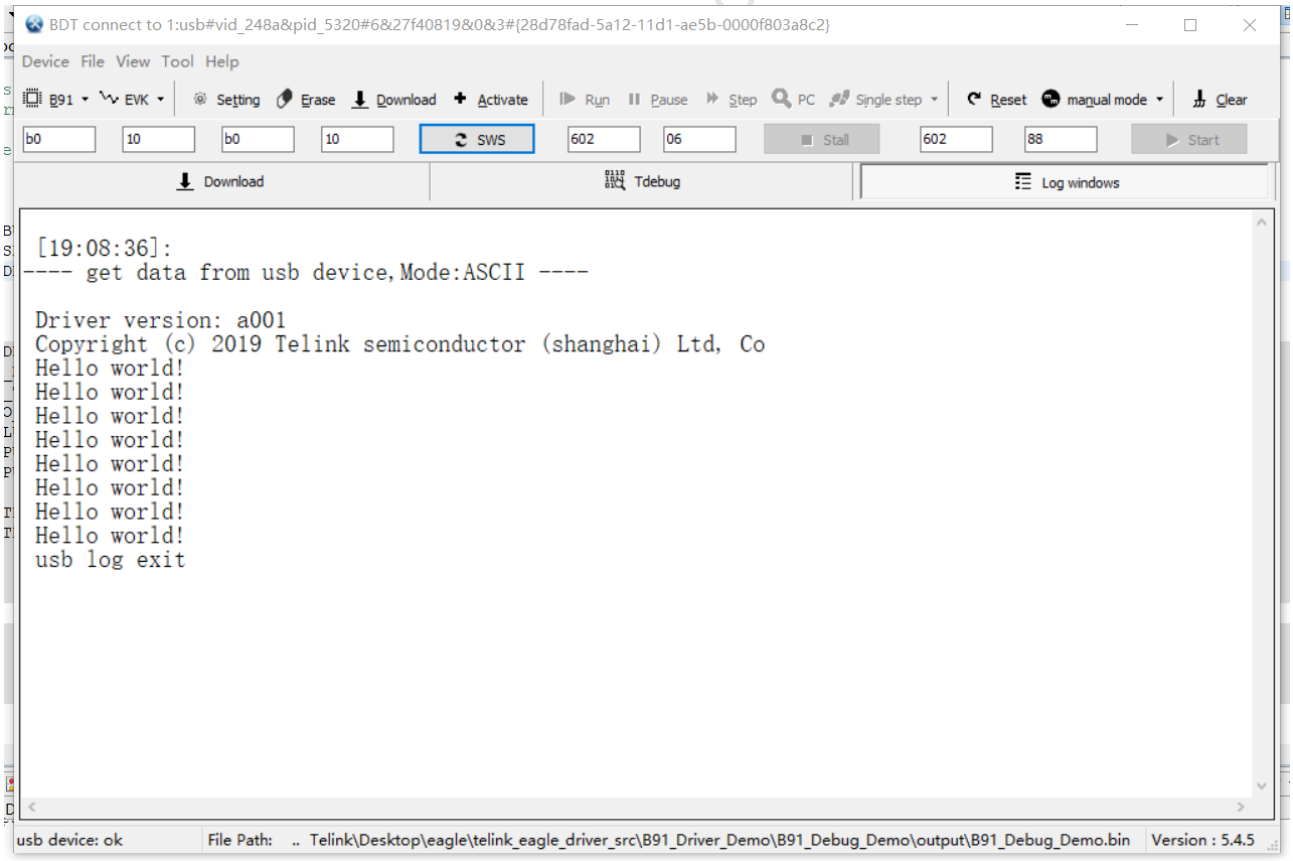


Figure 4.4: 实验现象

5 中断

PLIC （Platform-Level Interrupt Controller）兼容 RISC-V PLIC，具有两大功能：中断向量和中断优先级。

5.1 中断概述

中断机制，处理器在顺序执行程序指令流过程中突然被别的请求打断而中止执行当前的程序，转而去处理别的事情，待其处理完了别的事情，然后重新回到之前程序中断的点继续执行之前的程序指令流，而"别的请求"便称之为中断请求，别的"请求来源"称之为中断源，中断源通常来自于外围设备。处理器转而去执行"别的事情"便称之为中断处理程序。

5.2 中断类型

RISV-V 的架构模式有三种工作模式：机器模式（Machine Mode）、用户模式（User Mode）和监督模式（Supervisor Mode）。我们的使用的是机器模式，对应的机器模式下有三种中断类型：软件中断（Software Interrupt）、计时中断（Timer Interrupt）和外部中断（External Interrupt）。外部中断指 CPU 外部中断，例如 UART、GPIO 等产生的；计时中断是指来自计时器的中断，软件中断是来自软件自己触发的中断。

注意：

- 中断有两种模式，Vector 和 regular 模式，driver 默认的是 vector 模式。

5.3 外部中断

5.3.1 中断使能

驱动中断接口的使用，如果想打开中断，分下面的三个层级：

第一层

```
core_enable_interrupt()
```

使能 RISC-V core 中 CSR 寄存器中的对应 BIT，这个是总的中断开关。

第二层

```
plic_interrupt_enable()
```

使能 plic 模块中对应的模块的 BIT，这个是模块中断控制开关。

第三层

```
rf_set_irq_mask(FLD_ZB_RX_IRQ)
```

使能对应模块的 mask，以 rf 模块为例，需要用哪个，设置哪个即可。

5.3.2 Vector 模式下外部中断处理函数

在 plic 驱动中已经定义好了相应的中断处理函数，对应如下：

中断向量	中断处理函数
IRQ0_EXCEPTION	except_handler
IRQ1_SYSTIMER	stimer_irq_handler
IRQ2_ALG	analog_irq_handler
IRQ3_TIMER1	timer1_irq_handler
IRQ4_TIMER0	timer0_irq_handler
IRQ5_DMA	dma_irq_handler
IRQ6_BMC	bmc_irq_handler
IRQ7_USB_CTRL_EP_SETUP	usb_ctrl_ep_setup_irq_handler
IRQ8_USB_CTRL_EP_DATA	usb_ctrl_ep_data_irq_handler
IRQ9_USB_CTRL_EP_STATUS	usb_ctrl_ep_status_irq_handler
IRQ10_USB_CTRL_EP_SETINF	usb_ctrl_ep_setinf_irq_handler
IRQ11_USB_ENDPOINT	usb_endpoint_irq_handler
IRQ12_ZB_DM	rf_dm_irq_handler
IRQ13_ZB_BLE	rf_ble_irq_handler
IRQ14_ZB_BT	rf_bt_irq_handler
IRQ15_ZB_RT	rf_irq_handler
IRQ16_PWM	pwm_irq_handler
IRQ17_PKE	pke_irq_handler
IRQ18_UART1	uart1_irq_handler
IRQ19_UART0	uart0_irq_handler
IRQ20_DFIFO	audio_irq_handler
IRQ21_I2C	i2c_irq_handler
IRQ22_SPI_AHB	hspi_irq_handler
IRQ23_SPI_APB	pspi_irq_handler
IRQ24_USB_PWDN	usb_pwdn_irq_handler
IRQ25_GPIO	gpio_irq_handler
IRQ26_GPIO2RISC0	gpio_risc0_irq_handler

中断向量	中断处理函数
IRQ27_GPIO2RISC1	gpio_risc1_irq_handler
IRQ28_SOFT	soft_irq_handler
IRQ29_NPE_BUS0	npe_bus0_irq_handler
IRQ30_NPE_BUS1	npe_bus1_irq_handler
IRQ31_NPE_BUS2	npe_bus2_irq_handler
IRQ32_NPE_BUS3	npe_bus3_irq_handler
IRQ33_NPE_BUS4	npe_bus4_irq_handler
IRQ35_USB_RESET	usb_reset_irq_handler
IRQ36_NPE_BUS7	npe_bus7_irq_handler
IRQ37_NPE_BUS8	npe_bus8_irq_handler
IRQ42_NPE_BUS13	npe_bus13_irq_handler
IRQ43_NPE_BUS14	npe_bus14_irq_handler
IRQ44_NPE_BUS15	npe_bus15_irq_handler
IRQ46_NPE_BUS17	npe_bus17_irq_handler
IRQ50_NPE_BUS21	npe_bus21_irq_handler
IRQ51_NPE_BUS22	npe_bus22_irq_handler
IRQ52_NPE_BUS23	npe_bus23_irq_handler
IRQ53_NPE_BUS24	npe_bus24_irq_handler
IRQ54_NPE_BUS25	npe_bus25_irq_handler
IRQ55_NPE_BUS26	npe_bus26_irq_handler
IRQ56_NPE_BUS27	npe_bus27_irq_handler
IRQ57_NPE_BUS28	npe_bus28_irq_handler
IRQ58_NPE_BUS29	npe_bus29_irq_handler
IRQ59_NPE_BUS30	npe_bus30_irq_handler
IRQ60_NPE_BUS31	npe_bus31_irq_handler
IRQ61_NPE_COMB	npe_comb_irq_handler
IRQ62_PM_TM	pm_irq_handler
IRQ63_EOC	eoc_irq_handler

```
__attribute__((section(".ram_code"))) void default_irq_handler(void)
{

}
void stimer_irq_handler(void) __attribute__((weak, alias("default_irq_handler")));
```

默认状态下，所有的中断处理函数均被弱定义为 default_irq_handler 一个空函数。

弱函数

理论上，一个工程中是不允许有两个相同名称的函数的，这里使用 _weak 指明其中一个是弱函数即可。

程序在编译的时候，如果发现有两个相同名字的函数，而且其中一个是弱函数，就会忽略弱函数，使用正常的函数进行编译，如果发现只有一个弱函数，那还是会使用弱函数参与编译。

上层使用的时候，只要再定义一个一模一样的函数，但是不需要指明是弱函数了，然后在函数中加入一些用户代码即可。

中断现场保存与恢复

在中断处理中没有发现中断保存和恢复，但是功能正常，原因如下:

attribute ((interrupt ("machine"), aligned(4)));

attribute 里有中断的声明，编译器看到会插入修改保护寄存器的代码。

5.3.3 外部中断内的优先级

多个中断源同时向处理器发起请求时，需要对这些中断源仲裁，哪些中断源被优先处理，就存在中断优先级的概念。当中断优先级不使能的情况下，新的中断不会打断正在处理的中断，等完成中断服务函数后才能响应新的中断请求。默认中断抢占功能是不打开的，如果需要使用中断抢占功能需要三个步骤：

a) 设置 plic_set_threshold(). 只有中断优先级大于阈值（priority> threshold）才能产生中断，threshold 默认为 0，priority 默认为 1.

b) plic_preempt_feature_en();//使能中断抢占功能

c) plic_set_priority();//设置中断优先级

注意：

> • SoC 有 4 个中断优先级 0-3，没有设置中断优先级的中断源，默认优先级为 1, 数字越大优先级越高，设置为 0 优先级不会产生中断（threshold 默认 0，不满足条件：priority> threshold），高优先级的中断源可以打断比它低的中断源，不能打断同级中断。

Demo 里配置了 3 个中断，stimer 中断、timer0 中断和 rf_tx 中断。

a) 开中断

```
core_interrupt_enable();
```

b) 使能中断抢占，设置优先级，默认 threshold 为 0

```
plic_preempt_feature_en();  //使能中断抢占
plic_set_priority(IRQ1_SYSTIMER,IRQ_PRI_LEV3);  //设置 stimer 优先级
plic_set_priority(IRQ4_TIMER0, IRQ_PRI_LEV2);   //设置 timer0 优先级
plic_set_priority(IRQ15_ZB_RT, IRQ_PRI_LEV1);  //设置 rf 优先级
```

在中断处理函数延时前后，设置了观测 IO，延时前拉高，延时后拉低。Demo 中，stimer 延时 250us，timer0 延时 600us，在 rf_tx 延时 800us。

```
_attribute_ram_code_sec_noinline_ void irq_handler(void)
{ ...
    gpio_set_high_level(LED3);
    delay_us();
    gpio_set_low_level(LED3);
}
```

5.34 结果观测

如下图，中断嵌套打开，rf_tx 间里被优先级高的 timer0 打断，以此类推 Time0 中断处理函数被优先级更高的 stimer 打断。

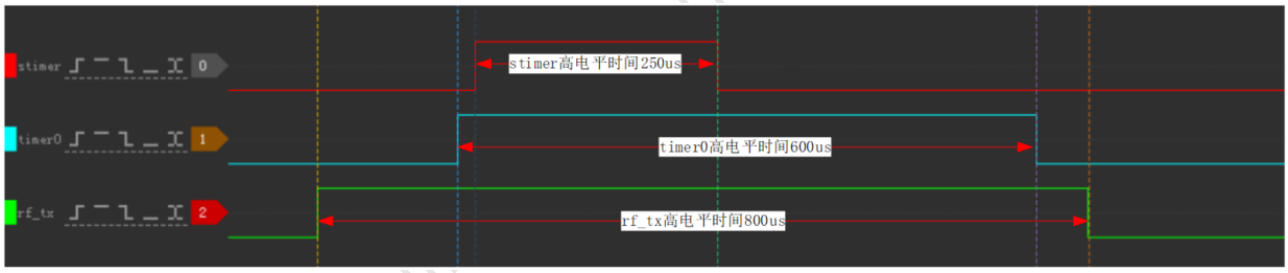


Figure 5.1: 中断嵌套打开

如下图，中断嵌套未打开，可以对比发现三个中断源默认的优先级都是 1，执行完前一个中断处理函数，再去处理下个中断处理函数，相互之间不会被打断。

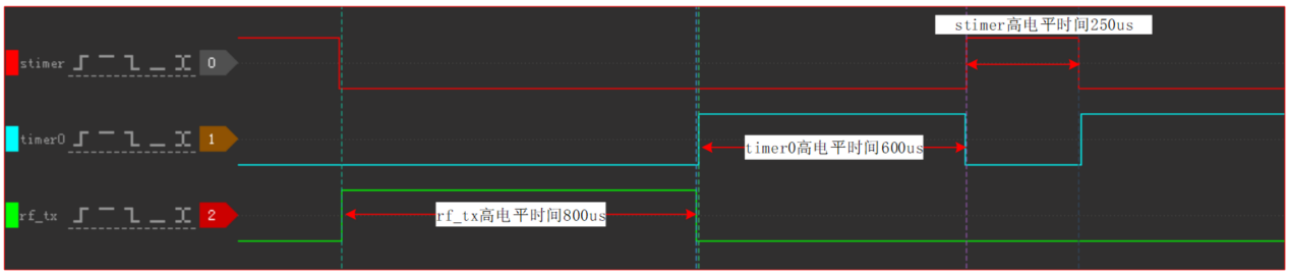


Figure 5.2: 中断嵌套未打开

6 GPIO

SoC 的 GPIO 驱动实现了对 GPIO 的输入输出、上下拉、中断等功能的配置。

6.1 中断

GPIO 可以配置用于产生中断，中断硬件结构如下图所示，每个 GPIO 通过配置可以产生 GPIO_IRQ、GPIO2RISC0_IRQ、GPIO2RISC1_IRQ 三种类型的中断。GPIO_IRQ 是最基本的 GPIO 中断，GPIO2RISC0_IRQ 和 GPIO2RISC1_IRQ 除了包含 GPIO_IRQ 功能外，还可以在 Timer（定时器外设）的应用中产生计数或者控制信号。

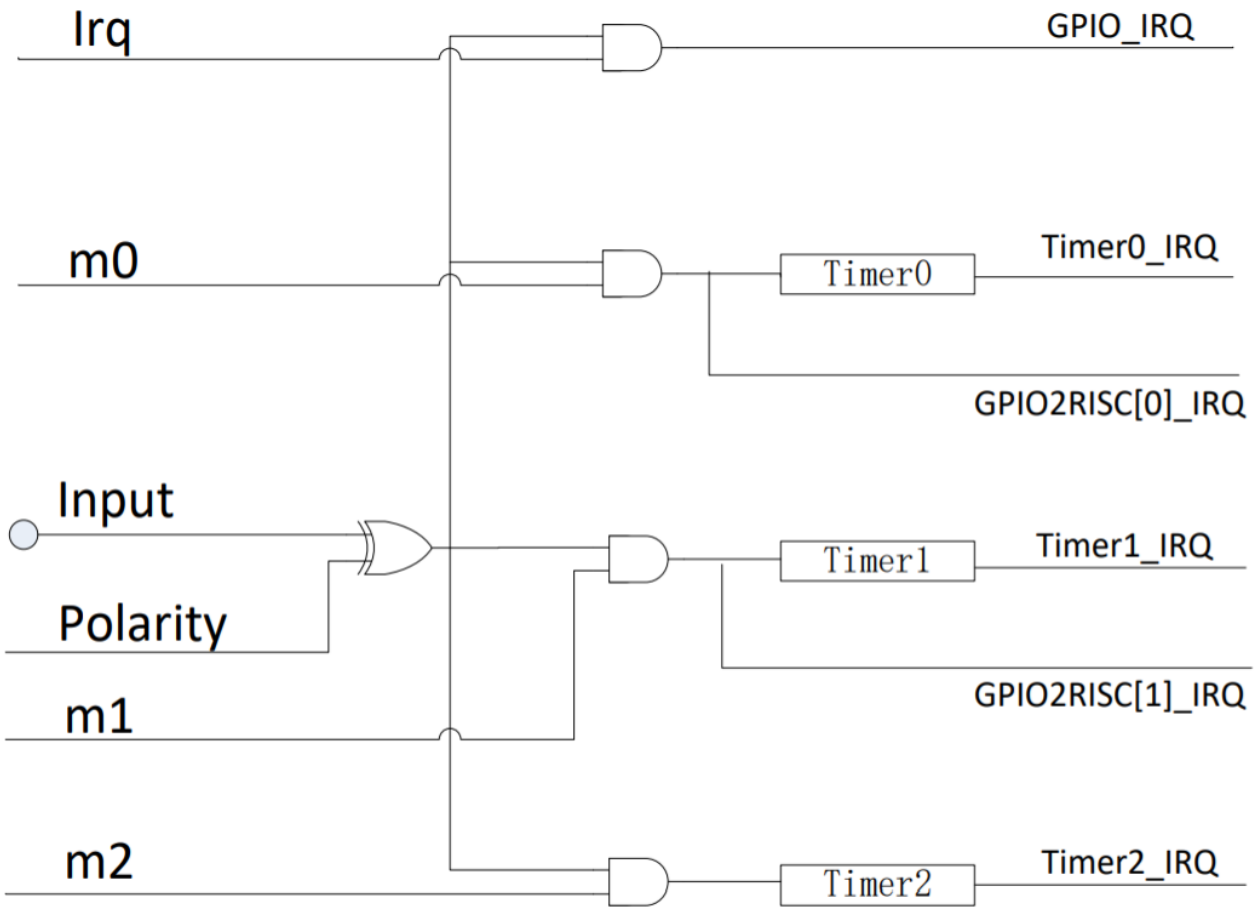


Figure 6.1: 中断硬件结构

6.1.1 机制说明

如下图，GPIO 设置为上升沿触发，MCU 的机制是：将 GPIO 的电平信号作为产生中断的信号，上升沿触发中断。

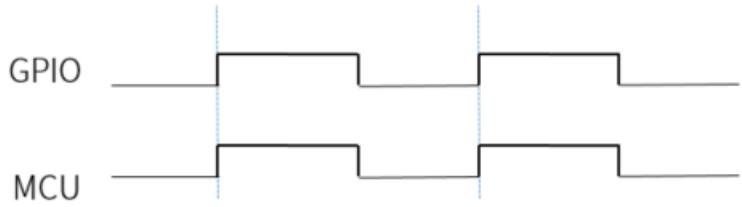


Figure 6.2: GPIO 设置为上升沿触发

如下图，GPIO 设置为下降沿触发，MCU 的机制是：将 GPIO 的电平信号取反，然后将取反后的信号作为中断产生的信号，上升沿触发中断。

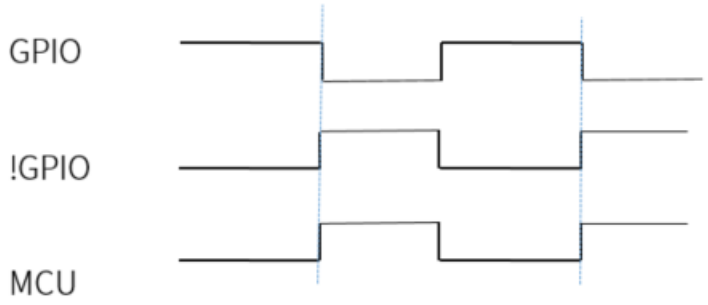


Figure 6.3: GPIO 设置为下降沿触发

如下图，若两个 GPIO 设置为一种中断，上升沿触发，MCU 的机制是：将两个 GPIO 的电平信号做或运算，然后用得到的信号作为产生中断的条件，上升沿触发中断。图中只有 GPIO0 触发了中断。

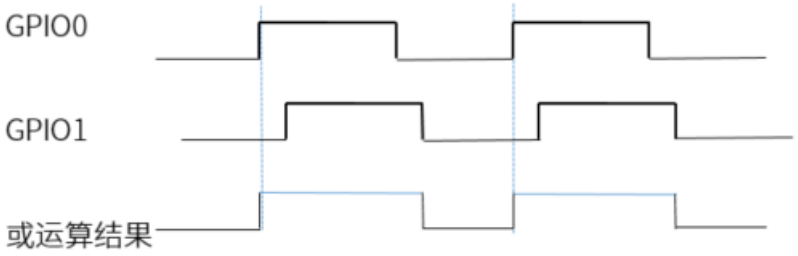


Figure 6.4: 两个 GPIO 设置为一种中断，上升沿触发

如下图，若两个 GPIO 设置为一种中断，下降沿触发，MCU 的机制是：分别将两个 GPIO 的电平信号取反，然后将取反信号做或运算，用得到的信号作为产生中断的条件，也是上升沿触发中断，即最终 MCU 均是用最终信号进行上升沿触发中断。图中只有 GPIO0 触发了中断。

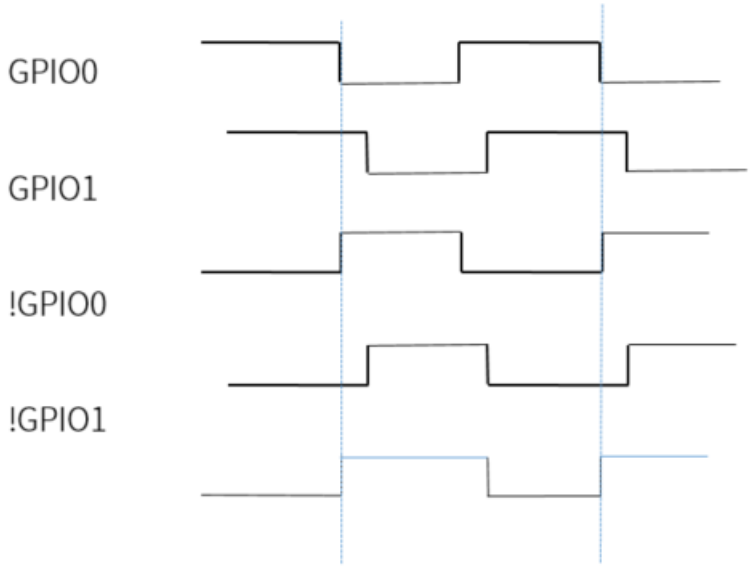


Figure 6.5: 两个 GPIO 设置为一种中断，下降沿触发

如下图，若 GPIO0 设置上升沿触发，GPIO1 设置下降沿触发，MCU 的机制是：将 GPIO1 的电平信号取反，然后将 GPIO0 与（!GPIO1）做或运算，用这个得到的信号作为产生中断的条件，同样也是上升沿触发中断。即最终 MCU 均是用最终信号进行上升沿触发中断。图中只有 GPIO1 触发了中断。

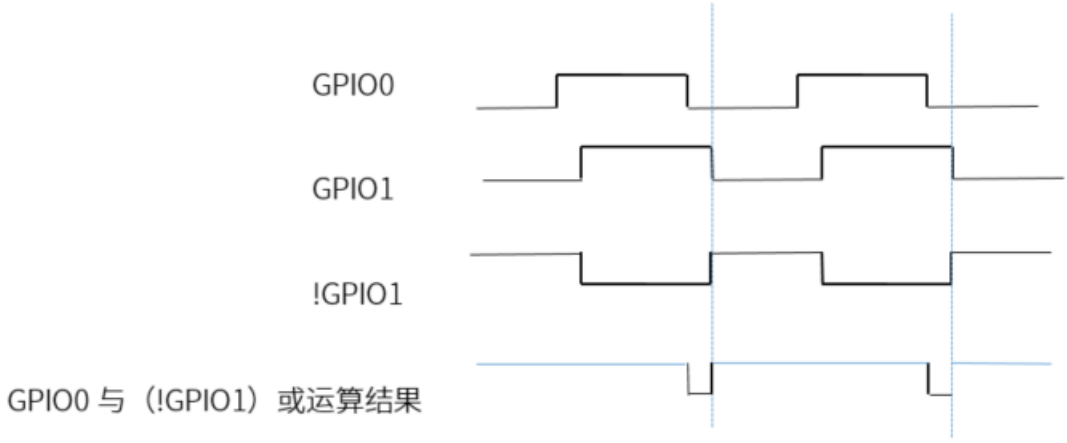


Figure 6.6: GPIO0 设置上升沿触发，GPIO1 设置下降沿触发

6.1.2 结论

两个或多个 GPIO 设置成一种中断，根据输入 GPIO 的时序，触发中断的情况是不确定的，不推荐使用。但不同的 GPIO 中断的机制相互独立，一个 GPIO 设置成一种中断，两个 GPIO 的中断就都可以触发了。

比如 GPIO0 设置成 GPIO_IRQ 中断，GPIO1 设置成 GPIO_IRQ_RSIC0 中断，均配置成上升沿触发。如下图，信号 0 和 1 分别是 GPIO0 的输入时序和中断处理函数内设置的观测 GPIO0 中断的 toggle 信号，信号 2 和 3 分别是 GPIO1 的输入时序和中断处理函数内设置的观测 GPIO1 中断的 toggle 信号。图中 GPIO1 和 GPIO2 均触发了中断。

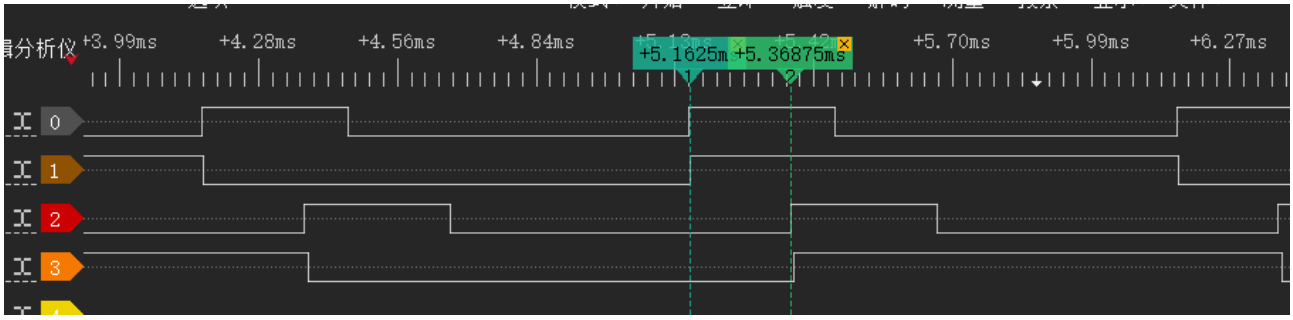


Figure 6.7: 两个或多个 GPIO 设置不同中断

注意：

- 因为 GPIO 中断只有 GPIO_IRQ、GPIO_IRQ_RSIC0、GPIO_IRQ_RSIC1 三种，因此最多可以同时设置三个 GPIO 中断。

6.2 注意事项

如设置触发类型为高电平或者上升沿触发，应该设置下拉电阻，设置触发类型为低电平或者下降沿触发，应该设置上拉电阻。

另外两个不需要应用层关心的问题，也在这里简单说明下（驱动接口中已处理）。

设置下降沿触发时，需要在设置 gpio 的极性后清掉中断位，再使能 mask。否则 gpio 设置下降沿触发时，在使能 gpio 中断时刻产生一次非下降沿导致的中断触发。此部分已经在 gpio 对应的中断使能函数中处理，不需要应用层关心。

GPIO 复用功能切换注意事项：

(1) 开始就是 GPIO 功能，那么需要先将所需要的功能 MUX 配置好后，再将 GPIO 功能 disable。

(2) 开始是功能 IO，需要改成 GPIO output，先设置对应的 IO 的 output 值和 OEN，再 enable GPIO 功能。

(3) 开始是功能 IO，需要改成 GPIO input。

需要该 IO 上拉：

情况 1（数字上拉）：先将 output 设置成 1，OEN 设置为 1；

情况 2（模拟上拉）：将 pullup 设置为 1。

不需要上拉：

情况 1（数字上拉）：将 output 设置为 0，OEN 设置为 1；

情况 2（模拟上拉）：将 pullup 设置为 0。

最后 enable GPIO 功能。

7 Clock

7.1 简述

Risc-V Platform SoC 的 clock 相对比较复杂，如下图所示为部分时钟的时钟树（datasheet 有完整的时钟树结构）。这里介绍几个比较重要的时钟，pll_clk/cclk/hclk/pclk/mspi_clk。

pll_clk：即图中的 PLL，它是很多模块时钟源的源头，包括 sys_clk 一般使用的也是从 PLL 分频过来的。

cclk：即 cpu clk，程序运行的速度是有该 clock 决定的，同时它也是 hclk 和 pclk 的唯一时钟源头。

hclk：所有挂在 AHB 总线上的模块均使用 hclk。

pclk：所有挂在 APB 总线上的模块均使用 pclk。

mspi_clk：mspi 连接 flash，进行 flash 的相关操作，包括取指，读写 flash 等，均是受该时钟控制。

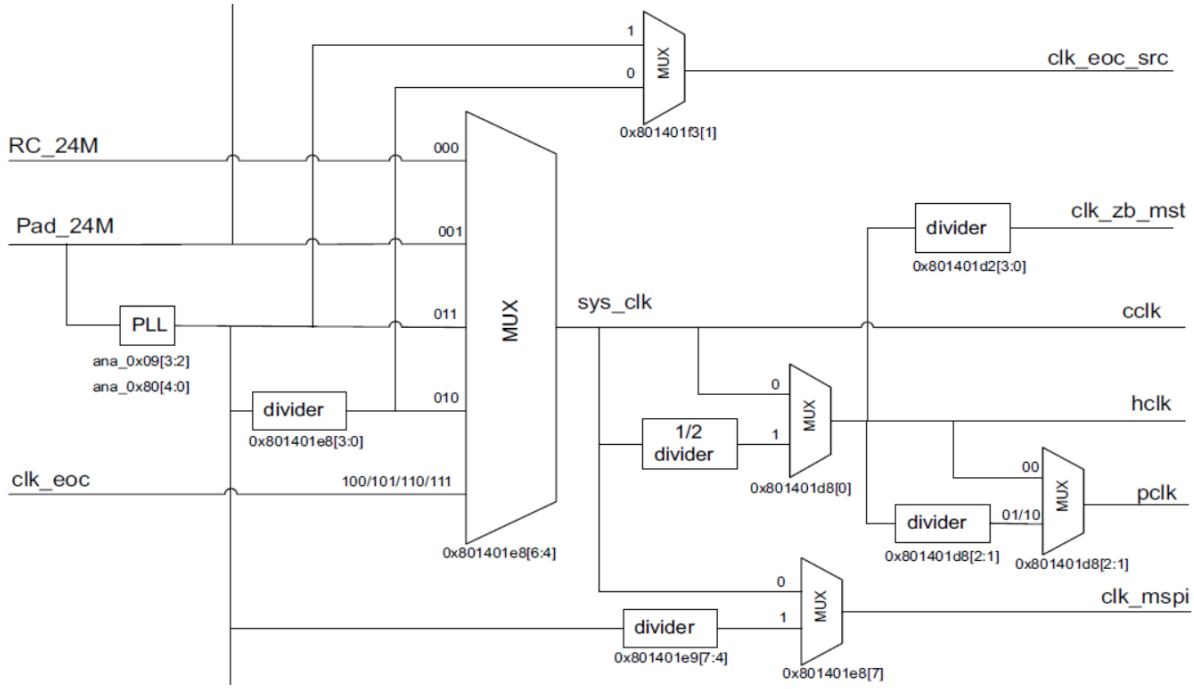


Figure 7.1: 时钟树

7.2 clock_init

Driver 提供了 clock_init 接口可以配置如上的 5 个时钟。

注意：

- 这个函数除了设置参数中的几个 clock，还会设置 USB 的 clock，USB 的 clock 固定使用 48M 时钟。

目前支持的最大 clock 如下表所示：

clk	cclk	hclk	pclk	mspi
fre_max	96M	48M	24M	外挂 flash：48M；内置 flash：64M

7.2.1 PLL_CLK

可使用时钟较多，但如果没有特殊需求，需统一使用 192M。

注意：

- USB 使用固定 48M 时钟，配置 PLL_CLK 时需要注意如果时钟不能分频得到 48M 的话可能导致 USB 不能正常使用。
- 另外 Audio 驱动中的一些频率均是按照 192M 来设置的，如果修改 PLL_CLOCK 会导致 Audio 出错。

7.2.2 CCLK

cclk 可选的时钟源有 4 种，分别为 RC_24M、PAD_24M、PAD_PLL、PAD_PLL_DIV，其中 PAD_PLL_DIV 可选的分频比有 2-15。

7.2.3 HCLK

hclk 由 cclk 分频而来，可以 1、2 分频。

7.24 PCLK

pclk 由 hclk 分频而来，可以 1、2、4 分频。

注意：

- 当 hclk = 1/2 cclk 时，pclk 不能设置 4 分频，这个和硬件相关，接口里已经做了处理，如果这样设置的话，程序会停住，不会往下执行。
- 另外，在使用 REBOOT 或者 PM 的情况下如果 hclk 选择 2 分频时有概率导致死机。

7.2.5 MSPI_CLK

mspi clock 可选两种时钟源，一种是等于 cclk，另一种是由 PLL_clock 分频得到。

8 AES

支持硬件 AES128 加密，所有接口中使用的 buffer 数据，均使用的小端模式。

注意：

- 因为 AES 计算时使用的数据的存放地址是有限制要求的——必须是在 base_address+64K 的地址空间内。
- 目前驱动中处理方式如下：定义 base_address 为 0xc0000000（即 IRAM 的起始地址,），在 IRAM 的前 64K 的范围 aes_data 段（可以在 link 文件中看到），AES 计算时需要处理的数据均放到 aes_data 段进行处理。
- 如果不改变 link 文件中 aes_data 段的位置，可以不关心上述流程，直接使用。
- 如果不满足需求，也可以根据需求自行调整，驱动提供 aes_set_em_base_addr 接口来修改 base_address。
- 另外该地址和 BT 共用，修改该地址还需要确保 BT 使用的数据也在这个地址空间内。

9 EMI

EMI 例程是配合 EMI_Tool" 和 "Non_Signaling_Test_Tool"工具使用的，本文档主要介绍 EMI 测试例程中相关功能和注意事项。

9.1 协议

通信协议请参考《Telink SoC EMI Test User Guide》。

9.2 程序说明

Eagle 中 EMI 测试支持 carrieronly 模式、continue 模式、burst 模式、收包模式。

支持的无线通信方式包括 Ble1M、Ble2M、Ble125K、Ble500K、Zigbee250K。

9.2.1 CarrierOnly 模式

功能：CarrierOnly 模式用于产生单通信号，该模式下可设定单通信号的通道（channel）与能量值（power）以及通信方式（rf mode）。

实例说明：

测试工具：选用 EMI_Tool

无线通信方式设定：Ble1M/Ble2M/Ble125K/Ble500K/Zigbee250K

能量设定：XXdB

通道设定：2402~2480MHz

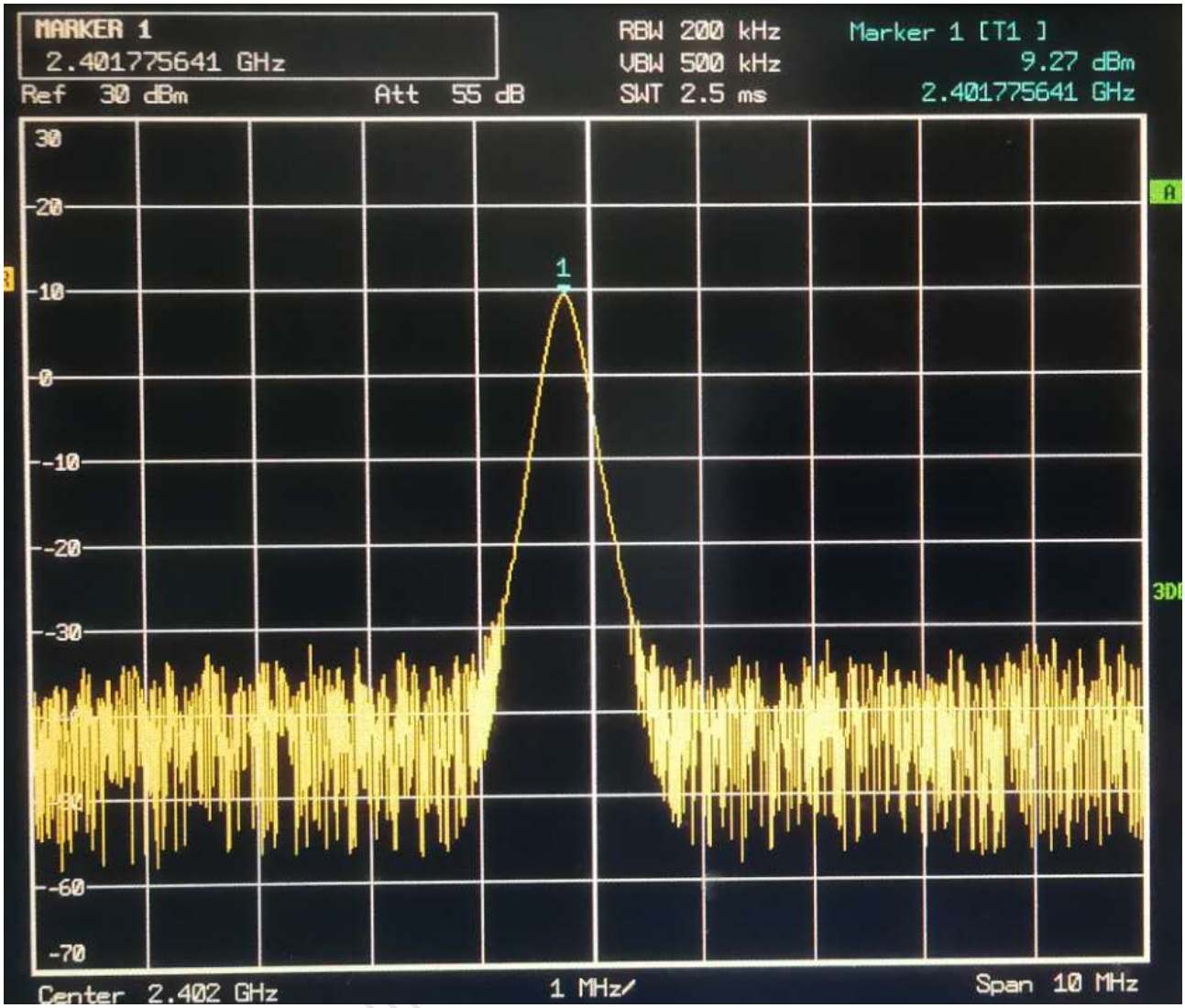


Figure 9.1: CarrierOnly 模式

9.2.2 Continue **模式**

功能：Continue 模式用于产生连续信号，该模式下可设定通道（channel）与能量值（power）以及通信方式（rf mode）。

连续模式下发送包（Payload）数据分为了 Prbs9、0x0f、0x55 三种，并可设置跳频（hop）。

实例说明：

测试工具：选用 EMI_Tool

无线通信方式设定：Ble1M/Ble2M/Ble125K/Ble500K/Zigbee250K

能量设定：XXdB

通道设定：2402~2480MHz

注意：

- 目前 EMI_Tool 在 continue 模式下只支持发送 prbs9 数据包。

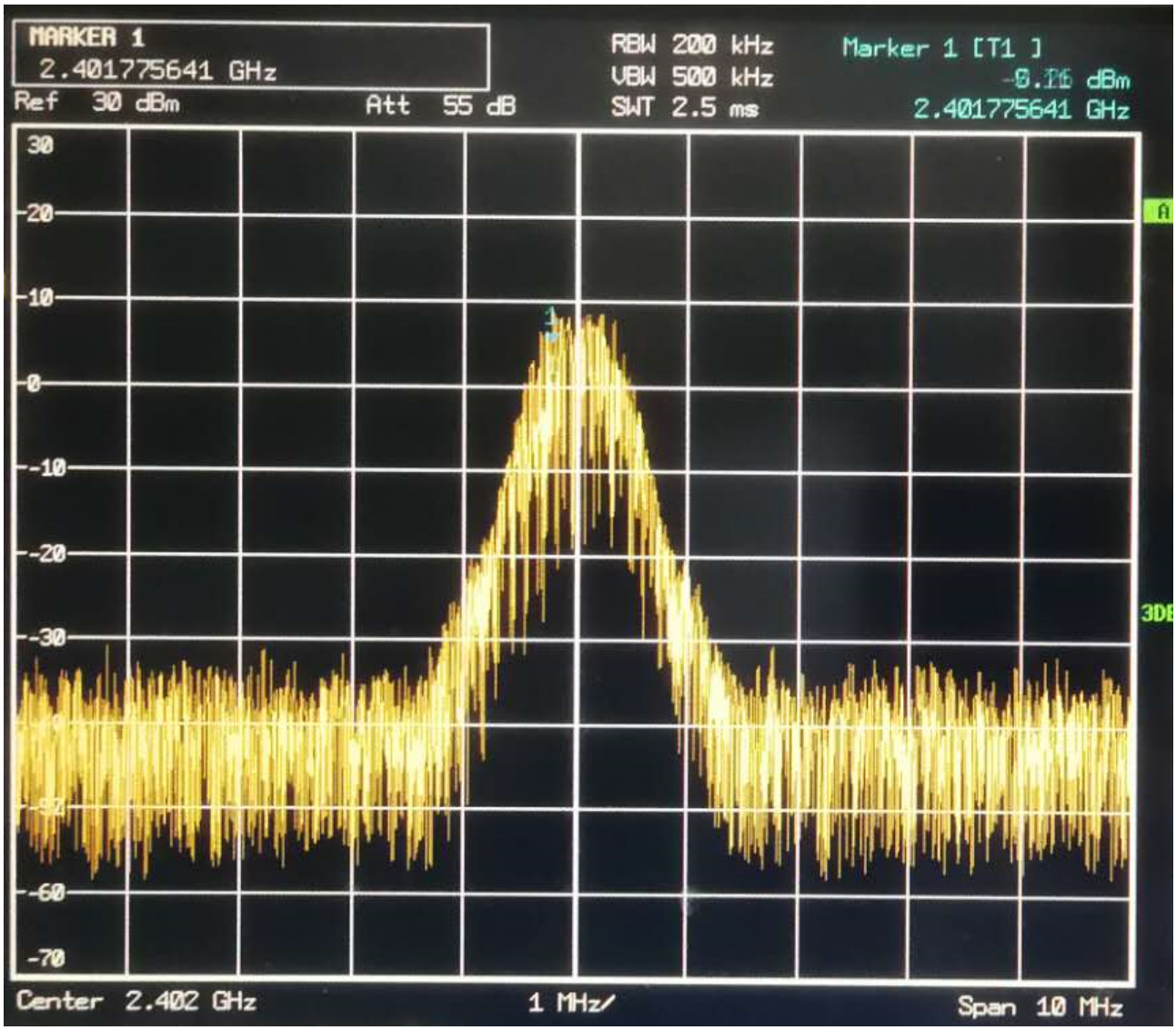


Figure 9.2: Continue 模式

9.2.3 Burst **模式**

功能：Burst 模式下可设定 Burst 信号的通道（channel）与能量值（power）以及通信方式（rf mode）。Burst 模式下发送包（Payload）数据分为了 Prbs9、0x0f、0x55 三种。

实例说明：

测试工具：选用 Non_Signaling_Test_Tool

无线通信方式设定：Ble1M/Ble2M/Ble125K/Ble500K/Zigbee250K

能量设定：XXdB

通道设定：2402~2480MHz

注意：

- Burst 模式下由于信号不连续，可使用频谱分析仪的 Single Sweep 和 MaxHold 设置进行信号抓取。

下面前三幅图为 Single Sweep 设置抓取的结果，第四幅图为 MaxHold 设置抓取的结果

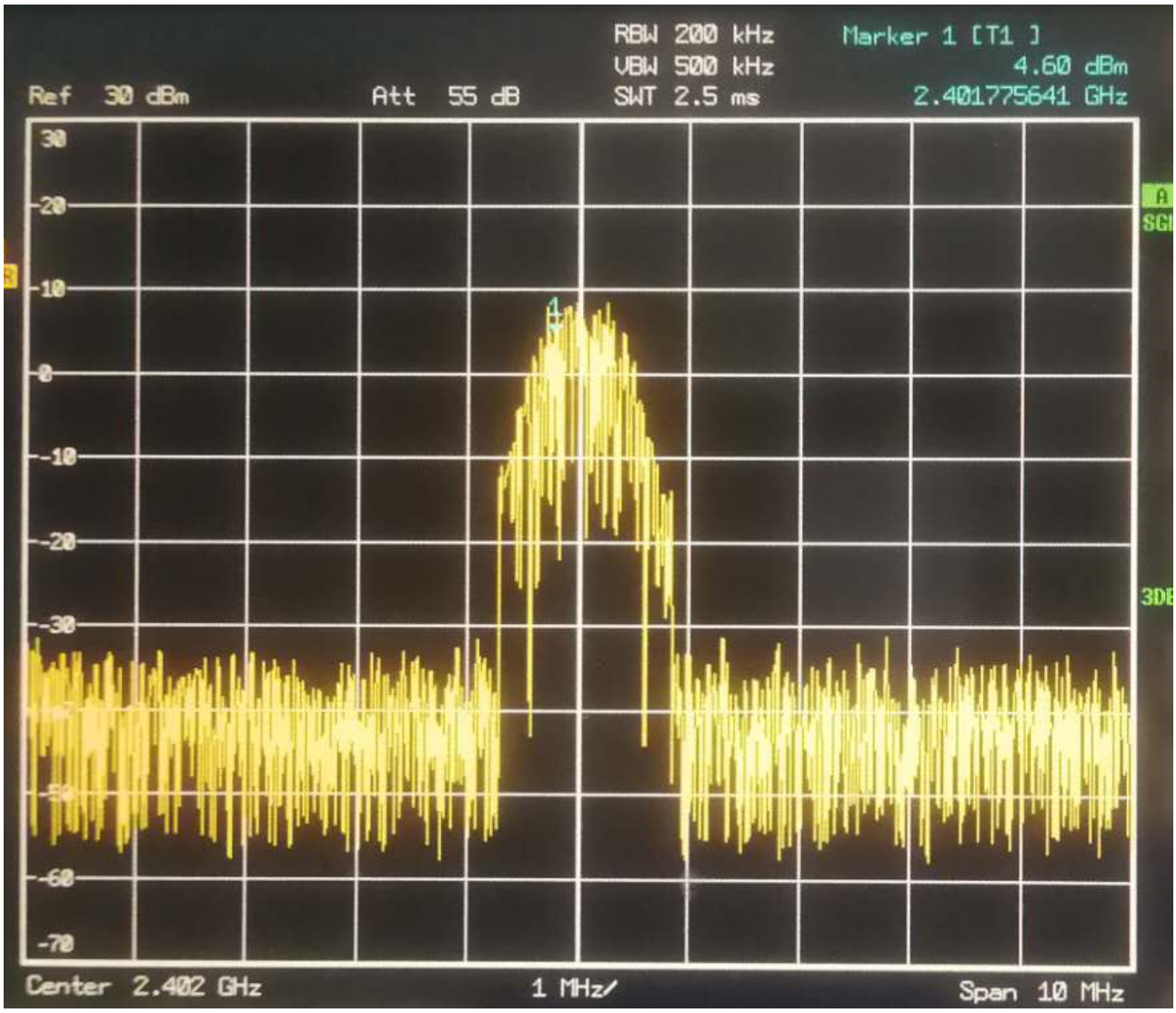


Figure 9.3: Burst 模式 1

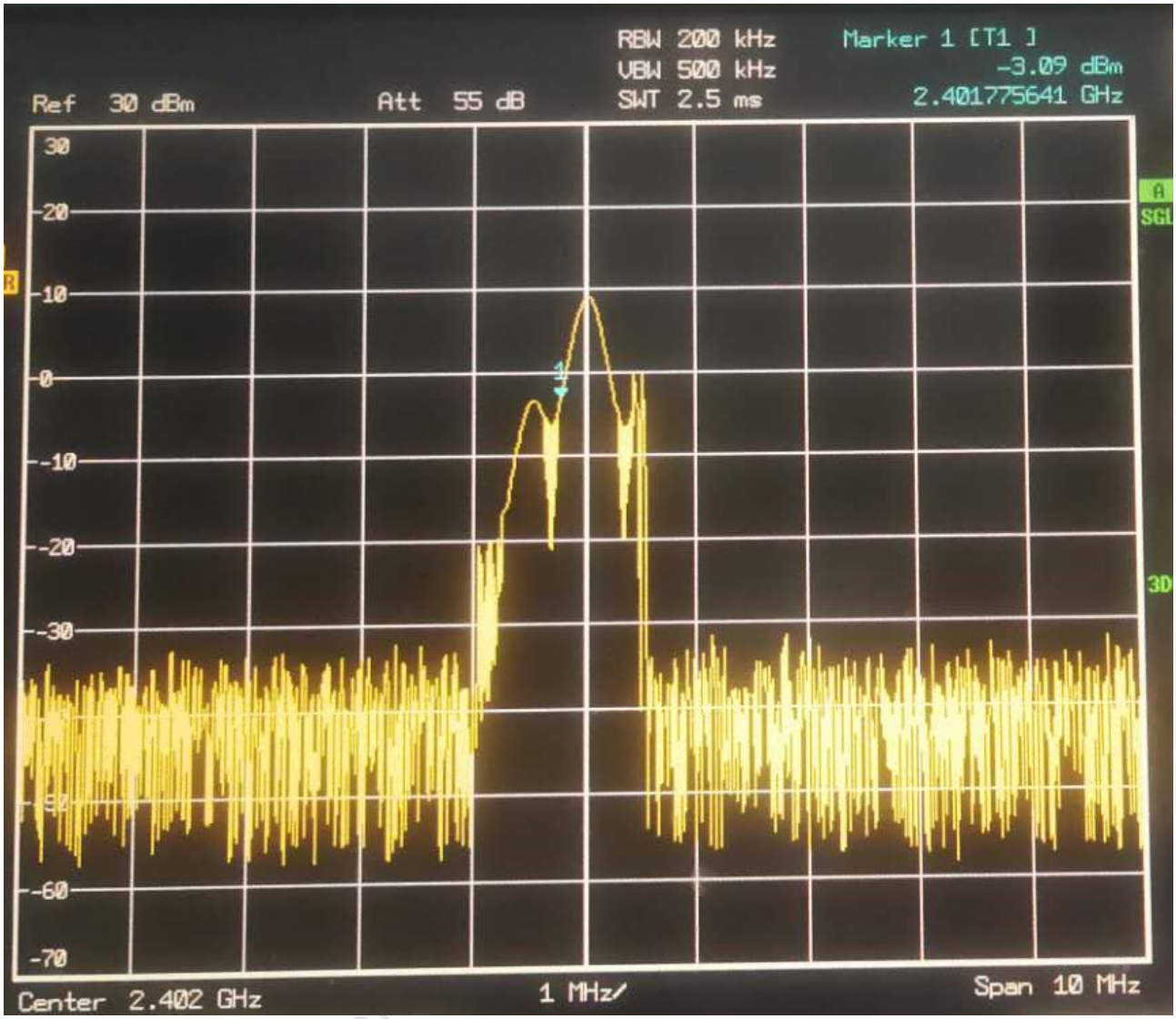


Figure 9.4: Burst 模式 2

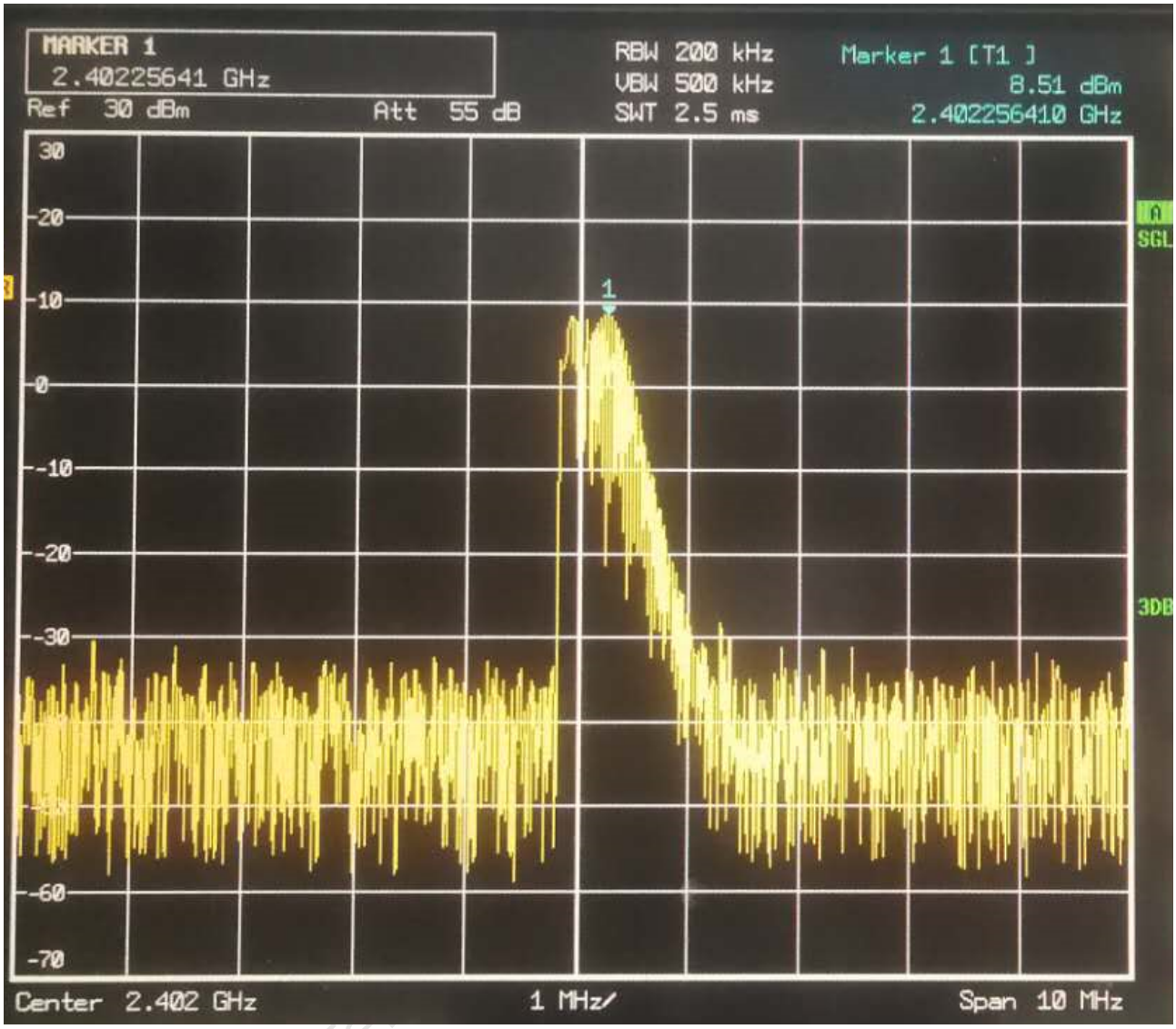


Figure 9.5: Burst 模式 3

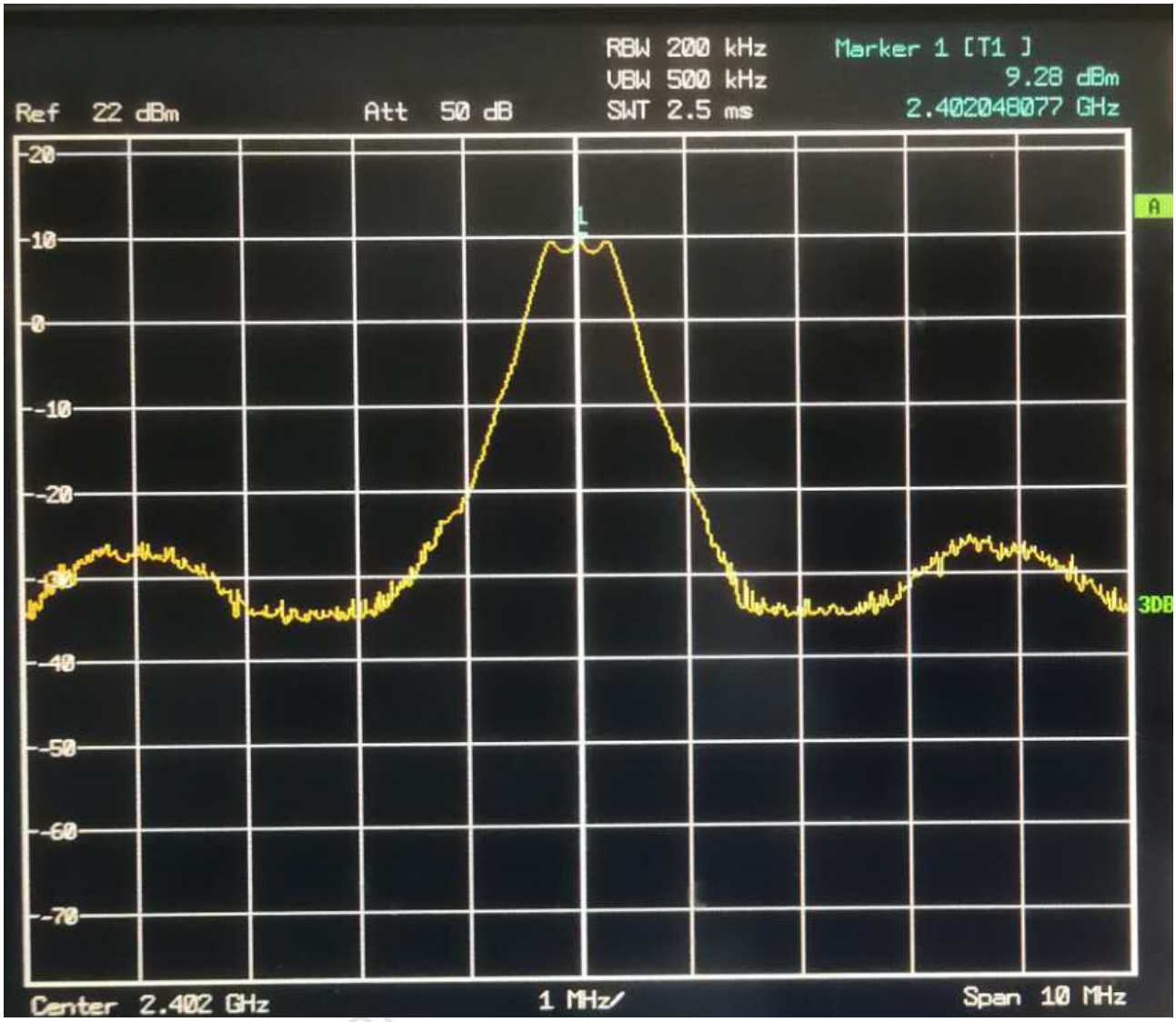


Figure 9.6: Burst 模式 4

9.24 RX **模式**

功能：收包模式下可设定通信方式进行收包，具体可配合非信令测试工具"Non_Signaling_Test_Tool"获取收包数和 RSSI。

实例说明：

测试工具：选用 Non_Signaling_Test_Tool

无线通信方式设定：Ble1M/Ble2M/Ble125K/Ble500K/Zigbee250K

通道设定：2402~2480MHz

10 Timer

SoC 支持 2 个定时器：Timer0、Timer1。

支持以下四种模式：System Clock Mode、GPIO Trigger Mode、GPIO Pulse Width Mode、Tick Mode。

还有一个 watchdog timer，配置为"watchdog"来监测系统运行，如果出现异常，则重启。

10.1 功能说明

当使用 Timer0、Timer1 定时器时，需要指定哪种模式，通过 timer_set_mode 函数接口设置：

```
void timer_set_mode(timer_type_e type, timer_mode_e mode,unsigned int init_tick, unsigned int
↪  cap_tick).
```

当使用 Timer0、Timer1 的 GPIO Trigger Mode，GPIO Pulse Width Mode 时，Timer 的时钟源是由 GPIO 提供的，需要通过如下接口进行设置，可以选择 gpio 以及极性，同时该接口中还会将对应的 GPIO 的相关中断 mask 也设置好，以便可以正常的产生 timer 中断。

```
void timer_gpio_init(timer_type_e type, gpio_pin_e pin, gpio_pol_e pol ).
```

10.1.1 System Clock Mode

时钟源	功能	机制说明
pclk	定时产生中断	设置为该模式后，每当检测到 pclk 的上升沿，则定时器的计数寄存器加 1，直到达到 capture value，产生中断，同时会自动装载 initial_tick，重新计数，当达到 capture value，再次进入中断，timer 使能不关，这样的操作就会一直循环进行。

设置步骤（如下代码所示）：

```
timer_set_mode(TIMER0, TIMER_MODE_SYSCLK, 0, 50*sys_clk.pclk*1000);
timer_start(TIMER0);
```

10.1.2 GPIO Trigger Mode

时钟源	功能	机制说明
由 GPIO 提供的	指定个数的 GPIO 上升沿/下降沿触发中断。	设置为该模式后，GPIO 每发生一个上升沿/下降沿，定时器就会计数加 1，当定时器的计数值达到指定的设定个数，就会进行中断，定时器清零重新开始计数，定时器使能中断不关，这样的操作就会一直循环进行。

设置步骤（如下代码所示）：

```
timer_gpio_init(TIMER0, SW1,POL_RISING);
timer_set_mode(TIMER0, TIMER_MODE_GPIO_TRIGGER,0,TIMER_MODE_GPIO_TRIGGER_TICK);
timer_start(TIMER0);
```

10.1.3 GPIO Pulse Width Mode

时钟源	功能	机制说明
pclk	捕获脉冲宽度	设置为该模式后，设置的 GPIO 如果检测到上升沿/下降沿，就会触发定时器定时，每一个 pclk，定时器就会计数加 1，当检测到该 GPIO 电平再次翻转就会进入中断，定时器计数停止，此时可以读取当前的 tick 计数值，去计算 gpio 脉冲的宽度，此中断触发一次，不会一直操作自动循环。

设置步骤（如下代码所示）：

```
timer_gpio_init(TIMER0, SW1, POL_FALLING);
timer_set_mode(TIMER0, TIMER_MODE_GPIO_WIDTH,0,0);
timer_start(TIMER0);
```

例子说明：如果极性设置为 POL_FALLING，则是下降沿触发计时，上升沿产生中断。

10.1.4 Tick Mode

时钟源	功能	机制说明
pclk	可作为时间指示器来用，该模式不会产生中断	设置为该模式后，每当检测到 system clock 的上升沿，定时器的计数器加 1，可以判断定时器的计数值是否达到指定的设定时间，手工将其定时器的初始计数值设为 0，定时器就会又从 0 重新开始定时。如果没有在指定的时间将定时器的初始计数值设为 0，那么定时器就会一直加 1，直到定时器计数溢出，自动将定时器的初始计数值设为 0，又开始计时，定时器就像钟表一样一直循环计时。

设置步骤（如下代码所示）：

```
timer_set_mode(TIMER0, TIMER_MODE_TICK,0,0);
timer_start(TIMER0);
```

10.1.5 Watchdog Mode

时钟源	功能	机制说明
pclk	在设定的时间内没有"喂狗"就会复位	设置为该模式后，看门狗开始计时，如果在指定的时间内没有喂狗，程序就会复位，喂狗函数为：wd_clear_cnt，该函数会清除计时，重新开始定时。如果不使用看门狗需要将看门狗关闭，以免程序复位。

设置步骤（如下代码所示）：

```
wd_set_interval_ms(1000,sys_clk.pclk*1000);
wd_start();
```

10.2 Demo **说明**

通过 Timer_Demo/app_config.h 中的宏 TIMER_MODE 选择使用哪一种模式。

```
#define TIMER_SYS_CLOCK_MODE      1
#define TIMER_GPIO_TRIGGER_MODE   2
#define TIMER_GPIO_WIDTH_MODE     3
```

10.2.1 GPIO System Clock Mode

Demo 设置：

Timer0，设置为 system clock mode，设置 initial_tick=0，capture value=50ms，使能 timer0。中断中会用 LED2 进行翻转。

执行结果：

LED2 会每 50ms 进行反转，结果如下：

通道 1（LED2）是中断标识 GPIO，大约 50ms 左右，反转一次。

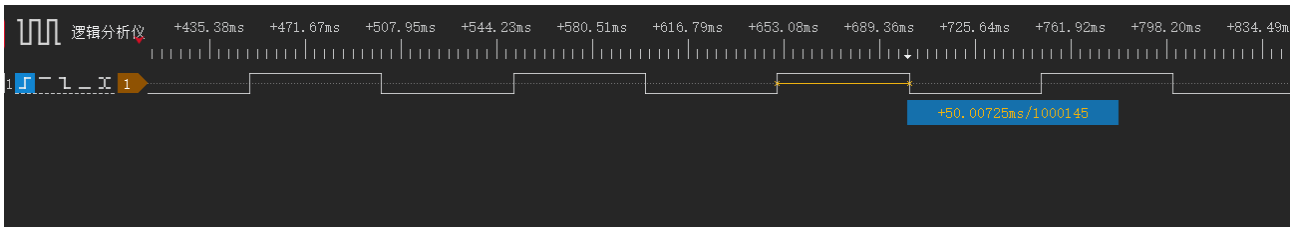


Figure 10.1: GPIO System Clock Mode

10.2.2 GPIO Trigger Mode

Demo 设置：

Timer0，设置为 gpio trigger mode，初始化 GPIO，将 SW1 配置为 timer0 的时钟源，上升沿触发，设置 initial_tick=0，capture value=0xf，使能 timer0。将 GPIO_PA2 和 SW1 两个引脚互连，GPIO_PA2 每 500ms 产生一个上升沿，达到 capture value，进入中断。中断中会用 GPIO_PB5 进行翻转。

执行结果：

GPIO_PA2 每产生 15 个上升沿，LED2 反转一次，结果如下：

通道 1（LED2）：是中断标识 GPIO，GPIO_PA2 每产生 15 个上升沿，LED2 产生一次翻转。

通道 0（GPIO_PA2）：是触发信号引脚 GPIO_PA2。

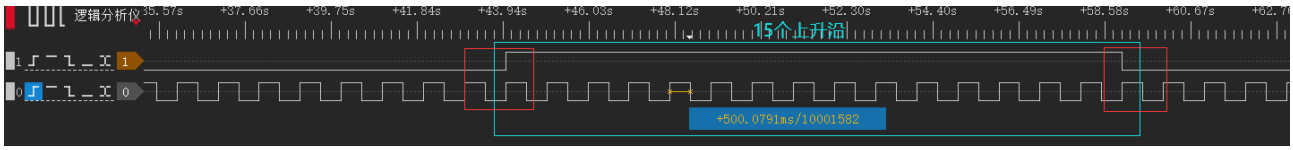


Figure 10.2: GPIO Trigger Mode

接下来，对红框进行详细说明：

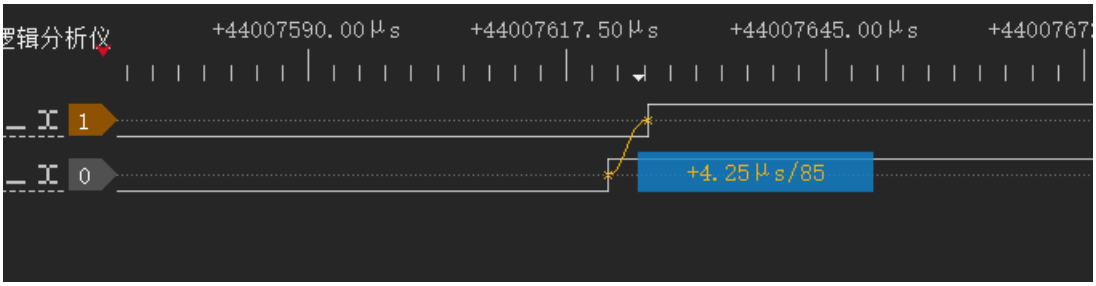


Figure 10.3: 红框 1

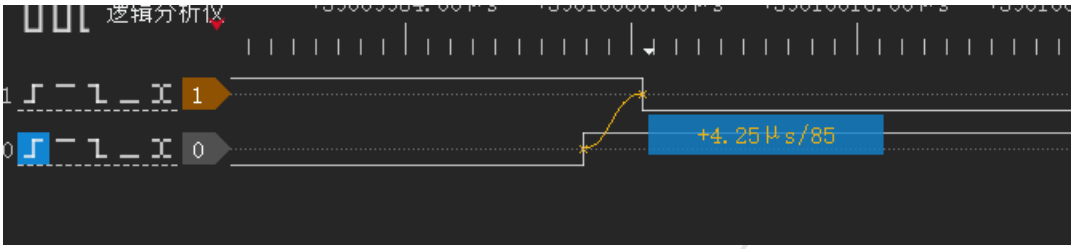


Figure 10.4: 红框 2

从红框 1、2 可以看出，当上一次中断的第 15 个上升沿发生以后，大约时延 4.25us，就会进入当前中断，LED2 翻转，当前 LED2 持续 15 个上升沿之后，大约时延 4.25us, 就会进入下一次中断。

10.2.3 GPIO Pulse Width Mode

Demo 设置：

Timer0，设置为 gpio pulse width mode，初始化 GPIO，将 SW1 配置为 timer0 的触发源，下降沿触发，设置 initial_tick=0，capture value=0，使能 timer0。将 GPIO_PA2 和 SW1 两个引脚互连，GPIO_PA2 产生下降沿，触发定时器计时，延时 250ms，GPIO_PA2 产生上升沿，进入中断。中断中会用 LED2 进行翻转。

执行结果：

GPIO_PA2 产生上升沿时，LED2 反转，结果如下：

通道 0（GPIO_PB4）：是触发源 GPIO_PA2 的波形。

通道 1（LED2）：是中断标识 GPIO，检测到 GPIO_PA2 产生上升沿，发生中断，LED2 产生翻转。

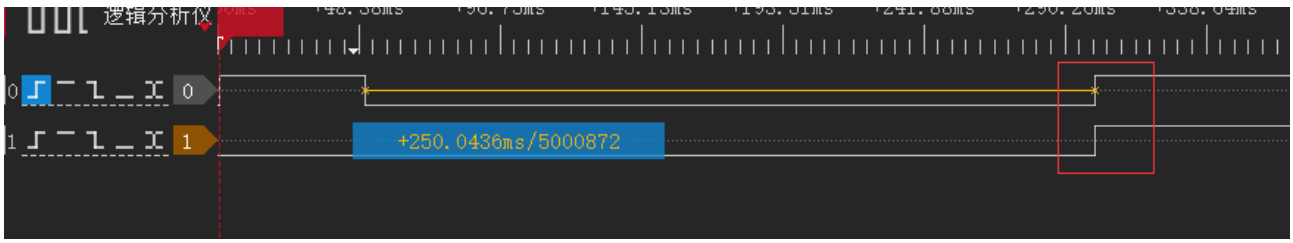


Figure 10.5: GPIO Pulse Width Mode

红框说明：

Figure 10.6: 红框说明

产生中断的时延 4.75us，CPU 进入中断需要一定的软件和硬件处理的时间。

读取定时器的计时寄存器，结果如下：

timer0_gpio_width	8001c	4	005b8e01

Figure 10.7: 计时寄存器

对 0x005b8e01 十六进制转化为十进制，结果是 6000129，使用的是系统时钟，频率为 24M，每 1/24M 秒计数一次，6000129*(1/24M) 计算得到大约是 250ms。

10.24 Tick Mode

Demo 设置：

Timer0，设置为 tick mode，设置 initial_tick=0，capture value=0，使能 timer0。每隔 500ms，手工设置定时器计时从头开始，LED2 翻转一次。

执行结果：

LED2 会每 500ms 进行反转，结果如下：

通道 1（LED2）：是 tick mode 标识 GPIO，大约 500ms 左右，反转一次。

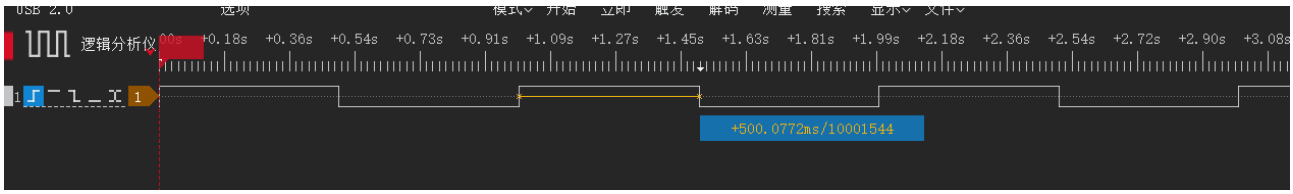


Figure 10.8: Tick Mode

10.2.5 Watchdog Mode

10.2.5.1 **喂狗测试**

Demo 设置：

设置看门狗时间，reg_wt_target=1000ms。每隔 990ms，喂狗，LED2 翻转一次。

执行结果：

LED2 会每隔 990ms 进行反转，结果如下：

通道 1（LED2）：是喂狗测试标识 GPIO，每隔 990ms 左右，反转一次。

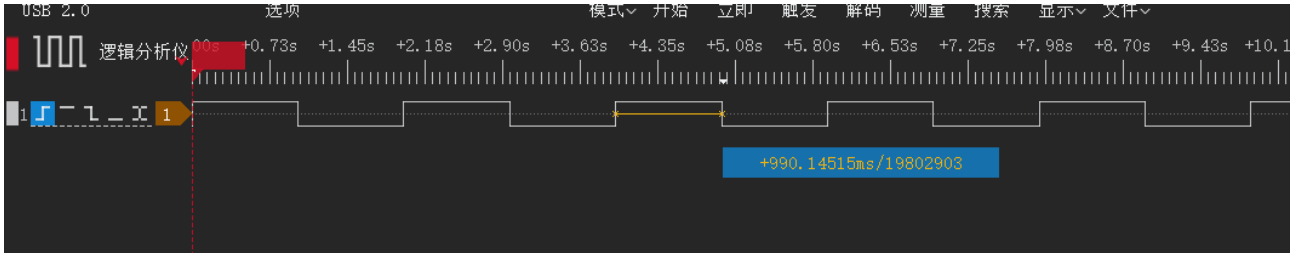


Figure 10.9: 喂狗测试

10.2.5.2 **不喂狗测试**

如果在看门狗设定的时间内没有喂狗：

Demo 设置：

```
delay_ms(990);
//wd_clear_cnt(); //将喂狗操作取消
gpio_toggle(LED2);
```

执行结果：

LED2 会周期性输出低电平 991.3847ms，高电平 10.0074ms 的波形，结果如下：

通道 1（LED2）：是不喂狗测试标识 GPIO。

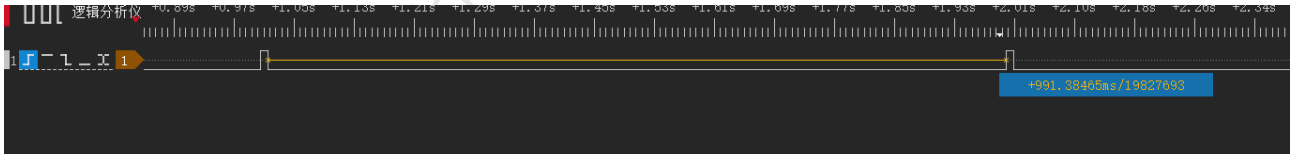


Figure 10.10: 不喂狗测试

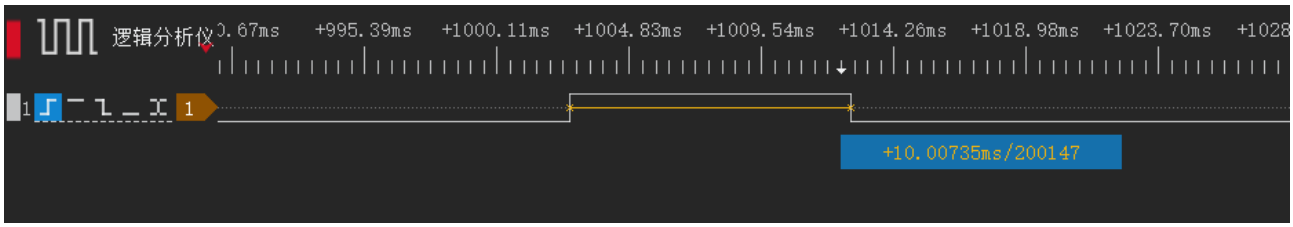


Figure 10.11: 不喂狗测试

结果说明：看门狗设定的时间为 1000ms，延时 990ms 之后，翻转 LED2，由于没有喂狗，LED2 高电平状态维持 10ms 之后，达到看门狗设定的时间，程序复位，一直重复以上的操作。

11 Analog

analog 驱动是用于读写模拟寄存器的，支持单个 Byte/Hword/Word，数据 Buffer、DMA 通道读写的功测试结果。

11.1 注意事项

使用 DMA 从模拟寄存器读数据到 Buffer 时，对应的目的地址的 Buffer 大小一定要是 4 的倍数。

原因：每次 DMA 都会送到 Buffer 4 个 Byte。即使配置读取长度不够 4，也会往目的地址写 4 个 Byte。

例如：定义数组 Buffer 大小为 5Byte，配置 DMA 从模拟寄存器读取 5Byte 到 Buffer，这时候 DMA 实际上传送了 2 次共 8 个 Byte 到 Buffer，多的 3 个 Byte 数据会从数组溢出，溢出的数据会覆盖别的变量。这时如果配置数组大小为 8Byte，多的 3 个 Byte 数据就会存放在数组中，不会溢出，避免了潜在的风险。

11.2 速度测试

在 cclk、pclk、hclk 都设置为 24MHz 条件下，加载各接口函数到 RAM 中运行，分别测得各模式读写 4byte 和 8byte 花费的时间如下表：

Mode	Write 4byte 时间	Read 4byte 时间
ALG_WORD_MODE	6us	12.9us
ALG_DMA_WORD_MODE	8.4us	10.2us
ALG_DMA_BUFF_MODE	8.4us	12.4us
ALG_BUFF_MODE	10us	12us
ALG_HWORD_MODE	12.8us	17.6us
ALG_DMA_ADDR_DATA_MODE	13.1us	不支持
ALG_BYTE_MODE	22.6us	26.8us

Mode	Write 8byte 时间	Read 8byte 时间
ALG_DMA_BUFF_MODE	11.2us	15.5us
ALG_BUFF_MODE	12.8us	16.6us
ALG_WORD_MODE	15.8us	19.3us
ALG_DMA_ADDR_DATA_MODE	18.9us	不支持
ALG_DMA_WORD_MODE	23.5us	20.1us
ALG_HWORD_MODE	24.8us	32.9us

Mode	Write 8byte 时间	Read 8byte 时间
ALG_BYTE_MODE	44.1us	53.9us

12 Flash

Flash 是一种非易失性（Non-Volatile）内存，在没有电流的供应下也能长久的保持数据，其存储特性相当于硬盘。Flash 可以对称为块的存储器单元进行擦写和再编程。任何 flash 器件的写入操作只能在空或者已擦除的单元内进行，所以在进行写入操作之前必须先执行擦除。

12.1 读操作

注意：

- 当读取地址超过 Flash 的最高地址，还是可以读到值，但是读的地址是按照有效地址位进行计算的。如 flash 大小是 1M bytes，则最大地址是 0xfffff，即低 20bit 是有效地址位，如果读取 0x100000 地址的值，则是读取 0x0 地址的值。

12.2 写操作

注意：

- 写之前必须要先擦除，最新的 flash_write_page 函数支持跨 page 写。

13 BQB

该章节介绍 BQB 测试 Demo 的使用方法。

13.1 功能描述

- 支持 2-wire 模式；
- 支持 BLE1M、BLE2M、BLE_LR_S2/S8；
- 支持长包（payload 为 255）；
- 支持通过写 Flash 来手动调整频偏值；
- 支持通过写 Flash 来手动调整串口的配置。

13.2 频偏值设置

Flash 地址 0x7e000（512K）、0xfe000（1M）、0x1fe000（2M）用来设置频偏值，写完频偏值之后通过复位来使设置生效。如果不写值（0xff）则为默认的频偏值。

13.3 通信验证

串口配置完成之后可以通过以下方式验证串口是否通信正常：

打开电脑端的串口工具，通过串口工具以 16 进制的格式向 Eagle 开发板发送“C0 00”，如果能够接受到开发板返回的“80 00”则说明通信正常。

14 PWM

14.1 PWM 简介

SoC 共有 6 路 PWM：PWM0~PWM5。

PWM 支持的相关模式介绍如下：

PWM0 支持的模式:

- continuous mode
- counting mode
- IR mode
- IR FIFO mode
- IR DMA FIFO mode

PWM1~PWM5 支持的模式:

- continuous mode

14.1.1 时钟

PWM 的时钟源是有两路，可以选择 pclk，也可以选择 32K，如下图所示：

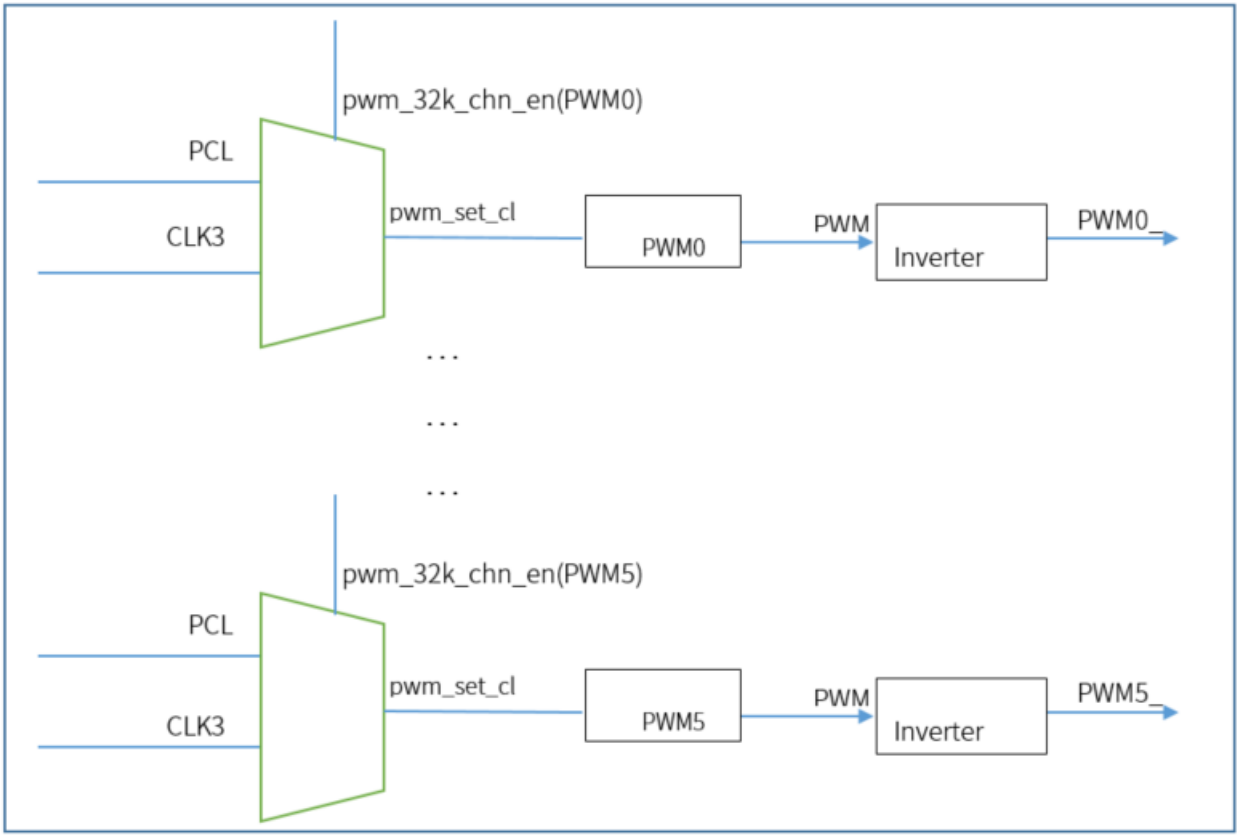


Figure 14.1: PWM 的时钟源

pclk：

功能：可以进行分频，然后分频后的时钟作为 PWM 使用的时钟源。

接口配置：static inline void pwm_set_clk(unsigned char pwm_clk_div)；

其中：pwm_clk_div = pclk_frequency /pwm_frequency -1;（pwm_clk_div：0~255）

32K：

功能：不支持分频，并且只支持 continuous mode、counting mode。该配置主要是为了实现 suspend 模式下也能发 PWM 波形。

接口配置：static inline void pwm_32k_chn_en(pwm_clk_32k_en_chn_e pwm_32K_en_chn)；

注意：

- 所有通道默认 pclk 时钟源，如果想使用 32K 时钟源，调用 pwm_32k_chn_en 使能对应通道，没有使能的通道仍是 pclk 时钟源。
- 32K 时钟源，PWM 设计的时候只考虑了 suspend 场景，在 continuous mode、counting mode 下使用中断，会提前一个 32K 时钟周期进入中断，在这 32K 时钟周期，会出了中断又会进入中断。使用 32K PWM 时如果需要中断，建议可以用 GPIO 中断来实现。
- 32K 时钟源时，在运行过程中如果需要更新占空比，只调用设置占空比的函数不会生效，必须设置好占空比后，再调用如下函数后，才会生效。

具体函数接口如下：

```
static inline void pwm_32k_chn_update_duty_cycle(void);
```

14.1.2 占空比

PWM 的一个 signal frame 由两个部分组成，分别为 Count status（高电平时间）和 Remaining status（低电平时间），一个信号帧的具体波形如下，其中 tmax 是周期时间。

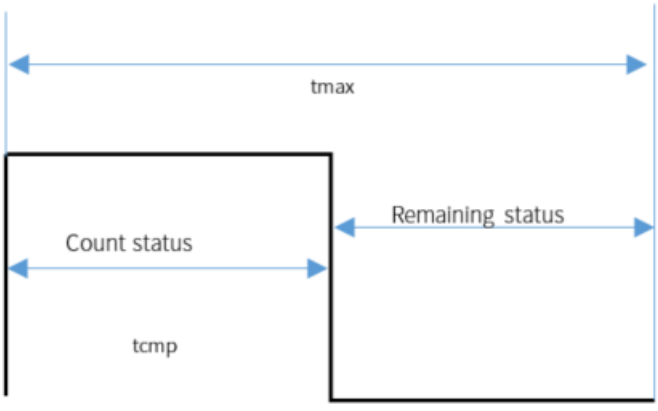


Figure 14.2: 信号帧的具体波形

驱动中设置 signal frame 周期和占空比的函数均使用的是 tcmp 和 tmax 作为参数。

a）通用的设置占空比的函数接口（支持所有通道）：

```
static inline void pwm_set_tcmp(pwm_id_e id, unsigned short tcmp);
static inline void pwm_set_tmax(pwm_id_e id, unsigned short tmax);
```

pwm_set_tcmp：

id：选择哪个 PWM 通道；

tcmp：设置高电平持续时间。

pwm_set_tmax：

id：选择哪个 PWM 通道；

tmax: 设置周期。

注意：

- 在 pwm_set_tmax 函数的第二参数设置 PWM 的周期，参数类型是 short 型，tmax 的最小取值为 1，不能为 0，如果为 0，则 pwm 处于不工作的状态，所以 tmax 的取值范围为：1~65535。
- 在 pwm_set_tcmp 函数的第二参数设置 PWM 的占空比，参数类型是 short 型，tcmp 的最小取值可以为 0，当为 0 的时候，这时 pwm 的波形一直是低电平，最大取值可以为 tmax，这时 pwm 的波形一直是高电平，所以 tcmp 的取值范围为：0~tmax。

b) 当使用 PWM0 的 IR FIFO Mode、IR DMA FIFO Mode 时，还会使用到另外一个函数接口：

```
static inline void pwm_set_pwm0_tcmp_and_tmax_shadow(unsigned short tmac_tick, unsigned short
↪  cmp_tick);
```

注意：

- 在 pwm_set_pwm0_tcmp_and_tmax_shadow 函数中，参数 tmac_tick 设定 pwm0 的周期，参数 cmp_tick 设定 pwm0 的高电平持续时间，tmac_tick 的取值范围：1 ~ 65536，cmp_tick 的取值范围：0 ~ cycle_tick。

c) 当使用 PWM0 的 counting mode、IR mode 时，pwm0 需要设定脉冲的输出个数功能，使用到的函数接口：

```
static inline void pwm_set_pwm0_pulse_num(unsigned short pulse_num);
```

pulse_num：脉冲的个数。

d) 当 pwm0 向 fifo 中写入 cfg data，使用到的函数接口：

```
static inline void pwm_set_pwm0_ir_fifo_cfg_data(unsigned short pulse_num, unsigned char
↪  use_shadow, unsigned char carrier_en);
```

use_shadow：

1：使用 pwm_set_pwm0_tcmp_and_tmax_shadow 函数下设置的周期和占空比。

0：使用 pwm_set_tmax、pwm_set_tcmp 函数下设置的周期和占空比。

carrier_en：

1：按照参数 pulse_num、use_shadow 的设置输出 pulse。

0：输出低电平，持续时间按照参数 pulse_num、use_shadow 进行计算。

14.1.3 Invert/polarity

(1) 通过 pwm_set_tcmp 和 pwm_set_tmax 设定的波形，默认情况下先输出高电平 Count status，后输出低电平 Remaining status。

(2) 如果使用 PWM*_PIN，输出的波形与通过 pwm_set_tcmp 和 pwm_set_tmax 设定的波形一致。

(3) 如果使用 PWM*_N_PIN，输出的波形与 PWM*_PIN 波形相反。

(4) 如果使用 pwm_invert_en 使能 PWM* 通道的 invert 功能，将会翻转 PWM*_PIN 波形。

(5) 如果使用 PWM* 通道的 invert 功能，将会翻转 PWM*_N_PIN 波形。

(6) 如果使用 pwm_n_invert_en 使能 PWM* 通道的 polarity 功能，所有的 PWM_PIN 都会按照如下规则输出：Count status 输出低，Remaining status 输出高。

14.2 功能说明

14.2.1 Continuous mode

该模式会一直持续按照设置的占空比发送 signal。如果想停止则设置 stop，设置之后则立刻停止。在发送期间，可以更新占空比，占空比后会在下一个 frame 生效。

14.2.2 Counting Mode

发送设置数量的 signal frame 则停止。该模式下如果 stop 会立刻停下。该模式下发送过程中，修改占空比，不会改变占空比。

14.2.3 IR Mode

IR mode 会连续发送 pulse groups。中间可以改变占空比，会在下一个 pulse group 生效。如果想立刻停止，可以直接 stop。IR mode 与 count 的区别，count 发送一个 pulse groups 就停止不再发送，而 IR mode 会持续不断地发送给 pulse groups。

如果想停止 IR mode 并想完成当前的 pulse group，则可以切换到 counting mode。

如果想立刻停止，可以直接 stop。

注意：

- 如果想停止 IR mode 并想完成当前的 pulse group，则在中断中可以切换到 counting mode，但在切换的过程中，会在当前 IR mode 模式下的 pulse group 发完，才会切换过来。

14.2.4 IR FIFO mode

在没有 MCU 的干预下，可以发送长代码模式，IR 的载波频率是由系统时钟分频得到的，可以支持通用频率，“Fifo cfg data”这个 element 作为 IR 波形的基础单元，硬件会解析 cfg 信息发出对应的 signal。

IR FIFO Mode 依次取出 FIFO 中的 cfg data 并发出对应 signal，直到 fifo 为空为止。在该模式下，可以使用 stop，但只是停止当前 cfg data 的执行，不影响 fifo 后面的 cfg data 执行。

注意：

- 在 IR FIFO 模式下，只要 fifo 有数据就会一直往外发送（自动发送），不需要 start 信号，同时 IR DMA FIFO Mode 也不需要 start 信号，但在其他模式下，都需要 pwm_start 信号。
- 每调用函数 pwm_set_pwm0_ir_fifo_cfg_data，则 FIFO 的 cnt 会加 1（如果这时 FIFO 为满，就会等待，直到 FIFO 不满，就会写入），硬件从 FIFO 中取出一个，则 FIFO 中的 cnt 会减 1。FIFO 的深度为 8bytes。数据从 FIFO 中取出后，执行发送 signal 动作，只有当前 signal 执行完之后，才会从 FIFO 中取出下一个。

14.2.5 IR DMA FIFO mode

IR DMA FIFO 模式，与 IR FIFO 模式相似，只是配置不是直接由 MCU 写在 FIFO 中，而是通过 DMA 写到 FIFO 中。

注意：

- 在中断里面需要更新 DMA 的部分配置：源地址的更新，DMA 触发等。
- 在该模式下, 与 IR FIFO 不同的地方，并不是 FIFO 中的 cfg data 数量为空的时候触发中断，而是将 fifo 中的配置 pwm 信号帧全部执行完才会触发中断。

14.3 中断

PWM 支持的中断设置说明如下（硬件不会自动清除中断标志位，需要软件手工清除）。

PWM0 支持的中断：

a) FLD_PWM0_FRAME_DONE_IRQ：每个 signal frame 完成，会产生中断。

b) FLD_PWM0_PNUM_IRQ：每发完一个 pulse groups，则会产生中断。

c) FLD_PWM0_IR_FIFO_IRQ：当 FIFO 里面的 cfg data 小于（不包括等于）设置的值（trigger_level）时，进入中断。

d) FLD_PWM0_IR_DMA_FIFO_IRQ：当 FIFO 执行完 DMA 发送的 cfg data 之后，进入中断。

PWM1 支持的中断：

a) FLD_PWM1_FRAME_DONE_IRQ：每个 signal frame 完成，会产生中断。

PWM2 支持的中断：

a) FLD_PWM2_FRAME_DONE_IRQ：每个 signal frame 完成，会产生中断。

PWM3 支持的中断：

a) FLD_PWM3_FRAME_DONE_IRQ：每个 signal frame 完成，会产生中断。

PWM4 支持的中断：

a) FLD_PWM4_FRAME_DONE_IRQ：每个 signal frame 完成，会产生中断。

PWM5 支持的中断：

a) FLD_PWM5_FRAME_DONE_IRQ：每个 signal frame 完成，会产生中断。

一个脉冲组（pluse groups）包含几个 frame 可以通过 pwm_set_pwm0_pulse_num 函数接口进行配置：

```
static inline void pwm_set_pwm0_pulse_num(unsigned short pulse_num).
```

IR FIFO 模式 trigger_level 的值可以通过 pwm_set_pwm0_ir_fifo_irq_trig_level 函数接口进行配置：

```
static inline void pwm_set_pwm0_ir_fifo_irq_trig_level(unsigned char trig_level).
```

注意：

- 响应中断时，存在时延，时间大约在 2~4us 左右。

144 Continuous mode

144.1 功能说明

该 demo 实现的功能如下：LED1 会持续发送 signal frame（高电平时间为 50us，周期为 100us），同时，每产生一次中断，LED4 会翻转一次（该 GPIO 是为了做中断测试设置的一个标识信号）。

144.2 实例结果

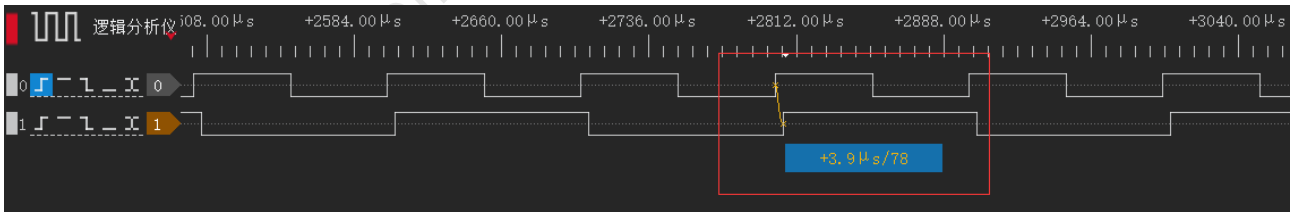


Figure 14.3: Continuous mode 实验结果

上图为使用逻辑分析仪抓到的实验结果：

通道 0（LED1）：是 PWM 输出信号。

通道 1（LED4）：是中断标识 GPIO，每发送一个信号帧，产生一次中断。

红框说明：产生中断的时延，CPU 进入中断需要一定的软件和硬件处理的时间。

144.3 其他验证结果

144.3.1 Stop

使用下面实验验证 continuous mode 下，执行 stop 后，signal 会立刻停止。

代码实现如下，在 stop 之后，反转 LED3 的状态。

```
pwm_start(PWM_ID);
delay_ms(1);
pwm_stop(PWM_ID);
gpio_toggle(LED3);
```

使用逻辑分析仪抓到的实验结果：

通道 0（LED1）：是 PWM 输出信号。

通道 1（LED3）：是 stop 标识 GPIO。

由图中可以看到，LED3 翻转后，PWM 的 signal 立刻停止。

Figure 14.4: 其他验证结果

144.3.2 占空比

使用下面实验验证 continuous mode 下，在发送期间，可以更新占空比，更新占空比后会在下一个 frame 生效。代码实现如下，在修改占空比的状态之后，反转 LED3 的状态。

```
pwm_start(PWM_ID);
delay_ms(1);
pwm_set_tcmp(PWM_ID,10 * CLOCK_PWM_CLOCK_1US);
gpio_toggle(LED3);
```

使用逻辑分析仪抓到的实验结果：

通道 0（LED1）：是 PWM 输出信号。

通道 1（LED3）：是占空比更新标识 GPIO。

由图中可以看到，LED3 翻转后，PWM 修改占空比后，会在下一个 frame 生效。

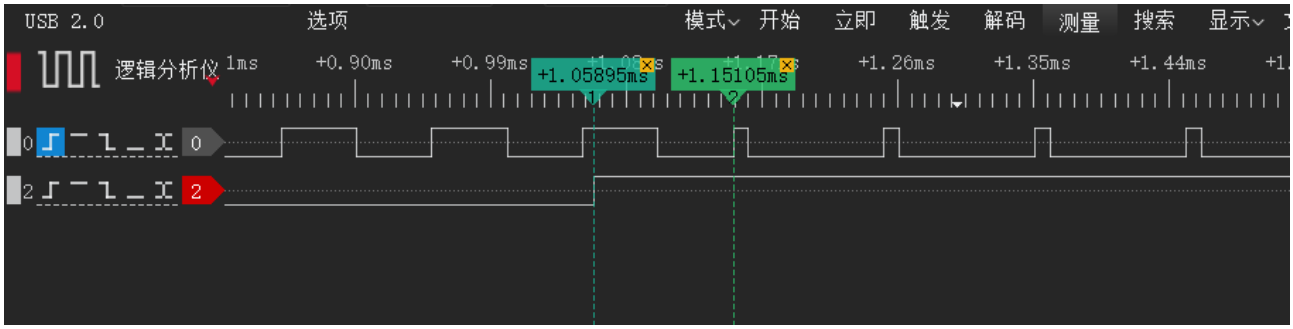


Figure 14.5: 更新占空比

14.5 Counting mode

通过使用 app_pwm_count.c 中的宏选择哪一种中断方式。

```
#define COUNT_FRAME_INIT 1
#define COUNT_PNUM_INIT 2
#define SET_COUNT_INIT_MODE COUNT_FRAME_INIT
```

14.5.1 COUNT_FRAME_INIT

14.5.1.1 **功能说明**

该 demo 实现的功能如下，LED1 会输出数量为 16 的 signal frame（高电平时间为 50us，周期为 100us），同时，每发送一个信号帧，产生一次中断，LED4 会翻转一次（该 GPIO 是为了做中断测试设置的一个标识信号）。

14.5.1.2 **实例结果**

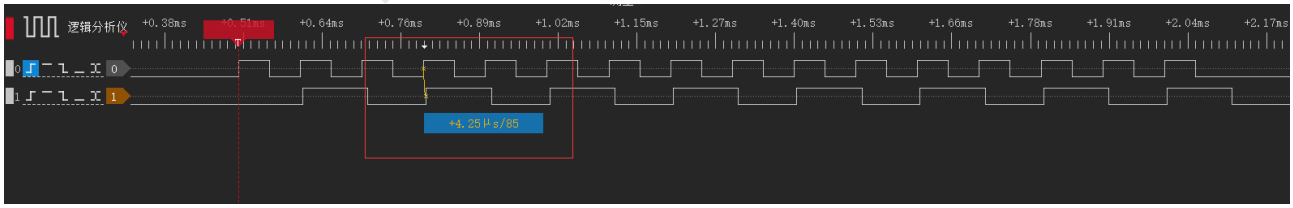


Figure 14.6: COUNT_FRAME_INIT 实例

通道 0（LED1）：是 PWM 输出信号。

通道 1（LED4）：是中断标识 GPIO，每发送一个信号帧，产生一次中断。

红框说明：产生中断的时延，CPU 进入中断需要一定的软件和硬件处理的时间。

14.5.2 COUNT_PNUM_INIT

14.5.2.1 **功能说明**

该 demo 实现的功能如下：LED1 会输出数量为 16 的 signal frame（高电平时间为 50us，周期为 100us），当发送完指定的脉冲个数之后，产生中断，LED4 翻转（该 GPIO 是为了做中断测试设置的一个标识信号）。

14.5.2.2 **实例结果**

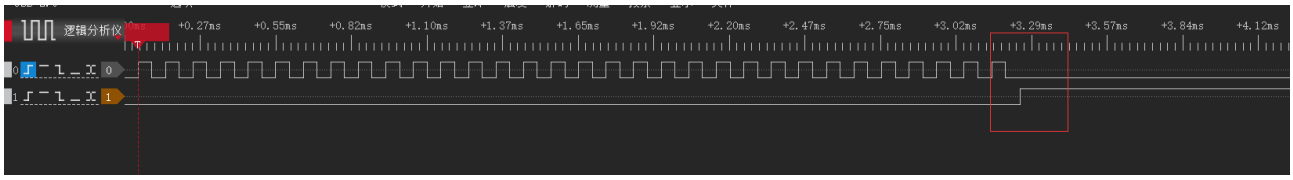


Figure 14.7: COUNT_PNUM_INIT 实例

红框的具体情况如下：

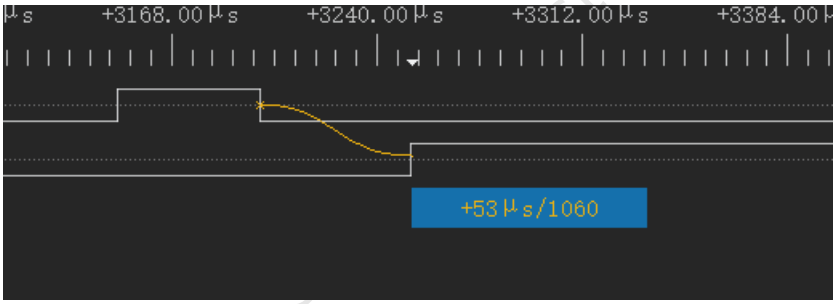


Figure 14.8: 红框具体情况

通道 0（LED1）：是 PWM 输出信号。

通道 1（LED4）：是中断标识 GPIO，指定脉冲个数发完，产生中断。

红框说明：比 50us 多 3us，说明进入中断有一定的时延，CPU 进入中断需要一定的软件和硬件处理的时间。

14.5.3 **其他验证结果**

14.5.3.1 Stop

使用下面实验验证 counting mode 下，执行 stop 后，signal 会立刻停止。

代码实现如下，在 stop 之后，反转 LED3 的状态。

```
pwm_start(PWM_ID);
delay_ms(1);
pwm_stop(PWM_ID);
gpio_toggle(LED3);
```

使用逻辑分析仪抓到的实验结果：

通道 0（LED1）：是 PWM 输出信号。

通道 1（LED3）：是 stop 标识 GPIO。

由图中可以看到，LED3 翻转后，PWM 的 signal 立刻停止。

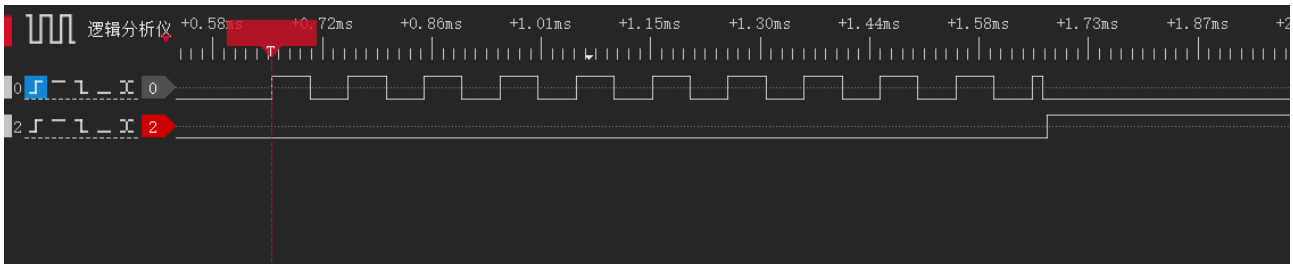


Figure 14.9: 其他验证结果

14.5.3.2 占空比

使用下面实验验证 counting mode 下，在发送期间，更改占空比，占空比不可以改变。

代码实现如下，在修改占空比的状态之后，反转 LED3 的状态。

```
pwm_start(PWM_ID);
delay_ms(1);
pwm_set_tcmp(PWM_ID,10 * CLOCK_PWM_CLOCK_1US);
gpio_toggle(LED3);
```

使用逻辑分析仪抓到的实验结果：

通道 0（LED1）：是 PWM 输出信号。

通道 1（LED3）：是占空比更新标识 GPIO。

由图中可以看到，LED3 翻转后，PWM 的 signal 没有发生改变。

实验结果如下：

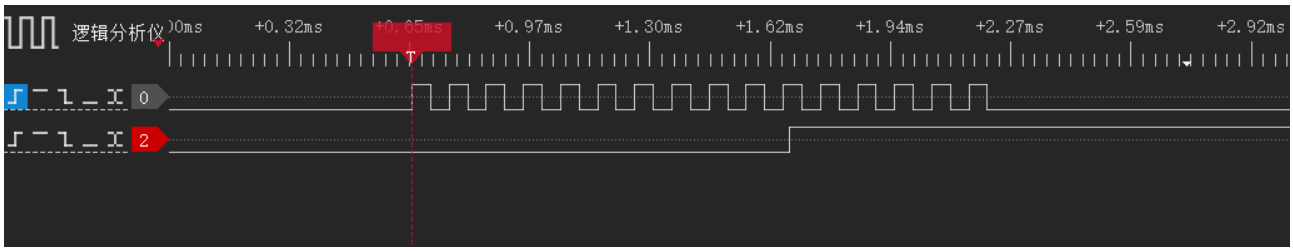


Figure 14.10: 更改占空比

14.6 IR mode

14.6.1 功能说明

该 demo 实现的功能如下，LED1 会输出 6 个脉冲组（每个脉冲组脉冲个数为 4，高电平时间为 50us，周期为 100us），同时，每发完一个 pulse groups，产生一次中断，LED4 会翻转一次（该 GPIO 是为了做中断测试设置的一个标识信号）。

14.6.2 实例结果

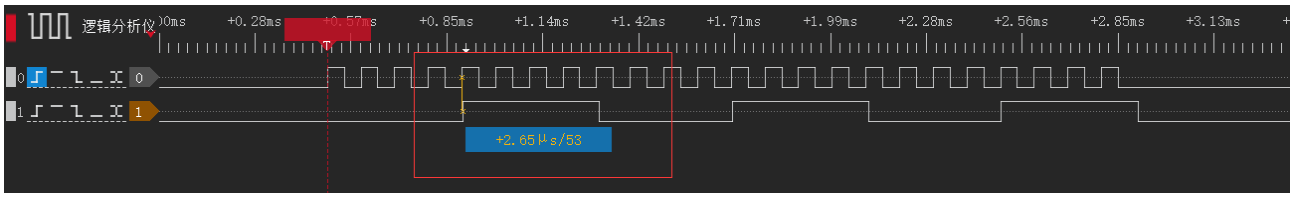


Figure 14.11: IR mode 实例

上图为使用逻辑分析仪抓到的实验结果：

通道 0（LED1）：是 PWM 输出信号。

通道 1（LED3）：是中断标识 GPIO，一个脉冲组发完翻转一次。

红框说明：产生中断的时延，CPU 进入中断需要一定的软件和硬件处理的时间。

14.6.3 其他验证结果

14.6.3.1 Stop

使用下面实验验证 IR mode 下，执行 stop 后，signal 会立刻停止。

代码实现如下，在 stop 之后，反转 LED3 的状态。

```
pwm_start(PWM_ID);
delay_ms(1);
pwm_stop(PWM_ID);
gpio_toggle(LED3);
```

使用逻辑分析仪抓到的实验结果：

通道 0（LED1）：是 PWM 输出信号。

通道 1（LED3）：是 stop 标识 GPIO。

由图中可以看到，LED3 翻转后，PWM 的 signal 立刻停止。

Figure 14.12: 其他验证结果

14.6.3.2 **占空比**

使用下面实验验证 IR mode 下，中间可以改变占空比，但会在当前 pulse group 执行完生效。

代码实现如下：

```
pwm_start(PWM_ID);
delay_ms(1);
pwm_set_tcmp(PWM_ID,10 * CLOCK_PWM_CLOCK_1US);
gpio_toggle(LED3);
```

使用逻辑分析仪抓到的实验结果：

通道 0（LED1）：是 PWM 输出信号。

通道 1（LED3）：是占空比更新标识 GPIO。

由图中可以看到，LED3 翻转后，pwm 的 signal 并没有立刻改变，而是在当前 pulse group 执行完生效。

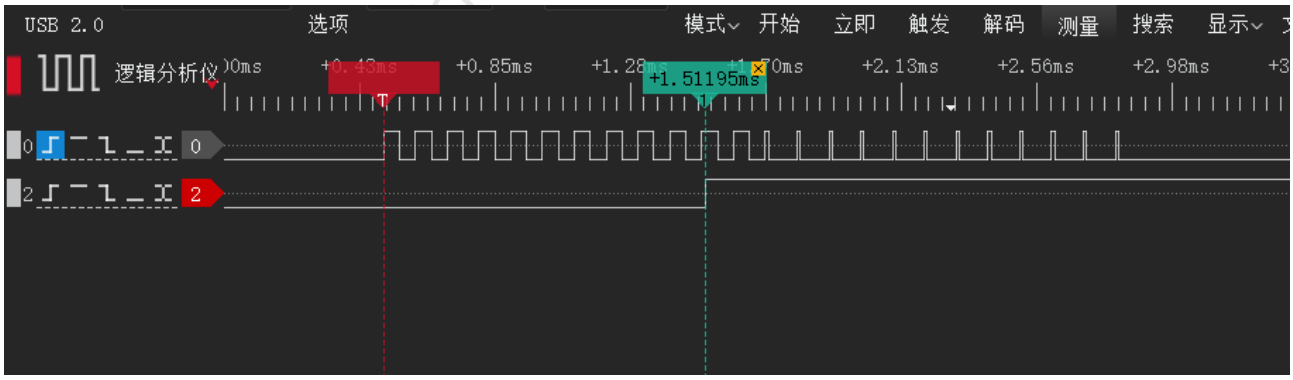


Figure 14.13: 更改占空比

14.7 IR FIFO Mode

14.7.1 **功能说明**

IR FIFO MODE 的 demo 的大致流程介绍：一开始先向 FIFO 写入两组 cfg data1, cfg data2, 当 FIFO 里面的 cfg data 小于（不包括等于）设置的值（trigger_level 为 1）进入中断，在中断配置了相同的两组 cfg data1, cfg

data2。

该 demo 实现的功能如下，LED1 以两组脉冲组为单位依次不断地发送，两组脉冲组的设置如下：

(1) cfg data1, high level time of 50us, period of 100us

(2) cfg data2, high level time of 100us, period of 200us

每产生一次中断，LED4 会翻转一次（该 GPIO 是为了做中断测试设置的一个标识信号）。

14.7.2 **实例结果**

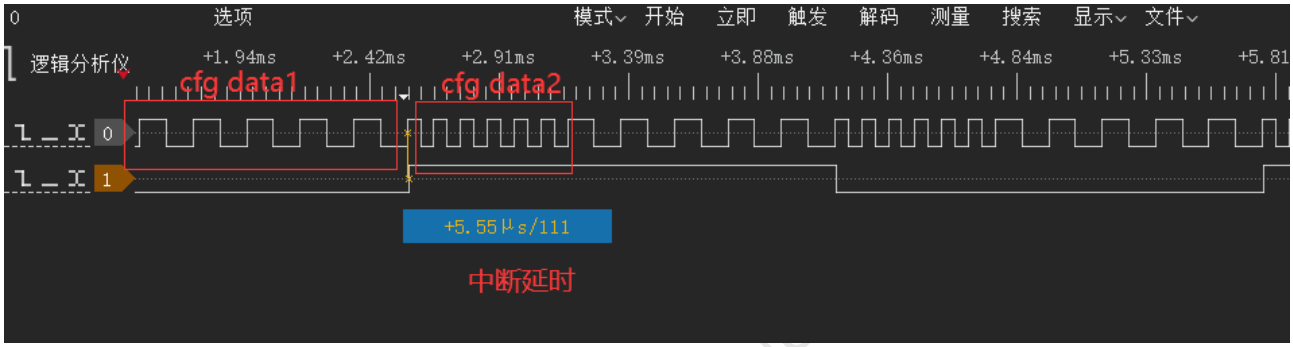


Figure 14.14: IR FIFO mode 实例

通道 0（LED1）：是 PWM 输出信号。

通道 1（LED4）：是中断标识 GPIO，当 FIFO 里面的 cfg data 小于（不包括等于）设置的值（trigger_level 为 1）的时候翻转一次（执行完 cfg data1，从 FIFO 取出 cfg data2 时，cnt 的值为 0，小于 trigger level(值为 1)，进入中断）。

红框说明：产生中断的时延，CPU 进入中断需要一定的软件和硬件处理的时间。

14.7.3 **其他验证结果**

14.7.3.1 Stop

使用下面实验验证 IR FIFO mode 下，执行 stop 后，只是停止当前 cfg data 的执行，不影响 fifo 后面的 cfg data 执行。

代码实现如下，在 stop 之后，反转 LED3 的状态。

```
delay_ms(10);
pwm_stop(PWM_ID);
gpio_toggle(LED3);
```

使用逻辑分析仪抓到的实验结果：

通道 0（LED1）：是 PWM 输出信号。

通道 1（LED3）：是 stop 标识 GPIO。

在 IR FIFO MODE 功能说明中：

cfg data1：脉冲个数为 5，高电平时间为 50us，周期为 100us。

cfg data2：脉冲个数为 6，高电平时间为 100us，周期为 200us。

由图中可以看到，执行 stop 后，LED3 反转，停止当前 cfg data1 的执行，不影响 fifo 后面的 cfg data2 的执行。

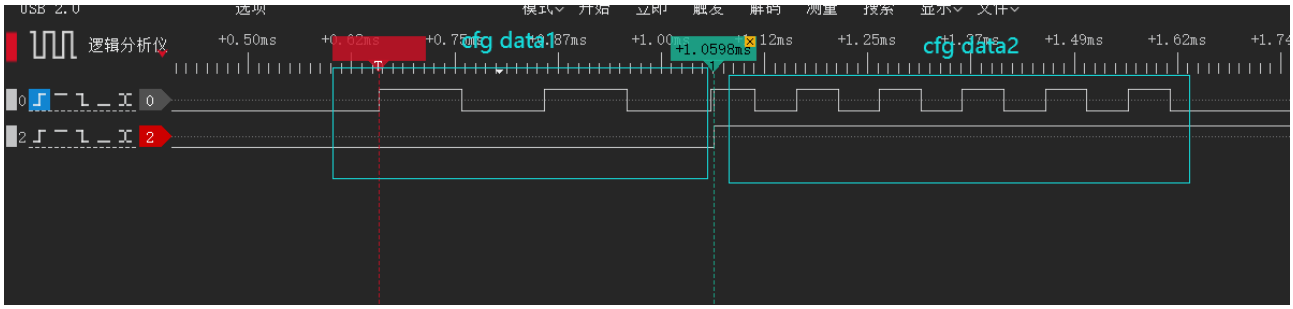


Figure 14.15: 其他验证结果

14.8 DMA FIFO mode

通过使用 app_pwm_ir_dma.c 中的宏选择工作方式。

```
#define     PWM_IR_FIFO_DMA     1
#define     PWM_CHAIN_DMA       2
#define     SET_PWM_DMA_MODE    PWM_IR_FIFO_DMA
```

14.8.1 PWM_IR_FIFO_DMA

在 IR DAM FIFO Mode 中，需要不断触发中断请求 DMA 将 cfg data 发送到 FIFO 中。

IR DMA FIFO Mode 程序大致流程：在主程序中通过 DMA 请求，向 FIFO 中传送三组 cfg data，分别为 cfg data1、cfg data2、cfg data3，当 FIFO 中的 cfg data 全部执行完之后，进入中断，在中断通过 DMA 请求，向 FIFO 中传送两组 cfg data，分别为 cfg data4、cfg data5。

14.8.1.1 **功能说明**

该 demo 实现的功能如下，LED1 一开始会发送一个三组脉冲组：

cfg data1：脉冲个数为 5，高电平时间为 50us，周期为 100us。

cfg data2：脉冲个数为 4，高电平时间为 50us，周期为 100us。

cfg data3：脉冲个数为 6，高电平时间为 100us，周期为 200us。

接下来，会以两组脉冲为单位不断发送两组脉冲组：

cfg data4：脉冲个数为 4，高电平时间为 50us，周期为 100us。

cfg data5：脉冲个数为 4，高电平时间为 50us，周期为 100us。

同时，每产生一次中断，LED4 会翻转一次（该 GPIO 是为了做中断测试设置的一个标识信号）。

14.8.1.2 **实例结果**

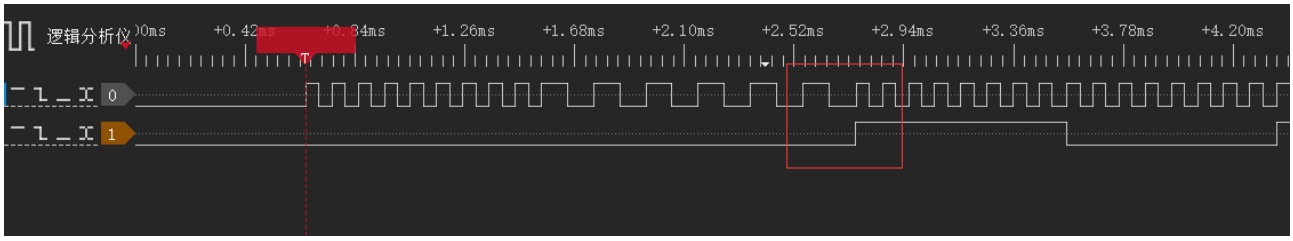


Figure 14.16: PWM_IR_FIFO_DMA 实例

通道 0（LED1）：是 PWM 输出信号。

通道 1（LED4）：是中断标识 GPIO。

从上图可以看出，在该模式下，当 FIFO 的 cfg data 数据全部执行完才触发中断，与 IR FIFO Mode 的中断机制不一样的。

下图为上图中红色框的放大框，从图中可以看出 cfg data3 的最后一个低电平的维持时间是 105us（并不是 cfg data3 设置的 100us），所以可以看出，在这种使用方法下，发完第一组 DMA 数据后，中断中重新触发 DMA，在 signal 上是有一段时延的。（这个实例相当于实际输出是这样的：cfg1+cfg2+cfg3+delay(低电平 5us 左右)+cfg4+cfg5），这 5us 的时间包括进入中断以及重新设置相关设置的时间。

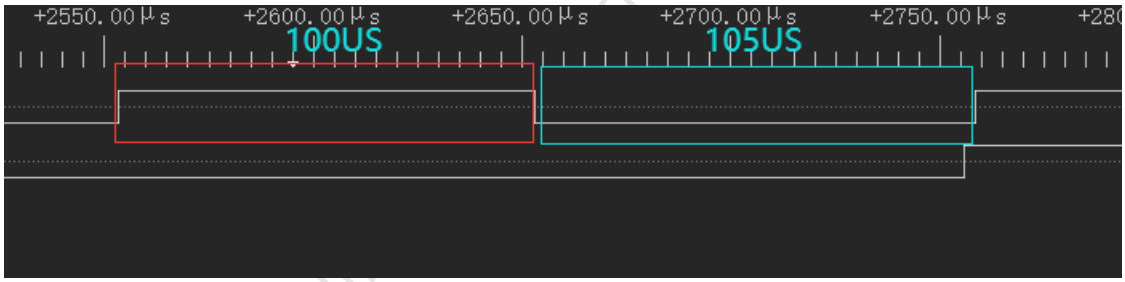


Figure 14.17: PWM_IR_FIFO_DMA 实例放大框

14.8.2 PWM_CHAIN_DMA

SoC 芯片的 DMA 有一种链表形式，可以结合使用到 DMA FIFO mode 中，这个的好处就是，可以没有 MCU 干预，一直反复的发送想要的 signal。在前面的 IR DAM FIFO Mode 例子中，需要不断触发中断请求 DMA 将 cfg data 发送到 FIFO 中，使用链表的方法在不需要中断参与的情况下，也可以完成连续发送的功能。

在 pwm_chain_dma 的例子中，链表结构如下：

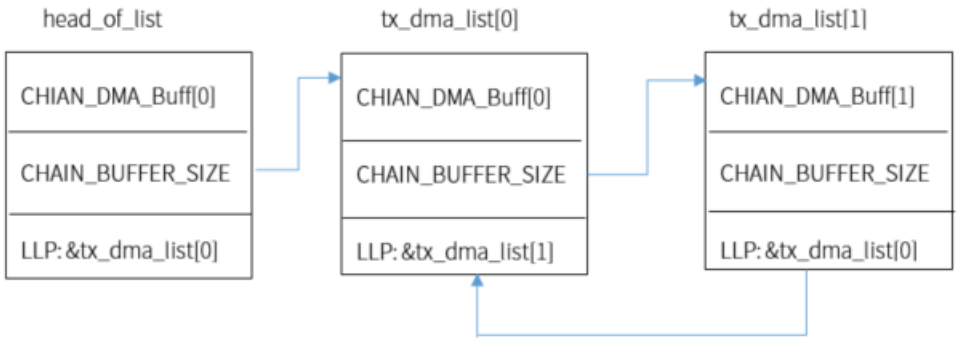


Figure 14.18: PWM_CHAIN_DMA 链表结构

先建立头结点 head_of_list，然后向循环链表添加结点，在 demo 中，添加了两个循环链表结点，分别是 tx_dma_list[0] 和 tx_dma_list[1]。

注意：

- 头结点需要配置 DMA 源地址、DMA 长度、下一个结点配置所在地址，不能仅仅配置下一个结点配置所在的地址。

从流程图可以看出，先开始执行头指针，然后执行 tx_dma_list[0]，然后执行 tx_dma_list[1]，再执行 tx_dma_list[0]，依次循环执行，直到 LLP 设置为 0 才会停止。

这只是实现两个循环链表通过 DMA 实现不断连续发送，如果想实现更多数组循环发送，可以按照以上的结构添加相对应的指针结构体即可。

具体 DMA 链表配置如下：

```
pwm_set_dma_config(DMA_CHN);
pwm_set_dma_chain_llp(DMA_CHN,(u16*)(&CHIAN_DMA_Buff[0]),MIC_BUFFER_SIZE,&tx_dma_list[0]);
pwm_set_tx_dma_add_list_element(DMA_CHN,&tx_dma_list[0],&tx_dma_list[1],(u16*)
 ↪  (&CHIAN_DMA_Buff[0]),CHAIN_BUFFER_SIZE);
pwm_set_tx_dma_add_list_element(DMA_CHN,&tx_dma_list[1],&tx_dma_list[0],(u16*)
 ↪  (&CHIAN_DMA_Buff[1]),CHAIN_BUFFER_SIZE)
pwm_ir_dma_mode_start(DMA_CHN);
```

DMA 链表与 PWM_IR_FIFO_DMA 的不同在于：

pwm_set_dma_chain_llp 函数和 pwm_set_tx_dma_add_list_element 函数。

通过 pwm_set_dma_chain_llp 函数设置链表头：

```
void pwm_set_dma_chain_llp(dma_chn_e chn,u16 * src_addr, u32 data_len,dma_chian_config_t *
 ↪  head_of_list)
```

chn：DMA 配置。

src_addr：DMA 源地址，即 cfg data 数组。

data_len：DMA 长度。

head_of_list：下一个结点配置的所在地址。

每个结点中包括的内容：DMA 配置、当前结点配置的所在地址、下一个结点配置所在地址、DMA 源地址，DMA 长度，通过 pwm_set_tx_dma_add_list_element 函数设置循环链表的结点：

```
void pwm_set_tx_dma_add_list_element(dma_chn_e chn, dma_chian_config_t *config_addr,
↪  dma_chian_config_t *llponit , u16 * src_addr, u32 data_len)
```

chn：DMA 配置。

config_addr：当前结点配置的所在地址。

llponit：下一个结点配置的所在地址。

src_addr：DMA 源地址，即 cfg data 数组。

data_len：DMA 长度。

14.8.2.1 功能说明

该 demo 实现的功能如下，LED1 一开始会先发送头结点配置的 cfg data 数组，然后会以结点 1 配置的 cfg data 数组和结点 2 配置的 cfg data 数组交替循环执行不断发送。

结点 1 配置的 cfg data 情况如下：

cfg data1：脉冲个数为 5，高电平时间为 100us，周期为 200us。

cfg data2：脉冲个数为 4，高电平时间为 100us，周期为 200us。

cfg data3：脉冲个数为 6，高电平时间为 100us，周期为 200us。

cfg data4：脉冲个数为 3，高电平时间为 100us，周期为 200us。

结点 2 配置的 cfg data 情况如下：

cfg data 5：脉冲个数为 5，高电平时间为 50us，周期为 100us。

cfg data6：脉冲个数为 4，高电平时间为 50us，周期为 100us。

cfg data7：脉冲个数为 6，高电平时间为 50us，周期为 100us。

cfg data8：脉冲个数为 3，高电平时间为 50us，周期为 100us。

14.8.3 实例结果

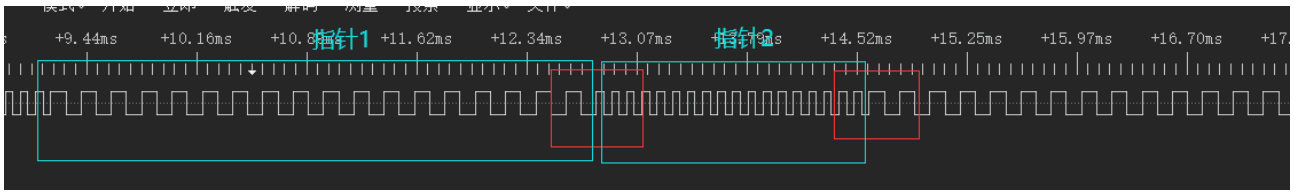


Figure 14.19: PWM_CHAIN_DMA 实例

通道 0（LED1）：是 PWM 输出信号。

对红框进行详细说明，指针 1 和指针 2 的 cfg data 在切换的过程中，没有时延的产生。

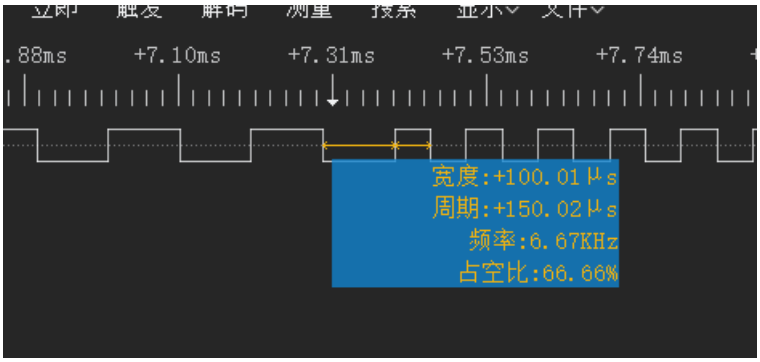


Figure 14.20: 红框详细说明 1

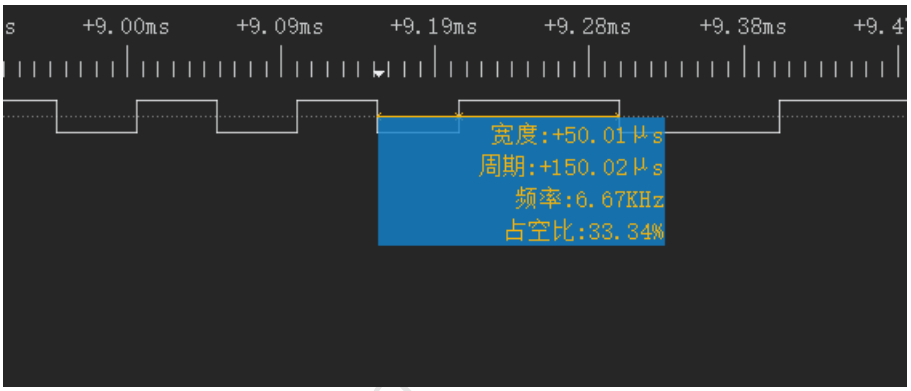


Figure 14.21: 红框详细说明 2

由图可得，没有延时的产生，所以使用链表的方式可以发出连续的 PWM 波形。

15 I2C

15.1 简介

I2C，由数据线 SDA 和时钟 SCL 构成的串行总线，可发送和接收数据，是半双工通信方式。时钟是由 master 端提供控制的。I2C 通信协议具体如下：

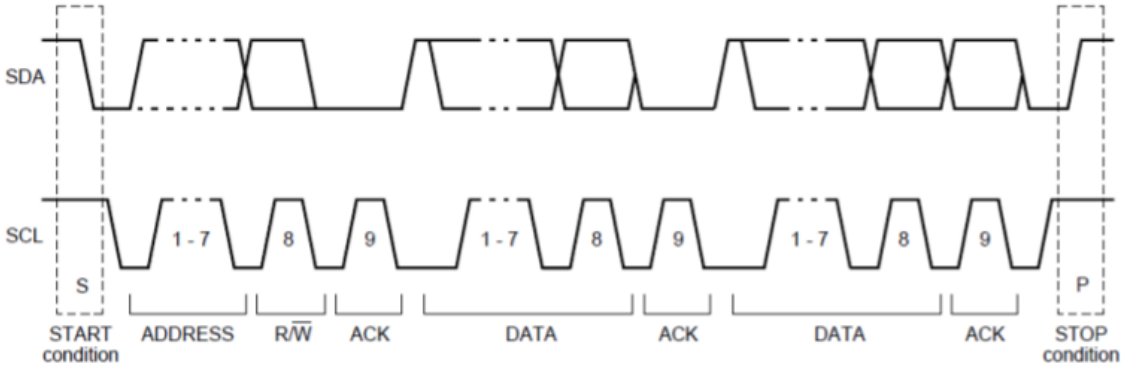


Figure 15.1: I2C 通信协议

I2C 通信协议具体介绍如下：

状态	过程
空闲状态	I2C 总线的 SDA 和 SCL 两条信号线同时处于高电平时，规定为总线的空闲状态。
开始信号	当 SCL 为高期间，SDA 由高到低的跳变。
停止信号	当 SCL 为高期间，SDA 由低到高的跳变。
应答信号	对于反馈有效应答位 ACK 的要求是，接收器在第 9 个时钟脉冲之前的低电平期间将 SDA 线拉低，并且确保在该时钟的高电平期间为稳定的低电平。如果接收器是主控器，则在它收到最后一个字节后，发送一个 NACK 信号，以通知被控发送器结束数据发送，并释放 SDA 线，以便主控接收器发送一个停止信号 P。
数据的有效性	I2C 总线进行数据传送时，时钟信号为高电平期间，数据线上的数据必须保持稳定，只有在时钟线上的信号为低电平期间，数据线上的高电平或低电平状态才允许变化。即：数据在 SCL 的上升沿到来之前就需准备好。并在在下降沿到来之前必须稳定。
数据传输	在 I2C 总线上传送的每一位数据都有一个时钟脉冲相对应（或同步控制），即在 SCL 串行时钟的配合下，在 SDA 上逐位地串行传送每一位数据。

15.2 中断

I2C 模式下的中断相关介绍：

I2C no-DMA 模式：

相关中断：

a) I2C_RX_BUF_MASK：与 rx_irq_trig_lev 设置相关，当 fifo_data_cnt>=rx_irq_trig_lev 产生中断。

b) I2C_RX_DONE_MASK：当接收完发送的数据之后，产生中断。

是否手动清除中断标志位：

I2C_RX_BUF_MASK、I2C_RX_DONE_MASK 无需手动清除中断标志位，当不满足条件的时候就会自动清除。

I2C DMA 模式：

相关中断：

a) TC_MASK：当 DMA 传输完接收的数据之后，DMA 的 TC 中断就会起来。

b) I2C_TX_DONE_MASK：当发送完数据之后，产生中断。

是否手动清除中断标志位：

TC_MASK、I2C_TX_DONE_MASK 需要手动清除中断标志位，否则就会一直进中断出中断。

注意：

- 非 DMA 模式下，slave 端使用中断接收数据时，为了兼容所有长度可能，slave 必须同时使用 I2C_RX_BUF_MASK 和 I2C_RX_DONE_MASK 才能判断收完一帧。原因是：
  （1）I2C_RX_DONE_MASK 如果数据长度是 trigger level 的倍数的时候，捕捉不到；
  （2）I2C_RX_BUF_MASK 在数据长度不是 trigger level 倍数的时候，尾包中断检测不到。
- 在 DMA 模式下，I2C_RX_DONE_MASK 中断标志位捕捉不到，使用 DMA 的 TC_MASK 中断代替。
- DMA 的所有中断 MASK 默认都是打开的，需要将其他不用的 MASK 关掉，以免影响实验的结果。
- 使用 I2C_TX_DONE_MASK 中断之前，需要手动清除 I2C_TX_DONE_CLR 状态，不清除的话，会一直进中断。
- I2C_TX_DONE_MASK 中断不代表一帧发送完毕（只是表示数据部分发送完毕，没有等 stop 信号），产生 I2C_TX_DONE_MASK 中断后，再查询 busy 信号，直到 IDLE 才代表结束。

15.3 I2C mode

芯片支持作为 master 也支持作为 slave。如果作为 master，需要进行 master 初始化，并且需要设置时钟信号，接口如下：

```
void i2c_master_init(void);
void i2c_set_master_clk(unsigned char clock);
```

如果作为 slave，则调用如下接口进行初始化，每个设备有唯一的地址 (id)：

```
void i2c_slave_init(unsigned char id).
```

15.3.1 I2C no-DMA mode

15.3.1.1 Master

master 发送、接收数据的相关接口配置如下：

```
unsigned char i2c_master_write(unsigned char id, unsigned char *data, unsigned char len);
unsigned char i2c_master_read(unsigned char id, unsigned char *data, unsigned char len);
```

i2c_master_write、i2c_master_read 的功能如下：

master 读写过程中，slave 都正常响应（回 ACK）的情况下，结果显示如下：

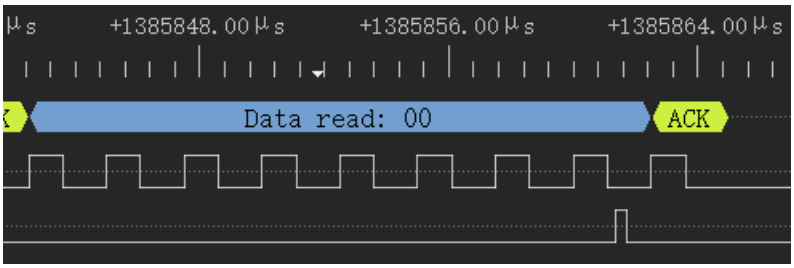


Figure 15.2: master 读写过程中，slave 正常响应的结果

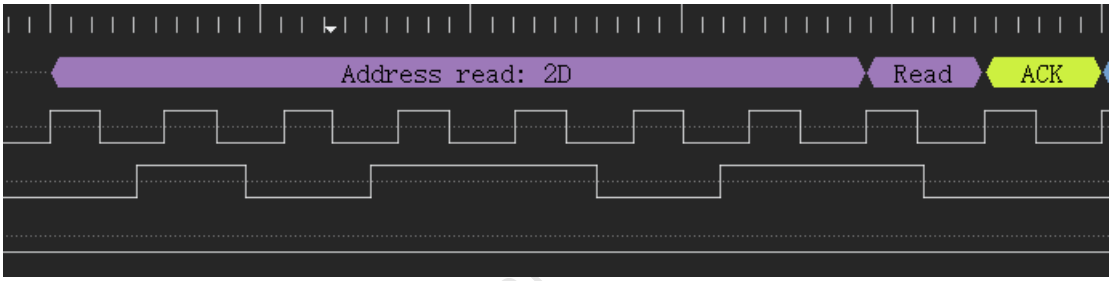


Figure 15.3: master 读写过程中，slave 正常响应的结果

master 发完地址帧后，如果 slave 端回 NACK，则 master 会发 stop 停止传输，结果显示如下：

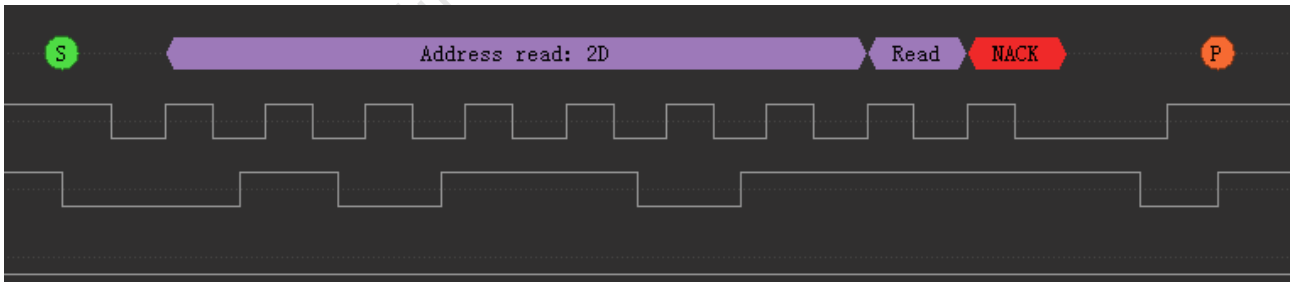


Figure 15.4: master 发完地址帧后，slave 端回 NACK 的结果

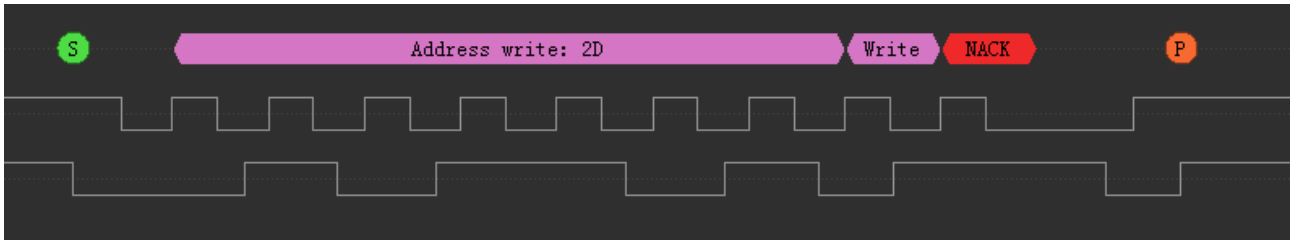


Figure 15.5: master 发完地址帧后，slave 端回 NACK 的结果

如果数据阶段，slave 回 NAK 的话，master 还是会正常发完剩余的数据。结果显示如下：

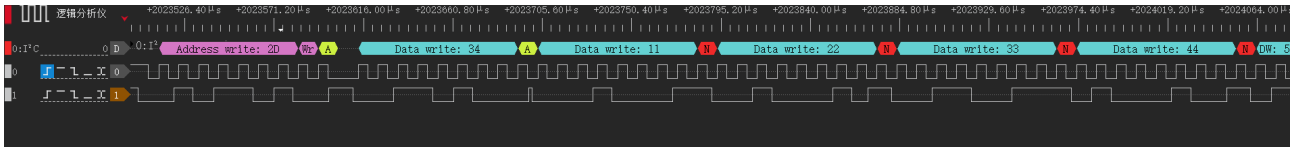


Figure 15.6: 数据阶段，slave 回 NAK 的结果

注意：

- 目前仅 mcu mode 支持地址帧后检测到 NACK 则停止的功能，dma 模式不支持，dma 模式不管 slave 端是否会 NAK，会将整帧数据发完。

15.3.1.2 Slave

slaver 接收、发送数据的相关接口配置如下：

```
void i2c_slave_write(unsigned char* data , unsigned char len ).
void i2c_slave_read(unsigned char* data , unsigned char len ).
```

Slave 端接收数据：可以用中断方式，在非 DMA 模式下的时候，使能 rx（I2C_RX_BUF_MASK）中断和 rx_done（I2C_RX_DONE_MASK）中断，用来判断是否接收完一帧数据。

Slave 端中断的具体配置如下：

```
i2c_set_irq_mask(FLD_I2C_MASK_RX|FLD_I2C_MASK_RX_DONE);
i2c_rx_irq_trig_cnt(SLAVE_RX_IRQ_TRIG_LEVEL);
core_interrupt_enable();
plic_interrupt_enable(IRQ21_I2C);
```

注意：

- 目前存在的问题：I2C 作为 slave 时，软件不能区分 master 发过来的是读指令还是写指令。Slave 端不知道是读还是写，所以 slave 需要在 maste 端发送读指令之前，提前将数据写到 FIFO 中。但是从应用角度来讲，因为不知道 master 端什么时候来读数据，提前塞数据的时间是很难控制的。
- 因此会导致如下问题：在非 DMA 模式下，slave 端需要在 master 发送读指令之前，提前将发送的数据放在 fifo 中。fifo 的大小只有 8 个字节，如果 master 不读走的话，就会一直卡在 i2c_slave_write 函数中，所以要使用该功能时，需要软件层面控制好时间。

15.3.2 I2C DMA mode

15.3.2.1 Master

master 发送、接收数据的相关接口配置如下：

```
void i2c_master_write_dma(unsigned char id, unsigned char *data, unsigned char len);
void i2c_master_read_dma(unsigned char id, unsigned char *rx_data, unsigned char len);
```

在 master 端判断是否发送完、接收完数据，使用的相关接口如下：

```
static inline bool i2c_master_busy(void).
```

注意：

- TX_DONE 中断不代表一帧发送完毕（只是表示数据部分发送完毕，没有等 stop 信号），产生 TX_DONE 中断后，再查询 busy 信号，直到 idle 才代表结束。

15.3.2.2 Slave

slave 发送、接收数据的相关接口配置如下：

```
void i2c_slave_read_dma(unsigned char *data, unsigned char len);
void i2c_slave_write_dma(unsigned char *data, unsigned char len);
```

slave 端接收中断需要配置 DMA TC_MASK，具体配置如下：

```
i2c_clr_txdone_irq_status (I2C_TX_DONE_CLR);
i2c_set_irq_mask(I2C_TX_DONE_MASK);
core_interrupt_enable();
plic_interrupt_enable(IRQ21_I2C);
```

slave 端发送中断需要配置 I2C_TX_DONE_MASK ，具体配置如下：

```
dma_set_irq_mask(I2C_RX_DMA_CHN, TC_MASK);
```

注意：

- Master 端向 slave 端发送读指令，slave 端需要在 master 发送读指令之前，使用 i2c_slave_write_dma。
- 函数提前将发送的数据放在 DMA 中。（注，需要保证此次填入就是需要回复的数据，因为调用该函数后，DMA 就会将指定的 buffer 数据放在 fifo 中，等 master 的 clock 来了之后将数据读出）。

15.4 I2C demo 说明

通过 app.c 和 app_dma.c 中的宏 I2C_MASTER_WRITE_READ_MODE 选择使用哪一种模式。

```
#define I2C_MASTER_WRITE_READ_NO_DMA            1

#define I2C_MASTER_WRITE_READ_DMA               2

#define I2C_MASTER_WRITE_READ_MODE              I2C_MASTER_WRITE_READ_NO_DMA
```

通过宏 I2C_DEVICE 选择 master 和 slave 模式。

```
#define I2C_MASTER_DEVICE            1

#define I2C_SLAVE_DEVICE             2

#define I2C_DEVICE                   I2C_MASTER_WRITE_READ_NO_DMA
```

注意：

- 通过两个板子测试 I2C 通信时，当将 master 端和 salver 端的代码都烧到板子之后，将两个板子都断电，先给 slave 端上电，然后再给 master 上电（避免数据错误），另外两个板子之间需要共地。

154.1 功能说明

节点	功能
master	不断地发送写操作，读操作，然后将发送的数据与读取的数据进行比较，如果比较的结果不一样，LED2 就会翻转。
slave	将接收到是数据返回给 master。

154.2 示例结果

使用逻辑分析仪抓取 I2C 的发送数据和接收数据的时序，执行结果如下：

通道 0：代表 I2C 的 SCL 信号。

通道 1：代表 I2C 的 SDA 信号。

通道 2：LED2 代表 master 端判断发送数据和接收数据是否一样的标志位，如果对比的数据不一样就会翻转。

master 端发送数据：

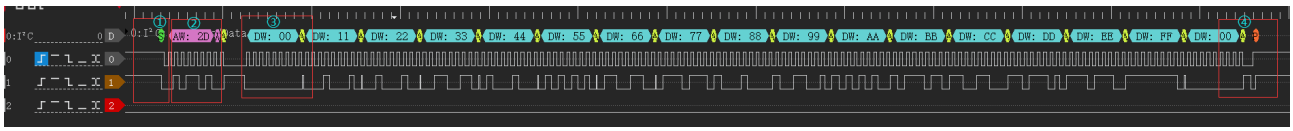


Figure 15.7: Master 端发送数据

master 端接收数据：

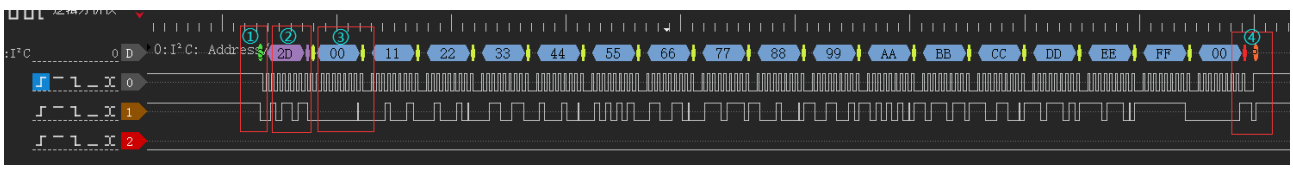


Figure 15.8: Master 端接收数据

特别说明：

(1) i2c write 的时候，master 在写完最后一个字节之后 slave 会回 ACK，然后 master 发送 stop 信号结束通信。

(2) i2c read 的时候，master 在接收完 slave 发送的最后一个字节之后，slave 端发送 NACK 信号，并释放 SDA 线，以便主控接收器发送一个停止信号 P。

16 UART

16.1 简介

UART 是一种异步全双工串行通信协议，由 Tx 和 Rx 两根数据线组成，由于没有时钟参考信号，所以使用 UART 的通信双方必须约定串口波特率、数据位宽、奇偶校验位、停止位等配置参数，从而按照相同的速率进行通信。

通常异步通信以一个字符为传输单位，通信中两个字符间的时间间隔多少是不固定的，但是在同一个字符中两个相邻位之间的时间间隔是固定的。当波特率为 9600bps 时，传输一个 bit 的时间间隔大约为 104.16us，波特率为 115200bps 时，传输一个 bit 的时间间隔大约为 8us。

数据传输速率用波特率来表示，即每秒传送的二进制位数。每一个字符由 11 位（1 个起始位，8 个数据位，1 个校验位，1 个结束位）组成，例如数据传输速率为 120 字符每秒，则其传送的波特率为 11×120=1320 字符/秒 =1320 波特。

16.2 数据通信时序

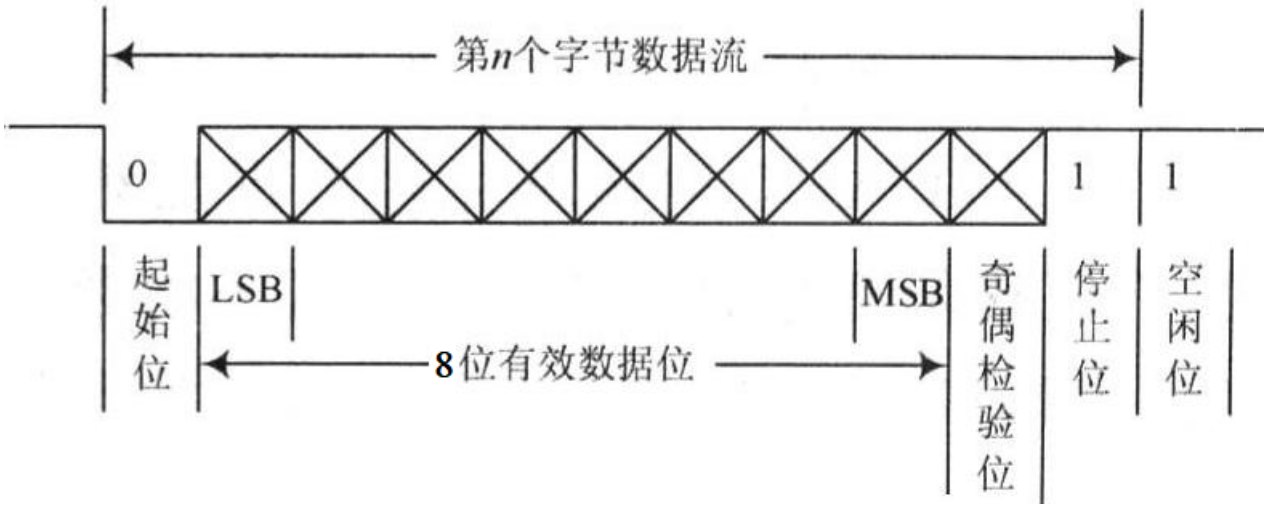


Figure 16.1: 数据通信时序

其中各位的意义如下：

起始位：先发出一个逻辑"0"，表示传输字符的开始。

数据位：可以是 5-8 位的逻辑"0"或者"1"，如 ASCII 码（7 位），扩展 BCD 码（8 位），发送方式为小端传输，即 LSB 先发，MSB 后发。

校验位：数据位加上这一位后，使得"1"的位数应为偶数（偶校验）或者奇数（奇校验）。

停止位：它是一个字符数据结束的标志，可以是 1 位，1.5 位，2 位的高电平（用于使双方同步，停止位时间越长，容错能力就越强）。

空闲位：处于逻辑"1"的状态，表示当前线路上没有数据传送。

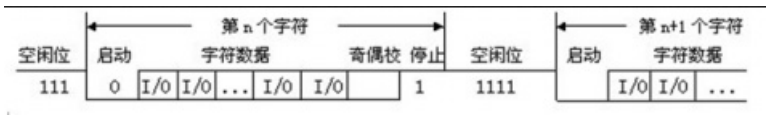


Figure 16.2: 异步通信时序

如上图，值得注意的是，异步通信是按照字符传输的，接受设备在收到起始信号之后只要在一个字符的传输时间内和发送设备保持同步就能正确接收。下一个字符起始位到来之后又需要同步重新校准（依靠检测起始位来实现发送与接收方的时钟自同步）。

16.3 通信原理

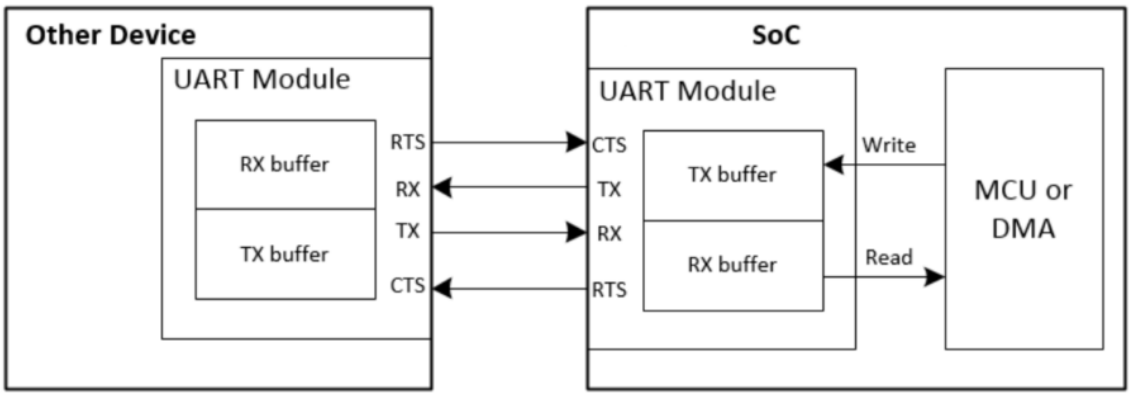


Figure 16.3: UART 通信原理

以 Telink SoC 中 UART 模块为例，需要被发送的数据首先被芯片的 MCU 或者 DMA 写入 TX 缓冲区，然后 UART 模块通过 TX 引脚将 TX 缓冲区中的数据发送到其他设备。在接收数据的设备中，数据首先通过 UART 模块的 RX 引脚被写入 RX 缓冲区，然后数据被接收设备的 MCU 或者 DMA 读取。

如果芯片的 RX 缓冲区接近溢出状态，芯片会通过其 RTS 引脚发出一个信号（可以配置为高电平或者是低电平）给与其连接设备，即表明该设备应该停止继续发送数据。与其类似，如果芯片通过其 CTS 引脚接收到了一个信号，即表明其他设备的 RX 缓冲区接近溢出，芯片应该停止继续发送数据。

164 功能简介

164.1 初始化

注意事项如下：

(1) 每次使用 UART 端口时，最好都先调用 uart_reset() 函数复位一次，这样可以避免之前使用 UART 的操作对本次使用产生影响。（如 UART 相关寄存器遗留设置）

```
uart_reset(uart_num_e uart_num)
```

(2) 使用 uart_set_pin() 用来设置 UART 中 TX/RX 引脚

```
uart_set_pin(uart_tx_pin_e tx_pin,uart_rx_pin_e rx_pin)
```

接线注意：当前设备的 TX/RX 与其他设备的 TX/RX 连接规则为 TX-RX,RX-TX。

(3) DMA 模式，注意选用没有被其他模块占用的 DMA 的通道

(4) NDMA 模式，如果使用中断，需要使用如下配置：

a) uart_rx_irq_trig_level(): 用来设置接收字符中断触发个数 level，如设置为 1，则接收一个字符即进入中断一次。(推荐设置为 1)。

b) uart_tx_irq_trig_level(): 用来设置发送字符中断触发 level，当发送缓冲区字符个数达到 level，即进入发送中断。设置为 0 表明缓冲区有数据即进行发送。

注意：

- NDMA 模式在使用中断接收数据时，建议在初始化时设置 level=1（比较通用），原因如下：

(1) NDMA 模式接收数据时，可以在初始化时手动设置接收数据中断 level，level 值可以为 1-4。我们默认设置 level 为 1。如果将 level 设置为 2，3，4，就需要接收数据的长度满足一定条件才能正常完成数据的接收。例如：设置 level 为 2，如果实际接收数据长度为 2 的倍数，可以正常完成数据接收。但如果实际接收数据长度不是 2 的倍数，例如 3，前两个数据可以正常接收，第三个数据会丢失。(NDMA 模式从缓冲区取数据按照 level 的长度来取，如果最后一次取数据，长度达不到 level，数据将取不出来)。level 设置为 3，4 与之类似，需要接收数据长度为 3，4 的倍数，才能正常完成数据的接收。

(2) level 设置为 1 时，需要注意：由于接收中断触发过于频繁，如果接收数据过快（波特率过高），可能会出现前一个数据还没有接收完，下一个数据已经到来的情况。如果出现这种情况，可以通过适当降低波特率来解决。

164.2 波特率

164.2.1 函数调用

计算波特率会用到两个函数：

(1) uart_cal_div_and_bwpc() 函数会根据输入的波特率和系统时钟计算出最佳的时钟分频数 div 和位宽 bwpc。

```
uart_cal_div_and_bwpc(unsigned int baudrate, unsigned int sysclk, unsigned short* div,
↪  unsigned char *bwpc)
```

(2) uart_init() 将上面计算的时钟分频数 div 和位宽 bwpc 传入该函数，才真正的设置了波特率，同时该函数还设置了 UART 端口，设置停止位、校验位等。

```
uart_init(uart_num_e uart_num, unsigned short div, unsigned char bwpc, uart_parity_e
↪  parity, uart_stop_bit_e stop_bit)
```

有一些应用对时序要求较高的，可以先计算好时钟分频数 div 和位宽 bwpc，然后直接调用 uart_init 即可，这样省去了 uart_cal_div_and_bwpc 函数的执行时间。

164.2.2 **实测数据**

在 DMA 和 NDMA 模式下，我们分别测试了不同波特率对数据传输准确性的影响。

测试数据：0x00，0x11, 0x22 …… 0xff 共 16 个数据

测试条件：两块 B91 开发板 TX/RX 互连，每隔一秒通信一次（实验表明，无时间间隔和有时间间隔现象一致，这里为了现象更为稳定，采用 1s 的时间间隔），时钟设置为 16MHZ-PCLK，16MHZ-HCLK，16MHZ-CCLK。

测试芯片：B91 80pin-EVK

测试程序：UART-DEMO。

NDMA 模式

波特率	收	发
2000000	不通过	不通过
1500000	通过	通过
1000000	通过	通过
500000	通过	通过
256000	通过	通过
115200	通过	通过
57600	通过	通过
38400	通过	通过
19200	通过	通过
9600	通过	通过
4800	通过	通过
2400	通过	通过
1200	通过	通过
600	通过	通过
300	通过	通过

DMA 模式

波特率	收	发
2000000	通过	通过
1500000	通过	通过
1000000	通过	通过

波特率	收	发
500000	通过	通过
256000	通过	通过
115200	通过	通过
57600	通过	通过
38400	通过	通过
19200	通过	通过
9600	通过	通过
4800	通过	通过
2400	通过	通过
1200	通过	通过
600	通过	通过
300	通过	通过

由测试可知，在 300 到 1.5M 波特率之间，NDMA 和 DMA 都可以正常通信，在 2M 波特率下，DMA 模式仍可以正常通信，但 NDMA 模式已经出现了错误（收发数据出现乱码）。

注意：

- 进一步的测试表明，16MHZ-PCLK 下 DMA 极限波特率可达 5M。
- 当 PCLK 提升到 24MHZ，HCLK 和 CCLK 提升到 48MHZ 时，NDMA 模式极限波特率可达 5M，DMA 极限波特率可达 8M，PCLK 是 UART 波特率上限的主要影响因素。

164.3 中断

中断	产生条件	自动清除还是手动清除
TX DMA TC	每当接收一帧数据，会产生中断	需要手动清除。
UART_TXDONE	NDMA 和 DMA 共用。默认值为 1，开始发送数据时会置 0，当数据发送完成后，硬件会自动置 1。	需要手动清除。
UART_RXDONE (是否可以使用需要查看芯片差异)	该中断只能在 DMA 模式下使用，默认值为 0，接受完一包数据会置 1。	需要手动清除。
UART_RXBUF_IRQ_STATUS	当接收 BUFF 缓冲区数据量达到初始化时设置的 level 时，会产生中断。	当缓冲区数据被读取后，该标志位自动清除。
UART_TXBUF_IRQ_STATUS	当发送 BUFF 缓冲区数据量达到初始化时设置的 level 时，会产生中断。	当缓冲区数据被发送出去后，该标志位会自动清除。

中断	产生条件	自动清除还是手动清除
UART_RX_ERR（是否可以使用需要查看芯片差异）	UART 接收出错标志。当 UART 接收数据出错时(例如奇偶校验出错或停止位错误)，会产生中断，中断后将停止接收数据。	该标志位需要手动清除。

1644 DMA 模式

1644.1 发送数据

使用该函数发送数据，该函数只是触发发送动作，并没有实际发完，需要使用查询或者中断的方式判断是否发完。

```
unsigned char uart_send_dma(uart_num_e uart_num,unsigned char * addr,unsigned char len)
```

查询

```
uart_send_dma(UART0, (unsigned char*)tx_byte_buff, 16);
while(uart_tx_is_busy(UART0));
```

需要使用 uart_tx_is_busy 接口查询当前状态，如果该标志位为 0，则表明上一帧数据发送结束，可以进入到当前帧数据的发送中。

中断

使用 TX_DONE 中断，需要设置对应的 mask：

```
uart_set_irq_mask(UART0, UART_TXDONE_MASK);
```

除了正常的使用中断相关函数的调用，还需要注意以下事项：

初始化的时候，需要调用下面函数（将 TX_DONE 信号置 0，否则会一直进中断）：

```
uart_clr_tx_done(UART0);
```

中断处理函数中需要按照如下方式处理：

```
_attribute_ram_code_sec_noinline_ void uart0_irq_handler(void)
{
    if(uart_get_irq_status(UART0,UART_TXDONE))//判断中断标志位
    {
        .........
        uart_clr_tx_done(UART0); //将 TX_DONE 信号置 0
    }
 }
```

标志位：UART_TXDONE

使用中断方式时，产生 UART_TXDONE 中断，表示前一帧数据发送结束，可以进入到下一帧数据的发送中。

注意：

- DMA 模式使用中断方式发送数据时，我们需要利用 UART_TXDONE 的状态。由于 UART_TXDONE 的初始默认值是 1，为了可以正常的产生中断，在初始化时使用 uart_clr_tx_done(UART0) 函数将 UART_TXDONE 置 0，这样等发送完成 UART_TXDONE 会自动置 1，此时进入 UART_TXDONE 中断，然后在中断的处理程序中使用 uart_clr_tx_done(UART0) 函数再将 UART_TXDONE 拉低。

1644.2 接收数据

使用 DMA 接收数据时，在初始化设有数据包接收结束的判断，即 rx_timeout。

```
uart_set_dma_rx_timeout(uart_num_e uart_num, unsigned char bwpc, unsigned char bit_cnt,
↪  uart_timeout_mul_e mul)
```

其中，bwpc 和 bit_cnt 设置传输一个 byte 所用时间，一个 byte 传输所需时间为 (bwpc+1)*bit_cnt。mul 设置超时时间，可以选择设置为 0，1，2，3。mul 设置为 0 表示接收时如果超过 1 个 byte 的时间没有接收到数据，则认为接收数据包结束。设置为 1 表示超过 2 个 byte 的时间没有接收到数据则认为数据包结束。

使用 DMA 模式接收数据时有两种方式：

(1) RX_DONE 中断

RX_DONE 中断接收数据是较为通用的方式，这种方式允许我们接收未知长度的数据包，但有些芯片不支持这种方式。芯片是否支持这种方式请查看 UART 的芯片差异章节。

(2) DMA 中断

如果芯片不支持 RX_DONE 中断，可以使用 DMA 中断接收数据，但这种方式有使用限制，只能接收已知长度的数据包。

RX_DONE 中断

使用 RX_DONE 中断，需要设置对应的 mask。

```
uart_set_irq_mask(UART0, UART_RXDONE_MASK);
```

标志位：UART_RXDONE

使用中断方式接收数据时，产生 RX_DONE 中断，表示接收完数据，该标志位需要手动清除。

注意：

- 使用 DMA-RX_DONE 中断时，在中断处理函数中设有接收数据长度 rec_data_len 的计算，这样在使用 RX_DONE 方式接收数据时，我们可以明确知道接收数据的长度。

DMA 中断

通过中断标志位 DMA_TC_IRQ，当有中断请求时，表示接收完数据，该标志位需要手动清除。

```
void uart_receive_dma(uart_num_e uart_num, unsigned char * addr,unsigned char rev_size)
```

注意：

- 必须已知数据长度，才能产生中断。参数 rev_size 需要和接收数据的实际长度 len 满足一定关系，这样才不会有数据遗漏，并准确产生中断。（例如 4*(n-1)<len≤4*n, 则 rev_size 需要满足 4*(n-1)<rev_size≤4*n，其中 n 为同一值）。（例如：rev_size 设置为 8，接收数据长度是 5/6/7/8 才会产生中断）

注意事项：

(1) rec_buff 接收数组的大小要多预留部分区域。原因是：DMA 以 4byte 为单位搬运数据，例如接收数据的实际长度为 5，rev_size 设置为 5 - 8 均可,DMA 会搬运两次，第二次搬运虽然有效数据只剩下一个，DMA 仍会搬运 4 个数据，其中后三个数据为无用数据。

(2) 接收数据的长度 len 满足一定关系，例如当 4*(n-1)<len≤4*n 时，将 rec_buff 大小设置为大于等于 4n。这样 rec_buff 前 len 个为有效数据，后面的为无效数据，我们需要进行准确区分。

(3) 在实际使用 DMA 接收数据时最好设置为：当接收数据长度 len 在 4*(n-1)<len≤4*n 时，设置 rec_buff=rev_size=4n.

示例：rec_buff 长度设置为 8，rev_size 设置为 8，分别发送 4，5，6，7，8 个数据对比。

程序设置 DMA 为接收模式，初始化 rec_buff 全为 0。通过 BDT 查看 rec_buff 值。

发送数据	接收数据	中断是否产生	无效数据个数	DMA 搬运次数
0x11, 0x22, 0x33, 0x44	0x11, 0x22, 0x33, 0x44	否	0	1
0x11, 0x22, 0x33, 0x44, 0x55	0x11, 0x22, 0x33, 0x44, 0x55, 0xXX, 0xXX, 0xXX	是	3	2
0x11, 0x22, 0x33, 0x44, 0x55, 0x66	0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0xXX, 0xXX	是	2	2
0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77	0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0xXX	是	1	2
0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88	0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88	是	0	2

注意：

- 0xXX 表示无效数据值，通常显示为乱码

164.5 NDMA 模式

164.5.1 发送数据

```
void uart_send_byte(uart_num_e uart_num, unsigned char tx_data)
```

注意：

- NDMA 模式还为我们提供了以 hword（2byte）和 word（4byte）形式发送数据，需要注意，发送数据时先发送低字节数据，再发送高字节数据。如一串数据为 0x11223344，如果要在接收端接收数据也为 0x11223344，以 hword 形式发送顺序为 0x2211,0x4433。以 word 形式发送顺序为 0x44332211。

查询

```
uart_send_byte(UART0, uart0_tx_buff_byte[i]);
while(uart_tx_is_busy(UART0));
```

需要使用 uart_tx_is_busy 接口查询当前状态，如果该标志位为 0，则表明上一帧数据发送结束，可以进入到当前帧数据的发送中。

中断

标志位：UART_TXBUF_IRQ_STATUS

在初始化中，通过 uart_tx_irq_trig_level() 设置了 TX 的中断触发个数 level，当发送缓冲区数据达到该 level 时，发送即开始。此时将进入发送中断。

注意：

- DEMO 中给出 TX 中断触发 level 为 0，即缓冲区有数据即进行发送。此时如果使用 TX 中断，则程序将会一直处于 TX 中断状态，不利于实际使用。因此，DEMO 中不给出 TX 中断的判断和处理，这样更符合我们的实际使用习惯（有发送需求直接发送即可），如果确实需要对发送状态进行判断，建议使用查询方式。

164.5.2 接收数据

```
unsigned char uart_read_byte(uart_num_e uart_num)
```

标志位：UART_RXBUF_IRQ_STATUS

在初始化中设置了接收中断触发个数 level，当接收字符个数达到该 level 时，该标志位即置 1，此时即进入中断，将缓冲区数据移到我们预先定义的接收数组当中。

164.6 流控

因为 UART 双方处理速度的差异，在进行数据传送时，接收速率与发送速率存在很大的差距，这样在进行数据的发送和接收过程中可能出现接收方来不及接收的情况。为了防止数据丢失，这是就需要发送方进行控制，这就是所谓的流量控制。

CTS (clear to send) 允许发送

RTS (request to send) 请求发送

如果 UART0 与 UART1 进行通信，则 UART0 的 RTS 管脚与 UART1 的 CTS 管脚相连，UART0 的 CTS 管脚与 UART1 的 RTS 管脚相连。

164.6.1 CTS

```
uart_cts_config(uart_num_e uart_num,uart_cts_pin_e cts_pin,u8 cts_parity)
```

uart_cts_config() 函数用来配置使用 CTS 流控的端口号，CTS 引脚。当 CTS 引脚输入电平和 cts_parity 相等时，UART 将停止发送。

164.6.2 RTS

```
uart_rts_config(uart_num_e uart_num,uart_rts_pin_e rts_pin,u8 rts_parity,u8 auto_mode_en)
uart_rts_trig_level_auto_mode(uart_num_e uart_num,u8 level)
```

uart_rts_config() 函数用来配置 UART 端口号，RTS 引脚。

rts_parity 只在 auto 模式下生效，它表明当接收数据量达到 level 值时 RTS 引脚跳变方向，为 1 时电平从低到高跳变，为 0 时从高到低跳变。

注意：

> - RTS 在配置时有两种模式可选，UART_RTS_MODE_AUTO/UART_RTS_MODE_MANUAL. 即自动模式和手动模式。自动模式下，RTS 引脚在接收到 RTS-THRESH 个数据时自动进行 RTS_INVERT 相关的跳变。手动模式下，我们需要计算接收的数据长度，当数据长度达到预期值时，我们需要使用 uart_set_rts_level() 函数手动拉低或拉高 RTS 引脚。

uart_rts_trig_level_auto_mode() 为 RTS 跳变触发设置，level 表明触发门槛，如设置为 5 时，接收 5 个数据 RTS 引脚即进行跳变。

16.5 DEMO 简介

在头 UART_DEMO/app_config.h 中可以选择配置 UART 工作模式（DMA 和 NDMA），如下图，分别对应 app_dma.c 与 app.c 中的内容。

```
#define UART_DMA      1 //uart use dma
#define UART_NDMA     2 //uart not use dma
#define UART_MODE     2
```

其中 NDMA 与 DMA 模式又可以具体选择流控模式，代码如下：

```
#define BASE_TX      0 //just for NDMA
#define NORMAL       1
#define USE_CTS      2
#define USE_RTS      3
#define FLOW_CTR     1
```

16.5.1 DMA 模式

UART 模块通过 DMA 不停的发送接收到的数据，初始 rec_buff[] 全为 0。

通过串口工具进行验证：

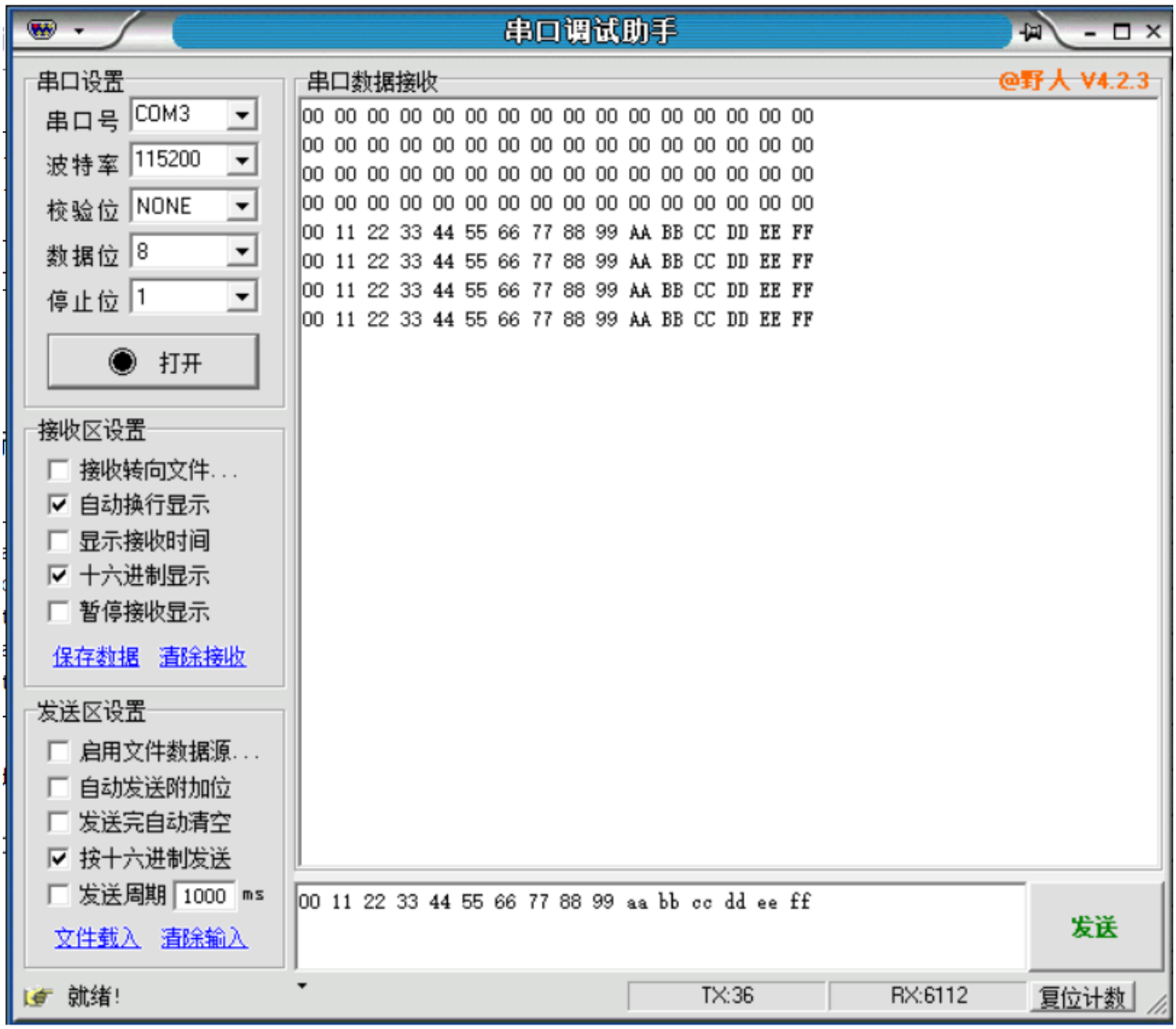


Figure 16.4: DMA 串口验证

16.5.2 NDMA 模式

当接收字符个数达到预设的接受长度（即 UARTO_RX_IRQ_LEN）时，uartO_rx_flag 为 1，此时进入到此中断服务程序中，进行的处理为：将接收的字符发送出去。

通过串口工具进行验证：

Figure 16.5: NDMA 串口验证

16.5.3 RTS 与 CTS

以 NDMA 为例，由于硬件条件限制，只单一演示 RTS 或者 CTS。

CTS：设置 STOP_VOLT=1，即高电平停止发送 TX。

当 CTS 引脚为 0 时，持续发送，利用串口工具查看：

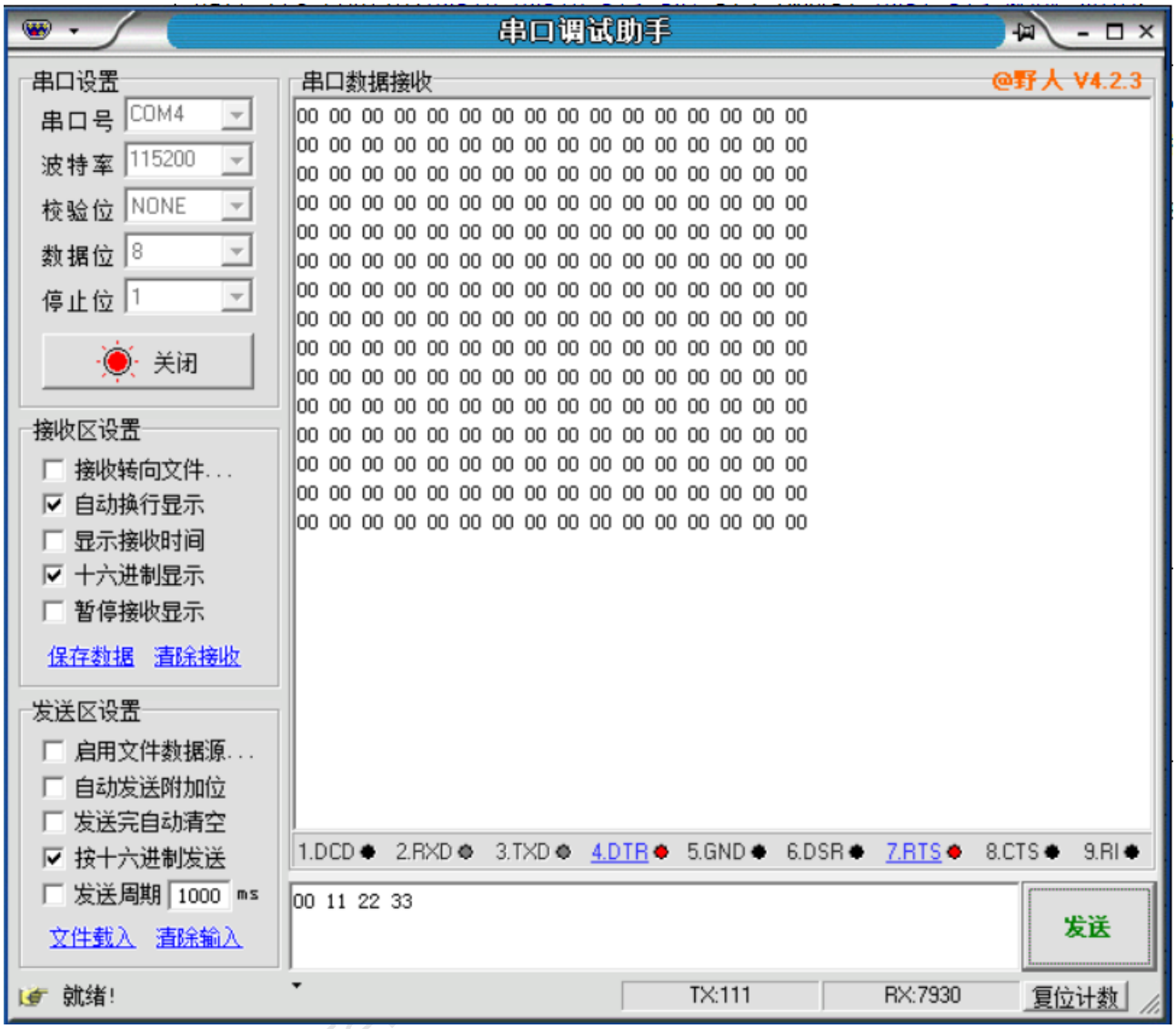


Figure 16.6: CTS 引脚为 0

将 CTS 置高电平，发送停止。

RTS：自动模式下，设置 RTS_THRESH=5,RTS_INVERT=1, 即累计当收到 5 个数据时，RTS 引脚从低到高跳变。

为使现象较为显著，利用逻辑分析仪查看跳变结果。

使用串口工具发送 4 个数据：

Figure 16.7: 使用串口工具发送 4 个数据

RTS 跳变未发生

发送 5 个数据:

Figure 16.8: 使用串口工具发送 5 个数据

RTS 引脚完成了从低到高的跳变。

Figure 16.9: 跳变

注意：

- 实际使用 UART 的流控时，我们需要注意合理配置 CTS 与 RTS 的跳变方式。例如，一个设备的 RTS 引脚设置为触发时由低到高跳变，那么与之相连的另一个设备的 CTS 引脚就需要设置为高电平停止发送。这样 UART 流控才能正常工作。

16.6 芯片差异

16.6.1 UART_RXDONE 中断

支持使用该中断才可以有效判断一个数据包的结束（即使不知道数据包的长度）。

芯片	DMA 模式 RX_DONE 中断
B91 A0	不支持
其他	支持

16.6.2 UART_RX_ERR 中断

B91 A0 芯片：

(1) DMA 模式：UART_RX_ERR 中断不能使用。(原因：硬件检测到该中断后自动清除，软件检测不到)

(2) NDMA 模式：UART_RX_ERR 中断可以使用。

建议使用方法：清除中断状态，清除 RX-FIFO，硬件指针和软件指针归零。

原因：接收出错时，FIFO 中可能存在错误数据，为了不影响后续正确数据的接收，需要清空 RX-FIFO（即指针归零，uart_reset() 使硬件指针清零，同时软件指针保持一致也需要清零）。

总结：该模式下检测到 UART_RX_ERR 中断后，需要执行以下操作：

```
uart_clr_irq_status(UART0,UART_CLR_RX);//NDMA 模式下会清除 RX-FIFO 和 RX_ERR_IRQ，RX_BUFF_IRQ(注
↪ 意：如果进入 RX_ERR 中断，说明接收数据出错，清除 RX 会同时清除 RX_BUFF 中断)。
uart_reset(); //硬件指针清零
uart_clr_rx_index();//软件指针清零
```

其他芯片：

(1) DMA 模式：UART_RX_ERR 中断可以使用。

建议使用方法：只清除中断状态。

原因：DMA 模式接收数据出错，当进入到 RX_DONE 中断时，DMA 会将 RX-FIFO 中的数据搬运完，FIFO 指针自动归零，不需要 uart_reset()，错误数据不会对后续正确数据的接收造成影响。(注意：如果发生 UART_RX_ERR 中断，接收数组中的数据将是错误数据，无法使用)

总结：该模式下检测到 UART_RX_ERR 中断后，需要执行以下操作：

```
uart_clr_irq_status(UART0,UART_CLR_RX); //DMA 模式下会清除 RX_FIFO 和 RX_DONE_IRQ,RX_ERR 状态。
```

17 SPI

17.1 简介

17.1.1 标准 SPI 接口

串行外设接口（Serial Peripheral Interface）简称 SPI 接口，是一种同步串行外设接口，允许嵌入式处理器与各种外围设备通过串行的方式进行通信、数据交换等。

标准 SPI 接口一般使用 4 条线通信：

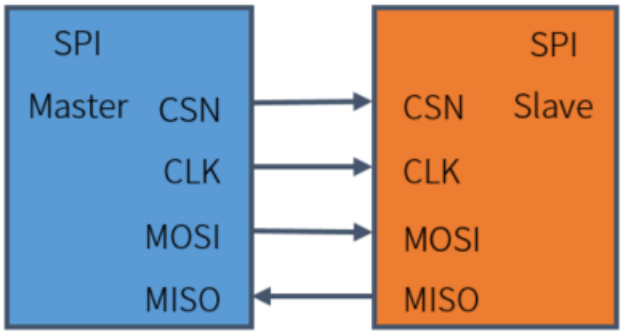


Figure 17.1: SPI 接口

名称	含义
CSN	设备片选信号线，低电平有效
CLK	时钟信号线
MOSI	Master 数据输出 Slave 数据输入线
MISO	Master 数据输入 Slave 数据输出线

17.1.2 SPI 通信过程

下图是 SPI 通信的一个简单例子：

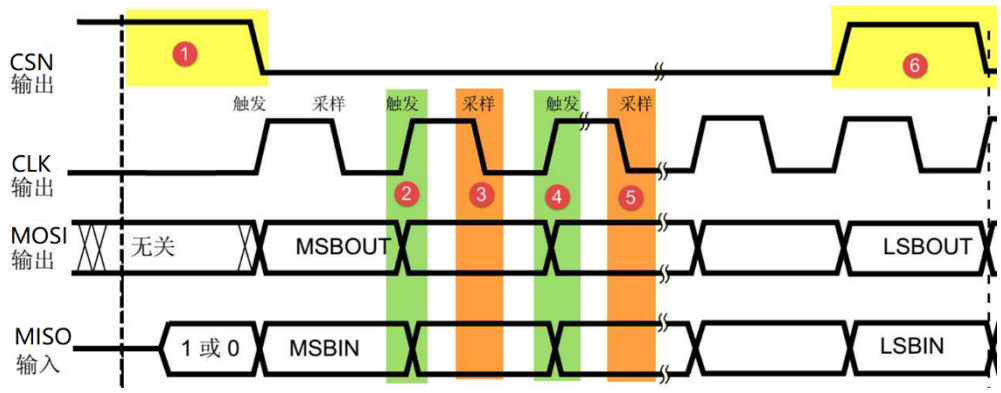


Figure 17.2: SPI 通信

这是一个标准 SPI 的通信时序。CSN、CLK、MOSI 信号都由 Master 产生，数据输出通过 MOSI 线，而 MISO 的信号由 Slave 产生，Master 通过该信号线读取 Slave 的数据。MOSI 与 MISO 的信号只在 CSN 为低电平的时候才有效，数据在 CLK 上升沿或下降沿时触发采样。在 CLK 的每个时钟周期 MOSI 和 MISO 传输 1bit 数据，8 个时钟周期就可以实现 1Byte 数据传输。

根据空闲时 CLK 时钟极性 CPOL(Clock Polarity) 和采样时刻的 CLK 时钟相位 CPHA(Clock Phase) 的不同，SPI 区分出四种工作模式，见下表，主机与从机需要工作在相同的模式下才可以正常通讯。

SPI 工作模式	CPOL	CPHA
SPI_MODE0	0	0
SPI_MODE1	0	1
SPI_MODE2	1	0
SPI_MODE4	1	1

CPOL: Clock Polarity

- 当 CPOL=0, CLK 在空闲时刻保持低电平；
- 当 CPOL=1, CLK 在空闲时刻保持高电平。

CPHA: Clock Phase

- 当 CPHA =0, 在 CLK 的奇数边缘触发采样；
- 当 CPHA =1, 在 CLK 的偶数边缘触发采样。

17.1.3 多样化的 SPI 接口

在标准 SPI 的基础上为适应不同的应用场景，逐渐衍生出很多类别的 SPI 接口。

3line SPI：3line SPI 顾名思义只有 3 根线，CLK, CSN, MOSI，数据收发共用一根线，为半双工通信。

Dual SPI ：它拓展了 MOSI 和 MISO 的用法，让它们工作在半双工，同向传输数据，用以加倍数据传输。也就是对于 Dual SPI，MOSI 变成 IO0，MISO 变成 IO1，这样一个时钟周期内就能传输 2 个 bit 数据，加倍了数据传输。

Quad SPI：与 Dual SPI 类似，它拓展了 WP 和 HOLD 的用法，WP 变成 IO2 和 HOLD 变成 IO3，也就是同时拥有了四根数据线，一个时钟周期内可以传输 4 个 bit，传输速度会大幅提升。

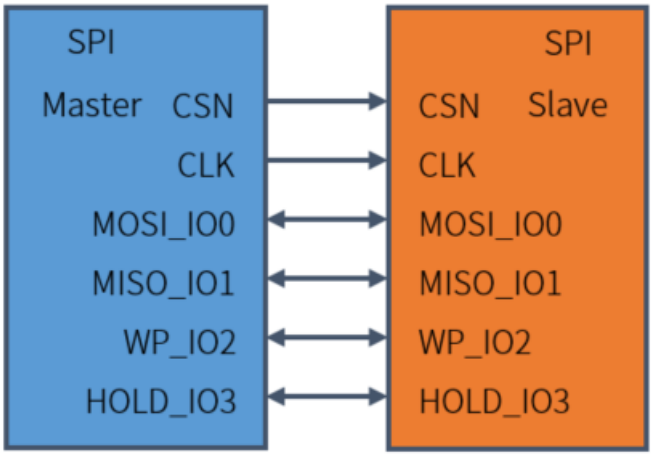


Figure 17.3: 多样化的 SPI 接口

17.2 功能说明

SoC 包含多种 SPI 模块，有 HSPI、PSPI 和 SPI Slave，针对每个模块都配有相应的驱动支持。

17.2.1 接口说明

接口命名规则：

- spi 前缀：hspi 和 pspi 均可以使用的接口。
- pspi 前缀：仅供 pspi 使用。
- hspi 前缀：仅供 hspi 使用。
- dma 后缀：dma 模式会用到的接口。
- plus 后缀：支持更丰富的读写模式和操作指令。

例如：spi_master_write_read_dma_plus 接口在使用时就会使用 DMA 通道，先写地址到 SPI Slave，再从 SPI Slave 对应地址读出数据。

注意：

- 带有"read"字段"write_read"的接口的功能都是从 SPI Slave 读出数据。
- 二者区别在于：
  (1) 带有"read"字段的接口支持硬件自动发送 address 帧，适用于使能硬件 address 帧支持的应用场景，接口直接可以读取对应地址的数据。
  (2) 带有"write_read"字段的接口是在没有使能硬件 address 帧的情况下使用的。地址信息先要通过"write"的方式写 SPI Slave，然后才能读取对应地址的数据。

17.2.2 HSPI 与 PSPI

HSPI 与 PSPI 是 SoC 所支持的两种高级的 SPI 接口，均支持 Master 和 Slave 模式。HSPI/PSPI Slave 均会自动解析 cmd，数据接收和发送需要用软件进行操作。

注意：

- 例程中，HSPI/PSPI Slave 接收和发送数据是通过软件方式实现的，优点是配置更灵活而且支持 Quad I/O 模式，缺点是配置方式不如直接使用 SPI Slave 模块简单，而且会消耗运算资源。在使用时如果需要简单配置本设备作为 Slave，建议使用 SPI Slave 模块。

17.2.2.1 Master

标准 SPI Master

SoC 的 HSPI/PSPI 模块支持标准 SPI Master 模式，此模式下调用的函数接口不带有"plus"后缀，功能和模式比较简单。

初始化接口如下：

```
spi_master_config(SPI_MODULE_SEL, SPI_NOMAL);
```

支持的读写操作接口为：

```
void spi_master_write(spi_sel_e spi_sel, u8 *data, u32 len);
void spi_master_write_dma(spi_sel_e spi_sel, u8 * data, u32 len);
void spi_master_write_read(spi_sel_e spi_sel, u8 * addr , u32 addr_len, u8 * data , u32
↪ data_len);
void spi_master_write_read_dma(spi_sel_e spi_sel, u8 *addr, u32 addr_len, u8 *data, u32
↪ data_len);
```

HSPI/PSPI Master

功能说明:

SoC 的 HSPI/PSPI Master 对常用协议中的几种帧信息增设了对应的硬件配置，包括 cmd 帧，address 帧，dummy 帧（空周期）。

如下表，在 HSPI/PSPI Master 所支持的功能中，Y 代表支持，N 代表不支持。

SPI 模块	cmd_en	cmd_fmt	address_en	address_fmt	3line	Dual	Quad
HSPI	Y	Y	Y	Y	Y	Y	Y
PSPI	Y	N	N	N	Y	Y	N

- cmd_en：硬件 cmd 帧
- cmd_fmt：cmd 帧跟随对应 Dual/Quad I/O 的编码格式
- address_en：硬件 address 帧
- address_fmt：cmd 帧跟随对应 Dual/Quad I/O 的编码格式
- 3line：3line SPI 模式
- Dual：Dual SPI 模式
- Quad：Quad SPI 模式

数据帧格式:

HSPI/PSPI Master 支持的数据发送的格式为:【cmd】+【adress】+【dummy】+data。【】代表可选

Step 1 通过调用如下结构体接口来配置 HSPI 支持的数据帧格式，PSPI 类似:

```
hspi_config_st hspi_slave_protocol_config = {
    .hspi_io_mode = HSPI_QUAD,
    .hspi_ dummy _cnt = 6,
    .hspi_ cmd _en = 1,
    .hspi_addr_en = 1 ,
    .hspi_addr_len = 3,//when hspi_addr_en = false,invalid set.
    .hspi_cmd_fmt_en = 0,//when hspi_cmd_en = false,invalid set.
    .hspi_addr_fmt_en = 1 ,//when hspi_addr_en = false,invalid set.
};
```

这段代码配置了 HSPI Master 的各种帧参数:

- 模块配置为 Quad I/O 模式;
- dummy 帧长度为 6 个 clock;
- 硬件 cmd 使能，cmd 传输为 Single I/O 模式，不跟随对应 Dual/Quad I/O 的编码格式 (hspi_cmd_fmt_en = 0);
- 硬件 address 使能，address 帧长度为 3Bytes，跟随对应 Dual/Quad I/O 的编码格式 (hspi_addr_fmt_en = 1)。

对应的通信时序图如下:

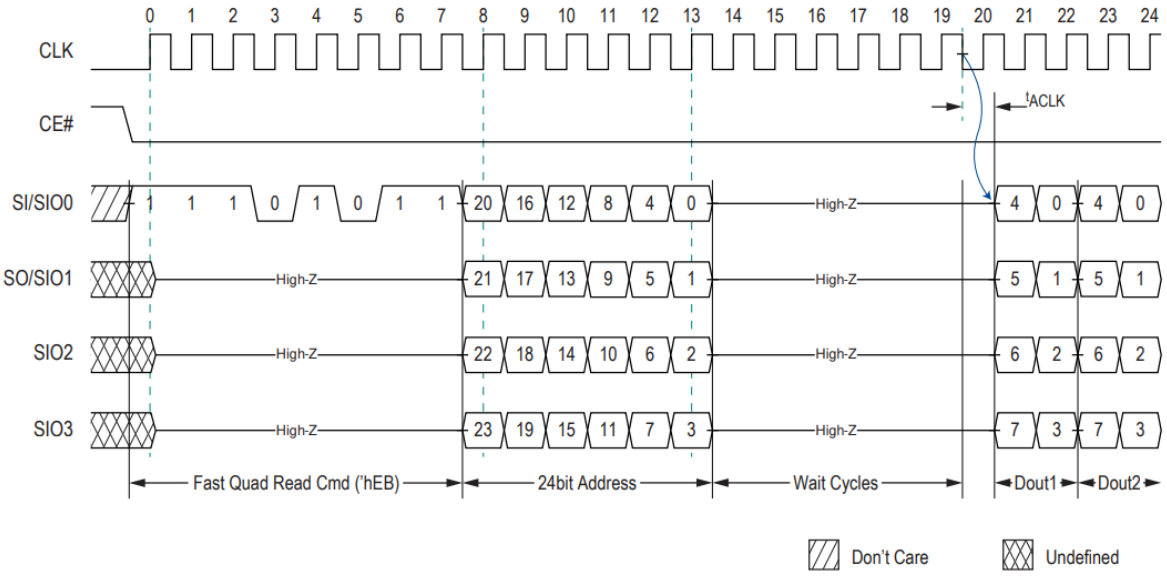


Figure 17.4: 通信时序

调用下面接口使能帧参数的配置:

```
hspi_master_config_plus(&hspi_slave_protocol_config);
```

读写操作调用的接口:

HSPI/PSPI Master 读写操作调用的接口如下:

```
void spi_master_write_plus(spi_sel_e spi_sel, u8 cmd, u32 addr, u8 *data, u32 data_len,
 ↪  spi_wr_tans_mode_e wr_mode);
void spi_master_write_dma_plus(spi_sel_e spi_sel, u8 cmd, u32 addr, u8 *data, u32 data_len,
 ↪  spi_wr_tans_mode_e wr_mode);

void spi_master_read_plus(spi_sel_e spi_sel, u8 cmd, u32 addr, u8 *data, u32 data_len,
 ↪  spi_rd_tans_mode_e rd_mode);
void spi_master_read_dma_plus(spi_sel_e spi_sel, u8 cmd, u32 addr, u8 *dst_addr, u32 data_len,
 ↪  spi_rd_tans_mode_e rd_mode);

void spi_master_write_read_plus(spi_sel_e spi_sel, u8 cmd, u8 *addrs, u32 addr_len, u8 *data,
 ↪  u32 data_len, spi_rd_tans_mode_e wr_ mode);
void spi_master_write_read_dma_plus(spi_sel_e spi_sel, u8 cmd, u8 *addr, u32 addr_len, u8
 ↪  *rd_data, u32 rd_len, spi_rd_tans_mode_e rd_mode);
```

读写方式:

HSPI/PSPI Master 通过操作指令对 SPI Slave 进行操作，同时也需要配置 HSPI/PSPI 的读写方式来配合操作过程。读写方式用来表示操作是否需要 dummy(空周期) 帧，指令操作到底是读还是写。

读写方式的枚举定义如下:

```
typedef enum{
    SPI_MODE_WR_WRITE_ONLY = 1,//write
    SPI_MODE_WR_DUMMY_WRITE = 8,//dummy_write
}spi_wr_tans_mode_e;

typedef enum{
    SPI_MODE_RD_READ_ONLY = 2,//must enbale CmdEn
    SPI_MODE_RD_DUMMY_READ = 9,//dummy_read
}spi_rd_tans_mode_e;

typedef enum{
    SPI_MODE_WR_RD       = 3,//must enbale CmdEn
    SPI_MODE_WR_DUMMY_RD = 5,//write_dummy_read
}spi_wr_rd_tans_mode_e; .
```

例如在读数据时，SPI Slave 要求要有 dummy 空闲帧，读数据应选择 SPI_MODE_RD_DUMMY_READ 方式。

17.2.2.2 Slave

SoC 的 HSPI 作为 Slave 时，支持 Single、Dual 和 Quad I/O 模式。PSPI 作为 Slave 时，支持 Single、Dual I/O 模式。二者均会自动解析 cmd ，但 Slave 数据接收和发送需要用软件进行操作。

通信数据帧格式

HSPI/PSPI Slave 支持的通信数据帧格式如下表。

(1) HSPI/PSPI SINGLE WRITE

MOSI_IO0	cmd (write 8bit)	dummy (8clock)	data0	...
MISO_IO1	-	-	-	-

(2) HSPI/PSPI SINGLE READ

MOSI_IO0	cmd (read 8bit)	dummy (8clock)	-	...
MISO_IO1	-	-	data0	-

(3) HSPI/PSPI DUAL WRITE

MOSI_IO0	cmd (write 8bit)	dummy (8clock)	D6	D4	D2	D0	...
MISO_IO1	-	-	D7	D5	D3	D1	...

(4) HSPI/PSPI DUAL READ

MOSI_IO0	cmd (read 8bit)	dummy (8clock)	D6	D4	D2	D0	...
MISO_IO1	-	-	D7	D5	D3	D1	...

(5) HSPI/PSPI QUAD WRITE

MOSI_IO0	cmd (write 8bit)	dummy (8clock)	D4	D0	...
MISO_IO1	-	-	D5	D1	...
WP_IO2	-	-	D6	D2	...
HOLD_IO3	-	-	D7	D3	...

(6) HSPI/PSPI QUAD READ

MOSI_IO0	cmd (read 8bit)	dummy (8clock)	D4	D0	...
MISO_IO1	-	-	D5	D1	...
WP_IO2	-	-	D6	D2	...
HOLD_IO3	-	-	D7	D3	...

HSPI/PSPI Slave 支持的操作指令

HSPI/PSPI Slave 支持的操作指令在 Demo 中用枚举定义来表示，使用时对应读写函数接口的 cmd 参数。

```
enum{
    SPI_READ_STATUS_SINGLE_CMD = 0x05,
    SPI_READ_STATUS_DUAL_CMD = 0x15,
    HSPI_READ_STATUS_QUAD_CMD = 0x25,
    SPI_READ_DATA_SINGLE_CMD = 0x0B,
    SPI_READ_DATA_DUAL_CMD = 0x0C,
    HSPI_READ_DATA_QUAD_CMD = 0x0E,
    SPI_WRITE_DATA_SINGLE_CMD = 0x51,
    SPI_WRITE_DATA_DUAL_CMD = 0x52,
    HSPI_WRITE_DATA_QUAD_CMD = 0x54,
};
```

注意：

- 操作指令和 HSPI 或者 PSPI 的模式是相关的，SPI 前缀表示 HSPI 和 PSPI 通用，HSPI 前缀表示只能 HSPI 使用，DUAL_CMD 对应 Dual SPI 模式，QUAD_CMD 对应 Quad SPI 模式。

17.2.2.3 时钟设置

在 SoC 中，HSPI 的时钟源为 hclk，PSPI 的时钟源为 pclk。

Master 时钟

在 Master 模式下时钟计算公式为：

$$F_{Master} = \frac{F_{hclk\&pclk}}{(spi\_clk\_div + 1) * 2}$$

其中：spi_clk_div 为分频系数。

F_{Master} 为 Master 输出的 CLK 频率。

$F_{hclk\&pclk}$ 为时钟源的的频率，其中 HSPI 为 hclk，PSPI 为 pclk。

Master 的时钟频率配置接口为一个宏定义：

```
#define SPI_CLK   500000
```

时钟配置的使能接口为：

```
spi_master_init(SPI_MODULE_SEL, sys_clk.pclk * 1000000 / (2 * SPI_CLK) - 1, SPI_MODE0);
```

实际使用时只要修改 SPI_CLK 宏定义值为对应的时钟频率即可。

注意：

- 当分频系数为 0xff 时，Master 输出时钟频率最快可达到源时钟的频率。
- SPI_CLK 的配置范围在 Demo 中有说明，配置时可以参考，超出配置范围可能会导致通信失败。

Slave 的时钟

Slave 的时钟由 Master 输入，Slave 无需配置相应的时钟，但 Master 输入给 Slave 的时钟，需要满足条件：

$$F_{Master} \leq \frac{F_{hclk\&pclk}}{3}$$

其中：F_{Master} 为 SPI Master 输入给 Slave 的 CLK 频率。

$F_{hclk\&pclk}$ 为 Slave 设备自己内部的时钟源频率，PSPI Slave 对应 pclk 频率，HSPI Slave 对应 hclk 频率

17.2.24 中断

SoC 支持多种 SPI 中断类型，使用时可以根据应用场景灵活配置，各中断类型特征如下表：

模式	相关中断	是否是异常中断	是否需手动清除中断标志位	是 Master 还是 Slave 产生
非 DMA	SPI_RXF_OR_INT：RX FIFO over run 中断。接收数据时，程序读取数据不够快，RX FIFO 将会被新的数据覆盖，这种数据的丢失被称为 over run。	Y	Y	Slave
非 DMA	SPI_TXF_UR_INT：TX FIFO under run 中断。发送数据时，程序写入数据到 TX FIFO 中的速度跟不上发送的速度，发送数据出现间断，这种现象被称为 under run。	Y	Y	Slave
非 DMA	SPI_RXF_INT：RX FIFO 阈值（临界值）中断。在该中断使能的情况下，RX FIFO 数据达到或者超过阈值会触发该中断。	N	Y	Master 和 Slave
非 DMA	SPI_TXF_INT：TX FIFO 阈值（临界值）中断。在该中断使能的情况下，TX FIFO 数据小于或者达到阈值会触发该中断。	N	Y	Master 和 Slave

模式	相关中断	是否是异常中断	是否需手动清除中断标志位	是 Master 还是 Slave 产生
DMA、非 DMA 通用	SPI_END_INT: 数据传输结束中断，一笔数据传输完成会触发该中断。	N	Y	Master 和 Slave
DMA、非 DMA 通用	SPI_SLV_CMD_INT: 当配置成 Slave mode 时，每接收到 1Byte command 会触发该中断。	N	Y	Slave

17.2.2.5 DMA 模式

使用宏定义来选择 DMA 通道，相关接口如下：

```
#define TX_DMA_CHN DMA2
#define RX_DMA_CHN DMA3
hspi_set_tx_dma_config(TX_DMA_CHN);
hspi_set_rx_dma_config(RX_DMA_CHN);
```

示例这里配置 DMA2 为 tx 通道，配置 DMA3 为 rx 通道，PSPI 类似，具体可以在 Demo 中查看。

DMA 模式下判断是否发送、接收完数据使用的相关接口如下：

查询方式:

```
static inline _Bool spi_is_busy(spi_sel_e spi_sel)
```

中断方式:

```
spi_set_irq_mask(SPI_MODULE_SEL, SPI_END_INT_EN);//endint_en
```

注意：

- SPI_END_INT 中断不代表数据传输完毕（只是表示 fifo 数据传输结束，CSN 并没有拉高），产生 SPI_END_INT 中断后，再查询 busy 信号，直到 IDLE 才代表结束。
- 使用 DMA 进行传输时，定义发送（接收）的结构体或者数组要进行四字节对齐，Demo 中通过 __attribute__((aligned(4))) 来体现。
- 使用 DMA 接收 SPI 数据到 Buffer 时，对应的目的地址的 Buffer 大小一定要是 4 的倍数。原因是：每次 DMA 都会送到 Buffer 4 个 Byte。即使配置读取长度不够 4，也会往目的地址写 4 个 Byte。例如，定义数组 Buffer 大小为 5Byte，配置 DMA 从 SPI 读取 5Byte 到 Buffer，这时候 DMA 实际上传送了 2 次共 8 个 Byte 到 Buffer，多的 3 个 Byte 数据会从数组溢出，严重时溢出的数据会覆盖别的变量。这时如果配置数组大小为 8Byte，多的 3 个 Byte 数据就会存放在数组中，不会溢出，避免了潜在的风险。

17.2.2.6 3Line

HSPI/PSPI Master/Slave 支持 3line 模式，需要在 Slave 中使能。

调用的接口为：

```
void spi_set_3line_mode(spi_sel_e spi_sel)
```

注意：

- 3line 模式的读写指令兼容 HSPI/PSPI 的 SINGLE_CMD。

17.2.2.7 多 SPI Slave 结构

对于多 SPI Slave 的应用场景，可以为每个 Slave 都分配一个 CSN 引脚。一笔数据传输完成，CSN 会拉高。这时可以切换 CSN，达到切换 Slave 的效果。

HSPI Master 调用接口如下：

```
void hspi_cs_pin_dis(hspi_csn_pin_def_e pin)
void hspi_cs_pin_en(hspi_csn_pin_def_e pin)
```

对于 PSPI Master 调用接口类似，可在接口的代码中查看。

17.2.2.8 XIP 模式

XIP：eXecute In Place，即芯片内执行，指应用程序可以直接在外置存储设备内取指、译码、执行。HSPI 支持 XIP 模式，可以通过 HSPI 接口扩展 SoC 的地址空间至外置存储设备，为运行数据量庞大应用程序提供了硬件基础。

配置 XIP 模式

通过下面接口对 XIP 模式进行配置：

```
hspi_xip_seq_mode_en();//must
hspi_xip_page_size(4);
hspi_xip_en();
```

代码中的 seq_mode(sequential mode) 表示一种间隔收发模式，它表示把数据分成一个个 2^{page_size} Bytes 大小的块进行间隔收发，每收发完一块就会拉高一次 CS，直至数据传输完毕。

发送指令

通过下面接口发送指令：

```
void hspi_master_write_xip_cmd_data(u8 cmd, u32 addr_offset, u8 data_in, spi_wr_tans_mode_e
↪ wr_mode)
```

数据读写

读写数据调用接口如下：

```
void hspi_master_write_xip(u8 cmd, u32 addr_offset, u8 *data, u32 data_len, spi_wr_tans_mode_e
 ↪  wr_mode)
void hspi_master_read_xip(u8 cmd, u32 addr_offset, u8 *data, u32 data_len, spi_rd_tans_mode_e
 ↪  rd_mode)
```

XIP 片内运行程序

XIP 模式运行程序需要把 PC 指针切换到 XIP 设备中程序对应的地址，然后才可以运行。这里 XIP 对应的基地址为 0x1000000，通过下面两句指令切换 PC 指针到 XIP 设备的对应地址（基地址 0x1000000+ 相对地址 0x00），之后就可以用 XIP 模式运行程序了。

```
__asm__("li t0,0x1000000");
__asm__("jarr t0");
```

17.2.3 SPI Slave

顾名思义，SPI Slave 只支持 Slave 模式。程序中只需要将 SPI Slave 对应 GPIO 配置为 SPI 功能即可。其硬件会自动对接收到的 SPI 数据进行协议解析（读写对应地址的值），并做相应的响应动作。除此之外，SPI Slave 还支持 Dual I/O 模式。

17.2.3.1 通信数据帧格式

SPI Slave 模块支持的通信数据格式如下表。

(1) SPI SLAVE SINGLE WRITE

MOSI_IO0	cmd (write 8bit)	addr(32bit) high -> low	data0	data1	...
MISO_IO1	-	-	-	-	-

(2) SPI SLAVE SINGLE READ

MOSI_IO0	cmd (read 8bit)	addr(32bit) high -> low	dummy (8cycle)	-	-
MISO_IO1	-	-	-	data0	-

(3) SPI SLAVE DUAL WRITE

MOSI_IO0	cmd (write 8bit)	addr(32bit) high -> low	D6	D4	D2	D0	...
MISO_IO1	-	-	D7	D5	D3	D1	...

(4) SPI SLAVE DUAL READ

MOSI_IO0	cmd (read 8bit)	addr(32bit) high -> low	dummy (8cycle)	D6	D4	D2	D0	...
MISO_IO1	-	-	-	D7	D5	D3	D1	...

注意：

- "addr(32bit) high -> low"表示地址的排列顺序高字节在前。

17.2.3.2 SPI Slave 支持的操作指令

SPI Slave 支持的操作指令在 Demo 中用枚举定义来表示，使用时对应读写函数接口的 cmd 参数。

```
typedef enum{
    SPI_SLAVE_WRITE_DATA_CMD = 0x00,
    SPI_SLAVE_WRITE_DATA_DUAL_CMD,
    SPI_SLAVE_WRITE_ADDR_DUAL_CMD,
    SPI_SLAVE_WRITE_DATA_DUAL_4CYC_CMD,
    SPI_SLAVE_WRITE_ADDR_DUAL_4CYC_CMD,
    SPI_SLAVE_WRITE_DATA_AND_ADDR_DUL_4CYC_CMD,
}spi_slave_write_cmd_e;

typedef enum{
    SPI_SLAVE_READ_DATA_CMD,
    SPI_SLAVE_READ_DATA_DUAL_CMD,
    SPI_SLAVE_READ_ADDR_DUAL_CMD,
    SPI_SLAVE_READ_DATA_DUAL_4CYC_CMD,
    SPI_SLAVE_READ_ADDR_DUAL_4CYC_CMD,
    SPI_SLAVE_READ_DATA_AND_ADDR_DUL_4CYC_CMD,
}spi_slave_read_cmd_e;
```

注意：

- 操作指令和 SPI 的模式是相关的，DUAL_CMD 对应 Dual SPI 模式，4CYC_CMD 对应 dummy 4cycle 模式。这些指令调用时是相或的，例如 address 和 data 都支持 Dual I/O 编码，那么读指令格式为：SPI_SLAVE__DATA_DUAL_CMD | SPI_SLAVE_READ_ADDR_DUAL_CMD。

17.3 Demo 说明

17.3.1 Demo 结构说明

SPI Demo 的应用.c 文件分别为 app.c，app_dma.c 和 app_hspi_xip.c，分别对应 DMA、NDMA（非 DMA）、XIP 传输模式。

通过 SPI_Demo/app_config.h 中的宏 SPI_MODE 选择使用哪一种传输模式。

```
#define SPI_NDMA_MODE    1
#define SPI_DMA_MODE     2
#define SPI_XIP_MODE     3
#define SPI_MODE         SPI_NDMA_MODE
```

在 DMA 和 NDMA 传输模式中，通过配置各模式下的宏 SPI_DEVICE 选择为 Master 和 Slave 模式。

```
#define SPI_MASTER_DEVICE      1
#define SPI_SLAVE_DEVICE       2
#define SPI_DEVICE             SPI_MASTER_DEVICE
```

通过宏 SPI_MODULE_SEL 选择使用 HSPI 或者 PSPI 模块。

```
#define PSPI_MODULE          0
#define HSPI_MODULE          1
#define SPI_MODULE_SEL       HSPI_MODULE
```

在 DMA 和 NDMA 传输模式中，通信协议按照 Slave 设备的不同分为三类，通过宏 SPI_TRANS_MODE 选择。

```
#define KITE_VULTURE_SLAVE_PROTOCOL      1
#define HSPI_PSPI_SLAVE_PROTOCOL         2
#define SPI_SLAVE_PROTOCOL               3
#define SPI_TRANS_MODE                   SPI_SLAVE_PROTOCOL
```

KITE_VULTURE_SLAVE_PROTOCOL：是为 Telink 的 Kite（TLSR825x）或 Vulture（TLSR827x）等作为 Slave 的使用场景而设计的模式。

HSPI_PSPI_SLAVE_PROTOCOL：是为 SoC 的 HSPI/PSPI 作为 Slave 的使用场景而设计的模式。

SPI_SLAVE_PROTOCOL：是为 SoC 的 SPI SLAVE 作为 Slave 的使用场景而设计的模式。

注意：

- 通过两个板子测试 SPI 通信时，将 Master 端和 Slave 端的代码都烧到板子之后，先给 Slave 端上电，再给 Master 上电，还有两个板子之间需要稳定共地。

17.3.2 硬件连接

Demo 中不同的 SPI_TRANS_MODE，接线方式会有区别。

针对 KITE_VULTURE_SLAVE_PROTOCOL 的硬件连接如下：

HSPI/PSPI Master (SoC)	Slave (Kite/Vulture)
CLK	CLK
CSN	CSN
MOSI_IO0	SDI
MISO_IO1	SDO

针对 HSPI_PSPI_SLAVE_PROTOCOL 的硬件连接如下：

HSPI/PSPI Master (SoC)	HSPI/PSPI Slave (SoC)
CLK	CLK
CSN	CSN
MOSI_IO0	MOSI_IO0
MISO_IO1	MISO_IO1
WP_IO2(HSPI only)	WP_IO2(HSPI only)
HOLD_IO3(HSPI only)	HOLD_IO3(HSPI only)

针对 SPI_SLAVE_PROTOCOL 的硬件连接如下：

HSPI/PSPI Master (SoC)	SPI Slave (SoC)
CLK	CLK
CSN	CSN
MOSI_IO0	MOSI_IO0
MISO_IO1	MISO_IO1

17.3.3 HSPI/PSPI Master/Slave 的初始化配置

HSPI/PSPI Master/Slave 的初始化流程如下图所示：

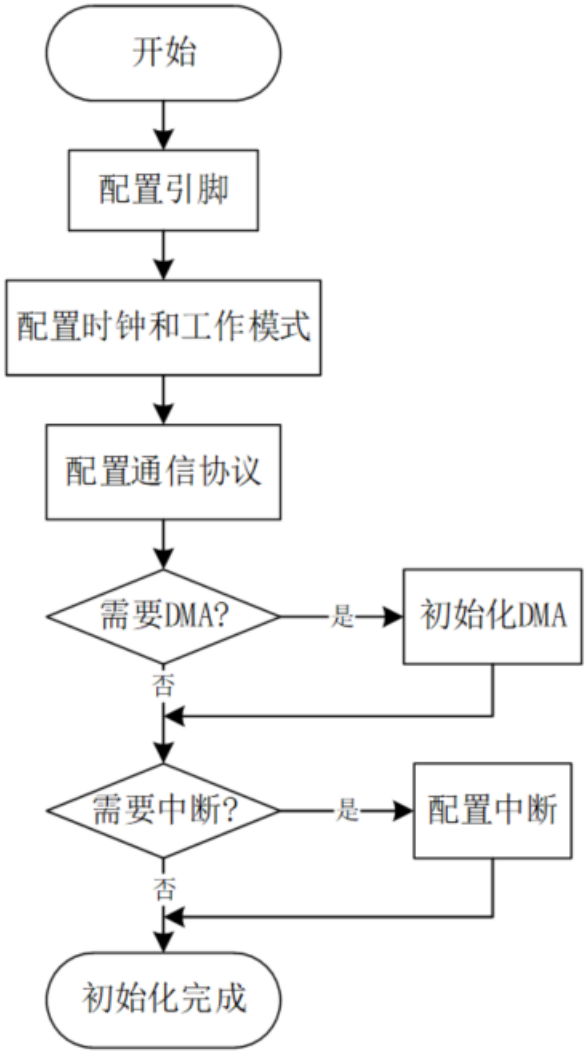


Figure 17.5: HSPI/PSPI Master/Slave 的初始化流程

17.3.4 HSPI/PSPI Master **的读写操作**

HSPI/PSPI Master 的读写操作流程如下图所示：

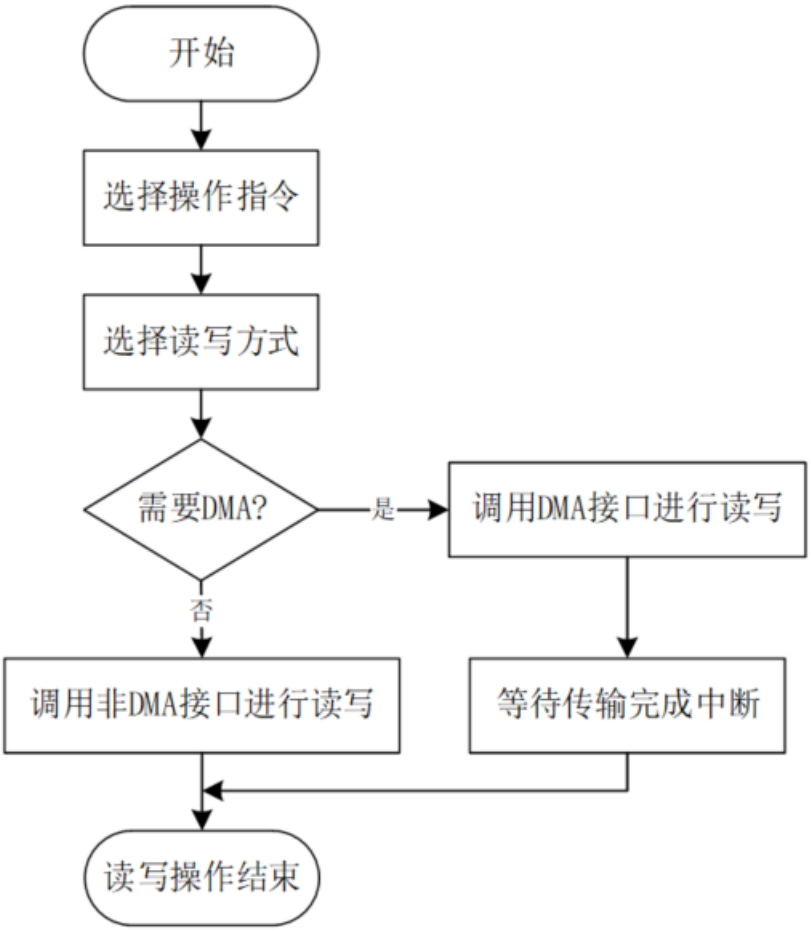


Figure 17.6: HSPI/PSPI Master 的读写操作流程

17.34.1 测试实例

Demo 配置 SPI_SLAVE_PROTOCOL 模式下的 HSPI 为 Dual SPI，HSPI Master 通过 Dual I/O 写命令 SPI_SLAVE_WRITE_DATA_DUAL_CMD | SPI_ SLAVE_WRITE_ADDR_DUAL_CMD 使用 DMA 往 SPI Slave 写入 16Bytes 数据，然后通过 Dual I/O 读命令 SPI_READ_DATA_DUAL_CMD | SPI_READ_ADDR_DUAL_CMD 使用 DMA 读出。测试成功，逻辑分析仪波形如下：

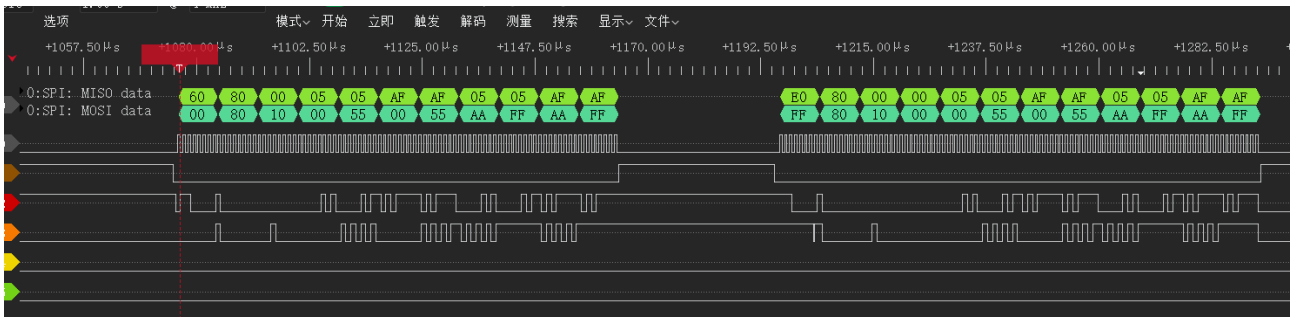


Figure 17.7: 逻辑分析仪波形

17.3.5 SPI_XIP_MODE 模式

Demo 适配的 XIP Device 是一款型号为 APS1604M-3SQR 的 PSRAM，APS1604M-3SQR 支持传统 SPI 和 QUAD SPI 等模式。

17.3.5.1 通信格式

Demo 中对 PSRAM 的通信格式通过宏定义来使能，宏定义及含义如下：

```
#define SPI_XIP_SERIAL_CMD_READ                1
#define SPI_XIP_SINGLE_CMD_FAST_READ           2
#define SPI_XIP_SINGLE_CMD_FAST_QUAD_READ      3
#define SPI_XIP_QUAD_CMD_FAST_READ             4
#define SPI_XIP_QUAD_CMD_FAST_QUAD_READ        5
#define SPI_XIP_LOAD_PROGRAM_TO_PSRAM          6

#define SPI_XIP_TEST_MODE        SPI_XIP_SERIAL_CMD_READ
```

SPI_XIP_SERIAL_CMD_READ: 单线指令数据读写模式。

SPI_XIP_SINGLE_CMD_FAST_READ: 单线指令数据读写的升级模式，支持更高的 CLK。

SPI_XIP_SINGLE_CMD_FAST_QUAD_READ: 单线指令四线数据的读写模式。

SPI_XIP_QUAD_CMD_FAST_READ: 四线指令四线数据的读写模式。

SPI_XIP_QUAD_CMD_FAST_QUAD_READ: 四线指令四线数据的读写的升级模式，支持更高的 CLK。

SPI_XIP_LOAD_PROGRAM_TO_PSRAM: 芯片内运行程序的模式。

注意：

- APS1604M-3SQR 支持的通信格式多种多样，有单线的 cmd 帧，有四线的 cmd 帧，dummy 的个数也随着模式区别而不一样，这些在 Demo 中都有体现。这里仅表明调用接口的含义，如果用户想去具体了解应用，开发时可以参考 Demo 和相关 PSRAM 手册。

17.3.5.2 配置 XIP 模式

APS1604M-3SQR 硬件上要求单次数据传输时间不能超过 8us（有的版本为 4us，具体查看产品手册），否则会有出错风险，所以在配置 XIP 模式的时候要使能 seq_mode。下图 page_size 设置为 1，在 PSRAM 的基地址 0x000000 上写 8 个字节 (0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x77) 数据，数据被分成 4 块，每次发送 2^{page_size} = 2bytes，分四次发送完成。相邻 CSN 拉高之间的时间间隔小于 8us。

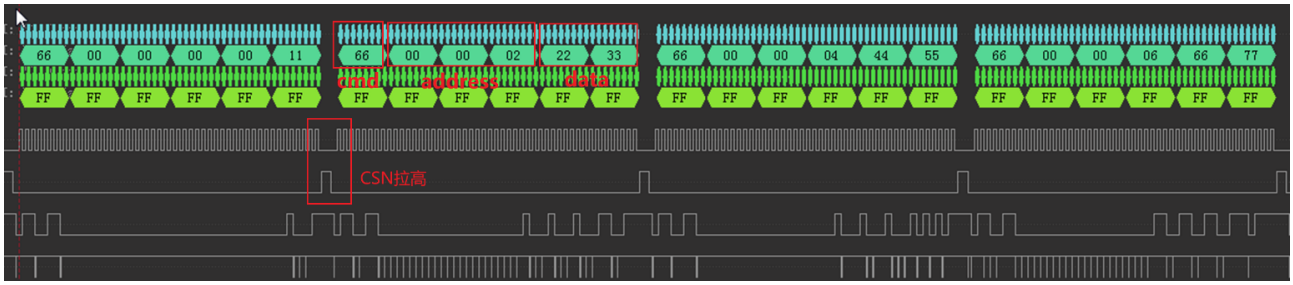


Figure 17.8: 配置 XIP 模式

注意：

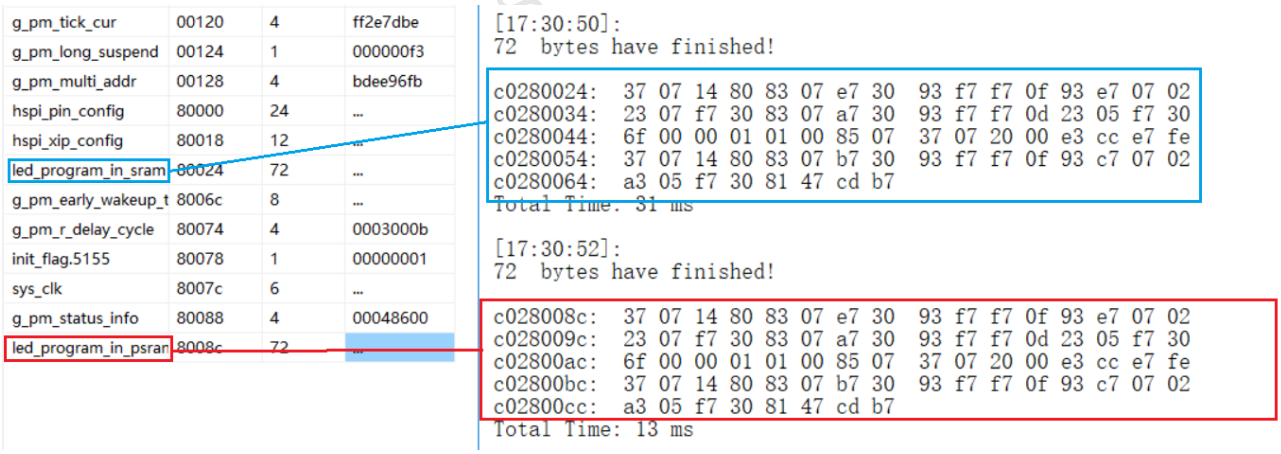
- 因为 APS1604M-3SQR 要求的 8us 间隔，就不得不提高 CLK 频率来减小每次传输 2^{page_size} Bytes 数据所消耗的时间，使其满足 8us 这个要求，所以 XIP 模式下 HSPI 的 SPI_CLK 比一般 SPI 应用要高。当调节 SPI_CLK 无法满足 8us 的要求时，可以通过配置 page_size 的大小来减少每次传输的字节数，增加传输次数，来满足时间要求。

17.3.5.3 测试实例

Demo 配置 HSPI 为 QUAD XIP 模式，通过 HSPI XIP 往 PSRAM 地址 0x00 写入数组 led_program_in_sram 中的 LED 灯闪烁程序，然后跳转到 PSRAM 的对应地址执行该程序。

测试发现 LED2 间隔闪烁，证明 PSRAM 片内运行程序成功。

读取 PSRAM 中的程序到数组 led_program_in_psram，通过 BDT 工具读取数组 led_program_in_psram 和 led_program_in_sram 进行比较，两组数据完全一样，证明读写 PSRAM 成功。



g_pm_tick_cur	00120	4	ff2e7dbe
g_pm_long_suspend	00124	1	000000f3
g_pm_multi_addr	00128	4	bdee96fb
hspi_pin_config	80000	24	...
hspi_xip_config	80018	12	...
led_program_in_sram	80024	72	...
g_pm_early_wakeup_t	8006c	8	...
g_pm_r_delay_cycle	80074	4	0003000b
init_flag.5155	80078	1	00000001
sys_clk	8007c	6	...
g_pm_status_info	80088	4	00048600
led_program_in_psram	8008c	72	...

```
[17:30:50]:
72  bytes have finished!

c0280024:  37 07 14 80 83 07 e7 30  93 f7 f7 0f 93 e7 07 02
c0280034:  23 07 f7 30 83 07 a7 30  93 f7 f7 0d 23 05 f7 30
c0280044:  6f 00 00 01 01 00 85 07  37 07 20 00 e3 cc e7 fe
c0280054:  37 07 14 80 83 07 b7 30  93 f7 f7 0f 93 c7 07 02
c0280064:  a3 05 f7 30 81 47 cd b7
Total Time: 31 ms

[17:30:52]:
72  bytes have finished!

c028008c:  37 07 14 80 83 07 e7 30  93 f7 f7 0f 93 e7 07 02
c028009c:  23 07 f7 30 83 07 a7 30  93 f7 f7 0d 23 05 f7 30
c02800ac:  6f 00 00 01 01 00 85 07  37 07 20 00 e3 cc e7 fe
c02800bc:  37 07 14 80 83 07 b7 30  93 f7 f7 0f 93 c7 07 02
c02800cc:  a3 05 f7 30 81 47 cd b7
Total Time: 13 ms
```

Figure 17.9: 测试实例

18 PM

MCU 正常执行程序时处于 working mode，电流会在 mA 级别。如果需要省功耗需要进入低功耗模式。

18.1 功能说明

低功耗模式（low power mode）又称 sleep mode，有三种，分别是：

(1) Suspend

(2) deep sleep without SRAM retention（下面简称 deep）

(3) deep sleep with SRAM retention（可保留部分 SRAM 空间的数据）（下面简称 deep retention）

每种模式又因唤醒源不同分为 PAD 唤醒、32k_timer 唤醒（分内部 32k、外部 32k 时钟源）、MDEC 唤醒、LPC 唤醒、CORE 唤醒，其中 CORE 唤醒仅支持 suspend 模式，其他唤醒源支持所有模式。

目前支持 deep retention 模式的芯片型号包括：blackHawk(8K)、kite (8K/16k/32k)、vulture (16k/32k)、eagle (32k/64k)。

三种低功耗模式下 Sram、数字寄存器（digital register）、模拟寄存器（analog register）状态如下：

module	suspend	deep retention	deep
SRAM	100% keep	First 16K/32K/64K keep, others lost	100% lost
digital register	99% keep	100% lost	100% lost
analog register	100% keep	99% lost	99% lost

三种低功耗模式介绍如下。

18.1.1 Suspend

Suspend 模式下，程序停止运行，类似一个暂停功能，当 suspend 被唤醒后，程序继续执行。Suspend 模式下，PM 模块正常工作，SRAM 不掉电（数据不丢失），全部的 analog register 不掉电，少量的 digital register 掉电。为了节省功耗，软件可以设置把 RF/USB/Audio 等模块掉电，此时这几个模块相应的部分 digital register 会丢失，例如 RF 在唤醒后需要重新进行初始化才能发包，其余寄存器均不丢失。如果想要在唤醒后直接能够发包，可以不设置相应模块掉电，但是相应的功耗会增加。可以通过 IO、timer 等方式唤醒；这里需要注意的是，pad 唤醒模式下为了避免误触发，需要做相应的上下拉确保初始电平正确，且维持不变直到 pad 触发唤醒。

18.1.2 Deep

Deep 模式下，程序停止运行，MCU 绝大部分的硬件模块都断电，当 deep 被唤醒后，MCU 将重新启动，类似于重新上电，程序重新开始初始化。Deep 模式下，pm 模块正常工作，SRAM 掉电，数据丢失，大部分的 3.3V analog register 会保存，其他的 analog register 掉电，全部的 digital register 掉电。可以通过 IO、Timer 等方式唤醒，SRAM 的数据丢失。

18.1.3 Deep retention

Deep 模式电流很低，但是无法存储 Sram 信息；suspend 模式 Sram 和 register 可以保持不丢，但是电流偏高。为了实现一些需要 sleep 时电流很低又要能够确保 sleep 唤醒后能立刻恢复状态的应用场景，增加了 deep retention 模式。Deep retention 模式更接近 deep 模式，与 deep 的唯一区别是可以保存 Sram，可根据实际需求选择 Sram retention area 的大小，保存的 Sram 越大，功耗就越大。

Deep retention 模式下，程序停止运行，MCU 绝大部分的硬件模块都断电，当 deep retention 被唤醒后，MCU 将重新启动，类似于重新上电，程序重新开始初始化。Deep retention 模式下，pm 模块正常工作，SRAM 保持部分空间不掉电，其他都掉电，大部分的 3.3V analog register 会保存，其他的 analog register 掉电，全部的 digital register 掉电，比 deep 模式增加的电流值就是保持 Sram 所消耗的电流值。可以通过 IO、Timer 等方式唤醒，由于 deep retention 会保存 Sram，因此醒来之后可以省去一部分从 flash 搬代码/数据到 ram 的动作。另外，程序里还可以定义 retention data，定义为 retention data 的变量在唤醒后不会去 flash 取值，直接保存在 SRAM 中，因此会保存最后一次修改的值。

18.14 低功耗模式工作流程

不同的睡眠模式，MCU 的运行流程不一致。下面详细介绍 suspend、deepsleep、deepsleep retention 3 种睡眠模式被唤醒后的 MCU 运行流程。请参考下图。

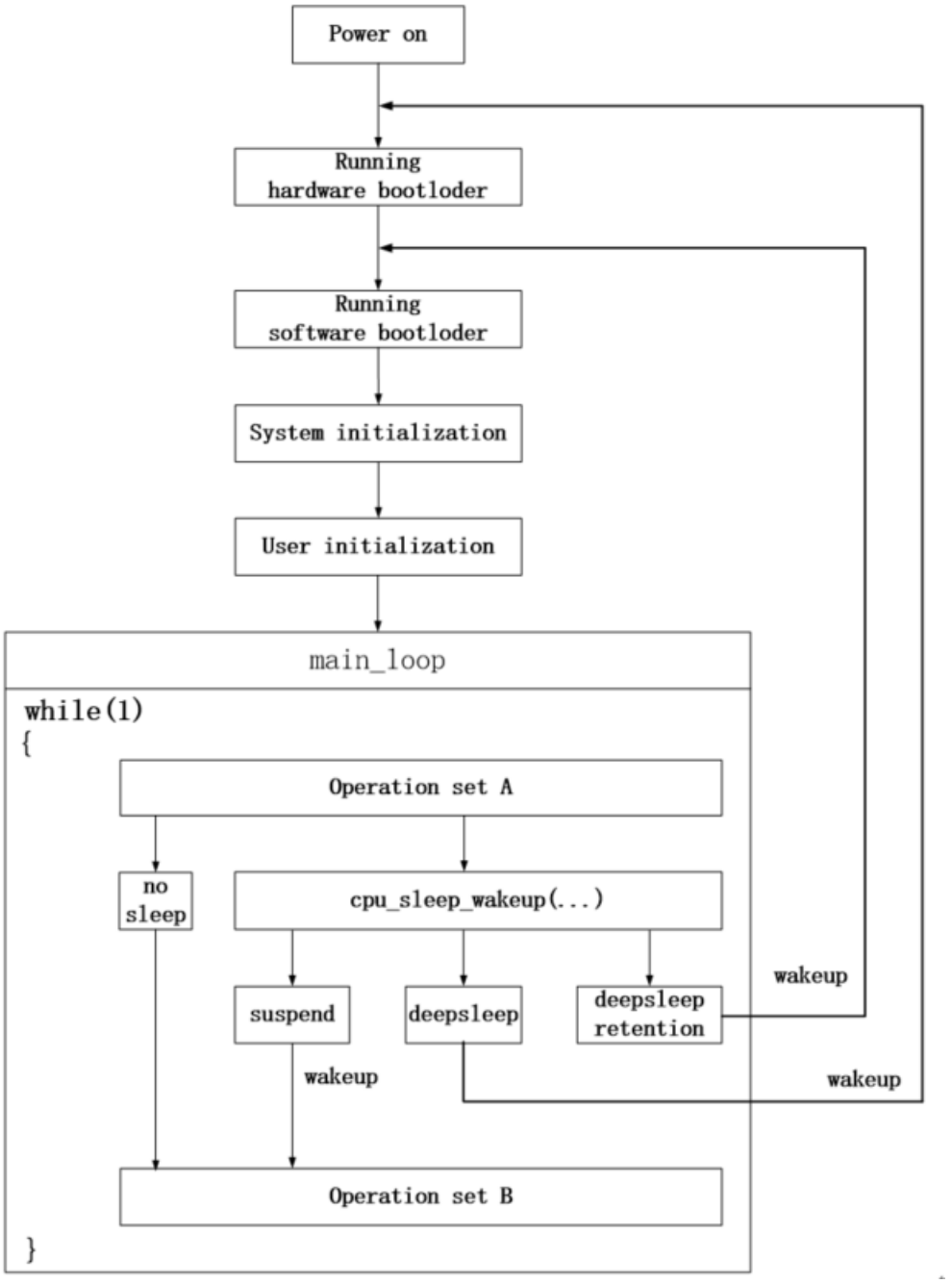


Figure 18.1: MCU 运行流程

流程图中各模块说明：

a) 运行硬件 bootloader (Run hardware bootloader)

MCU 硬件上执行一些固定的动作，这些动作固化在硬件上，软件无法修改。

b) 运行软件 bootloader (Run software bootloader)

hardware bootloader 运行结束之后，MCU 开始运行 software bootloader。Software bootloader 就是 vector 段，对应 S 文件里面的汇编程序。Software bootloader 是为了给后面 C 语言程序的运行设置好内存环境，可以理解为整个内存的初始化。

c) 系统初始化 (System initialization)

System initialization 对应 main 函数中 sys_init、clock_init 等各硬件模块初始化，设置各硬件模块的数字/模拟寄存器状态。

d) 用户初始化 (User initialization)

User initialization 对应函数 user_init 等。

e) main_loop

User initialization 完成后，进入 while(1) 控制的 main_loop。main_loop 中进入 sleep mode 之前的一系列操作称为"Operation Set A"，sleep 唤醒之后一系列操作称为"Operation Set B"。

各睡眠模式的流程分析：

(1) no sleep

没有睡眠，MCU 的运行流程为在 while(1) 中循环，反复执行"Operation Set A"->"Operation Set B"。

(2) suspend

调用 cpu_sleep_wakeup 函数进入 suspend 模式，当 suspend 被唤醒后，类似于 cpu_sleep_wakeup 函数的正常退出，MCU 继续运行到"Operation Set B"。在 suspend 睡眠期间所有的 Sram 数据保持不变，绝大多数的数字/模拟寄存器状态保持不变（只有几个特殊的例外）；所以 suspend 唤醒后，程序接着原来的位置运行，几乎不需要考虑任何 sram 和寄存器状态的恢复。

(3) deepsleep

调用 cpu_sleep_wakeup 函数进入 deep 模式，当 deep 被唤醒后，类似于重新上电，MCU 回到 hardware bootloader 重新运行。在 deep 睡眠期间所有的 Sram 和绝大多数的数字/模拟寄存器都会掉电（只有一些特殊的模拟寄存器例外），所有的软硬件初始化都重新做。

(4) deepsleep retention

deep retention 是介于 suspend 和 deep 之间的一种睡眠模式。调用 cpu_sleep_wakeup 函数进入 deep retention 模式，当 deep retention 被唤醒后，MCU 回到 Run software bootloader 开始运行。在 deep retention 睡眠期间 Sram 只保留一部分 SRAM 不掉电，绝大多数的数字/模拟寄存器都会掉电（只有一些特殊的模拟寄存器例外）。唤醒后，Sram 前面的一部分数据是保持的，可以跳过"Run hardware bootloader"这一步，由于 Sram 上 retention area 有限，"run software bootloader"无法跳过，必须得执行；由于 deepsleep retention 无法保存寄存器状态，所以 system initialization 必须执行，寄存器的初始化需要重新设置。由于程序里还可以定义 retention data，user initialization 可以做一些优化改进，与上电/deep 唤醒后的 user initialization 做区分处理。

18.2 驱动说明

驱动层提供了一些接口资源给上层应用层使用，下面是对这些接口的介绍：

18.2.1 预留保留信息 BUF

芯片进入 deep/retention 状态时，大多数的寄存器都会丢失，芯片预留了 8 个寄存器，在 deep/retention 不会丢失，这样应用层可以记录一些睡眠状态下，想要保存的信息。驱动中定义如下（其中 0x38<0>、0x39<0> 被驱动使用，剩余的留给应用层使用）：

(1) 下面的寄存器在 watchdog、硬件/软件 reset、上电三种情况下会被清零。

```
//watchdog, chip reset, RESET Pin, power cycle
#define PM_ANA_REG_WD_CLR_BUF0      0x38 // initial value 0xff. [Bit0] is already occupied. The
 ↪  customer cannot change!
```

(2) 下面的寄存器只有在重新上电才会清零。

```
#define PM_ANA_REG_POWER_ON_CLR_BUF0   0x39 // initial value 0x00. [Bit0] is already occupied.
 ↪  The customer cannot change!
#define PM_ANA_REG_POWER_ON_CLR_BUF1   0x3a // initial value 0x00
#define PM_ANA_REG_POWER_ON_CLR_BUF2   0x3b // initial value 0x00
#define PM_ANA_REG_POWER_ON_CLR_BUF3   0x3c // initial value 0x00
#define PM_ANA_REG_POWER_ON_CLR_BUF4   0x3d // initial value 0x00
#define PM_ANA_REG_POWER_ON_CLR_BUF5   0x3e // initial value 0x00
#define PM_ANA_REG_POWER_ON_CLR_BUF60x3f // initial value 0x0f
```

18.2.2 状态信息

驱动中定义了一个全局变量 g_pm_status_info，会在 **sys_init** 函数中更新 pm 的相关状态，包含的内容如下：

```
typedef struct{
    unsigned char is_pad_wakeup;  //本次是否是被 pad 唤醒
    unsigned char wakeup_src; //本次是被哪种唤醒源唤醒，包括 PAD、TIMER、MDEC、LPC 和 CORE 的唤醒
     ↪  状态。
    pm_mcu_status mcu_status; //mcu 是从哪种状态回来，包括 power on/watchdog/deep/deep ret 四种
 ↪  状态
    unsigned char rsvd;
}pm_status_info_s;

extern _attribute_aligned_(4) pm_status_info_s g_pm_status_info;
```

注意：

- 唤醒源 WAKEUP_STATUS_TIMER，只要 timer 时间到设定的时间，标志就会自己置起来，即使没有设置 timer 唤醒开启，标志也会置起来，只是不唤醒。清掉标志以后，等到了时间仍然会置起来。
- 唤醒源 WAKEUP_STATUS_CORE，属于数字部分的唤醒方式，唤醒过程数字寄存器不能断电，所以仅支持 suspend 睡眠模式。寄存器配置默认打开 core 唤醒，power on/deep/deep ret 回来唤醒源标志位会置起来，需要忽略这个标志位，suspend 回来唤醒源标志位显示正常。
- 唤醒源 WAKEUP_STATUS_PAD，在进入睡眠时，如果 pad 满足唤醒条件，程序会不进入睡眠，继续往下跑。如果是 deep 模式，driver 睡眠函数设置进入 reboot，程序按照 reboot 处理。如果 retention 模式，driver 睡眠函数未作处理，上层软件可按照 suspend 处理，退出睡眠函数后，需再对 rf 模块初始化。

18.2.3 Suspend power **设置**

接口函数：pm_set_suspend_power_cfg

该函数配置 suspend 的时候，basebend，usb，npe 三个模块是否断电。默认状态时，是全部 power down 的，如果想在 suspend 模式下，可以在进入 suspend 之前调用该函数进行设置。

18.24 LPC **唤醒**

在测试中发现设置 872mv 和 50% 时，2.02V 以下都能唤醒（正常应该是 1.744V 以下才可以唤醒），原因是 LPC 功能在 LPC_LOWPOWER 模式下精度较低。

18.2.5 USB **唤醒**

在测试中发现 USB 唤醒标志会被误置，因为 USB 唤醒的触发条件是 USB 引脚 DP、DM 上有电压变化，产生了数据，所以在设置 USB 唤醒前，需要软件配置把 DP 引脚接上拉，DM 引脚接下拉，这样能保证电平的稳定。

注意：

- DM 引脚接下拉是为了模拟和 host 连接的状态，这只有测试 USB 唤醒的时候才需要这有配置。

18.3 Demo **说明**

18.3.1 **流程说明**

Demo 的流程图如下：

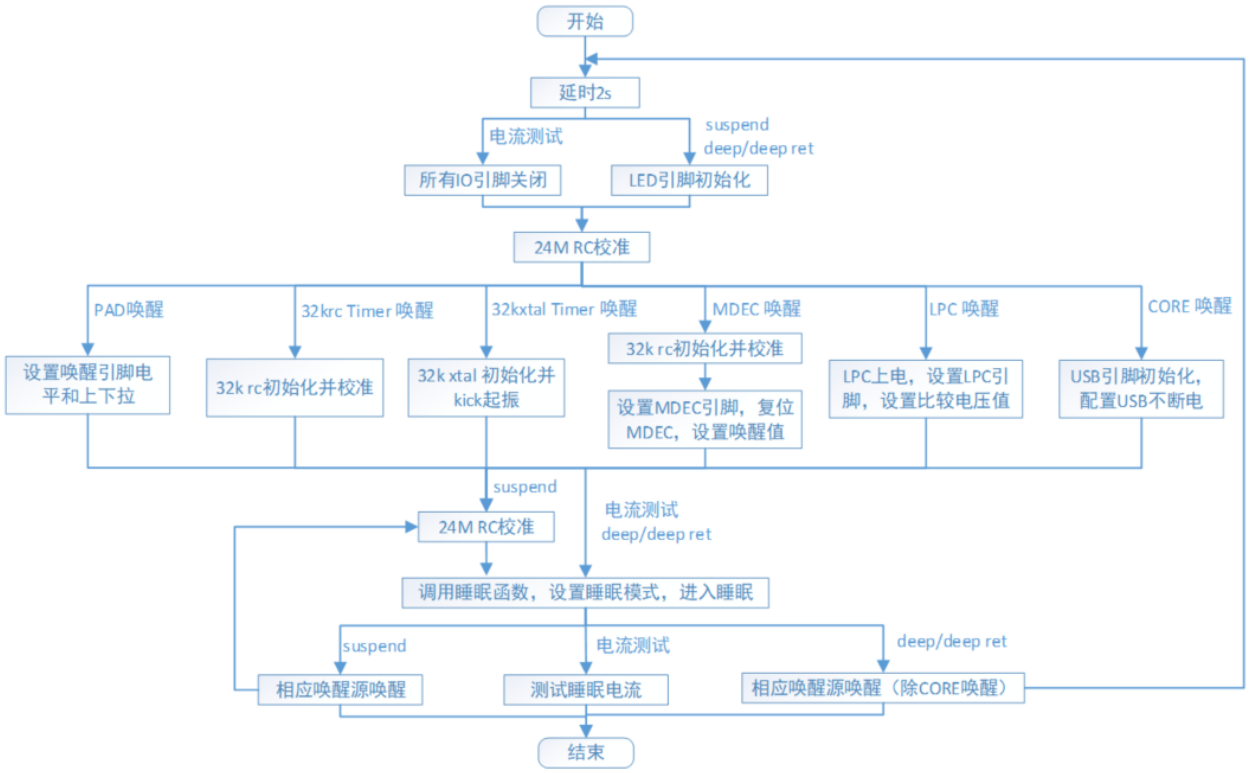


Figure 18.2: Demo 的流程图

流程图说明：

(1) 开始加 2s 的延时是为了保持可以通信，因为进入睡眠后，swire 就不通了，这样使用 BDT 的时候容易 active 失败，不能烧录程序。

(2) 电流测试开始前将所有的 IO 引脚关闭防止漏电。

(3) CORE 唤醒只支持 suspend 睡眠模式，不支持 deep 模式和 deep retention 模式。

(4) Suspend 模式在睡眠前唤醒后打开和关闭 LED2，deep 和 deep retention 模式进入睡眠时，芯片掉电，LED1 自动关闭。（LED 只是为了指示状态）

(5) 因为 RC 时钟是不准确的，并且会随着温度而变化，所以一般需要定时校准。给出以下建议：

a) 24M RC：

可以每 10s 校准一次，睡眠之前校准，这个准确度会影响睡眠后晶振的起振时间，睡眠醒来后，硬件有用 24M RC 的时钟去 kick crystal，时间越准确，则起振时间越快。

b) 32K RC：

在上电、deep 起来会校准一次。因为 PM 中用的是 tricking 的方式（用 16M 去数 32K 的固定周期的时间），所以这里对于 Timer 唤醒的时间准确度是没有影响的。

如果是其他模块有用到 32K RC，需要根据应用需求进行处理。

c) 32K xtal：

上电后都需要再重新 kick。另外使用 32K xtal 的时候需要外面焊接电容。

184 芯片差异

184.1 睡眠电流值

CURRENT_TEST 电流测试宏置 1，测试睡眠电流，单位 uA（此为一颗芯片的测试数据，仅作为参考，具体的可以参照 datasheet）。

B91 A0 芯片的睡眠电流值：

-	Pad ldo	Pad dcdc	32k rc ldo	32k rc dcdc	32k xtal ldo	32k xtal dcdc	mdec ldo	mdec dcdc	lpc ldo	lpc dcdc
deep	0.7	0.7	1.3	1.2	1.7	1.6	1.4	1.4	1.6	1.6
deep retention 32k sram	1.8	1.8	2.4	2.4	2.8	2.7	2.6	2.6	2.8	2.8
deep retention 64k sram	2.7	2.7	3.2	3.2	3.7	3.6	3.4	3.4	3.7	3.8

B91 A1 芯片的睡眠电流值：

#9	Pad ldo	Pad dcdc	32k rc ldo	32k rc dcdc	32k xtal ldo	32k xtal dcdc	mdec ldo	mdec dcdc	comp. ldo	comp. dcdc	core ldo	core dcdc
suspend	36.6	36.7	37.1	37.1	36.8	37.1	37.1	36.8	37.9	37.7	37	36.8
deep	0.6	0.5	1.1	1.1	1.5	1.4	1.1	1.0	1.5	1.5	-	-
deep ret 32k	1.8	1.7	2.3	2.2	2.7	2.5	2.3	2.2	2.7	2.7	-	-
deep ret 64k	2.7	2.6	3.1	3.1	3.5	3.4	3.2	3.1	3.6	3.1	-	-

19 LPC

19.1 简介

低功耗电压比较器（Low Power Compare，后简称 LPC）将乘以所选比例系数的输入电压与参考电压进行比较，并输出比较结果。LPC 有两种工作模式，分别是：

(1) Normal mode, 内部基准电压来自 bandgap（简称 BG），精度较高，功耗大，工作在芯片正常供电的环境下。

(2) Low power mode, 内部基准电压来自 UVLO，精度较低，功耗小，工作在芯片睡眠的环境下。

低功耗比较器的输出也可用作从低功耗模式唤醒系统的信号。

19.2 工作原理

LPC 需要将 32K RC 时钟源用作比较器时钟。比较结果如下：

(1) 如果 [输入电压 * 缩放比例] 的值大于参考电压，则输出将为低（" 0"）。

(2) 如果 [输入电压 * 缩放比例] 的值小于参考电压，则输出将为高（" 1"）。

(3) 如果 [输入电压 * 缩放比例] 的值等于参考电压，或选择输入通道为 float，则输出将不确定。

19.3 Demo 说明

Demo 流程图：

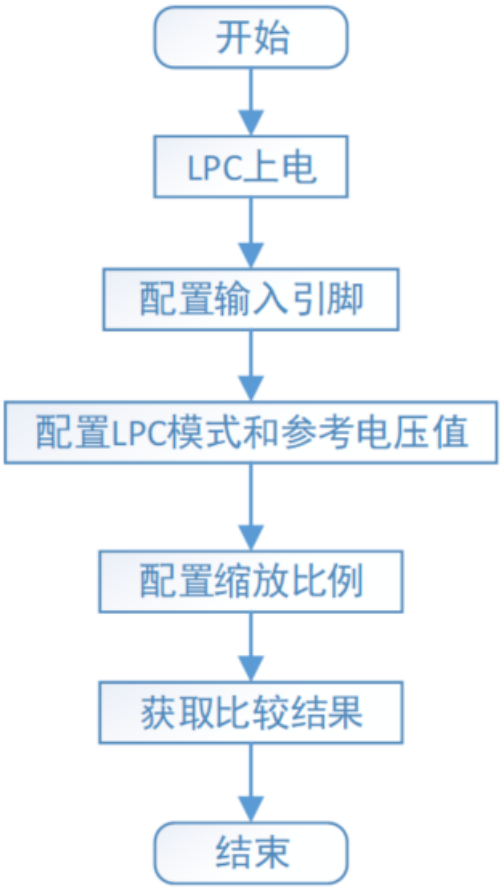


Figure 19.1: Demo 流程图

参考电压值设置为 872mv，缩放比例设置为 50%，那么当输入电压为 0 ~ 1.744V 时，lpc_get_result 返回值为"1"；当输入电压为 1.744 ~ 3.3V 时，lpc_get_result 返回值为"0"。

20 MDEC

MDEC 即曼彻斯特解码模块。

20.1 测试环境搭建

实现 MDEC 功能的工作环境除了开发板之外还有另外两块电路板：

(1) AT9001H-V1.1：接收射频数据，并将数据通过 MCU_1 发送出去。

(2) E21480094v-0：发送射频数据给 AT9001H—V1.1 板子。

如图 AT9001H—V1.1，黑线代表 GND，红线代表 MCU_1。右边的长铁片为 GND，其上方的呈对角状的铁片为 VBAT，接线时注意需将 MCU_1 接入开发板中已设置的 MDEC 功能引脚（图示为 PA0）。电路板 E21480094v—0 红线为 24V+，黑线为 24V-。

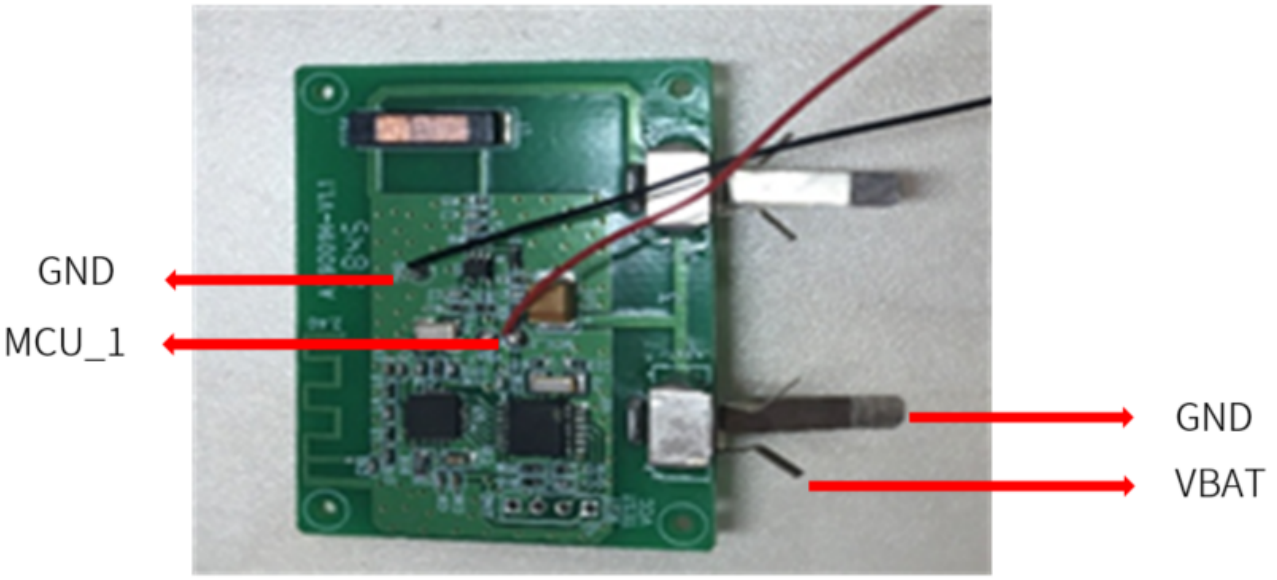


Figure 20.1: 电路板 AT9001H-V1.1

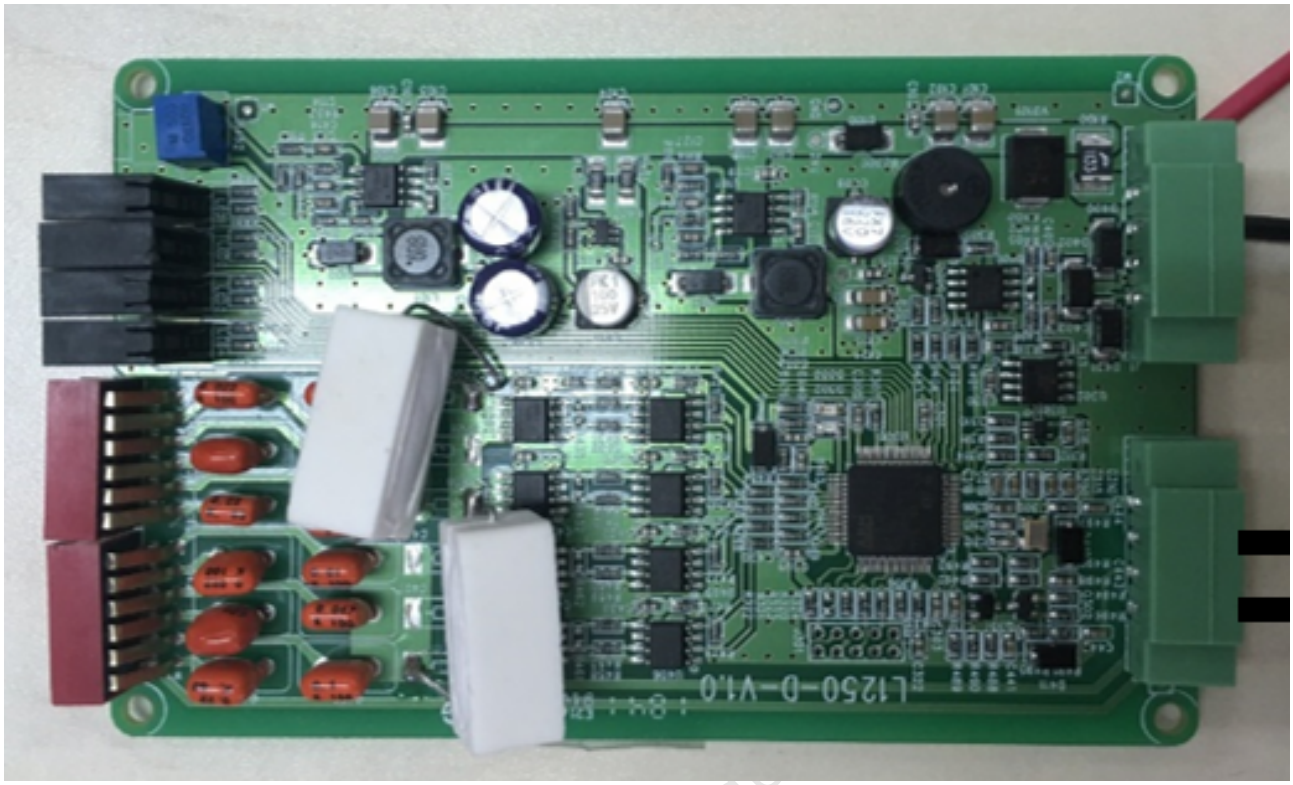


Figure 20.2: 电路板 E21480094v-0

此外，电路板 E21480094v—0 需要 24V 稳压源供电。具体的接线示意图如下：

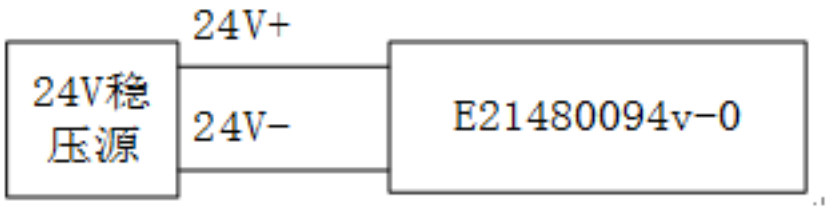


Figure 20.3: 接线示意图

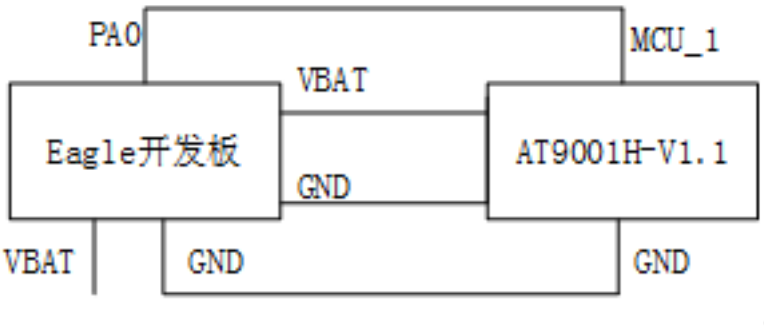


Figure 20.4: 接线示意图

当环境搭建完成并上电后，电路板 E21480094v—0 会连续发送无线包数据内容，当电路板 AT9001H—V1.1 收到后会通过 MCU_1 以电平的方式发送到 SoC 的曼彻斯特接口 IO，注意摆放两块电路板的时候尽量靠近，正确的电平波形如下：

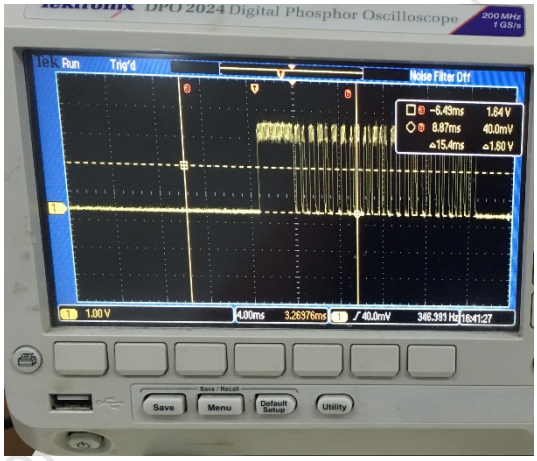


Figure 20.5: 电平波形

注意：

- 如果看到的波形比较乱，请重新摆放一下两块电路板的位置，并调整一下 E21480094v—0 下方接口中的黑色长线圈。

20.2 功能说明

使用 MDEC 模块需要开启 32K 时钟，其中 32K RC 和 32K Xtal 均可。通过读取曼彻斯特输入引脚的电平，获得并判断相关数据。可以用于低功耗模式下的 MDEC 唤醒。

注意：

- 使用 32K 时钟源，是因为 MDEC 设计的时候考虑了在 deep 以及 suspend 等场景中，MDEC 作为唤醒源存在。

21 RF

RF 驱动所支持的收发模式包含了 BLE1M、BLE2M、BLE125K、BLE500K、Zigbee250K、Hybee1M、Hybee2M、Hybee500K、Private1M、Private2M、Private250K、Private500K、ANT 模式。其中 BLE1M、2M 还包含了打开 PN 和关闭 PN 两种模式。

BLE 模式可在符合标准的 1Mbps BLE 模式、2Mbps 增强 BLE 模式、125Kbps BLE 远程模式（S8）、500kbps BLE 远程模式（S2）下工作。Zigbee 模式在符合 IEEE 802.15.4 标准的 250Kbps 模式下工作。

21.1 初始化

RF 模块初始化：

```
rf_mode_init(); //RF initialization
rf_set_ble_1M_mode(); //different modes call different mode initialization, here take BLE_1M
 ↪  mode as an example
rf_set_power_level(RF_POWER); //Set the transmit energy
rf_access_code_comm(ACCESS_CODE);// Zigbee, hybee mode has no access code concept, so this step
 ↪  is not needed
rf_set_tx_dma(0,128);//1 FIFO, each FIFO size is 128bytes
rf_set_rx_dma(rx_packet,RX_FIFO_NUM-1,RX_FIFO_DEP);
rf_set_ble_chn(17);//for BLE open PN 2440MHz
```

如果想要收发包，有两种使用方式可以选择：

(1) Manual Mode：tx、rx 所有的使用过程，由软件流程控制，比如，设置 tx mode，等 PLL 稳定再发包等。

(2) Auto Mode：只要触发对应状态机模式，后面的动作由硬件自动控制，不需要软件控制。

21.2 能量设置

目前设置能量的接口有两个：

```
void rf_set_power_level (rf_power_level_e level)
```

适用模式	参数 chn_num
所有模式	设置能量所对应的枚举变量值

```
void rf_set_power_level_index (rf_power_level_index_e idx)
```

适用模式	参数 chn
所有模式	传入的为能量所对枚举变量在对应数组中的 index 值。

注意：

- 两者区别仅在于传参的方式不一致，应用中可以根据需要选择使用其中的哪一个。

21.3 频点设置

目前设置频点的接口有两个：

```
void rf_set_ble_chn(signed char chn_num)
```

适用模式	参数 chn_num
ble_1M, ble_2M, ble_250K, ble_500K	实际设置的频点为 (chn_num+2400) MHz

```
void rf_set_chn(signed char chn)//所有模式均可以使用
```

适用模式	参数 chn
支持所有模式	传入的为频点的 index 值。通过索引的变换设置为相对应的频点，索引关系如下。

rf_set_ble_chn 函数参数 chn_num 与频点的对应关系如下表：

对应的频点 (MHz)	chn_num
2402	37
2404	0
2406	1
2408	2
2410	3
2412	4
2414	5
2416	6
2418	7

对应的频点 (MHz)	chn_num
2420	8
2422	9
2424	10
2426	38
2428	11
2430	12
2432	13
2434	14
2436	15
2438	16
2440	17
2442	18
2444	19
2446	20
2448	21
2450	22
2452	23
2454	24
2456	25
2458	26
2460	27
2462	28
2464	29
2466	30
2468	31
2470	32
2472	33
2474	34
2476	35

对应的频点 (MHz)	chn_num
2478	36
2480	39

2.14 中断

下面所有中断都需要软件手动清零。

模式	相关中断
Auto	**FLD_RF_IRQ_RX_TIMEOU**: 若从 trigger 到 timeout 设置的时间内仍未收到包，则产生中断，状态机回到 idle 状态。通过 void rf_set_rx_timeout(); 设置 timeout 时间，timeout 起始于 trigger RX。
Auto	**FLD_RF_IRQ_CMD_DONE**: 状态机在完成一次正常的收包或者发包操作后，正常回到 IDLE 状态，产生中断
Auto	**FLD_RF_IRQ_RX_CRC_2**: BTX,BRX,PTX,PRX 在收包 (连续收包) 过程中连续两次检测到 CRC 错误，会产生中断
Auto	**FLD_RF_IRQ_FSM_TIMEOUT**: 含有从接收切换到发送的状态机使用的，规定整个状态机的时间，在 TX_WAIT 状态中判断是否超出规定的时间
Auto	**FLD_RF_IRQ_TX_RETRYCNT**: 在 ptx retry 次数超过设定的 r_max_retry_cnt 产生中断
Auto	**FLD_RF_IRQ_TX_DS**: PTX、PRX 发送的 payload 长度!= 0，产生 tx_ds 中断
Auto	**FLD_RF_IRQ_RX_DR**: PRX、PTX、SRX 接收到的数据包检测到数据包的 payload length ! = 0，产生 rx_dr 中断
Auto	**FLD_RF_IRQ_STX_TIMEOUT**: 在 STX 状态中规定时间内没等到 tx_done，从而 timeout，产生中断
Auto	**FLD_RF_IRQ_INVALID_PID**: PTX 或 PRX 接收到 invalid pid，产生中断
Auto	**FLD_RF_IRQ_FIRST_TIMEOUT**: BRX、PRX、SRX、SRT 第一次 rx timeout，当第一次收包超时时会产生中断
Auto	**FLD_RF_IRQ_WIFI_DENY**: 蓝牙芯片在与 WiFi 芯片连接时接收到 WiFi 芯片发来的 wifi_deny 信号后，蓝牙芯片产生中断
Auto/ Manual	**FLD_RF_IRQ_RX**: 每收完一个包都会产生中断
Auto/ Manual	**FLD_RF_IRQ_TX**: 每发完一个包都会产生中断
Auto/ Manual	**FLD_RF_IRQ_SUPP_OF**: 该中断主要用于 AOA 和 AOD，若 iq 采样频率过高硬件 FIFO 会溢出出错，产生中断 iq sample 同步 fifo overflow

21.5 包格式

每种模式下的收发包数据包在 ram 的格式有所不同，下面根据收包和发包来分别介绍不同模式下的包内容在 ram 中的格式。

所有 mode 中的发包格式的前四个字节都是 DMA_LEN_INFO，可以调用如下函数获得并填入:

```
DMA_LEN_INFO = rf_tx_packet_dma_len (data_len).
```

注意：

- 接收/发送 buffer 必须四字节对齐。例如：unsigned char rx_packet[128*4] __attribute__ ((aligned (4)));

21.5.1 BLE 包格式

21.5.1.1 BLE 发包格式

发送数据包在 RAM 内的格式如下所示（其中，payload length 为数据长度，data 是要发送的数据）：

BLE TX Packet:

Address	Content
addr, addr + 1, addr + 2, addr + 3	DMA_LEN_INFO: rf_tx_packet_dma_len(payload length+2).
addr + 4	header0
addr + 5	header (payload length)
addr + 6	data(0)
addr + 7	data(1)
addr + 8	data(1)
.....	.....
addr +6+(length-1)	data(length-1)

21.5.1.2 BLE 收包格式

当 SoC 处于 BLE 模式进行接收包时，接收到的包数据在 ram 中的存储格式如下表：

BLE 收包格式：

Address	Content	Description
rba-4, rba-3, rba-2, rba-1	-	-

Address	Content	Description
rba	header0	refer to Bluetooth low energy spec
rba+1	header1 (payload length)	indicate length of payload only, do not include 3 crc bytes
rba+2	data(0)	payload
rba+3	data(1)	payload
rba+4	data(2)	payload
.....	.....	payload
rba+2+(length-1)	data(length-1)	payload
rba+2+(length)	crc(0)	crc byte0
rba+2+(length+1)	crc(1)	crc byte1
rba+2+(length+2)	crc(2)	crc byte2
rba+2+(length+3)	r_tstamp[7:0]	time stamp byte0
rba+2+(length+4)	r_tstamp[15:8]	time stamp byte1
rba+2+(length+5)	r_tstamp[23:16]	time stamp byte2
rba+2+(length+6)	r_tstamp[31:24]	time stamp byte3
rba+2+(length+7)	pkt_fdc[7:0]	low byte of recorded frequency offset after demodulation
rba+2+(length+8)	{1'b0,rx_packet_chn_efuse[2:0]}, {1'b0,pkt_fdc[10:8]}	high byte of recorded frequency offset after demodulation
rba+2+(length+9)	pkt_rssi	recorded packet RSSI
-	[0]	crc error
-	[1]	sfd error
-	[2]	link layer error
-	[3]	power error
-	[4]	long range 125K indicator
-	[6:5]	N/A
-	[7]	NoACK indicator

21.5.1.3 BLE **收包数据解析**

根据前面章节介绍的收包格式，接收包中常用的信息可以通过接口获取，已经封装了相关的函数：

函数	说明
rf_ble_packet_crc_ok(p)	判断收包 CRC 是否正确
rf_ble_dma_rx_offset_crc24(p)	获取 CRC 在包中位置的 index
rf_ble_dma_rx_offset_time_stamp(p)	获取 time_stamp 在包中的 index 值
rf_ble_dma_rx_offset_freq_offset(p)	获取 frequency offset 在包中的 index 值
rf_ble_dma_rx_offset_rssi(p)	获取 rssi 在包中的 index 值

21.5.14 **收包解析示例**

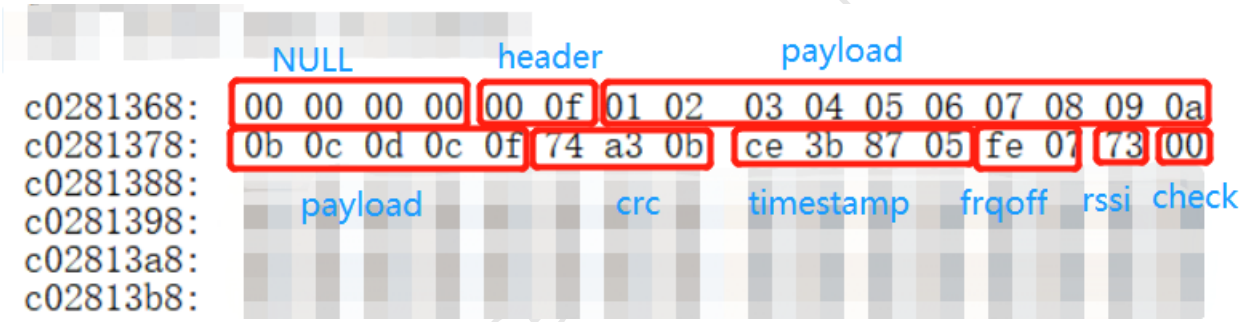


Figure 21.1: 收包解析示例

21.5.2 Zigbee/hybee **包格式**

21.5.2.1 Zigbee/hybee **发包格式**

Zigbee 250K 以及 hybee 模式的发包数据在 ram 中的存储格式如下表：

Zigbee/Hybee TX Packet:

Address	Content
tba, tba + 1, tba + 2, tba + 3	DMA_LEN_INFO: rf_tx_packet_dma_len(payload length-1)
tba + 4	Payload length
tba + 5	data(0)
tba + 6	data(1)
tba + 7	data(1)
.....	.....

Address	Content
tba+5+(length-3)	data(length-3)

21.5.2.2 Zigbee/hybee 收包格式

当 SoC 处于 zigbee 250K 或者 hybee 模式下收包，收到的包数据在 ram 中存储格式如下表所示：

Zigbee/hybee 收包格式:

Address	Content	Description
rba-4, rba-3, rba-2, rba-1	-	-
rba	length (payload+crc)	indicate length of payload and 2 crc bytes
rba+1	data(0)	payload
rba+2	data(1)	payload
rba+3	data(2)	payload
.....	.....	payload
rba+1+(length-3)	data(length-3)	payload
rba+1+(length-2)	crc(0)	crc byte0
rba+1+(length-1)	crc(1)	crc byte1
rba+1+(length)	r_tstamp[7:0]	time stamp byte0
rba+1+(length+1)	r_tstamp[15:8]	time stamp byte1
rba+1+(length+2)	r_tstamp[23:16]	time stamp byte2
rba+1+(length+3)	r_tstamp[31:24]	time stamp byte3
rba+1+(length+4)	pkt_fdc[7:0]	low byte of recorded frequency offset after demodulation
rba+1+(length+5)	{1'b0,rx_packet_chn_efuse[2:0]} {1'b0,pkt_fdc[10:8]}	high byte of recorded frequency offset after demodulation
rba+1+(length+6)	pkt_rssi	recorded packet RSSI
rba+1+(length+7)	[0]	crc error
rba+1+(length+7)	[1]	sfd error
rba+1+(length+7)	[2]	link layer error
rba+1+(length+7)	[3]	power error

Address	Content	Description
rba+1+(length+7)	[4]	long range 125K indicator
rba+1+(length+7)	[6:5]	N/A
rba+1+(length+7)	[7]	NoACK indicator

21.5.2.3 收包数据解析

根据包格式对接收到包内信息的 index 进行获取

函数	说明
rf_zigbee_packet_crc_ok(p)	判断收包 CRC 是否正确
rf_zigbee_dma_rx_offset_crc(p)	获取 CRC 在包中位置的 index
rf_zigbee_dma_rx_offset_time_stamp(p)	获取 time_stamp 在包中的 index 值
rf_zigbee_dma_rx_offset_freq_offset(p)	获取 frequency offset 在包中的 index 值
rf_zigbee_dma_rx_offset_rssi(p)	获取 rssi 在包中的 index 值

21.5.24 收包解析示例

根据前面章节中的 zigbee/hybee 模式下收包的存储格式，可以对收到的数据进行解析，示例如下：

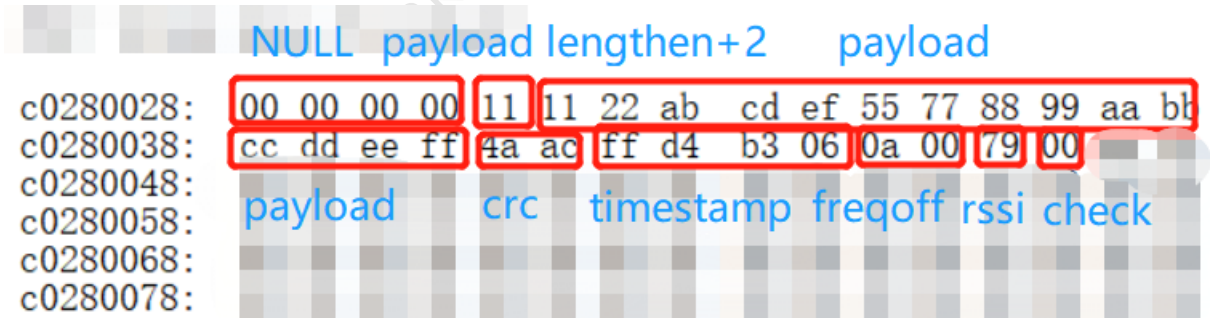


Figure 21.2: 收包解析示例

21.6 Private 包格式

Private 模式下可以分为 SB 和 TPLL(Telink Proprietary Link Layer) 两种，本节分别介绍两者的收发包格式

21.6.1 Private TPLL 发包格式

当 SoC 在 private 模式下采用 TPLL 的方式发包时，包数据的存储格式如下表：

Private TPLL TX Packet:

Address	Content
tba, tba + 1, tba + 2, tba + 3	DMA_LEN_INFO: rf_tx_packet_dma_len(payload length).
tba + 4	Payload length
tba + 5	data(0)
tba + 6	data(1)
.....	.....
tba+5+(length-1)	data(length-1)

21.6.2 Private TPLL 收包格式

当 SoC 处于 Private TPLL 接收数据包时，接收到的数据包在 RAM 中的格式如下所示：

Private TPLL 模式下的收包格式:

Address	Content	Description|
rba-4, rba-3	0	-
rba-2	-	-
rba-1	-	-
rba	payload length	indicate length of payload only, do not include 2 crc bytes
rba+1	data(0)	payload
rba+2	data(1)	payload
rba+3	data(2)	payload
.....	.....	payload
rba+1+(length-1)	data(length-1)	payload
rba+1+(length)	crc(0)	crc byte0
rba+1+(length+1)	crc(1)	crc byte1
rba+1+(length+2)	r_tstamp[7:0]	time stamp byte0
rba+1+(length+3)	r_tstamp[15:8]	time stamp byte1
rba+1+(length+4)	r_tstamp[23:16]	time stamp byte2
rba+1+(length+5)	r_tstamp[31:24]	time stamp byte3

Address	Content	Description|
rba+1+(length+6)	pkt_fdc[7:0]	low byte of recorded frequency offset after demodulation
rba+1+(length+7)	{1'b0,rx_packet_chn_efuse[2:0]} {1'b0,pkt_fdc[10:8]}	high byte of recorded frequency offset after demodulation
rba+1+(length+8)	pkt_rssi	recorded packet RSSI
rba+1+(length+9)	[0]	crc error
rba+1+(length+9)	[1]	sfd error
rba+1+(length+9)	[2]	link layer error
rba+1+(length+9)	[3]	power error
rba+1+(length+9)	[4]	long range 125K indicator
rba+1+(length+9)	[6:5]	N/A
rba+1+(length+9)	[7]	NoACK indicator

21.6.3 TPLL **收包解析**

根据包格式对接收到包内信息的 index 进行获取

函数	说明
rf_pri_tpll_packet_crc_ok(p)	判断收包 CRC 是否正确
rf_pri_tpll_dma_rx_offset_crc(p)	获取 CRC 在包中位置的 index
rf_pri_tpll_dma_rx_offset_time_stamp(p)	获取 time_stamp 在包中的 index 值
rf_pri_tpll_dma_rx_offset_freq_offset(p)	获取 frequency offset 在包中的 index 值
rf_pri_tpll_dma_rx_offset_rssi(p)	获取 rssi 在包中的 index 值

21.64 TPLL 收包解析示例

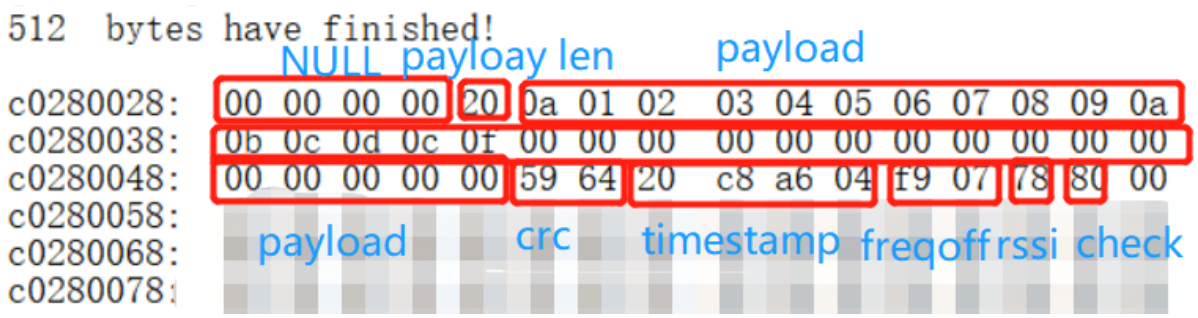


Figure 21.3: TPLL 收包解析示例

21.6.5 Private SB 发包格式

当 SoC 处于 private SB 模式下进行发包时，数据在 ram 中的存储结构如下表：

Private SB TX Packet:

Address	Content
tba, tba + 1, tba + 2, tba + 3	DMA_LEN_INFO: rf_tx_packet_dma_len(payload length).
tba + 4	Payload length
tba + 7	data(1)

注意：

- 在 private SB 模式下发包格式中不包含 payload 长度信息，所以发包格式有所不同。Payload 长度可以通过函数 void rf_set_private_sb_len() 进行设置。

21.6.6 Private SB 收包格式

当 SoC 处于 private SB 模式进行收包时，收到的包数据在 ram 中的存储数据如下表所示：

Private SB 接收数据包格式:

Address	Content	Description
rba-4, rba-3	0	-
rba-2	-	-
rba-1	-	-
rba	data(0)	payload
rba+1	data(1)	payload

Address	Content	Description
rba+2	data(2)	payload
.....	.....	payload
rba+1+(length-1)	data(length-1)	payload
rba+1+(length)	crc(0)	crc byte0
rba+1+(length+1)	crc(1)	crc byte1
rba+1+(length+2)	r_tstamp[7:0]	time stamp byte0
rba+1+(length+3)	r_tstamp[15:8]	time stamp byte1
rba+1+(length+4)	r_tstamp[23:16]	time stamp byte2
rba+1+(length+5)	r_tstamp[31:24]	time stamp byte3
rba+1+(length+6)	pkt_fdc[7:0]	low byte of recorded frequency offset after demodulation
rba+1+(length+7)	{1'b0,rx_packet_chn_efuse[2:0]} {1'b0,pkt_fdc[10:8]}	high byte of recorded frequency offset after demodulation
rba+1+(length+8)	pkt_rssi	recorded packet RSSI
rba+1+(length+9)	[0]	crc error
rba+1+(length+9)	[1]	sfd error
rba+1+(length+9)	[2]	link layer error
rba+1+(length+9)	[3]	power error
rba+1+(length+9)	[4]	long range 125K indicator
rba+1+(length+9)	[6:5]	N/A
rba+1+(length+9)	[7]	NoACK indicator

21.6.7 SB 收包解析示例

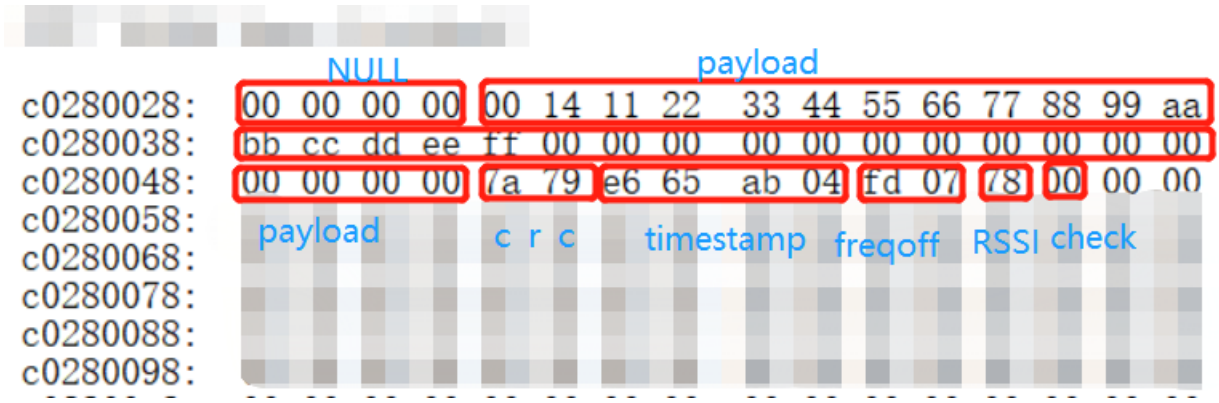


Figure 21.4: SB 收包解析示例

注意：

- 所有模式的 demo 框架基本一致，但在 private 模式下还需要区分 TPLL 和 SB 两种情况；在使用 TPLL 方式时操作基本与其他模式一致，但当使用 SB 模式收发包时需要配置相应的寄存器打开 SB 模式，对应接口 void rf_private_sb_en(void)。同时因为 SB 模式下包格式中未包含 payload length 信息，因此需要对接收和发送端 payload length 进行设置，对应接口 void rf_set_private_sb_len(int pay_len)。

21.7 Manual mode

21.7.1 Manual TX

21.7.1.1 单频发送

在 manual TX 模式下进行单频点发送，设置步骤：

```
rf_set_txmode(); //进入 tx 模式等待触发发包
delay_us(113); //等待 PLL 稳定
while(1)
{
    rf_tx_pkt(ble_tx_packet); //触发发包
    while(! (rf_get_irq_status(FLD_RF_IRQ_TX))); //等待发包结束
    rf_clr_irq_status(FLD_RF_IRQ_TX); //清除中断标志
}
```

注意：

- 在 manual 模式下，tx_settle 时间需要人为的去控制等待 settle 阶段的 PLL 稳定。Settle 时间最小为 112.5us，由于在 manual 模式下设置完 rf_set_txmode() PLL 一直在工作，所以该动作只需要在打开 manual tx 时做一次。

21.7.1.2 跳频发送

若在发送过程中需要切换频点，则需要等待一个包发送完成才能进行频点切换并进行下一次发包，设置步骤：

```
//首先完成初始化动作
rf_set_ble_chn(17);// 2440MHz
rf_set_txmode();
delay_us(113);//等待 PLL 稳定
rf_tx_pkt(ble_tx_packet);//触发发包
while(! (rf_get_irq_status(FLD_RF_IRQ_TX))); //等待发包完成
rf_clr_irq_status(FLD_RF_IRQ_TX); //清中断状态
//跳频发送时通常需要等待前一个包发完再进行频点切换，然后触发下一次发包。
rf_set_ble_chn(37); //切换频点到 2402MHz
rf_tx_pkt(ble_tx_packet); //再次触发发包
```

21.7.2 Manual RX

21.7.2.1 单频接收

在完成初始化设置后如果需要使用 manual 模式进行发包，可以通过 rf_set_rxmode() 接口进行设置进入收包状态。设置步骤：

```
rf_set_rxmode();//进入收包模式，该阶段收不到包，需等待 PLL 稳定
    delay_us(85); //等待 PLL 稳定之后进入真正收包阶段，如果不切换状态的话，会一直处于 RX 状态，一
↪   直可以收包
    while(1)
    {
        if(rf_get_irq_status(FLD_RF_IRQ_RX)) //判断收包是否完成
        {
            if(rf_ble_packet_crc_ok(rx_packet)) //判断收到包 CRC 是否正确
            {
                rx_cnt++; //收包计数
            }
            rf_clr_irq_status(FLD_RF_IRQ_RX); //清收包中断状态
        }
    }
```

注意：

- 在 manual 模式下，rx_settle 时间需要人为的去控制等待 settle 阶段的 PLL 稳定。Settle 时间最小为 85us，由于在 manual 模式下设置完 rf_set_rxmode() PLL 一直在工作，所以该动作只需要在打开 manual rx 时做一次。
- 当收完一个包后 manual 模式下 PLL 一直处于打开状态，所以在收完一个包后清除中断标志就会进入下一个收包状态。

```
void rf_set_rx_dma (unsigned char *buff,unsigned char wptr_mask, unsigned short fifo_byte_size)
```

函数说明：

参数	说明
buff	收包地址
wptr_mask	收包时 dma 写指针 mask = FIFO 个数 - 1
fifo_byte_size	FIFO 大小

注意：

- manual 模式下，由于硬件不会维护读写指针，因此只会使用一个 FIFO 进行接收，所以在设置 dma 时 FIFO 个数通常设置为 1，设置示例：rf_set_rx_dma(rx_packet,0,128) 一个 FIFO，FIFO 大小为 128byte。

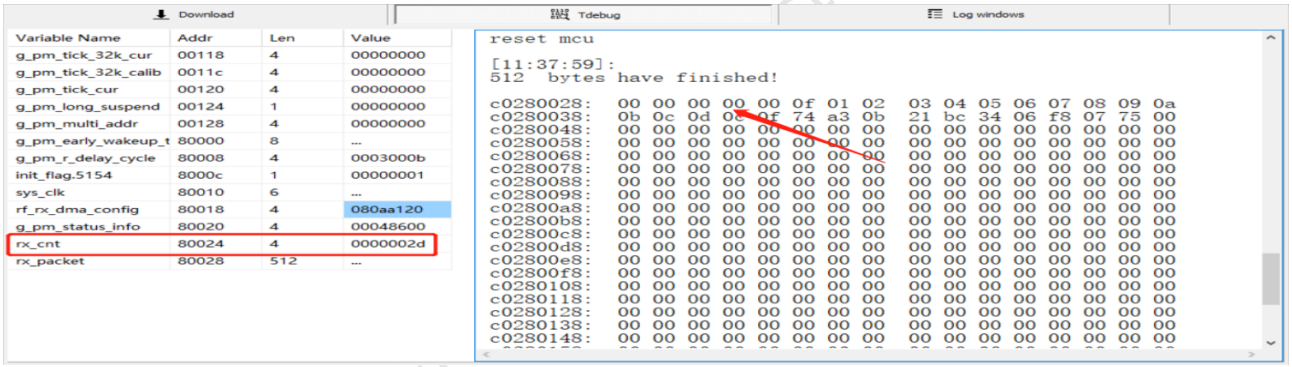


Figure 21.5: manual 模式设置示例

21.7.2.2 跳频接收

如在收包过程中需要进行跳频接收，设置示例如下：

```
// 首先完成初始化动作
rf_set_ble_chn(17);// 2440MHz
rf_set_rxmode();//进入收包模式
delay_us(85);//等待 PLL 稳定，稳定后进入 actual 收包阶段
if(rf_get_irq_status(FLD_RF_IRQ_RX))
{
    if(rf_ble_packet_crc_ok(rx_packet))
    {
                rx_cnt++;
    }
    rf_clr_irq_status(FLD_RF_IRQ_RX);
```

```
}
while(rf_receiving_flag());//等待收包结束（目前使用上会等待收包完成，收包状态直接打断是否存在问
↪ 题仍在确认）
rf_set_ble_chn(37);//切换频点到 2402MHz 等待下一次收包
```

21.7.2.3 发送接收切换

如在运行过程中需要切换收发模式，代码示例：

```
// 首先完成初始化动作
rf_set_ble_chn(17);// 2440MHz
rf_set_txmode();//进入收包模式
delay_us(113);//等待 PLL 稳定，稳定后进入 actual 收包阶段
while(! (rf_get_irq_status(FLD_RF_IRQ_TX))); //等到发包结束
rf_clr_irq_status(FLD_RF_IRQ_TX); //清除中断状态
//切换状态的过程中需要等待上一个状态结束后才能进行状态切换
rf_tx_rx_off();//关闭 tx，rx
rf_set_rx_mode(); //进入 rx 状态
delay_us(85);//等待 PLL 稳定
```

21.8 Auto mode

21.8.1 STX

在 Auto mode 下状态机的工作流程如下：

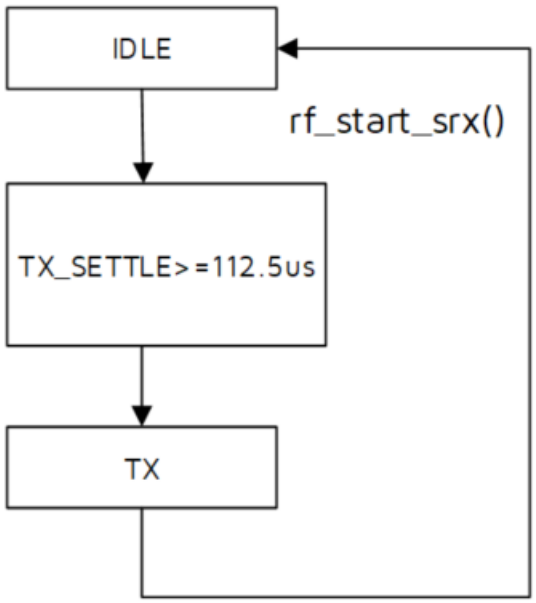


Figure 21.6: Auto mode 下状态机的工作流程

调用函数 void rf_start_stx (void* addr, unsigned int tick) 触发 STX，进入 tx settle，settle 结束进入 actual tx 状态。

```
void rf_start_stx (void\* addr, unsigned int tick)
```

函数说明：

参数	说明
addr	发包地址
tick	当前 tick 值大于等设置 tick 时立即触发

注意：

- TX_SETTLE 时间默认值为 150us，可以调用接口 void rf_tx_settle_us(unsigned short txstl_us) 对 settle 时间进行调整，但是 tx settle 时间不应小于 112.5us。

21.8.1.1 单频发送

调用 **rf_start_stx** 函数将触发状态机进入 TX（包含 tx settle），发包完成，回到 IDLE 状态，设置步骤：

```
// 首先完成初始化动作
rf_start_stx(ble_tx_packet,clock_time());//触发第一次发包，发包完成后状态机自动回到 IDLE 状态
    while(1)
    {
        while(! (rf_get_irq_status(FLD_RF_IRQ_TX))); //判断发包是否完成
        rf_clr_irq_status(FLD_RF_IRQ_TX);//
        rf_start_stx(ble_tx_packet,clock_time());
    }
```

21.8.1.2 跳频发送

若在发包过程中需要切换频点，设置步骤如下：

```
// 首先完成初始化动作
rf_set_ble_chn(17);// 2440MHz
rf_start_stx(ble_tx_packet,clock_time());//触发第一次发包，发包完成后状态机自动回到 IDLE 状态
while(! (rf_get_irq_status(FLD_RF_IRQ_TX)));//判断发包是否完成
rf_clr_irq_status(FLD_RF_IRQ_TX);//清除发包中断状态
rf_set_ble_chn(37);//2402MHz 切换频点
rf_start_stx(ble_tx_packet,clock_time());//触发再次发包
```

21.8.2 SRX

Auto mode 下收包时状态机的工作过程如下图：

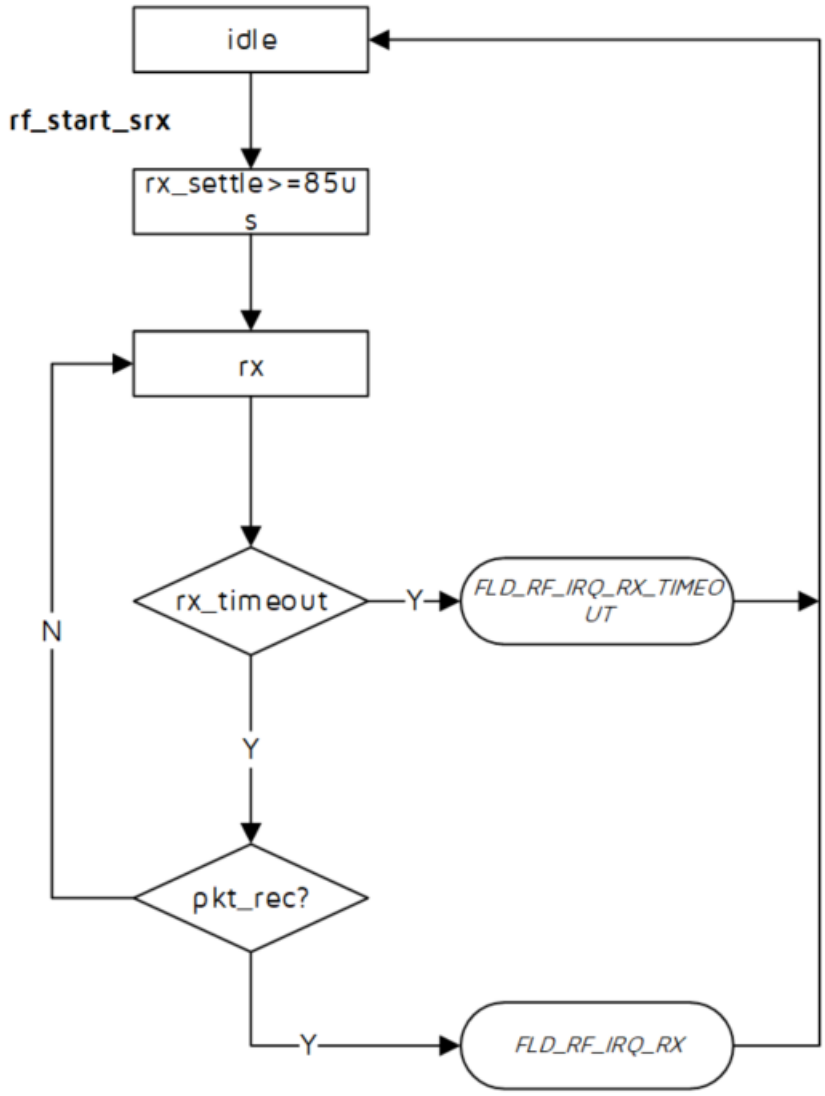


Figure 21.7: Auto mode 下收包时状态机的工作过程

auto mode 使用函数 void rf_start_srx(unsigned int tick); 触发 SRX, 进入 rx settle，settle 结束进入 actual rx，收包完成自动回到 idle 状态，在规定时间内没有同步到包则触发 FLD_RF_IRQ_RX_TIMEOUT 中断。

```
void rf_start_srx (unsigned int tick)
```

函数说明：

参数	说明
tick	当前 tick 值大于等设置 tick 时立即触发

注意：

- rx_settle 时间默认值为 150us，可以调用接口 void rf_rx_settle_us(unsigned short txstl_us) 对 settle 时间进行调整，但是 tx settle 时间不应小于 85us。

21.8.2.1 单频接收

如在 auto 模式下进行单频点接收，设置步骤：

```
// 首先完成初始化动作
rf_start_srx(clock_time());//触发后状态机会从 IDLE 状态进入同步状态，同时 timeout 开始计时
    while(1)
    {
        if(rf_get_irq_status(FLD_RF_IRQ_RX))//判断收包是否结束
        {
            u8* raw_pkt = rf_get_rx_packet_addr(RX_FIFO_NUM,RX_FIFO_DEP,rx_packet);//查找当前收
↪  包地址，仅 auto mode 下需要。
            if(rf_ble_packet_crc_ok(raw_pkt))//判断包的 CRC 是否正确
            {
                rx_cnt++;//记录收包数
                rf_clr_irq_status(FLD_RF_IRQ_RX);//清除中断状态
                rf_start_srx(clock_time());//触发下次收包
        }
     }
```

注意：

- 在 auto mode 下 DMA 会根据 FIFO 个数和 FIFO 深度在往 ram 地址中搬运数据时自动进行偏移。因此每次收包完成后需要通过 rf_get_rx_packet_addr(int fifo_num,int fifo_dep,void* addr) 来查找本次收包内容在 ram 中的地址进行 crc 校验。

rf_get_rx_packet_addr 函数说明如下：

参数	说明
fifo_num	FIFO 个数
fifo_dep	每个 FIFO 大小
addr	收包地址

当使用函数 rf_set_rx_dma(rx_packet,3,128); 设置 DMA 时会有 4 个 FIFO 进行收包。

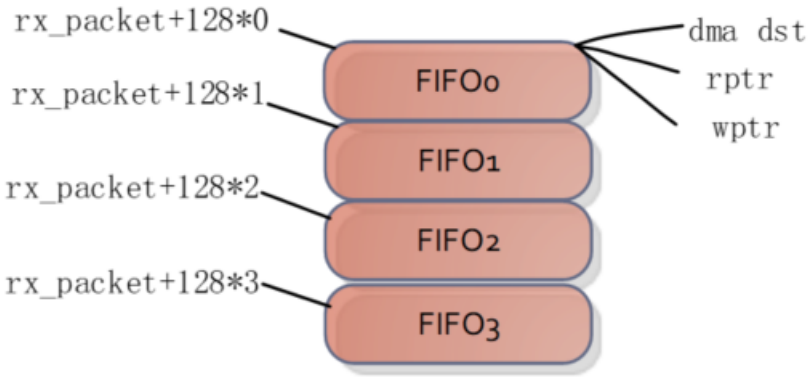


Figure 21.8: 收第一个包前

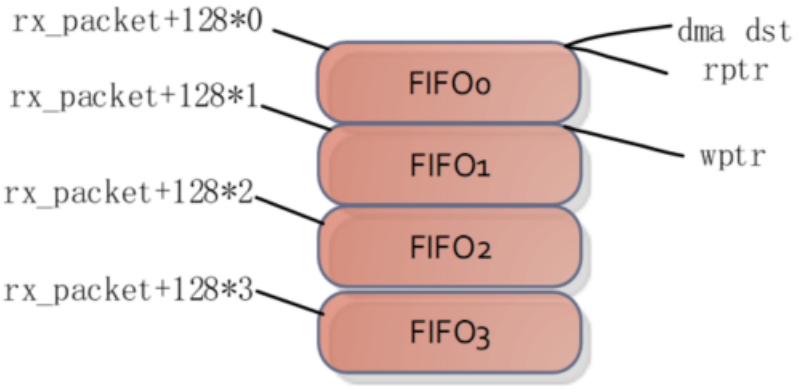


Figure 21.9: 收第二个包前

直到 4 个 FIFO 收满，回到第一个 FIFO。下图为 auto 模式下收包结果，其中 FIFO 个数为 4，FIFO 大小为 128byte。

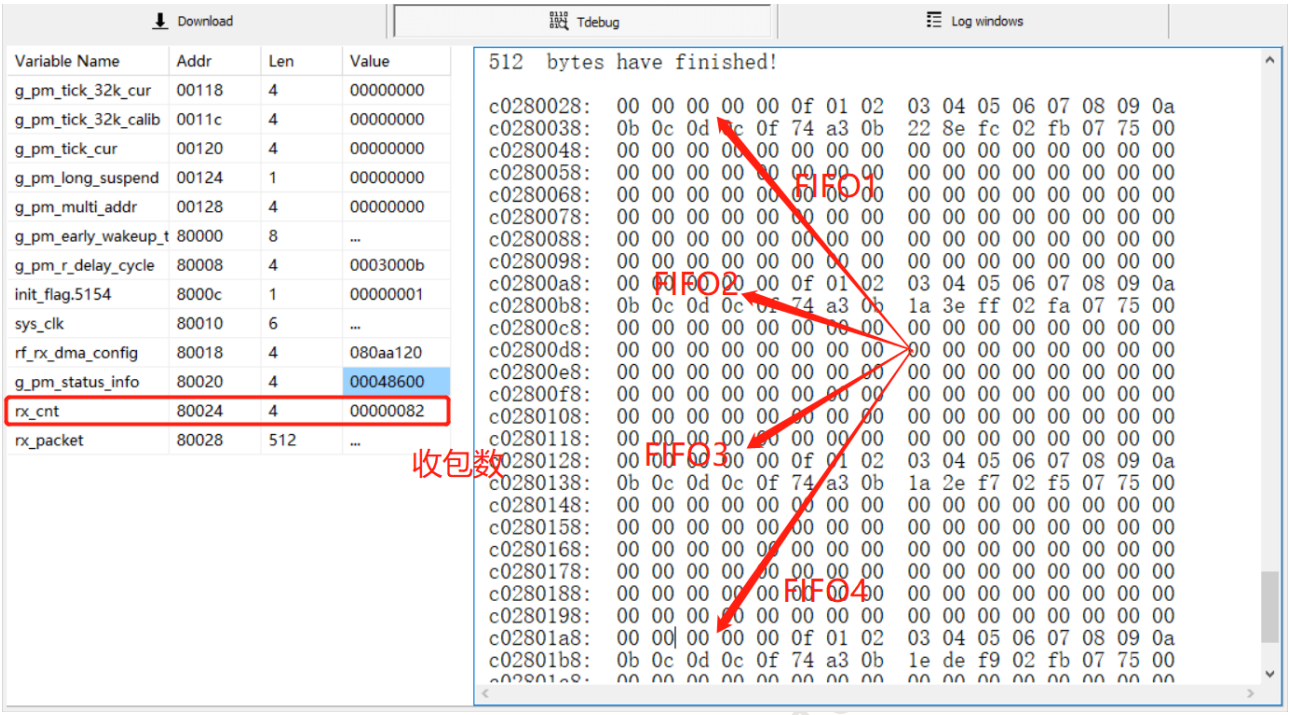


Figure 21.10: auto 模式下收包结果

21.8.2.2 跳频接收

若在收包过程中需要切换频点，设置步骤如下：

```
// 首先完成初始化动作
rf_set_ble_chn(17);// 2440MHz
rf_start_srx(clock_time());//触发收包，收包完成后状态机自动回到 IDLE 状态
if(rf_get_irq_status(FLD_RF_IRQ_RX))//判断收包结束
{
    u8* raw_pkt = rf_get_rx_packet_addr(RX_FIFO_NUM,RX_FIFO_DEP,rx_packet);//查找收包地址
    if(rf_ble_packet_crc_ok(raw_pkt))
    {
            rx_cnt++;
    }
    rf_clr_irq_status(FLD_RF_IRQ_RX);
}
while(rf_receiving_flag());//判断发包是否完成
rf_clr_irq_status(FLD_RF_IRQ_RX);//清除收包中断状态
rf_set_ble_chn(37);// 2402MHz 切换频点
rf_start_srx(clock_time());//触发再次收包
```

21.8.2.3 自动模式切换

如若在 auto 模式下进行收发切换，代码示例：

```
// 首先完成初始化动作
rf_set_ble_chn(17);// 2440MHz
rf_set_stx(ble_tx_packet,clock_time());
while(! (rf_get_irq_status(FLD_RF_IRQ_TX))); //等待发包结束
rf_clr_irq_status(FLD_RF_IRQ_TX); //清除中断状态
// 在进行状态切换时需等待前面一个状态结束才可进行状态切换
rf_set_tx_rx_off_auto_mode();//关闭 tx，rx
rf_start_srx(clock_time());//触发收包
```

注意：

- 目前在进行状态切换时通常等到前一个状态结束后再停状态机，然后切换下一个状态。

22 ISO-7816

22.1 ISO-7816 协议简介

ISO-7816 即国际智能卡通信标准，规定了接触式智能卡的相关规范，包括物理特性、接口规范、传输协议、命令交换格式等。

Telink SoC 集成了 ISO-7816 通信模块，支持与接触式 IC 卡的通信，本文简单介绍了建立 Telink SoC 与接触式 IC 卡通信的方法。

22.2 ISO-7816 使用方法

22.2.1 硬件连接

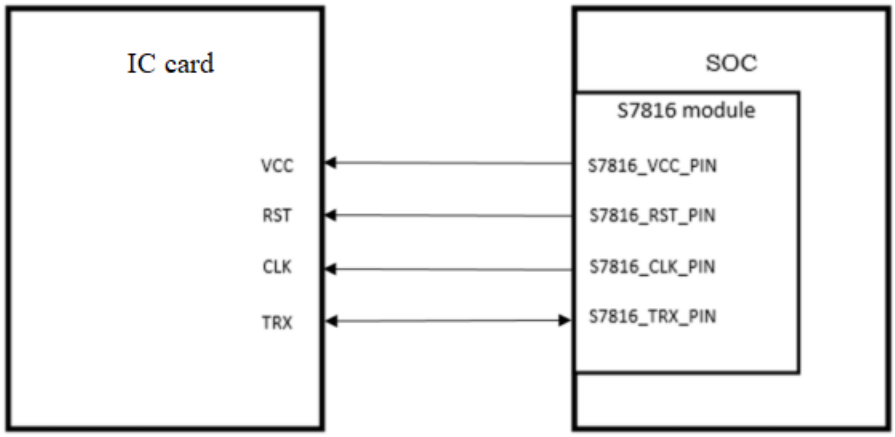


Figure 22.1: 硬件连接

在实际使用时，我们需要将 IC 卡的各个触点一一与 SoC 相连，这是 SoC 与 IC 卡通信的硬件基础。

(1) VCC 是 IC 卡的电源电压，RST 是 IC 卡的复位信号。我们可以在 SoC 的空余 GPIO 引脚中任意选择两个与之相连。

(2) CLK 是 IC 卡的时钟触点。由 SoC 供给时钟给 IC 卡。

(3) TRX 即 I/O 触点是 IC 卡输入输出触点，由于 IC 卡只支持半双工通信，在某一刻 I/O 触点只支持输入或输出，所以在实际使用时需要注意时序。

注意：

- ISO7816-3 协议规定了 IC 卡 3 种工作电压：A 类-5V，B 类-3V，C 类-1.8V，在实际使用时需要 SoC 提供的电压和卡片的工作电压相匹配。

22.2.2 初始化

```
s7816_set_pin(gpio_pin_e rst_pin,gpio_pin_e vcc_pin,gpio_pin_e clk_pin,gpio_pin_e trx_pin)
s7816_init(uart_num_e uart_num,s7816_clock_e clock,int f,int d)
```

s7816_set_pin() 用来配置 RST,VCC,CLK,TRX 引脚。

S7816_init() 用来选择 UART 通道（UART0 和 UART1），配置 IC 卡时钟，以及 IC 卡时钟频率调节因子 F（默认为 372）和比特率调节因子 D（默认为 1）。

```
s7816_en(uart_num_e chn)
```

配置完成后需要使用函数 s7816_en() 来使能 7816 模块。

注意：

- S7816 是通过 SoC 的 UART 功能来实现的，所以使用时会占用对应的 UART 模块。

22.2.3 IC 卡激活与冷复位

```
s7816_coldreset();
```

ISO7816 协议规定，复位引脚拉起之后，冷复位应答将在之后的 400-40000 个时钟周期之内开始。

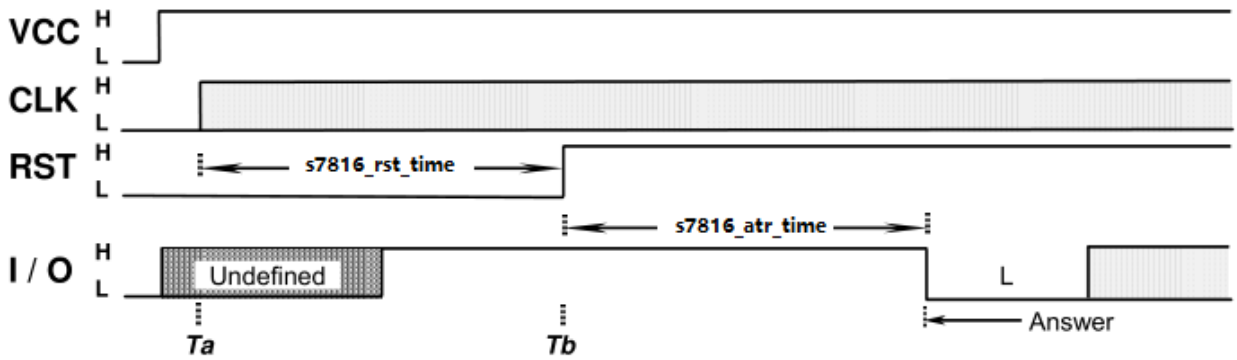


Figure 22.2: 冷复位

以 SoC 时钟配置 4MHZ 为例，冷复位过程：

a) VCC 拉起之后，从 Ta 时刻 SoC 输出 CLK，SoC 配置 TRX 引脚，SoC 的 TRX 引脚配置之后，置 SoC 的 TRX 引脚为接收状态。

b) 在 Ta 后的 40000 周期内（10000us），RST 拉高。

c) RST 拉高之后的 40000 个周期内 (10000us)，IC 卡会传回复位字符给 SoC。

```
s7816_set_time(int rst_time_us)
```

s7816_set_time() 用来对冷复位中的 s7816_rst_time 时间进行重设。

(1) rst_time_us 对应 s7816_rst_time，即时序中 Ta 到 Tb 的复位等待时间，默认为 40000 个时钟周期。

(2) s7816_atr_time，即时序中 Tb 之后的等待 ATR 返回时间，范围在 400-40000 个时钟周期不定。在实际使用时需要等 ATR 字符接收完成之后再进行下一步操作 (可以在主函数进行协议解析来判断 ATR 字符是否接收完成)。

22.24 热复位

```
s7816_warmreset()
```

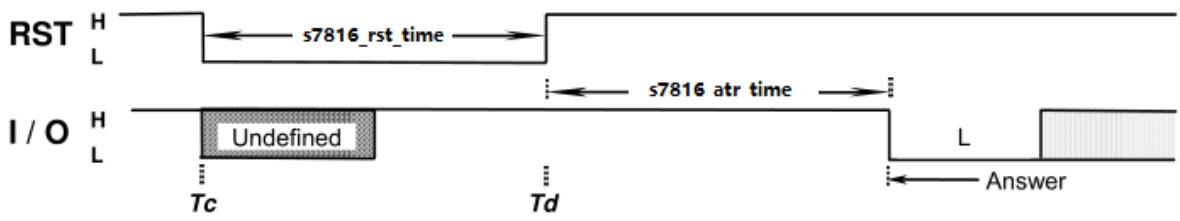


Figure 22.3: 热复位

IC 卡对终端的复位应答有着规定的规格和内容，如果中断收到的复位应答不符合规定要求时，终端将启动一个热复位并从 IC 卡获得复位信号。

以时钟配置 4MHZ 为例，热复位过程:

a) CLK 和 VCC 始终保持正常状态 (CLK 施加，VCC 拉起)。

b) 在 Tc 时刻，将 RST 从高置低。

c) 在 Tc 时刻的 200 个周期内 (50us)，SoC 置 TRX 引脚为接收状态。

d) 在 Tc 后 40000 个周期内 (10000-10000us)，将 RST 置高电平。IC 卡的复位应答将在 Td 后 40000 个周期内开始 (10000us)。

和冷复位相同，热复位同样可以使用 s7816_set_time() 函数重设 s7816_rst_time (默认值 40000 个时钟周期)。需要等待 ATR 字符接收完成才可以进行下一步操作。

注意：

- 冷复位和热复位的复位应答 (初始 ATR) 是相同的。两者的差别在于：冷复位伴随着 IC 卡的激活，热复位是在 IC 卡激活后进行的。

22.2.5 触点释放

```
s7816_release_trig()
```

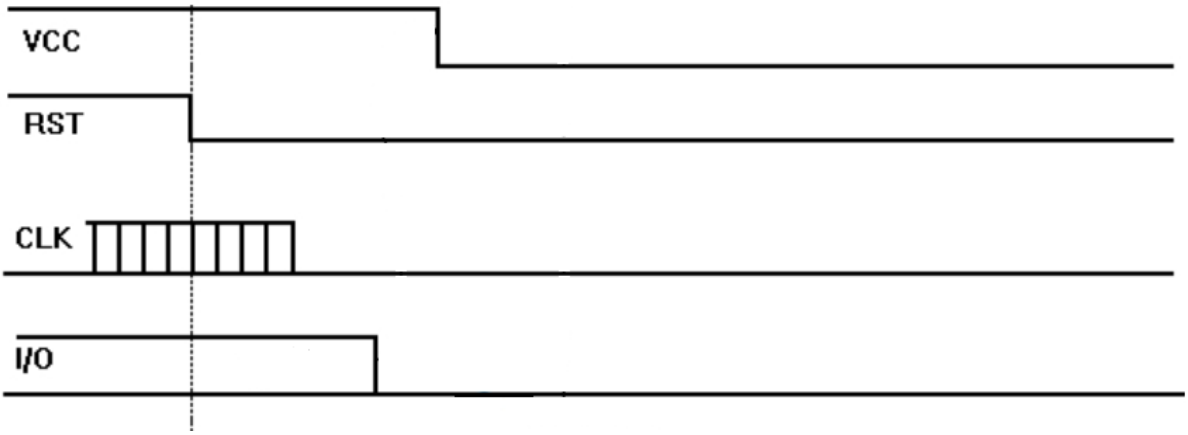


Figure 22.4: 触点释放

冷复位或者热复位时，如果 IC 卡没有在规定时间内进行复位应答，则终端需要启动一个触点释放时序。

(1) 终端以置 RST 为低电平状态开始触点释放序列。

(2) 在置 RST 为低电平之后且 VCC 断电之前，终端将 CLK 和 I/O 也置低电平。

(3) 最后，在实际断开触点之前，终端将 VCC 去电。

22.3 Demo 简介

以经过初始化的 SMARTCOS-PSAM 卡为例。Demo 先进行冷复位，再取随机数。

取随机数指令为：0x00,0x84,0x00,0x00,0x04（取 4 字节的随机数）

冷复位获得初始 ATR：

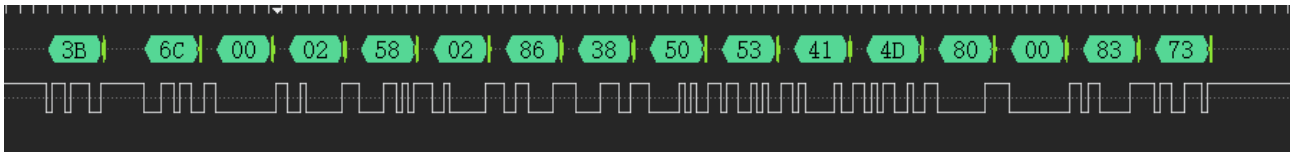


Figure 22.5: 冷复位获得初始 ATR

初始 ATR 共 16 个 byte。初始 ATR 分析如下：

(1) 3B：正向约定。

(2) 6C：即 T0，6 二进制为 0110，表示 TB1 和 TC1 存在，C 表示历史字符为 12 个。

(3) 由 T0 得知，TB1 为 00，表示无需额外的编程电压。

(4) 由 T0 得知，TC1 为 02，需要额外两个保护时间，即由终端向 IC 卡发送数据时，每两个 byte 之间需要额外增加两个 etu 的时间。

(5) 由 T0 得知，12 个历史字符为 0x58,0x02,0x86,0x38,0x50,0x53,0x41,0x4d,0x80,0x00,0x83,0x73.

(6) IC 卡使用 T=0 协议，没有 TCK 校验字符。

取随机数:

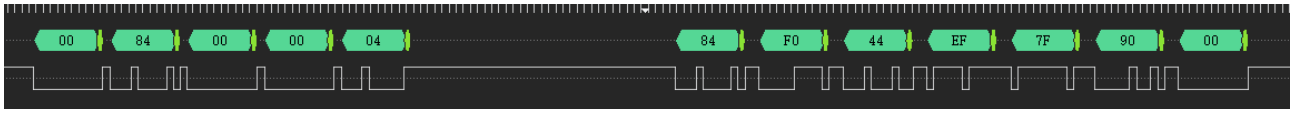


Figure 22.6: 取随机数

其中，0x00,0x84,0x00,0x00,0x04 为终端向 IC 卡发送的取随机数指令。

取得的随机数为 0xf0,0x44,0xef,0x7f 共 4 个 byte，每次取的随机数都会不同。

数据尾 9000 表示指令执行成功。

23 ADC

23.1 简介

ADC 驱动可用于 ADC 采样外部 GPIO 电压、电池电压和温度传感器。

23.2 工作原理

23.2.1 内部结构

SAR_ADC 的内部结构如下图所示：

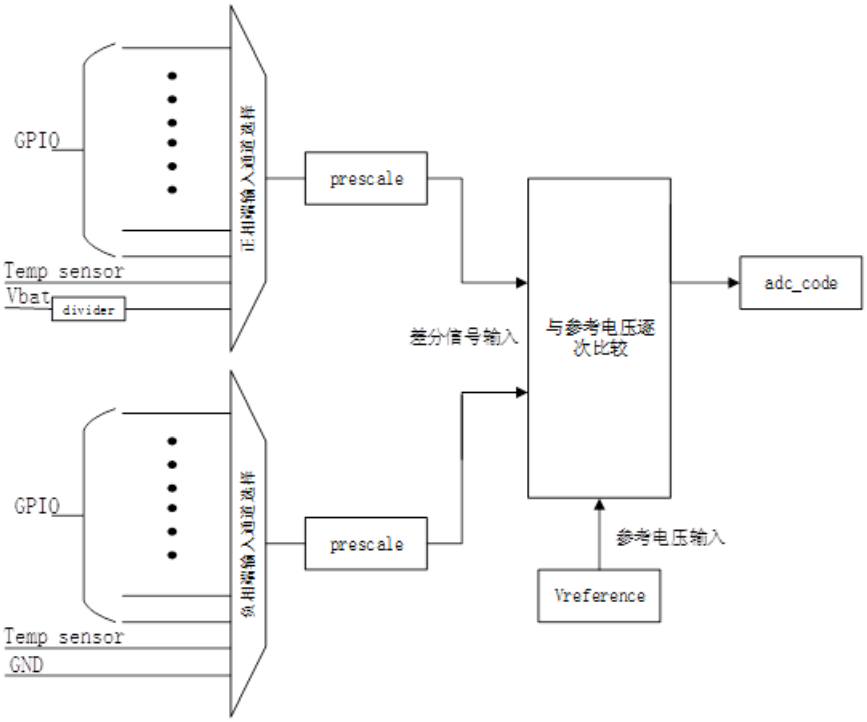


Figure 23.1: ADC 的内部结构

SAR ADC 只支持差分模式，通过差分采样获取 code 值，然后由驱动换算成电压值或温度值输出。

应用场景	P 端	N 端
单个 GPIO 采样	ADC_GPIO	GND
两个 GPIO 差分采样	ADC_GPIO1	ADC_GPIO2
Vbat channel	VBAT	GND
Temp sensor	ADC_TEMSENSORP_EE	ADC_TEMSENSORN_EE

P 端和 N 端电压需满足的条件：

单个 GPIO 采样的范围，可以参照下面的（1）和（2）来确定最终的采样范围。

两个 GPIO 差分采样的适用范围，需要满足以下四个条件：

（1）P 端和 N 端的电压均不允许超过 prescal*Vreference；

（2）P 端和 N 端的电压均不允许超过 V_ioh（IO 输出高电平时的电压值）；

（3）P 端和 N 端的差值电压不允许超过 prescal*Vreference；

（4）((Vp+Vn)/2)<(prescal*Vreference)。

芯片	V_ioh(IO 输出高电平时的电压值)
B85	V_ioh=vbat<3.6v
B87	V_ioh=vbat<3.6v
B91	（1）当应用场景中 vbat 电压一定低于 3.6v 时，设置 VBAT_MAX_VALUE_LESS_THAN_3V6 模式，V_ioh=vbat；（2）当应用场景中 vbat 电压可能高于 3.6v 时，设置 VBAT_MAX_VALUE_GREATER_THAN_3V6 模式，且 vbatV_ldo 时，V_ioh=V_ldo。(V_ldo=3.3v(+-10%))

23.2.2 采样电压值计算

模拟输入电压 (V_{IN}) 与参考电压 (V_{REF}) 比较，按比例生成 N 位采样 code 值，存储在寄存器中。实际应用时一般会对 V_{IN} 进行预分压来支持更大的采样范围，以 14 位分辨率采样 code 值为例，当预分压系数 pre_scale = 1/4 时，V_{IN} 与 code 值的换算公式为：

$$\frac{\frac{1}{4} * V_{IN}}{V_{REF}} = \frac{adc\_code}{ref\_code}$$

其中：adc_code 为采样 V_{IN} 得到的 code 值。

ref_code 为 V_{REF} 换算得到的 code 值，14 位分辨率对应 0x1fff(bit13 为符号位)。

以此反推可以得到 V_{IN} 的采样电压值。

注意：

- 对于 GPIO，经过 prescale 分压后就可以作为差分信号与参考电压进行比较。但是 Vbat 需要经过 Vbat divider 和 prescale 两次分压才可以进行比较，最终的分压系数为两次分压系数的乘积。例预分压系数 pre_scale = 1 且 Vbat_divider = 1/3 时，V_{IN} 与 code 值的换算公式为：

$$\frac{\frac{1}{3} * 1 * V_{IN}}{V_{REF}} = \frac{adc\_code}{ref\_code}$$

23.3 B91 ADC 使用说明

23.3.1 接口说明

接口命名规则：

- init 后缀：初始化用到的接口。
- dma 后缀：dma 模式采样会用到的接口。
- 无 dma 后缀：手动采样会用到的接口。

234 Demo 说明

234.1 Demo 结构说明

ADC Demo 的应用.c 文件分别为 app.c，ADC_Demo/app_config.h 中的宏 ADC_MODE 选择使用哪一种采样模式。

```
#define ADC_DMA_MODE                1
#define ADC_NDMA_MODE               2

#define ADC_MODE              ADC_NDMA_MODE
```

在 ADC_NDMA_MODE（手动采样模式）和 ADC_DMA_MODE（DMA 采样模式）中，通过配置宏 ADC_SAMPLE_MODE 选择 ADC 的使用场景为 GPIO 模拟信号输入、电池电压（Vbat）和温度传感器中的一种。

```
#define ADC_GPIO_SAMPLE          1 //GPIO voltage
#define ADC_VBAT_SAMPLE          2 //Vbat channel Battery Voltage
#define ADC_TEMP_SENSOR_SAMPLE          3 //Temp test

#define ADC_SAMPLE_MODE             ADC_GPIO_SAMPLE
```

234.2 ADC 的初始化配置

ADC 的初始化流程如下图所示：

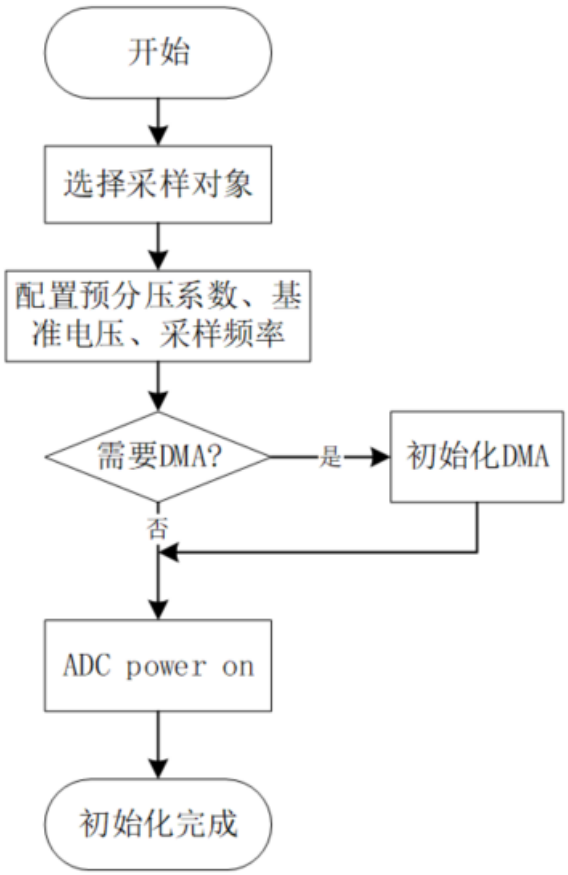


Figure 23.2: ADC 的初始化流程

234.3 ADC 的采样和转换过程

ADC 的采样和转换如下图所示：

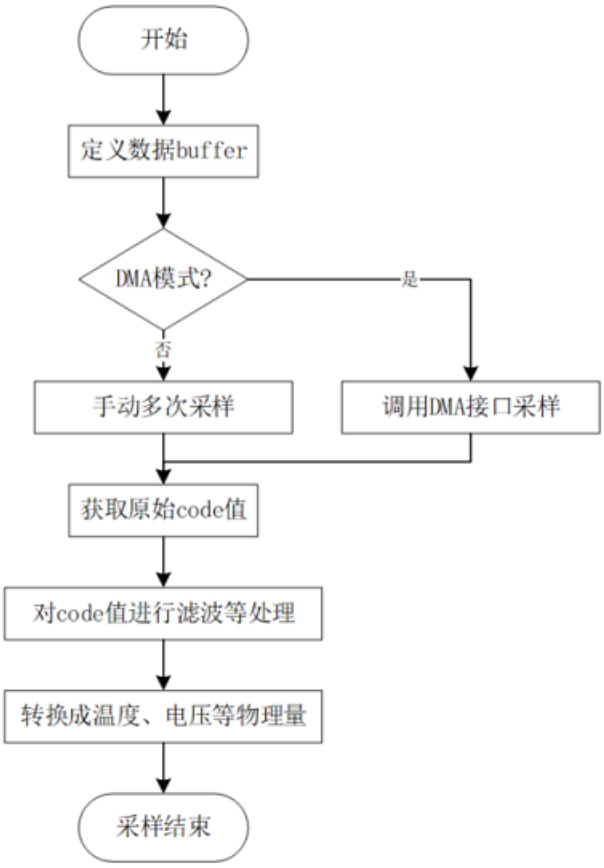


Figure 23.3: ADC 的采样和转换

注意：

- 手动（ADC_NDMA_MODE）采样时，一次只能获取一个 adc_code，期间会关闭 ADC 采样功能，且在使用时一定要保证连续获取 adc_code 的时间间隔大于 2 个采样周期。

2344 Demo **测试实例**

配置 DMA 方式采样 Vbat。

通过 BDT 工具查看采样结果如下：

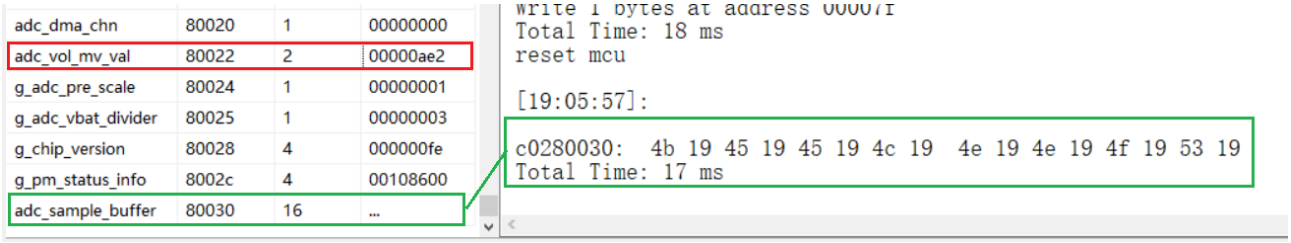


Figure 23.4: 采样结果

图中：adc_sample_buffer 存储的为 8 组采样的 code 值，adc_vol_mv_val 代表采样电压值，0xae2 换算成 10 进制为 2786mV（电压表测得 2790mV）。

23.5 芯片差异

23.5.1 功能支持差异

芯片	GPIO 采样	Vbat 采样方式	是否支持 Temp 采样
B85	PB0-7/ PC4-5	不支持 vbat channel 模式，是利用将 gpio 输出高进行 gpio 采样的方式进行 vbat 采样的，此时 GPIO 的电压就是 Vbat 的电压（该方法是不需要硬件连线的，可以设置一个没有封装出的 pin, 以节省 gpio 资源）	不支持
B87	PB0-7/ PC4-5	vbat channel	支持
B91	PB0-7/ PD0-1	硬件电路上使用外部分压，软件使用 GPIO 方式进行采样	支持

23.5.2 校准配置说明

下面列出有芯片级校准的配置，推荐使用这些配置以减少误差（如果使用其他配置，校准值就不准了），如果想要误差更小，可以在产线上用夹具进行板级校准。

芯片	出厂带的校准值	采样误差 (测试数据量较少，仅供参考)
B85	GPIO 采样，采样率 96KHz，预分压系数 1/8，参考电压 1.2V。	误差在-14~12mV。（29 颗样品芯片）
B87	GPIO 采样，采样率 96KHz，预分压系数 1/8，参考电压 1.2V。	误差在 9~12mV。（20 颗样品芯片）
B87	vbat channel 采样，采样率 96KHz，预分压系数 1，vbat 分压系数 1/3，参考电压 1.2V。	误差在 10mV 以内。（10 颗样品芯片）
B91	GPIO 采样，采样率 48KHz，预分压系数 1/4，参考电压 1.2V。	误差在-11~7mV。（19 颗样品芯片）

信噪比和误差概念的区分

B91 的 datasheet 注明信噪比为 10.5bit，含义如下：

信噪比（有效位）为 10.5 位，对应模拟量 1200mV/（2^10.5）≈0.82mV（假设参考电压选择 1200mV），意思是 ADC 采样精度为 0.82mV，即 1 单位的 code 值表示 0.82mV 电压值。

误差的概念，比如误差是 10mV，输入 500mV，采样结果是 510mV。

外部分压电路

当需要采样的电压超过 ADC 采样范围时，必须使用外部分压电路将原电压分压至采样范围，再输入到采样点，并通过 GPIO 采样。建议的外部分压电路配置如下（可以根据各自的应用需求去选择）：

硬件电路参考：

由于芯片内阻就有几十 M，所以分压电路的总阻值不能过大（最好不要超过 2M），否则电流过小，ADC 无法正常采样。

M 级分压电路：漏电小，采样慢

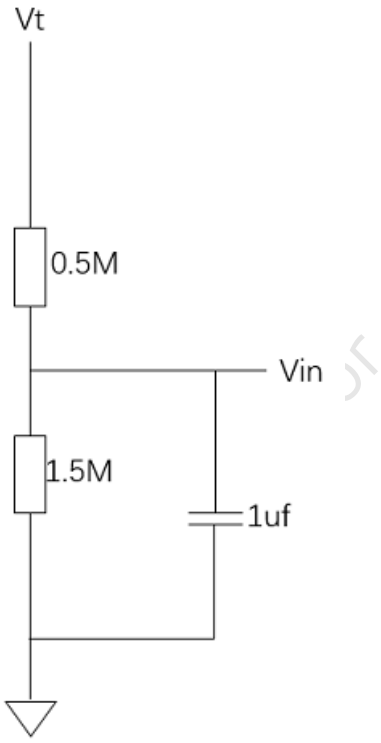


Figure 23.5: M 级分压电路

百 K 级分压电路：漏电大，采样快

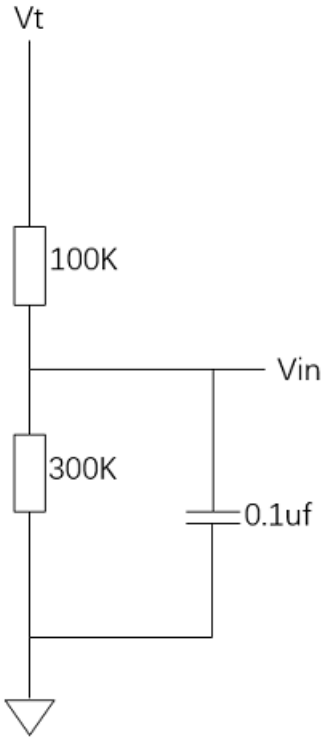


Figure 23.6: 百 K 级分压电路

软件使用要求：

sar adc 的输入等效是一个 15pF 左右的电容，在采样时会有一个等效 $f*c*v$ 的动态电流，所以电阻分压的采样点的电压值会慢慢的产生一个误差电压。所以

(1) 打开 ADC 立刻采样时，采样点的值还是处在比较准确的地方，所以采样值较准确。

(2) 关闭 ADC 后延时，由于没有动态电流，分压点电阻值会重新回到准确的值。

(3) 采样率越高，误差电压越大。

针对不同工作模式下，有不同的解决方法。以 B91 为例说明。

(下面的标注出的延时部分，会和采样频率以及电压电流等相关，所以使用时仅需要参考下面的逻辑处理方式，但是相关延时部分，请根据实际应用进行测试，并预留一定的余量出来。)

硬件配置：B91 校准芯片 + M 级分压电路

软件配置 (ADC_Demo)：

- 1.2V Vref 基准电压
- 1 / 4 的 pre_scale 预分压系数
- 采样频率 23K

(1) 正常工作模式时：

a. 打开ADC Power立刻采样，然后立刻关闭ADC Power
b. 延时（大于50ms，小于这个值误差会偏大些）
c. 进行下一次采样，采样步骤跟a和b一样

注意：

- 如果需要切换 PIN，切换 PIN 的操作加在步骤 a 前即可。

（2）deep 或者 deep retention 模式时，采样后会进入 deep 或者 deep retention，不需要额外操作。

（3）suppend 模式时，采样后会进入 suppend，配置 suspend 的时间大于 200ms。

芯片级校准误差为-9 ~ 5mV；板级校准误差为-3 ~ 8mV。(10 颗样本芯片)

硬件配置：B91 校准芯片 + 百 K 级分压电路

软件配置 (ADC_Demo)：

- 1.2V Vref 基准电压
- 1 / 4 的 pre_scale 预分压系数
- 采样频率 23K

(1) 正常工作模式时：

a. 打开ADC Power立刻采样，然后立刻关闭ADC Power
b. 延时（大于5ms，小于这个值误差会偏大些）
c. 进行下一次采样，采样步骤跟a和b一样

注意：

- 如果需要切换 PIN，切换 PIN 的操作加在步骤 a 前即可。

（2）deep 或者 deep retention 模式时，采样后会进入 deep 或者 deep retention，不需要额外操作。

（3）suppend 模式时，采样后会进入 suppend，配置 suspend 的时间大于 50ms。

芯片级校准误差为-5 ~ 7mV；板级校准误差为-4 ~ 6mV。(10 颗样本芯片)

24 USB 简介

USB (Universal Serial Bus) 即通用串行总线，是一个外部总线标准，用于规范电脑与外部设备的连接和通讯，是应用在 PC 领域的接口技术。USB 接口支持设备的即插即用和热插拔功能。USB 是在 1994 年底由英特尔、康柏、IBM、Microsoft 等多家公司联合提出的。如下图所示，USB 由四根线组成，即 VCC、GND、D-(DM)、和 D+(DP)，USB 可选择通过主机供电也可自供电，目前大部分 USB 设备是通过主机供电。

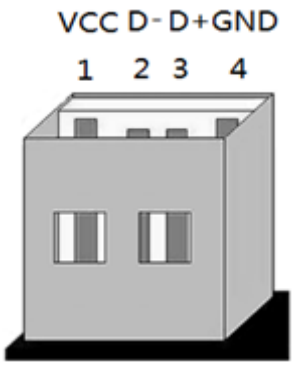


Figure 24.1: USB 接口

USB 通信是指控制器与设备间的通信，电脑主机即为控制器，Telink USB 为设备，后文中引用的主机默认为控制器。USB 总线是一种单向总线，通信只能由控制器发起，设备收到控制器请求时，将数据发给控制器。控制器是每隔 n 个单位时间向设备发送请求，n 为用户配置参数。

USB 有超高速 (5.0Gbit/s)、高速 (480Mbit/s)、全速 (12Mbit/s) 和低速 (1.5Mbit/s) 等四种工作速度，其中全速和低速 USB 总线的通信帧周期（连续两帧数据的发送间隔）为 1ms，高速 USB 总线的通信帧周期为 125us。USB1.1 仅支持全速和低速，USB2.0 支持高速、全速和低速，超高速仅在 USB3.0 中有支持。

24.1 USB 包格式和传输过程

包 (Packet) 是 USB 数据传输的最基本单元，即每次数据传输都是以包的形式进行。而包要组成事务才能进行有效通信。根据通信的需求有各种包，用于组成各种事务 (IN、OUT、SETUP)。一个或多个事务组成一个传输 (控制传输、批量传输、终端传输和等时传输)。

包是 USB 总线上数据传输的最小单位，不能被打断或干扰，否则会引发错误。若干个数据包组成一次事务，一次事务也不能打断，即属于一次事务的几个包。

24.1.1 USB 包结构

包 (Packet) 是 USB 系统中信息传输的基本单元，所有数据都是经过打包后在总线上传输的。

如下图所示，USB 包由七部分组成，即同步域 (SYNC)、包标识 (PID)、地址域 (ADDR)、端点域 (ENDP)、帧号域 (FRAM)、数据域 (DATA) 和校验域 (CRC)。注意，并不是每个 USB 包都包含上述的七个域，也就是说有些包只包含其中的几个域。

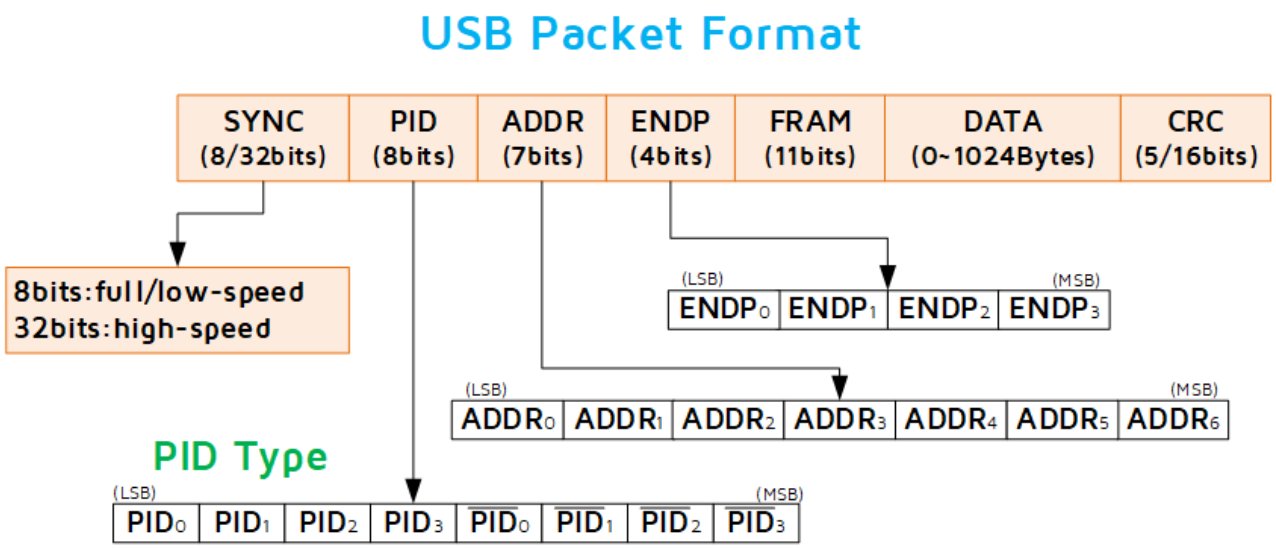


Figure 24.2: USB 包通用格式

(1) 同步域

同步域主要是通知对方数据传输开始，并提供同步时钟。对于低速设备和全速设备，同步域使用的是 0000 0001（二进制数）；对于高速设备使用的是 00000000 00000000 00000000 00000001。

(2) 包标识 (PID)

包标识主要用于标识包的类型，由 8 位组成：低 4 位是 PID 编码，高 4 位是校验字段，是对低 4 位取反得到，USB 中各种包是通过 PID 字段来区分。

(3) 地址域

由于接入 USB 总线的设备可能有多个，因此需要引入地址域，以便于区分当前通信的设备是哪个设备。地址域包含 7 个数据位，最多可以指定 128 个地址，地址 0 用作缺省地址，不分配给 USB 设备。对于 USB 总线上的每个设备，地址唯一。

(4) 端点域

端点域用于指定 USB 总线上某个设备的一个端点号，包含 4 个数据位；全速/高速设备最多可以含有 16 个端点，低速设备最多含有 3 个端点。所有 USB 设备都必须含有一个端点号为 0 的端点，用于主机与设备间交换基本信息。除端点 0 外，其余的端点都是具体 USB 设备所特有的。地址域和端点域组合，明确了主机与设备间通信的通道。

(5) 帧号域

帧号字段用于指出当前帧的帧号，它仅在每帧/微帧开始的 SOF 令牌包中被发送，其数据位长度为 11 位，每传输一帧，主机就将其加 1，当达到最大值 7FFH 时归零。

(6) 数据域

数据字段包含主机和 USB 设备间需要传输的数据，以字节为单位，最大长度为 1024，而实际长度取决于传输的具体情况。

(7) 校验域

校验域主要是为了校验通信数据的正确性。USB 令牌包和数据包中都使用了 CRC。但是，CRC 是发送方在进行位填充之前产生的，这样要求接收方在去除位填充之后，再对 CRC 字段进行译码。信息包中的 PID 字段本身含有校验，所以 CRC 计算不包含有 PID 部分。令牌包的 CRC 采用的是 5 位 CRC，数据包中的数据字段使用的是 16 位 CRC。

24.1.1.1 令牌包

SOF 包由主机发送给设备：对于 full-speed 总线，每隔 1.00 ms±0.0005 ms 发送一次；对于 high-speed 总线，每隔 125us±0.0625us 发送一次。

SOF 包格式

	(LSB)			(MSB)
Field	SYNC	PID	FRAM	CRC5
Bits	8/32	8	11	5

Figure 24.3: SOF 包格式

IN，OUT，SETUP 包格式

	(LSB)				(MSB)
Field	SYNC	PID	ADDR	ENDP	CRC5
Bits	8/32	8	7	4	5

Figure 24.4: IN OUT SETUP 包格式

24.1.1.2 数据包

数据包 (DATA0, DATA1, DATA2, MDATA)

	(LSB)			(MSB)
Field	SYNC	PID	DATA	CRC16
Bits	8/32	8	0~8192	16

Figure 24.5: 数据包

24.1.1.3 握手包

PRE, ACK, NAK, STALL, NYET 包格式

	(LSB)	(MSB)
Field	SYNC	PID
Bits	8/32	8

Figure 24.6: PRE ACK NAK STALL NYET 包格式

24.1.2 USB 传输过程

24.1.2.1 USB 事务

在 USB 的一次数据接收或发送的处理过程称为事务处理 (Transaction)，事务通常是由一系列包组成，事务不同，组成的包也不同。USB 数据传输中，常见的事务有输入 (IN) 事务、输出 (OUT) 事务和设置 (SETUP) 事务。注意，SOF 只是指示一帧的开始，无有效数据，并不是一次事务；EOF 帧发送结束后的一种电平状态，也不是事务。

事务通常由两个或者三个包组成：令牌包、数据包和握手包，令牌包启动事务，数据包传输数据，握手包的发送者通常为数据接收者，当数据正确接收后，发送握手包，设备也可以使用 NACK 表示数据未准备好。

24.1.2.2 输入事务

输入 (IN) 事务是主机从 USB 设备的某个端点中获取数据的过程，如下图所示，一个输入事务有三种状态，即正常的输入事务（图 (a)）、设备忙或无数据时的输入事务（图 (b)）和设备出错时的输入事务（图 (c)），正确的输入事务包括令牌包、数据包和握手包三个阶段。

输入事务处理流程

1. Token (Host->Device)	SYNC	IN	ADDR	ENDP	CRC5
2. Data (Device->Host)	SYNC	DATA0/DATA1			CRC16
3. Handshake (Host->Device)	SYNC	ACK			

Figure 24.7: (a) 正常的输入事务

1. Token (Host->Device)	SYNC	IN	ADDR	ENDP	CRC5
2. Handshake (Device->Host)	SYNC	NAK			

Figure 24.8: (b) 设备忙或无数据时输入事务

1. Token (Host->Device)	SYNC	IN	ADDR	ENDP	CRC5
2. Handshake (Device->Host)	SYNC	STALL			

Figure 24.9: (c) 设备出错时的输入事务

结合正常的输入事务实例，对正常的输入事务进行介绍和分析，如下图所示，一个正常的输入事务包含三个交互流程：(1) Host 向 Device 发送一个 IN 令牌包；(2) Device 收到 IN 令牌包后，将待发送数据发给主机；(3) 主机收到数据包后，回复一个 ACK 包来确认数据包被正确接收。

IN txn [7 POLL]	12 01 10 01 00 00 00 08
[7 IN-NAK]	
IN packet	69 00 10
DATA1 packet	4B 12 01 10 01 00 00 00 08 11 77
ACK packet	D2

Figure 24.10: 正常输入事务实例

24.1.2.3 输出事务

输出 (OUT) 事务是主机向 USB 设备的某个端点中发送数据的过程，如下图所示，一个输出事务有三种状态，即正常的输出事务（图 (a)）、设备忙时的输出事务（图 (b)）和设备出错时的输出事务（图 (c)）。正确的输出事务包括令牌、数据和握手三个阶段。

输出事务处理流程

1. Token (Host->Device)	SYNC	OUT	ADDR	ENDP	CRC5
2. Data (Host->Device)	SYNC	DATA0/DATA1			CRC16
3. Handshake (Device->Host)	SYNC	ACK			

Figure 24.11: (a) 正常的输出事务

1. Token (Host->Device)	SYNC	OUT	ADDR	ENDP	CRC5
2. Data (Host->Device)	SYNC	DATA0/DATA1			CRC16
3. Handshake (Device->Host)	SYNC	NAK			

Figure 24.12: (b) 设备忙时的输出事务

1. Token (Host->Device)	SYNC	OUT	ADDR	ENDP	CRC5
2. Data (Host->Device)	SYNC	DATA0/DATA1			CRC16
3. Handshake (Device->Host)	SYNC	STALL			

Figure 24.13: (c) 设备出错时的输出事务

下面结合正常的输出事务实例，对正常的输出事务进行介绍和分析，如下图所示，一个正常的输出事务包含三个交互流程：（1）Host 向 Device 发送一个 OUT 令牌包；（2）Host 向 Device 发送数据包；（3）Device 收到数据包后，回复一个 ACK 包来确认数据包被正确接收。

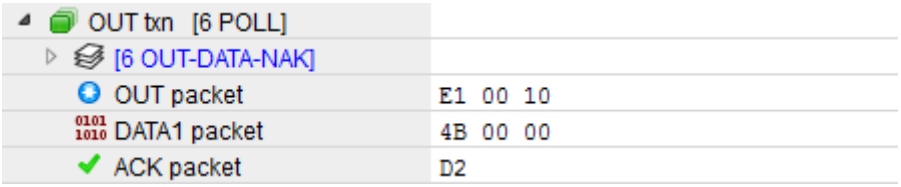


Figure 24.14: 正常输出事务实例

24.1.24 设置事务

设置 (SETUP) 事务处理并定义了 Host 与 Device 之间的特殊的数据传输，它仅适用于 USB 控制传输的建立阶段。如下图所示，设置事务通常有三种状态，即正常设置事务（图 (a)）、设备忙设置事务（图 (b)）和设备出错设置事务（图 (c)），正确的设置事务包括令牌、数据和握手三个阶段。

设置事务处理流程

1. Token (Host->Device)	SYNC	SETUP	ADDR	ENDP	CRC5
2. Data (Host->Device)	SYNC	DATA0			CRC16
3. Handshake (Device->Host)	SYNC	ACK			

Figure 24.15: (a) 正常设置事务

1. Token (Host->Device)	SYNC	SETUP	ADDR	ENDP	CRC5
2. Data (Host->Device)	SYNC	DATA0			CRC16
3. Handshake (Device->Host)	SYNC	NAK			

Figure 24.16: (b) 设备忙时的设置事务

1. Token (Host->Device)	SYNC	SETUP	ADDR	ENDP	CRC5
2. Data (Host->Device)	SYNC	DATA0			CRC16
3. Handshake (Device->Host)	SYNC	STALL			

Figure 24.17: (c) 设备出错时的设置事务

下面结合正常的设置事务实例，对正常的设置事务进行介绍和分析，如下图所示，一个正常的设置事务包含三个交互流程：（1）Host 向 Device 发送一个 SETUP 令牌包；（2）Host 向 Device 发送 DATA0 数据包；（3）Device 收到数据包后，回复一个 ACK 包来确认数据包被正确接收。

SETUP txn	80 06 00 01 00 00 40 00
SETUP packet	2D 00 10
DATA0 packet	C3 80 06 00 01 00 00 40 00 DD 94
ACK packet	D2

Figure 24.18: 正常设置事务实例

24.1.3 USB 传输

传输由 OUT、IN 或 SETUP 等事务构成，USB 标准协议中定义了 4 种传输类型：控制传输 (Control Transfer)、批量传输 (Bulk Transfer)、中断传输 (Interrupt Transfer) 和同步传输 (Isochronous Transfer)。四种传输的优先级由高到底一次为：同步传输、中断传输、控制传输、批量传输。

24.1.3.1 控制传输

控制传输 (Control Transfer) 是 USB 中最基础、最重要的传输方式，也是端口 0 的默认传输方式。控制传输典型地用在主机和 USB 外设之间的端点 (Endpoint)0 之间的传输，但是指定供应商的控制传输可能用在其它的端点上控制传输主要用来主要进行查询，配置和给 USB 设备发送通用的命令。控制传输是单向传输（端点 0 除外，端点 0 为双向传输），数据量通常较小。控制传输的数据包最大长度取决于其工作速度，低速模式最大包长固定为 8 字节、高速模式最大包长度为 64 字节，全速模式可在 8、16、32 和 64 字节中选择。

注意：

- Telink USB 的端点 0 的包长度为固定长度 8 字节，无法配置其他值。
- 控制传输分为三个阶段，即建立阶段、数据阶段（可选）和状态阶段组成，每个阶段都由一次或多次（数据阶段）事务组成。
- 建立阶段：如图建立阶段由 SETUP 事务组成。SETLUP 事务的数据阶段总是用 DATA0，且定长 8 字节。

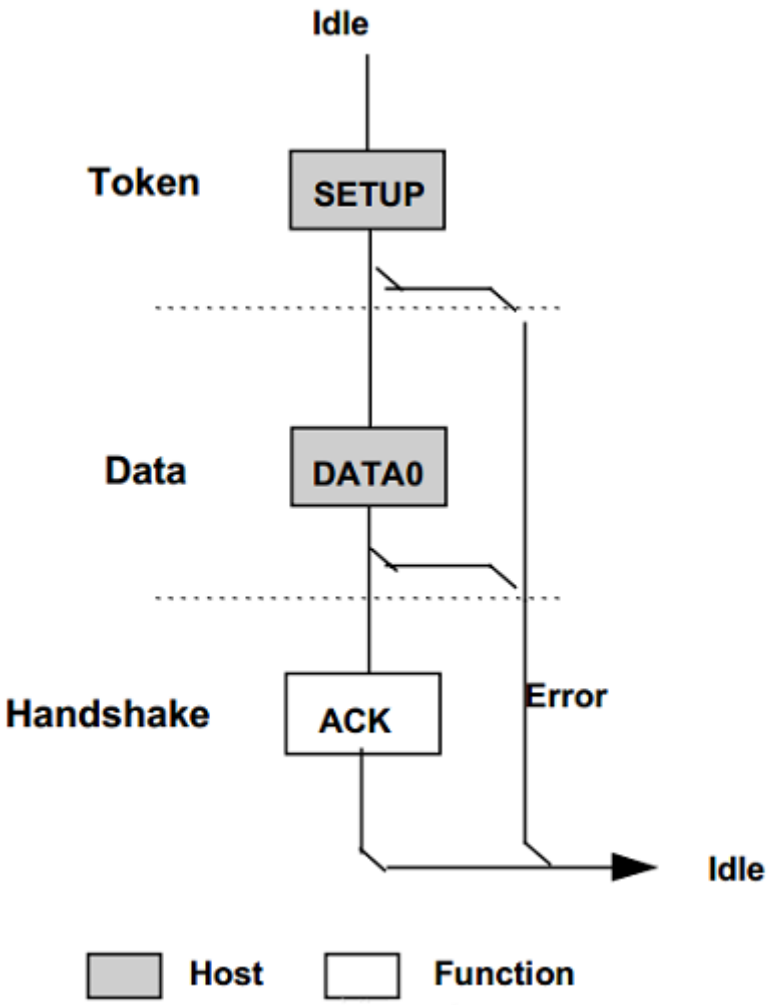


Figure 24.19: 建立事务输流程图

- 数据阶段：数据阶段可选。如果有数据阶段，则包括一个或多个 IN/OUT 事务。用于传输建立阶段要求的，具有 USB 定义格式的数据。数据阶段的事务具有相同的方向，即要么全是 IN，要么全是 OUT。如果要传输的数据大于一个包的长度，则主控器将其分成多个包传输，一旦数据传输方向改变，就会认为进入到状态过程，数据过程的第一个数据包必须是 DATA1 包，然后每次争取传输一个数据包后就在 DATA0 和 DATA1 之间交换。如果最后一个包大小等于最大包大小，则应再传一个 0 大小的包，以确定结束。根据数据阶段的数据传输的方向，控制传输又可分为 3 种类型，即控制写入 (Control Write)、控制读取 (Control Read) 和无数据控制 (No-data Control)，如下图所示

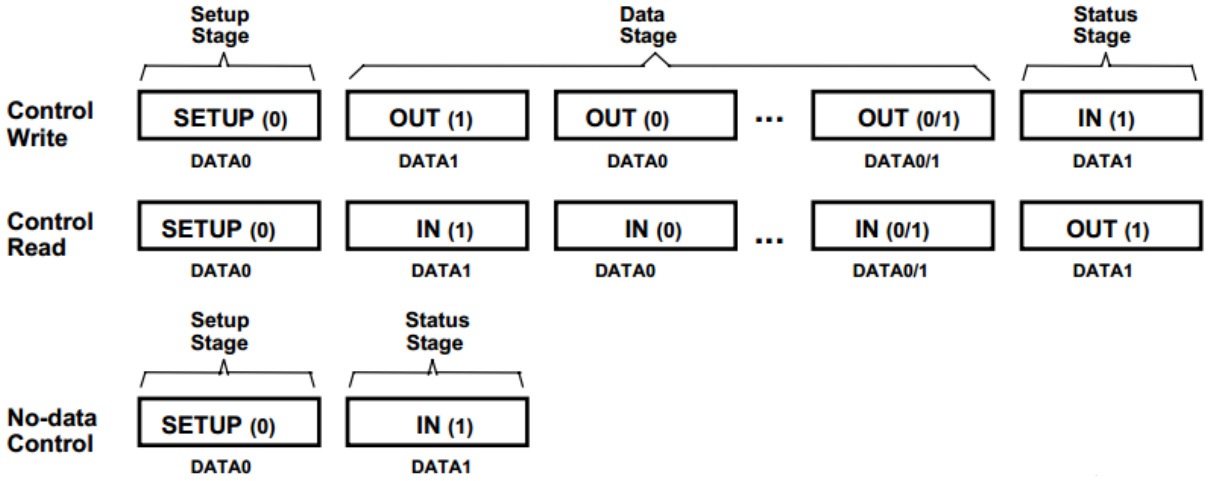


Figure 24.20: 控制传输序列图

- 状态阶段：状态阶段是控制事务处理的最后一个阶段，由一个 IN 或 OUT 事务组成，总是使用 DATA1 数据包组成。状态阶段与数据阶段传输的方向相反，即如果数据阶段是 IN，则状态阶段是 OUT，反之亦然。用于报告建立阶段和数据阶段的传输结果。

24.1.3.2 中断传输

中断传输 (Interrupt Transfer) 在流程上除不支持 PING 和 NYET 包之外，其他的和批量传输相同，因此其序列图可参考批量传输。中断传输与批量传输的主要区别体现在两点：（1）优先级不一样，中断传输的优先级高于批量传输，仅次于同步传输；（2）支持最大包长度不同，中断传输低速模式最大包长上限为 8 字节，全速模式最大包长上限为 64 字节，高速模式最大包长上限为 1024 字节。

需要注意的是，这里所说的中断和硬件上的中断是不一样的。由于 USB 不支持硬件的中断，所以必须靠主机周期性地轮询，以便获知是否有设备需要传送数据给主机。由此可知，中断传输也是一种轮询过程，轮询的周期由用户设备决定（全速设备的轮询间隔为 1ms ~ 255ms，低速设备 10ms ~ 255ms），主机只需要保证在不大于该时间间隔内，安排一次传输即可。轮询周期非常重要，如果太快，会占用太多的总线带宽，如果太低，数据可能会丢，因此用户需要根据自身数据的状况来设置。

中断传输通常用在数据量不大，但是对时间要求较严格的设备中，如人机接口设备 (HID) 中的键盘、鼠标等。中断传输也可用来不断检测设备状态，当条件满足时，再使用批量传输来传送大量数据。中断传输的端点类型一般为 IN 端点，即从 Device 到 Host（IN 事务），很少用在 OUT 端点上，有些电脑甚至不支持中断传输的 OUT 事务。

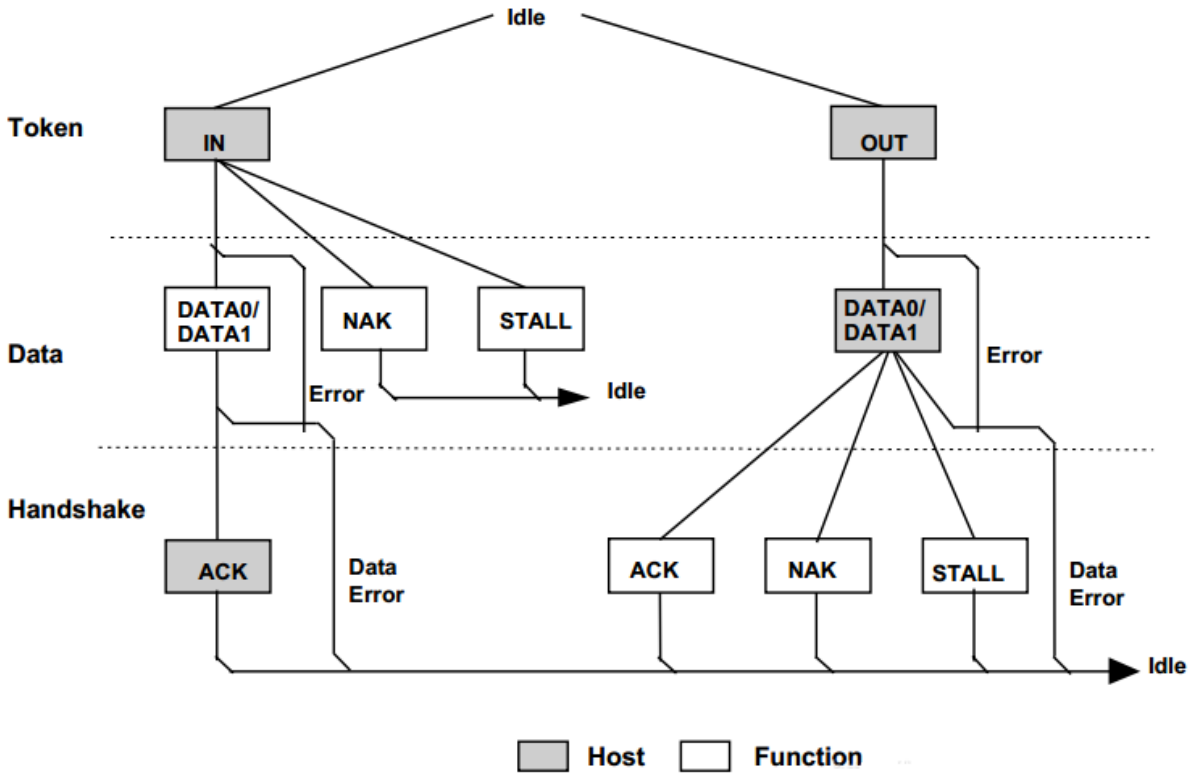


Figure 24.21: 中断传输流程图

24.1.3.3 同步传输

同步传输 (Isochronous Transfer)，又叫等时传输，是不可靠传输。同步传输只有令牌包 (IN/OUT 令牌包) 与数据包 (DATAx) 两个阶段，它没有握手包，也不支持 PID 翻转，主机在排定传输时，同步传输有最高的优先级。同步传输数据包最大长度为全速模式上限为 1023 字节，高速模式上限为 1024 字节，低速模式不支持同步传输。

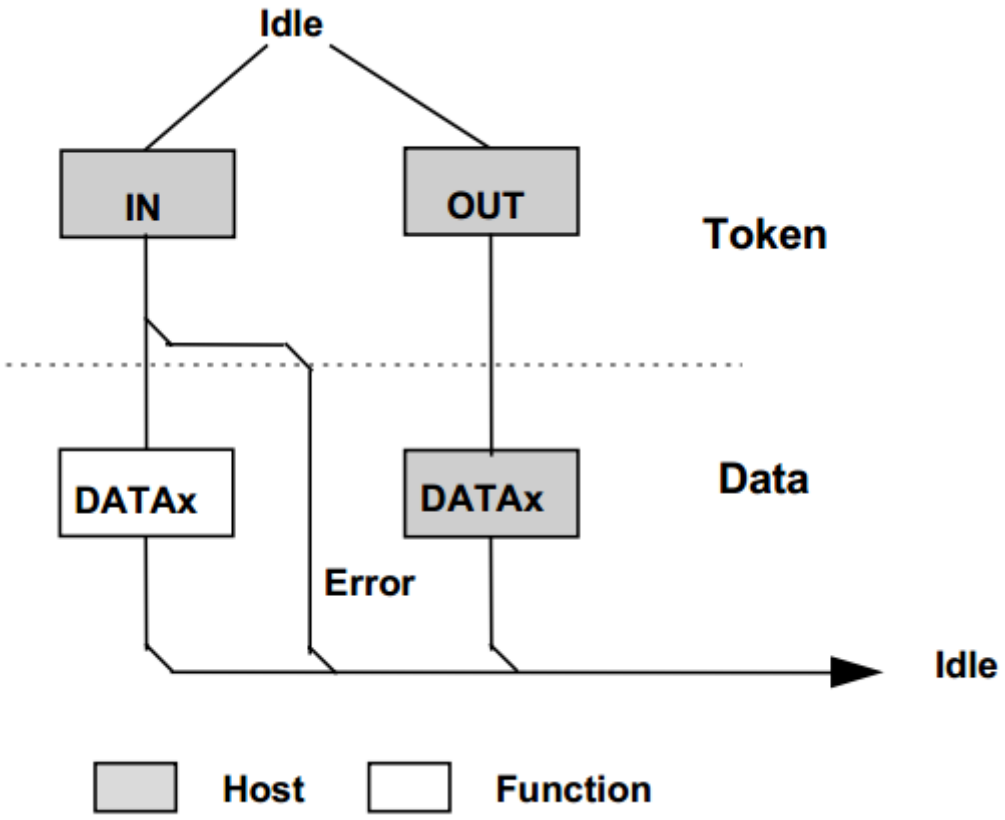


Figure 24.22: 同步传输流程图

同步传输适用于必须以固定速率抵达或在指定时刻抵达，可以容忍偶尔错误的数据上。USB 为其保留总线带宽，保证能在每帧/小帧内都得到服务。速率准确，传输时间可预测。但不采用差错控制和重传机制，不保证每次传输都成功，适用于音频、视频设备。

24.1.3.4 批量传输

批量传输 (Bulk Transfer)，又称为块传输，是单向可靠传输，由一个或多个 IN/OUT 事务组成，事务中的数据包按照 DATA0-DATA1-DATA0-...方式翻转，以保证传输端和接收端的同步，如下图所示。

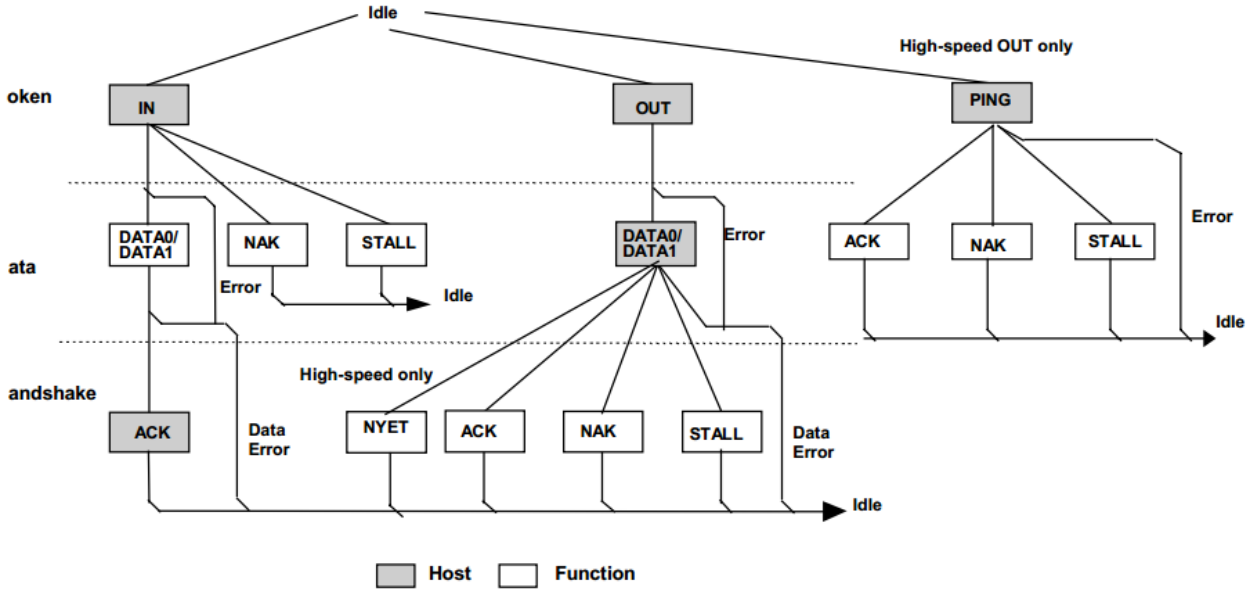


Figure 24.23: 批量传输流程图

USB 中的错误检测和重传机制是通过硬件来完成的，若本次传输错误，DATA 包不会翻转，并重新发送该包。同时，接收端接收到连续 PID 相同的 DATA 包，将视为重传包。USB 允许连续 3 次以下的传输错误，超过 3 次，主机认为该端点功能错误 (STALL)，放弃该端点的传输任务。

24.2 USB 应用

本章节所讲的 USB 应用并不是指 USB 的用途，而是位于 USB 驱动层之上的应用层设计。本章节将从用户的角度，来详细讲解 USB 的基本概念和工作原理，以方便用户熟悉和掌握 USB 的基础知识和使用方法。

24.2.1 基本概念

USB 硬件设备和软件设备的关系为，一个 USB 硬件设备可以对应一个或多个软件设备，这取决于用户的枚举信息（配置描述符信息）。软件设备是 PC 将硬件设备中，实现同一个功能的类的接口抽象出来，并可统一操作的虚拟设备。一个软件设备包含一个或多个接口，一个接口包含一个或多个端点（端点将在下面讲解），而接口和端点是硬件设备中所有的概念。

端点 (Endpoint) 是 USB 设备中的可以进行数据收发的最小单元。除端点 0（固定为双向控制传输）之外，其他所有端点仅支持单向通信，即输入端点（数据流为从设备到主机）或输出端点（数据流为从主机到设备）。设备支持端点的数量是有限制的，除默认端点 0 外，低速设备最多支持 2 组端点（2 个输入，2 个输出），高速和全速设备最多支持 15 组端点。

接口 (Interface) 是 USB 设备中组成一个基本功能的端点的集合，是 USB 设备驱动程序控制的对象（主机会根据接口，在 PC 端虚拟一个可直接操作的 USB 设备，该虚拟设备即是一个 USB 设备类）。从主机的角度来看，一个 USB 设备可由一个或多个接口组成，如集成了鼠标和键盘的 USB 设备，有两个 Interface，一个键盘，另一个是鼠标；如音频设备就是由一个用于命令传输的接口和一个用于数据传输的接口组成。

总结如下：

端点：端点是 USB 设备的唯一可识别部分，其是主机和设备之间的通信流的终点，是一个 USB 设备或主机上的一个数据缓冲区，用来存放和发送 USB 的各种数据。

接口：可以理解为一个功能。

配置：对接口的组合，在连接期间选定是那种组合。

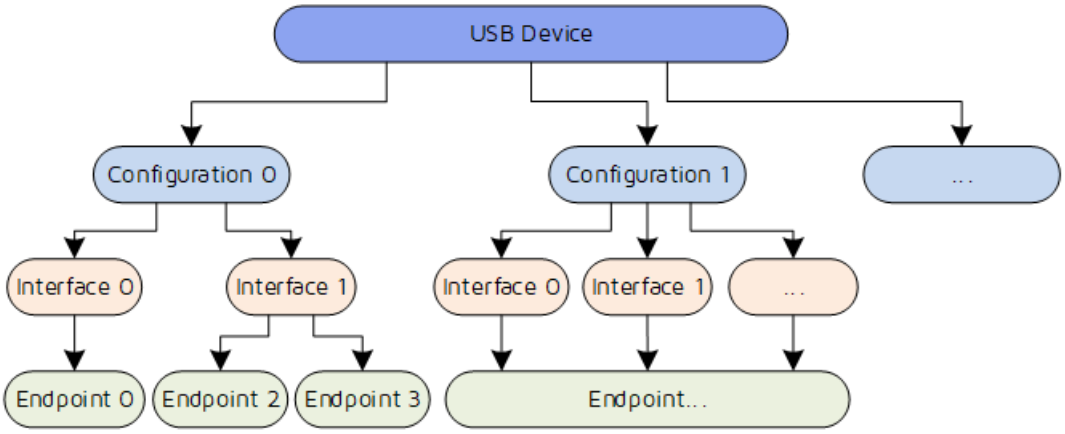


Figure 24.24: USB 应用

24.3 标准描述符

描述符 (Descriptor) 是用来描述设备的属性的数据结，分为标准描述符和专有描述符。标准描述符是所有 USB 设备类通用的属性描述，包括设备描述符、配置描述符、接口描述符、端点描述符和字符串描述符等，其中字符串描述符又分为序列号描述符、产品描述符、厂商描述符以及语言 ID 描述符。专有描述符是每个设备类所特有的描述符，如 HID 类特有的描述符有 HID 描述符、报告描述符和实体描述符等，下图为 USB 协议规定的标准设备请求结构。

Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request (refer to Table 9-3)
2	*wValue*	2	Value	Word-sized field that varies according to request
4	*wIndex*	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	*wLength*	2	Count	Number of bytes to transfer if there is a Data stage

Figure 24.25: 标准设备请求的数据结构

24.3.1 设备描述符

设备描述符描述了有关 USB 设备的基本信息，设备有且只有一个设备描述符。下图给出了标准设备描述符的结构，前 8 个字节概括了 USB 的基本属性，这也是 USB 枚举中主机首先要获取的信息。

偏移量	域	大小	值	描述
0	bLength	1	数字	此描述表的字节数
1	bDescriptorType	1	常量	描述符的类型（此处应为 0x01，即设备描述符）
2	bcdUSB	2	BCD 码	此设备与描述表兼容的 USB 设备说明版本号（BCD 码）
4	bDeviceClass	1	类	设备类码

偏移量	域	大小	值	描述
5	bDeviceSubClass	1	子类	子类掩码
6	bDeviceProtocol	1	协议	协议码
7	bMaxPacketSize0	1	数字	端点 0 的最大包大小（仅 8,16,32,64 为合法值）
8	idVendor	2	ID	厂商标志（由 USB-IF 组织赋值）
10	idProduct	2	ID	产品标志（由厂商赋值）
12	bcdDevice	2	BCD 码	设备发行号（BCD 码）
14	iManufacturer	1	索引	描述厂商信息的字符串描述符的索引值。
15	iProduct	1	索引	描述产品信息的字串描述符的索引值。
16	iSerialNumber	1	索引	描述设备序列号信息的字串描述符的索引值。
17	bNumConfigurations	1	数字	可能的配置描述符数目

注意：

- idVendor(VID) 和 idProduct(PID) 用来唯一标识一个设备，但是对于 Windows 系统来说，仅给定 VID 和 PID，并不能唯一确定设备，表现为不停安装新的驱动。此时，还需要考虑序列号字符串，也就是说只有 VID、PID 和序列号都一致时，Windows 只需要安装一次驱动。
- 三个字符串描述符的索引值应该是不同的值（0 除外）。

24.3.2 配置描述符

配置描述符定义了设备的配置信息，一个设备可以有多个配置描述符。

偏移量	域	大小	值	描述
0	bLength	1	数字	此描述表的字节数长度。
1	bDescriptorType	1	常量	配置描述表类型（此处为 0x02）
2	wTotalLength	2	数字	此配置信息的总长（包括配置，接口，端点描述符）
4	bNumInterfaces	1	数字	此配置所支持的接口个数
5	bConfigurationValue	1	数字	在 SetConfiguration（x）请求中用作参数来选定此配置
6	iConfiguration	1	索引	描述此配置的字串描述表索引（0-无）
7	bmAttributes	1	位图	配置特性：D7：保留（设为一）D6：自给电源 D5：远程唤醒 D4..0：保留（设为一）
8	MaxPower	1	mA	在此配置下的总线电源耗费量，以 2mA 为一个单位

24.3.3 接口描述符

接口描述符说明了接口所提供的配置，一个配置所拥有的接口数量通过配置描述符的 bNumInterfaces 决定。

偏移量	域	大小	值	描述
0	bLength	1	数字	此表的字节数
1	bDescriptorType	1	常量	接口描述表类（此处应为 0x04）
2	bInterfaceNumber	1	数字	接口号，当前配置支持的接口数组索引（从零开始）。
3	bAlternateSetting	1	数字	可选设置的索引值。
4	bNumEndpoints	1	数字	此接口用的端点数量，端点 0 除外
5	bInterfaceClass	1	类	接口所属的类值
6	bInterfaceSubClass	1	子类	子类码
7	bInterfaceProtocol	1	协议	协议码：bInterfaceClass 和 bInterfaceSubClass 域的值而定。
8	iInterface	1	索引	描述此接口的字串描述表的索引值。

24.34 端点描述符

USB 设备中的每个端点都有自己的端点描述符，由接口描述符中的 bNumEndpoint 决定其数量。

偏移量	域	大小	值	描述
0	bLength	1	数字	此描述表的字节数长度
1	bDescriptorType	1	常量	端点描述表类（此处应为 0x05）
2	bEndpointAddress	1	端点	此描述表所描述的端点的地址、方向：Bit 3..0：端点号. 端点编号在改配置中不能重复；Bit 6..4：保留, 为零；Bit 7: 方向, 如果控制端点则略。0：输出端点（主机到设备）1：输入端点（设备到主机）
3	bmAttributes	1	位图	端点的特性。Bit 1..0：传送类型 00= 控制传送 01= 同步传送 10= 批传送 11= 中断传送
4	wMaxPacketSize	2	数字	当前配置下此端点能够接收或发送的最大数据包的大小。对于中断传输，批量传输和控制传输，端点可能发送比之短的数据包。
6	bInterval	1	数字	主机轮询该端点的间隔，对于批量传送和控制传送的端点忽略；对于同步传送的端点，必须为 1；对于中断传输，低速模式此处为 10 ~ 255(ms)，全速模式为 1 ~ 255(ms)。

24.3.5 字符串描述符

字符串描述符是可选的．如果不支持字符串描述符，其设备、配置、接口描述符内的所有字符串描述符索引都必须为 0，语言字符串描述符的索引为 0。

偏移量	域	大小	值	描述
0	bLength	1	数字	此描述表的字节数（bString 域的数值 N ＋ 2）
1	bDescriptorType	1	常量	字串描述表类型（此处应为 0x03）
2	bString	N	数字	UNICODE 编码的字串

24.4 USB 枚举

枚举就是主机从设备端读取一些信息，知道设备是什么样的设备，如何进行通信，这样主机就可以根据这些信息加载合适的驱动程序。调试 USB 设备，很重的一点就是查看 USB 的枚举，只要枚举成功了，那么就已经成功了大半。下面结合 USB 枚举序列图（下图（a））和 Telink 鼠标类 USB 枚举实例（下图（b）），详细介绍 USB 的枚举流程。

24.4.1 USB 枚举序列

下图（a）中给出了 USB 枚举序列图，从图中我们可以看出，USB 枚举过程分为 8 步完成，其中 Step 1 ~ 7 为标准 USB 的枚举过程，Step 8 USB 设备类专有的枚举流程。

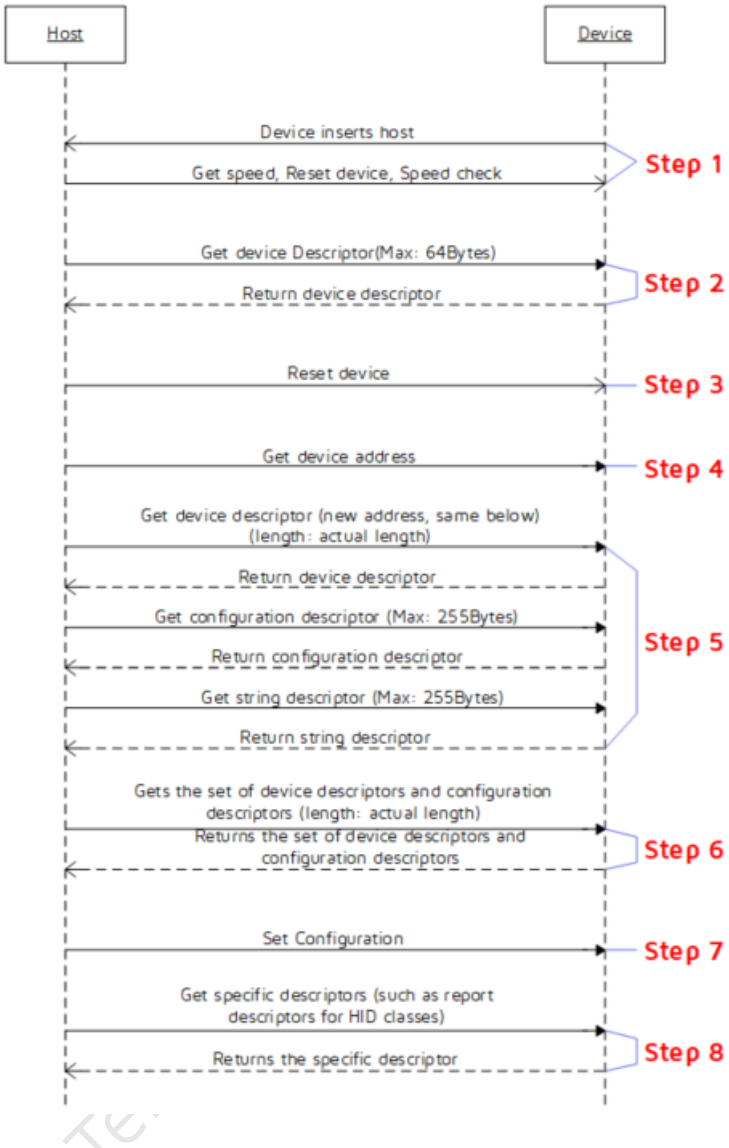


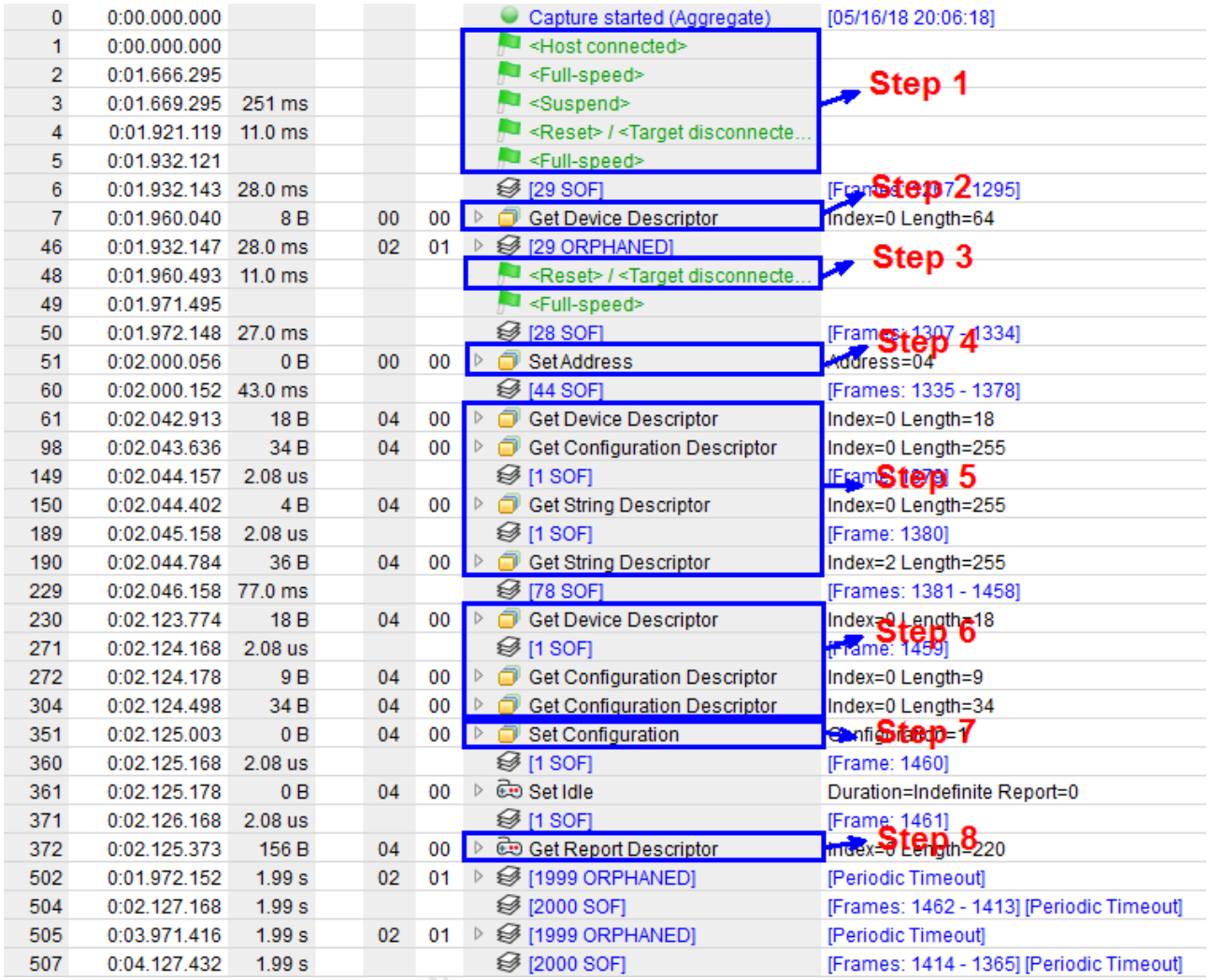
Figure 24.26: （a）USB 枚举序列图

- Step 1 主机检测到有设备接入后：首先，主机根据差分信号线上的电平状态，来判断该设备是低速设备，还是全速设备（高速设备在上电初始时，默认为全速设备）；然后主机等待设备电源稳定 (>=100ms) 后，给设备发送复位信号（D+ 和 D-全为低电平，持续 >=10ms）；最后，如果为高速设备，且主机（Hub）支持高速模式时，主机与设备进行高速检测和握手后，设备可切入高速模式，否则仍维持全速模式。
- Step 2 进行完 Step 1 后，主机会使用端点 0（默认端点，控制传输），发送 GetDescriptor 请求（设备地址为 0），设备接收到请求后，会将自己的设备描述符发给主机，主机会根据此设备描述符（bMaxPacketSize0 字段）来进行下一步动作。需要注意的是：（1）只有设备在 Step 1 中接收到复位信号才去响应主机；（2）已经完成枚举的设备不响应该请求；（3）描述符的长度至少为 8 个字节（bMaxPacketSize0 字段在第 8 个字节）；（4）如果设备超时未响应或响应错误，主机会重新开始，并尝试三次，三次仍无法获取正确的响应，主机会认为该设备为无法识别的设备（下同）。
- Step 3 主机在正确完成 Step 2，获取端点 0 的最大数据包长度后，重新复位设备，之后将按照该长度来对数据包进行拆包和组包。
- Step 4 主机给设备分配一个非 0 的地址，该地址与集线器上其他设备不同，用于保证定向通信的稳定性。主机完成地址设备后，主机和设备的通信将一直按照新的地址进行通信，直至设备复位或移除。

- Step 5 主机按照 Step 4 中设备的新地址，依次获取设备的标准描述符（设备描述符、配置描述符、接口描述符、端点描述符和字符串描述符）。注意：（1）设备描述符的长度，主机在 Step 2 中就已经获得，因此主机指定长度（最大长度为设备描述符长度）获取设备描述符，其他描述符则按照最大长度 255 来获取，设备端只需要按照实际长度发送即可；（2）设备的接口描述符和端点描述符等，可能包含在配置描述符中，主机会根据配置描述符中的 wTotalLength 字段，来获取该配置的所有数据；（3）如果设备有多个配置，主机会分为多次来索要配置描述符；（4）主机会根据设备、配置、接口和端点等描述符中所包含的字符串描述符个数，按照其索引值，依次索要字符串描述符，索引值为 0 的是特殊字符串描述符（语音信息描述符）。
- Step 6 主机在进行完 Step 5 后，获取配置描述符的实际长度，依次获取配置描述符信息和配置描述符中所包含的其他信息（如接口描述符和端点描述符等）。
- Step 7 Step 1 ~ 6 为标准 USB 枚举过程，只有当 Step 1 ~ 6 全部正确后，主机发出 SetConfiguration 的指令，来激活并使用设备的一个配置，此时设备才是真正意义上的可用状态。主机在设备配置完成后，会根据设备的标准描述符来将设备分成一个或多个虚拟设备。
- Step 8 Step 7 完成后，主机上产生一个或多个虚拟设备，每个虚拟设备都有其类别标识，主机会根据其 VID、PID 和序列号查询对应的驱动，并安装驱动（如果主机上有备份，则直接使用，不再安装）。然后，主机根据驱动加载对应类专属描述信息。此处给出标准 HID 设备，主机自带驱动。

244.2 USB **枚举实例**

下图（b）是 Telink Dongle 做鼠标设备时的枚举流程，图中的 Step x 与上图（a）中的 Step x 一一对应，Step 8 为 HID 设备类专有的枚举流程，即获取报告描述符，有关报告描述符的结构可参照 USB HID 协议 Universal Serial Bus (USB)-Device Class Definition for Human Interface Devices (HID)。



0	0:00.000.000				Capture started (Aggregate)	[05/16/18 20:06:18]
1	0:00.000.000				<Host connected>	
2	0:01.666.295				<Full-speed>	
3	0:01.669.295	251 ms			<Suspend>	
4	0:01.921.119	11.0 ms			<Reset> / <Target disconnecte...	
5	0:01.932.121				<Full-speed>	
6	0:01.932.143	28.0 ms			[29 SOF]	[Frames: 1272 - 1295]
7	0:01.960.040	8 B	00	00	Get Device Descriptor	Index=0 Length=64
46	0:01.932.147	28.0 ms	02	01	[29 ORPHANED]	
48	0:01.960.493	11.0 ms			<Reset> / <Target disconnecte...	
49	0:01.971.495				<Full-speed>	
50	0:01.972.148	27.0 ms			[28 SOF]	[Frames: 1307 - 1334]
51	0:02.000.056	0 B	00	00	Set Address	Address=04
60	0:02.000.152	43.0 ms			[44 SOF]	[Frames: 1335 - 1378]
61	0:02.042.913	18 B	04	00	Get Device Descriptor	Index=0 Length=18
98	0:02.043.636	34 B	04	00	Get Configuration Descriptor	Index=0 Length=255
149	0:02.044.157	2.08 us			[1 SOF]	[Frame: 1379]
150	0:02.044.402	4 B	04	00	Get String Descriptor	Index=0 Length=255
189	0:02.045.158	2.08 us			[1 SOF]	[Frame: 1380]
190	0:02.044.784	36 B	04	00	Get String Descriptor	Index=2 Length=255
229	0:02.046.158	77.0 ms			[78 SOF]	[Frames: 1381 - 1458]
230	0:02.123.774	18 B	04	00	Get Device Descriptor	Index=0 Length=18
271	0:02.124.168	2.08 us			[1 SOF]	[Frame: 1459]
272	0:02.124.178	9 B	04	00	Get Configuration Descriptor	Index=0 Length=9
304	0:02.124.498	34 B	04	00	Get Configuration Descriptor	Index=0 Length=34
351	0:02.125.003	0 B	04	00	Set Configuration	Configuration=1
360	0:02.125.168	2.08 us			[1 SOF]	[Frame: 1460]
361	0:02.125.178	0 B	04	00	Set Idle	Duration=Indefinite Report=0
371	0:02.126.168	2.08 us			[1 SOF]	[Frame: 1461]
372	0:02.125.373	156 B	04	00	Get Report Descriptor	Index=0 Length=220
502	0:01.972.152	1.99 s	02	01	[1999 ORPHANED]	[Periodic Timeout]
504	0:02.127.168	1.99 s			[2000 SOF]	[Frames: 1462 - 1413] [Periodic Timeout]
505	0:03.971.416	1.99 s	02	01	[1999 ORPHANED]	[Periodic Timeout]
507	0:04.127.432	1.99 s			[2000 SOF]	[Frames: 1414 - 1365] [Periodic Timeout]

Figure 24.27: （b）Telink Dongle 做鼠标设备时的枚举流程

24.5 USB 硬件介绍

Telink USB 硬件模块内部固化了原始包和事务的处理，自动完成 IN 端点数据的保存和 OUT 端点数据的发送，将端点 0 的数据打包成标准用户数据包，这样不仅可以提高 USB 执行效率，而且最大程度上降低了 USB 开发的难度。

24.6 USB 端点

24.6.1 端点配置

Telink USB 支持 USB1.1 协议，采用外部供电模式，DP 管脚有一个 1.5k 上拉电阻（用户选配），用户可以通过设置模拟寄存器，来使能（置 1）或关闭（置 0）USB 功能，默认为关闭状态。Telink USB 共有 9 个端点，即端点 0 和端点 1 ~ 8，端点 1、2、3、4、7 和 8 可以配置为输入端点，而端点 5 和 6 配置为输出端点。端点 0 只能使用控制传输，端点 1 ~ 8 支持除控制传输以外的其他三种传输模式，在音频应用中，端点 6 和端点 7 默认为同步传输端点（用户可通过清除数字寄存器 0x38 的 bit6 和 bit7 来关闭端点 6 和 7 的同步传输功能）。端点 6 支持同步输出，端点 7 支持同步输入。

可配置端点类型	端点号
控制端点（既可输入又可输出）	0
输出端点	5、6
输入端点	1、2、3、4、7 和 8

24.6.2 端点内存分配

Telink USB 使用 8+256 字节的 USB 专属 RAM 来缓存各个端点的数据，端点 0 的 RAM 地址已固定且大小为 8 字节，其余端点共用 256 字节。用户可根据需要配置地址寄存器，来设置每个端点数据的起始位置，端点的缓存大小为下一个端点的起始地址减去本端点的起始地址，根据下表（USB 端点寄存默认分配表）的默认配置，计算得到 USB 端点资源分配图。硬件会根据用户的配置，将接收到的数据存入相应的 buffer，或者从相应缓冲 buffer 中取数据，并发送给主机。需要注意的是，用户在配置地址寄存器的时候，一定要计算好每个端点的地址，否则可能引发数据覆盖的问题。

端点起始地址	含义
0x00	Endpoint 1 起始地址
0x08	Endpoint 2 起始地址
0x10	Endpoint 3 起始地址
0x20	Endpoint 6 起始地址
0x30	Endpoint 7 起始地址
0x40	Endpoint 4 起始地址
0x80	Endpoint 8 起始地址
0xc0	Endpoint 5 起始地址

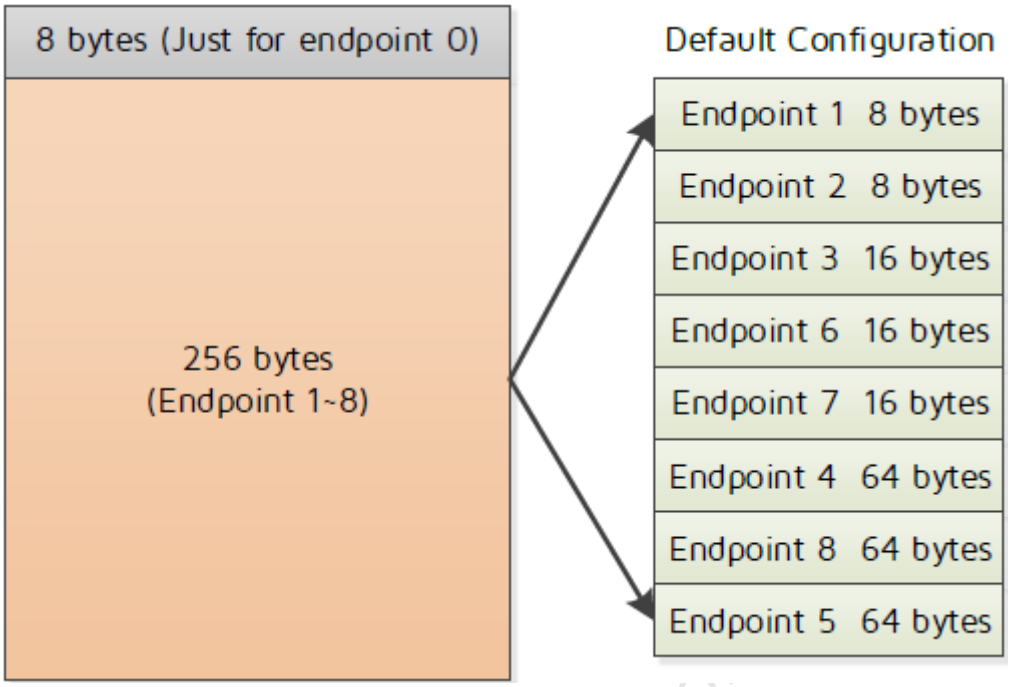


Figure 24.28: USB 端点资源分配图

注意：

- 端点缓存大小由下列两个条件决定：

a) 每个端点的起始位置，端点的缓存大小为下一个端点的起始地址减去本端点的起始地址，例如端点 1 的起始地址为 0x00，端点 2 的起始地址为 0x08，则端点 1 的缓存大有为 0x08 个 bytes。默认的端点地址不是按照端点 1-8 的顺序排列的，而是按照端点的起始地址排序。

b) 端点缓存最大值由 max 寄存器决定（端点 7 除外，可分配到所有缓存空间），默认为 64bytes，所以每个端点的缓存大小，最大不应超过 64Bytes。

- 端点起始地址，只有在枚举设备中使用了，才对端点的缓存分配起作用。例如，usb audio 只使用了端点 6 和 7，其他默认起始地址，不起作用。

24.7 中断

USB 中断可分为三类，端点 0 中断、端点 1-8 中断和 suspend/250us/reset 中断，如下表

中断	产生条件	自动清除还是手动清除
CTRL_EP_SETUP(IRQ7)	端点 0 控制传输 setup 阶段	手动清 status
CTRL_EP_DATA (IRQ8)	端点 0 控制传输 data 阶段	手动清 status
CTRL_EP_STATUS (IRQ9)	端点 0 控制传输状态阶段	手动清 status

中断	产生条件	自动清除还是手动清除
Endpoint(1-8) interrupts(IRQ11) FLD_USB_EDP8_IRQ (in) FLD_USB_EDP1_IRQ (in) FLD_USB_EDP2_IRQ (in) FLD_USB_EDP3_IRQ (in) FLD_USB_EDP4_IRQ (in) FLD_USB_EDP5_IRQ (out) FLD_USB_EDP6_IRQ (out) FLD_USB_EDP7_IRQ (in)	1. 除同步端点：输出端点：host out 事务，状态寄存器的相应位置 1，产生中断，接收完回 ACK。输入端点：数据填充完成后，配置 ACK 通知硬件，并产生中断，硬件在收到 host in 事务中，将数据发送给 host。 2. 同步端点：端点 6,7 可设置为同步端点，定时 1ms 产生中断。	手动清 status
USB_IRQ_USB_SUSPEND (IRQ24)	USB 总线空闲，例如拔掉 USB 接口，host 休眠	手动清 status
USB_IRQ_250us (IRQ34)	250us 定时中断	手动清 status
USB_IRQ_RESET (IRQ35)	Host 发送 reset 时序	手动清 status

注意：

- Driver 枚举过程的传输都是采用轮询的方式处理，没有使用中断方式。

24.8 自动和手动模式

Telink USB 有两种模式，即自动模式和手动模式：

用户可通过设置端点 0 的配置寄存器，来控制是选择自动模式还是手动模式。端点 0 的配置寄存器默认为 0xFF，即自动模式，此时与 USB 端点 0 相关的所有编解码均由 Telink 硬件自动驱动完成，Telink 自带驱动为 Print 设备，使用端点 8 作为控制接口的端点，打印机数据是从端点 0 发送的；

手动模式需要用户修改 EDP0CFG 寄存器（下图），一般是将 bit[7] 和 bit[5] 设为 0，即由用户去完成标准 USB 的枚举和使用用户定义的描述符。

0x 04	EDP0CFG	[0]: Decode SetAddress Mode (0-Manu, 1-Auto) [1]: Decode SetConfig Mode (0-Manu, 1-Auto) [2]: Decode SetInterface Mode (0-Manu, 1-Auto) [3]: Decode GetStatus Mode (0-Manu, 1-Auto) [4]: Decode SyncFrame Mode (0-Manu, 1-Auto) [5]: Decode GetDescriptor Mode (0-Manu, 1-Auto) [6]: Decode SetFeature Mode (0-Manu, 1-Auto) [7]: Decode Standard Mode (0-Manu, 1-Auto)	RW	0xFF

Figure 24.29: 端点 0 配置寄存器

24.9 USB 软件基础

24.10 USB 运行流程

Telink USB 的软件运行流程可以分为两个阶段，即初始化阶段和循环检测阶段，如下图所示。

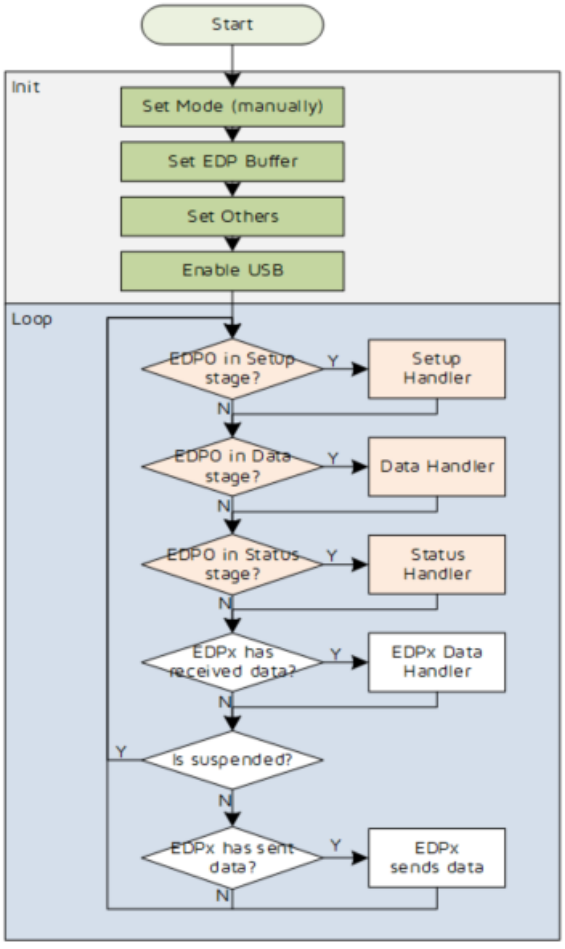


Figure 24.30: Telink USB 运行流程图

初始化阶段主要完成 USB 的相关配置和使能 USB。USB 配置选项主要包括模式切换（自动模式和手动模式）、设置 USB 数据缓冲区和设置其他配置项；模式切换主要是将 USB 的工作模式切换为手动模式，此时相关的枚举过程和描述符由用户控制，设备会将用户准备好的枚举信息上报给主机；设置 USB 缓冲区是用户根据自己端

点（端点 0 除外）的使用情况，分别给相关端点指定一段缓冲区（缓冲区总大小 256Bytes，没有使用到的端点不用指定）；设置其他配置是用来进行其他配置的操作，用户如果不想使用系统默认的配置项，可选择自行配置，如以中断形式传输数据等。

循环检测主要是不断检测数据接收和发送缓冲区是否有数据，如果有数据则进行相关操作，程序运行过程中会反复执行 usb_handle_irq。端点 0 的操作流程可分为三个阶段，如下图所示，分别对应控制传输的 SETUP 阶段、DATA 阶段和 STATUS 阶段，主要完成 USB 的识别和配置等相关操作，如 USB 的枚举，该过程是在 main loop 中完成，SETUP 解析主机下发的指令，并根据主机的指令，来准备相应的数据；DATA 是将在数据阶段准备的数据发给主机或者接收主机发送的数据；STATUS 是双方握手的过程。

```
void usb_handle_irq(void) {
    u32 irq = usbhw_get_ctrl_ep_irq();
    if (irq & FLD_CTRL_EP_IRQ_SETUP) {
        usbhw_clr_ctrl_ep_irq(FLD_CTRL_EP_IRQ_SETUP);
        usb_handle_ctl_ep_setup();
    }
    if (irq & FLD_CTRL_EP_IRQ_DATA) {
        usbhw_clr_ctrl_ep_irq(FLD_CTRL_EP_IRQ_DATA);
        usb_handle_ctl_ep_data();
    }
    if (irq & FLD_CTRL_EP_IRQ_STA) {
        usbhw_clr_ctrl_ep_irq(FLD_CTRL_EP_IRQ_STA);
        usb_handle_ctl_ep_status();
    }
```

Figure 24.31: 端点 0 数据操作流程

24.11 数据接收与发送

24.11.1 数据接收

Telink USB 数据接收由硬件完成，硬件会将接收到的数据保存到 RAM 中，接收完成后硬件会产生一个中断来通知用户，用户只需要在检测到中断后，读取数据即可。数据检测和接收工作应该在 usb.c 中的 usb_handle_irq 函数中进行，结合下图，具体分析 Telink USB 数据接收的处理流程：

(1) 用户需要检测相关的中断标识位（reg_usb_irq）是否置 1，如果置 1 则进入数据接收阶段。

(2) 一旦检测到数据后，用户需要将中断标识位清除，即 reg_usb_irq = BIT((USB_EDP_CUSTUM_OUT & 0x07))。

(3) 用户读取数据之前，需要使用 reg_usb_ep_ptr(USB_EDP_CUSTUM_OUT) 来获取接收到的数据长度。

(4) 用户获取完数据长度后，即可通过反复读取 usbhw_read_ep_data(USB_EDP_CUSTUM_OUT) 来获取本次接收到的所有数据。

(5) 用户接收完数据后，需要将调用 usbhw_data_ep_ack(USB_EDP_CUSTUM_OUT)。(注意，这一步很重要，只有将 OUT 端点的 ACK 置起，硬件才会去接收主机下发到该端点的数据，并在接收完成后产生中断。)

```
irq = reg_usb_irq;
if(irq & BIT((USB_EDP_CUSTUM_OUT & 0x07))){
    reg_usb_irq = BIT((USB_EDP_CUSTUM_OUT & 0x07));

    gUsbRecvLen = reg_usb_ep_ptr(USB_EDP_CUSTUM_OUT);
    usbhw_reset_ep_ptr(USB_EDP_CUSTUM_OUT);

    for(int i=0; i<gUsbRecvLen; i++)
    {
        gUsbRecvBuff[i] = usbhw_read_ep_data(USB_EDP_CUSTUM_OUT);
    }

    usbhw_data_ep_ack(USB_EDP_CUSTUM_OUT);

    usb_receive_irq_handler(gUsbRecvBuff, gUsbRecvLen);
}
```

Figure 24.32: Telink USB 数据接收

24.11.2 数据发送

Telink USB 数据发送和数据接收一样，是由硬件完成，用户只需要将数据填充到相应的 USB RAM 中，并将数据 ACK 位置 1。用户在填充数据之前，需要首先检测 USB RAM 中是否有待发送数据，如果有待发送数据，则需要等待发送完成之后再填充新的数据，否则会发生数据覆盖现象。

下图给出了 Telink SDK 中，发送数据的实例，下面结合该实例，详细分析 USB 数据发送流程：

(1) 用户发送数据前，需要检测操作端点是否繁忙，如果繁忙（有待发送数据），则需要等待发送完成后，再填充数据。

(2) 如果端点处于空闲状态，需要先重置端点计数器，即 reg_usb_ep_ptr(USB_EDP_CUSTUM_CMISC_IN) = 0。

(3) 重置完端点计数器后，用户就可向端点中填充数据。需要注意的时，reg_usb_ep_dat(USB_EDP_CUSTUM_CMISC_IN) = data[i] 是将数据放到 USB RAM 中（硬件操作）。

(4) 用户填充完数据后，需要调用 reg_usb_ep_ctrl(USB_EDP_CUSTUM_CMISC_IN) = FLD_EP_DAT_ACK 来通知硬件数据已经准备好了，硬件在收到该指令后，会在下一个主机索取数据的时候，将数据发送给主机。

```
if(usbhw_is_ep_busy(USB_EDP_CUSTUM_IN)) return -1;

reg_usb_ep_ptr(USB_EDP_CUSTUM_IN) = 0;

for(int i=0; i<length; i++)
{
    reg_usb_ep_dat(USB_EDP_CUSTUM_IN) = data[i];
}

reg_usb_ep_ctrl(USB_EDP_CUSTUM_IN) = FLD_EP_DAT_ACK;

return 0;
```

Figure 24.33: Telink USB 数据发送

24.12 USB demo

USB 应用主要介绍了 USB 标准设备类中的 HID（Human Interface Device）设备，Audio 设备，CDC（Communication Device Class）设备的简单应用，客户可根据需求自由组合。

HID 类设备是 USB 设备中常用的设备类型，是直接与人交互的 USB 设备，如 USB mouse，USB keyboard；

USB Audio 类设备最为常见的是 microphone 和 speaker；

USB 的 CDC 类是 USB 通信设备类简称，虚拟串口设备是 CDC 类设备的一种类型。

在头 USB_DEMO/app_config.h 中可以选择配置成不同的设备。

```
#define     USB_MOUSE           1
#define     USB_KEYBOARD        2
#define     USB_MICROPHONE      3
#define     USB_SPEAKER         4
#define     USB_CDC             5
#define     USB_MIC_SPEAKER     6

#define     USB_DEMO_TYPE       USB_MOUSE
```

24.12.1 USB mouse

24.12.1.1 Mouse **处理流程**

USB HID 设备是通过 report 来传输数据，一个报告描述符可以描述多个报告，不同的报告通过 ID 来识别，报告 ID 为报告的第一字节，没有规定报告 ID 时，报告没有 ID 字段，开始就是数据，详细的报告描述符资料可参考 USB HID 协议以及 HID 用途表（HID Usage Tables）。

首先 host 将 Telink USB 识别成 mouse 设备，需要经历枚举阶段，设备枚举成功后，进入数据收发阶段。根据 mouse 报告描述符的内容，报告 ID 为 USB_HID_MOUSE 的描述符中，有 4 个字节。第 1 字节字的低 5 位表示按键是否按下，高 3 位为常数无用；第 2 字节为 X 轴的改变量；第 3 个字节为 Y 轴的改变量；第 4 个字节为滚轮的改变量。通过函数 usbmouse_hid_report(USB_HID_MOUSE,mouse,4) 返回报告。

Demo 程序中，定义了数组 unsigned char mouse[4]，其中：mouse[0]：BIT(0) - left key; BIT(1) - right key; BIT(2) - middle key; BIT(3) - side key; BIT(4) - external key. 对应的 bit 置 1，代表对应鼠标键按下；mouse[1]：相对于 x 坐标的改变量；mouse[2]：相对于 y 坐标的改变量；mouse[3]：滚轮的改变量。

24.12.1.2 Mouse **测试**

按下测试

Demo 程序中，报告 ID：USB_HID_MOUSE=1,

数组 mouse 赋值：mouse[0]=BIT(1)，mouse[1]= -2（补码），mouse[2]=2, mouse[3]=2。

将开发板上的引脚 PD1 接地后再拔出，会执行函数 usbmouse_hid_report(USB_HID_MOUSE,mouse,4)。

可以观测到桌面的鼠标右键按下，鼠标光标向左下移动，如下图所示，也可以从 USB 抓包工具 Input Report[1] 中看到：x:-2,Y:2,wheel:0,Btns=[2]。

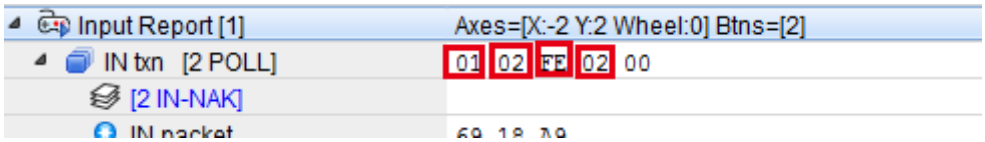


Figure 24.34: Mouse Input Report 抓包图

释放测试

对引脚 PD2 进行和 PD1 一样的操作，mouse 数组清零，按键释放。

24.12.2 USB keyboard

24.12.2.1 Keyboard **处理流程**

根据 keyboard 报告描述符的内容，有输入和输出报告，其中输入报告规定了 8 个字节，第 1 个字节的 8 位表示特殊键是否按下。

BYTEO：BIT(0) – 左 Ctl；BIT(1) – 左 Shift；BIT(2)-左 Alt；BIT(3) - 左 GUI

BIT(4) – 右 Ctl；BIT(5) – 右 Shift；BIT(6) - 右 Alt；BIT(7) - 右 GUI

第 2 字节为保留值，都为 0

BYTE1：0

第 3 到 8 字节是一个普通键键值，当没有键按下时，全部 6 字节都为 0，这 6 个字节的第一字节值即为按键的键值，当有多个按键同时按下时，则同时返回这些按键值，键值在数组的先后顺序无关。具体键值请参考 HID 用途表文档，例如：0x59 对应数值键盘 1；0x5a 对应数值键盘 2；0x5b 对应数值键盘 3；0x39 对应大小写切换键。

24.12.2.2 Keyboard **测试**

按下测试

Demo 程序中，定义数组 kb_data[6]，赋值为 kb_data[0] = 0;kb_data[1] = 0;kb_data[2] = 0x59;kb_data[3] = 0x5a; kb_data[4] = 0x39;kb_data[5] = 0;

将开发板上的引脚 PC1 接地后再拔出，会执行函数 usbkb_hid_report_normal(0x10,kb_data)，其中参数 1 对应第一字节，参数 2 是对应 3 ~ 8 字节的数组。

可以观测到在编辑窗口输入界面，会输入数字 1 和 2，右 Ctrl 键和大小写键按下。

如下图，也可以从 USB 抓包工具 Input Report 中看到：Keys=[Rctrl 1 2 CapsLk]

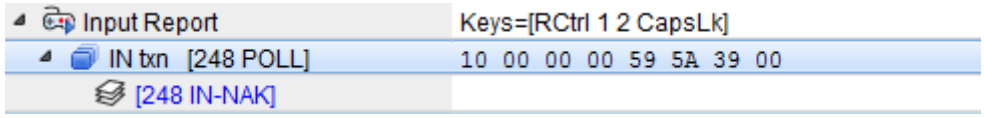


Figure 24.35: Keyboard Input Report 抓包图

释放测试

对 GPIO_PC2 进行和 GPIO_PC1 一样的操作，特殊键和 kb_data 数组清零，按键释放。

24.12.3 USB MIC

24.12.3.1 MIC **处理流程**

AMIC 为例，USB microphone 设备是将 device 的 AMIC 数据通过 USB 传输到 host，需要保证整条数据通道采样率和通道数匹配。Demo 程序中，主要是将数据上传到 usb 部分不同, Mic 端点中断是 1ms 定时中断，也即 1ms 进一次中断，根据不同采样率，1ms 产生数据量不同，例如 16K 采样率收音，单声道数据，1 个 sample 为 2 bytes，1ms 数据为 32bytes，将对应 audio buff 的填入 usb sram 中。

24.12.3.2 Mic demo **测试**

Audio 相关的测试，借助 Audacity 软件，如下图麦克风选择 Telink Audio16，扬声器选择 PC 扬声器。

Device Mic 收音，PC 扬声器播放，如录音的人声能不失真通过扬声器播放出来，说明 Mic 工作正常。

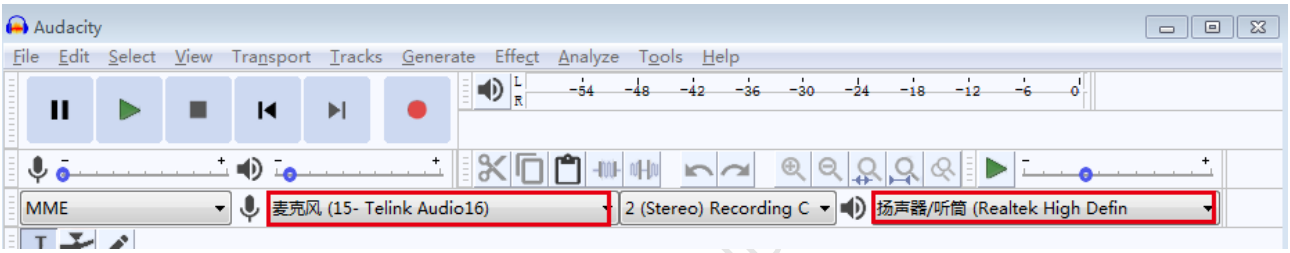


Figure 24.36: Audacity 软件设置 (Mic)

24.124 USB speaker

24.124.1 Speaker **处理流程**

USB speaker 设备是将 host 的音频数据通过 USB 传输到 device ，这个处理过程也在 1ms 中断完成的，读出 usb sram 数据长度，将对应的数据填充到 audio buff。

24.124.2 Speaker demo **测试**

在 Audacity 软件中，麦克风选择 PC 麦克风，扬声器选择 Telink Audio16。通过输出音频接口（3.5mm 耳机插口）。如录音的人声能不失真的通过耳机播放出来，说明 speaker 工作正常。

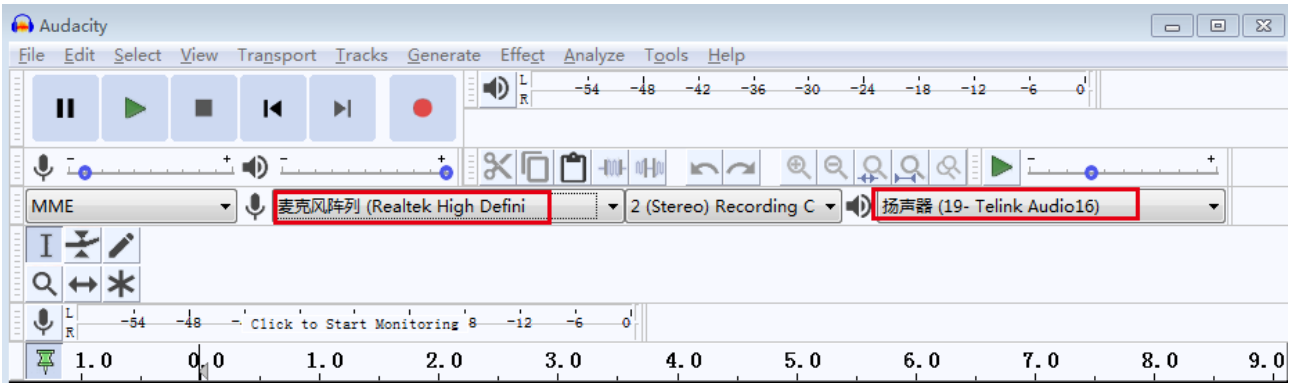


Figure 24.37: Audacity 软件设置 (Spk)

24.12.5 USB CDC

CDC 设备有两个接口，CDC 控制接口和 CDC 数据接口，控制接口分配端点 2，作为中断输入端点传输。数据接口分配端点 5（out），端点 4（out），一定要先将端点 5 的 ACK 置起，端点才可以从 USB 主机接收数据。

24.12.5.1 CDC 处理流程

CDC 设备 USB 安装 host 首次识别成 CDC 设备需要手动安装.inf 文件，如下图在 8278_USB_Demo 路径下。

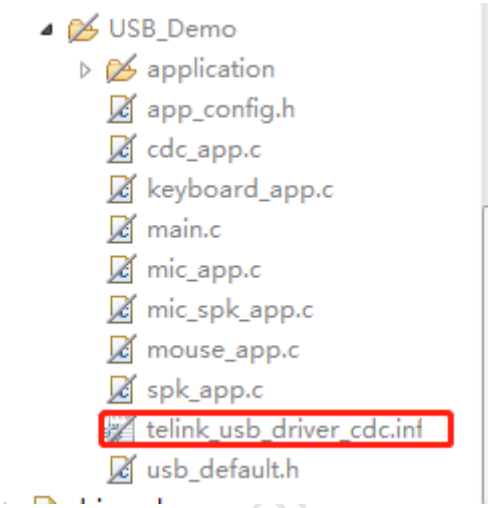


Figure 24.38: .inf 文件路径

数据接收 (host to device)

Demo 程序中，在函数 void usb_cdc_irq_data_process(void) 中 host 发送数据给 device，端点 5 产生中断后，函数 usb_cdc_rx_data_from_host(usb_cdc_data) 进行数据接收。

数据发送 (device to host)

如下图所示,在 main_loop 里当判断接收 buff 数据长度不为 0 时,通过函数 usb_cdc_rx_data_from_host(usb_cdc_data) 将接收的数据发送给 host。

```
45 void main_loop (void)
46 {
47     usb_handle_irq();
48
49     if(usb_cdc_data_len!=0)
50         {
51         usb_cdc_tx_data_to_host(usb_cdc_data,usb_cdc_data_len);
52         usb_cdc_data_len = 0;
53     }
54
55 }
```

Figure 24.39: 数据发送

24.12.5.2 CDC demo **测试**

测试现象如下图所示，将串口助手发送的数据返回。

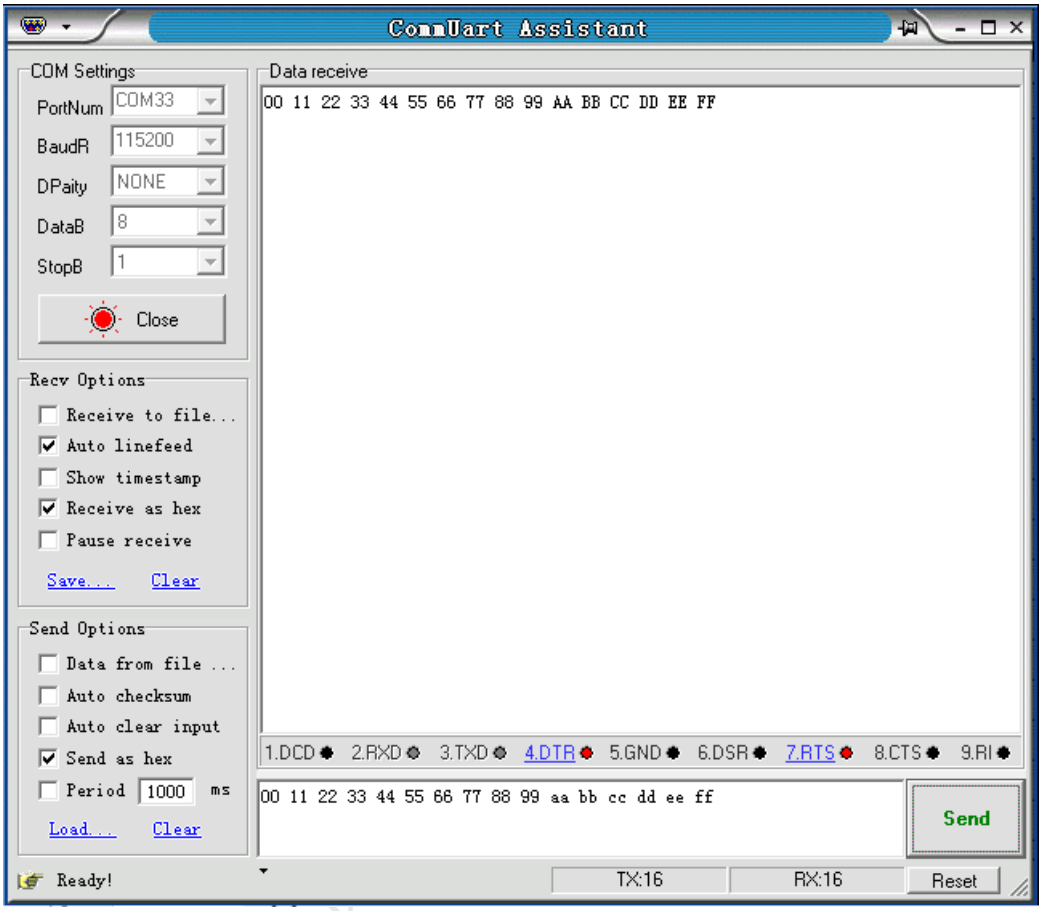


Figure 24.40: CDC 数据收发测试

25 CPU 性能测试

衡量处理器的一个重要指标是功耗，另外一个重要指标便是性能。在处理器领域的 Benchmarks 非常众多，在嵌入式处理器领域最为知名和常见的 Benchmarks 为 Dhrystone 和 CoreMark。

25.1 Dhrystone

Dhrystone 标准的测试方法很简单，就是单位时间内跑了多少次 Dhrystone 程序，其指标单位为 DMIPS/MHz。MIPS 是 Million Instructions Per Second 的缩写，每秒处理的百万级的机器语言指令数。DMIPS 中的 D 是 Dhrystone 的缩写，它表示了在 Dhrystone 标准的测试方法下的 MIPS，主要用于测整数计算能力。

25.2 CoreMark

CoreMark 程序使用 C 语言写成，包含如下四类运算法则：数学矩阵操作（普通矩阵运算）、列举（寻找并排序）、状态机（用来确定输入流中是否包含有效数字）、CRC（循环冗余校验）。与 Dhrystone 类似，CoreMark 标准的测试方法：在某配置参数组合下单位时间内跑了多少次 CoreMark 程序，其指标单位为 CoreMark/MHz。CoreMark 数字越高，意味着性能更高。

25.3 测试

Dhrystone 和 CoreMark 测试对应的 Demo 分别为 Dhrystone_Demo 和 CoreMark_Demo。测试的信息由 USB 输出。

Benchmarks	测试结果	备注
Dhrystone	2.7 (DMIPS)	attach1
CoreMark	3.38	attach2

注意：

- 为提高测试性能，上述两个 demo 对应的 link 文件为 flash_boot_ramcode.link。

26 Audio

26.1 Audio 简介

26.1.1 声音基础

声音（sound）是由物体振动产生的声波，是一种机械波。音频录制（record）过程是模数转换过程，播放（playback）过程相反。

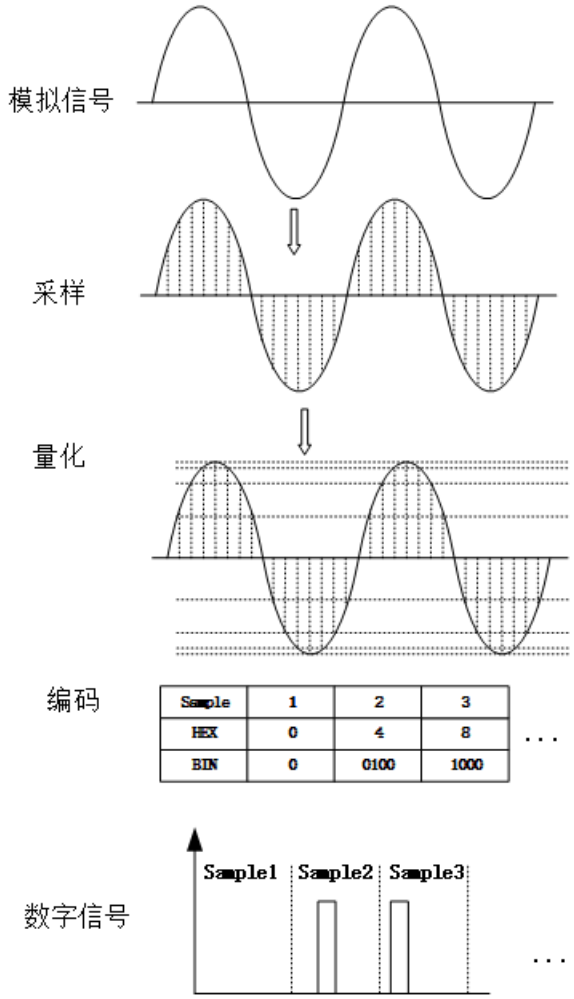


Figure 26.1: 模数转换过程

- 采样：对模拟信号隔一定的时间间隔取一个点。
- 量化：给纵坐标加刻度，根据近似取整数值，使采样得到的点的值都是整数。
- 编码：对量化取得的整数值按二进制进行编码。
- 数字信号：把编码得到的 0 和 1 的序列变为高低电平的信号。

上述整个模数转换的过程称为：脉冲编码调制（Pulse Code Modulation），简称 PCM。由上面的模数转换可知，PCM 格式文件存储的内容实际上就是编码得到的序列。

26.1.2 采样音频基本概念

采样频率：所谓采样就是在时间轴上对模拟信号进行数字化，根据奈奎斯特定理（采样定理），按照比声音最高频率 2 倍以上的频率进行采样（AD 转换）。频率在 20 Hz ~ 20 kHz 之间的声音是可以被人耳识别的。所以采样频率一般为 40kHz 左右，常用的音乐为 44.1kHz(44100 次/s 采样)、48kHz 等，电话的采样率为 8K。

采样位数：每个采样点能够表示的数据范围。采样位数通常有 8 bits 或 16 bits 两种，采样位数越大，所能记录声音的变化度就越细腻，相应的数据量就越大。16 bit 是最常见的采样精度。

声道数：声道数是指支持能不同发声的音响的个数，常用的声道数有单声道，立体声（左声道和右声道）。

例如，cd 音质的相关参数为，采样位宽 16bit，采样率 44100，声道数 2，用来衡量音频数据单位时间内的容量大小，cd 音质的数据比特率则为：44100 * 16 * 2 = 1411.2 kbps。

PCM 音频数据： PCM 是采样量化后的未压缩音频数据，由模拟信号经过采样、量化、编码转换成的标准的数字音频数据。如果是单声道的音频文件，采样数据按时间的先后顺序依次存入（如果是双声道的话就按照 LRLR 的方式存储，存储的时候还和机器的大小端有关）。大端模式如下图所示。

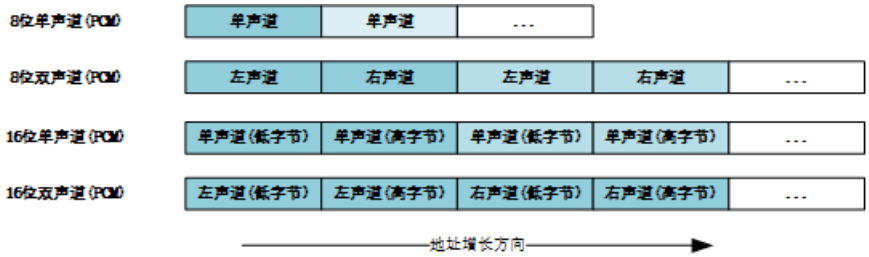


Figure 26.2: PCM 数据格式

26.1.3 I2S 协议

I2S（Inter-IC Sound）总线, 又称集成电路内置音频总线，是飞利浦半导体公司（现为恩智浦半导体公司）针对数字音频设备之间的音频数据传输而制定的一种总线标准，该总线专门用于音频设备之间的数据传输。

26.1.3.1 I2S 信号线

（1）串行时钟 BCLK

串行时钟 BCLK，对应数字音频的每一位数据，BCLK 都有一个脉冲。

（2）帧时钟 LRCK

帧时钟 LRCK，用于切换左右声道的数据。LRCLK（Left/Right CLOCK），LRCK 的频率 = 采样频率。

（3）串行数据（SDATA）

就是用二进制补码表示的音频数据，（MSB —> LSB：数据由高位到低位依次传输）。

26.1.3.2 I2S 数据格式

根据 data 相对于 LRCLK 与 BCLK 位置的不同，分为 I2S 标准格式（I2S），左对齐（LJ）和右对齐（RJ），发送和接收端必须使用相同的数据格式。

注意：

- 驱动默认的是 I2S 格式。

(1) I2S format

数据的最高位总在 LRCLK 变化（也就是一帧开始）后的第 2 个 BCLK 脉冲处，时序图如下图所示。

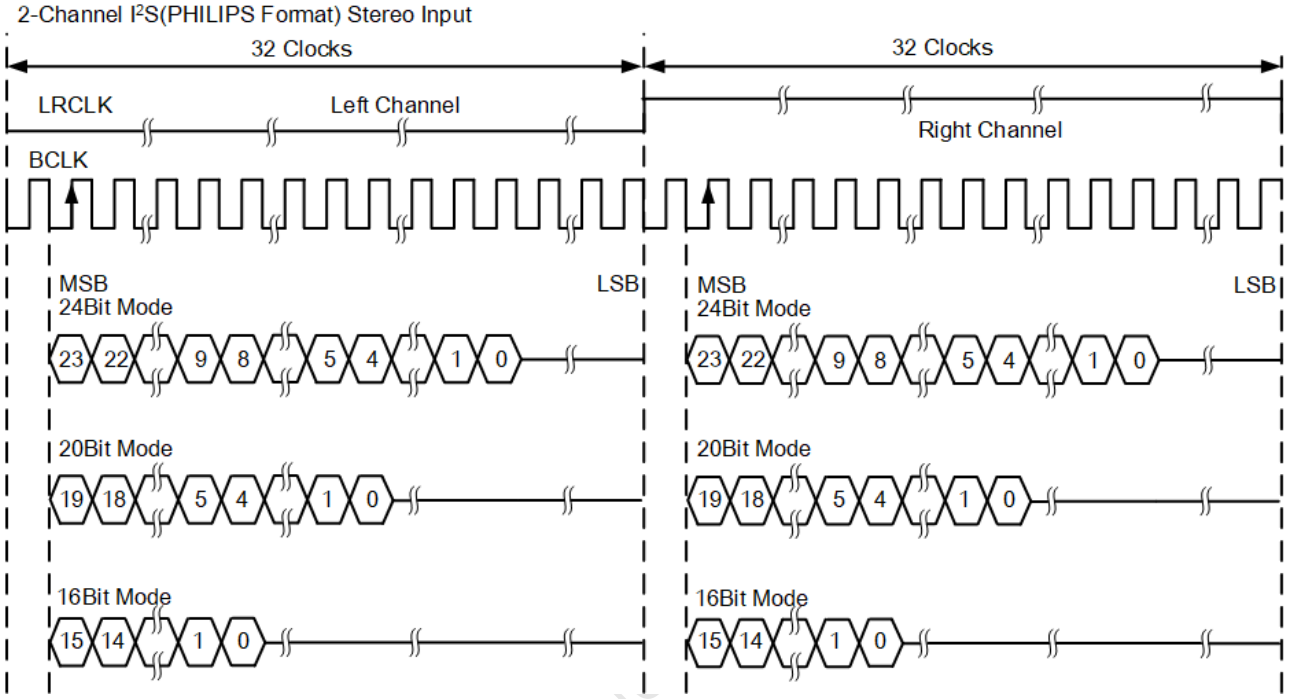


Figure 26.3: I2S format

(2) LJ format

在 LRCLK 发生翻转的同时开始传输数据。注意此时 LRCLK 为 1 时，传输的是左声道数据，这刚好与 I2S Philips 标准相反。左对齐 (MSB) 标准时序图如下图所示。

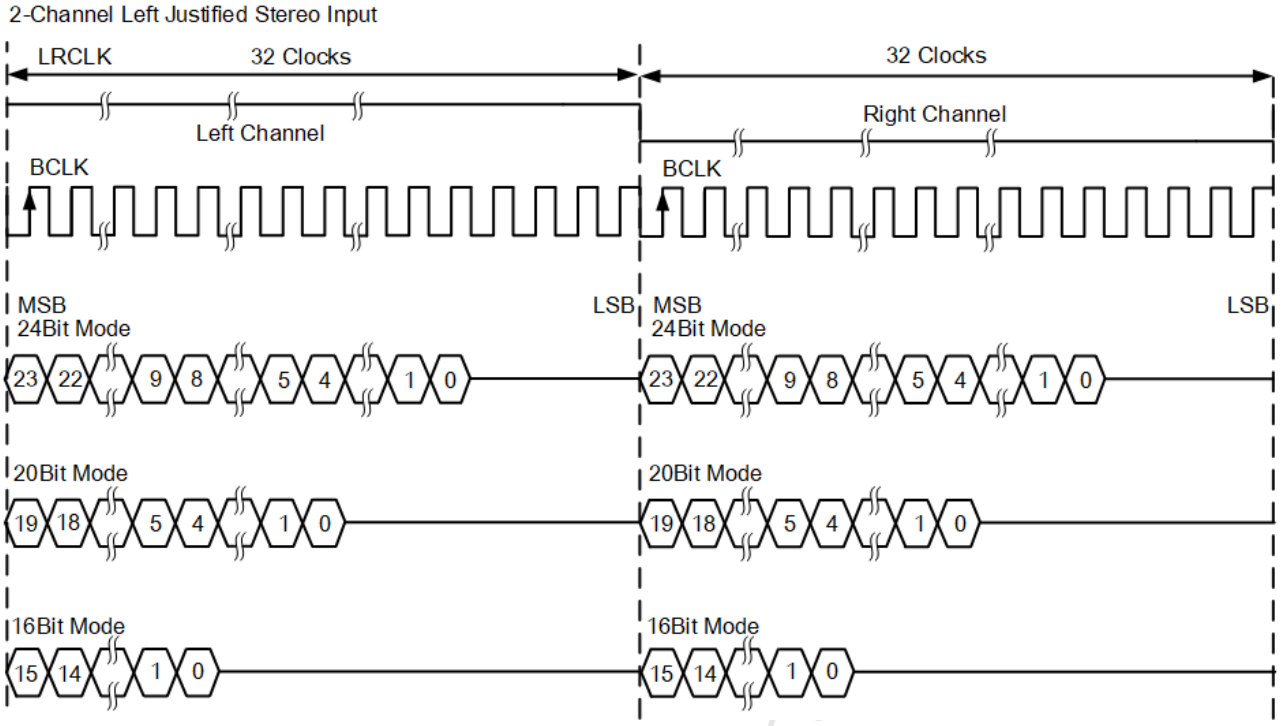


Figure 26.4: LJ format

(3) RJ format

声音数据 LSB 传输完成的同时，LRCLK 完成第二次翻转（刚好是 LSB 和 LRCLK 是右对齐的，所以称为右对齐标准）。注意此时 LRCLK 为 1 时，传输的是左声道数据。

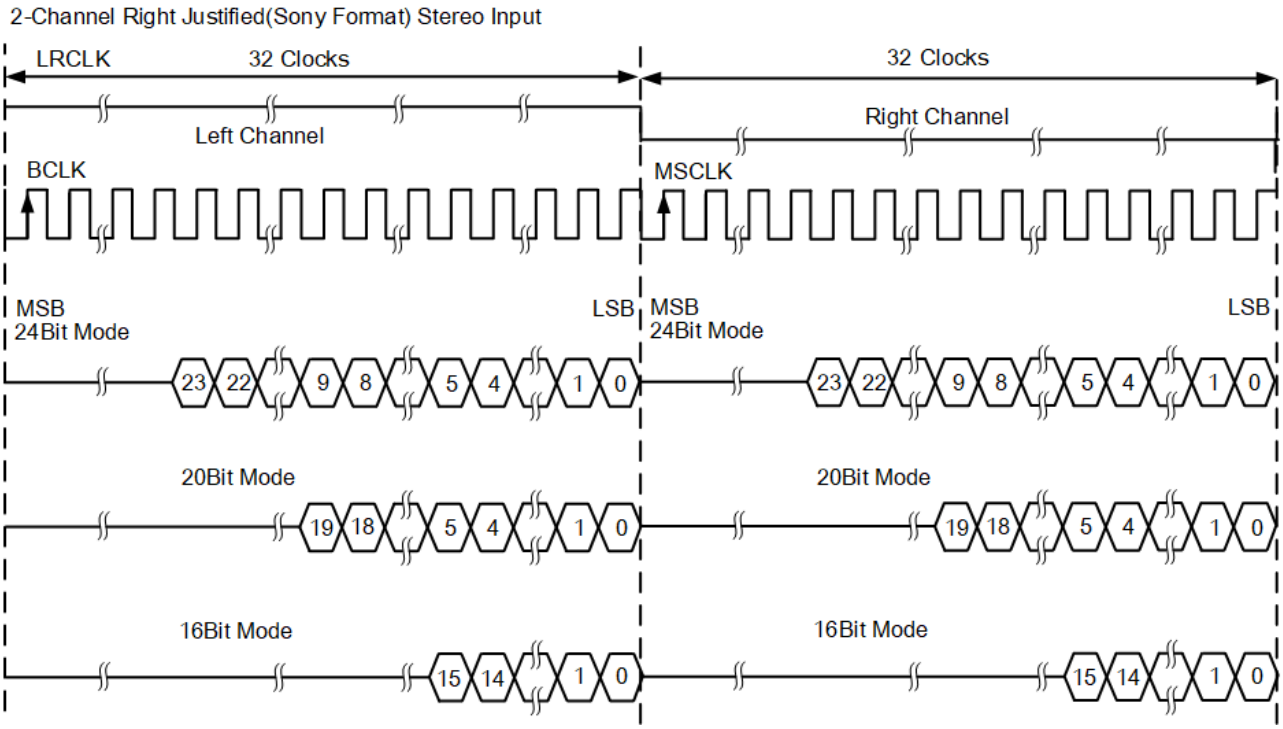


Figure 26.5: RJ format

(4) DSP/PCM mode

DSP/PCM mode 分为 Mode-A 和 Mode-B 共 2 种模式。不同芯片的 datasheet 中有的称为 PCM mode 有的称为 DSP mode。I2S 左右声道分别为高低电平，PCM 只有一个起始信号，左声道数据紧跟右声道。如下图 A，Mode-A 数据在第 1 个 BCLK 脉冲处。如下图 B，Mode-B 数据在第 2 个 BCLK 脉冲处。

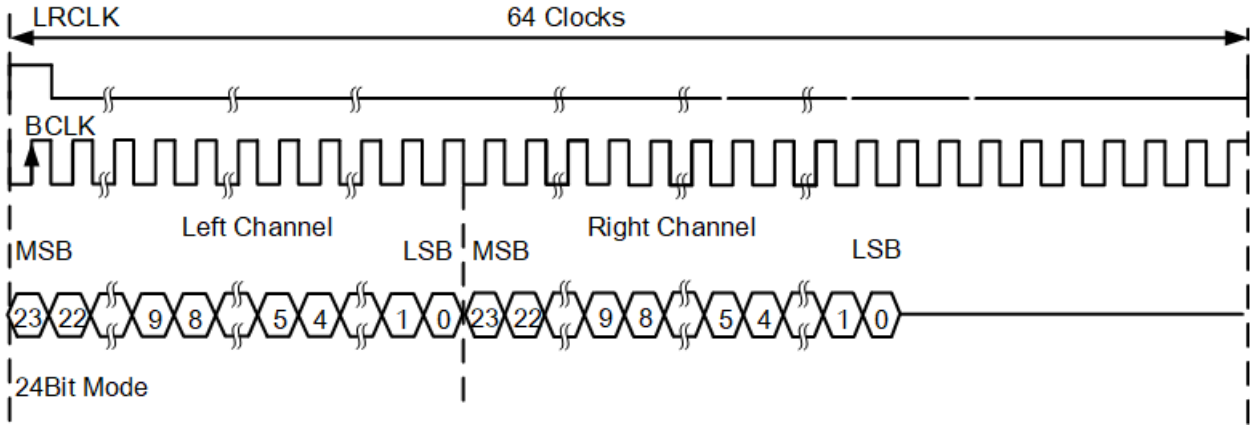


Figure 26.6: DSP format (Mode-A)

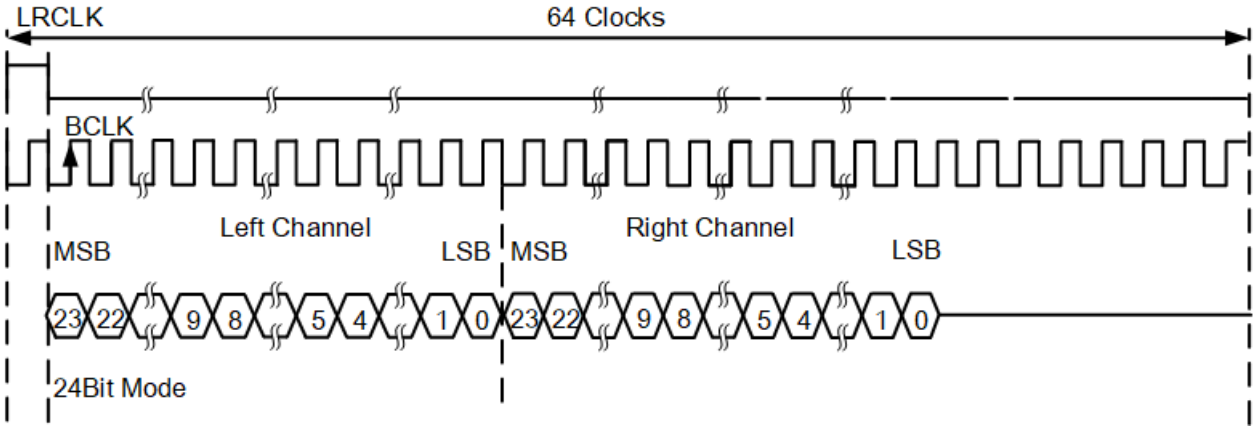


Figure 26.7: DSP format (Mode-B)

26.2 Audio 框架说明

26.2.1 CODEC 介绍

如下图所示，音频编解码器（CODEC）的模数转换器（ADC），CODEC 将 AMIC、Line-in 输入的模拟信号进行 A/D 转换，把模拟的信号转变 CPU 能够处理的数字信号；数模转换器（DAC），将 PCM 等信号进行 D/A 转换，把数字的音频信号转换为模拟信号。

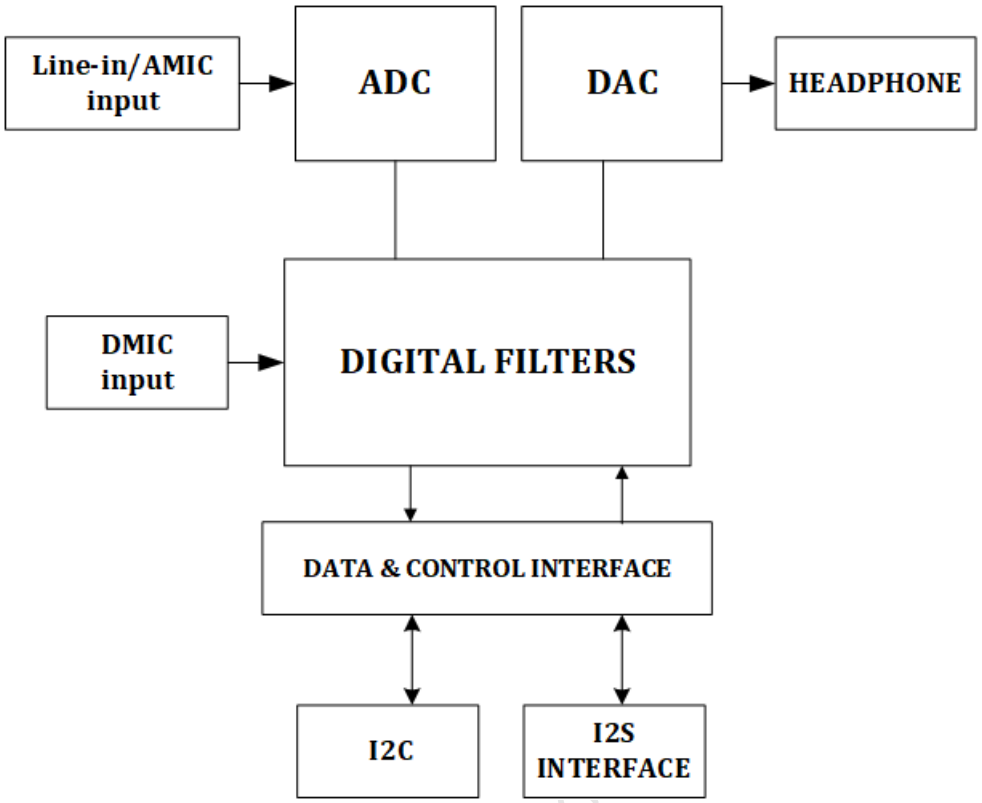


Figure 26.8: CODEC 框架

26.2.2 Audio **框架**

如下图所示，音频模块包括 3 个部分：数据输入输出接口（不同 SoC 可能会有差异，在 Input_Path 有 I2S RX 接口、音频编解码器（CODEC）ADC 接口，在 Output_Path 有 I2S TX 接口、CODEC DAC 接口）；FIFO 和 DMA 组成数据交互接口（蓝色框图）；储存 PCM 数据的 BUFF（橙色框图）。

- Input_Path：CODEC 将 A/D 转换后的数字信号传入 BUFF 或者通过 I2S RX 接口直接将数据直接传入 BUFF。
- Output_Path：将 BUFF 中的 PCM 等信号进行 D/A 转换，把数字的音频信号转换为模拟信号或者通过 I2S TX 直接输出。

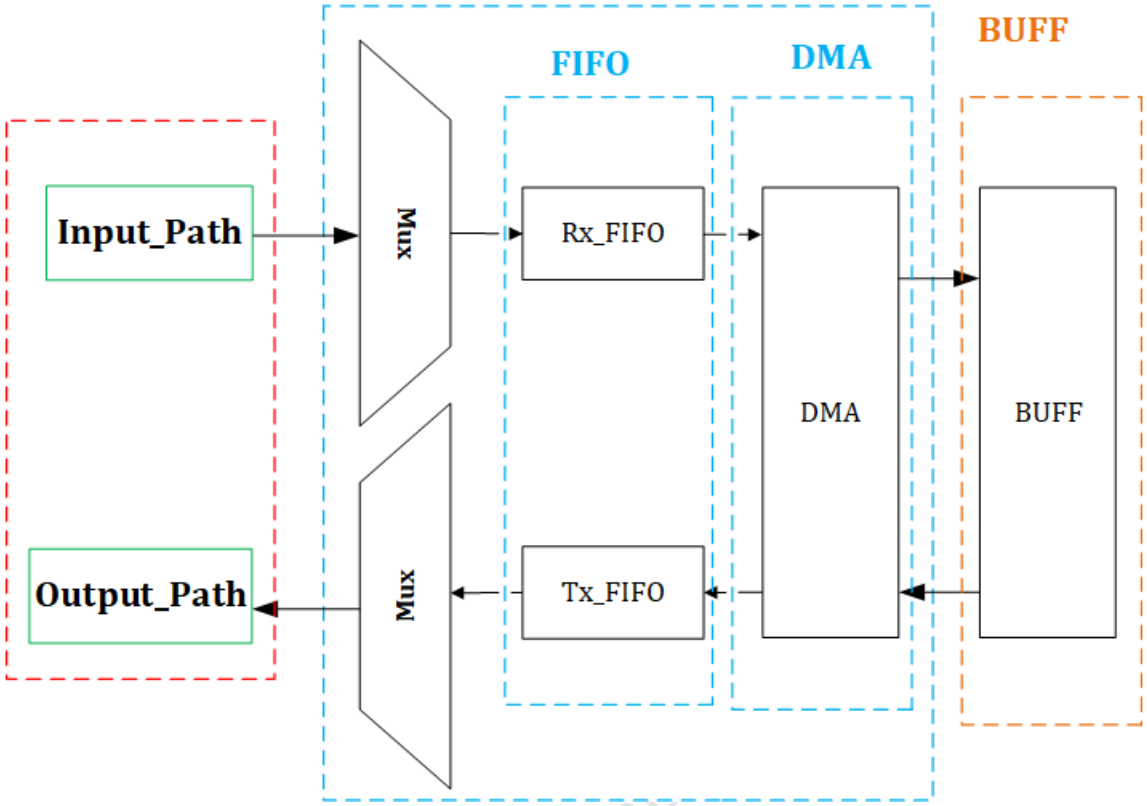


Figure 26.9: Audio 框架图

26.2.3 Audio I2S **时钟**

如下图所示，当 SoC 作为 master 为适应不同的采样率（LRCLK），支持 8K/16K/32K/48K/44.1K，需要设置不同的分频系数计算出相应的采样率，其中 Audio 模块时钟源的 MCLK 时钟和 I2S_CLK 都直接来自于 PLL=192M，I2S_CLK 再分出 BCLK 和 LRCLK。

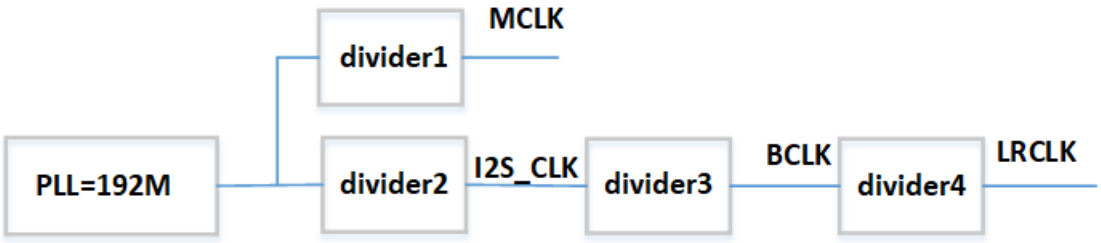


Figure 26.10: SoC audio clock source

其中，MCLK=PLL/diver1，LRCLK=PLL/divder2/divider3/divider4。

26.3 Audio 驱动说明

26.3.1 DMA 传输

Audio 数据传输是通过 DMA（Direct Memory Access）传输，介绍下 DMA 和 Audio BUFF 的工作机制。

26.3.1.1 DMA 传输

DMA 传输是将外设的数据不经过 CPU 直接送入内存储器，或者，从内存储器不经过 CPU 直接送往外部设备。传输动作本身是由 DMA 控制器来实行和完成，传输过程中不需要 CPU 的参与，如下图所示，Audio FIFO 与 BUFF 数据交互是通过 DMA 传输，录音（Rx）DMA 的源地址为 Rx_FIFO 的首地址，深度为 8 word，目的地址为 Rx_BUFF 的首地址，深度可配。放音（Tx）DMA 的目的地址为 Tx_FIFO 的首地址，深度为 8 word，源地址为 Tx_Buff 的首地址，深度可配。如果 Rx_BUFF 与 Tx_BUFF 公用一块 BUFF，那 Rx 的目的地址和 Tx 的源地址一致。

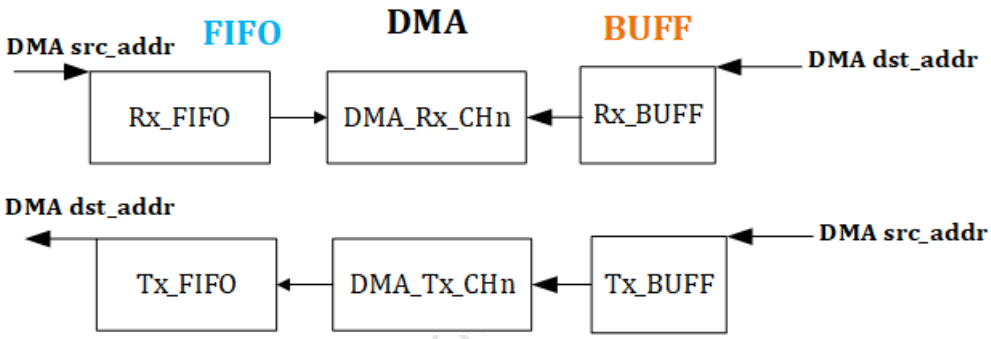


Figure 26.11: DMA 与 Audio FIFO 交互图

26.3.1.2 DMA 链传输

Audio 数据流是不间断产生，DMA 传输一次完成，再进行下一次传输需要配置相关 control 寄存器和 dma length 才能再次 trigger dma 传输。DMA 链表可以解决这一问题，使用链表的方法在不需要 CPU 参与下，完成连续传输的功能。DMA 链表包括以下内容：DMA control、src_addr、dst_addr、DMA length 和 LLP_ptr（下一个链表的地址），DMA 完成第一传输后，设置好链表头（Head_of_list），dma 会按照链表配置的内容进行传输，再链接到下一个链表进行传输，循环往复。

(1) DMA ping-pong buff 链传输

如下图所示，以 Tx 为例，ping-pong buff 的链式传输，先建立头结点 Head_of_list，然后向循环链表添加两个链表，分别是 Tx_dma_list[0] 和 Tx_dma_list[1]，Tx_dma_list[0] 对应的源地址为 Tx_BUFF[1]，Tx_dma_list[1] 对应的 Tx_BUFF[0]。按其流程先搬运 Tx_BUFF[0] 的数据，然后根据链表头 LLP 指向 Tx_dma_list[0]，搬运 Tx_BUFF[1] 的数据，Tx_dma_list[0] 中 LLP 指向 Tx_dma_list[1] 搬运 Tx_BUFF[0] 的数据，其 LLP 又指向了 Tx_dma_list[0]，收尾相接形成一个 ping-pong buff。

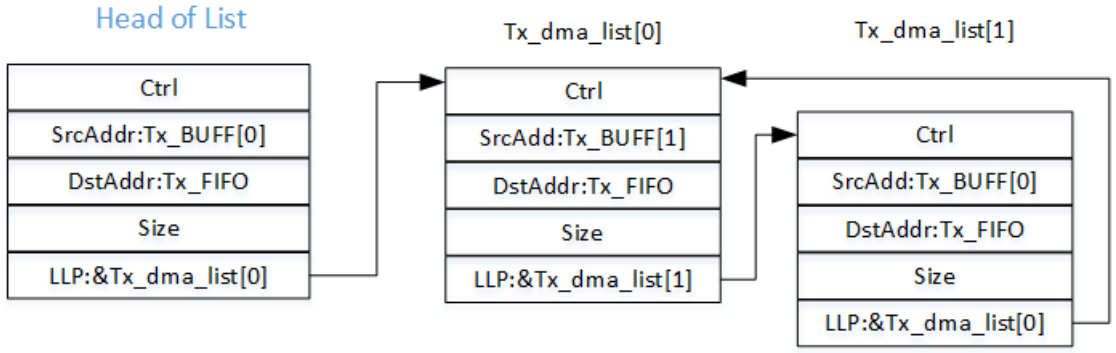


Figure 26.12: ping-pong BUFF 链式

(2) DMA 单个 buff 链传输

如下图所示，以 Tx 为例，在 ping-pong buff 的基础上，使 Tx_dma_list 的 LLP 指向自己，发 dma 就是一直搬运单个 Tx_BUFF 的数据。

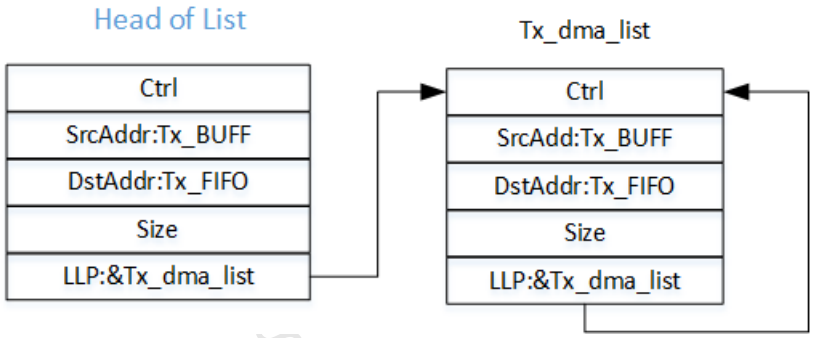


Figure 26.13: 单 BUFF 链式

26.3.2 Audio buff **工作机制**

Audio 使用的是环形 buff，录音为 Rx，放音为 Tx，数据通过 DMA 传输，Rx 的 DMA 的源地址为 audio 外设 FIFO 地址，目的地址为 RX_BUFF 首地址，Tx 的 DMA 的源地址为 Tx_BUFF，目的地址为 audio 外设 FIFO 地址。

26.3.2.1 Rx Path

如下图所示，设定 BUFF_size 长度的 Rx_BUFF , 录音数据由 DMA 搬入 Rx_BUFF，获取由硬件维护的 rx_wptr (不同芯片获取的方式有差异)。根据软件从 Rx_BUFF 获取数据操作，记录 rx_rptr，由软件维护。红色线条部分为可读数据长度 read_len 为：

- rx_wptr > rx_rptr, read_len = rx_wptr - rx_rptr;
- rx_wptr < rx_rptr, read_len = buff_size - (rx_rptr - rx_wptr)。

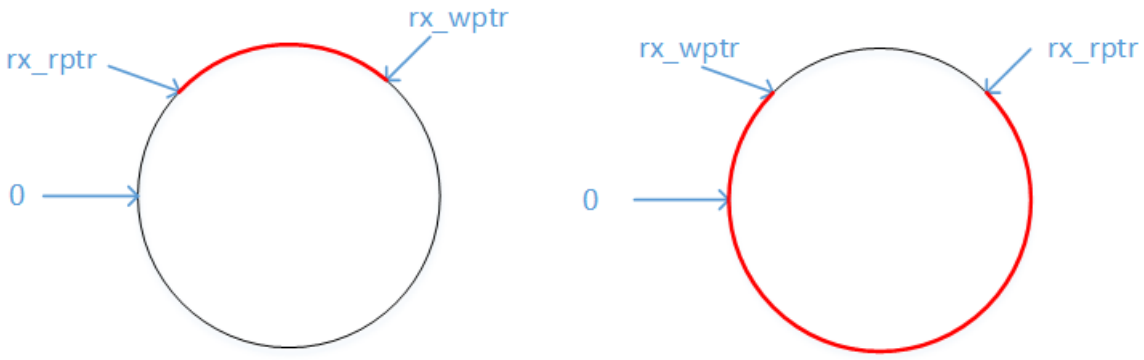


Figure 26.14: Rx_BUFF 示意图

26.3.2.2 Tx Path

如下图所示，设定 BUFF_size 长度的 Tx_BUFF，放音是将 BUFF 数据由 DMA 搬出 BUFF，获取由硬件维护的 tx_rptr（不同芯片获取的方式有差异）。根据软件往 Tx_BUFF 填数据操作，记录 tx_wptr，由软件维护。红色线条部分为可写数据长度 write_len:

- tx_rptr > tx_wptr, write_len = tx_rptr - tx_wptr;
- tx_rptr < tx_wptr, write_len = buff_size - (tx_wptr - tx_rptr)。

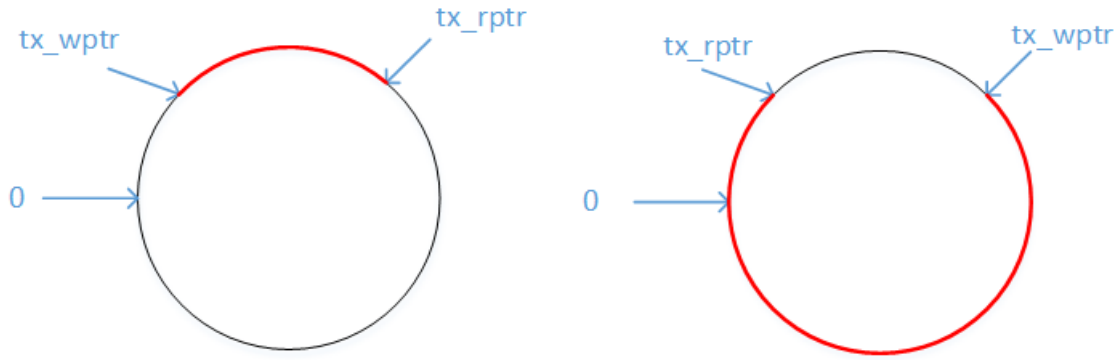


Figure 26.15: Tx_BUFF 示意图

26.3.3 Audio_Demo

在头 Audio Demo 里配置 app_config.h 中的宏 AUDIO_MODE 来选择不同的 AUDIO 模式。

Demo	功能
LINEIN_TO_LINEOUT	模拟音频在输入插孔输入，在输出插孔接扬声器或耳机可以实时听到经过 CODEC 经过处理的音频
AMIC_TO_LINEOUT	AMIC 录音，在输出插孔接扬声器或耳机可以实时听到经过 CODEC 处理的音频

Demo	功能
DMIC_TO_LINEOUT	DMIC 录音，在输出插孔接扬声器或耳机可以听到实时经过 CODEC 处理的音频
BUFFER_TO_LINEOUT	将 BUFF 里的 PCM 数据经过 CODEC 处理输出，在输出插孔接扬声器或耳机可以听到 BUFF 里的音频（一般为 1K 正弦波）
FLASH_TO_LINEOUT	将 FALSH 里的 PCM 数据，读取出来按一定方式填入 AUDIO BUFF，再经过 CODEC 处理输出，在输出插孔接扬声器或耳机可以听到 FLASH 里的音频
EXT_CODEC_LINEIN_LINEOUT	SoC 的 I2S 接口跟外部 CODEC（WM8731 为例）进行数据交互，外部 CODEC LINE_IN to LINE_OUT

264 芯片差异

264.1 Input Path 与 Output Path 差异

264.1.1 B91 Audio Input Path

参考Audio 框架图中的 Input_Path，如下图所示，audio 输入有 2 种方式：经内部 CODEC（没有特殊说明，CODEC 都指内部 CODEC）处理后的 I2S 信号（AMIC/DMIC/LINE-IN）；从外部 CODEC 输入的 I2S 信号。

I2S 输入：

通过 Mux 选择将 I2S 接口的 IO 连接到内部 CODEC 或者外部 CODEC，I2S 支持 I2S、LJ、RJ、DSP 格式；位宽支持 16bit、20bit、24bit 和 32bit 的数据格式，传输数据时 I2S_Rx 会将接收的串行 I2S 数据转为并行数据写进 Rx_BUFF。

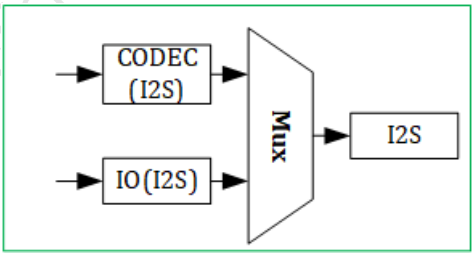


Figure 26.16: Audio input path

264.1.2 B91 Audio output Path

参考Audio 框架图中的 Output_Path，如下图所示，Audio 输出有 2 种方式：Tx_BUFF 里的音频数据，通过 I2S 接口输出到内部 CODEC 或者外部 CODEC。

I2S 输出：

Tx_BUFF 里的音频数据经过 I2S_Tx 转成的串行 I2S 格式的数据，输入给内部的 CODEC 或外部 CODEC。

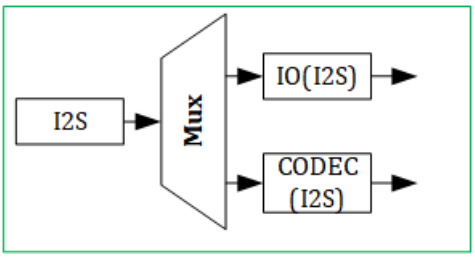


Figure 26.17: Audio output path

264.2 Audio Demo **差异**

264.2.1 LINEIN_TO_LINEOUT

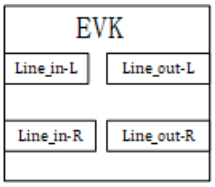


Figure 26.18: LINEIN_TO_LINEOUT

注意：

- ADC 输入支持单端和差分，默认为差分模式，DAC 只支持差分输出。
- MONO 模式下，可以选择单左声道输出或者左右声道同时输出，此时两声道的数据是一样的，默认是后者。
- STEREO 模式下，左路输入对应左路输出，右路输入对应右路输出。

264.2.2 AMIC_TO_LINEOUT

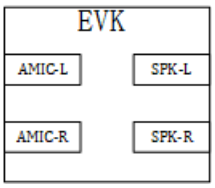


Figure 26.19: AMIC_TO_LINEOUT

注意：

- ADC 输入支持单端和差分，默认为差分模式，DAC 只支持差分输出。
- MONO 模式下，输入通道：默认是左声道的 AMIC 输入，可以调用 audio_set_mono_chn 接口设置为右声道 AMIC 输入。输出通道：可以选择单左声道输出或者左右声道同时输出，此时两声道的数据是一样的，默认是后者。
- STEREO 模式下，左路输入对应左路输出，右路输入对应右路输出。

264.2.3 DMIC_TO_LINEOUT

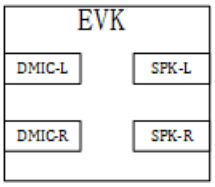


Figure 26.20: DMIC_TO_LINEOUT

注意：

- MONO 模式下，单 DMIC 只需要 2 根信号线 data 和 clk, clk 频率固定为 3M。
- STEREO 模式下，左路输入对应左路输出，右路输入对应右路输出，双 DMIC 有 1 根信号线 data 和 2 根 clk, 双 DMIC 共用 data，2 路 clk 时序是一样的，在 clk 上沿采集一 DMIC 数据，在 clk 下沿采集另一个 DMIC 数据。

264.24 BUFFER_TO_LINEOUT

Demo 里提供了频率为 1K 采样率为 44.1K，单声道音频数据和频率为 1K 采样率为 48K 单声道音频数据。

注意：

- 这里的 BUFF 的首地址就是 DMA 的源地址，将 BUFFER 作为 audio_buff。
- 无论在 MONO 还是在 STEREO 模式下，输出双路道输出会有一 sample 的相位差，调用 audio_invert_i2s_lr_clk 使 i2s clk 的取反，可以消除相位差。

264.2.5 EXT_CODEC_LINEIN_LINEOUT

该 demo 的功能实现：SoC 的 I2S 接口跟外部 CODEC（WM8731 为例）进行数据交互，外部 CODEC LINE_IN to LINE_OUT。

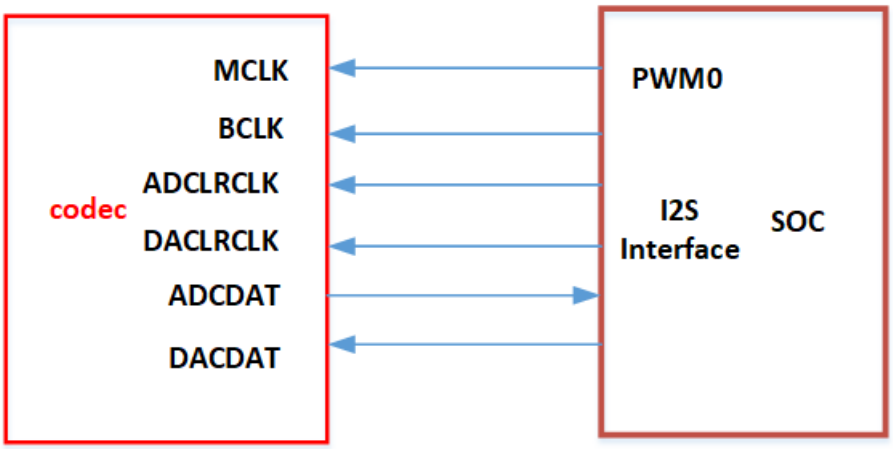


Figure 26.21: EXT_CODEC_LINEIN_LINEOUT

注意：

- 利用 PWM0 提供 MCLK(12M)。
- 默认采样率为 32K，MONO，BIT_16。

264.2.6 FLASH_TO_LINEOUT

该 demo 的功能实现：将 FALSH 里的 PCM 数据，读取出来按照定长填入 AUDIO_BUFF，再通过 LINE_OUT 输出，参考AUDIO BUFF 工作机制, BUFF 的长度为 4K（AUIDO_BUFF_SIZE），Flash 数据量很大，需要分批填入 BUFF 中。

B91 audio 模块没有效的中断去填充和获取音频数据：

- 录音数据由 DMA 以 word 单位搬入 Rx_BUFF，DMA 的目的地址每次自加 4，自加到 BUFF_size-4 后归 0，目的地址相对于 Rx_BUFF 首地址的偏移量记为 rx_wptr。
- 放音是将 BUFF 数据由 DMA 以 word 单位搬出 BUFF，DMA 源地址每次自加 4，自加到 BUFF_size-4 后归 0，源地址相对于 Tx_BUFF 首地址的偏移量记为 tx_rptr。

先介绍函数接口：

```
u32 audio_get_tx_dma_rptr (dma_chn_e chn) //参数 chn 是配置为 tx dma 通道，获取对应通道 dma 源地
↪ 址
```

初始化 DMA 的源地址配置为 audio_buff 的首地址，audio_buff 的大小为 4K, 放音是将 buff 数据由 dma 以 word 单位搬出 buff，每搬一次，dma 的源地址会加 4，自加到 buff_szie-4 后归 0，将 dma 的源地址减去 audio_buff 的首地址，可以表征硬件在 audio_buff 读的状态，记为读指针 tx_rptr：

```
tx_rptr= ((audio_get_tx_dma_rptr (DMA3)-(u32)auido_buff));
```

而 audio_buff_buff 的写状态，由软件维护，记为 tx_wptr，参考 AUDIO BUFF 工作机制中Tx Path的计算公式，audio_buff 的剩余可写空间：

```
if((tx_wptr&(AUIDO_BUFF_SIZE - 1))>tx_rptr)
    {
        unused_buff=AUIDO_BUFF_SIZE-(tx_wptr&(AUIDO_BUFF_SIZE-1))+tx_rptr;
    }
    else
    {
        unused_buff=tx_rptr-tx_wptr;
    }
```

例如按定长数据（AUIDO_THD_SIZE）填入 BUFF，其在 while（1）的流程如下。

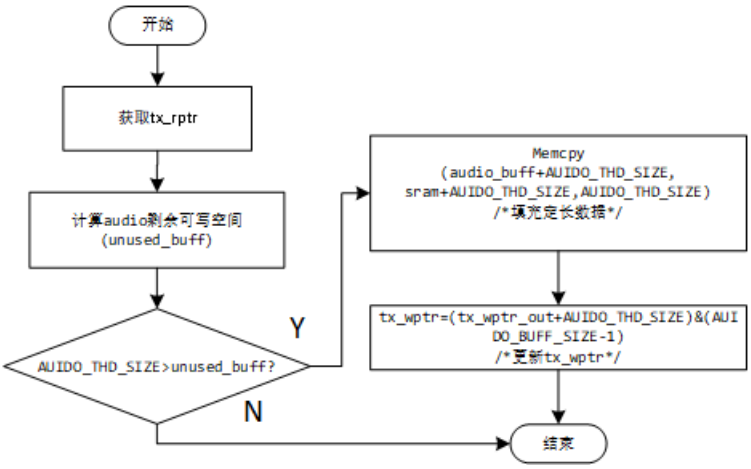


Figure 26.22: 流程

注意：

- ADC 输入支持单端和差分，默认为差分模式，DAC 只支持差分输出。
- MONO 模式下，可以选择单左声道输出或者左右声道同时输出，此时两声道的数据是一样的，默认是后者。
- STEREO 模式下，左路输入对应左路输出，右路输入对应右路输出。