玩转新塘M0/M4

32-bit M0/M4 NuMicro™ Family

Document Information

Abstract	初级篇从创建开发环境开始一步一步引导大家熟悉NuMicro家族芯片环境的搭建，BSP库的结构使用，以及每个IP的初始化流程。中级篇介绍CAN、USB以及ISO7816的使用方法。 技巧篇介绍各个IP的一些使用技巧
Apply to	NuMicro家族所有芯片：包括M051系列，NUC100系列，NUC200系列，NUC131系列，MINI51系列，NANO系列，NUC472/NUC442、M451等等一切NuMicro家族的芯片

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

Table of Contents

1 前言

一块板子拿在手里，大家最急切的就是该芯片都有什么功能？用什么编译？怎么编译？怎么下载？怎么调试？怎么烧录？ BSP结构怎样？BSP怎么使用？每个IP怎么初始化？别急，下面就一步一步为大家解开NuMicro家族的面纱。

1.1 概述

MCU的外设像：UART、SPI、I2C、USB、I2S等我们称之为IP。新唐M0和M4的所有芯片，为了省电，每个IP默认都没有时钟，寄存器都不能访问。所以每个IP要使用之前都需要选择时钟源并使能时钟，然后才能进行IP初始化。

时钟源一般有5个：

1) 内部高速振荡器 12M/16M/22.1184M（不同的芯片内部晶振频率不同）
2) 内部低速振荡器 10K/32K（不同的芯片内部晶振频率不同）
3) 外部高速 4~24M
4) 外部低速 32.768K
5) PLL

一般IP有上面5种时钟源可以选，有的IP不能选择只能用HCLK。

新唐M0和M4的所有芯片，目前全部内嵌Flash + SRAM，除了NUC505它是内嵌SPI Flash+SRAM的。

1.2 NuMicro 家族

先为大家介绍一下NuMicro家族，NuMicro家族的芯片内核目前有M0和M4。**下面列出一部分型号。**

核为M0的有：

型号	子型号
NUC100系列	NUC100/NUC120
M051系列	M052/M054/M058/M0516
MINI51系列	MINI51/MINI52/MINI54
NUC130CN/NUC140CN	
M0518	
MINI58	

M0519	
NANO系列	NANO100/NANO110/NANO120/NANO130
NANO102/NANO112	
NUC123	
NUC200系列	NUC200/NUC220/NUC230/NUC240

这些芯片的核都是Cortex-M0，CPU速度从24M ~ 72M，APROM4K～ 128K， RAM 2K ~ 20K。大都带I2C、SPI、UART、Timer、PWM、WDT、ADC、PDMA和RTC；有的还带USB、ISO7816、CAN、PS2D、I2S、DAC、ACMP等；PWM多的有20几个，UART多的有6个。总之外设都很丰富。

核为M4的有：

型号	子型号
NUC472	
NUC442	
M451/M451M	
NUC505	

CPU速度到100M，APROM到512M，SRAM到128M。外设除了上面M0拥有的，还有Ethernet、Image Capture、OP、QEI、DES/AES、SD卡以及USB OTG

2 初级篇

M0/M4的芯片内嵌的Flash一般分4块：LDROM、APROM、Dataflash、Config区域

- LDROM 一般用来放 ISP 代码，使用 UART 或者 USB 升级代码
- APROM 用来放用户程序
- Dataflash 用来放数据，有的 MCU 是单独的一块，有的通过 Config 区域从 APROM 中分
- Config 区域用于配置出厂设置，例如：用来配置出厂时从 LDROM 启动还是 APROM 启动、默认启动外部高速晶振还是内部高速晶振、是否使能 Dataflash 等等

2.1 环境搭建

NuMicro家族的BSP都支持Keil和IAR两种编译环境，下面的环境以Keil为例说明。安装过程分三步

- 安装 Keil MDK
- 安装 Nu-Link Keil 驱动
- 安装 BSP

  Nu-Link 是新塘出的 NuMicro 家族的调试、下载、量产工具，所有 NuMicro 家族的芯片通用

2.1.1 新唐官网

鉴于官网上东东太多，给大家介绍一下常用的M0/M4的东东去哪里找，进入www.nuvoton.com之后会看到下面的分类，M0和M4就分别点击下面两个链接进入

点击M0的话会进入M0的主画面，画面左边有下图的资源类

资源

应用手册 (2)
规格数据 (26)
发展工具 (36)
勘误表 (5)
示例代码 (1)
在线训练 (33)
产品简介 (17)
软件 (23)
技术参考手册 (25)
用户指南 (52)

规格数据就是各个系列的DataSheet，如果想快速了解该芯片的特性，可以下载这个文件

勘误表这个大家应该比较熟悉了，就是使用该芯片的一些注意事项和限制都在里面

软件就是各种软件下载，**包括Nu-Link-Keil驱动和BSP都在里面**

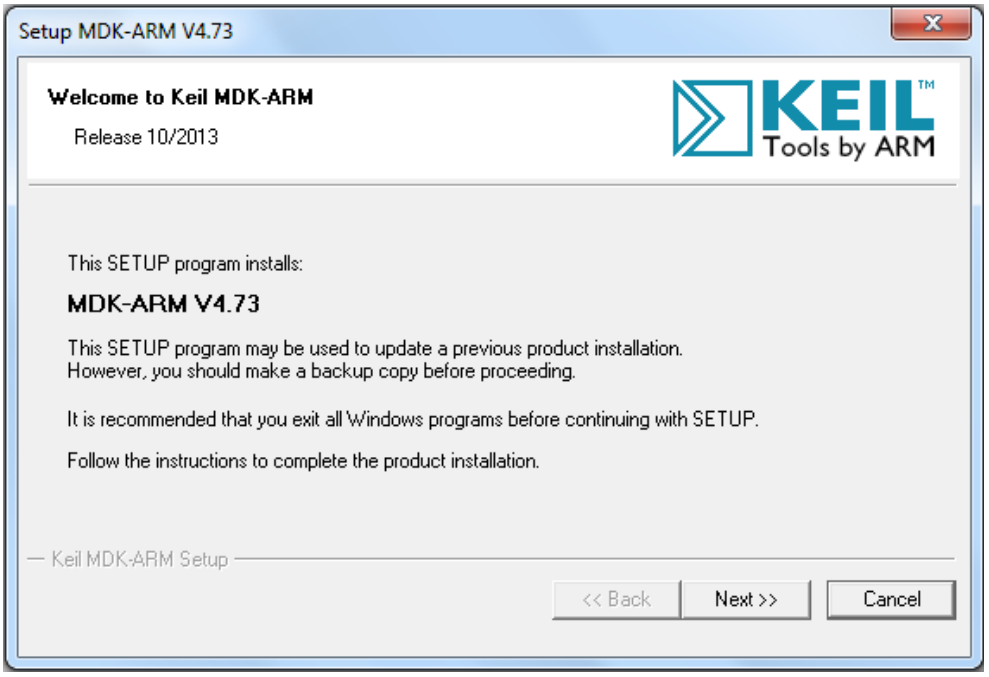
技术参考手册就是TRM，该文档里面有该芯片各个寄存器的详细介绍

2.1.2 安装 MDK

以Mini51为例说明MDK安装过程

如果大家有新唐的安装盘或者到www.nuvoton.com\NuMicroDVD下载电子档，点击autorun.exe，就会出现下面的画面

如果没有安装盘也没有关系，请到keil网站www.keil.com下载MDK，然后安装

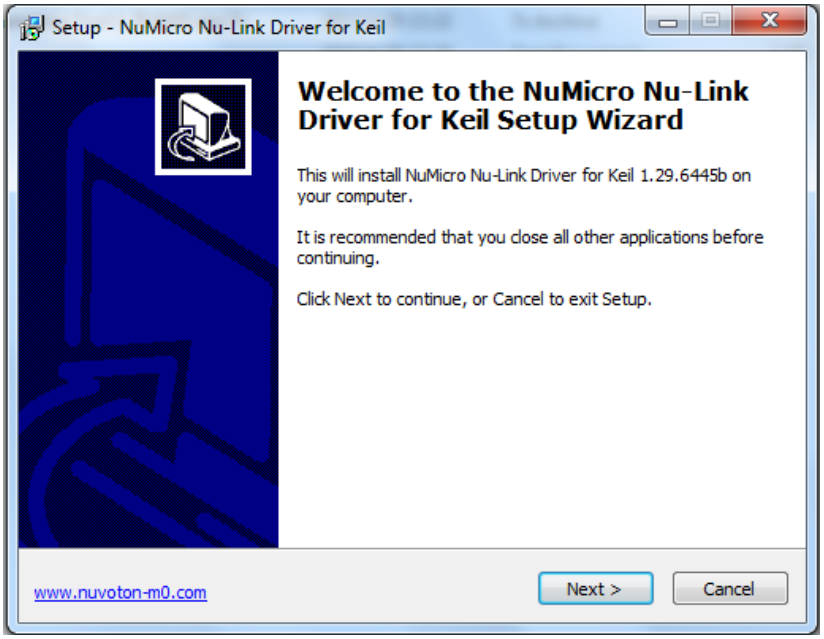


之后一路Next就可以了，keil默认安装在C:\keil下，注意keil安装目录奥，等会Nu-Link-Keil驱动安装的时候要用。

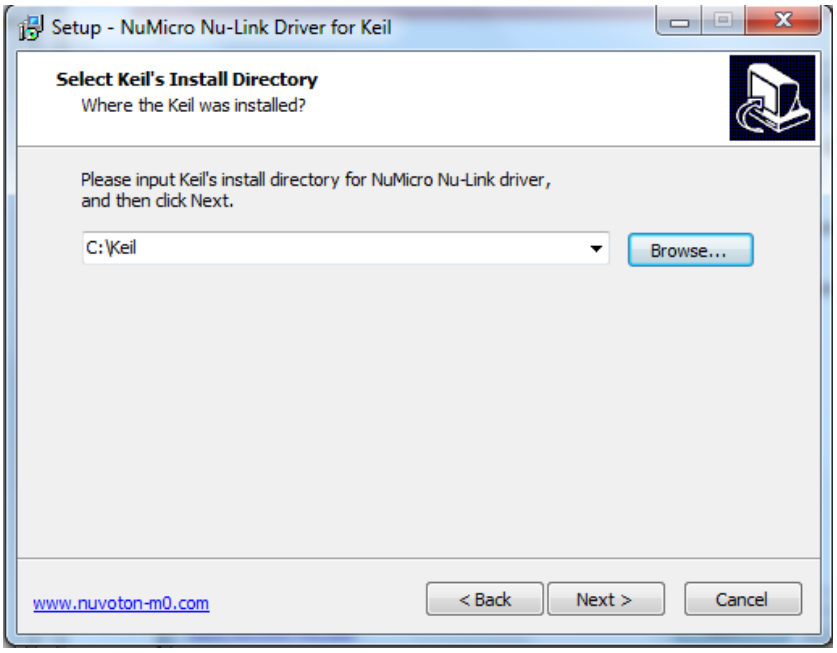
2.1.3 安装 Nu-Link Keil 驱动

到www.nuvoton.com下载Nu-Link_Keil_Driver_x.xx.xxxx 和NuMicro_ICP_Programming_Tool_x.xx.xxxx

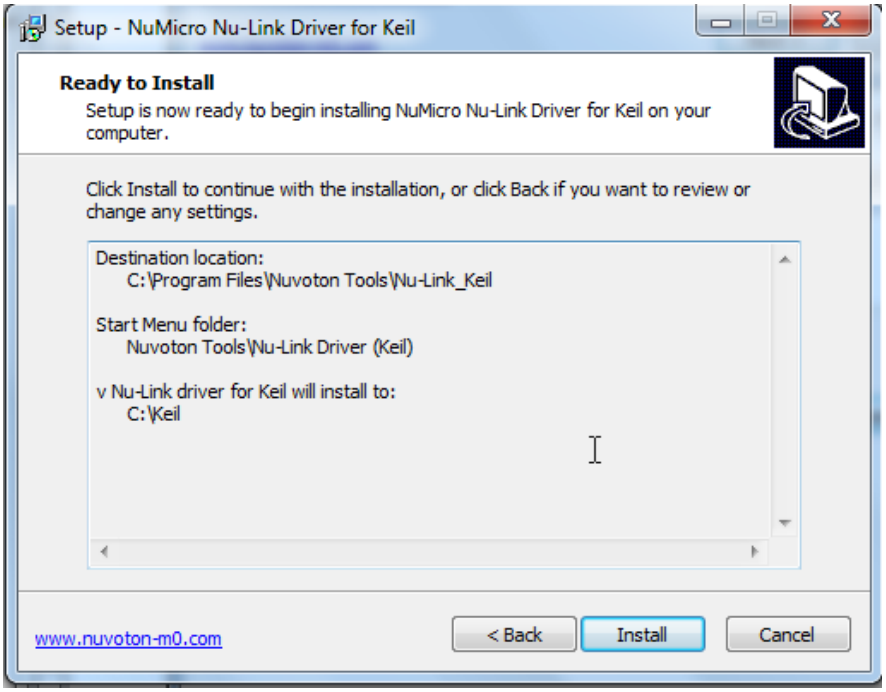
解压之后，双击就可以安装。ICP是windows软件，不用keil的时候可以用这个tool下载



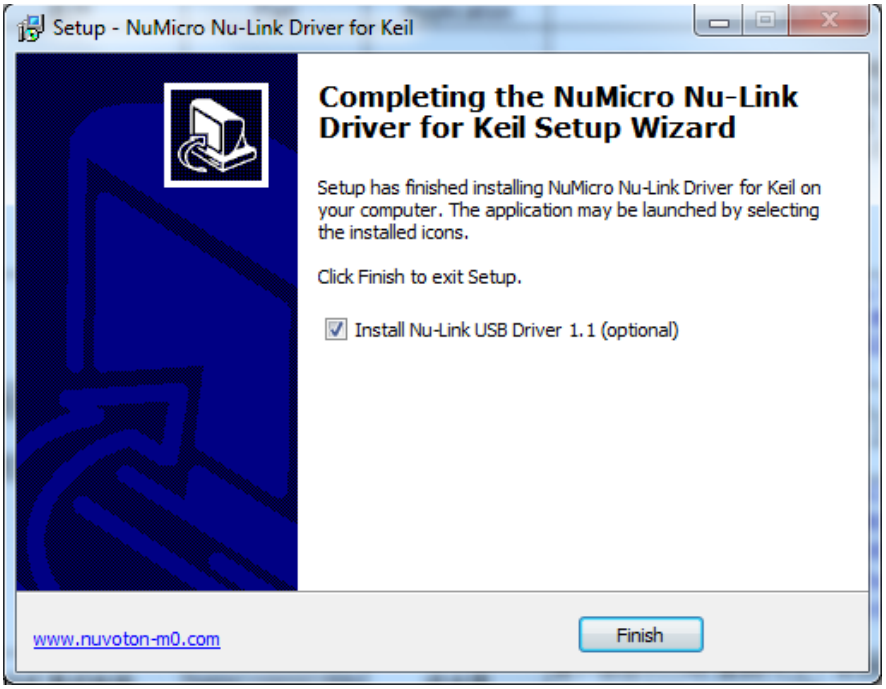
Next几次之后会出现选择keil安装目录的画面，这个一定要选对，不然keil找不到nu-link驱动



之后到达下面的画面，点击install，安装就开始了



下面这个画面不要忘记将Install Nu-Link USB Driver 1.1勾上，不然有些系统通过Nu-Link下载很慢



2.1.4 安装 BSP

到[www.nuvoton.com下载对应芯片的BSP]

M0518_Series_BSP_CMSIS_V3.00.002

M051SeriesBSP_DirectRegisterAccess_SC_v1.01.001

M051SeriesBSP_DirectRegisterAccess_v1.01.003

M051_Series_BSP_CMSIS_Rev3.00.002

M058S_Application_8x8x8LEDcube_BSP_V1.0

M058S_BSP_v3.00.002

Mini51DESeriesBSP_CMSIS_v3.00.002

NANO102_112_SeriesBSP_CMSIS_V3.01.000

NUC029FAESeriesBSP_CMSIS_V3.00.000

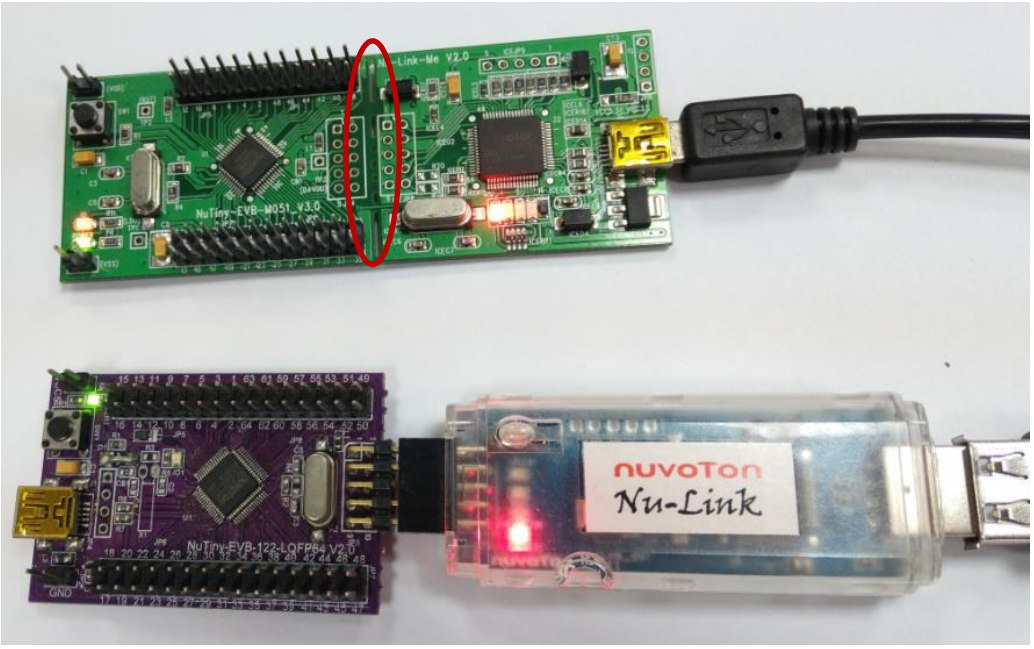
NUC029xANSeriesBSP_CMSIS_V3.00.001

每个系列都有自己的BSP，里面定义的API各个系列兼容。但是宏定义有点差别。BSP下载下来之后是一个ZIP压缩文件，解压到你自己的目录下就行了。OK！到此环境搭建完毕，下面介绍一下Nu-Link和ICP tool的使用方法。

2.2 Nu-Link 介绍

新唐所有MCU调试、下载都用一样的工具：Nu-Link和Nu-Link_Me，图片如下。M0/M4使用同一份FW，8051的话需要更新成调试8051的FW。

这两个设备功能基本一样，Nu-Link_Me基本上在新唐出的板子上都有带，锯下来之后可以用来调试任何M0/M4的芯片。这两个设备之间最大的区别就是Nu-Link上有一颗SPI flash所以支持off-line（离线）下载，这在量产上比较好用。可以将bin档通过ICP tool提前烧录到SPI flash中，还可以设定烧录次数，以及加密SPI flash中的bin档；而Nu-Lini_Me就没有off-line下载功能了，但是它们调试、下载功能都是一样的

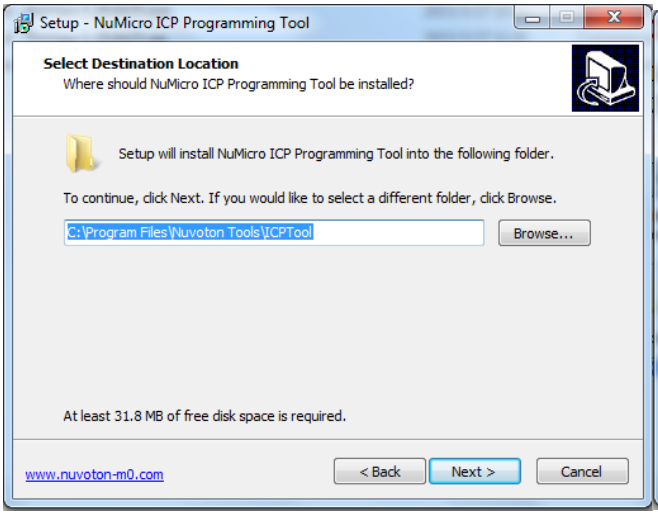


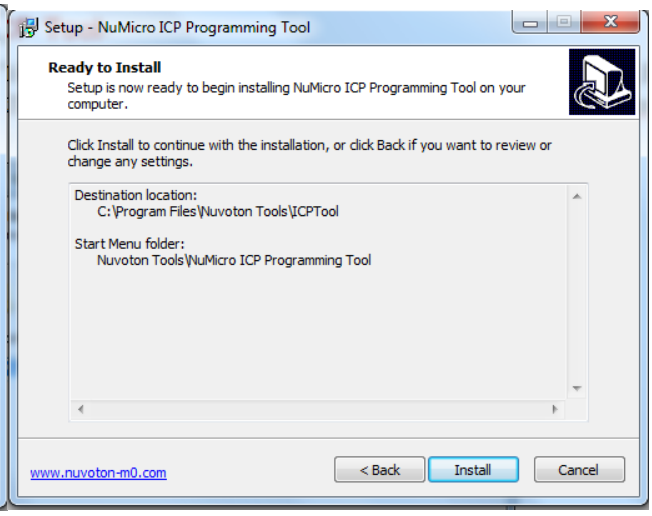
- 绿色的叫 Tiny 板，左边是目标板，右边是 Nu-Link_Me，Nu-Link_Me 通过 USB 接到 PC 上。Nu-Link_Me 可以从红色框的地方锯下来，拿去调试其他的 M0/M4 芯片
- 紫色的板子是目标板，它的 Nu-Link_Me 已经被掰掉了，可以拿 Nu-Link_Me 接，也可以拿 Nu-Link 接。图中接的是 Nu-Link，然后通过 USB 接到 PC 上

2.3 ICP Tool

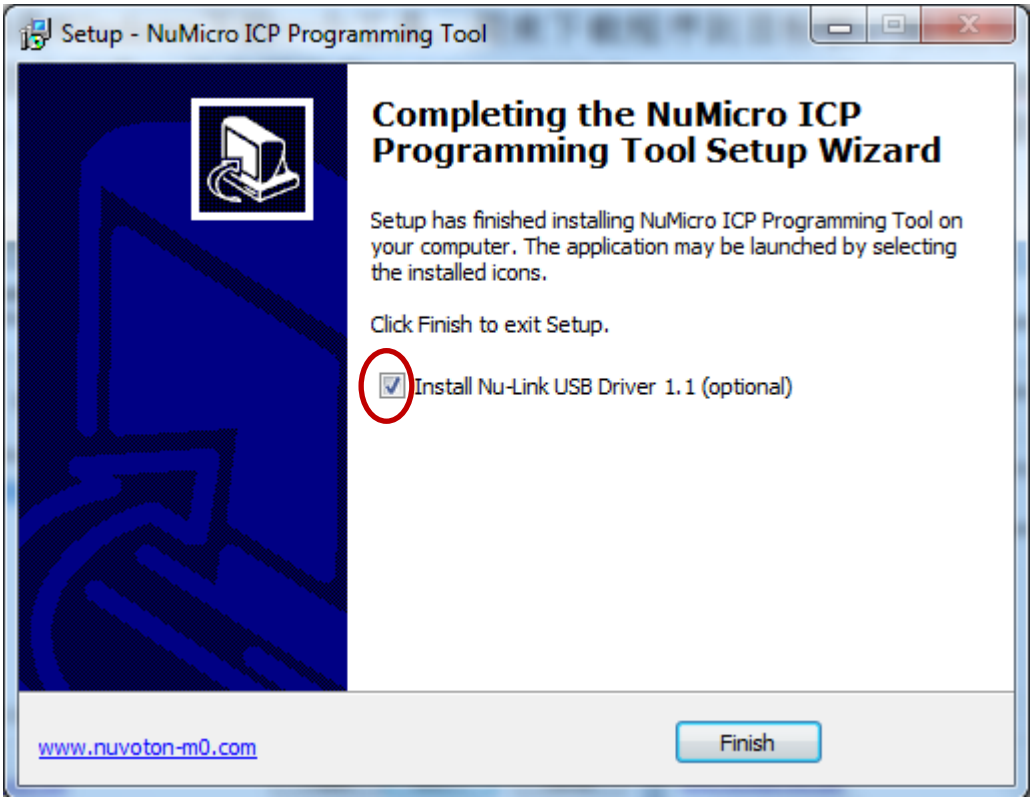
2.3.1 安装

ICP tool是Windows下的一个工具，用来下载程序到目标板。如果大家不想用Keil下载程序，可以选择这个工具。从官网下载NuMicro ICP Programming Tool x.xx.xxxx ，解压之后双击就会开始安装。然后一路next下去，就会到左边的画面，默认就好。然后到右边的画面开始安装，大概要花1min时间





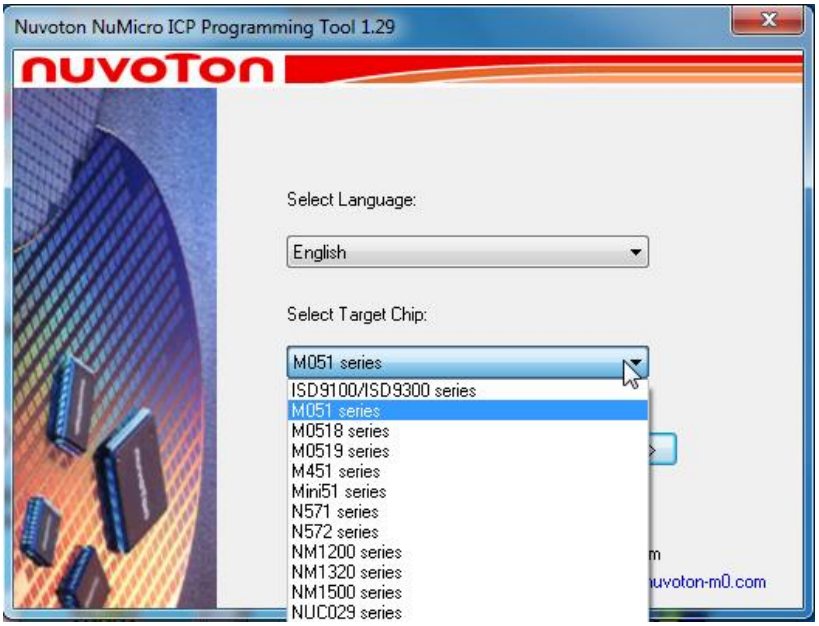
到下面的画面时，如果安装Nu-Link Keil驱动时已经安装了USB驱动，就不要打勾了。



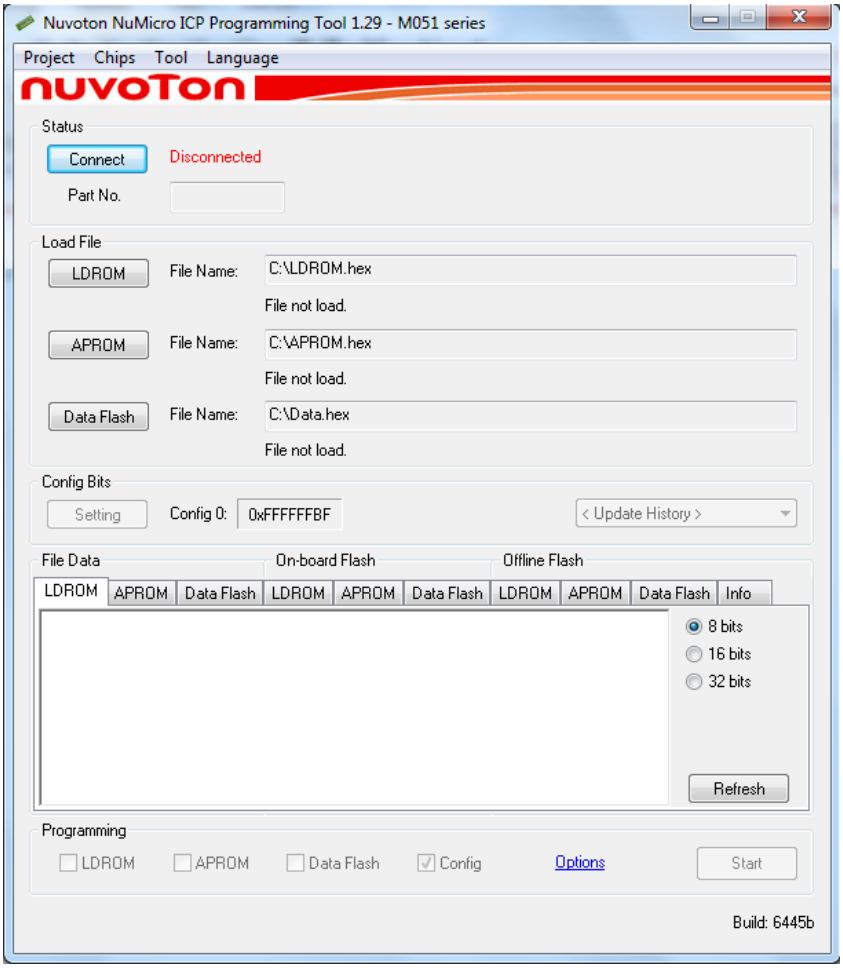
点击Finish之后安装完成

2.3.2 使用

打开ICP Tool之后画面如下，下拉框选择正确的芯片



之后点击”Continue”画面如下。将Nu-Link 一头通过USB接到PC上，另一头接到目标板上。然后点击”Connect” ， ICP Tool就会连接到目标板上，之后就可以下载程序/数据到LDROM、APROM和Data Flash中了。



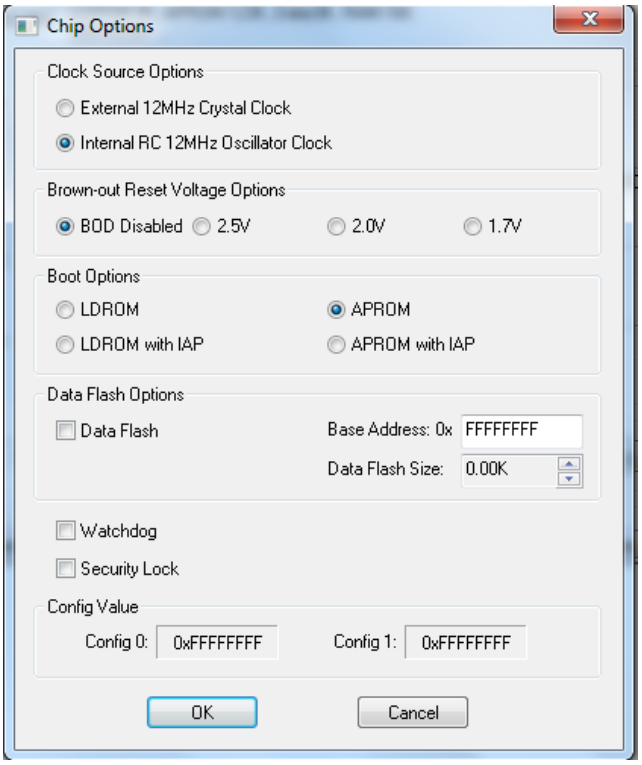
点击Connect后，画面如下，如果想选其它芯片连接，可以点击菜单中Chips改变

该工具分为6个区域：

① 是芯片信息区域。显示芯片的型号、LDROM/APROM/RAM 的大小、UID/UCID 的值。还有 Nu-Link 的 ID 号

② 是选择下载文件区域。选择要下载到 LDROM/APROM/DataFlash 中的文件。SPROM 是比较小的一块 Flash，一般 512B 大小，用于放一些需要保密的关键函数，或者数据。

③ 是用户配置区。就是 Config Area。芯片中该区域一般有几个寄存器，用于用户产品出厂时配置出厂设定。例如：上电从 LDROM 还是 APROM 启动，是否使能 BOD，是否使能 DataFlash 等。下拉框可以选择显示当前目标板的设定，还是上次烧录的数据。点击”Setting”画面如下（个各系列的芯片显示的内容稍有差异）：



由上图可以看到 NANO130 Config area 有 2 个寄存器 Config0 和 Config1

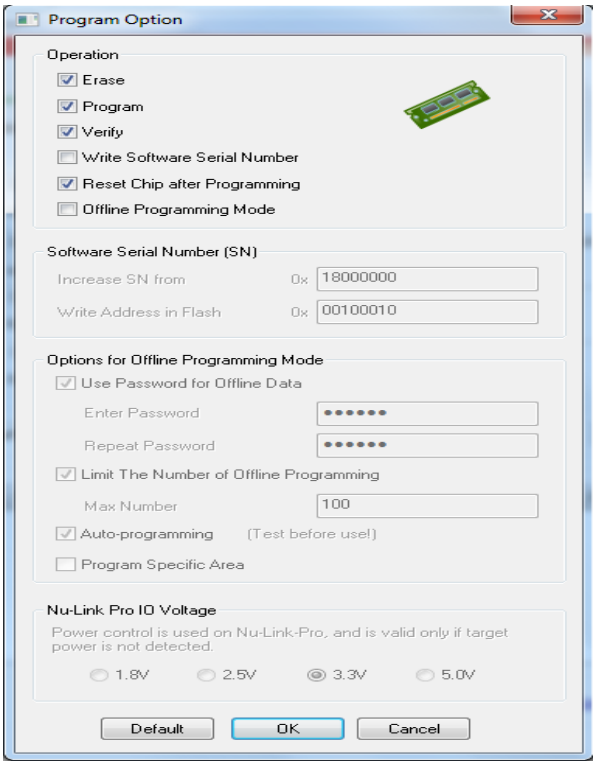
注意：Config area 修改之后需要复位才能起做用。如果不用 ICP tool 修改 Config area，而用软件修改，修改之后需要软件发出 CHIP reset 才能起做奥！

④ 是文件数据。包括 3 块：

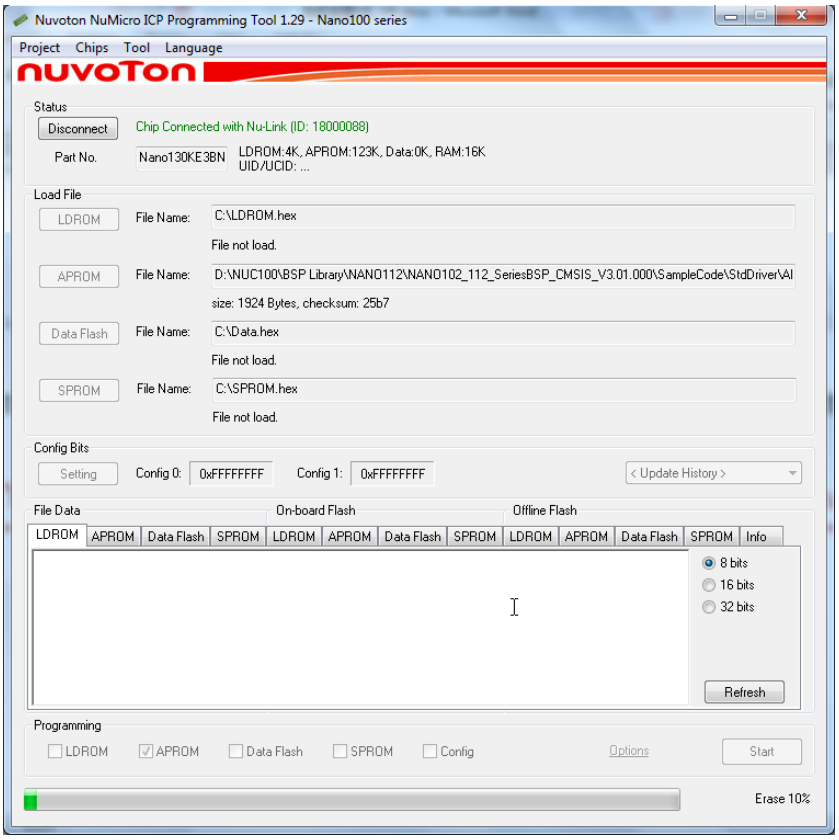
1) 当前选择的要烧录到 LDROM/APROM/DataFlash/SPROM 中的文件的数据
2) 目标板上 LDROM/APROM/DataFlash/SPROM 中的数据
3) 离线下载 SPI Flash 中保存的要烧录到 LDROM/APROM/DataFlash/SPROM 中的数据

⑤ 选择要编程的区域。包括 LDROM、APROM、DataFlash、SPROM 和配置区域

⑥ 点击 Options 画面如下，如果要进行离线烧录，就选择 Offline Programming Mode，然后程序会烧录到 Nu-Link 中的 SPI flash 里面



点击”Start”开始烧录，画面如下。它会根据 Options 里面的选择，进行 Erase/Program/Verify 的动作

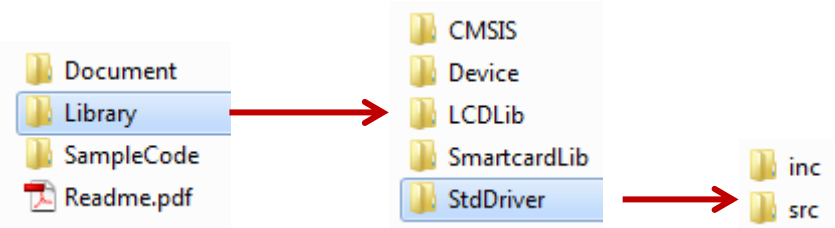


OK! 到此环境搭建完毕，也会使用ICP tool下载，下面进入BSP看看都有些什么东东吧！

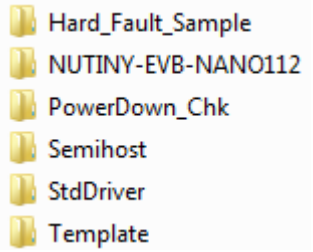
2.4 BSP 库的结构以及使用

2.4.1 库的结构

假如下载了这个BSP：NANO102_112_SeriesBSP_CMSIS_V3.01.000，打开，目录如下左边第一个图，Library目录打开如中间图示，StdDriver打开目录如右边图示：



SampleCode打开目录如下



各个目录内容如下：

- ✧ Document\目录里面是 Driver API 和一些结构体的说明 chm help 文件
- ✧ Library\目录就是芯片每个外设的驱动 API 源码，展开之后有 5 个目录
  - ■ CMSIS\目录是 ARM 写的一些 code，包括 M0/M4 核中 ARM 做的一些外设：Systick 和 NVIC。还有一些函数
  - ■ Device\目录是 ARM 定义的一些接口，由原厂实现。包括所有寄存器的定义都在这个目录下的 Nano1x2Series.h 里面，其它芯片.h 文件的名字不同
  - ■ StdDriver 目录是芯片所有外设的 API 源码和宏定义都在这里，该目录打开包含 2 个目录
    - ◆ Inc\目录。头文件在这里定义，每个外设有一个自己的头文件，如下图 2-1
    - ◆ Src\目录。API 源码都定义在这里
  - ■ LCDLib\因为 NANO112 有段式屏的驱动，所以新唐的一些开发板上有带段式屏，这里定义了一些在屏上显示字符的 API

- SmartcardLib\目录。因为 NANO112 支持 ISO7816 接口，初始化卡、读、写卡的函数都在这里

✧ SampleCode\目录。顾名思义就是所有的 demo code 都在这里。该目录下有 6 个子目录

- Hard_Fault_Sample\是 Hardfault 中断处理函数的范例
- NUTINY-EVB-NANO112\是 NANO112EVB 板子上跑的一些 sample code，包括 power down 的一些范例都在这里
- PowerDown_Chk\进行 power down 时一些设定的检查，会列出哪个 IO 没有设为 GPIO 功能，晶振引脚没有设为晶振，某个脚为低电平提示不能打开 PULL high 电阻，等等一些提示信息
- Semihost\是 keil 下使用 semihost 功能打印字符到 keil 窗口的范例
- StdDriver\：所有 IP 的常用范例都在该目录下，包括 UART、Timer、SPI 等。我截取了一部分，如下图 2-2
- Template\：模板，新建一个 project 比较麻烦，要设定使用的库，头文件搜索目录等等。如果 StdDriver 下的范例中的*.uvproj 文件都不想用，可以用这个

下图是inc目录下的文档

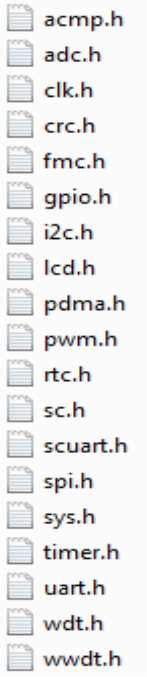


图 2-1 inc目录下的文件

下图是StdDriver目录

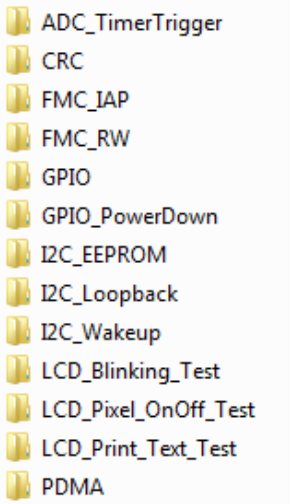


图 2-2 StdDriver目录下的文件

2.4.2 初识 Sample code

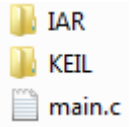
下面我们通过讲解一些sample code来更进一步了解BSP。

常用Sample code都在目录NANO102_112_SeriesBSP_CMSIS_V3.01.000\SampleCode**StdDriver**下面。

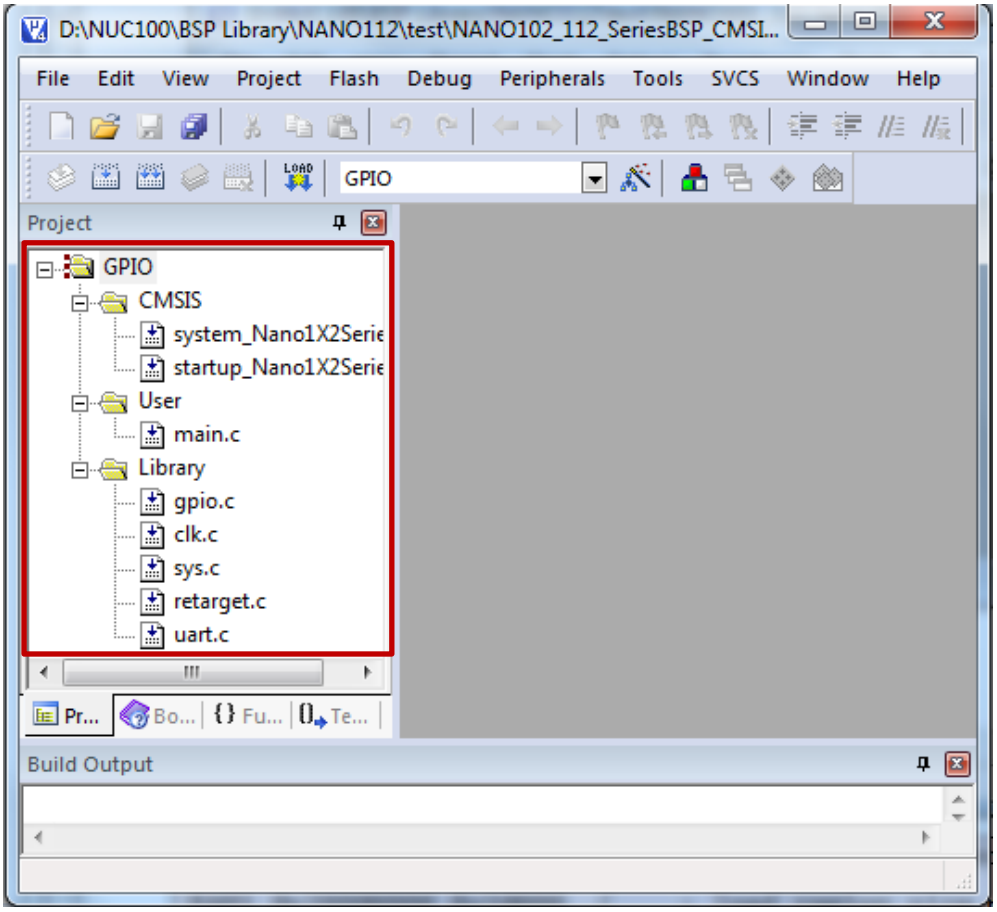
通过讲解GPIO范例，大家可以了解API的使用方法以及通过Nu-Link下载、调试的方法。因为使用UART打印信息，所以UART也会被初始化。

2.4.3 GPIO 范例

打开GPIO目录，下面有2个目录和一个文件



1) 进入 keil 目录，双击项目文件 GPIO.uvproj，keil 就会被打开，如下图

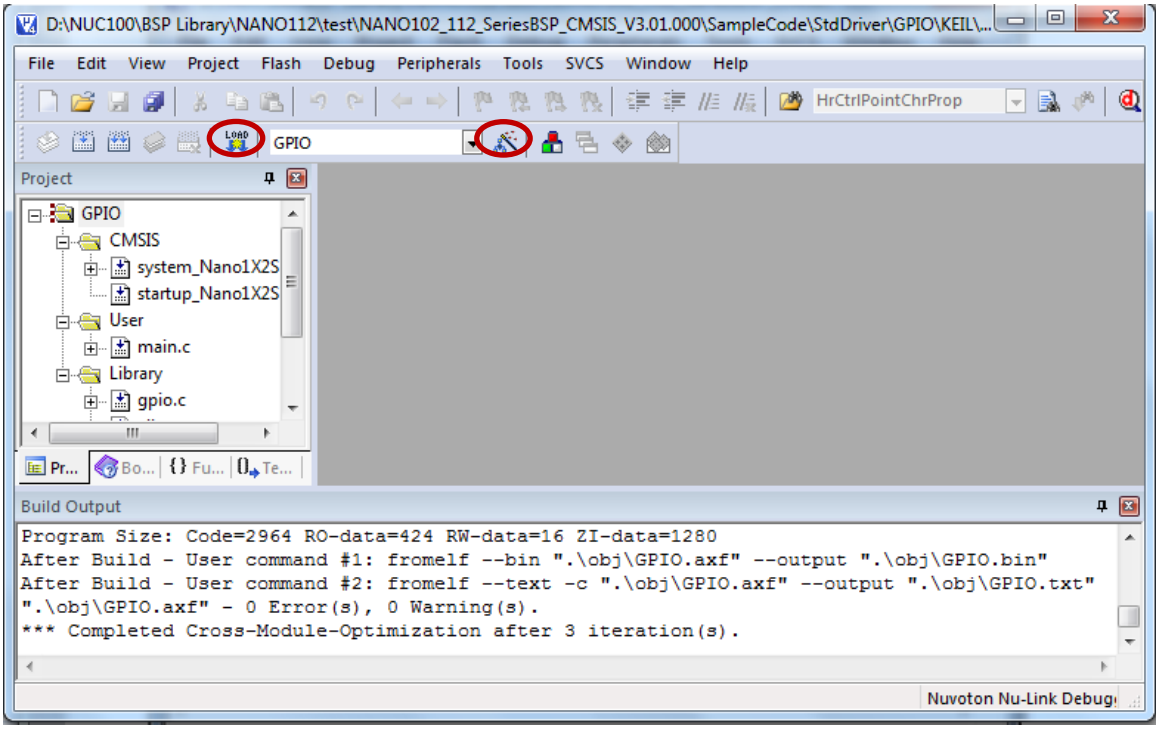


上图中，CMSIS下是ARM CMSIS规范规定的2个文件

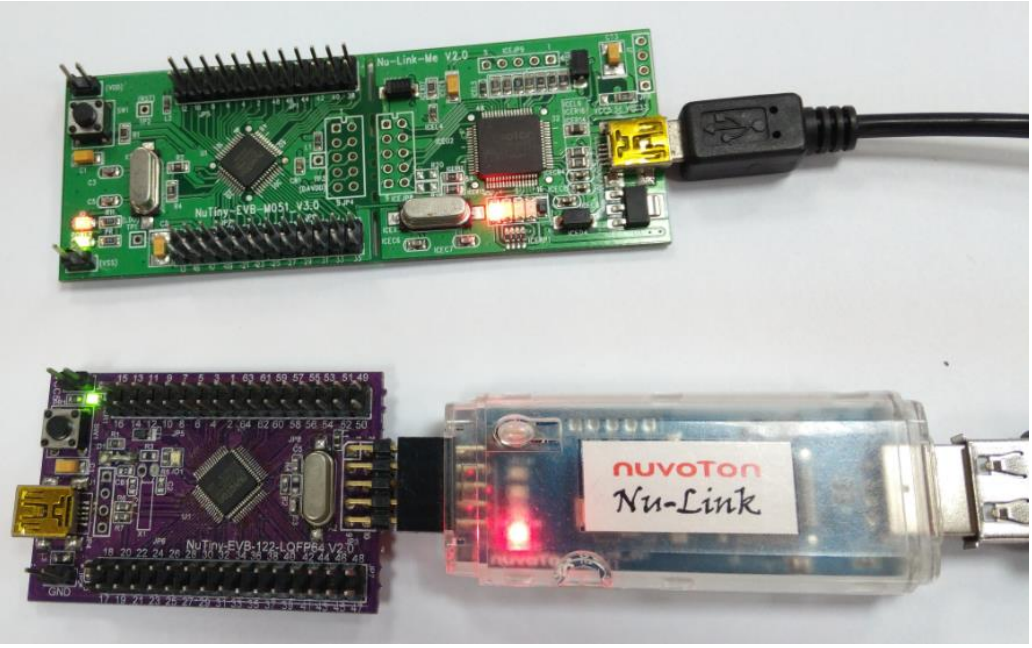
User下是main.c

Library下是Library\StdDriver\src目录下的文件，项目用到的就包在这个目录下

2) 点击 F7 进行编译，keil Build Output 窗口会显示编译结果，GPIO.bin 生成，在 obj 目录下

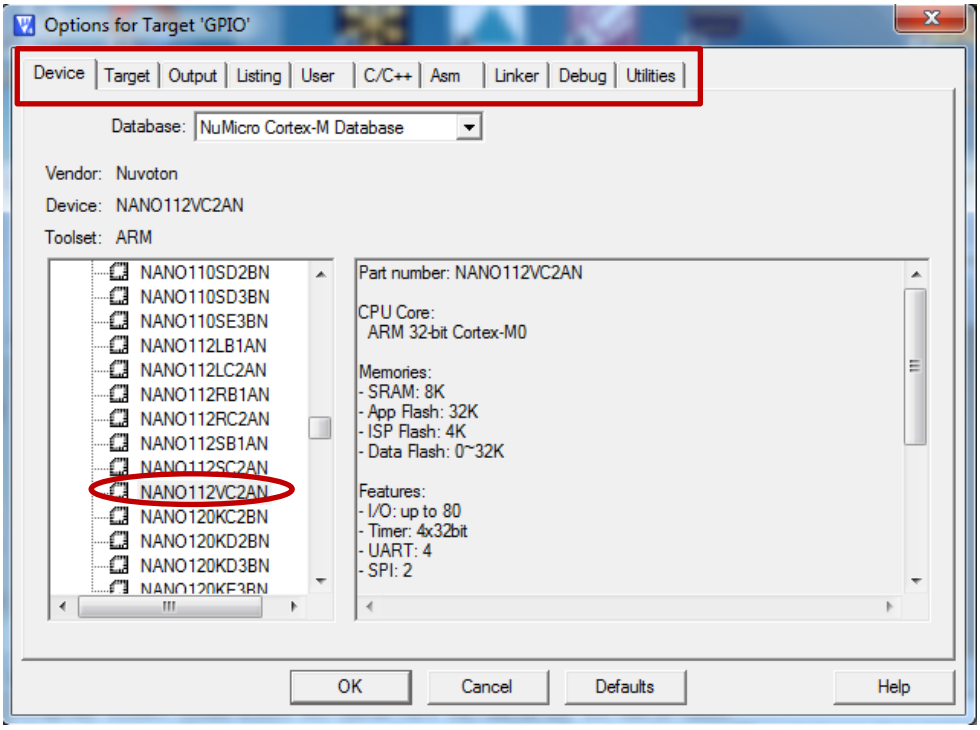


3) 将板子通过 Nu-Link 或者 Nu-Link-Me 接到 PC 上

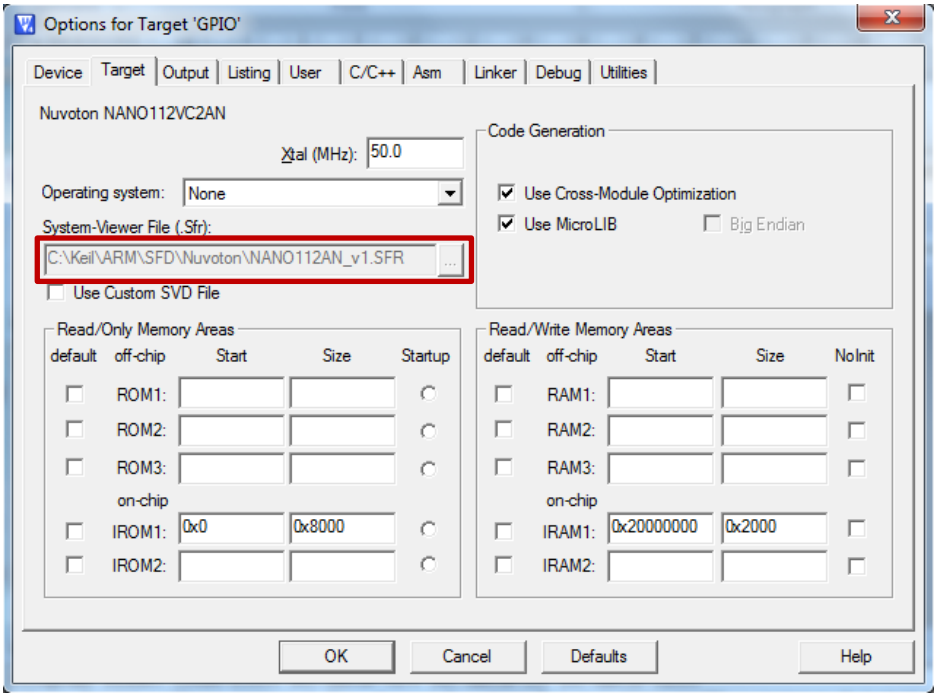


4) 点击 Load 进行下载。下载之前让我们看一下怎么选择用 Nu-Link 进行下载

点击Target Option进入下面的画面，大家注意该画面东西特多，瞪大眼睛，注意看，该窗口有10个frame

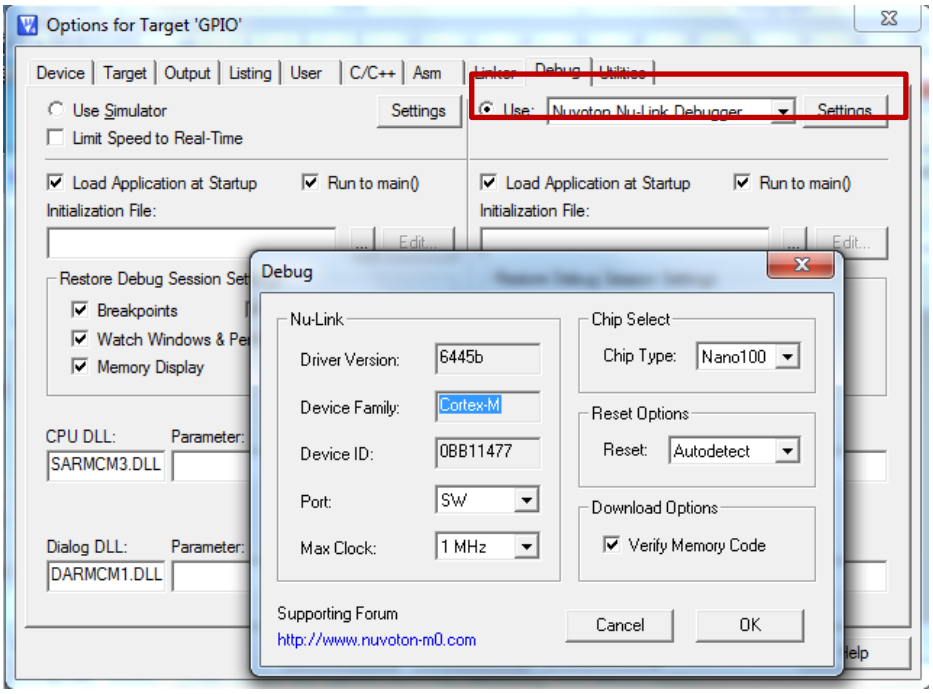


✧ 首先在 Device 窗口选择正确的型号，例如我们选择 NANO112VC2AN。这样等会 Debug 的时候才能看到寄存器列表：注意 Target 里面下图红框内不能是空的，否则 Debug 的时候看不到寄存器列表

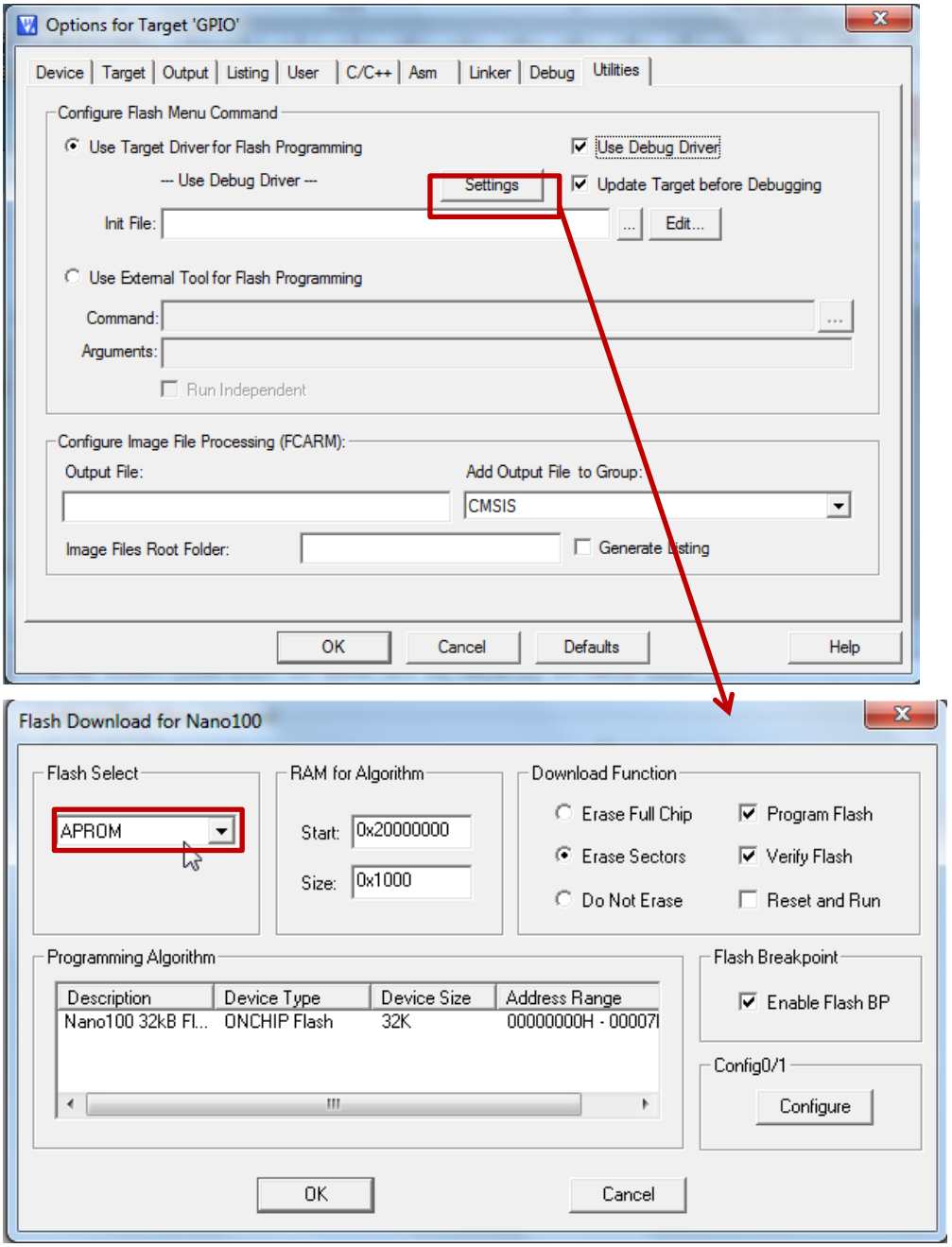


✧ 然后进入 Debug 选择 Nuvoton Nu-Link Debugger。点击 Settings 会看到当前使用的 Nu-Link Keil 驱动版本号

如果看不到该项选择，说明Nu-Link-Keil-Driver安装的目录不对或者Nu-Link没有被识别，目录不对需要选择正确的目录重新安装，注意要安装到Keil的安装目录下；Nu-Link没有被识别的话，如果USB驱动安装都没有问题，就只能是驱动有问题或者你的windows系统里面缺少安装包，可以联系新唐技术支持。

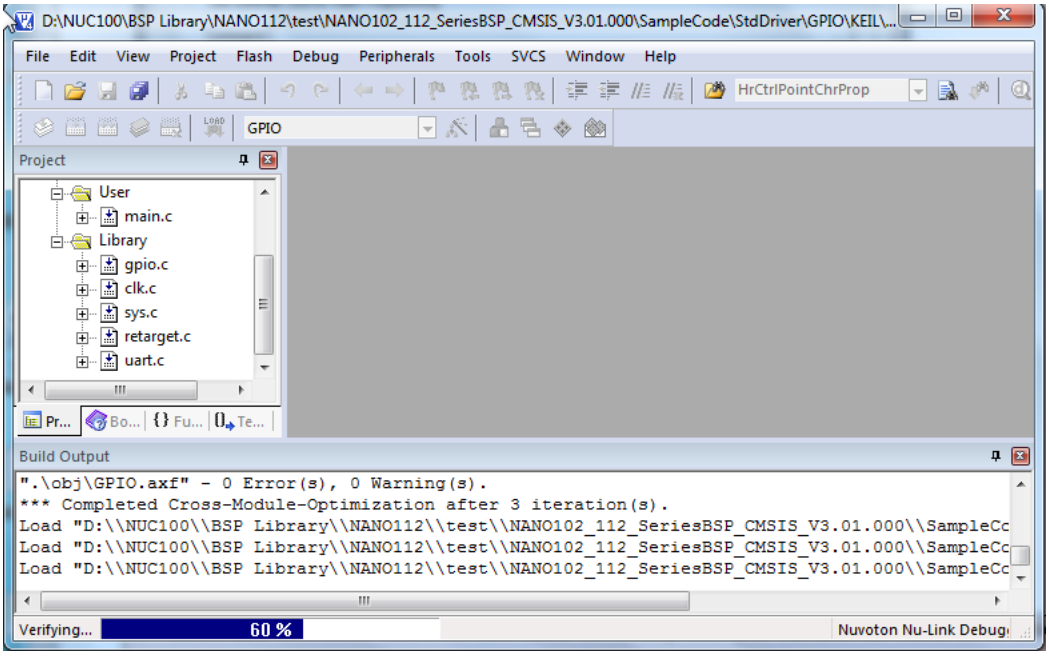


✧ 最后进入 Utilities，打勾 Usb Debug Driver 和 Update Target before Debugging。点击 Settings

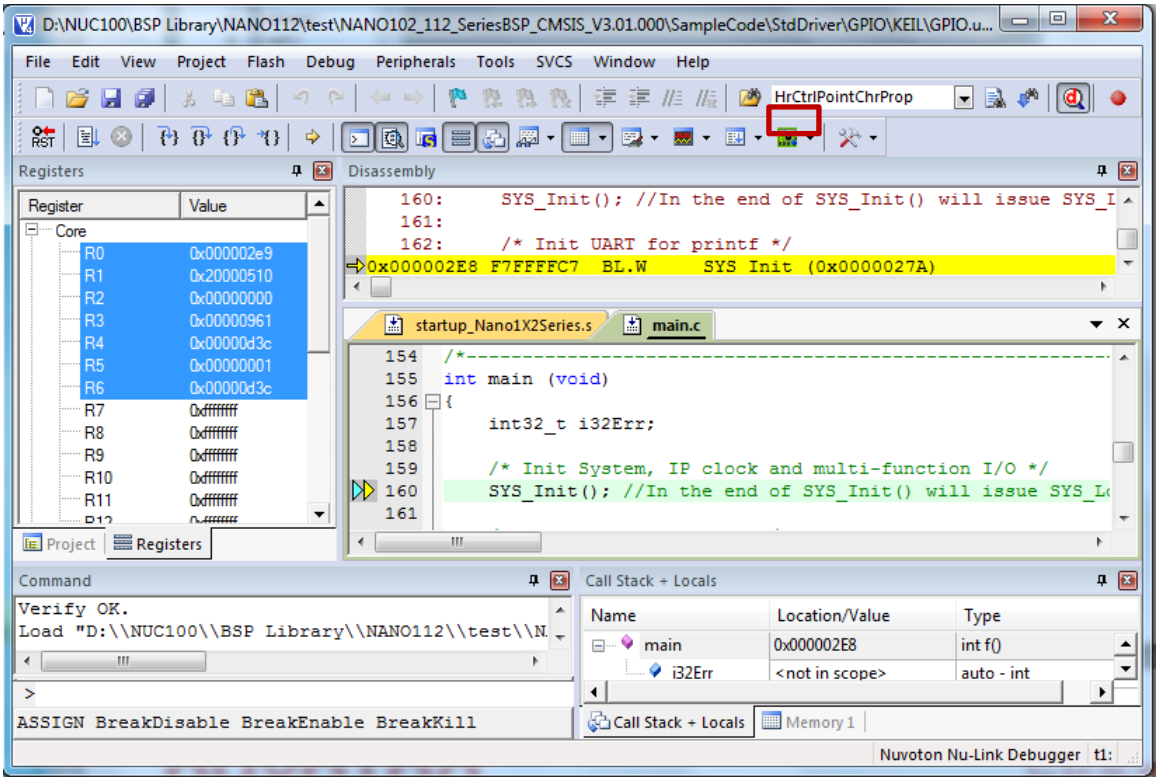


可以选择要烧录的位置，一般都烧录到APROM

✧ 然后关闭 option 窗口，点击 Load 就会下载了
看到下面的进度条，说明环境，BSP 安装都没有问题了



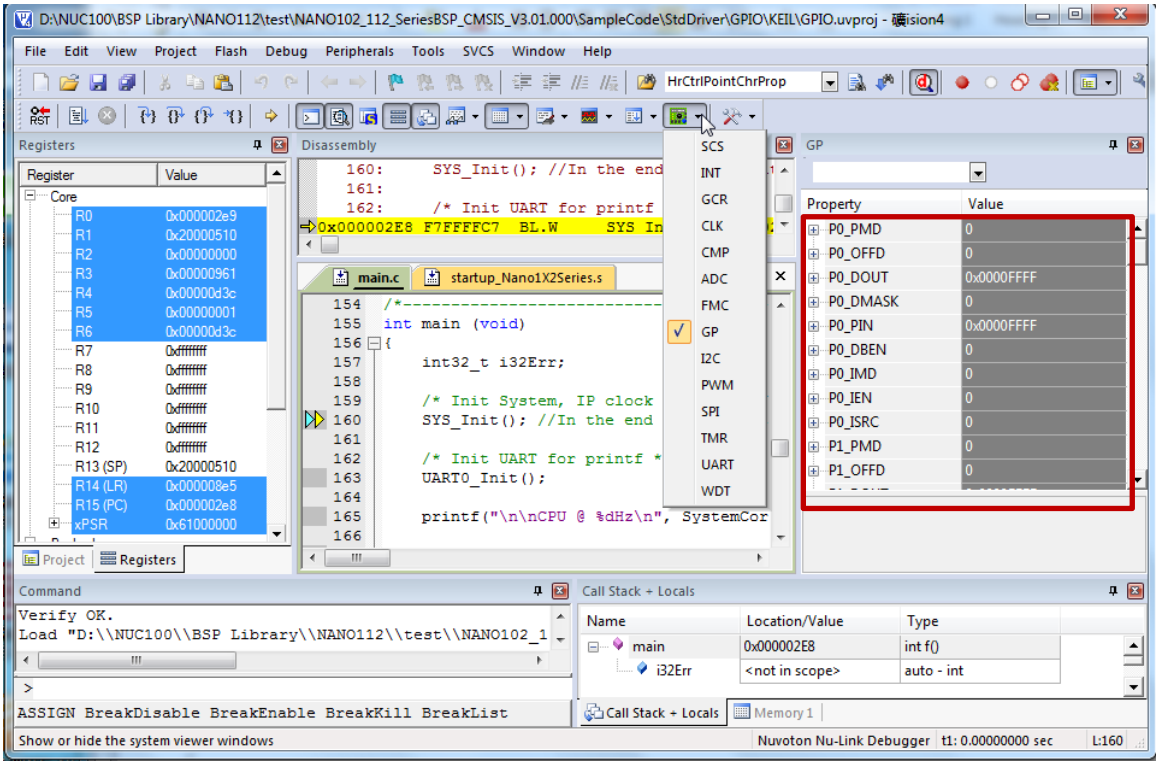
5) 点击 进入调试画面，如下



Keil会自动停在main函数，左边显示的是通用寄存器r0 – r13, lr pc cpsr寄存器的值

点击上图红框中绿色的按钮，就可以看到寄存器列表，点击GP，就会看到GPIO的所有寄存

器，如下图红框所示



6) 之后就可以进行单步/Free run 等调试动作

可以编译、调试、下载，环境基本就OK了。

下面是main.c的一部分代码，重复的函数我都拿掉了，例如：EINT0中断处理函数和EINT1中断处理函数，两个函数中，我只讲EINT0。

```
#include <stdio.h>
#include "nano1x2series.h"/*所有的寄存器都定义在这个文件里面*/

void GPABC_IRQHandler(void)
{
    /*检查是否PB.5发生了中断*/
    if (PB->ISRC & BIT5) {
        PB->ISRC = BIT5; /*清除中断标志*/
        PD0 = PD0 ^ 1; /*将PD0反转，可以用示波器监测发生了一次PB.5中断*/
        printf("PB.5 INT occurred. \n");

    } else {
        /*其它不期望的中断如果有发生，只是清除中断标志*/
        PA->ISRC = PA->ISRC;
        PB->ISRC = PB->ISRC;
        PC->ISRC = PC->ISRC;
```

```
        printf("Un-expected interrupts. \n");
    }
}

void EINT0_IRQHandler(void)
{
    /*清除PA.2中断标志*/
    PA->ISRC = BIT2;
    PD0 = PD0 ^ 1; /*将PD0反转，可以用示波器监测发生了一次PA.2中断*/
    printf("PA.2 EINT0 occurred. \n");
}
/*系统初始化函数，包括使能晶振，选择时钟源，使能外设时钟，设定GPIO为特殊功能*/
void SYS_Init(void)
{
    /*解锁保护寄存器*/
    SYS_UnlockReg();

    /*使能外部高速晶振HXT*/
    CLK->PWRCTL = (CLK->PWRCTL &~CLK_PWRCTL_HXT_EN_Msk)|(0x1 << CLK_PWRCTL_HXT_EN_Pos); //
HXT Enabled

    /*等待外部高速晶振HXT稳定*/
    CLK_WaitClockReady( CLK_CLKSTATUS_HXT_STB_Msk);

    /*CPU选择外部晶振做时钟源*/
    CLK->CLKSEL0 = (CLK->CLKSEL0 &~CLK_CLKSEL0_HCLK_S_Msk) | CLK_CLKSEL0_HCLK_S_HXT;

    /*UART选择外部晶振做时钟源*/
    CLK->CLKSEL1 = (CLK->CLKSEL1 &~CLK_CLKSEL1_UART_S_Msk) | (0x0 <<
CLK_CLKSEL1_UART_S_Pos);// 时钟源来自外部12 MHz 或者 32 KHz 晶振
     /*使能UART0时钟*/
    CLK->APBCLK |= CLK_APBCLK_UART0_EN;

    /* 更新 System Core Clock */
    /* 用户使用 SystemCoreClockUpdate() 自动更新 PllClock, SystemCoreClock 和 CycylesPerUs
       这些变量用于计算波特率，延时等需要用到系统时钟的地方
    */
    SystemCoreClockUpdate();

    /* 初始化多功能IO引脚 */
    /* 将 PB 多功能引脚设为 UART0 RXD 和 TXD ， PB13用作UART0 RX， PB14用作UART0 TX */
```

```
    SYS->PB_H_MFP &= ~(SYS_PB_H_MFP_PB13_MFP_Msk | SYS_PB_H_MFP_PB14_MFP_Msk);
    SYS->PB_H_MFP |= (SYS_PB_H_MFP_PB13_MFP_UART0_RX | SYS_PB_H_MFP_PB14_MFP_UART0_TX);

    /* 重新锁住写保护寄存器 */
    SYS_LockReg();
}

void UART0_Init()
{
     /* 初始化UART0，波特率设为115200,默认字节长度8bit，1个STOP位，没有校验位 */
    UART_Open(UART0, 115200);
}

/* MAIN function
*/
int main (void)
{
    int32_t i32Err;

    /* 使能晶振，设定CPU时钟，使能外设时钟，设定多功能引脚 */
    SYS_Init();

    /* 初始化UART0 用于 printf */
    UART0_Init();

    printf("\n\nCPU @ %dHz\n", SystemCoreClock);

    printf("+-------------------------------------+ \n");
    printf("|    NANO1x2 GPIO Driver Sample Code   | \n");
    printf("+-------------------------------------+ \n");

    /* GPIO 基本功能测试 --- 从引脚输入/输出high/low */
    printf("  >> Please connect PB.0 and PD.4 first << \n");
    printf("     Press any key to start test by using [Pin Data Input/Output Control]
\n\n");
    getchar();/* 使用超级终端从UART0任意输入一个字符开始测试 */

    /* 将 PB.0 配置为输出模式，PD.4配置为输入模式 */
    GPIO_SetMode(PB, BIT0, GPIO_PMD_OUTPUT);
    GPIO_SetMode(PD, BIT4, GPIO_PMD_INPUT);
```

```
    i32Err = 0;
    printf("  GPIO Output/Input test ...... \n");

    PB0 = 0; /* 从PB.0输出0 */
    if (PD4 != 0) {/* 查看PD.4是否也变成0？ */
        i32Err = 1; /* 如果PD.4没有变成0，说明PB.0和PD.4要么没有接到一起，要么模式没有设定对 */
    }

    PB0 = 1; /* 从PB.0输出1 */
    if (PD4 != 1) {/* 查看PD.4是否也变成1？ */
        i32Err = 1; /* 如果PD.4没有变成1，说明PB.0和PD.4要么没有接到一起，要么模式没有设定对 */
    }

    /* 如果出错，将打印一些信息，报告出错 */
    if ( i32Err ) {
        printf("  [FAIL] --- Please make sure PB.0 and PD.4 are connected. \n");
    } else {
        printf("  [OK] \n");
    }

    /* GPIO 中断功能测试，将PB.0 和 PD.4 都配置为输入模式 */
    GPIO_SetMode(PB, BIT0, GPIO_PMD_INPUT);
    GPIO_SetMode(PD, BIT4, GPIO_PMD_INPUT);

     printf("\n  PB5, PE2, PA2(INT0) and PF2(INT1) are used to test interrupt\n  and
control LED(PD0)\n");

    /*PD.0外接LED灯，将PD.0配置为输出模式 */
    GPIO_SetMode(PD, BIT0, GPIO_PMD_OUTPUT);

    /* 将 PB.5 配置为输入模式并使能上升沿触发中断 */
    GPIO_SetMode(PB, BIT5, GPIO_PMD_INPUT);
    GPIO_EnableInt(PB, 5, GPIO_INT_RISING); /* 使能 PB.5 上升沿触发中断 */
    NVIC_EnableIRQ(GPABC_IRQn); /* 使能GPA/GPB/GPC中断向量，它们共用一个中断向量 */


    /*  将PE2 配置为输入模式并打开内部上拉，使能下降沿触发中断 */
    GPIO_SetMode(PE, BIT2, GPIO_PMD_INPUT);
    GPIO_ENABLE_PULL_UP(PE, BIT2); /* 使能PE2内部上拉电阻 */
    GPIO_EnableInt(PE, 2, GPIO_INT_FALLING); /* 使能 PE2 下降沿触发中断 */
    NVIC_EnableIRQ(GPDEF_IRQn); /* 使能GPD/GPE/GPF中断向量，它们共用一个中断向量 */
```

```
    /* 将 PA2 配置为外部中断0: EINT0，并使能下降沿触发中断 */
    SYS->PA_L_MFP = (SYS->PA_L_MFP & ~0xf00) | 0x100; /* PA2用作外部中断0: EINT0 */
    GPIO_SetMode(PA, BIT2, GPIO_PMD_INPUT); /* 将PA2设为输入模式 */
    GPIO_EnableEINT0(PA, 2, GPIO_INT_FALLING); /* 下降沿触发中断 */
    NVIC_EnableIRQ(EINT0_IRQn); /* 使能外部中断0向量 */

    /* 将 PF2 配置为外部中断1: EINT1，并使能上升沿/下降沿都触发中断 */
    SYS->PF_L_MFP = (SYS->PF_L_MFP & ~0xf00) | 0x100; /* PA2用作外部中断1: EINT1 */
    GPIO_SetMode(PF, BIT2, GPIO_PMD_INPUT); /* 将PF2设为输入模式 */
    GPIO_EnableEINT1(PF, 2, GPIO_INT_BOTH_EDGE); /* 上升沿/下降沿都触发中断 */
    NVIC_EnableIRQ(EINT1_IRQn); /* 使能外部中断1向量 */

    /* 使能IO中断消抖功能，并选择消抖采样周期，注意power down时，没有消抖功能 */
    GPIO_SET_DEBOUNCE_TIME(GPIO_DBCLKSRC_HCLK, GPIO_DBCLKSEL_1); /* 选择HCLK作为消抖时钟源 */
    GPIO_ENABLE_DEBOUNCE(PB, BIT5); /* 使能PB.5引脚的消抖功能 */
    GPIO_ENABLE_DEBOUNCE(PE, BIT2); /* 使能PE.2引脚的消抖功能 */
    GPIO_ENABLE_DEBOUNCE(PA, BIT2); /* 使能PA.2引脚的消抖功能 */
    GPIO_ENABLE_DEBOUNCE(PF, BIT2); /* 使能PF.2引脚的消抖功能 */

    /* 等待发生中断 */
    while (1);

}
```

我们的 M0/M4 系列因为有多个时钟源，所以 CPU 和外设都有多个时钟源可以选择。系统初始化的步骤如下：

- ✧ 使能外部/内部晶振，并等待晶振稳定（PWRCTL 寄存器和 CLKSTATUS 寄存器）
- ✧ 选择 CPU 时钟源（CLKSEL0 寄存器）
- ✧ 选择外设时钟源（CLKSEL1/CLKSEL2 等寄存器）
- ✧ 使能外设时钟（APBCLK 寄存器）
- ✧ 设定 GPIO 的功能（PA_L_MFP/PA_H_MFP 等寄存器）

M0/M4 系列一般有 5 个时钟源可以选择。内部有 2 个晶振：内部高速晶振（HIRC）一般 22.1184M 或者 12M 或者 16M；内部低速晶振(LIRC)，一般是 10K，有的 32.768K
外部可以接 2 个晶振：外部高速晶振(HXT)，一般可以接 4M ~ 24M；外部低速晶振(LXT)，一般接 32.768K
另外再加锁相环(PLL)，可以选择 HXT 和 HIRC 超频到 50M，或者更高

这 5 种时钟源，每个外设和 CPU 大都可以选择。

外设选好上面 5 种时钟哪种做时钟源之后，写 APBCLK 寄存器使能时钟

注意：**如果外设时钟不等于 CPU 时钟(比 CPU 时钟慢)，写外设寄存器的时候需要 SW 自己同步。**看门狗一般选择内部 10K 时钟做时钟源，我们就以看门狗代码为例，代码如下：

```
WDT->CTL = WDT_TIMEOUT_2POW14| 0 |
            (0 << WDT_CTL_WTRE_Pos) |
            (1 << WDT_CTL_WTWKE_Pos);/*喂狗周期 2^14，使能唤醒功能*/
Delay(200);/*Delay 200us*/
WDT->CTL | = WDT_CTL_WTE_Msk ;/*使能看门狗*/
```

上面的代码连续写两次CTL寄存器，第二次写和第一次写之间需要等2个看门狗时钟周期，就是2个10K时钟，大概200us

如果上面的代码改成这样：WDT->CTL = WDT_TIMEOUT_2POW14| WDT_CTL_WTE_Msk |0 | (0 << WDT_CTL_WTRE_Pos) | (1 << WDT_CTL_WTWKE_Pos);/*喂狗周期 2^14，使能唤醒功能，同时使能看门狗*/

这样就不需要 delay 了，因为同一个寄存器只写一次。

了解调试、下载、仿真的方法之后，接下来我们介绍一下量产工具。

2.5 量产工具

量产分为先贴片再下载和先下载再贴片两种方式。Nu-Link和Nu-Link_MP就属于先贴片，然后通过板子的SWD接口烧录FW；Nu-Gang和第三方烧录工具例如：河洛和希尔特就属于先烧录再贴片。

2.5.1 使用 Nu-Link 量产

先用上位机ICP Tool将FW下载到Nu-Link中，然后就可以通过Nu-Link上的按键离线下载。缺点是一次只能烧录1片板子。

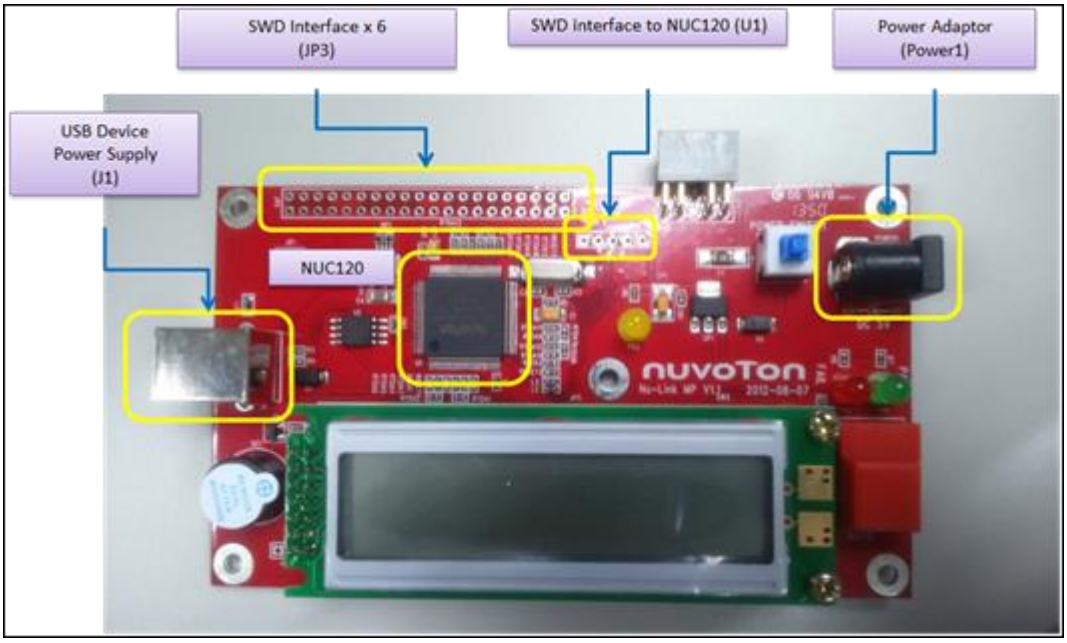
但是ICP tool 提供DOS Command 模式，这样可以在DOS下，使用批处理文件烧录。一台PC最多可以插16个Nu-Link。每个Nu-Link有独立的ID号。Nu-Link Command Tool的用法可以看Tool带的文档，里面解释的很清楚

2.5.2 使用 Nu-Link-MP 量产

先用ICP Tool将FW下载到Nu-Link_MP中，然后就可以通过Nu-Link-MP上的按键离线下载。一次可以烧录6片板子。

但是该板子里面是不带程序的，需要客户和新唐签NDA拿到ICP library和demo code(负责将应用程序烧录到目标板)，然后将该demo code和应用程序一起编译成一个bin文件，烧录到Nu-Link-MP里面。上电后，应用程序就会被同时烧录到6片目标板。

下图就是Nu-Link_MP，那两排孔就是SPI接口，可以接6片目标板



2.5.3 使用 Nu-Gang 量产

Nu-Gang如上图所示，是新唐出的烧录工具，可以一次烧录4片芯片。它需要先烧录再贴片

2.5.4 第三方工具

目前支持新唐芯片烧录的第三方工具包括：河洛、希尔特等

2.6 总结

至此大家会编译、调试、下载。知道Nu-Link和Nu-Link_ME是做什么用的。了解了BSP的结构。

其他一些总结如下：

1) 为了省电，整个系统的晶振除了内部 HIRC，其他晶振默认都是不起振的。
2) 每个 IP 的时钟默认也是关闭的。

3) 所有的 IP 包括 CPU 有多个时钟源可以选择。
4) 整个系统要使用的晶振都需要软件起振。
5) 每个 IP 在使用之前要选择时钟源+使能时钟，然后配置多功能引脚。
6) 要新建 project 的话建议拿我们 BSP 下面已有的*.uvproj 文件来改

这些了解之后，基本上对新唐的芯片有了一个整体的认识。下面介绍一下每个IP的初始化流程

3 IP的初始化流程

芯片一般包含5种时钟源：

- ✧ 内部高速振荡器 HIRC(一般是 22.1184M，也有的是 12M 或者 16M)
- ✧ 内部低速振荡器 LIRC(一般是 10K，有的是 32K)
- ✧ 外部高速晶振 HXT(范围一般是 4~24M)
- ✧ 外部低速晶振 LXT(32K)
- ✧ PLL

注意：有的芯片外部晶振HXT和LXT是共用引脚的。

为了省电，上电之后，默认只有HIRC是起振的，其他的晶振都需要软件使能才会起振。起振之后也不是马上供给各个IP，也*需要软件使能之后该IP才会有时钟*，它的寄存器才能被访问和读/写。

每个IP的初始化包括2部分，系统初始化和IP本身功能的初始化，流程如下：

1) 系统初始化
   - ■ 选择 IP 时钟源，一般有上面 5 种可以选择，如果某个 IP 的时钟源不能选择，则它的时钟源就是 HCLK（就是 CPU 的时钟）。**选择时钟源之前，IP 默认选择的时钟源和将选择的时钟源必须都要保持使能**。不然 IP 将不工作。一般 IP 默认选择的时钟都是 HIRC，如果 IP 想选择 HXT 做时钟源，必须先使能 HXT(如果没有修改过 Config0，上电后 HIRC 默认是使能的)，然后 IP 切换时钟源，最后如果 CPU 和所有 IP 都没有使用 HIRC，为了省电可以将 HIRC 关闭。千万不能先关闭 HIRC，再切 IP 的时钟源，这样 IP 会不工作的，或者工作不正常。**虽然在切时钟源的时候 IP 也许并没有工作，但是也要保证 2 个时钟源（当前选择的和将要选择的）都是使能的才能切时钟源。**
   - ■ 使能 IP 时钟
   - ■ 配置 MFP 寄存器（多功能引脚寄存器）
2) 最后配置 IP 的功能

每个 IP 系统初始化步骤都是一样的：选择时钟源、使能时钟、配置多功能引脚

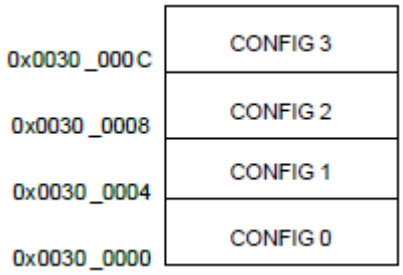
下面的代码基本上都使用的 NANO112 BSP: *NANO102_112_SeriesBSP_CMSIS_V3.01.000*。其它系列的 BSP 和 NANO112 的 BSP 函数基本上一样，但是宏定义有些不同。不过各个 IP 的使用方法大同小异，学会了使用 NANO112，其它的用起来也不难。

3.1 系统内存映射

整个系统ROM分为4块：APROM、LDROM、Dataflash和config Area。有的芯片还有SPROM，那就有5块ROM

上电执行APROM还是LDROM中的程序由config area寄存器决定

- ✧ APROM：一般用来放用户的程序
- ✧ LDROM：一般用来放 ISP，可以通过 UART 或者 USB 更新 APROM 中的程序
- ✧ Dataflash：一般用来存放用户的数据。这部分 ROM 有的芯片是独立的一块，有的芯片是从 APROM 中分一块，由软件决定大小
- ✧ Config area：用于设定上电之后芯片的一些行为，某些寄存器的缺省值是从 Config area 拿的。例如：从 APROM 还是 LDROM 开始执行代码，是否使能 BOD，是否使能 dataflash 等等。Config 区域映射如下：



有的芯片只有 CONFIG0 和 CONFIG1 有的芯片有 CONFIG0、CONFIG1、CONFIG2、CONFIG3。具体有多少个 CONFIG 寄存器，详细请参考 TRM

注意：Config area修改之后需要复位才能起做用。如果不用ICP tool 修改Config area，而用软件修改，修改之后需要软件发出CHIP reset才能起做奥！

大多芯片都支持IAP功能，这个功能的好处就是：

- ✧ 如果 LDROM 不用，而 APROM 不够了，可以将程序放一部分到 LDROM 中
- ✧ 如果 ISP 代码太大，LDROM 中放不下，可以放一部分到 APROM 中
- ✧ 甚至程序可以不用从地址 0x00000000 开始执行，从 APROM/LDROM 中任意地方开始都行

大家如果对IAP感兴趣，可以看一下《AN_0000_IAP Application Note_xx_xxx.pdf》

3.2 系统初始化

系统初始化包含了时钟(clock)初始化和多功能引脚（Multi Function Pin 简称MFP寄存器）配置。

```
void SYS_Init(void)
{
    /* 解锁保护寄存器 */
    SYS_UnlockReg();/*芯片中很多寄存器是写保护的，例如PWRCTL寄存器，要写这些寄存器需要先解锁*/

    /* 使能外部高速晶振，一般范围是 (4~24 MHz) */
    CLK->PWRCTL |= (0x1 << CLK_PWRCTL_HXT_EN_Pos); // HXT Enabled
```

```
    /* 等待外部时钟稳定，一般是12M */
    CLK_WaitClockReady( CLK_CLKSTATUS_HXT_STB_Msk);

    /* HCLK就是CPU 的时钟，切为外部晶振HXT */
    CLK->CLKSEL0 = (CLK->CLKSEL0 &~CLK_CLKSEL0_HCLK_S_Msk) | CLK_CLKSEL0_HCLK_S_HXT;

    /* 使能UART0和UART1两个IP的时钟 */
    CLK->APBCLK |= CLK_APBCLK_UART0_EN; // UART0 Clock Enable
    CLK->APBCLK |= CLK_APBCLK_UART1_EN; // UART1 Clock Enable

    /* 选择UART时钟源 */
    CLK->CLKSEL1 = (CLK->CLKSEL1 & ~CLK_CLKSEL1_UART_S_Msk)  | CLK_CLKSEL1_UART_S_HXT;//
选择外部 12 MHz or 32 KHz 做时钟源

    /* Update System Core Clock */
    /* 可以通过 SystemCoreClockUpdate() 来自动计算 PllClock, SystemCoreClock 和 CycylesPerUs
     */
    SystemCoreClockUpdate();

    /* 初始化 I/O 多功能引脚 */
    /* PB13用作UART0 接收，PB14用作UART0发送
     */
    SYS->PB_H_MFP &= ~(SYS_PB_H_MFP_PB13_MFP_Msk | SYS_PB_H_MFP_PB14_MFP_Msk);
    SYS->PB_H_MFP |= (SYS_PB_H_MFP_PB13_MFP_UART0_RX | SYS_PB_H_MFP_PB14_MFP_UART0_TX);

    /* PB4用作UART1 RTS，PB5用作UART1接收，PB6用作UART1 发送，PB7用作UART1 CTS
     */
    SYS->PB_L_MFP &= ~(SYS_PB_L_MFP_PB4_MFP_Msk | SYS_PB_L_MFP_PB5_MFP_Msk |
                       SYS_PB_L_MFP_PB6_MFP_Msk | SYS_PB_L_MFP_PB7_MFP_Msk);
    SYS->PB_L_MFP |= (SYS_PB_L_MFP_PB4_MFP_UART1_RTS | SYS_PB_L_MFP_PB5_MFP_UART1_RX |
                      SYS_PB_L_MFP_PB6_MFP_UART1_TX  | SYS_PB_L_MFP_PB7_MFP_UART1_CTS);

    /* 重新加锁 */
    SYS_LockReg();

}
```

3.2.1 时钟输出功能

新唐的M0/M4一般都有时钟输出功能，用于调试内部时钟频率，或者产生时钟给其它芯片使用。这个功能引脚一般叫CKO，设定寄存器是FRQDIV

```
Void CKO_Init()
{
    /*使能FRQDIV 时钟*/
    CLK->APBCLK |= CLK_APBCLK_FDIV_EN_Msk;
    /*选择FRQDIV 时钟源为HCLK，从CKO输出的时钟频率将与HCLK有关*/
    CLK->CLKSEL2 = (CLK-
>CLKSEL2&~(CLK_CLKSEL2_FRQDIV_S_Msk))|(CLK_CLKSEL2_FRQDIV_S_HCLK);//CKO using HCLK
    /*将P3.6配置为CKO功能*/
    SYS->P3_MFP &= ~( SYS_MFP_P36_Msk);
    SYS->P3_MFP |= (SYS_MFP_P36_CKO);//HCLK 从P3.6输出
    /*  CKO 输出的频率为 HCLK/1 */
    CLK->FRQDIV = CLK_FRQDIV_DIVIDER1_Msk | CLK_FRQDIV_DIVIDER_EN_Msk;
}
```

有的芯片输出的频率至少要/2，那么输出的频率就是HCLK/2了，这个要注意看CLK IP的FRQDIV寄存器的描述。

3.2.2 HIRC 补偿功能

新唐有些芯片可以用外部32K trim内部HIRC。一般HIRC全温度范围误差2%左右，在有些场合这个精度不够，这时候就可以用外部32K来trim内部HIRC，trim之后HIRC精度可以达到0.25%左右。Trim功能一旦使能就一直在工作，不会停止。如果外部32K晶振出错，trim就会停止，并发生中断。所以需要在中断里面重新启动trim功能。

```
/*LXT 时钟出错或者尝试次数达到限制将发生中断*/
void HIRC_IRQHandler(void)
{
    uint32_t u32IRCStatus;

    /*清除中断标志*/
    u32IRCStatus = SYS->IRCTRIMINT;
    SYS->IRCTRIMINT = u32IRCStatus;
    /*重新启动trim功能*/
    SYS->IRCTRIMCTL = SYS_IRCTRIMCTL_LOOP_32CLK| SYS_IRCTRIMCTL_TRIM_12M;
}
/*使能trim功能*/
void SYS_EnableAutoTrim()
{
    /*取32个32K时钟周期的平均值来trim HIRC*/
```

```
    SYS->IRCTRIMCTL = SYS_IRCTRIMCTL_LOOP_32CLK| SYS_IRCTRIMCTL_TRIM_12M;
    /*一旦出错将发生中断*/
    SYS->IRCTRIMIEN = SYS_IRCTRIMIEN_32KERR_EN | SYS_IRCTRIMIEN_FAIL_EN;
    NVIC_EnableIRQ(HIRC_IRQn);
}
```

只要调用SYS_EnableAutoTrim就可以使能Auto trim功能来trim HIRC了。

3.2.3 复位

新唐的芯片一般有3种复位方式：CPU reset、Chip reset和System reset

- ✧ CPU reset：就是将 CPU 执行指针 PC 直接指到 0 的地方重新执行程序
- ✧ Chip reset：就是整个芯片复位，类似于 POR 上电复位的方式，让程序重新执行
- ✧ System reset：类似于 Chip reset，除了不复位晶振电路和 Config Area 的值不会重新加载，其它的电路都会被复位

我们常用System reset切到APROM运行，或者切到LDROM运行。系统上电从APROM运行还是从LDROM运行由Config area决定，但是有时侯软件希望切到某个区域运行，这时候用System reset比较好。其实用CPU reset也可以，但是System reset的好处是它会将所有的IP都复位，防止它们在新的程序里面乱动作。

3.3 UART 初始化

新唐的M0/M4 UART都有16级或者64级FIFO，用来缓存UART数据的收/发。例如:如果RX FIFO中断触发级别设为14，UART接收14个字节才会发生RDA（接收数据可得）中断。这样可以降低CPU的loading。上面的情况，如果RX只接收到10个字节怎么办呢？这时候就要用到接收超时中断。当RX FIFO中收到1个字节以后，定时器就开始计数，如果定时器超时都没有再收到下一个字节就会发生接收超时中断(RTO)。

每个IP的初始化都需要先初始化时钟，然后才是IP功能初始化。初始化UART之前需要使能要用的晶振，然后选择时钟源并使能时钟。最后将UART用到的引脚切换为UART功能。

```
void UART0_Init()
{
    /* UART选择HIRC做时钟源 */
    CLK->CLKSEL1 = (CLK->CLKSEL1 & ~CLK_CLKSEL1_UART_S_Msk)  | CLK_CLKSEL1_UART_S_HIRC;
    /* 使能UART0 IP的时钟 */
    CLK->APBCLK |= CLK_APBCLK_UART0_EN;
    /* PB13用作UART0 接收，PB14用作UART0发送
     */
    SYS->PB_H_MFP &= ~(SYS_PB_H_MFP_PB13_MFP_Msk | SYS_PB_H_MFP_PB14_MFP_Msk);
    SYS->PB_H_MFP |= (SYS_PB_H_MFP_PB13_MFP_UART0_RX | SYS_PB_H_MFP_PB14_MFP_UART0_TX);
```

```
    /* Init UART0  */
    UART_Open(UART0, 115200);/*默认数据长度为8bit，没有奇偶校验，1个停止位*/
    /*初始化UART1，波特率9600，数据长度8bit，1个停止位，偶校验*/
    UART_SetLine_Config(UART1, 9600, UART_WORD_LEN_8, UART_PARITY_EVEN, UART_STOP_BIT_1);
}
```

UART_Open会根据UART选择的时钟源计算波特率。

上面的代码执行之后，UART_WRITE(UART0, 0x31)就会从UART TX引脚发送0x31了。如果要使用printf打印信息，keil project中加入retarget.c就可以，然后由宏定义#define DEBUG_PORT UART0决定printf从哪个UART口打印。

如果想使用中断接收数据，代码如下：

```
void UART0_Init()
{
    /* 选择UART时钟源 */
    CLK->CLKSEL1 = (CLK->CLKSEL1 & ~CLK_CLKSEL1_UART_S_Msk)  | CLK_CLKSEL1_UART_S_HIRC;
    /* 使能UART0 IP的时钟 */
    CLK->APBCLK |= CLK_APBCLK_UART0_EN;
    /* PB13用作UART0 接收，PB14用作UART0发送
     */
    SYS->PB_H_MFP &= ~(SYS_PB_H_MFP_PB13_MFP_Msk | SYS_PB_H_MFP_PB14_MFP_Msk);
    SYS->PB_H_MFP |= (SYS_PB_H_MFP_PB13_MFP_UART0_RX | SYS_PB_H_MFP_PB14_MFP_UART0_TX);

    /* Init UART0  */
    UART_Open(UART0, 115200);/*默认数据长度为8bit，没有奇偶校验，1个停止位*/
    /*初始化UART1，波特率9600，数据长度8bit，1个停止位，偶校验*/
    UART_SetLine_Config(UART1, 9600, UART_WORD_LEN_8, UART_PARITY_EVEN, UART_STOP_BIT_1);
    /*设置接收超时时间为40，单位波特率*/
    UART_SetTimeoutCnt(UART0, 40);
    /*设置接收FIFO触发级别为14B*/
    UART_SET_RX_FIFO_INTTRGLV(UART0, UART_TLCTL_RFITL_14BYTES);
    /*使能接收FIFO 阀值中断，和接收超时中断*/
    UART_ENABLE_INT(UART0, (UART_IER_RDA_IE_Msk | UART_IER_RTO_IE_Msk));
    NVIC_EnableIRQ(UART0_IRQn);
}
/*UART0中断处理函数*/
void UART0_IRQHandler(void)
{
    uint8_t u8InChar=0xFF;
    uint32_t u32IntSts= UART0->ISR;
    /*发生接收阀值中断或者接收超时中断*/
```

```
    if(u32IntSts & (UART_ISR_RDA_IS_Msk| UART_ISR_RTO_IS_Msk)) {

        /* 读走接收FIFO中所有的数据，直到接收FIFO为空 */
        while(UART_GET_RX_EMPTY(UART0)==0) {
            /* 从接收FIFO中读一个数据 */
            u8InChar = UART_READ(UART0);
        }
    }
}
```

UART IP有个复位函数：SYS_ResetModule(UART0_RST); 其实新唐的芯片每个IP都有单独的复位控制。什么时候需要复位UART0呢？如果该代码是通过CPU reset执行到的，那原本UART可能在工作，RX FIFO中可能有数据，可能会发生中断等等一些无法预估的事情，这时候最好复位一下UART IP的逻辑。

3.4 GPIO 初始化

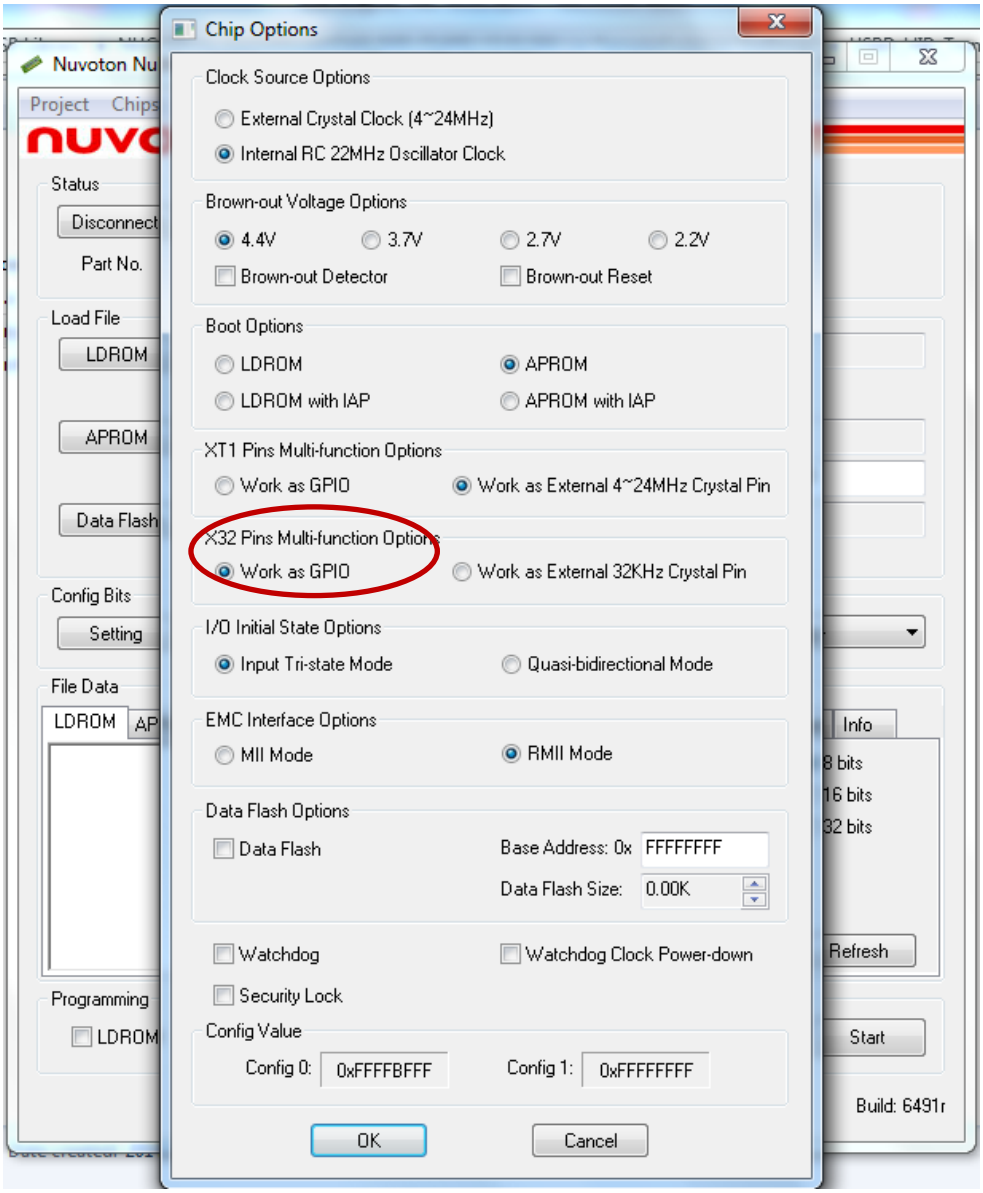
新唐所有的M0/M4芯片基本上所有的IO都可以发生中断，为了符合大家的习惯还是有所谓的外部中断EINT0和EINT1。有2跟GPIO脚可以配置为EINT0功能和EINT1功能，分别将发生EINT0中断和EINT1中断。其它的IO脚也会导致发生中断，但是为很多IO共用一个向量的方式，例如：PA/PB/PC共用中断号4，PD/PE/PF共用中断号5。

每个IO内部一般都带内部上拉电阻，软件可以打开。一般用于按键，或者I2C不想外部加上拉电阻的情况。

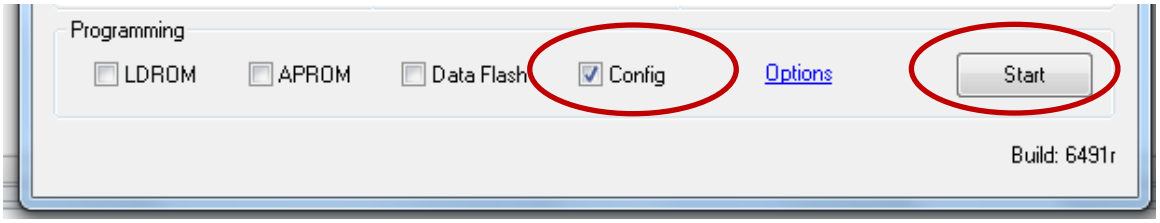
每个IO都有边沿中断消抖功能，一般用作按键的时候或者外部输入信号会抖动的情况下需要打开。

NUC472/NUC442和M451的IO比较特别，它们有多个Power Domain，就是所有的IO的供电电压可以不同，在使用时一定要注意。

- NUC472 和 NUC442 Vbat 负责 PG.14、PG.15、PA.0 和 PA.1，这些脚由 Vbat 负责供电，所以 **Vbat 一定要有电**，即使不接电池也要和 VDD 接到一起。另外 PG.14 和 PG.15 默认为晶振功能用于外接 32K 晶振，如果想改为 GPIO 需要修改 Config area。我们一般用 ICP tool 修改 Config area，因为 Config area 修改之后需要复位才能起做用。用 USB 线将 Nu-Link 和目标板接到 PC 上，打开 ICP tool，连接到目标板。点击 Settings，然后如红色框所示，将”Work as GPIO” 打勾



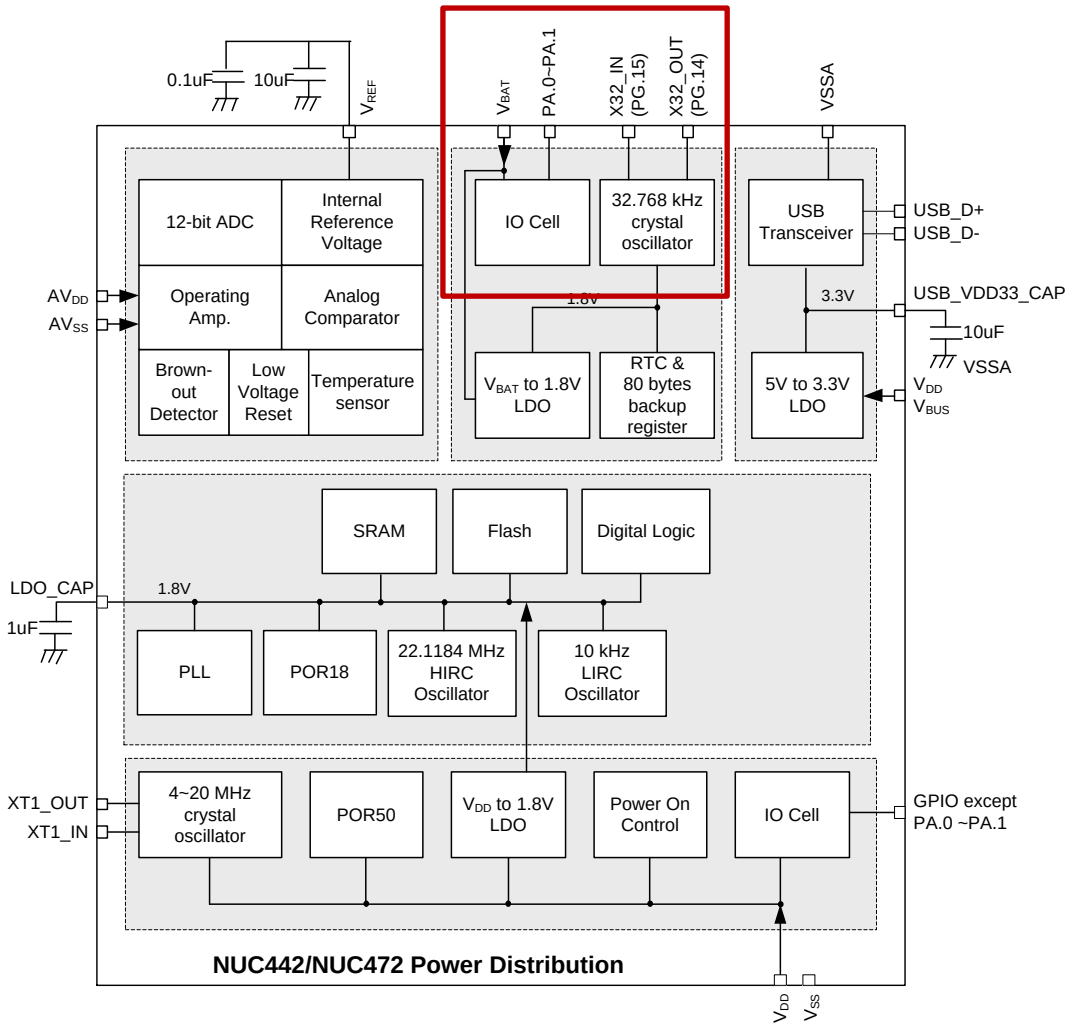
之后打勾 Config，点击 Start 就修改成功了



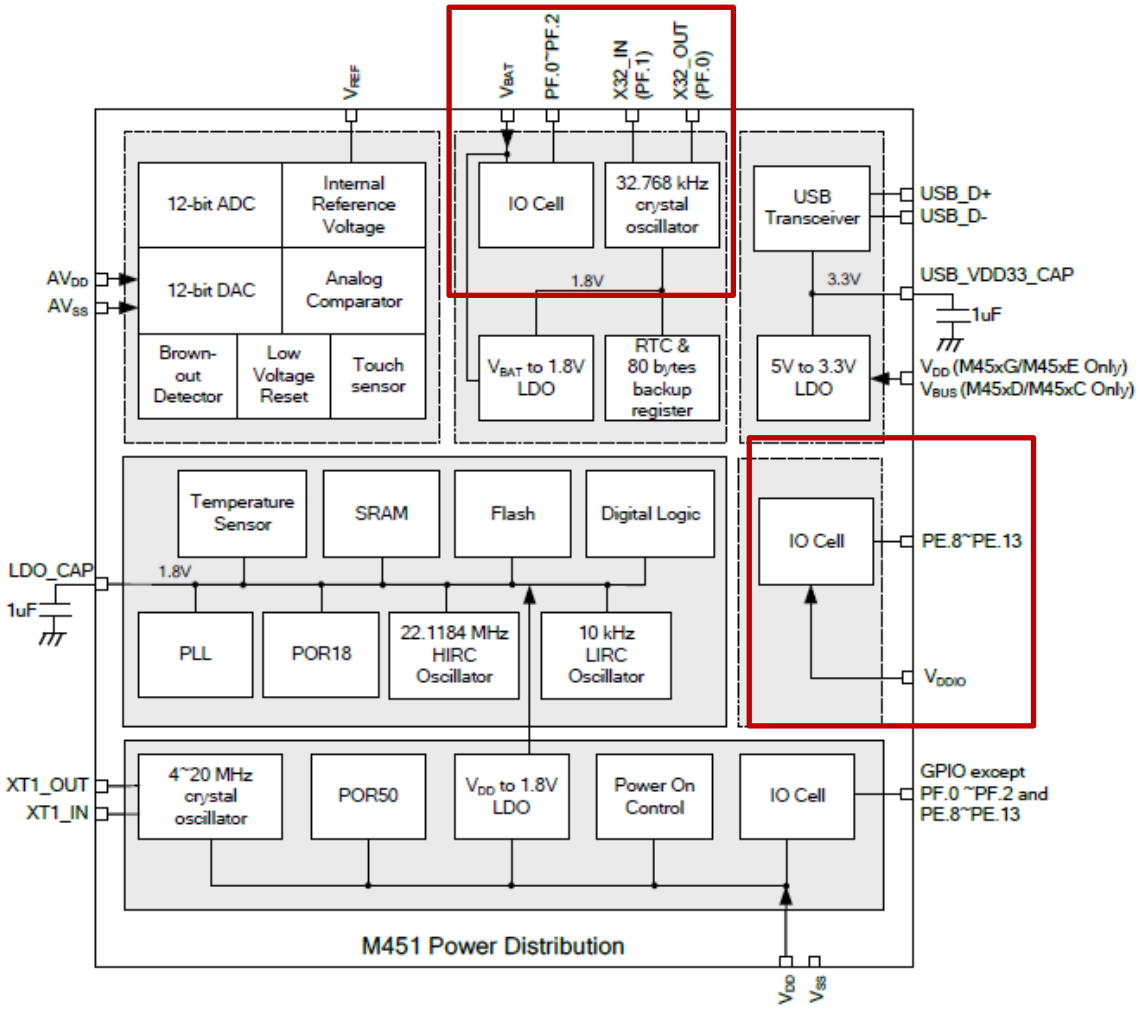
- M451 有 Vbat、VDDIO 和 VDD 共 3 个 power domain。Vbat 负责 PF0 ~ PF2，VDDIO 负责 PE.8 ~ PE.13

  并且 Vbat 和 VDD 以及 VDDIO 这 3 个电压可以不同，例如：VDD 给 5V， VDDIO 可以给 3V，Vbat 可以给 3.6V，这可以用于板子上有多个电压的情况。

NUC472 Power Domain详细资料如下图：



M451 Power Domain详细资料如下图：



下面介绍一下IO的初始化，IO初始化无非就是将IO设为输入/输出/开漏/准双向等模式，然后toggle IO，或者读IO的状态。另外也可能配置中断。

```
Void GPIO_Init()
{
    /* PB.0 配置为输出，PD.4 配置为输入模式 */
    GPIO_SetMode(PB, BIT0, GPIO_PMD_OUTPUT);
    GPIO_SetMode(PD, BIT4, GPIO_PMD_INPUT);
    /* PD.0 配置为输出*/
    GPIO_SetMode(PD, BIT0, GPIO_PMD_OUTPUT);
    /*从PB0输出0*/
    PB0 = 0;
    /* 将 PB5 配置为输入模式并使能上升沿中断 */
    GPIO_SetMode(PB, BIT5, GPIO_PMD_INPUT);
    GPIO_EnableInt(PB, 5, GPIO_INT_RISING);
    NVIC_EnableIRQ(GPABC_IRQn);
```

```
    /*  将 PE2 配置为输入模式，打开内部上拉，并使能下降沿中断 */
    GPIO_SetMode(PE, BIT2, GPIO_PMD_INPUT);
    GPIO_ENABLE_PULL_UP(PE, BIT2);
    GPIO_EnableInt(PE, 2, GPIO_INT_FALLING);
    NVIC_EnableIRQ(GPDEF_IRQn);

    /* 将 PA2 配置为 EINT0 引脚，并使能下降沿中断 */
    SYS->PA_L_MFP = (SYS->PA_L_MFP & ~ SYS_PA_L_MFP_PA2_MFP_Msk) |
SYS_PA_L_MFP_PA2_MFP_EINT0;
    GPIO_SetMode(PA, BIT2, GPIO_PMD_INPUT);
    GPIO_EnableEINT0(PA, 2, GPIO_INT_FALLING);
    NVIC_EnableIRQ(EINT0_IRQn);
    /* 使能消抖功能并选择消抖采样周期。因为PB.5和PE.2以及PA.2都使能了中断，所以最好打开消抖功能 */
    GPIO_SET_DEBOUNCE_TIME(GPIO_DBCLKSRC_HCLK, GPIO_DBCLKSEL_1);
    GPIO_ENABLE_DEBOUNCE(PB, BIT5);
    GPIO_ENABLE_DEBOUNCE(PE, BIT2);
    GPIO_ENABLE_DEBOUNCE(PA, BIT2);
}
void GPABC_IRQHandler(void)
{
    /* 检查是否发生PB.5 中断 */
    if (PB->ISRC & BIT5) {
        PB->ISRC = BIT5;/*清除PB.5中断标志*/
        PD0 = PD0 ^ 1;
        printf("PB.5 INT occurred. \n");

    } else {
        /* 不期望的其它中断 */
        PA->ISRC = PA->ISRC;
        PB->ISRC = PB->ISRC;
        PC->ISRC = PC->ISRC;
        printf("Un-expected interrupts. \n");
    }
}

void GPDEF_IRQHandler(void)
{
    /*检查是否发生PE.2 中断*/
    if (PE->ISRC & BIT2) {
        PE->ISRC = BIT2;
        PD0 = PD0 ^ 1;
```

```
        printf("PE.2 INT occurred. \n");
    } else {
        /* 不期望的其它中断 */
        PD->ISRC = PD->ISRC;
        PE->ISRC = PE->ISRC;
        PF->ISRC = PF->ISRC;
        printf("Un-expected interrupts. \n");
    }
}
void EINT0_IRQHandler(void)
{
    /* 清除 PA.2中断标志 */
    PA->ISRC = BIT2;
    PD0 = PD0 ^ 1;/*反转PD.0*/
    printf("PA.2 EINT0 occurred. \n");
}
```

GPIO IP有个寄存器叫OFFD(OFF Digital)，关闭相应管脚的数字通路的意思。如果某个管脚配置为模拟功能，例如：ADC、DAC、ACMP、SEG、COM、DH1、DH2、V1、V2、V3等功能，这些管脚对应的OFFD bit应该使能，关闭到数字区域的通路，避免内部数字器件不停的打开、关闭而漏电。

3.5 Timer 初始化

新唐的定时器一般有很多功能：普通的定时功能，事件计数功能，捕获功能，超时触发ADC等等。大家如果感兴趣可以读一下《NANOB Timer功能介绍以及在弱灌注中的应用.pdf》，虽然各个系列Timer功能有差异，但是使用方法上基本一致，只是细节上有些差异。用到的时钟记得在Sys_Init中使能并等待时钟稳定。

这里只介绍简单的定时功能。

```
/*Timer0中断处理函数，Timer0发生超时，捕获等都会发生该中断。但是因为下面的代码只使能了超时中断，所以该代码只demo超时的处理*/
void TMR0_IRQHandler(void)
{
    static uint32_t sec = 1;
    printf("%d sec\n", sec++);/*打印时间*/

    // 清除中断标志
    TIMER_ClearIntFlag(TIMER0);

}
```

```
int Timer_Init(void)
{
    /* Timer0选择HXT做时钟源，并且不除频 */
    CLK_SetModuleClock(TMR0_MODULE, CLK_CLKSEL1_TMR0_S_HXT, CLK_TMR0_CLK_DIVIDER(1));
    /* 使能Timer0的时钟 */
    CLK_EnableModuleClock(TMR0_MODULE);
    /* 初始化Timer0，周期模式，每秒发生一次中断*/
    TIMER_Open(TIMER0, TIMER_PERIODIC_MODE, 1);
    /*如果要修改预分频和比较寄存器的值可以用下面两个宏*/
     //TIMER_SET_CMP_VALUE(TIMER0, 0xFFFFFF);//修改比较寄存器的值
     //TIMER_SET_PRESCALE_VALUE(TIMER0, 0x0);//修改预分频的值

    /* 使能Timer0中断 */
    TIMER_EnableInt(TIMER0);
    NVIC_EnableIRQ(TMR0_IRQn);

    // 启动 Timer 0
    TIMER_Start(TIMER0);
}
```

这个Timer_Init函数里面使能时钟和IP选择时钟源的函数和3.1节系统初始化中的不太一样，该节中IP选择时钟源和使能时钟调用的是函数，而3.1节系统初始化中是直接访问寄存器的方式。这两种方式都可以。3.1节的方式比较节省ROM，本节的方式容易修改。

3.6 ADC 初始化

新唐的ADC采样速度目前从300KSPS~1MSPS不等，通道8～16个不等。一般只有1组ADC，各个通道轮流采样。也有的是2组ADC，可以同时采样2个通道，例如:M0519。有的芯片有外部参考引脚Vref，有的只能使用AVDD（模拟电路电源脚）做参考。有的芯片有内部参考和温度传感器。但是内部参考一般不太准，全温度范围一般误差10%左右。使用内部参考的时候如果有Vref引脚可以从Vref输出，然后接1uF到地，这样内部参考更稳定。内部温度传感器使用ADC采样，规格如下，在Datasheet文件里面：

PARAMETER	SYM.	SPECIFICATIONS				TEST CONDITION (supply voltage = 3V)
		MIN.	TYP.	MAX.	UNIT	
Detection Temperature	T_{DET}	-40		+85	°C	
Operating current	I_{TEMP}	-	5	-	μA	
Gain	V_{TG}	-1.76	-1.68	-1.60	mV/°C	
Offset	V_{TO}	735	745	755	mV	Tempeature at 0 °C

Note: Internal operation voltage comes form LDO.

ADC采样得到的电压值，减去offset，然后每降低1.68mV温度上升一度。

下面代码演示的是ADC 单端，单次采样模式。照样需要先选择ADC IP时钟源，使能ADC IP的时钟，然后配置多功能引脚。用到的时钟记得在Sys_Init中使能并等待时钟稳定。这里只列出跟ADC有关的初始化。

```
int32_t ADC_Init (void)
{
    /* 选择时钟源，ADC选HIRC,并且除频=12M/5 */
    CLK_SetModuleClock(ADC_MODULE,CLK_CLKSEL1_ADC_S_HIRC,CLK_ADC_CLK_DIVIDER(5));
    /* 使能ADC IP的时钟 */
    CLK_EnableModuleClock(ADC_MODULE);
    /* 配置 PA.0 用作ADC 通道0 */
    SYS->PA_L_MFP = (SYS->PA_L_MFP & ~SYS_PA_L_MFP_PA0_MFP_Msk) |
SYS_PA_L_MFP_PA0_MFP_ADC_CH0;

    /* 关闭PA.0 的数字通路 */
    PA->OFFD |= ((1 << 0) << GP_OFFD_OFFD_Pos);
    /* ADC工作在单端single模式，并使能通道0准备采样 */
    ADC_Open(ADC, ADC_INPUT_MODE_SINGLE_END, ADC_OPERATION_MODE_SINGLE, ADC_CH_0_MASK);

    /* Power on ADC*/
    ADC_POWER_ON(ADC);

    /* 使用AVDD电压作为参考*/
    ADC_SET_REF_VOLTAGE(ADC,ADC_REFSEL_VREF);

    /* 使能ADC 中断，转换完成将发生中断*/
    ADC_EnableInt(ADC, ADC_ADF_INT);
    NVIC_EnableIRQ(ADC_IRQn);

    u8ADF = 0;
```

```
    /*开始转换*/
    ADC_START_CONV(ADC);

    while (u8ADF == 0);/*等待转换完成*/
    /*取得转换结果*/
    u32Result = ADC_GET_CONVERSION_DATA(ADC, 0);
    printf("Channel 0 conversion result is 0x%x\n",u32Result);

    ADC_DisableInt(ADC, ADC_ADF_INT);
}
```

ADC有3种工作模式：单次、单次循环和连续循环模式。

- ✧ 单次：就是在某个使能的通道上完成一次转换就停止
- ✧ 单次循环：就是在所有使能的通道上完成一次转换就停止
- ✧ 连续循环：就是在所有使能的通道完成一次转换，再完成一次转换，连续不断的转换，直到软件将其停止

ADC的信号有2种输入模式：单端，差分

- ✧ 单端：就是采样单个通道
- ✧ 差分：就是两个通道的信号相减再采样

3.7 I2C 初始化

I2C相信大家都不陌生，很多人都用GPIO模拟过I2C。但是模拟的太占CPU资源，并且一般只能模拟Master，模拟slave还是挺困难的。

下面介绍一下I2C IP。

这个是I2CON寄存器，I2C的控制寄存器

INTEN	Reserved	I2C_STS	START	STOP	ACK	IPEN

从左往右依次为：中断使能位，状态变化指示位，发送START信号，发送STOP信号，回ACK，使能I2C IP

- ✧ I2C_STS：I2C 的状态发生变化该位就会置 1
- ✧ START：请求 I2C IP 发送 START 信号。一旦发送成功，该 I2C 就作为 I2C Master
- ✧ STOP：请求 I2C IP 发送 STOP 信号
- ✧ ACK：如果收到数据的时候该位为 1，I2C IP 将回 ACK 给对方，否则回 NACK

其实从I2C的波形就可以看出，I2C协议完全是状态驱动的，从START信号开始，发送/接收地址字节之后收到ACK/NACK状态，发送/接收数据之后收到ACK/NACK状态，发送STOP

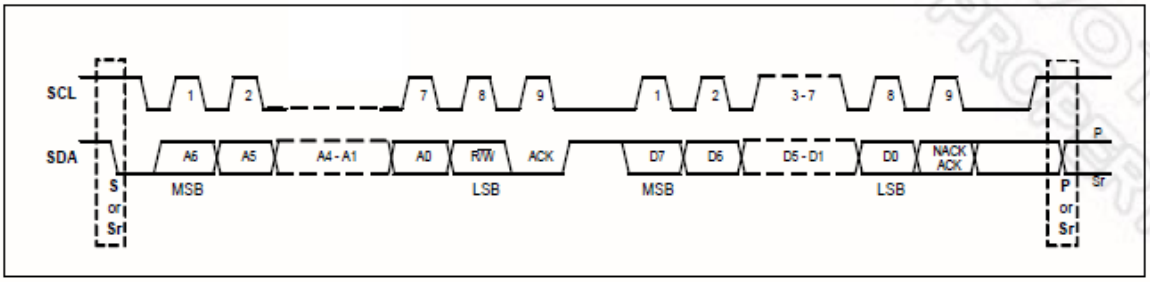


Figure 6-74 I²C Protocol

这个就是状态寄存器，状态一旦发生改变，就会发生I2C中断，然后读该寄存器就知道发生了何事。

STATUS

下图就是I2C作为Master和Slave时各个状态的含义表格：

Master Mode		Slave Mode	
STATUS	Description	STATUS	Description
0x08	Start	0xA0	Slave Transmit Repeat Start or Stop
0x10	Master Repeat Start	0xA8	Slave Transmit Address ACK
0x18	Master Transmit Address ACK	0xB0	Slave Transmit Arbitration Lost
0x20	Master Transmit Address NACK	0xB8	Slave Transmit Data ACK
0x28	Master Transmit Data ACK	0xC0	Slave Transmit Data NACK
0x30	Master Transmit Data NACK	0xC8	Slave Transmit Last Data ACK
0x38	Master Arbitration Lost	0x60	Slave Receive Address ACK
0x40	Master Receive Address ACK	0x68	Slave Receive Arbitration Lost
0x48	Master Receive Address NACK	0x80	Slave Receive Data ACK
0x50	Master Receive Data ACK	0x88	Slave Receive Data NACK
0x58	Master Receive Data NACK	0x70	GC mode Address ACK
0x00	Bus error	0x78	GC mode Arbitration Lost
		0x90	GC mode Data ACK
		0x98	GC mode Data NACK
0xF8	Bus Released Note: Status "0xF8" exists in both master/slave modes, and it won't raise interrupt.		

0xF8是总线空闲的状态值，也是STATUS寄存器的缺省值。

下面详细介绍一下上表的各个状态。

Master状态介绍：

1) 发送 START 信号成功，发生 I2C 中断，STATUS 寄存器的值=0x08
2) 作为 I2C Master 没有发送 STOP 又发送 START 信号成功，发生 I2C 中断，STATUS 寄存器的值=0x10
3) 发送地址+W 成功并收到 ACK，发生 I2C 中断，STATUS 寄存器的值=0x18
4) 发送地址+w 成功并收到 NACK，发生 I2C 中断，STATUS 寄存器的值=0x20
5) 发送数据成功并收到 ACK，发生 I2C 中断，STATUS 寄存器的值=0x28
6) 发送数据成功并收到 NACK，发生 I2C 中断，STATUS 寄存器的值=0x30
7) Master 发生仲裁失败，发生 I2C 中断，STATUS 寄存器的值=0x38
8) 发送地址+R 成功并收到 ACK，发生 I2C 中断，STATUS 寄存器的值=0x40
9) 发送地址+R 成功并收到 NACK，发生 I2C 中断，STATUS 寄存器的值=0x48
10) 收到数据并返回 ACK，发生 I2C 中断，STATUS 寄存器的值=0x50
11) 收到数据并返回 NACK，发生 I2C 中断，STATUS 寄存器的值=0x58
12) 总线错误，发生 I2C 中断，STATUS 寄存器的值=0x00

Slave状态介绍：

1) 收到 RE-START 信号或者 STOP 信号，发生 I2C 中断，STATUS 寄存器的值=0xA0
2) 收到 SLA+R 信号并返回 ACK，发生 I2C 中断，STATUS 寄存器的值=0xA8
3) 作为 Master 仲裁失败 HW 会自动转为 Slave，之后收到 SLA+R 信号，发生 I2C 中断，STATUS 寄存器的值=0xB0
4) 发送数据并收到 ACK，发生 I2C 中断，STATUS 寄存器的值=0xB8
5) 发送数据并收到 NACK，发生 I2C 中断，STATUS 寄存器的值=0xC0
6) 从接发送最后一个数据，但是居然收到的是 ACK，发生 I2C 中断，STATUS 寄存器的值=0xC8
7) 从接收到 SLA+W 并返回 ACK，发生 I2C 中断，STATUS 寄存器的值=0x60
8) 作为 Master 仲裁失败 HW 会自动转为 Slave，之后收到 SLA+W 信号，发生 I2C 中断，STATUS 寄存器的值=0x68
9) 收到数据并返回 ACK，发生 I2C 中断，STATUS 寄存器的值=0x80
10) 收到数据并返回 NACK，发生 I2C 中断，STATUS 寄存器的值=0x88
11) 广播模式收到 SLA+W 并返回 ACK，发生 I2C 中断，STATUS 寄存器的值=0x70
12) 广播模式仲裁失败，发生 I2C 中断，STATUS 寄存器的值=0x78
13) 广播模式收到数据并返回 ACK，发生 I2C 中断，STATUS 寄存器的值=0x90

知道所有的状态之后，配置好I2C，然后在中断里面查看STATUS寄存器，处理状态就可以了。超时中断是用户设定时间内没有发生状态改变，就会发生超时中断，这个可以防止软件被hold住。

下面是一个读/写EEPROM的例子。照样是先选择用到的IP的时钟源，使能各个IP的时钟，然后配置多功能引脚，然后是I2C IP功能初始化。另外是I2C 的中断处理函数。中断处理函数中根据STATUS寄存器的值，进行状态处理。只要调用函数I2C_SET_CONTROL_REG(I2C0, I2C_SI);清除状态改变标志，I2C IP就会自动开始下一个状态。

```
void I2C0_IRQHandler(void)
{
    uint32_t u32Status;

    /* 清除中断标志 */
    I2C0->INTSTS |= I2C_INTSTS_INTSTS_Msk;
    /*取得STATUS寄存器的值*/
    u32Status = I2C_GET_STATUS(I2C0);

    /*检查中断标志*/
    if (I2C_GET_TIMEOUT_FLAG(I2C0)) {/*发生超时中断*/
        /* 清除超时中断标志 */
        I2C_ClearTimeoutFlag(I2C0);
    } else {/*状态改变中断*/
        if (s_I2C0HandlerFn != NULL)
            s_I2C0HandlerFn(u32Status);
    }
}

/*  I2C 接收回调函数
*/
void I2C_MasterRx(uint32_t u32Status)
{
    if (u32Status == 0x08) {        /* START 被发送，准备SLA+W */
        I2C_SET_DATA(I2C0, (g_u8DeviceAddr << 1)); /* 写 SLA+W 到数据寄存器 I2CDAT */
        I2C_SET_CONTROL_REG(I2C0, I2C_SI); /*清除状态改变标志，I2C IP开始发送SLA+W*/
    } else if (u32Status == 0x18) {             /* SLA+W 已经发送并且收到ACK */
        I2C_SET_DATA(I2C0, g_au8TxData[g_u8DataLen++]);/*将要发送的数据写到I2CDAT寄存器*/
        I2C_SET_CONTROL_REG(I2C0, I2C_SI); /*清除状态改变标志，I2C IP开始发送数据 */
    } else if (u32Status == 0x20) {             /* SLA+W 已经被发送并且收到NACK */
        I2C_SET_CONTROL_REG(I2C0, I2C_STA | I2C_STO | I2C_SI); /*发送STOP并重新发送START信号*/
    } else if (u32Status == 0x28) {             /* DATA 已经被发送并且收到ACK */
        if (g_u8DataLen != 2) {
            I2C_SET_DATA(I2C0, g_au8TxData[g_u8DataLen++]);/*继续写数据到I2CDAT寄存器*/
            I2C_SET_CONTROL_REG(I2C0, I2C_SI); /*清除状态改变标志，I2C IP开始发送数据*/
```

```
        } else {
            I2C_SET_CONTROL_REG(I2C0, I2C_STA | I2C_SI); /*清除状态改变标志并再次发送START信号*/
        }
    } else if (u32Status == 0x10) {             /* Repeat START 已经被发送，准备SLA+R */
        I2C_SET_DATA(I2C0, (g_u8DeviceAddr << 1) | 0x01);  /* 写 SLA+R 到I2CDAT寄存器 */
        I2C_SET_CONTROL_REG(I2C0, I2C_SI); /*清除状态改变标志，I2C IP开始发送SLA+R*/
    } else if (u32Status == 0x40) {             /* SLA+R 已经发送并且收到ACK信号 */
        I2C_SET_CONTROL_REG(I2C0, I2C_SI); /*清除状态改变标志,I2C IP准备接收数据*/
    } else if (u32Status == 0x58) {             /* DATA 收到并且返回NACK */
        g_u8RxData = I2C_GET_DATA(I2C0);    /*读取数据*/
        I2C_SET_CONTROL_REG(I2C0, I2C_STO | I2C_SI); /*清除状态改变标志并发送STOP信号*/
        g_u8EndFlag = 1;
    } else {
        /* TO DO */
        printf("Status 0x%x is NOT processed\n", u32Status);
    }
}

/*  I2C 发送回调函数
*/
void I2C_MasterTx(uint32_t u32Status)
{
    if (u32Status == 0x08) {                    /* START 已经发送，准备SLA+W */
        I2C_SET_DATA(I2C0, g_u8DeviceAddr << 1);  /* 写 SLA+W 到寄存器 I2CDAT */
        I2C_SET_CONTROL_REG(I2C0, I2C_SI); /*清除状态改变标志，I2C IP开始发送SLA+W */
    } else if (u32Status == 0x18) {             /* SLA+W 已经发送并且收到ACK信号 */
        I2C_SET_DATA(I2C0, g_au8TxData[g_u8DataLen++]);/*将要发送的数据写到I2CDAT寄存器*/
        I2C_SET_CONTROL_REG(I2C0, I2C_SI); /*清除状态改变标志，I2C IP开始发送数据*/
    } else if (u32Status == 0x20) {             /* SLA+W 已经发送但是收到NACK */
        I2C_SET_CONTROL_REG(I2C0, I2C_STA | I2C_STO | I2C_SI); /*发送STOP并重新发送START信号*/
    } else if (u32Status == 0x28) {             /* DATA已经发送并收到ACK */
        if (g_u8DataLen != 3) {
            I2C_SET_DATA(I2C0, g_au8TxData[g_u8DataLen++]);/*将要发送的数据写到I2CDAT寄存器*/
            I2C_SET_CONTROL_REG(I2C0, I2C_SI); /*清除状态改变标志，I2C IP开始发送数据*/
        } else {
            I2C_SET_CONTROL_REG(I2C0, I2C_STO | I2C_SI); /*清除状态改变标志并发送STOP信号*/
            g_u8EndFlag = 1;
        }
    } else {
        /* TO DO */
        printf("Status 0x%x is NOT processed\n", u32Status);
```

```
    }
}

void SYS_Init(void)
{
    /* Init System Clock */
    /* 解锁保护寄存器 */
    SYS_UnlockReg();

    /* 使能 12MHz HXT, 32KHz LXT 和 HIRC */
    CLK_EnableXtalRC(CLK_PWRCTL_HXT_EN_Msk | CLK_PWRCTL_LXT_EN_Msk |
CLK_PWRCTL_HIRC_EN_Msk);

    /* 等待晶振稳定 */
    CLK_WaitClockReady(CLK_CLKSTATUS_HXT_STB_Msk | CLK_CLKSTATUS_LXT_STB_Msk |
CLK_CLKSTATUS_HIRC_STB_Msk);

    /*  使能PLL，并且 HCLK 时钟源切为PLL */
    CLK_SetCoreClock(32000000);

    /* 选择IP 时钟源，I2C时钟源只能是HCLK */
    CLK_SetModuleClock(UART0_MODULE, CLK_CLKSEL1_UART_S_HIRC, CLK_UART_CLK_DIVIDER(1));
    CLK_SetModuleClock(I2C0_MODULE, 0, 0);

    /* 使能 IP 时钟 */
    CLK_EnableModuleClock(UART0_MODULE);
    CLK_EnableModuleClock(I2C0_MODULE);

    /* Update System Core Clock */
    /* User can use SystemCoreClockUpdate() to calculate PllClock, SystemCoreClock and
CycylesPerUs automatically. */
    SystemCoreClockUpdate();

    /* Init I/O Multi-function */
    /* 配置 PB 作为 UART0 RXD and TXD  */
    SYS->PB_L_MFP &= ~(SYS_PB_L_MFP_PB0_MFP_Msk | SYS_PB_L_MFP_PB1_MFP_Msk);
    SYS->PB_L_MFP |= (SYS_PB_L_MFP_PB0_MFP_UART0_TX | SYS_PB_L_MFP_PB1_MFP_UART0_RX);

    /* 配置PC0和PC1用作 I2C0 */
    SYS->PC_L_MFP = (SYS_PC_L_MFP_PC0_MFP_I2C0_SCL | SYS_PC_L_MFP_PC1_MFP_I2C0_SDA);

    /* 重新加锁 */
```

```
    SYS_LockReg();
}

void I2C0_Init(void)
{
    /* 打开 I2C0 并设置波特率为100k */
    I2C_Open(I2C0, 100000);

    /* 打印I2C0 总线时钟 */
    printf("I2C clock %d Hz\n", I2C_GetBusClockFreq(I2C0));

    /* 设置2个 I2C0 从机地址，如果做从机的话，就用这个函数设置从接地址 */
    I2C_SetSlaveAddr(I2C0, 0, 0x15, I2C_GCMODE_DISABLE);   /* Slave Address : 0x15 */
    I2C_SetSlaveAddr(I2C0, 1, 0x35, I2C_GCMODE_DISABLE);   /* Slave Address : 0x35 */

    /* 使能 I2C0 中断 */
    I2C_EnableInt(I2C0);
    NVIC_EnableIRQ(I2C0_IRQn);
}
/*  Main Function
*/
int32_t main (void)
{
    uint32_t i;

    /* Init System, IP clock and multi-function I/O */
    SYS_Init();

    /* Init UART to 115200-8n1 for print message */
    UART_Open(UART0, 115200);

    /*
        该例程设置I2C 总线时钟为100kHz. 然后, 访问 EEPROM 24LC64 进行Byte Write
       和Byte Read 操作, 并检查读到的数据是否等于写入的数据.
    */

    printf("+-----------------------------------------------------+\n");
    printf("|    Nano1x2 Series I2C Sample Code with EEPROM 24LC64  |\n");
    printf("+-----------------------------------------------------+\n");

    /* Init I2C0 to access EEPROM */
```

```
    I2C0_Init();

    /* EEPROM从机地址为0x50 */
    g_u8DeviceAddr = 0x50;

    /*地址0写入3，地址1写入4，然后读回比较*/
    for (i = 0; i < 2; i++) {
        g_au8TxData[0] = (uint8_t)((i & 0xFF00) >> 8);
        g_au8TxData[1] = (uint8_t)(i & 0x00FF);
        g_au8TxData[2] = (uint8_t)(g_au8TxData[1] + 3);

        g_u8DataLen = 0;
        g_u8EndFlag = 0;

        /* 写数据到EEPROM */
        s_I2C0HandlerFn = (I2C_FUNC)I2C_MasterTx;

        /* I2C 作为Master发送 START信号 */
        I2C_SET_CONTROL_REG(I2C0, I2C_STA);

        /* 等待 I2C 发送完成 */
        while (g_u8EndFlag == 0);
        g_u8EndFlag = 0;

        /* 从EEPROM读数据 */
        s_I2C0HandlerFn = (I2C_FUNC)I2C_MasterRx;

        g_u8DataLen = 0;
        g_u8DeviceAddr = 0x50;

        /* I2C 作为Master发送 START信号 */
        I2C_SET_CONTROL_REG(I2C0, I2C_STA);

        /* 等待I2C 接收完成 */
        while (g_u8EndFlag == 0);

        /* 比较数据 */
        if (g_u8RxData != g_au8TxData[2]) {
            printf("I2C Byte Write/Read Failed, Data 0x%x\n", g_u8RxData);
            return -1;
        }
```

```
    }
    printf("I2C Access EEPROM Test OK\n");

    return 0;
}
```

总结上面I2C_MasterRx的流程为：

1) I2C_SET_CONTROL_REG(I2C0, I2C_STA); 发送 START 信号，之后该 I2C IP 就作为 Master
2) 发生 I2C 中断，STATUS=0x08，表示 START 信号发送成功，然后软件写 I2CDAT 寄存器，发送 I2C 地址+W 给 EEPROM
3) 发生 I2C 中断，STATUS=0x18，表示 SLA+W 发送成功并收到 ACK，然后发送高位地址给 EEPROM
4) 发生 I2C 中断，STATUS=0x28，表示高位地址发送成功并收到 ACK，然后发送低位地址给 EEPROM
5) 发生 I2C 中断，STATUS=0x28，表示低位地址发送成功并收到 ACK，然后再次发送 START 信号——Repeat START
6) 发生 I2C 中断，STATUS=0x10，表示 Repeat START 发送成功，然后发送 I2C 地址+R 给 EEPROM
7) 发生 I2C 中断，STATUS=0x40，表示 SLA+R 发送成功，并收到 ACK，然后不设 ACK bit，将回 NACK 给 EEPROM
8) 发生 I2C 中断，STATUS=0x58，收到 EEPROM 返回的数据并回 NACK 给 EEPROM

把这个流程对应I2C_MasterRx函数仔细看一下，相信不难懂。

发送的流程也类似，在此就不再赘述。

3.8 I2S 初始化

新唐的I2S接口外接audio codec，例如：接新唐的NAU8822，NAU8810等。一般支持I2S数据格式和MSB Justified数据格式，有一些支持PCM mode A和PCM mode B格式。可以作为I2S Master和I2S slave，一般情况下都是作为I2S slave使用，不然播放不同采样率的音频，MCLK引脚出不同的频率给audio codec比较困难。

I2S 接口如下，共5根脚：

→ I2SMCLK
← I2SLRCLK
→ I2SDO
← I2SBCLK
← I2SDI

- ✧ I2SMCLK：由 I2S Master 提供。I2S 做 Master 时提供 MCLK 给 audio codec，它应该=采样频率*2*256，例如：如果采样频率=44.1K 则 MCLK 应该给=44.1*2*256 = 22.5792M
- ✧ LRCLK：由 I2S Master 提供。是左右声道的时钟信号
- ✧ I2SDO：I2S 数据输出
- ✧ I2SBCLK：由 I2S Master 提供。I2S 收发数据时每个 bit 的时钟。它应该=分辨率*采样率*通道数。例如: 16bit， 48K，stereo 的数据，BCLK 输出的频率=16×48×2 = 1536K
- ✧ I2SDI：I2S 数据输入

下面的波形是MSB Justified的波形，左声道LRCLK拉high, 右声道LRCLK拉low。MSB优先，BCLK上升沿latch数据。各种不同的I2S格式，波形也是不同的。

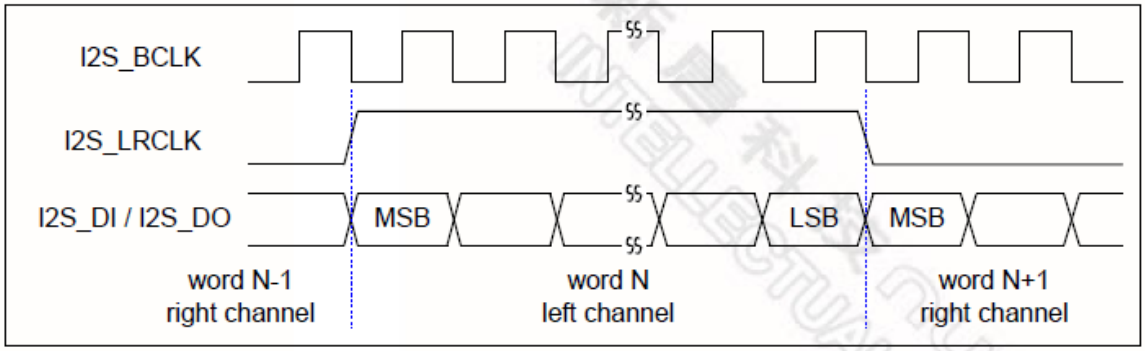


Figure 6.16-26 MSB Justified Data Format Timing Diagram

下面以M451的I2S 为例说明I2S的用法。M451 I2S和SPI1两个IP共用逻辑电路，所以I2S的说明放在SPI IP里面。SPI1和SPI2可以作为2个I2S接口使用，而SPI0是单纯的SPI。

该I2S发送和接收各有4级32bit FIFO，一般我们设定TX/RX 阀值中断是2。对于TX来说，当发送FIFO中有效数据<=2时，会发生TX 阀值中断；对于RX来说，当接收FIFO中的有效数据>=2时，会发生RX 阀值中断。

也就是将该4级FIFO分成2块做ping-pong。对于发送来说先把4个FIFO都填满，然后发生TX threshold中断时表示第一块已经发送出去，可以填2笔数据到TX FIFO中，此时TX在发送第二块；再次发生TX threshold中断时，表示第二块已经发送出去，可以填2笔数据到TX FIFO中。对于RX也是一样的，每次发生RX 阀值中断，可以读两笔数据出来。

下面的demo code， I2S作为slave，数据宽度16bit，stereo，I2S 波形。采样率由I2S Master决定。只demo I2S可以收发数据而已，如果连接了audio code，还要通过I2C去初始化audio codec。我们的M451 BSP里面目录SampleCode\StdDriver下面有2个demo ：I2S_Master和I2S_Slave。将两片

M451板子的I2S对接，一个跑I2S_Master，一个跑I2S_Slave，就可以测试I2S的特性。

```
void I2S_Init(void)
{
    uint32_t u32RxValue1, u32RxValue2;

    /* SPI1选择PCLK1作为时钟源，默认PCLK1的时钟等于HCLK的时钟 */
    CLK_SetModuleClock(SPI1_MODULE, CLK_CLKSEL2_SPI1SEL_PCLK1, MODULE_NoMsk);

    /* 使能SPI1的时钟 */
    CLK_EnableModuleClock(SPI1_MODULE);
    /* 配置 SPI1 相关引脚 */
    /* GPA[7:4] : SPI1_CLK (I2S1_BCLK), SPI1_MISO (I2S1_DI), SPI1_MOSI (I2S1_DO), SPI1_SS (I2S1_LRCLK) */
    SYS->GPA_MFPL &= ~(SYS_GPA_MFPL_PA4MFP_Msk | SYS_GPA_MFPL_PA5MFP_Msk |
SYS_GPA_MFPL_PA6MFP_Msk | SYS_GPA_MFPL_PA7MFP_Msk);
    SYS->GPA_MFPL |= (SYS_GPA_MFPL_PA4MFP_SPI1_SS | SYS_GPA_MFPL_PA5MFP_SPI1_MOSI |
SYS_GPA_MFPL_PA6MFP_SPI1_MISO | SYS_GPA_MFPL_PA7MFP_SPI1_CLK);

     /* I2S1使用SPI1接口，配置为16bit， Stereo，I2S format，使能TX和RX FIFO threshold中断，连续发送数据0xAA00AA01,
0xAA02AA03, ..., 0xAAFEAAFF 每个发送5000次，然后改变TX的值*/
    /* I2S 外设时钟频率等于 PCLK1 的时钟频率 */
    I2S_Open(SPI1, I2S_MODE_SLAVE, 0, I2S_DATABIT_16, I2S_STEREO, I2S_FORMAT_I2S);

    /* 初始化发送数据计数 */
    g_u32DataCount = 0;
    /* 初始化发送/接收数据 */
    g_u32TxValue = 0xAA00AA01;
    u32RxValue1 = 0;
    u32RxValue2 = 0;
    /* 使能TX/RX threshold 中断 */
    I2S_EnableInt(SPI1, I2S_FIFO_TXTH_INT_MASK| I2S_FIFO_RXTH_INT_MASK);
    NVIC_EnableIRQ(SPI1_IRQn);

    while(1)
    {
        /* 改变要发送的数据 */
        if(g_u32DataCount >= 50000)
        {
            g_u32TxValue = 0xAA00AA00 | ((g_u32TxValue + 0x00020002) & 0x00FF00FF); /*
g_u32TxValue: 0xAA00AA01, 0xAA02AA03, ..., 0xAAFEAAFF */
            printf("TX value: 0x%X\n", g_u32TxValue);
            g_u32DataCount = 0;
        }
```

```
    }
}

uint32_t g_au32PcmRecBuf[96];
/*I2S1 中断处理函数*/
void SPI1_IRQHandler()
{
   uint32_t u32I2SIntFlag;
    uint32_t i, u32Idx;

    u32I2SIntFlag = SPI1->I2SSTS;
    /*发生Tx Threshold中断*/
    if(u32I2SIntFlag & SPI_I2SSTS_TXTHIF_Msk)
    {
         /* 写2 笔TX 值到 TX FIFO */
        I2S_WRITE_TX_FIFO(SPI1, g_u32TxValue);
        I2S_WRITE_TX_FIFO(SPI1, g_u32TxValue);
        g_u32DataCount += 2;
    }
    /*发生Rx Threshold中断*/
    if(u32I2SIntFlag & SPI_I2SSTS_RXTHIF_Msk)
    {
         g_au32PcmRecBuf[g_u32RecPos++] = I2S_READ_RX_FIFO(SPI1);
         g_au32PcmRecBuf[g_u32RecPos++] = I2S_READ_RX_FIFO(SPI1);
         if(g_u32RecPos >= 96)
              g_u32RecPos = 0;
    }
}
```

上面的demo，通过I2S发送一些数据出去，就是说I2S不一定非得传输音频数据，也可以拿来单纯的传输数据用。通过RX收到的数据我们也没有做进一步的处理，只是放在变量g_au32PcmRecBuf中。如果做USB声卡应用，g_au32PcmRecBuf里面的数据可以通过USB发送到PC上。

3.9 LCD 初始化

这里的LCD指的是段式LCD驱动IP，新唐的NANO100系列和NANO112都有带段式屏的驱动。一般最多6个COM，40个SEG。驱动方式有R-Type(电阻分压)，C-Type（电荷泵），以及Ext_C-Type(外部电容分压模式)。

- ✧ C-Type 优点是可以保持电压，即使系统 VDD 已经掉到段式屏期望电压以下，V1/V2/V3 输出的电压仍可以保持，缺点是比较耗电
- ✧ R-Type 和 Ext_C-Type 都不能保持电压，就是说当系统 VDD 往下掉的是，LCD IP 打出的波形电压也会跟着往下掉。R-Type 是在 V1/V2/V3 上接电阻分压，缺点是因为到低一直有个通路，将 always 漏电。优点是 IP 内部有带分压电阻，外面引脚可以不用再接电阻了，但是仍需要接电容到地。
- ✧ Ext_C-Type 是最省电的方式，在 V/V2/V3 接电容到地即可。这种模式不会漏电。

在TRM（技术参考手册）里面有各个模式的参考电路，大家一看就明白了。

```
void LCD_Init(void)
{
    /* LCD使用外部32K做时钟源，需要使能晶振 */
    CLK->PWRCTL |= (0x1 << CLK_PWRCTL_LXT_EN_Pos); // LXT Enable

    /* 等待晶振稳定 */
    CLK_WaitClockReady(CLK_CLKSTATUS_LXT_STB_Msk);

    /* 使能LCD的时钟 */
    CLK->APBCLK |= CLK_APBCLK_LCD_EN;

    /* 配置LCD多功能引脚： COMs, SEGs, V1 ~ V3, DH1, DH2，并关闭相应引脚的数字通路 */
    MFP_LCD_TYPEA();

    /* LCD 初始化，使用C-Type, 4个COM，1/3 bias，频率除以64，充电泵充电到3V */
    LCD_Open(LCD_C_TYPE, 4, LCD_BIAS_THIRD, LCD_FREQ_DIV64, LCD_CPVOl_3V);

    LCD_EnableDisplay();
    /*全屏点亮*/
    LCD->MEM_0 = 0x3F3F3F3F;
    LCD->MEM_1 = 0x3F3F3F3F;
    LCD->MEM_2 = 0x3F3F3F3F;
    LCD->MEM_3 = 0x3F3F3F3F;
    LCD->MEM_4 = 0x3F3F3F3F;
    LCD->MEM_5 = 0x3F3F3F3F;
    LCD->MEM_6 = 0x3F3F3F3F;
    LCD->MEM_7 = 0x3F3F3F3F;
    LCD->MEM_8= 0x3F3F3F3F;
}
```

如果LCD会闪，或者某些地方特别黑/特别淡，这个一般有2个原因：LCD频率太快、给LCD的电压太高/太低

一般查看如下几点：

- ✧ 频率太快，可以调整 LCD_FREQ_DIV64 的值
- ✧ LCD 的电压太高，如果是 C-Type 就调整一下充电泵的电压，如果是 R-Type/Ext_C-Type 调整一下 VLCD 的电压。
- ✧ 另外量一下 V1/V2/V3 输出的电压是不是 1/3 VLCD，2/3VLCD，VDD 和 GND，如果不是，可能 V1/V2/V3 外面没有接电容到地，所以电压稳不住。接上电容应该就可以了。

3.10 PWM 初始化

PWM就是脉冲宽度调制。说白了就是可以输出波形，波形的频率、高电平宽度/低电平宽度都可以调节。PWM IP的功能很多，除了输出基本的波形，还能捕获、输出互补的波形、中心对齐的波形或者触发ADC进行采样，以及刹车等等功能。

新唐芯片的PWM有的多达24路，频率可以到100M或者更高。

下面的demo，只demo简单的PWM输出波形的功能。从PWM0的通道0输出100HZ的波形，高电平占30%。要使用更复杂的功能，需要详细研读TRM(技术参考手册)

```
int32_t PWM_init (void)
{
    /* 使能PWM0通道0/1的时钟 */
    CLK_EnableModuleClock(PWM0_CH01_MODULE);
    /* PWM0 IP 时钟源选择HCLK */
    CLK_SetModuleClock(PWM0_CH01_MODULE, CLK_CLKSEL1_PWM0_CH01_S_HCLK, 0);

    /* 配置 PB8 和 PB9用作PWM0的通道0和通道1 */
    SYS->PB_H_MFP = (SYS->PB_H_MFP & ~(SYS_PB_H_MFP_PB8_MFP_Msk |
                SYS_PB_H_MFP_PB9_MFP_Msk)) | SYS_PB_H_MFP_PB8_MFP_PWM0_CH0 |
                SYS_PB_H_MFP_PB9_MFP_PWM0_CH1;

    /* PWM0 frequency is 100Hz, duty 30% */
    PWM_ConfigOutputChannel(PWM0, 0, 100, 30);
    /* 使能PWM0的通道0输出功能 */
    PWM_EnableOutput(PWM0, PWM_CH_0_MASK);
    /* 开始输出波形 */
    PWM_Start(PWM0, PWM_CH_0_MASK);

    while(1);
}
```

之后从PB8就会量到100HZ的波形，占空比30%。

函数PWM_ConfigOutputChannel(PWM0, 0, 100, 30); 通过修改下面四个寄存器实现功能

/*PWM0选择HCLK做时钟源，假设HCLK工作在32M，预分频之后PWM0频率=1M*/

PWM0->PRES = (PWM0->PRES&~PWM_PRES_CP01_Msk) | 0x1F;//divided by (CP01 + 1)

PWM0->CLKSEL = PWM_CLK_DIV_1<<PWM_CLKSEL_CLKSEL0_Pos;//通道0输入时钟再除以1

/* PWM0 Timer 0工作在连续模式，将会连续输出PWM波形*/

PWM0->CTL = PWM_CTL_CH0MOD_Msk;//continuous mode

/*设置频率和占空比。频率=1M/10000 = 100HZ*/

PWM0->DUTY0 = 3000<< PWM_DUTY0_CM_Pos| 10000;

我觉得直接修改寄存器更有弹性，不过就是要熟悉这四个寄存器。

- ✧ PRES 就是 PWM 的预分频寄存器。PWM 的时钟源经过 PRES 预分频之后传给 PWM0 的 CLKSEL 寄存器
- ✧ PWM0 的 CLKSEL 可以再次进行分频，所以 PWM 可以输出很慢的频率
- ✧ CTL 寄存器用于设定 PWM Timer 工作在 one-shot 模式还是连续模式，以及波形是否要反转等等
- ✧ DUTY 寄存器用于设定 PWM 输出波形的频率和占空比

3.11 RTC 初始化

RTC实时时钟，就是用来显示时/分/秒、年/月/日/星期等信息的。上电之后，初始化完成，只要不断电，它就一直自动记录当前时间。任何时候都可以从它内部的寄存器读出当前的时间和日期

RTC一般使用外部32K做时钟源，RTC准不准，关键就看32K晶振是否准了。一般32K晶振都会有一定的误差。如果希望RTC非常准，32K晶振就要进行补偿。新唐的RTC一般都带频率补偿寄存器。补偿值需要软件自行填入。一般一批32K晶振误差都差不多，使用频率计数器测一下晶振，然后将补偿值填入RTC就可以了。补偿值具体应该填多少，TRM（技术参考手册）里面RTC部分介绍的很详细，这里就不再赘述。

RTC除了做时钟，还能产生Tick中断和Alarm中断，并能唤醒系统。

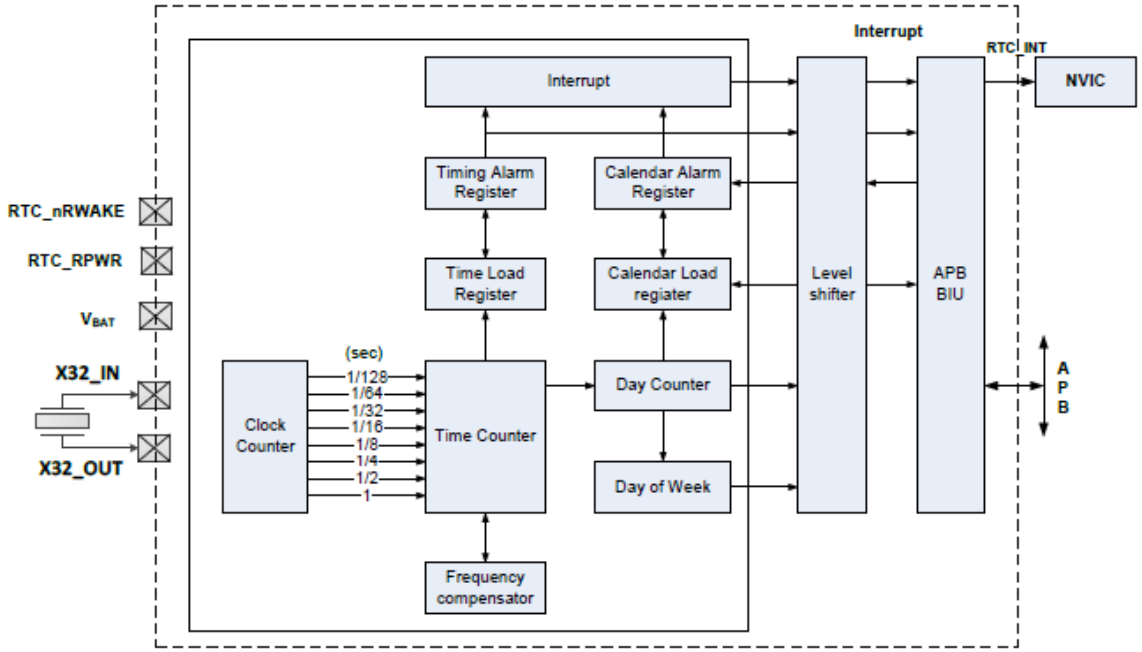
RTC中有两个特别的寄存器INIR和AER，上电之后需要写0xa5eb1357到该寄存器，让RTC退出复位状态；AER是读/写使能寄存器，默认RTC的大多寄存器只能读，有的读都不可以。写0xA965到该寄存器可以解锁寄存器读/写 512 ～ 1024个 RTC时钟周期不等，不同的芯片这块规定不同。

RTC的时钟源为32K，512个clock就是16ms，一般设定RTC功能，这个时间都够用了。

3.11.1 NUC505 的 RTC

NUC505的RTC需要特别介绍一下，因为这个RTC支持按键开机/关机功能。

下图是RTC的方块图，大家注意左边，除了Vbat脚和32K晶振脚之外，RTC还有2个脚：RTC_nRWAKE、RTC_RPWER。



这两个脚的作用描述如下： nRWAKE拉low一定时间（时间长度由RTC_FREQADJ寄存器的PKEYTIME[26:24]决定，这个寄存器下面会介绍），RPWR就会输出high，RPWR可以接到片外LDO，使能LDO给整个芯片供电。

下面介绍一下RTC几个特别的寄存器。

3.11.1.1 POWCTL寄存器

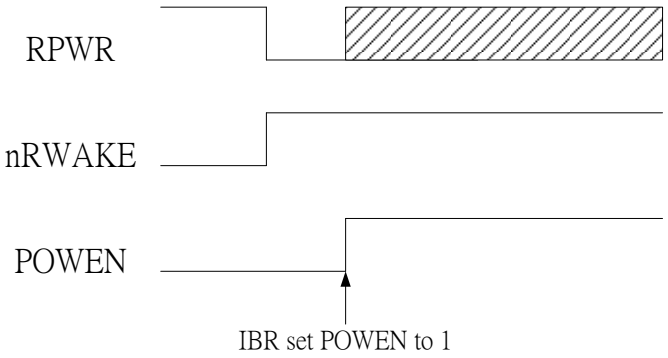
RTC Power Time-out Register (RTC_POWCTL)

Register	Offset	R/W	Description	Reset Value
RTC_POWCTL	RTC_BA+0x034	R/W	RTC Power Time-out Register	0x0005_0000

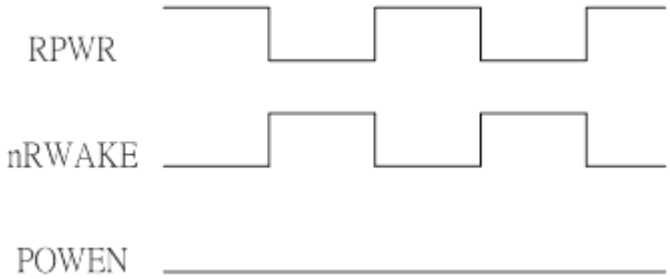
31	30	29	28	27	26	25	24
RALMTIME							
23	22	21	20	19	18	17	16
RALMTIME				POWOFFT			
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
POWKEY	Reserved	EDGE_TRIG	RALMIEN	ALMIEN	POWOFFEN	SWPOWOFF	POWEN

这个寄存器用于控制系统上电和断电。

- POWEN：开机时由 IBR(NUC505 内部有一块 ROM 用于存放启动代码，用户不能修改该区域)设为 1，使得 USER 在放开 nRWAKE 后仍保持供电 (RPWR 保持为 High)



开机时如果POWEN没有设置为1， USER在放开nRWAKE后(high)，RPWR就恢复为Low。



一般来说nRWAKE为Low，RPWR则为High (两种关机情况例外- 软件关机&硬件关机)

- SWPOWOFF 和 POWOFFEN 这两个 bit：SWPOWOFF 用于软件关机，POWOFFEN 用于硬件关机，一般就是长按关机键关机。
  - 只要 SWPOWOFF=1， POWEN=0 既可以实现软件关机。
    SWPOWOFF 设为 1 时，即使 nRWAKE 为 Low，RPWR 仍会为 Low。如果希望 nRWAKE 为 low 时先不要关机等 nRWAKE 为 high 时再关，则只要 POWEN=0 就行了，SWPOWOFF 不要置为 1。
  - POWOFFEN 与 POWOFFT 是搭配使用的，功能类似 Notebook 长按电源键强迫关机 (是由硬件控制关机)。
    POWOFFEN:使能长按强制关机功能
    POWOFFT : 长按时间= (POWOFFT + 3)秒
    长按 nRWAKE 键(POWOFFT + 3)秒后，会强制关机

- ALMIEN 和 RALMIEN 用于配置 Alarm 功能。Alarm 可以设定绝对时间 alarm 或者相对时间 alarm。
  - ALMIEN：绝对时间 Alarm

■ RALMIEN：相对时间 Alarm

RTC可以从Deep Sleep或者Power Off唤醒系统，但是只有Alarm可以唤醒，Tick不行（只有NUC505 Tick不能唤醒系统，其它的例如：NUC100, NANO都可以唤醒）。如果是从Power off唤醒系统，则不会发生Alarm中断，但是Alarm 中断标志会被置。

总结：系统进入power off mode有2种方法

1、软件方法：SWPOWOFF = 1， PWREN = 0

2、硬件方法：长按 nRWAKE 按键关机

3、长按 nRWAKE 开机时间由下面的寄存器中的 PKEYTIME 决定，默认 0.25s

Register	Offset	R/W	Description	Reset Value
RTC_FREQADJ	RTC_BA+0x008	R/W	RTC Frequency Compensation Register	0x007F_FF00

31	30	29	28	27	26	25	24
ADJTRG	Reserved	Reserved	Reserved	PKEYTIME			
23	22	21	20	19	18	17	16
INTEGER							
15	14	13	12	11	10	9	8
INTEGER							
7	6	5	4	3	2	1	0
Reserved		FRACTION					

[27:24]	PKEYTIME	Minimum Duration That Power Key Must Be Pressed to Turn On Core Power Minimum power key duration = 0.25*(PKEYTIME+1) sec.

4、相对 alarm 时间就是填 RALMTIME，单位 s

31	30	29	28	27	26	25	24
RALMTIME							
23	22	21	20	19	18	17	16
RALMTIME				POWOFFT			
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
POWKEY	Reserved	EDGE_TRIG	RALMIEN	ALMIEN	POWOFFEN	SWPOWOFF	POWEN

3.11.1.2 Alarm设定寄存器

绝对时间 Alarm 可以填日期和时间，寄存器如下。这两个寄存器除了日期和时间之外还有 6 个 Mask bit，如红框所示

RTC Time Alarm Register (RTC_TALM)

Register	Offset	R/W	Description	Reset Value
RTC_TALM	RTC_BA+0x01C	R/W	RTC Time Alarm Register	0x0000_0000

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
MSKHR	Reserved	TENHR		HR			
15	14	13	12	11	10	9	8
MSKMIN	TENMIN			MIN			
7	6	5	4	3	2	1	0
MSKSEC	TENSEC			SEC			

RTC Calendar Alarm Register (RTC_CALM)

Register	Offset	R/W	Description	Reset Value
RTC_CALM	RTC_BA+0x020	R/W	RTC Calendar Alarm Register	0x0000_0000

31	30	29	28	27	26	25	24
MSKWEEKDAY	WEEKDAY			Reserved			MSKYEAR
23	22	21	20	19	18	17	16
TENYEAR				YEAR			
15	14	13	12	11	10	9	8
MSKMON	Reserved		TENMON	MON			
7	6	5	4	3	2	1	0
MSKDAY	Reserved	TENDAY		DAY			

例如 MSKDAY bit 定义如下

[7]	MSKDAY	Mask Alarm by Day 0 = Activate. 1 = Mask.

这个bit置1，就是不care DAY的意思，就是设置的Alarm与哪天无关。

3.11.2 RTC 代码

下面的代码设定RTC时间和日期，并使能每秒Tick一次的中断。

```
void RTC_TickHandle(void)
{
    S_RTC_TIME_DATA_T sCurTime;

    /* 取得当前时间并打印 */
    RTC_GetDateAndTime(&sCurTime);

    printf(" Current
Time:%d/%02d/%02d %02d:%02d:%02d\n",sCurTime.u32Year,sCurTime.u32Month,sCurTime.u32Day,sCu
rTime.u32Hour,sCurTime.u32Minute,sCurTime.u32Second);

    g_u32TICK++;
}
/*RTC中断处理函数*/
void RTC_IRQHandler(void)
{
    /*发生Tick 中断*/
    if ( (RTC->RIER & RTC_RIER_TIER_Msk) && (RTC->RIIR & RTC_RIIR_TIF_Msk) ) {
        RTC->RIIR = 0x2;
        RTC_TickHandle();
    }
}
int32_t RTC_Init(void)
{
    S_RTC_TIME_DATA_T sInitTime;

    /*使能外部低速晶振LXT (32 KHz) */
    CLK->PWRCTL |= CLK_PWRCTL_LXT_EN_Msk;
    /* 等待外部32K晶振稳定 */
    CLK_WaitClockReady(CLK_CLKSTATUS_LXT_STB_Msk);
    /* 使能RTC的时钟 */
    CLK->APBCLK |= CLK_APBCLK_RTC_EN_Msk;

    /* RTC初始化为2015年7月9日12点30分0秒，礼拜四，24小时模式 */
    sInitTime.u32Year       = 2015;
    sInitTime.u32Month      = 7;
    sInitTime.u32Day        = 9;
    sInitTime.u32Hour       = 12;
```

```
    sInitTime.u32Minute      = 30;
    sInitTime.u32Second      = 0;
    sInitTime.u32DayOfWeek   = RTC_TUESDAY;
    sInitTime.u32TimeScale   = RTC_CLOCK_24;

    RTC_Open(&sInitTime);

    /* 设置Tick周期 */
    RTC_SetTickPeriod(RTC_TICK_1_SEC);

    /* 使能RTC Tick中断 */
    RTC_EnableInt(RTC_RIER_TIER_Msk);
    NVIC_EnableIRQ(RTC_IRQn);

    g_u32TICK = 0;
}
```

RTC使能函数如上，因为它需要用32K做时钟源，所以需要先使能32K时钟。RTC只能选择32K做时钟源。然后使能RTC时钟。然后就可以初始化时间和日期，并配置中断。

其它M0/M4的RTC都比NUC505的功能简单多了，只能设定时间、日期、alarm、Tick功能。但是使用方法和NUC505是一样的，除了没有控制开机/关机功能。

3.12 SPI 初始化

新唐的SPI IP除了有常用的4线1-bit模式，有的还有two-bit和Quad Mode。做slave时还可以不用片选信号SS，就是常说的3-线模式。另外，片选信号有的芯片有2根SS0和SS1。

SPI一般4根线SPI_SS、SPI_CLK、SPI_MOSI、SPI_MISO

- Two-Bit 模式有 2 根输出脚 MOSI0 和 MOSI1，以及 2 根输入脚 MISO0 和 MISO1，它们可以分别连到不同的 SPI slave 上，实现同一根片选，相同的 CLK，发送不同的数据到不同的设备上去
- Qual Mode 就是常说的 4-bit 模式，它比 1-bit 速度快。是否可以使用 4-bit 模式，一般由外接的 SPI slave 设备决定

SPI的CLK频率最快一般可以达到30M，不同的芯片有点差异，有的快些有的慢些。

另外，SPI还支持使用PDMA收/发数据，或者用FIFO收/发数据。FIFO一般2级~8级。其实测试下来FIFO模式收/发数据比PDMA快，因为SPI可以连续接收数据到FIFO中，SPI总线上的数据之间可以没有间隔，只要软件来得及读走，这个速度可以达到SPI的极限；但是PDMA收/发需要时间，SPI收到数据，通知PDMA，然后PDMA读走数据，再通知SPI可以触发下一笔了，这之

间需要很多个SPI IP时钟。目前测试最慢需要23个PCLK时钟。

所以如果要将SPI速度调到最大建议使用FIFO模式，如果SW要做其它事情，来不及收数据，PDMA+FIFO是很好的选择。

下面的代码将SPI配置为SPI_MASTER，MSB优先，波形为 SPI_MODE_0，每笔数据 32bit长，SPI_CLK频率2M。写SPI TX寄存器就会将数据发送出去，读SPI RX寄存器就会得到从MISO引脚收到的数据。可以将SPI0接口的MISO0和MOSI0对接，这样发送出去的数据就会从RX寄存器读到。

所谓波形就是设定CLK idle 极性，上升沿/下降沿发送，上升沿/下降沿锁存。

SPI_MODE_0就是CLK默认低电平，下降沿发送，上升沿锁存。

按照SPI规格定义，SPI总共有4种波形，其它波形也都大同小异。

```
void SYS_Init(void)
{
    /* Init System Clock */
    /* 解锁保护寄存器 */
    SYS_UnlockReg();
    /* 使能外部 12MHz HXT, 32KHz LXT 和 HIRC */
    CLK_EnableXtalRC(CLK_PWRCTL_HXT_EN_Msk | CLK_PWRCTL_LXT_EN_Msk |
CLK_PWRCTL_HIRC_EN_Msk);

    /* 等待晶振稳定 */
    CLK_WaitClockReady(CLK_CLKSTATUS_HXT_STB_Msk | CLK_CLKSTATUS_LXT_STB_Msk |
CLK_CLKSTATUS_HIRC_STB_Msk);

    /*  使能PLL到32M，并且HCLK选择PLL做时钟源 */
    CLK_SetCoreClock(32000000);

    /* UART0选HIRC做时钟源，SPI0选HCLK做时钟源 */
    CLK_SetModuleClock(UART0_MODULE, CLK_CLKSEL1_UART_S_HIRC, CLK_UART_CLK_DIVIDER(1));
    CLK_SetModuleClock(SPI0_MODULE, CLK_CLKSEL2_SPI0_S_HCLK, 0);

    /* 使能UART0和SPI0 IP的时钟 */
    CLK_EnableModuleClock(UART0_MODULE);
    CLK_EnableModuleClock(SPI0_MODULE);

    /* Update System Core Clock */
    /* User can use SystemCoreClockUpdate() to calculate PllClock, SystemCoreClock and
CycylesPerUs automatically. */
    SystemCoreClockUpdate();

    /* 初始化 I/O 多功能引脚 */
```

```
    /* 将 PB0和PB1配置为 UART0 RXD 和 TXD  */
    SYS->PB_L_MFP &= ~(SYS_PB_L_MFP_PB0_MFP_Msk | SYS_PB_L_MFP_PB1_MFP_Msk);
    SYS->PB_L_MFP |= (SYS_PB_L_MFP_PB0_MFP_UART0_TX | SYS_PB_L_MFP_PB1_MFP_UART0_RX);

    /* 将PB12、PB13、PB14、PB15配置为 SPI0 功能*/
    SYS->PB_H_MFP = (SYS_PB_H_MFP_PB12_MFP_SPI0_MOSI0 | SYS_PB_H_MFP_PB13_MFP_SPI0_MISO0
                     | SYS_PB_H_MFP_PB14_MFP_SPI0_SCLK | SYS_PB_H_MFP_PB15_MFP_SPI0_SS0);
    /* 重新加锁 */
    SYS_LockReg();
}

int SPI_Init(void)
{
    uint32_t u32DataCount, u32TestCount, u32Err;

    /* SPI0选HCLK做时钟源 */
    CLK_SetModuleClock(SPI0_MODULE, CLK_CLKSEL2_SPI0_S_HCLK, 0);

    /* 使能SPI0 IP的时钟 */
    CLK_EnableModuleClock(SPI0_MODULE);
   /* 将PB12、PB13、PB14、PB15配置为 SPI0 功能*/
    SYS->PB_H_MFP = (SYS_PB_H_MFP_PB12_MFP_SPI0_MOSI0 | SYS_PB_H_MFP_PB13_MFP_SPI0_MISO0
                     | SYS_PB_H_MFP_PB14_MFP_SPI0_SCLK | SYS_PB_H_MFP_PB15_MFP_SPI0_SS0);
    /* 将SPI0配置为master, MSB 优先, 每笔32-bit, SPI Mode-0 timing, clock is 2MHz */
    SPI_Open(SPI0, SPI_MASTER, SPI_MODE_0, 32, 2000000);
    /*LSB 优先可以用下面的宏*/
    //SPI_SET_LSB_FIRST(SPI0);

    /* 将SPI0片选配置为low level有效. */
    SPI0->SSR |= SPI_SSR_SS_LTRIG_Msk;//片选level 有效
    SPI0->SSR &= ~SPI_SSR_SS_LVL_Msk;//片选low 有效
    /*使能片选，SS引脚将拉low*/
    SPI0->SSR = SPI0->SSR | 0x1;

    /* SPI0 Loopback test，就是自己发自己收 */
    u32Err = 0;
    for(u32TestCount=0; u32TestCount<10000; u32TestCount++) {

        /* 将要发送的数据写到SPI0 TX0寄存器 */
        SPI_WRITE_TX0(SPI0, u32TestCount);
        /* 触发开始发送，发送的同时将接收 */
```

```
        SPI_TRIGGER(SPI0);

        g_au32DestinationData0[u32DataCount] = SPI_READ_RX0(SPI0);

        /*  检查收到的数据 */
        if(u32TestCount!= g_au32DestinationData0[u32DataCount])
                break;
        }
    }

    while(1);
}
```

3.13 ACMP 初始化

比较器有两根输入脚分别命名为：正端(positive)和负端(negative)外加一根输出脚。比较两根输入脚的电压，如果正端电压大于负端，输出高电平，否则输出低电平。比较结果可以输出到引脚上也可以读比较状态寄存器得到（输出引脚就可以当作普通IO使用）。

负端输入电压有几个选择：比较器负端引脚(由外部输入电压)、内部阶梯电阻分压、内部参考电压和模拟地。如果选择内部阶梯电阻分压，分压公式如下：

比较电压 = AVDD x (1/6 +CRV/24)， CRV = 0 ~ 15

比较器工作原理很好理解，要用好比较器，还需要理解它的很多电气特性。比较重要的有下面的7项

V_{OFF}	Input Offset Voltage		10	20	mV	-
V_{SW}	Output Swing	0.1	-	AV_{DD} - 0.1	V	-

V_{COM}	Input Common Mode Range	0.1	-	AV_{DD} – 0.1	V	-
-	DC Gain	40	70	-	dB	-
T_{PGD}	Propagation Delay	-	200	-	ns	V_{DIFF} = 100mV
V_{HYS}	Hysteresis	-	±10	-	mV	
T_{STB}	Stable time	-	-	1	μs	

- Input Offset Voltage：输入电压偏移，就是输出为 0 时，N 端接地，P 端需要施加的电压
- Output Swing：是指输出摆幅，就是输出信号最大值和最小值之间的差值
- Input Common Mode Range：是指输入共模电压范围，就是(N+P)/2 的范围
- DC Gain：是指放大倍数，40db 就是指放大 100 倍。输出/输入 放大 100 倍
- Propagation Delay：是指传输延迟。

- Hysteresis：是指迟滞。例如：P 比 N 大输出不是马上变 high 的，而是 P 要比 N 高 10mV 才会变 high；P 比 N 低的时候也不是马上变 low，而是 P 比 N 低 10mV 之后，输出才会变 low
- Stable time：比较器的输出从低电平变高电平，或者从高电平变低电平所需要的时间

下面是ACMP的初始化代码

```
void ACMP_Init()
{
    /*ACMP的配置非常简单：配置引脚，使能时钟，配置IP*/
    /*PA6配置为输出脚，PA5为负端，PA4为正端*/
   SYS->PA_L_MFP = (SYS->PA_L_MFP & ~SYS_PA_L_MFP_PA6_MFP_Msk ) |
SYS_PA_L_MFP_PA6_MFP_ACMP0_O;  /* ACMP CPO0  */
    SYS->PA_L_MFP = (SYS->PA_L_MFP & ~SYS_PA_L_MFP_PA5_MFP_Msk ) |
SYS_PA_L_MFP_PA5_MFP_ACMP0_N;  /* ACMP CPN0  */
    SYS->PA_L_MFP = (SYS->PA_L_MFP & ~SYS_PA_L_MFP_PA4_MFP_Msk ) |
SYS_PA_L_MFP_PA4_MFP_ACMP0_P;  /* ACMP CPP0  */
   /*使能ACMP时钟*/
   CLK_EnableModuleClock(ACMP_MODULE);
   /*配置ACMP0，负端电压=AVDD*5/24，并关闭迟滞功能*/
   ACMP_Open(ACMP,0,ACMP_CR_CN_CRV|1,ACMP_CR_ACMP_HYSTERSIS_DISABLE);
}
```

有了上面的代码ACMP就会动了。然后通过ACMP_GET_OUTPUT(ACMP, 0)就可以得到比较结果了。

如果希望使用中断的方式，下面的代码可以打开中断：

```
   ACMP_ENABLE_INT(ACMP, 0);
   NVIC_EnableIRQ(ACMP_IRQn);
   void ACMP_IRQHandler()
   {
       ACMP_CLR_INT_FLAG(ACMP, 0);
   }
```

但是ACMP IP不止上面的功能，有的芯片中的ACMP还支持Single SLOP和Sigma_Delta模式。Single SLOP就是单斜式，一般用于测量热敏电阻，例如：PT100的值。

Sigma_Delta 测量精度比较高，但是速度不快，一般10几次/s

这两个功能在BSP里面都有对应的demo code，放在BSP中目录SampleCode\StdDriver下面。这里就不讲了。

3.14 WDT 初始化

看门狗用来防止代码死机。软件设定喂狗间隔之后，如果到时间没有喂狗，说明软件跑飞了，这时候看门狗就会复位整个系统，从头重新跑。

喂狗间隔由WDT 控制寄存器WTIS决定。超时时间到，WDT_ISR寄存器的WDT_IS将置为'1'。软件可以轮询该标志喂狗或者使能WDT中断在中断里面喂狗。一般在主循环里面喂狗，这样主循环死掉了就会复位。如果在中断里面喂狗，就是程序死的中断都没有了才会复位，例如：发生了HardFault中断，因为它的优先级比较高，如果没有实现HardFault中断处理函数，默认的HardFault中断处理函数就是死循环，看门狗的中断没有机会再发生了，就会复位系统。

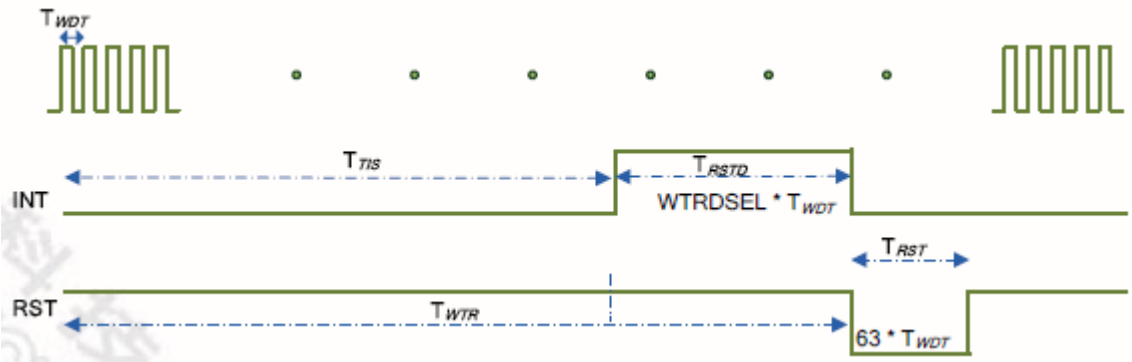
超时之后，WDT IP会延迟一段时间(复位延迟时间WTRDSEL)等软件喂狗，该时间内没有喂狗，才会发生复位。

WTIS	WTR Time-out Interval	Reset Delay Period T_{RSTD}	Time-out Interval WDT_CLK = 10 kHz T_{TIS}
000	$2^4 * T_{WDT}$	(3/18/130/1026) * T_{WDT}	1.6 ms
001	$2^6 * T_{WDT}$	(3/18/130/1026) * T_{WDT}	6.4 ms
010	$2^8 * T_{WDT}$	(3/18/130/1026) * T_{WDT}	25.6 ms
011	$2^{10} * T_{WDT}$	(3/18/130/1026) * T_{WDT}	102.4 ms
100	$2^{12} * T_{WDT}$	(3/18/130/1026) * T_{WDT}	407 ms
101	$2^{14} * T_{WDT}$	(3/18/130/1026) * T_{WDT}	1.638 s
110	$2^{16} * T_{WDT}$	(3/18/130/1026) * T_{WDT}	6.553 s
111	$2^{18} * T_{WDT}$	(3/18/130/1026) * T_{WDT}	26.214 s

Table 6-12 Watchdog Time-out Interval Selection

上图红色框中的两列分别是超时间隔和复位延迟。

看门狗超时+复位波形图如下：



T_{TIS}是喂狗超时间隔

T_{RSTD}是复位延迟

T_{RST}是复位时间，复位信号拉low 63个WDT时钟周期

看门狗初始化代码如下，WDT一般不能选时钟源，只能以内部低速晶振(10K)做时钟源。

```
void WDT_Init()
{
    /*解锁*/
    SYS_UnlockReg();
    /* 使能内部低速晶振 LIRC */
    CLK_EnableXtalRC(CLK_PWRCTL_LIRC_EN_Msk);
    /* 等待LIRC 稳定 */
    CLK_WaitClockReady( CLK_CLKSTATUS_LIRC_STB_Msk);
    /*使能看门狗时钟*/
    CLK_EnableModuleClock(WDT_MODULE);
    /* 重新加锁 */
    SYS_LockReg();

    /*配置看门狗*/
    /* WDT 寄存器是写保护的，所以配置WDT之前需要解锁*/
    SYS_UnlockReg();

    /* WDT 超时间隔选择 2^14 WDT时钟，关闭复位系统功能，关闭唤醒系统功能*/
    WDT_Open(WDT_TIMEOUT_2POW14, 0, FALSE, FALSE);

    while(1) {
        // 检查WDT 超时标志
        if(WDT_GET_TIMEOUT_INT_FLAG()) {
            // 喂狗
            WDT_CLEAR_TIMEOUT_INT_FLAG();
            printf("Reset WDT counter\n");
        }
    }
```

看门狗除了复位系统功能，还可以将系统唤醒。如果需要MCU进入idle或者掉电模式，可以用WDT唤醒。进入power down之前，硬件会自动喂一次狗。

3.15 WWDT 初始化

窗看门狗一般固定是用HCLK/2048做时钟源，超时时间是固定的64个时钟周期。但是WWDT有自己的预分频寄存器，可以将超时时间最多延长2048倍，如下图。

PERIODSEL	Prescaler Value	Timeout Period	Timeout Interval 12 MHz/2048 = 5.859 kHz WWDT_CLK=5.859 kHz
0000	1	1 * 64 * T_{WWDT}	10.9 ms
0001	2	2 * 64 * T_{WWDT}	21.8 ms
0010	4	4 * 64 * T_{WWDT}	43.7 ms
0011	8	8 * 64 * T_{WWDT}	87.4 ms
0100	16	16 * 64 * T_{WWDT}	174.7 ms
0101	32	32 * 64 * T_{WWDT}	349.5 ms
0110	64	64 * 64 * T_{WWDT}	699.1 ms
0111	128	128 * 64 * T_{WWDT}	1.3981 s
1000	192	192 * 64 * T_{WWDT}	2.0971 s
1001	256	256 * 64 * T_{WWDT}	2.7962 s
1010	384	384 * 64 * T_{WWDT}	4.1943 s
1011	512	512 * 64 * T_{WWDT}	5.5924 s
1100	768	768 * 64 * T_{WWDT}	8.3886 s
1101	1024	1024 * 64 * T_{WWDT}	11.1848 s
1110	1536	1536 * 64 * T_{WWDT}	16.7772 s
1111	2048	2048 * 64 * T_{WWDT}	22.3696 s

Table 6-13 Window Watchdog Prescaler Value Selection

WWDT计数器是下数计数器，它的值在0 ~ WINCMP之间时，才能写WWDTRLD寄存器喂狗，否则WWDT将复位系统。

看门狗和窗看门狗的区别就是WWDT上电之后一旦使能，就不能关闭，并且不能随意喂狗，这可以防止软件跑飞误将看门狗关闭，或者误喂狗。它用HCLK做时钟源，所以power down时它不能工作。

下面是WWDT的初始化代码，WWDT我们在中断里面喂狗

```
void WWDT_Init()
{
    /* 解锁写保护寄存器 */
    SYS_UnlockReg();

    /* 使能外部晶振HXT (4~24 MHz) */
    CLK_EnableXtalRC(CLK_PWRCTL_HXT_EN_Msk);

    /* 等待外部 12MHz 晶振稳定 */
    CLK_WaitClockReady( CLK_CLKSTATUS_HXT_STB_Msk);
    /* HCLK选择HXT做时钟源 */
    CLK_SetHCLK(CLK_CLKSEL0_HCLK_S_HXT,CLK_HCLK_CLK_DIVIDER(1));
    /* WWDT 寄存器是写保护的，所以配置WWDT之前需要解锁。但是前面我们没有加锁，其实这句是不需要的 */
    SYS_UnlockReg();
    /* WWDT 超时时间为768 * 64 WWDT 时钟，比较寄存器的值为32，时间就是 768 * 32 WWDT时钟 ,使能
WWDT 计数器比较中断 */
    WWDT_Open(WWDT_PRESCALER_768, 0x20, TRUE);
```

```
    // 使能 WWDT (和WDT共享中断号) 超时中断
    NVIC_EnableIRQ(WDT_IRQn);
}
/*中断处理函数*/
void WDT_IRQHandler(void)
{
    // 喂狗并清除中断标志
    WWDT_RELOAD_COUNTER();
    WWDT_CLEAR_INT_FLAG();
    printf("WWDT counter reload\n");
}
```

3.16 FMC 初始化

新唐的M0/M4都是内嵌ROM(Flash)的，FMC(Flash Memory Controller)就是用来读/写ROM的IP。它支持擦除/读/写等命令。

该IP的寄存器很简单，主要为：地址(ISPADR)、数据(ISPDAT)、命令(ISPCMD)、触发(ISPTRG)

地址寄存器：将要操作的ROM地址，用于存放读/写/擦除/向量重映射等的ROM地址

数据寄存器：将要写到ROM中数据，或者从ROM中读出来的数据

命令寄存器：将要进行的操作命令

触发寄存器：该寄存器写1，就执行命令寄存器中指定的命令

例如：擦除地址0x1000，代码如下：

ISPADDR = 0x1000;

ISPCMD = 0x22;

ISPTRG = 1;

while (ISPTRG & 0x1) ;

等ISPTRG 清为0， 擦除就完成了。

写Flash，只能由1变0，不能由0变1，所以如果想由0变1只能执行擦除的动作，将一个page 全部擦成0xFFFFFFFF。

芯片中所有的ROM都用该IP操作，所不同的只是地址不同.

- APROM 基地址为 0x00000000
- LDROM 基地址为 0x00100000
- Dataflash 基地址由 config1 寄存器决定，或者固定的。详细要看 TRM 中的描述
- Config 区域地址为 0x300000

注意：Config area修改之后需要复位才能起做用。如果不用ICP tool 修改Config area，而用软件修改，修改之后需要软件发出CHIP reset才能起做用奥！

ROM每次擦除一个page，每个page 512字节。写操作只能将ROM中'1'变成'0'，不能将'0'变成'1'，从'0'变成'1'只能用擦除命令。

ROM每次读/写4个字节，**所以注意读/写的地址一定要4对齐**。

下面是FMC读/写 dataflash的例子，dataflash提前用ICP tool在config0中使能，基地址在config1中设好，假设基地址=0x3000。

```
#define DATA_FLASH_TEST_BASE        0x3000
int32_t    u32Addr = DATA_FLASH_TEST_BASE;
uint32_t    u32Pattern = 0x5A5A5A5A;
void dataflash_test()
{
    /*解锁写保护寄存器*/
    SYS_UnlockReg();

    /* 使能 FMC ISP 功能 */
    FMC_Open();
    /* 擦除一个page */
    FMC_Erase(u32Addr);
    /*写4 Bytes到dataflash地址0*/
    FMC_Write(u32Addr, u32Pattern);
    /*读dataflash地址0*/
    u32data = FMC_Read(u32Addr);
    if (u32data != u32Pattern) {/*比较读到的数据是否等于写入的数据*/
        /*数据不相等*/
        return -1;
    }

    /*写4 Bytes到dataflash地址4*/
    FMC_Write(u32Addr+4, u32Pattern);
    /*读dataflash地址4*/
    u32data = FMC_Read(u32Addr+4);
    if (u32data != u32Pattern) {/*比较读到的数据是否等于写入的数据*/
        /*数据不相等*/
        return -1;
    }
}
```

读/写/擦除 LDROM/APROM或者Config area都用上面的函数，只是给的地址不同。

另外，为了防止软件误写LDROM/APROM/Config area，这几个部分写/擦除都需要写ISPCON寄存器使能相应的ROM update才行。对应函数如下：

FMC_EnableConfigUpdate()/*使能Config Area Update功能*/

FMC_EnableLDUpdate ()/*使能LDROM Update功能*/

FMC_EnableAPUpdate ()/*使能APROM Update功能*/

大家注意上面几个函数和函数FMC_Open，要调用这4个函数，注意先执行解锁。

3.17 PDMA 初始化

新唐很多芯片带PDMA， P是peripheral的意思，就是该DMA适用于外设和SRAM之间传输数据，当然SRAM和SRAM之间传输也可以，但是不能IP到IP直接传。另外有的PDMA还支持计算CRC。

PDMA一般有多个通道，每个通道可以由软件选择给哪个IP使用，例如：每个通道有下面一个选择list（只选择了一部分，实际很长）

```
00000 =  Connect to SPI0_TX.
00001 =  Connect to SPI1_TX.
00010 =  Connect to UART0_TX.
00011 =  Connect to UART1_TX.
00100 =  Reserved.
00101 =  Reserved.
00110 =  Reserved.
00111 =  Reserved.
01000 =  Reserved.
01001 =  Connect to TMR0.
01010 =  Connect to TMR1.
01011 =  Connect to TMR2.
01100 =  Connect to TMR3.
10000 =  Connect to SPI0_RX.
```

我们可以将PDMA CH1选给SPI0 TX， CH2选给SPI0 RX

PDMA有一个限制就是源地址和目标地址如果是SRAM地址，该地址必须4对齐。

一般支持PDMA的外设有SPI、UART、ADC、Timer捕获等等，我们以SPI为例说明PDMA的用法。

初始化分为两部分：SPI初始化，并使能PDMA；PDMA初始化通道1和通道2，通道1给SPI0 TX， 通道2给SPI0 RX。将SPI0 MISO和MOSI接到一起，做loopback测试。

```
/*PDMA中断处理函数，传输完成或者出错，或者超时等都会发生中断*/
void PDMA_IRQHandler(void)
{
    uint32_t status = PDMA_GET_INT_STATUS();

    if (status & 0x2) { /* 通道1发生中断 */
```

```
        PDMA_CLR_CH_INT_FLAG(1, PDMA_ISR_TD_IS_Msk);
    } else if (status & 0x4) { /* 通道2发生中断 */
        if (PDMA_GET_CH_INT_STS(2) & 0x2)
            u32IsTestOver = 1;
        PDMA_CLR_CH_INT_FLAG(2, PDMA_ISR_TD_IS_Msk);
    } else
        /*其它中断都没有使能，理论上不应该发生*/
        printf("unknown interrupt, status=0x%x !!\n", status);
}
void SYS_Init(void)
{
    /* 系统初始化 */
    /* 解锁写保护寄存器 */
    SYS_UnlockReg();

    /* 使能外部 12MHz HXT, 32KHz LXT 和 HIRC */
    CLK_EnableXtalRC(CLK_PWRCTL_HXT_EN_Msk | CLK_PWRCTL_LXT_EN_Msk |
CLK_PWRCTL_HIRC_EN_Msk);

    /* 等待晶振起振 */
    CLK_WaitClockReady(CLK_CLKSTATUS_HXT_STB_Msk | CLK_CLKSTATUS_LXT_STB_Msk |
CLK_CLKSTATUS_HIRC_STB_Msk);

    /*  使能PLL，并将HCLK的时钟源切成PLL */
    CLK_SetCoreClock(32000000);

    /* 选择 IP 时钟源,HCLK就是CPU的时钟 */
    CLK_SetModuleClock(UART0_MODULE, CLK_CLKSEL1_UART_S_HIRC, CLK_UART_CLK_DIVIDER(1));
    CLK_SetModuleClock(SPI0_MODULE, CLK_CLKSEL2_SPI0_S_HCLK, 0);
    CLK_SetModuleClock(SPI1_MODULE, CLK_CLKSEL2_SPI1_S_HCLK, 0);

    /* 使能 IP 时钟 */
    CLK_EnableModuleClock(UART0_MODULE);
    CLK_EnableModuleClock(SPI0_MODULE);
    CLK_EnableModuleClock(DMA_MODULE);

    /* Update System Core Clock */
    /* User can use SystemCoreClockUpdate() to calculate PllClock, SystemCoreClock and
CycylesPerUs automatically. */
    SystemCoreClockUpdate();

    /* 配置多功能引脚 */
```

```
    /* 配置 PB0/PB1 用作UART0 RXD 和 TXD  */
    SYS->PB_L_MFP &= ~(SYS_PB_L_MFP_PB0_MFP_Msk | SYS_PB_L_MFP_PB1_MFP_Msk);
    SYS->PB_L_MFP |= (SYS_PB_L_MFP_PB0_MFP_UART0_TX | SYS_PB_L_MFP_PB1_MFP_UART0_RX);

    /* 配置SPI0多功能引脚 */
    SYS->PB_H_MFP = (SYS_PB_H_MFP_PB12_MFP_SPI0_MOSI0 | SYS_PB_H_MFP_PB13_MFP_SPI0_MISO0
                     | SYS_PB_H_MFP_PB14_MFP_SPI0_SCLK | SYS_PB_H_MFP_PB15_MFP_SPI0_SS0);

    /* 重新加锁 */
    SYS_LockReg();
}
int main(void)
{
    uint32_t u32Err=0;
    uint32_t i;

    /* 系统初始化：使能晶振，选择IP时钟源，使能IP时钟，配置多功能引脚 */
    SYS_Init();

    /* 配置UART为 115200-8n1 用于printf */
    UART_Open(UART0, 115200);

    /* 将SPI0配置为master, MSB first, 每笔32-bit, SPI Mode-0 timing, 总线时钟为2MHz */
    SPI_Open(SPI0, SPI_MASTER, SPI_MODE_0, 32, 2000000);

    /* 使能硬件自动片选功能，收/发数据时硬件将自动拉片选，选择SPI0_SS0 引脚并配置为低电平有效. */
    SPI_EnableAutoSS(SPI0, SPI_SS0, SPI_SS0_ACTIVE_LOW);

    /*初始化SPI0 将发送的数据*/
    for(i=0; i<PDMA_TEST_COUNT; i++)
        g_au32SrcData[i] = 0x55550000 + i;
    /*初始化PDMA*/
    /* 打开通道1和通道2 */
    PDMA_Open(3 << 1);

    /* 配置通道 1，每笔数据32bit，共发送PDMA_TEST_COUNT笔 */
    PDMA_SetTransferCnt(1, PDMA_WIDTH_32, PDMA_TEST_COUNT);
    /*配置通道1源地址为g_au32SrcData ，增加；目的地址为SPI0->TX0寄存器，固定*/
    PDMA_SetTransferAddr(1, (uint32_t)g_au32SrcData, PDMA_SAR_INC, (uint32_t)&SPI0->TX0,
PDMA_DAR_FIX);
    /*关闭超时功能*/
```

```
    PDMA_SetTimeOut(1, 0, 0x5555);
    /*使能发送完成中断*/
    PDMA_EnableInt(1, PDMA_IER_TD_IE_Msk);

    /*配置通道 2，每笔数据32bit，共发送PDMA_TEST_COUNT笔*/
    PDMA_SetTransferCnt(2, PDMA_WIDTH_32, PDMA_TEST_COUNT);
    /*配置通道2源地址为SPI0->RX0，固定；目的地址为g_au32DstData，增加*/
    PDMA_SetTransferAddr(2, (uint32_t)&SPI0->RX0, PDMA_SAR_FIX, (uint32_t)g_au32DstData,
PDMA_DAR_INC);
    /*关闭超时功能*/
    PDMA_SetTimeOut(2, 0, 0x5555);
    /*使能发送完成中断*/
    PDMA_EnableInt(2, PDMA_IER_TD_IE_Msk);

    /* 将通道1配置为SPI0 TX，通道2配置为SPI0 RX，然后触发PDMA开始工作 */
    PDMA_SetTransferMode(1, PDMA_SPI0_TX, 0, 0);
    PDMA_SetTransferMode(2, PDMA_SPI0_RX, 0, 0);

    PDMA_Trigger(1);
    PDMA_Trigger(2);

    /* 使能 PDMA IRQ */
    NVIC_EnableIRQ(PDMA_IRQn);

    /* SPI使能RX PDMA和TX PDMA */
    SPI_TRIGGER_RX_PDMA(SPI0);
    SPI_TRIGGER_TX_PDMA(SPI0);

    /* 等待PDMA完成 */
    while(u32IsTestOver == 0);

    /* 检查收到的数据 */
    for(i=0; i<PDMA_TEST_COUNT; i++) {
        if(g_au32SrcData[i] != g_au32DstData[i]) {
            u32Err ++;
        }
    }

    if(u32Err)
        printf(" [FAIL]\n\n");
    else
```

```
        printf(" [PASS]\n\n");

    while(1);
}
```

3.18 DAC 初始化

我们以NANO100BN的DAC来说明DAC的用法和特点

- NANO100 支持 2 个 12-bit DAC 通道
- 支持组模式 (2 组 DAC 同步更新)
  - 2 个 DAC 通道同步更新
  - Channel 0 决定触发模式
  - 如果数据更新没有使用 PDMA, 在下一个转换开始之前，软件必须更新好两个 DACx_DAT 寄存器
- 有 3 种触发方式
  - 软件
  - 定时器
  - PDMA
- 上电稳定时间: 6 us
  - DACPWONSTBCNT (14 bit 可编程，单位 PCLK)
  - DAC 上电之后，等待 DACPWONSTBCNT+1 个 PCLK 开始转换
- 转换完成时间: 2 us (500k sample/sec)
  - WAITDACCONV (8 bit 可调，单位 PCLK)，用于 PDMA 触发 DAC 时，保证 DAC 转换完成

下面的代码使用软件触发的方式，组模式，不用选择时钟源，时钟源默认为PCLK

```
void DAC_Init()
{
    /* 使能 DAC 时钟 */
    CLK->APBCLK |= CLK_APBCLK_DAC_EN_Msk;
    /* 配置多功能引脚 PC.6 和 PC.7 用作 DAC */
    SYS->PC_L_MFP &= ~(SYS_PC_L_MFP_PC6_MFP_Msk | SYS_PC_L_MFP_PC7_MFP_Msk);
    SYS->PC_L_MFP |= SYS_PC_L_MFP_PC6_MFP_DA_OUT0 | SYS_PC_L_MFP_PC7_MFP_DA_OUT1;

    /* 关闭PC.6 and PC.7引脚的数字通路 */
    GPIO_DISABLE_DIGITAL_PATH(PC, (1 << 6) | (1 << 7));
    /* 软件触发 */
    DAC_Open(DAC, 0, DAC_WRITE_DAT_TRIGGER);
```

```
    DAC_Open(DAC, 1, DAC_WRITE_DAT_TRIGGER);
    /* 使能 DAC 组模式 */
    DAC_ENABLE_GROUP_MODE(DAC);

    // 使能 DAC0 中断. 组模式中使能一个通道就可以了.
    DAC_ENABLE_INT(DAC, 0);
    NVIC_EnableIRQ(DAC_IRQn);
    // 等待两个通道ready
    while(DAC_IS_BUSY(DAC, 0) == 1);
    while(DAC_IS_BUSY(DAC, 1) == 1);
    /*写第一笔数据到DATA寄存器*/
    DAC_WRITE_DATA(DAC, 0, a16Sine[index0]);
    DAC_WRITE_DATA(DAC, 1, a16Sine[index1]); // 写通道1将触发 DAC 开始转换
}
/*中断处理函数*/
void DAC_IRQHandler(void)
{
    // 清除中断标志
    DAC_CLR_INT_FLAG(DAC, 0);
    /*写下一笔数据*/
    DAC_WRITE_DATA(DAC, 0, a16Sine[index0]);
    DAC_WRITE_DATA(DAC, 1, a16Sine[index1]); // 写通道1将触发 DAC 开始转换
    index0 = (index0 + 1) % SINE_ARRAY_SIZE;
    index1 = (index1 + 1) % SINE_ARRAY_SIZE;

    return;
}
```

3.19 EBI 初始化

新唐的EBI接口可以用来外接IO设备、SRAM等符合INTEL接口的设备。接的外设访问地址为0x60000000，如果有多个EBI接口，其它的EBI接口访问地址详见芯片的技术参考手册(TRM)。一般EBI有两种接口：INTEL模式和MOTOROLA模式。INTEL模式就是读/写用/RD和/WR两根线分别控制；MOTOROLA模式就是一根RW线，高电平读，低电平写。

- EBI 可以支持的外设访问范围，从 64K~1MB 不等。
- 外部总线时钟 MCLK 的频率可以编程。
- 数据宽度支持 8bit 和 16bit 两种
- 下列时序可以改变
  - 数据访问时间 (tACC)

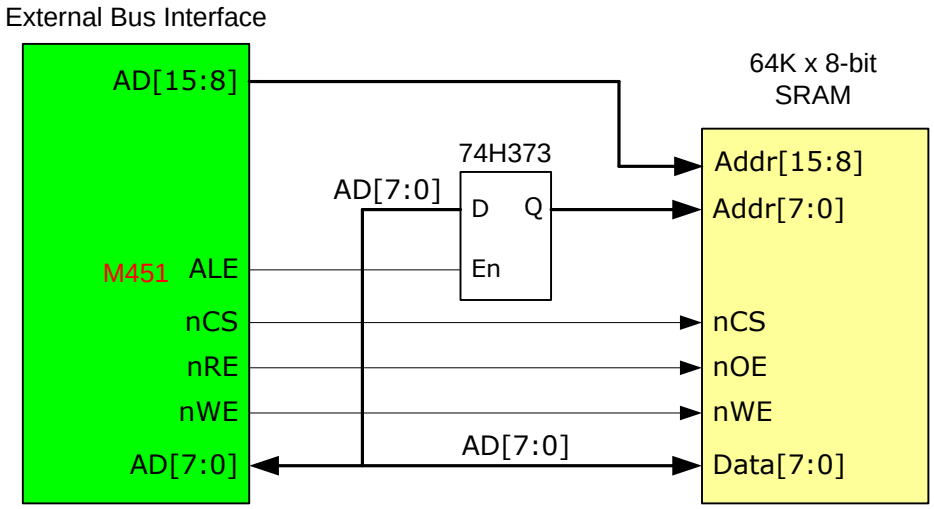
- 地址锁存使能时间 (tALE)
- 数据访问保持时间 (tAHD)

➢ 为了节省引脚数，地址和数据总线是复用的

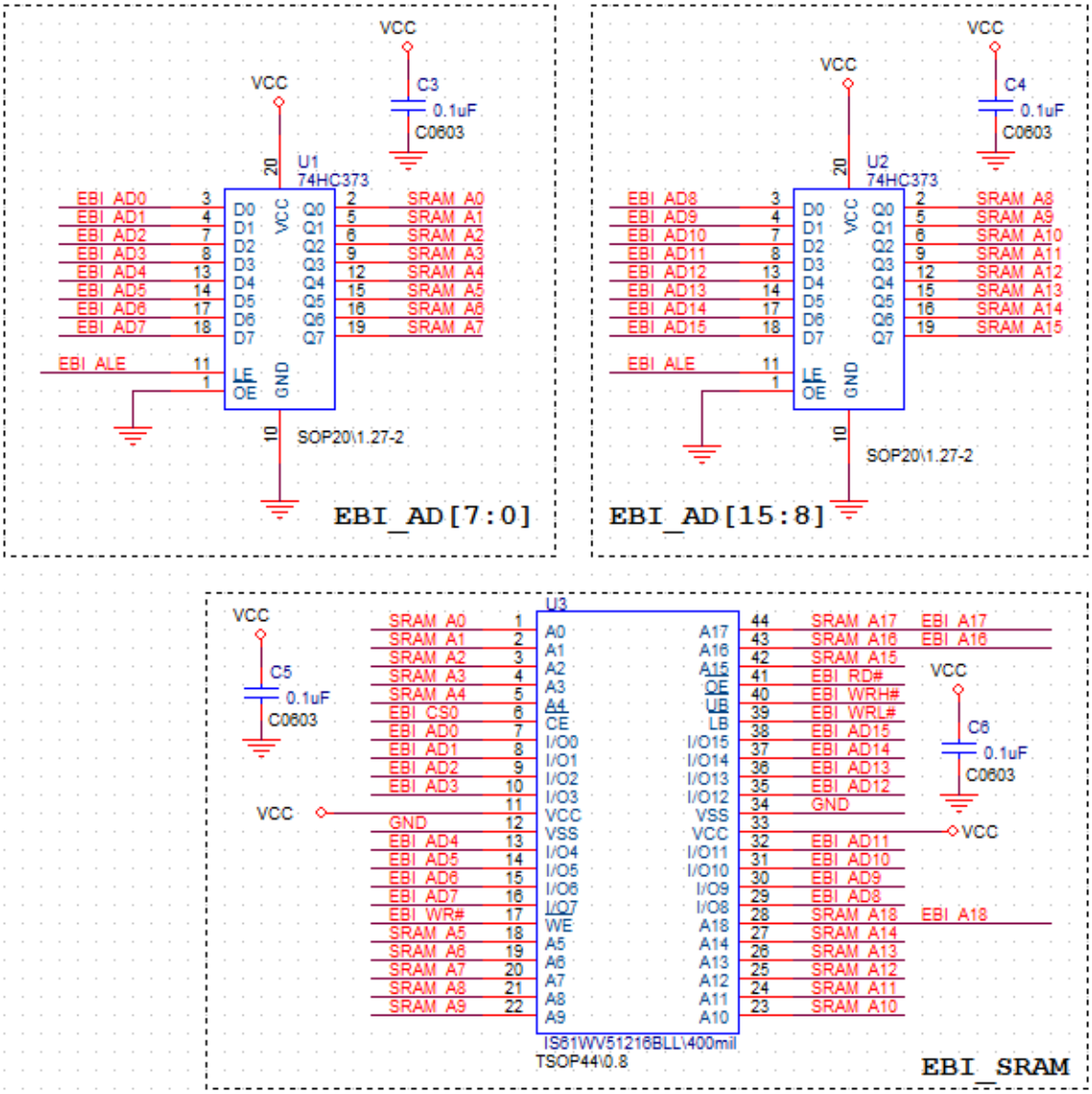
➢ 不同访问条件下, 支持 idle cycle 可配置, [0，15]MCLK

nWRL和nWRH用于Byte access 一个 16-bit 位宽的 IO 或者 SRAM，用于访问高位字节或者低位字节

EBI连接外设示意图如下，因为地址线和数据线是共用的，74H373用于锁存共用的地址线部分。例如下图，数据线只用8-bit，所以AD[7:0]是地址和数据共用的，AD[15:8]只用于地址。所以地址AD[7:0]先送到74H373锁存，然后AD[7:0]再给数据用。



原理图如下：



下面是M451 EBI外接SRAM的范例，EBI只能用HCLK做时钟源，所以它的时钟源不可以选择。EBI引脚配置如下：：

- AD0 ~ AD7 on PA.0 ~ PA.7
- AD8 ~ AD15 on PC.0 ~ PC.7
- AD16 ~ AD19 on PD.12 ~ PD.15
- nWR on PD.2
- nRD on PD.7
- nWRL on PB.0
- nWRH on PB.1
- nCS0 on PD.8

- ALE on PD.9
- MCLK on PD.3

引脚配置函数如下：

```
void Configure_EBI_16BIT_Pins(void)
{
    /* EBI AD0~7 pins on PA.0~7 */
    SYS->GPA_MFPL |= SYS_GPA_MFPL_PA0MFP_EBI_AD0 | SYS_GPA_MFPL_PA1MFP_EBI_AD1 |
                     SYS_GPA_MFPL_PA2MFP_EBI_AD2 | SYS_GPA_MFPL_PA3MFP_EBI_AD3 |
                     SYS_GPA_MFPL_PA4MFP_EBI_AD4 | SYS_GPA_MFPL_PA5MFP_EBI_AD5 |
                     SYS_GPA_MFPL_PA6MFP_EBI_AD6 | SYS_GPA_MFPL_PA7MFP_EBI_AD7;

    /* EBI AD8~15 pins on PC.0~7 */
    SYS->GPC_MFPL |= SYS_GPC_MFPL_PC0MFP_EBI_AD8 | SYS_GPC_MFPL_PC1MFP_EBI_AD9 |
                     SYS_GPC_MFPL_PC2MFP_EBI_AD10 | SYS_GPC_MFPL_PC3MFP_EBI_AD11 |
                     SYS_GPC_MFPL_PC4MFP_EBI_AD12 | SYS_GPC_MFPL_PC5MFP_EBI_AD13 |
                     SYS_GPC_MFPL_PC6MFP_EBI_AD14 | SYS_GPC_MFPL_PC7MFP_EBI_AD15;

    /* EBI AD16~19 pins on PD.12~15*/
    SYS->GPD_MFPH |= SYS_GPD_MFPH_PD12MFP_EBI_ADR16 | SYS_GPD_MFPH_PD13MFP_EBI_ADR17 |
                     SYS_GPD_MFPH_PD14MFP_EBI_ADR18 | SYS_GPD_MFPH_PD15MFP_EBI_ADR19;

    /* EBI nWR and nRD pins on PD.2 and PD.7 */
    SYS->GPD_MFPL |= SYS_GPD_MFPL_PD2MFP_EBI_nWR | SYS_GPD_MFPL_PD7MFP_EBI_nRD;

    /* EBI nWRL and nWRH pins on PB.0 and PB.1 */
    SYS->GPB_MFPL |= SYS_GPB_MFPL_PB0MFP_EBI_nWRL | SYS_GPB_MFPL_PB1MFP_EBI_nWRH;

    /* EBI nCS0 pin on PD.8 */
    SYS->GPD_MFPH |= SYS_GPD_MFPH_PD8MFP_EBI_nCS0;

    /* EBI ALE pin on PD.9 */
    SYS->GPD_MFPH |= SYS_GPD_MFPH_PD9MFP_EBI_ALE;

    /* EBI MCLK pin on PD.3 */
    SYS->GPD_MFPL |= SYS_GPD_MFPL_PD3MFP_EBI_MCLK;
}
```

引脚配置好之后，下面是EBI的功能初始化。M451有2个EBI接口，我们使用EBI0，总线数据宽

度16bit, CS信号低电平有效

```
void EBI_Test()
{
    uint32_t u32WriteData;
    uint32_t u32Idx;
    uint32_t u32EBIsize = 512 * 1024;
    /*使能EBI的时钟*/
    CLK_EnableModuleClock(EBI_MODULE);
    Configure_EBI_16BIT_Pins();
    /* 初始化 EBI bank0，访问外部 SRAM */
    EBI_Open(EBI_BANK0, EBI_BUSWIDTH_16BIT, EBI_TIMING_NORMAL, 0, EBI_CS_ACTIVE_LOW);
    /*写数据测试如下*/
    /*每次写1个字节*/
    u32Idx = 0;
    u32WriteData = 0x5A5A5A5A;
    while(u32Idx < u32EBIsize)/*外接的EBI设备的Size*/
    {
        EBI0_WRITE_DATA8(u32Idx, (uint8_t)(u32WriteData));
        u32Idx++;
    }
    /*每次写2个字节*/
    u32Idx = 0;
    u32WriteData = 0x00000000;
    while(u32Idx < u32EBIsize)
    {
        EBI0_WRITE_DATA16(u32Idx, (uint16_t)(u32WriteData));
        u32Idx += 2;
    }
    /*每次写4个字节*/
    u32Idx = 0;
    u32WriteData = 0xFFFFFFFF;
    while(u32Idx < u32EBIsize)
    {
        EBI0_WRITE_DATA32(u32Idx, (uint32_t)(u32WriteData));
        u32Idx += 4;
    }
}
```

读数据的函数如下，1字节读，2字节读，4字节读都是可以的

```
EBI0_READ_DATA8(u32Idx);
EBI0_READ_DATA16(u32Idx);
```

```
EBI0_READ_DATA32(u32Idx);
```

3.20 密码 IP 初始化

NUC442/NUC472有带一个密码IP，该IP支持伪随机数产生、AES、DES/TES、SHA

下面以AES加解密为例说明该IP的用法：

```
/*AES加密密码128-bit*/
uint32_t au32MyAESKey[8] = {
    0x00010203, 0x04050607, 0x08090a0b, 0x0c0d0e0f,
};
/*AES加密初始化向量*/
uint32_t au32MyAESIV[4] = {
    0x00000000, 0x00000000, 0x00000000, 0x00000000
};
/*要加密的数据*/
__align(4) uint8_t au8InputData[] = {
    0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88,
    0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff
};
__align(4) uint8_t au8EncOutData[32];
__align(4) uint8_t au8DecOutData[32];

/*密码IP中断处理函数*/
volatile void CRYPTO_IRQHandler()
{
    if (AES_GET_INT_FLAG()) {
        g_AES_done = 1;
        AES_CLR_INT_FLAG();
    }
}
```

下面这段代码执行以后，密码IP就开始工作了

```
Void AES_Test()
{
    /*使能密码IP的时钟*/
    CLK_EnableModuleClock(CRPT_MODULE);
    /*使能中断*/
    NVIC_EnableIRQ(CRPT_IRQn);
    AES_ENABLE_INT();
```

```
    /*AES ECB模式加密*/
    AES_Open(0, 1, AES_MODE_ECB, AES_KEY_SIZE_128, AES_IN_OUT_SWAP);
    /*设置加密密码*/
    AES_SetKey(0, au32MyAESKey, AES_KEY_SIZE_128);
    /*设置初始化向量*/
    AES_SetInitVect(0, au32MyAESIV);
    /*配置DMA*/
    AES_SetDMATransfer(0, (uint32_t)au8InputData, (uint32_t) au8EncOutData,
sizeof(au8InputData));

    g_AES_done = 0;
    /*启动DMA，加密一个block就结束*/
    AES_Start(0, CRYPTO_DMA_ONE_SHOT);
    while (!g_AES_done);/*等待密码IP加密结束*/

    /*AES ECB模式解密*/
    AES_Open(0, 0, AES_MODE_ECB, AES_KEY_SIZE_128, AES_IN_OUT_SWAP);
    /*设置解密密码*/
    AES_SetKey(0, au32MyAESKey, AES_KEY_SIZE_128);
    /*设置初始化向量*/
    AES_SetInitVect(0, au32MyAESIV);
    /*配置DMA*/
    AES_SetDMATransfer(0, (uint32_t) au8EncOutData, (uint32_t) au8DecOutData,
sizeof(au8InputData));

    g_AES_done = 0;
    /*启动DMA，解密一个block就结束*/
    AES_Start(0, CRYPTO_DMA_ONE_SHOT);
    while (!g_AES_done); /*等待密码IP解密结束*/
}
```

最后将看到au8DecOutData里面的数据等于au8InputData里面的数据。如果要加密的数据不是16B的倍数，硬件将自动补足。加密输出的字节数是16B的倍数。

3.21 SC 初始化

Smart Card IP接口符合ISO7816规范。其实ISO7816收/发数据的波形跟UART一样：1bit START+8bit 数据 + 1bit 偶校验 + 2bit STOP。

按照ISO7816的规定，发送数据之后多久卡必须回数据，不然就认为超时；连续发送数据之间至少需要间隔多久才可以；等等时序要求，都由硬件实现。虽然软件可以模拟这些超时时间，但是是无法过EMV2000认证的。另外，软件模拟也会花费MCU比较多的时间。关于SC协议，在4.3节ISO7816有详细的介绍。

SC接口有4根脚：SC_CLK、SC_DATA、SC_RST、SC_PWR。因为该四根脚之间有严格的时序要求，如果要符合ISO7816规范，卡槽和SC接口之间不要接任何开关、电容和电阻。如果不需要符合ISO7816规范，可以随意。但是SC_PWR给卡上电之后，需要额外的等待时间，这就需要用函数retval = SCLIB_ActivateDelay(2, FALSE, 33/* 额外等待时间 */); 代替retval = SCLIB_Activate(2, FALSE);

复位序列之后，卡会回ATR字串，能准确无误拿到该字串，就说明SC接口接的基本没有问题了

新唐有提供SC library，大家只要调用该library就可以了。

下面介绍一下用该library，如何拿到ATR。

中断处理函数：

```
/*我们使用SC2接到SC卡卡槽*/
void SC2_IRQHandler(void)
{
    // Please don't remove any of the function calls below
    if(SCLIB_CheckCDEvent(2))
        return; // Card insert/remove event occurred, no need to check other event...
    SCLIB_CheckTimeOutEvent(2);/*处理超时事件*/
    SCLIB_CheckTxRxEvent(2); /*处理收发事件*/
    SCLIB_CheckErrorEvent(2); /*处理出错事件*/

    return;
}
```

系统初始化：

```
void SC2_Init()
{
    /*使能SC2的时钟*/
    CLK_EnableModuleClock(SC2_MODULE);
    /*SC2选择外部12M做时钟源，然后除频12M/3= 4M， SC_CLK将输出4M时钟*/
    CLK_SetModuleClock(SC2_MODULE, CLK_CLKSEL3_SC2SEL_HXT, CLK_CLKDIV1_SC2(3));
    /*配置多功能引脚*/
    SYS->GPA_MFPL = SYS_GPA_MFPL_PA2MFP_SC2_DAT | SYS_GPA_MFPL_PA3MFP_SC2_CLK |
            SYS_GPA_MFPL_PA4MFP_SC2_PWR | SYS_GPA_MFPL_PA5MFP_SC2_RST;
    /*打开SC2，不检测卡插入引脚，SC_PWR引脚拉high有效*/
    SC_Open(SC2, SC_PIN_STATE_IGNORE, SC_PIN_STATE_HIGH);
    NVIC_EnableIRQ(SC2_IRQn);
}
```

Cold RESET IC卡，拿到ATR之后，发送一个命令给IC卡

```
/*要发送到IC卡的命令，以及命令长度*/
```

```
uint8_t g_cmd1buf[] = {0x00, 0xA4, 0x00, 0x00, 0x02, 0xDF, 0x01};
uint32_t g_cmd1len = 7;
void SC2_Test()
{
    SC2_Init();
    /* 等待卡插入，如果没有卡检测引脚，该函数将马上返回 */
    while(SC_IsCardInserted(SC2) == FALSE);
    // 发送Activate Timing到slot 2
    retval = SCLIB_ActivateDelay(2, FALSE, 33);/*如果PWR上电之后，需要比较久的时间稳定，用这个函数传入额外的等待时间，33个ETU*/
    /*成功拿到ATR*/
    if(retval == SCLIB_SUCCESS) {
        /*打印ATR的内容*/
        SCLIB_GetCardInfo(2, &s_info);
        printf("ATR: ");
        for(i = 0; i < s_info.ATR_Len; i++)
            printf("%02x ", s_info.ATR_Buf[i]);
        printf("\n");
        /*发送命令给IC卡*/
        SCLIB_StartTransmission(2, g_cmd1buf, g_cmd1len, g_cmdrsp, &g_cmd1rsplen);
        printf("get response\n");
        /*打印响应的结果*/
       for(i = 0; i < g_cmd1rsplen; i++)
           printf("%02x ", g_cmdrsp[i]);
        printf("\n");
    }
}
```

3.22 PS2D 初始化

新唐的芯片有的支持PS2 Device IP，可以接到PC上有PS2 Host接口的设备上。PS2 接口就两根线CLK和DATA，硬件提供16个字节的发送FIFO，用于缓存数据，接收没有FIFO。

1) PS2 Device 发送到 Host 的帧结构如下，类似 UART 的波形：1-bit START + 8-bit DATA+1-bit Parity + 1-bit STOP，奇校验：

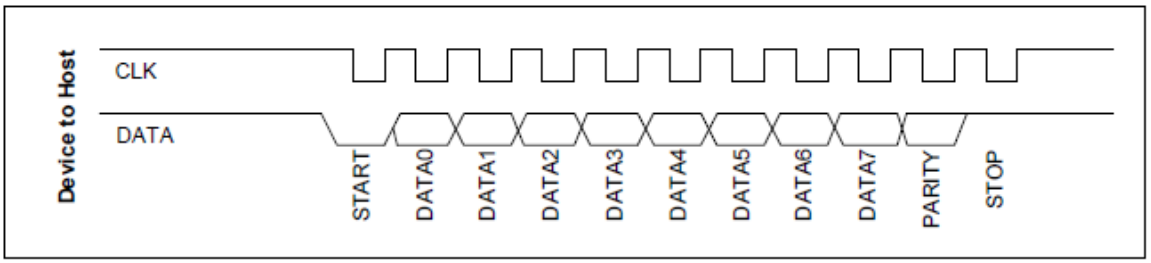


Figure 5-88 Data Format of Device-to-Host

当CLK为高电平时，Device可以改变数据；CLK为低电平时Host会将数据读走。连续两个字节之间的间隔为100us

2) PS2 Host 发送到 Device 的帧结构如下，也类似于 UART 的波形，但是多了一个 ACK 位：1-bit START + 8-bit DATA+1-bit Parity + 1-bit STOP + ACK，奇校验

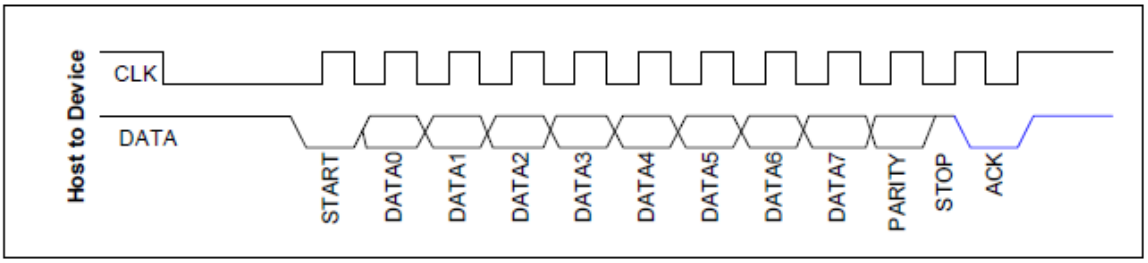


Figure 5-89 Data Format of Host-to-Device

CLK为低电平时Host改变数据，CLK为高电平时Device将数据读走。

Host要发送数据，必须先将CLK强制拉low 100us，然后将DATA拉low，再释放CLK。之后Device就会在CLK脚上产生时钟信号，一帧发送完毕，Device会将DATA 拉low，回ACK信号给Host。ACK信号之后，如果Host不释放DATA，Device将继续产生时钟信号，直到DATA被释放。

注意：PS2CLK和PS2DATA两根引脚配置为PS2功能之后，仍可以通过PS2CON寄存器的FPS2DAT和FPS2CLK这两个bit来控制这两根脚的状态，同时通过PS2STATUS寄存器来查看这两根脚的状态。

```
void PS2_Init(void)
{
    /* 使能P2SD IP的时钟 */
    CLK_EnableModuleClock(PS2_MODULE);
    /* 将PF.2和PF.3配置位PS2 PS2DAT 和 PS2CLK */
    SYS->GPF_MFP = SYS_GPF_MFP_PF2_PS2_DAT | SYS_GPF_MFP_PF3_PS2_CLK;
```

```
    PS2_Open();
    /*使能PS2D中断*/
    PS2_EnableInt(PS2_PS2CON_RXINTEN_Msk | PS2_PS2CON_TXINTEN_Msk);
    NVIC_EnableIRQ(PS2_IRQn);
}
/*中断处理函数*/
void PS2_IRQHandler(void)
{
    uint32_t u32RxData;

    /* 接收中断 */
    if(PS2_GET_INT_FLAG(PS2_PS2INTID_RXINT_Msk))
    {
        /* 清除PS2 接收中断标志 */
        PS2_CLR_RX_INT_FLAG();
        /* 读数据 */
        u32RxData = PS2_Read();
    }
    /* 发送中断 */
    if(PS2_GET_INT_FLAG(PS2_PS2INTID_TXINT_Msk))
    {
       PS2_CLR_TX_INT_FLAG();
    }
}
```

有了上面的代码，就可以通过PS2 接口收发数据了。

PS2初始化代码照例分为

- 使能 PS2D 时钟，配置 PS2D 多功能引脚
- PS2D 功能初始化

3.23 EMAC 初始化

新唐NUC472有带一个EMAC接口，符合IEEE802.3协议，支持10M/100Mbps，支持连接MII和RMII接口的PHY。

NUC472可以run到84M，此速度下，所有SRAM都给EMAC用的情况下，TCP连接下，传输速度可以达到18Mbps。一般情况下，SRAM不会全部给EMAC使用，使用EMAC的速度实际上慢多了。

默认BSP里面定义了4个发送descriptor，4个接收descriptor。这些descriptor轮流给EMAC和CPU

使用。接收时，descriptor 先切给EMAC使用，接收一包结束，EMAC会自动切给CPU使用，CPU用完再切给EMAC使用。发送时，descriptor先给CPU使用，要发送的数据填到descriptor里面之后，切给EMAC使用，EMAC发送完毕自动再切给CPU使用。

```
Void EMAC_Init()
{
    /*使能EMAC时钟*/
    CLK_EnableModuleClock(EMAC_MODULE);
    // 配置MDC 时钟频率: HCLK / (127 + 1) = 656 kHz ，系统run在 84 MHz
    CLK_SetModuleClock(EMAC_MODULE, 0, CLK_CLKDIV3_EMAC(127));
    // 配置 RMII 引脚
    SYS->GPC_MFPL = SYS_GPC_MFPL_PC0MFP_EMAC_REFCLK |
                        SYS_GPC_MFPL_PC1MFP_EMAC_MII_RXERR |
                        SYS_GPC_MFPL_PC2MFP_EMAC_MII_RXDV |
                        SYS_GPC_MFPL_PC3MFP_EMAC_MII_RXD1 |
                        SYS_GPC_MFPL_PC4MFP_EMAC_MII_RXD0 |
                        SYS_GPC_MFPL_PC6MFP_EMAC_MII_TXD0 |
                        SYS_GPC_MFPL_PC7MFP_EMAC_MII_TXD1;


    SYS->GPC_MFPH = SYS_GPC_MFPH_PC8MFP_EMAC_MII_TXEN;
    // 使能RMII接口相关引脚 high slew rate
    PC->SLEWCTL |= 0x1DF;

    // 配置 MDC, MDIO at PB14 & PB15
    SYS->GPB_MFPH = SYS_GPB_MFPH_PB14MFP_EMAC_MII_MDC | SYS_GPB_MFPH_PB15MFP_EMAC_MII_MDIO;

    // 配置EMAC IP，默认使能 RMII 接口
     EMAC_Open(g_au8MacAddr);
     /*使能发送和接收中断*/
     NVIC_EnableIRQ(EMAC_TX_IRQn);
     NVIC_EnableIRQ(EMAC_RX_IRQn);
     /*启动EMAC开始发送和接收*/
     EMAC_ENABLE_RX();
     EMAC_ENABLE_TX();

     While(1);
}
/*发送中断*/
void EMAC_TX_IRQHandler(void)
{
```

```
    // 移动发送Descriptor指针，发送下一包数据
    EMAC_SendPktDone();
}
uint8_t auPkt[1514];
uint32_t u32PktLen;
/*接收中断*/
void EMAC_RX_IRQHandler(void)
{
    while(1) {
        // 查看是否有收到数据包，如果有，将数据存到auPkt里面
        if(EMAC_RecvPkt(auPkt, &u32PktLen) == 0)
            break;
        // 分析收到的数据
        process_rx_packet(auPkt, u32PktLen);
        // 将当前接收Descriptor切给EMAC使用，并触发接收下一包
        EMAC_RecvPktDone();
    }
}
```

4 中级篇

4.1 CAN

新唐的芯片有的带1个CAN，有的带2个CAN。都是BOSCH的C_CAN，所以它的稳定性或者错误处理能力是毋庸置疑的。

4.1.1 CAN 协议介绍

CAN协议包含4部分：帧结构、错误侦测、错误抑制和比特率

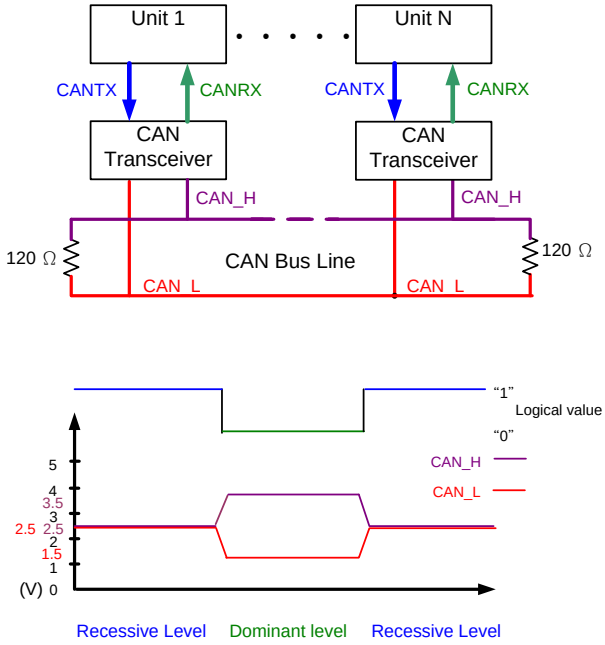
4.1.1.1 总线物理特性

CAN在1986年由BOSCH研发，用于汽车电子。1993年成为国际标准ISO 11891-1。

CAN总线除了电源和地线还有2根线：CAN_H和CAN_L，串行差分传输。实际接线图注意各有120欧姆终端电阻，用于避免信号反射 。

CAN定义了两种状态：显性(Dominant)和隐性(Recessive)，定义如下图。其实显性就是逻辑'0'，隐性就是逻辑'1'。因为CAN总线采用线'与'的规则进行总线仲裁：1&0 = 0，0胜出，所以0为显性。

0.5<CAN_H - CAN_L <0.9 状态未知



ISO11898(125kpbs to 1Mbps)

Recessive

CAN_H - CAN_L < 0.5v

Dominant

CAN_H - CAN_L > 0.9v

CAN总线通讯速度1K ~ 1Mbps， 1Mbps大概可以传40m，5kbps可以传10公里。

4.1.1.2 冲突检测

CAN总线是多主机的，所以必须加入冲突检测，方法为CSMA/CR：冲突检测多路访问/冲突解决。

基本思想：任何站点要向总线发送信息时，首先要侦听总线上是否有其他站点正在传送信息，如果总线空闲，则可以进行传送；如果已监听到介质上有载波，即有其它站点正在传送信息，则必须等待介质平静之后才能进行传送的处理，这样就会使信道上的冲突大大减少。CAN总线以消息的方式传递数据，速度1Kbps到1Mbps。通过远程帧，从远程节点请求数据；通过数据帧发送数据到远程节点。支持错误检测，错误通知，错误恢复和错误限制。

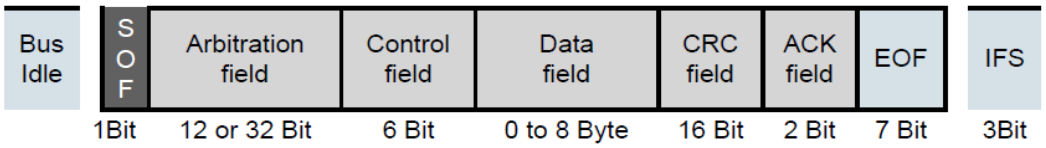
这些错误处理的机制导致CAN的一大特点：抗干扰能力很强。有人会将CAN和RS485比较，认为都是差分传输的，抗干扰能力是不是一样？差别在出错之后的处理。CAN检测到错误，硬件就帮忙处理了，而RS485需要软件做；另外CAN的传输距离非常远，5kbps可以达到10公里

4.1.1.3 帧结构

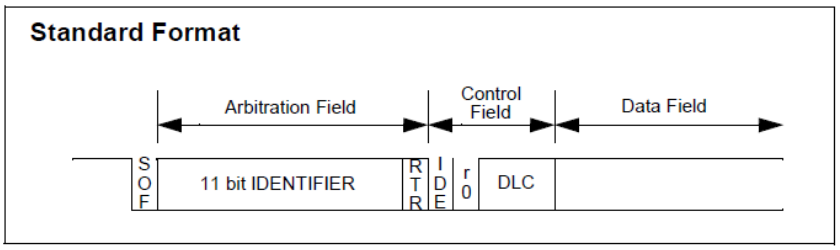
CAN有两种帧格式，11比特ID和29比特ID，就是发送的ID域长度不一样，这两种帧分别称为标准帧和扩展帧。

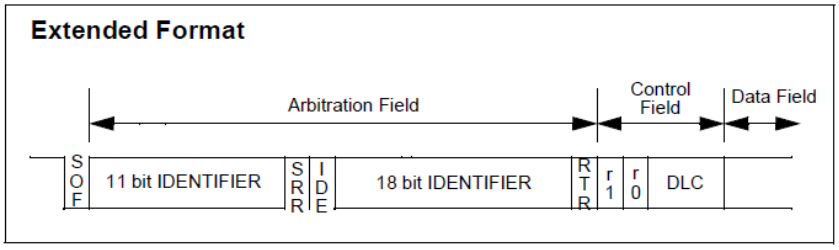
帧类型可以分为数据帧、远程帧、错误帧和超载帧。另外数据帧和远程帧之间还有帧间隔。下面详细介绍这四种帧类型

1) 数据帧，数据帧是用来发送数据给对方的，由 7 个不同的域组成，支持标准和扩展帧。大家可以看这 7 个域，仲裁域可以是 12 比特或者 32 个比特，就是因为 ID 的长度不同导致的



下图可以看出仲裁域(Arbitration Field)的ID有11-bit和29-bit两种



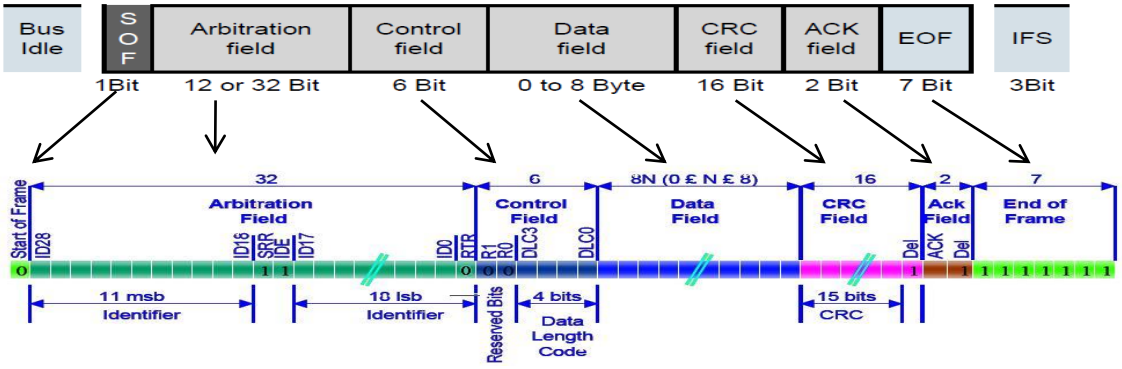


大家看上面这两张图就会明白标准格式和扩展格式数据帧的不同。标准格式ID为11位，扩展格式ID为29位。标准帧的RTR对应扩展帧的SRR位置，就是说它们两个在同一个位置，只是名字不同而已，IDE位置是一样的。RTR是远程帧和数据帧的标志位，数据帧RTR为0，远程帧RTR为1，扩展帧SRR固定为1。根据线与的原则可以看出，标准数据帧优先级高于扩展数据帧。IDE位是用来标志是否为扩展帧的，标准帧IDE为0，扩展帧IDE为1。

就是说到RTR/SRR这里，如果是0就是标准数据帧，如果是1需要再看IDE，如果IDE是0，就是标准远程帧，如果是1就是扩展帧，然后再看扩展帧中RTR位置的值决定是扩展数据帧还是扩展远程帧。

下图更详细的说明了各个域的长度：

- 帧起始标志 1-bit
- 仲裁域有 12-bit 或者 32-bit
- 控制域 6-bit
- 数据域最多 8 个字节
- CRC 域 16-bit
- ACK 域 2-bit
- 帧结束标志 7-bit
- 3-bit 的帧间隔

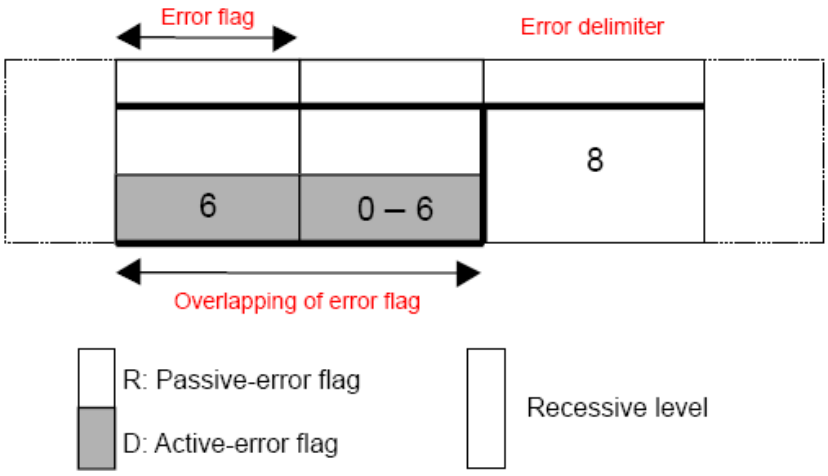


位；接下来是

控制域，这个域主要表示下面的数据域的长度是多少。然后是CRC校验和应答域，最后是帧结束。从这个帧就可以知道，要发送一个CAN的帧需要设定哪些域，包括：ID域，远程/数据帧、扩展帧/标准帧、发送的数据长度、以及要发送的数据

2) 远程帧。远程帧和数据帧非常类似，只是远程帧没有数据域。远程帧是用来**请求对方**发送数据用的，远程帧也支持标准和扩展帧。数据帧的 RTR 域填 1，然后拿掉数据域就是远程帧

3) 错误帧，顾名思义就是发生错误时发送的帧，不管什么错误都发送这个帧。错误帧的作用不是用来通知其他节点发生了什么错误，而只是通知所有节点发生了错误。错误帧分两个域，ERROR flag 域和分割符。Error flag 域为各个节点发送的 Error Flag 的叠加。Error flag 为何会叠加呢？因为一个节点发现错误发送 6 个 Error Flag 出来，根据填充原则：不可以有 6 个连续相同的比特，所以其他节点将发现填充错误，也会发送错误帧，这样 Error Flag 最多会有 12 个。Error flag 有 passive 和 Acitve 之分，这是因为节点处的状态不同，处于 ACTIVE 的节点，就发送 Active error flag,就是发 6 个‘0’，处于 Passive 状态的节点就发送 Passive error flag,就是发 6 个‘1’。节点的状态变化在后面讲

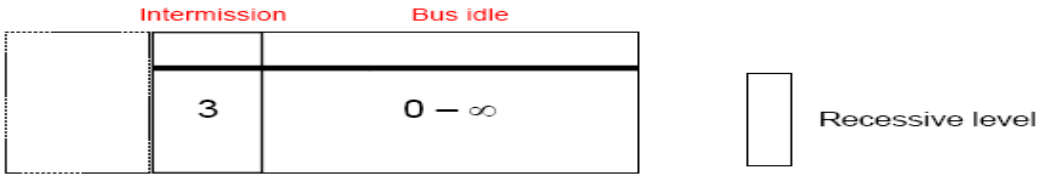


4) 超载帧。顾名思义就是某个节点来不及处理数据了，希望其他节点慢点发送数据帧或者远程帧。
另外还有多种情况也可以发送超载帧：接收端点 EOF 域最后一个 bit 检测到 0，会认为是有人要抢发，节点可以发送超载帧干扰发送。不知道大家有注意到没有，超载帧和错误帧是一样的。之所以有两个名字，是因为如果发送的是超载帧，错误计数不会增加，否则就会增加。错误计数就关系到节点的状态变化。

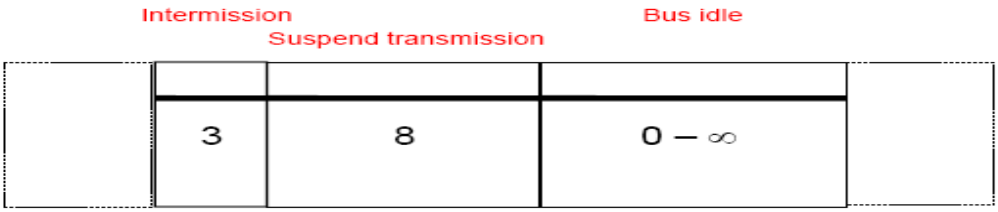
5) 帧间隔。用来间隔数据帧和远程帧。就是数据帧和数据帧之间，远程帧和远程帧之间，数据帧和远程帧之间都会加入帧间隔。帧间隔如图所示有两种格式，一种是 3 个 1，一种是

3+8 个 1。多出来的 8 个 1 是当节点是 passive 状态，并且发送了帧。为什么呢？因为处于 passive 状态的节点，本身错误计数就很多了，所以要减缓它发送的 bit 数据量

非‘error passive’的节点 或者为前一个帧的接收者.



‘error passive’的节点，并且前一个帧是该节点发送的.



4.1.1.4 错误侦测

错误有5种，都会导致发送错误帧。

- 比特错误，发送和监控到的比特不同；
- 填充错误，侦测到连续 6 个比特电平相同；
- 校验错误，收到帧之后计算出的 CRC 值和接收到的不同；
- 格式错误，某个固定格式的地方出现无效值
- 应答错误，发送端在应答域没有检测到 0

错误类型	出错条件	出错域	侦测单元
Bit Error	发送的比特和监控到的比特值不同.(填充比特和ACK比特除外)	• Data Frame (SOF~EOF) • Remote Frame (SOF~EOF) • Error Frame • Overload Frame	• Send Unit • Receive Unit
Stuff Error	侦测到6个连续相同电平	• Data Frame (SOF~CRC) • Remote Frame (SOF~CRC)	• Send Unit • Receive Unit

CRC Error	计算结果和收到的CRC不同	• Data Frame (CRC) • Remote Frame (CRC)	• Receive Unit
FORM Error	某个固定的格式位置出现无效比特	• Data Frame (CRC Delimiter, ACK Delimiter, EOF) • Remote Frame (CRC Delimiter, ACK Delimiter, EOF) • Error Frame Delimiter • Overload Delimiter	• Receive Unit
ACK Error	发送端在应答域没有收到0	• Data Frame (ACK slot) • Remote Frame (ACK slot)	• Send Unit

仲裁域如果发生bit error，不算错误，错误计数值不会增加。其它域发生bit error，会导致发送Active Error Flag或者Passive Error Flag。仲裁失败之后，要等对方发送完毕(EOF 7-bit)，BUS空闲3 bit time之后才能再次重传。

4.1.1.5 错误计数

前面提到节点会有很多状态，例如：Active状态、Passive状态

节点根据错误计数值改变状态，根据不同的错误条件 ,TEC和REC会有不同的变化(增加或减少)

而TEC和REC的总数,会导致节点状态的改变。

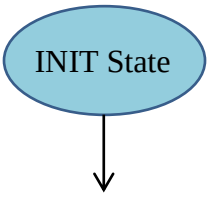
C_CAN内部Error counter的增减,也是根据这个表来做的

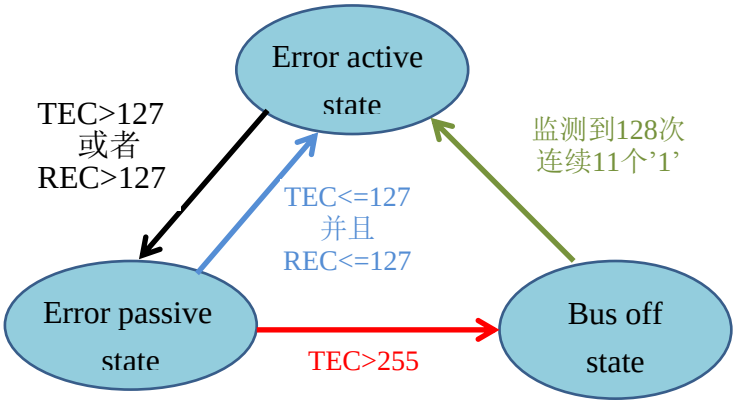
	错误条件	Transmit Error Counter (TEC)	Receive Error Counter (REC)
1	RECEIVER 端侦测到一个BIT ERROR错误，除了发送ACTIVE ERROR FLAG 和OVERLOAD FLAG时	-	+ 1
2	TRANSMITTER 发送 ERROR FLAG时	+ 8	
3	TRANSMITTER 端发送ACTIVE ERROR FLAG 或者OVERFLAG FLAG时侦测到 BIT ERROR	+ 8	

4	RECEIVER 发送ACTIVE ERROR FLAG 或者OVERFLAG FLAG时侦测到 BIT ERROR		+ 8
5	一个帧被成功发送之后（取得ACK 并且直到END OF FRAME 完成都没有错误）	- 1 If TEC=0, TEC will not be changed.	-
6	一个帧被成功接收（直到ACK域都没有检测到错误，并成功发送ACK比特）	-	1. if 1 <= REC <= 127 → REC -1 2. if REC = 0 → REC = 0 3. if REC > 127 → REC = a value between 119 to 127
7	在总线上检测到128 次连续的 11 个1，”bus oof”的节点允许变成不再是“bus off“.	Cleared to TEC = 0	Cleared to REC = 0

4.1.1.6 错误抑制

一个节点挂到CAN总线上之后，处于ACTIVE状态；TEC>127或者REC>127导致节点进入passive 状态；TEC>255之后节点处于bus off状态，就是不允许再往bus上发送东西了；处于bus off状态的节点，在检测到128个连续的11个1之后将回到active状态。

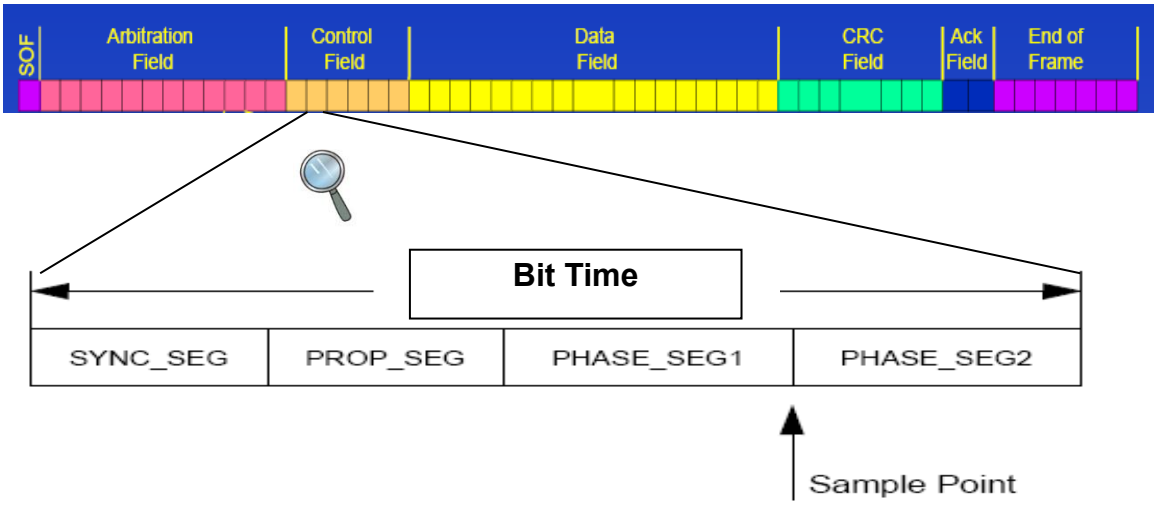




节点各个状态改变如上图所示

4.1.1.7 比特率

接下来给大家讲一下比特率，这个是CAN通讯的关键，比特率决定每个比特占用的时间，如图所示分为4部分：同步段、传输延迟、PHASE_SEG1、PHASE_SEG2。采样点落在PHASE_SEG1、PHASE_SEG2之间最好。



这4个段的时间单位称为tq: time quantum 的缩写，时间长度为1/CAN 总线工作频率。同步段长度固定，为一个tq，传输延迟段[1，8]个tq；PHASE_SEG1为[1，8]个tq；PHASE_SEG2为[1,8]个tq。比特时间为X = SYNC_SEG+PROP_SEG+PHASE_SEG1+PHASE_SEG2，X的范围为[8,25]个tq。假设希望的波特率为B，CAN总线工作频率为R则：B=R/X。所以决定波特率的关键有两个，CAN总线的工作频率和每个比特占用的tq数就是X的值。

4个段中满足上述条件的的有很多组合，随便挑一个就可以。

Parameter	Range	Remark
BRP	[1 .. 32]	defines the length of the time quantum tq
Sync_Seg	1 tq	fixed length, synchronization of bus input to APB clock
Prop_Seg	[1.. 8] tq	compensates for the physical delay times
Phase_Seg1	[1..8] tq	may be lengthened temporarily by synchronization
Phase_Seg2	[1.. 8] tq	may be shortened temporarily by synchronization
SJW	[1 .. 4] tq	may not be longer than either Phase Buffer Segment
This table describes the minimum programmable ranges required by the CAN protocol		

Table 5-16 CAN Bit Time Parameters

举个例子，CAN总线波特率公式如果写成这样：

CAN Speed (bps) = Fin / ((BPR+1)*(T1+T2+3))

where Fin: System clock freq.

BPR: System clock freq. divider

T1: Phase Segment 1 + Prop Segment - 1.

T2: Phase Segment 2 - 1.

Fin/(BRP+1)就是CAN总线的工作频率，就是上面的R。T1+T2+3就是每个比特所占时间。这个3的来由除了T1减掉1，T2减掉1之外，还有同步段固定的1个tq

例如：CPU runs 48MHz, Set T1=2,T2=3,BPR=5

➔CAN speed = 48000/(5+1)(2+3+3)=1000kps

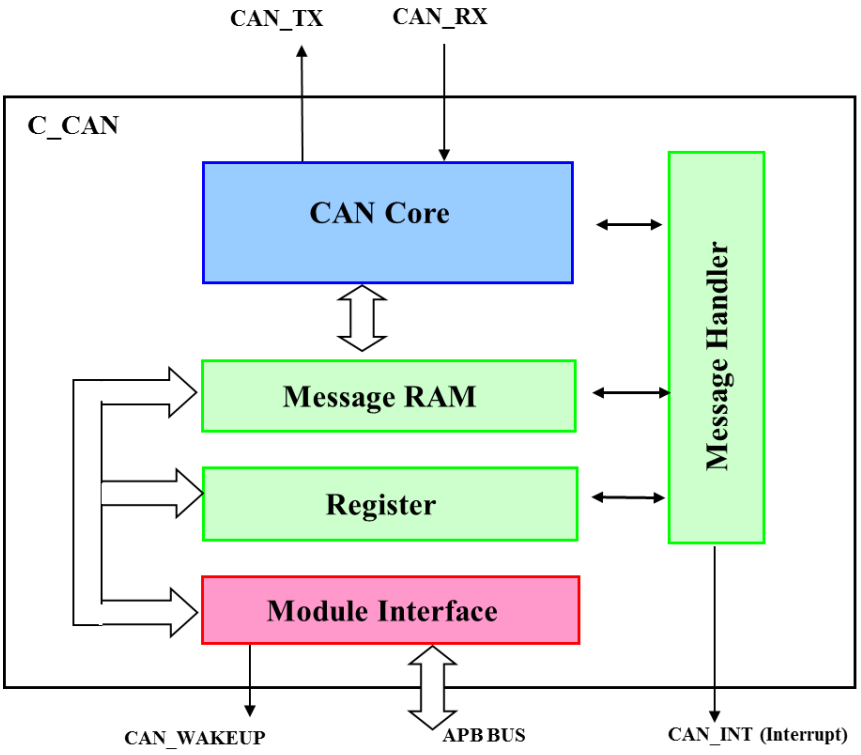
CAN协议中错误处理部分都由硬件实现了，软件要关心的协议部分就是数据帧和远程帧，ID的值，以及波特率。

下面讲解一下新唐CAN IP的用法。

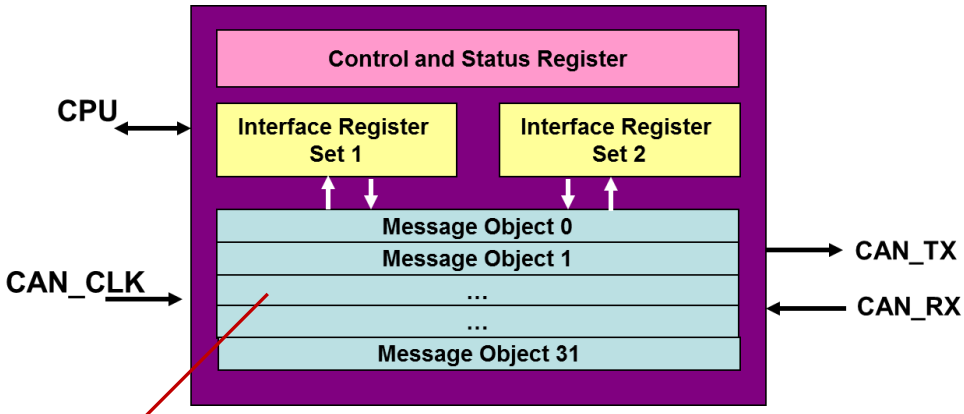
4.1.2 新唐 CAN IP 特点

新唐的CAN IP与CAN 2.0 PART A和B规格兼容。

内部有32个Message Object（就在Message RAM中），每个就是一个完整的CAN帧。IP内部结构如下图：



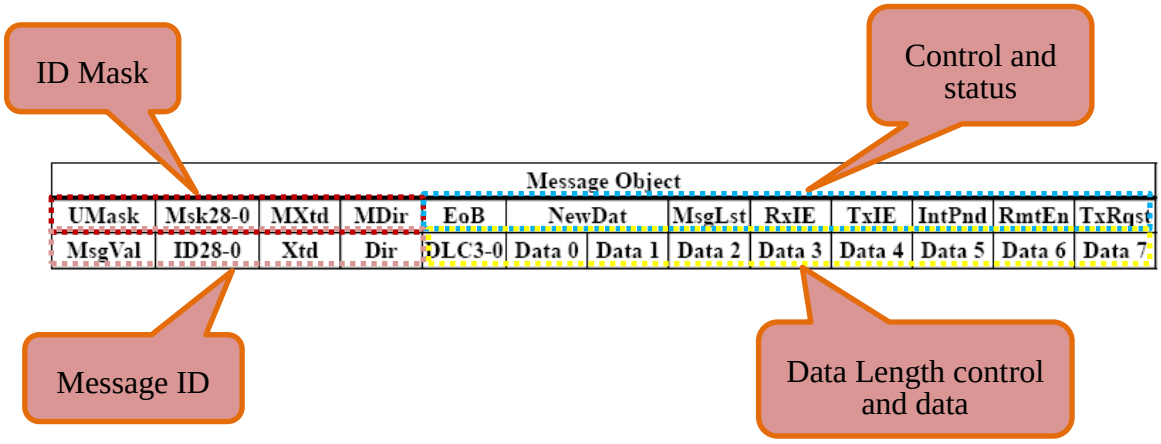
这32个Message Object每个都可以单独配置，也可以多个做成FIFO。由Message Handler和CAN Core共同访问。FIFO中最后一个Message Object的EoB=1，其它的EoB都为0。FIFO中所有的Message Object配置都一样，除了EoB。



Message Object												
UMask	Msk28-0	MXtd	MDir	EoB	NewDat		MsgLst	RxIE	TxIE	IntPnd	RmtEn	TxRqst
MsgVal	ID28-0	Xtd	Dir	DLC3-0	Data 0	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7

每个Message Object包含如上红色线指向的表中的信息。为了避免CPU读Message Object和Message Handler之间冲突，CPU不能直接访问Message Object，而只能通过2组接口寄存器：IF1和IF2。

每个接口的寄存器对应修改Message Object中下面的信息



上图中

- Msk28-0 定义在寄存器 IFn_MASK1 和 IFn_MASK2 中
- MXtd 和 MDir 定义在寄存器 IFn_MASK2 中
- EoB、NewDat、MsgLst、RxIE、TxIE、IntPnd、RmtEn、TxRqst 、DLC3-0、UMask 都定义在 IFn_MCON 寄存器中
- ID15-0 定义在 IFn_ARB1 寄存器中
- MsgVal、Xtd、Xdir 和 ID28-16 定义在 IFn_ARB2 寄存器中

ID Mask和Message ID是对应的，Message ID中的信息可以由ID Mask进行掩码。

- 没有设定 Mask 的时候，如果 BUS 上有个 Message ID 是 0x1, 而此节点 ID 也设为 0x1, 则此节点”会”将 Message 后面的 Data 收进来；反之, 如果此节点 ID 设为 0x3, 则此节点”不会”将 Message 后面的 Data 收进来
- 如果有设定 Mask 的时候,并将 Mask ID bit1 设为 0x0, 表示 bit1 “ don’t care”

依照上例, ID设为0x3的节点 “会”将BUS上0x1的Message后面的Data收进来

ID Mask：

- 掩码只跟接收数据有关，跟发送无关
- UMask: 是否使用掩码

- Msk28-0: ID 掩码
- MXtd: Xtd bit 掩码
- MDir: Message方向掩码
- 掩码为0表示 “Don’t care”

Message ID：

- 这些bit用来过滤CAN总线上的数据
- Dir: 发送 / 接收 (数据帧/远程帧)
- Xtd: 11-bits / 29-bits ID （标准帧/扩展帧）
- ID28-0: Message ID
- MsgVal: 该Message object 是否有效。要修改某个Message Object的内容，需要先将该位设为‘0‘，改好之后再恢复成‘1‘。

Control and status：

- EOB: 如果多个Message Object做FIFO的话，该位置‘1‘，说明是FIFO中最后一个Message Object。
- NewDat: 发送数据时，表示数据已经更新，数据开始发送时，HW会清0；接收数据时表示收到了新的数据
- MsgLst: 接收到数据的时候，如果NewDat 已经为1，表示上一包数据软件还没有来得及读走，该位将置为1
- RxIE: 成功接收到一个帧IntPnd将被置为1，中断将发生
- TxIE: 成功发送一个帧IntPnd将被置为1，中断将发生
- IntPnd:如果RxIE或者TxIE为‘1‘，如果该Message Object导致中断发生，该位将被置1
- RmtEn: 如果该位为‘1‘，收到远程帧，该Message Object 的TxRqst将由HW置1，返回数据给对方
- TxRqst: 如果该位为‘1‘，该Message Object请求发送。发送的可能是远程帧可能是数据帧，由Dir 决定，Dir = 0发送远程帧，Dir = 1发送数据帧。如果远程帧发送之前，先收到了数据帧，该Message Object的TxRqst将被清0，NewDat和IntPnd将被置为1，然后发生中断，通知MCU收到了数据

Data Length Control and Data就是要发送的数据长度和数据

下表列出发送数据帧、接收数据帧、发送远程帧、接收远程帧的配置。不论哪种情况下，ID都要配置，如果发送数据帧，则要发送的数据以及长度也要配置好。

作用	Rmten	Dir	Umask	TxRqst	NewDat
接收数据帧	0	0	0	0	1(hw)
接收到数据帧/远程帧都正常接收，如果RxIE被置，都会发生中断	0	x	1	0	1(hw)
发送远程帧	0	0	x	1	1(sw)
接收到数据帧/远程帧都丢弃	0	1	0	0	
发送数据帧	x	1	x	1	1(sw)
接收到远程帧HW将TxRqst置为1，并将该帧返回	1	1	x	0	

sw:表示由软件置1

hw:表示硬件会置1

4.1.2.1 接收数据帧

1) 软件配置 Message Object
   - 只接收数据帧：Dir = 0， RmtEn = 0，UMask = 0，TxRqst = 0
   - 数据帧和远程帧都接收：Dir = 0， RmtEn = 0，UMask = 1，TxRqst = 0
   - 接收远程帧，并且回数据帧给对方：Dir = 1， RmtEn = 1，UMask = 0/1，TxRqst = 0
2) 收到数据
   - 如果 RxIE 被置，则 IntPnd 将被置，中断将发生，然后读 CAN_IIDR 寄存器，如果值为 0x00001 ~ 0x0020 就是 Message Object 导致的中断
   - 写 0x7F 到 CMASK 寄存器，然后写要读的 Message Object 的 Number 到 CREQ 寄存器。将整个 Message Object 的内容传到 IFn 接口寄存器，同时将 IntPnd 和 NewDat 清为 0

4.1.2.2 发送数据帧

1) 软件配置 Message Object
   - Dir = 1，RmtEn = 0，TxRqst = 1
2) 数据帧发送完毕

- 如果 TxIE 被置 IntPnd 将被置 1，中断将发生，然后读 CAN_IIDR 寄存器，如果值为 0x00001 ~ 0x0020 就是 Message Object 导致的中断，可以得知是哪个 Message Object 发送完毕

3) 更新数据
   - 更新数据的时候，Message Object 里面的 MsgVal 和 TxRqst 不需要清为 0
   - 即使只更新一个字节，DAT_A1+DAT_A2 或者 DAT_B1+DAT_B2 寄存器中的 4 个字节必须一起写到 Message Object 里面
   - 写 0x87 到 CMASK 寄存器，然后写要更新的 Message Object 的 Number 到 CREQ 寄存器。数据将更新到 Message Object 里面，同时 TxRqst 将被置为 1
   - 如果更新数据的时候，该 Message Object 已经在发送了。发送完之后，TxRqst 将被 HW 置为 0。为了防止这种情况发生，可以将 MCON 寄存器中的 NewDat 一起置位，写到 Message Object 中。这样 NewDat 将阻止 TxRqst 被清为 0，刚更新的数据会再次被发送。

4.1.2.3 接收远程帧

1) 软件配置 Message Object
   - Dir = 1，RmtEn = 1，无论 UMask = 1/0，接收到远程帧之后，TxRqst 由 HW 置 1，然后该 Message Object 的数据将返回。如果 RxIE 和 TxIE 任意一个使能，都将发生中断。
   - Dir = 1，RmtEn = 0，UMask = 0，收到的远程帧将被丢弃
   - Dir = 1，RmtEn = 0，UMask = 1，收到的远程帧中的 ID + IDE + RTR + DLC 将保存到 Message Object 里面，并且 NewDat 将由 HW 置 1，有点像收到数据帧。如果 RxIE 使能将发生中断

4.1.2.4 发送远程帧

1) 软件配置 Message Object
   - Dir = 0，TxRqst = 1
2) 如果返回数据
   - 如果 RxIE 被置为 1，IntPnd 将被置为 1，中断将发生

4.1.2.5 中断处理

发生中断时，CAN_IIDR寄存器中读到的是优先级最高的中断号，而与Message Object发生中断的时间无关。

状态中断有最高的优先级，CAN Core更新CAN_STATUS寄存器就会发生该中断。CAN中断是否发生由CA_CON bit[3](EIE)和bit[2](SIE)决定。

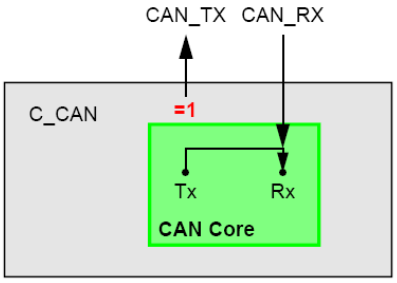
Message中断通过清除IntPnd清除，状态中断通过读CAN_STATUS寄存器清除。

4.1.2.6 特殊模式

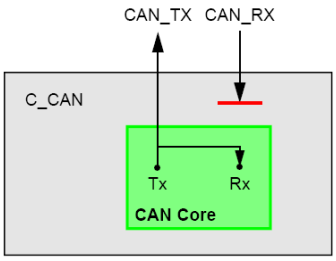
- **D**isabled **A**utomatic **R**etransmission mode

- Silent Mode
- Loopback Mode
- Basic Mode

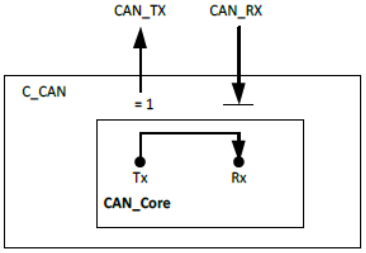
通过下面两张图，大家可以很容易理解Silent Mode和Loopback Mode。如果Silent Mode和Loopback Mode同时使能，就相当于Tx和Rx接到一起，并且不接到CAN BUS上。



Silent Mode



Loopback Mode



Silent Mode&
LoopbackMode

数据发送时如果发生仲裁失败，数据会自动重传。

Basic Mode就是如果你不想用Message Object的方式收发数据，可以直接用IF1和IF2这两个接口的寄存器直接收发。

4.1.3 代码分析

下面的代码分为发送和接收两部分，接收方配置0、5、31共3个Message Object，接收ID分别为0x7FF、0x12345、0x7FF01的数据帧。发送方配置1、2、3共3个Message Object，发送ID分别为0x7FF、0x12345、0x7FF01的数据帧。发送方发送完毕退出，接收方接收完毕也退出。双方CAN_Init函数是一样的，所以只在接收代码里面放了一份，发送代码里面照抄就行了。

CAN IP的初始化也是包括：使能时钟、配置多功能引脚、CAN IP功能配置

1) 接收代码

```
STR_CANMSG_T rrMsg[32];
static volatile uint32_t g_MessageNum;
/*初始化CAN*/
Void CAN_Init()
{
    /*使能CAN0 的时钟*/
    CLK_EnableModuleClock(CAN0_MODULE);
    /* 配置 CANTX0和CANRX0 引脚*/
```

```
    SYS->GPD_MFP &= ~(SYS_GPD_MFP_PD6_Msk | SYS_GPD_MFP_PD7_Msk);
    SYS->GPD_MFP = SYS_GPD_MFP_PD6_CAN0_RXD | SYS_GPD_MFP_PD7_CAN0_TXD;

    /* 配置比特率 */
    CAN_Open(CAN0,  500000, CAN_NORMAL_MODE);//500K
}
/*接收函数*/
void Test_NormalMode_Rx(CAN_T *tCAN)
{
     /*配置Message Object Number 0，接收ID为0x7FF的标准帧 */
    if(CAN_SetRxMsg(tCAN, MSG(0), CAN_STD_ID, 0x7FF) == FALSE) {
        printf("Set Rx Msg Object failed\n");
        return;
    }
    /*配置Message Object Number 5，接收ID为0x12345的扩展帧 */
    if(CAN_SetRxMsg(tCAN, MSG(5), CAN_EXT_ID, 0x12345) == FALSE) {
        printf("Set Rx Msg Object failed\n");
        return;
    }

    /*配置Message Object Number 31，接收ID为0x7FF01的扩展帧 */
    if(CAN_SetRxMsg(tCAN, MSG(31), CAN_EXT_ID, 0x7FF01) == FALSE) {
        printf("Set Rx Msg Object failed\n");
        return;
    }
#ifndef USING_INTERRUPT
     /* 轮询中断Message ID */
    while(1) {
        while(tCAN->IIDR == 0);                  /* 等待 IDR 改变 */
        /* 读Messsage Number为tCAN->IIDR – 1的Message Object，并清除中断标志和NewDat标志 */
        CAN_Receive(tCAN, tCAN->IIDR - 1, &rrMsg[tCAN->IIDR - 1]);
    }
#else
    /*使用中断函数处理Message Object*/
    CAN_EnableInt(tCAN, CAN_CON_IE_Msk);     /* 使能 CAN Message ID 改变中断 */
    NVIC_EnableIRQ(CAN0_IRQn);
#endif
}
/*中断处理函数*/
void CAN0_IRQHandler(void)
{
```

```
    uint32_t u8IIDRstatus;

    u8IIDRstatus = CAN0->IIDR;

    if(u8IIDRstatus == 0x00008000) {      /* 状态中断（错误状态和状态改变） */

        if(CAN0->STATUS & CAN_STATUS_RXOK_Msk) {
            CAN0->STATUS &= ~CAN_STATUS_RXOK_Msk;   /* 接收到数据包，与过滤规则无关 */
        }
        if(CAN0->STATUS & CAN_STATUS_TXOK_Msk) {
            CAN0->STATUS &= ~CAN_STATUS_TXOK_Msk;    /* 发送成功*/
        }
        /* 出错 */
        if(CAN0->STATUS & CAN_STATUS_EWARN_Msk) {
        }
        if(CAN0->STATUS & CAN_STATUS_BOFF_Msk) {
        }
    } else if(u8IIDRstatus != 0) {
        /*记录哪几个Message Object收到过数据*/
        g_MessageNum |= (1<<(u8IIDRstatus-1));
        /*接收数据并清除中断标志*/
        CAN_Receive(CAN0, u8IIDRstatus-1, &rrMsg[u8IIDRstatus-1]);
    } else if(CAN0->WU_STATUS == 1) {
        CAN0->WU_STATUS = 0;    /* Write '0' to clear */
    }
}
```

2) 发送方代码

```
/*发送函数*/
void Test_NormalMode_Tx(CAN_T *tCAN)
{
    STR_CANMSG_T tMsg;

    /* 发送 11-bit ID的标准帧，包含2个字节的数据域 */
    tMsg.FrameType = CAN_DATA_FRAME;
    tMsg.IdType    = CAN_STD_ID;
    tMsg.Id        = 0x7FF;
    tMsg.DLC       = 2;
    tMsg.Data[0]   = 7;
    tMsg.Data[1]   = 0xFF;
```

```
    /*将信息写到Message Object里面，并触发发送*/
    if(CAN_Transmit(tCAN, MSG(1), &tMsg) == FALSE) {
        printf("Set Tx Msg Object failed\n");
        return;
    }

    /* 发送 29-bit ID的扩展帧，包含3个字节的数据域 */
    tMsg.FrameType = CAN_DATA_FRAME;
    tMsg.IdType    = CAN_EXT_ID;
    tMsg.Id        = 0x12345;
    tMsg.DLC       = 3;
    tMsg.Data[0]   = 1;
    tMsg.Data[1]   = 0x23;
    tMsg.Data[2]   = 0x45;
    /*将信息写到Message Object里面，并触发发送*/
    if(CAN_Transmit(tCAN, MSG(2), &tMsg) == FALSE) {
        printf("Set Tx Msg Object failed\n");
        return;
    }

    /* 发送 29-bit ID的扩展帧，包含4个字节的数据域 */
    tMsg.FrameType = CAN_DATA_FRAME;
    tMsg.IdType    = CAN_EXT_ID;
    tMsg.Id        = 0x7FF01;
    tMsg.DLC       = 4;
    tMsg.Data[0]   = 0xA1;
    tMsg.Data[1]   = 0xB2;
    tMsg.Data[2]   = 0xC3;
    tMsg.Data[3]   = 0xD4;
    /*将信息写到Message Object里面，并触发发送*/
    if(CAN_Transmit(tCAN, MSG(3), &tMsg) == FALSE) {
        printf("Set Tx Msg Object failed\n");
        return;
    }
    while(g_MessageNum != 0xE);//等待所有的Message Number 发送结束
}
/*中断处理函数*/
void CAN0_IRQHandler(void)
{
    uint32_t u8IIDRstatus;
```

```
    u8IIDRstatus = CAN0->IIDR;

    if(u8IIDRstatus == 0x00008000) {      /* 状态中断（错误状态和状态改变） */

        if(CAN0->STATUS & CAN_STATUS_RXOK_Msk) {
            CAN0->STATUS &= ~CAN_STATUS_RXOK_Msk;   /* 接收到数据包，与过滤规则无关 */
        }
        if(CAN0->STATUS & CAN_STATUS_TXOK_Msk) {
            CAN0->STATUS &= ~CAN_STATUS_TXOK_Msk;    /* 发送成功*/
        }
        /* 出错 */
        if(CAN0->STATUS & CAN_STATUS_EWARN_Msk) {}
        if(CAN0->STATUS & CAN_STATUS_BOFF_Msk) {}
    } else if(u8IIDRstatus != 0) {
        /*记录哪几个Message Object收到过数据*/
        g_MessageNum |= (1<<(u8IIDRstatus-1));
        /*清除中断标志*/
        CAN_CLR_INT_PENDING_BIT(CAN0, (u8IIDRstatus - 1));
    } else if(CAN0->WU_STATUS == 1) {
        CAN0->WU_STATUS = 0;    /* 写 '0' 清除 */
    }
}
```

调用Test_NormalMode_Rx(CAN_T *tCAN)之后，Message Object就准备好接收指定的ID。

找两块板子一块run Test_NormalMode_Rx(CAN_T *tCAN)，一块run发送函数Test_NormalMode_Tx(CAN_T *tCAN)。两块板子的CAN通过CAN Tanceiver接到一起。先run接收代码，还是先run发送代码无所谓的，因为发送方没有收到ACK信号会自动重传。

4.2 USB Device

USB Device可以接到USB Host接口，例如：USB 键盘接到PC机上。PC机就能枚举USB设备，并安装相应的驱动，然后USB Host就能和USB Device通讯了。

PC机怎样知道此USB设备是什么设备呢？怎样知道如何通讯呢？这就要学习USB协议了。

注意：USB Device 不能自己发送数据到USB Host，必须等USB Host问USB Device要数据才能回。

4.2.1 USB 协议简介

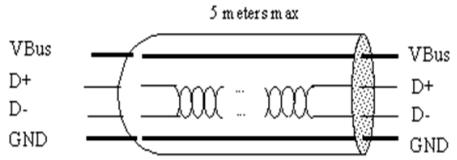
从USB org下载USB协议之后，主要看第九章就行了，其他章节快速浏览一遍就行。

插入USB Host的USB device会有下面6种状态：插入USB Host、上电、缺省、分配地址、配置、挂起/resume

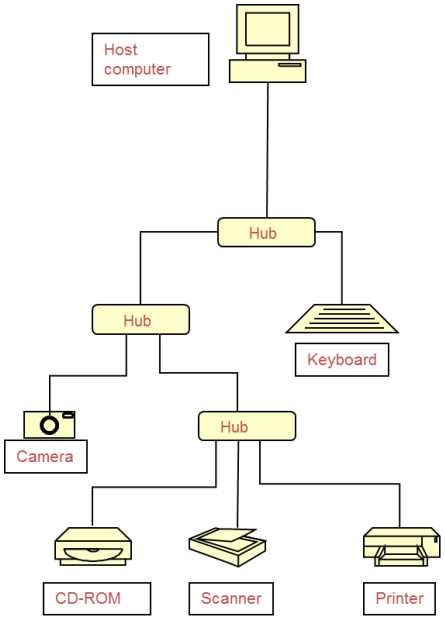
	Attached	Powered	Default	Address	Configured	Suspended	State
	No	--	--	--	--	--	Device is not attached to the USB. Other attributes are not significant.
插入主机	Yes	No	--	--	--	--	Device is attached to the USB, but is not powered. Other attributes are not significant.
	Yes	Yes	No	--	--	--	Device is attached to the USB and powered, but has not been reset.
收到BUS RESET	Yes	Yes	Yes	No	--	--	Device is attached to the USB and powered and has been reset, but has not been assigned a unique address. Device responds at the default address.
分配地址	Yes	Yes	Yes	Yes	No	--	Device is attached to the USB, powered, has been reset, and a unique device address has been assigned. Device is not configured.

Device收到SET_CONFIGURATION命令代表主机枚举成功

下图右边是USB总线拓扑结构，一个主机最多可以接127个从机。每段线缆长度限制：低速3m、高速/全速 5m。左边是USB线，一般有用的是4根脚，还有一根ID脚用于OTG时判断是Device还是Host用的。D+/D-两根脚做差分传输。



pin	Name	Description
1	Vcc	+5 Vdc
2	D-	Data-
3	D+	Data+
4	GND	Ground



4.2.1.1 USB设备地址

每个USB设备都有唯一的设备地址，在枚举的时候由主机分配。地址是用来识别USB设备的，总共7-bit，最多可以表示127个设备，其中地址0是所有USB设备的默认地址。USB设备一插入主机时，地址就是0，然后主机会尽快给它分配一个地址。

LSB MSB

Addr0	Addr1	Addr2	Addr3	Addr4	Addr5	Addr6

4.2.1.2 USB端点地址

USB设备中包含多个USB端点(Endpoint)，每个端点有端点地址和端点类型。USB IP根据端点地址操作对应的USB端点。

USB支持4种传输类型（端点类型）：控制传输、块传输、中断传输、等时传输。每种传输作用不同。

- 控制传输：用来收/发 USB 命令。它的端点地址是 0，这个是强制规定的。一个设备插入主机，主机就开始枚举过程。***枚举就是通过控制端点拿到设备的各种描述符。***
- 块传输：一般用于大量数据传输，只要主机有空就会传输该端点的数据
- 中断传输：一般用于鼠标和键盘。它的特点是主机每隔一定的时间来要数据，如果主机比较忙，这个间隔是不保证的

- 等时传输：一般用于传输语音数据。它的特点是主机每隔 1ms 传输一次数据，主机一定要尽力保证这个间隔。因为语音数据一旦跟不上，就会听到断音，而这个一般不允许。

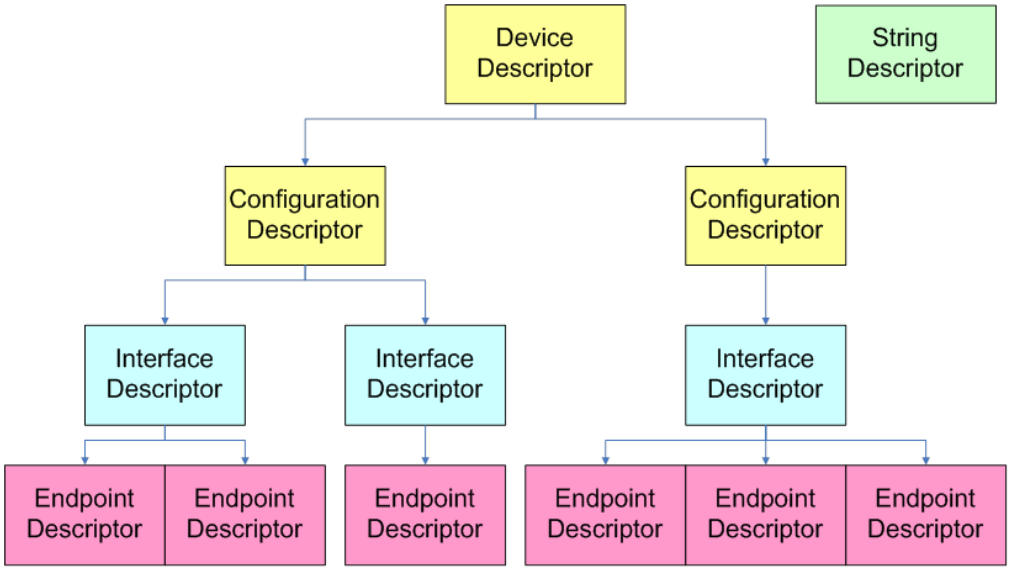
一个USB设备中，可以有多个端点，每个端点的作用不同，所以需要端点地址来标识端点。端点地址由端点号+方向组成，端点号4-bit，方向1-bit。方向就是指IN 和OUT，代表数据传输的方向。IN代表该端点是负责发送数据到Host的，OUT代表该端点是负责从Host接收数据的。大家注意在USB中方向都是对USB Host而言的，就是说IN是指传输数据到USB Host， OUT是指USB Host传输数据到USB Device。

端点是有方向的，同一个端点地址可以同时支持IN和OUT，也可以只支持IN或者OUT。

一个USB物理设备一般可以包含多个USB 设备，这种叫USB复合设备，例如：USB Audio + USB HID设备，这种设备插入USB Host之后，会同时枚举出两个USB 设备。

4.2.1.3 USB的描述符

USB Host要识别USB device靠的就是各种USB 描述符：设备描述符、配置描述符、接口描述符、端点描述符、字符串描述符。USB Spec中9.6节详细描述了各种USB描述符，它们的逻辑结构如下：



枚举时，USB Host会先要设备描述符，然后要配置描述符。要配置描述符的时候USB device要把接口描述符和端点描述符一起上传。

要实现一个USB设备关键就是准备USB描述符，下面是一个USB HID设备的描述符

```
/*USB 设备描述符，下面的VID和PID是很重要的，它决定了USB主机装哪个驱动。VID是跟USB组织申请的，PID是各个公司给每个产品分配的 */
const uint8_t gu8DeviceDescriptor[] =
{
```

```
    LEN_DEVICE,     /* bLength */
    DESC_DEVICE,    /* bDescriptorType */
    0x10, 0x01,     /* bcdUSB */
    0x00,           /* bDeviceClass */
    0x00,           /* bDeviceSubClass */
    0x00,           /* bDeviceProtocol */
    EP0_MAX_PKT_SIZE,   /* bMaxPacketSize0 */
    /* idVendor */
    USBD_VID & 0x00FF,
    (USBD_VID & 0xFF00) >> 8,
    /* idProduct */
    USBD_PID & 0x00FF,
    (USBD_PID & 0xFF00) >> 8,
    0x00, 0x00,     /* bcdDevice */
    0x01,           /* iManufacture */
    0x02,           /* iProduct */
    0x00,           /* iSerialNumber - no serial */
    0x01            /* bNumConfigurations */
};

/* USB 配置描述符，它包含了接口和端点描述符 */
const uint8_t gu8ConfigDescriptor[] =
{
    /*配置描述符*/
    LEN_CONFIG,     /* bLength */
    DESC_CONFIG,    /* bDescriptorType */
    /* wTotalLength */
    (LEN_CONFIG + LEN_INTERFACE + LEN_HID + LEN_ENDPOINT * 2) & 0x00FF,
    ((LEN_CONFIG + LEN_INTERFACE + LEN_HID + LEN_ENDPOINT * 2) & 0xFF00) >> 8,
    0x01,           /* bNumInterfaces */
    0x01,           /* bConfigurationValue */
    0x00,           /* iConfiguration */
    0x80 | (USBD_SELF_POWERED << 6) | (USBD_REMOTE_WAKEUP << 5),/* bmAttributes */
    USBD_MAX_POWER,         /* MaxPower */

    /* 接口描述符: HID */
    LEN_INTERFACE,  /* bLength */
    DESC_INTERFACE, /* bDescriptorType */
    0x00,           /* bInterfaceNumber */
    0x00,           /* bAlternateSetting */
    0x02,           /* bNumEndpoints */
```

```
    0x03,           /* bInterfaceClass */
    0x00,           /* bInterfaceSubClass */
    0x00,           /* bInterfaceProtocol */
    0x00,           /* iInterface */

    /* HID 描述符 */
    LEN_HID,        /* Size of this descriptor in UINT8s. */
    DESC_HID,       /* HID descriptor type. */
    0x10, 0x01,     /* HID Class Spec. release number. */
    0x00,           /* H/W target country. */
    0x01,           /* Number of HID class descriptors to follow. */
    DESC_HID_RPT,   /* Descriptor type. */
    /* Total length of report descriptor. */
    sizeof(HID_DeviceReportDescriptor) & 0x00FF,
    (sizeof(HID_DeviceReportDescriptor) & 0xFF00) >> 8,

    /* 端点描述符: 中断IN, 地址为INT_IN_EP_NUM */
    LEN_ENDPOINT,   /* bLength */
    DESC_ENDPOINT,  /* bDescriptorType */
    (INT_IN_EP_NUM | EP_INPUT), /* bEndpointAddress */
    EP_INT,         /* bmAttributes */
    /* wMaxPacketSize */
    EP2_MAX_PKT_SIZE & 0x00FF,
    (EP2_MAX_PKT_SIZE & 0xFF00) >> 8,
    HID_DEFAULT_INT_IN_INTERVAL,        /* bInterval */

    /* 端点描述符: 中断 out. 地址为INT_OUT_EP_NUM */
    LEN_ENDPOINT,   /* bLength */
    DESC_ENDPOINT,  /* bDescriptorType */
    (INT_OUT_EP_NUM | EP_OUTPUT),   /* bEndpointAddress */
    EP_INT,         /* bmAttributes */
    /* wMaxPacketSize */
    EP3_MAX_PKT_SIZE & 0x00FF,
    (EP3_MAX_PKT_SIZE & 0xFF00) >> 8,
    HID_DEFAULT_INT_IN_INTERVAL     /* bInterval */
};

/*字符串描述符 */
const uint8_t gu8StringLang[4] =
{
    4,              /* bLength */
```

```
    DESC_STRING,    /* bDescriptorType */
    0x09, 0x04
};

/*字符串描述符 */
const uint8_t gu8VendorStringDesc[] =
{
    16,
    DESC_STRING,
    'N', 0, 'u', 0, 'v', 0, 'o', 0, 't', 0, 'o', 0, 'n', 0
};

/*字符串描述符 */
const uint8_t gu8ProductStringDesc[] =
{
    26,             /* bLength          */
    DESC_STRING,    /* bDescriptorType  */
    'H', 0, 'I', 0, 'D', 0, ' ', 0, 'T', 0, 'r', 0, 'a', 0, 'n', 0, 's', 0, 'f', 0, 'e',
0, 'r', 0
};



/*字符串描述符 */
const uint8_t gu8StringSerial[26] =
{
    26,             // bLength
    DESC_STRING,    // bDescriptorType
    'A', 0, '0', 0, '2', 0, '0', 0, '1', 0, '4', 0, '0', 0, '9', 0, '0', 0, '3', 0, '0',
0, '4', 0
};
```

该HID设备除了控制端点0，还有两个端点：中断IN和中断OUT

INT_IN_EP_NUM和INT_OUT_EP_NUM是端点地址，它们可以一样，也可以不一样。

- 如果 INT_IN_EP_NUM=2，INT_OUT_EP_NUM=3，PC 就会看到中断 IN 端点地址为 2，中断 OUT 端点地址为 3。然后 PC 就会发送数据到端点地址 3，并从端点地址 2 接收数据。USB device 将要发送的数据放到地址 2 对应的 RAM 中，并从地址 3 对应的 RAM 中收数据。
- 如果 INT_IN_EP_NUM=2，INT_OUT_EP_NUM=2，PC 就会看到端点地址 2 既可以 IN 也可以 OUT。然后 PC 就会发送数据到端点地址 2，并从端点地址 2 接收数据。USB device 将要发送的数据放到地址 2 方向为 IN 对应的 RAM 中，并从地址 2 方向为 OUT 对应的 RAM 中收数据。

上面说了这么多，无非是告诉大家为何会有端点地址这个东东，希望大家看完之后，都能理解。

各个字符串描述符就是在USB HOST上查看USB设备时，看到的各个USB设备描述。例如：此设备的用途，公司名字等等。

下面介绍一下四种传输类型。

4.2.1.4 控制传输

控制传输整个R/W过程如下，包含3个阶段：SETUP阶段、数据阶段、状态阶段。

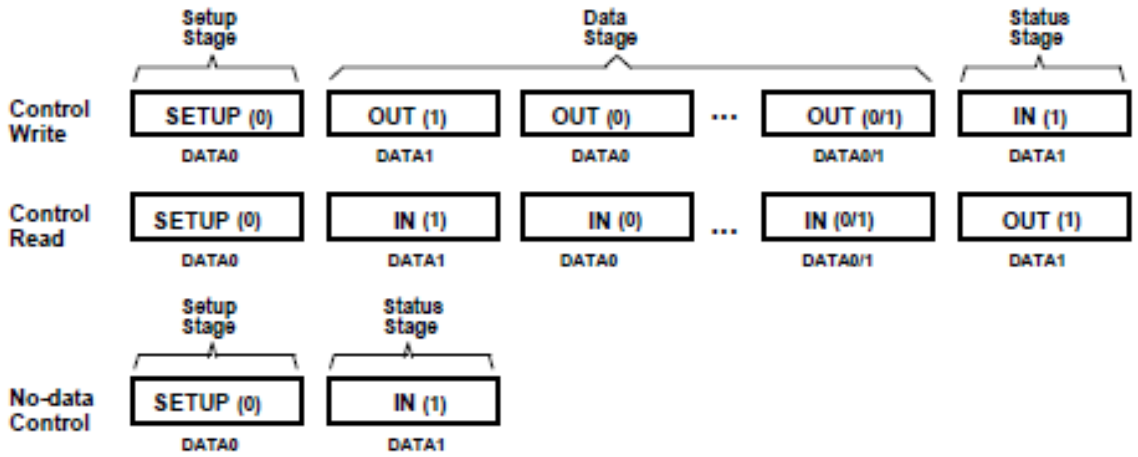


Figure 8-37. Control Read and Write Sequences

上图中SETUP、IN、OUT Token（令牌）都是硬件负责发送。

- 控制写：
  - SETUP 阶段：主机发送 SETUP 令牌，USB device 知道这是 USB 命令，会将后面的 8 个字节的命令存到单独的 SETUP buffer 中。
  - 数据阶段：然后 Host 就会发送 OUT 令牌到端点地址 0，Device 就知道这是控制端点的数据，会将数据放到控制端点的 OUT buffer 中。
  - 状态阶段：接收完毕 device 需要回一个 Zero 包给主机。Zero 包就是数据长度是 0 的数据包。
- 控制读：
  - SETUP 阶段：主机发送 SETUP 令牌，USB device 知道这是 USB 命令，会将后面的 8 个字节的命令存到单独的 SETUP buffer 中。
  - 数据阶段：然后 Host 会发送 IN 令牌到端点地址 0，Device 就知道这是 Host 要求发送数据到主机，如果此时控制端点的 IN buffer 中有数据，硬件就会将数据回给主机。否则回 NACK 给主机

- 状态阶段：接收完毕 Host 会回一个 Zero 包给 Device。Zero 包就是数据长度是 0 的数据包

由此可知，Device不能自己收/发数据，是由Hos控制的。

下图是所有的Token以及对应的值，每个Token都是8-bit，格式如图8-1。例如：OUT 令牌PID值是0xE1，IN令牌PID值是0x69。

Table 8-1. PID Types

PID Type	PID Name	PID<3:0>*	Description
Token	OUT	0001B	Address + endpoint number in host-to-function transaction
	IN	1001B	Address + endpoint number in function-to-host transaction
	SOF	0101B	Start-of-Frame marker and frame number
	SETUP	1101B	Address + endpoint number in host-to-function transaction for SETUP to a control pipe
Data	DATA0	0011B	Data packet PID even
	DATA1	1011B	Data packet PID odd
	DATA2	0111B	Data packet PID high-speed, high bandwidth isochronous transaction in a microframe (see Section 5.9.2 for more information)
	MDATA	1111B	Data packet PID high-speed for split and high bandwidth isochronous transactions (see Sections 5.9.2, 11.20, and 11.21 for more information)
Handshake	ACK	0010B	Receiver accepts error-free data packet
	NAK	1010B	Receiving device cannot accept data or transmitting device cannot send data
	STALL	1110B	Endpoint is halted or a control pipe request is not supported
	NYET	0110B	No response yet from receiver (see Sections 8.5.1 and 11.17-11.21)
Special	PRE	1100B	(Token) Host-issued preamble. Enables downstream bus traffic to low-speed devices.
	ERR	1100B	(Handshake) Split Transaction Error Handshake (reuses PRE value)
	SPLIT	1000B	(Token) High-speed Split Transaction Token (see Section 8.4.2)
	PING	0100B	(Token) High-speed flow control probe for a bulk/control endpoint (see Section 8.5.1)
	Reserved	0000B	Reserved PID

*Note: PID bits are shown in MSb order. When sent on the USB, the rightmost bit (bit 0) will be sent first.

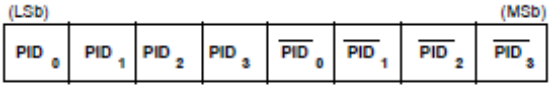


Figure 8-1. PID Format

下面详细解剖一下这3各阶段，了解这3各阶段很关键，特别是状态阶段一定不能错，如果主机不能完成状态阶段，它就会停止枚举。

1) SETUP 阶段

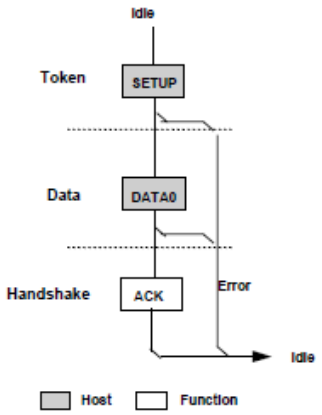


Figure 8-36. Control SETUP Transaction

主机发送 SETUP Token，DATA0 Token+数据，Device 回 Ack，SETUP 阶段结束

Sync	SETUP	ADDR	ENDP	CRC5	EOP	Pkt Len	Idle	Time Stamp
00000001	0xB4	0	0	0x08	250.000 ns	35 Bits (5 Bytes)	133.330 ns	0 . 237 796 650

Sync	DATA0	Data	CRC16	EOP	Pkt Len	Idle	Time Stamp
00000001	0xC3	80 06 00 01 00 00 40 00	0xBB29	250.000 ns	99 Bits (13 Bytes)	200.000 ns	0 . 237 799 700

Sync	ACK	EOP	Pkt Len	Idle	Time Stamp
00000001	0x4B	250.000 ns	19 Bits (3 Bytes)	7.399 us	0 . 237 808 150

上图是用 USB 逻辑分析仪从 PC 上抓到的 SETUP 阶段的包，可以很清楚的看到 SETUP+DATA0+ACK 令牌。

SETUP 数据包格式如下，固定的 8 个字节。第二个字节是 bRequest 域如右表。以上图的 SETUP 命令为例：0x06 表示取得描述符，接下来 wValue 域为 0x0100，表示取得设备描述符。软件将设备描述符填到控制传输 IN buffer 里面，主机就会发 IN 令牌拿走。同时软件别忘记准备接收状态阶段（触发控制传输 OUT buffer，准备接收 Zero 包）。而 wLength 域表示 IN/OUT 的数据长度，该长度不能超过设备描述符中说明的最大包大小。

bmRequestType D6..5 说明该命令的类型：

- 0：表示该命令是 USB 标准命令
- 1：表示该命令是 USB Class 命令，例如：USB Audio Class， USB HID Class 等等
- 2：表示是厂商自己定义的命令

Table 9-2. Format of Setup Data

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

Table 9-4. Standard Request Codes

bRequest	Value
GET_STATUS	0
CLEAR_FEATURE	1
Reserved for future use	2
SET_FEATURE	3
Reserved for future use	4
SET_ADDRESS	5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNCH_FRAME	12

Table 9-5. Descriptor Types

Descriptor Types	Value
DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5
DEVICE_QUALIFIER	6
OTHER_SPEED_CONFIGURATION	7
INTERFACE_POWER[1]	8

2) 数据阶段

以主机接收数据为例，主机发送 IN 令牌， DATA1 令牌+从机回数据，DATA0 令牌+从机回数据，DATA1 令牌+从机回数据，DATA0 令牌+从机回数据…主机回 ACK。

DATA0/DATA1 令牌是轮流的，不能出错。每个 DATAx 令牌后面跟的数据长度不能大于最大包大小，控制传输的最大包大小定义在设备描述符里面。

Sync	IN	ADDR	ENDP	CRC5	EOP	Pkt Len	Idle	Time Stamp
00000001	0x96	0	0	0x08	250.000 ns	35 Bits (5 Bytes)	165.330 ns	0 . 240 086 900

Sync	DATA1	Data	CRC16	EOP
00000001	0xD2	0: 12 01 10 01 00 00 00 40 1A 05 1B 51 00 00 01 02 16: 03 01	0x3916	266.660 ns

Sync	ACK	EOP	Pkt Len	Idle	Time Stamp
00000001	0x4B	250.000 ns	19 Bits (3 Bytes)	5.351 us	0 . 240 105 116

3) 状态阶段

以主机接收数据为例（数据阶段是 IN），则状态阶段主机发 OUT 令牌，DATA1 令牌+数据给从机，从机回 ACK。大家可以看到状态阶段的数据长度是 0，并且一定用 DATA1 令牌，不管数据阶段最后结束是 DATA0 还是 DATA1 都发 DATA1。并且状态阶段一定和数据阶段方向相反。数据阶段是 IN，状态阶段就是 OUT；数据阶段是 OUT，状态阶段就是 IN。

如果大家看一下 M0 BSP 中 USBD 的代码就会发现每次控制端点收到数据都会触发另一个端点发送一个 Zero 包，就是因为要完成状态阶段。USB2.0 IP 中清除 CEP 回 NACK（就是让控制端点回 ACK 的意思）就是用于状态阶段 。

Sync	OUT	ADDR	ENDP	CRC5	EOP	Pkt Len	Idle	Time Stamp
00000001	0x87	0	0	0x08	250.000 ns	35 Bits (5 Bytes)	133.330 ns	0 . 240 112 050

Sync	DATA1	Data	CRC16	EOP	Pkt Len	Idle	Time Stamp
00000001	0xD2	0 bytes	0x0000	250.000 ns	35 Bits (5 Bytes)	183.330 ns	0 . 240 115 100

Sync	ACK	EOP	Pkt Len	Idle	Time Stamp
00000001	0x4B	266.660 ns	20 Bits (3 Bytes)	136.049 us	0 . 240 118 200

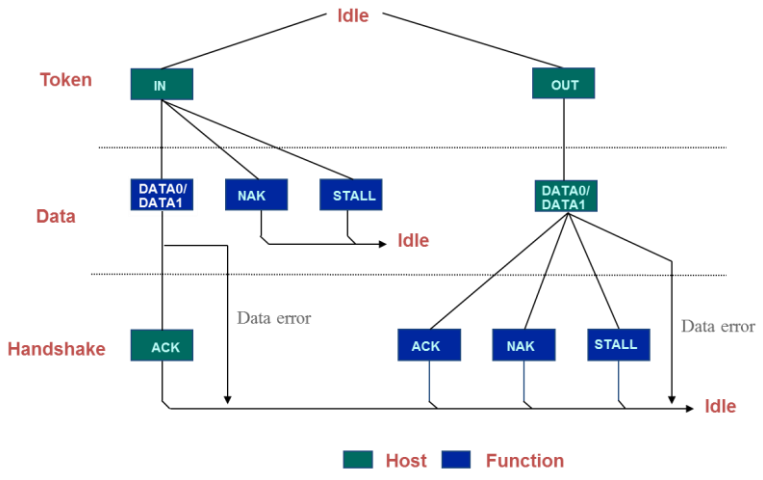
状态阶段

4.2.1.5 块传输

只要主机有空，并且有块传输等待处理，主机就会处理

- 主机发送 IN 令牌，DATA0 令牌+从机回数据，DATA1 令牌+从机回数据，DATA0 令牌+从机回数据，DATA1 令牌+从机回数据…..，主机回 ACK
- 主机发送 OUT 令牌，DATA0 令牌+数据，DATA1 令牌+数据，DATA0 令牌+数据，DATA1 令牌+数据…..，从机回 ACK

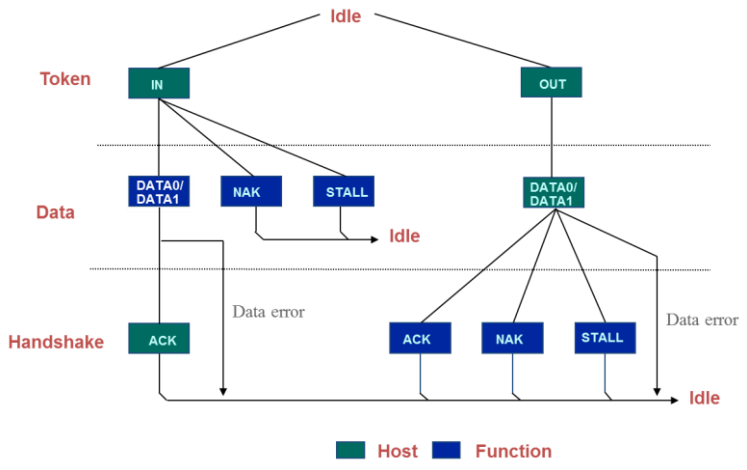
每个DATAx令牌后面跟的数据长度不能大于最大包大小，块传输的最大包大小定义在端点描述符里面。每一个DATAx令牌后面都跟一个ACK。



4.2.1.6 中断传输

- 主机发送 IN 令牌，DATA0 令牌+从机回数据，DATA1 令牌+从机回数据，DATA0 令牌+从机回数据，DATA1 令牌+从机回数据…..，主机回 ACK
- 主机发送 OUT 令牌，DATA0 令牌+数据，DATA1 令牌+数据，DATA0 令牌+数据，DATA1 令牌+数据…..，从机回 ACK

每个DATAx令牌后面跟的数据长度不能大于最大包大小，中断传输的最大包大小定义在端点描述符里面。但是对于HID设备，每次收发的字节数是固定的，定义在HID报告描述符里面。4.2.4节有HID报告描述符的例子，大家可以看一下。

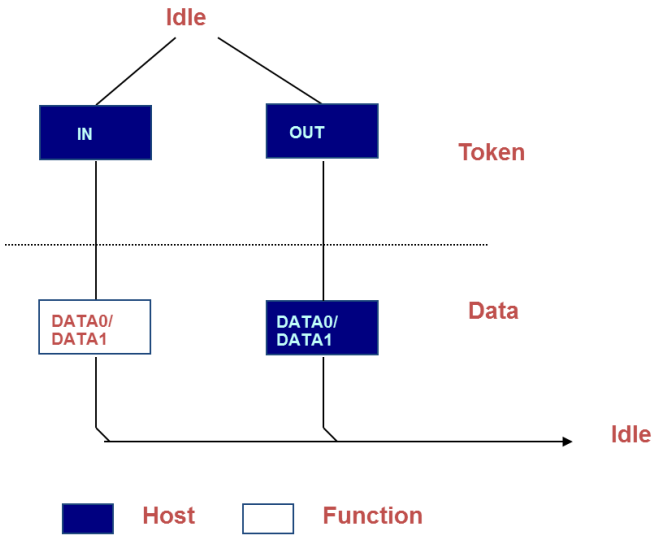


大家如果用心读了块传输，就会发现中断传输和块传输流程一模一样。所不同的只是中断传输主机是按照一定的间隔来拿数据的，最快 1ms 收/发一次。

4.2.1.7 等时传输

等时传输没有ACK，它只关心按时发送数据，而不关心数据是否发送成功

- 主机发送 IN 令牌，DATA0 令牌+从机回数据，DATA1 令牌+从机回数据，DATA0 令牌+从机回数据，DATA1 令牌+从机回数据…..
- 主机发送 OUT 令牌，DATA0 令牌+数据，DATA1 令牌+数据，DATA0 令牌+数据，DATA1 令牌+数据…..



4.2.1.8 最大数据包

上面描述符的例子中，端点描述符有个参数：最大包大小。这个参数很重要。USB主机从设备接收数据时，怎么知道数据传输完毕呢？

USB规定，只要传输的包size小于最大包size就认为传输结束

大家会说，前面SETUP包格式里面不是有数据阶段长度参数wLength吗？根据这个值主机不是就知道数据阶段已经结束了吗？这个问题只能说是USB的规定，如果USB主机收到一个包size小于最大包size，不用等到收到wLength个字节，传输就提前结束了。

例如：如果最大包大小 64B，传输了60B，就认为结束。又或者最大包大小64B，要传输的数据128B，2包传输完毕之后，如果主机继续发IN token，这时候必须回长度是0 的数据包给主机。

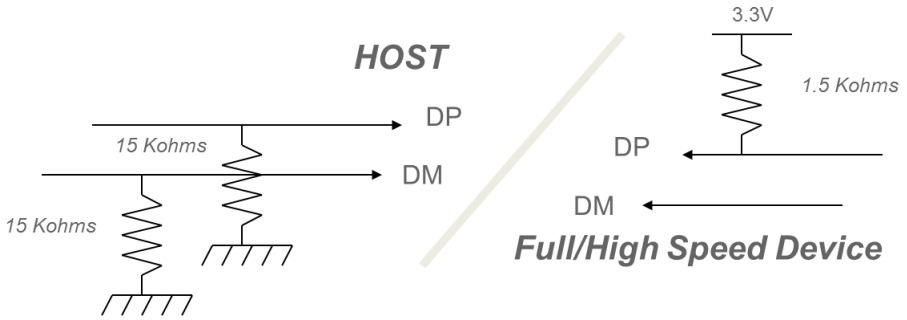
传输类型	最大数据包大小		
	低速	全速	高速
Control	**8**	**8, 16, 32, 64**	**64**
Bulk		**8, 16, 32, 64**	**512**

Interrupt	**<= 8**	**<= 64**	**<= 1024**
Isochronous		**1023**	**<= 1024**

4.2.1.9 设备检测

设备插入主机之后，主机怎么知道有设备插入了呢？怎么知道该设备是全速/高速还是低速设备呢？就靠DP/DM脚上的上拉电阻

- 主机（hub）下行端口整合了 2 个 15K 欧姆的下拉电阻。平时总线状态为 SE0（DP = DM = 0V），表示没有设备插入。
- 当设备插入时，总线状态变成 IDLE（DP = 3.3V DM = 0V），DP 由设备内部的上拉电阻拉高（1.5K），表示插入的是全速/高速设备。而低速设备上拉接在 DM 上。



4.2.1.10 总结

以上USB协议就介绍完了，总结如下：

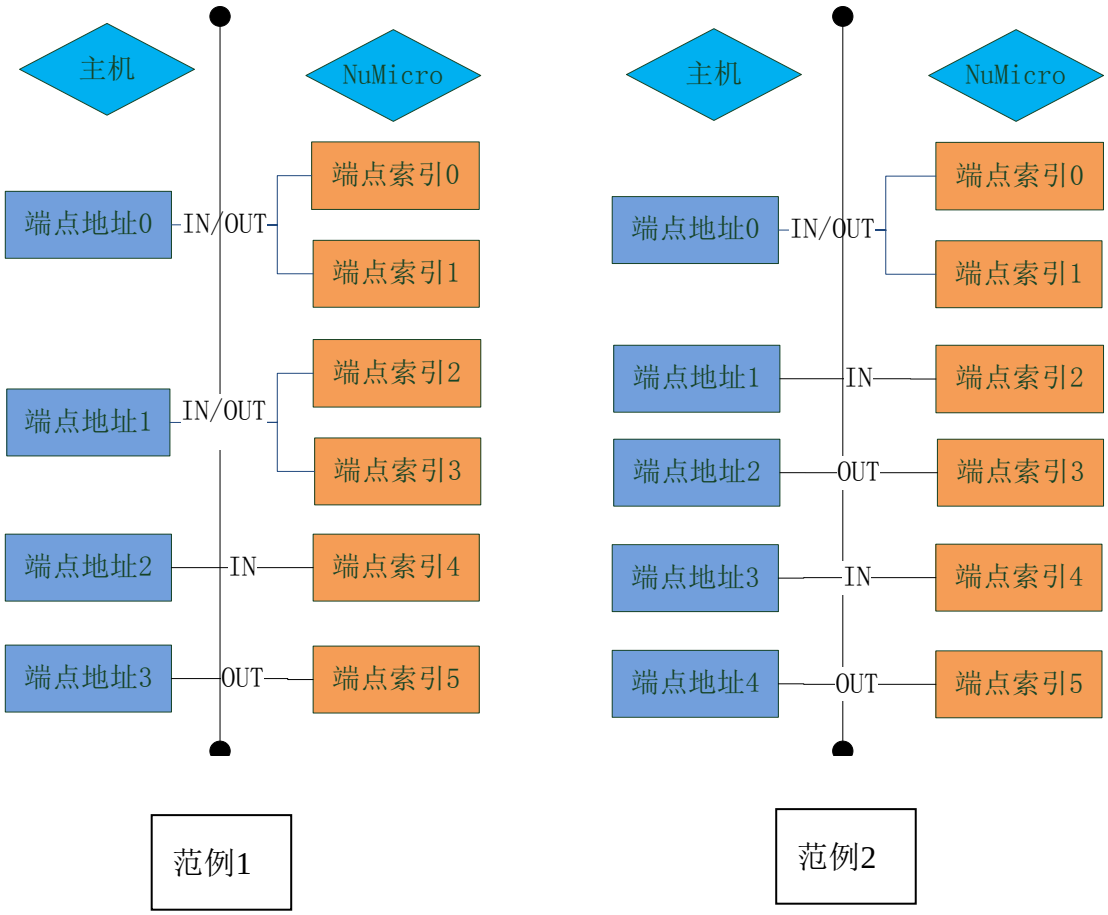
1) USB 有 4 种传输类型：控制、中断、块、等时
2) 每个 USB 设备有独立的设备地址，每个端点有独立的端点地址
3) 每个端点地址可以只支持 IN 或者 OUT，也可以同时支持 IN 和 OUT
4) 每个 USB 设备必须要有控制传输，端点地址固定为 0，可以 IN 也可以 OUT
5) 枚举过程主要通过控制传输拿到设备所有的描述符
6) 控制传输分为 3 个阶段：SETUP 阶段、数据阶段、状态阶段

4.2.2 新唐 USB1.1 IP 特点

新唐的USB1.1 IP 与USB1.1规格兼容，速度12Mbps，一般有6～8个端点。控制端点IN用一个，OUT用一个，剩下的可以给其它设备用，例如：如果还有4个端点，USB Audio录音一个，放音一个，HID设备中断IN一个，中断OUT一个。如果还有6个端点，就可以模拟USB录/放音+HID传输+HID鼠标3个设备。

客户经常把USB IP支持的端点索引号和端点地址混淆，大家看一下下面端点地址和内部端点的

索引是不是会清楚一点。



因为USB1.1的IP内部的端点要么是IN要么是OUT，只能选一个。如果某个端点地址同时支持IN和OUT，那么它就对应USB IP内部2个端点。

下面我们以NUC123的USB IP为例说明新唐全速USB device的特点。新唐目前各个全速USB Device都是兼容的，知道了NUC123的USB IP的用法，NANO120、NUC120也就通了。

4.2.2.1 NUC123 USB IP的特点

- 与 USB 2.0 Full-Speed specification 兼容.
- 提供 1 个中断向量(23)，4 种中断事件.
  - WAKEUP(唤醒主机)
  - FLO (热插拔检测中断)
  - USB (USB 事件, 像 IN ACK, OUT ACK)
  - BUS (USB Bus 事件, 像 suspend,reset)

- 支持 4 种传输类型
  - Control
  - Bulk
  - Interrupt
  - Isochronous
- 提供 8 endpoints，可以配置成 Control/Bulk/Interrupt/Isochronous 传输类型
- 包含 512 bytes 内部 SRAM 作为 USB buffer。这些 buffer 就是用作 SETUP buffer、IN buffer 和 OUT buffer 的，用来和主机之间收/发数据
- 提供远程唤醒功能，可以将处于休眠状态的主机唤醒.

所有的寄存器列表如下，上半部分是系统设定部分，下半部分是每个端点都有这样的一组寄存器

USB Base Address: USB_BA = 0x4006_0000				
USB_INTEN	USB_BA+0x000	R/W	USB Interrupt Enable Register	0x0000_0000
USB_INTSTS	USB_BA+0x004	R/W	USB Interrupt Event Status Register	0x0000_0000
USB_FADDR	USB_BA+0x008	R/W	USB Device Function Address Register	0x0000_0000
USB_EPSTS	USB_BA+0x00C	R	USB Endpoint Status Register	0x0000_0000
USB_ATTR	USB_BA+0x010	R/W	USB Bus Status and Attribution Register	0x0000_0040
USB_FLDET	USB_BA+0x014	R	USB Floating Detected Register	0x0000_0000
USB_BUFSEG	USB_BA+0x018	R/W	Setup Token Buffer Segmentation Register	0x0000_0000
USB_DRVSE0	USB_BA+0x090	R/W	USB Drive SE0 Control Register	0x0000_0001
USB_PDMA	USB_BA+0x0A4	R/W	USB PDMA Control Register	0x0000_0000
USB_BUFSEG0	USB_BA+0x500	R/W	Endpoint 0 Buffer Segmentation Register. It is also mirrored to USB_BA+0x020 for backward compatible.	0x0000_0000
USB_MXPLD0	USB_BA+0x504	R/W	Endpoint 0 Maximal Payload Register. It is also mirrored to USB_BA+0x024 for backward compatible.	0x0000_0000
USB_CFG0	USB_BA+0x508	R/W	Endpoint 0 Configuration Register. It is also mirrored to USB_BA+0x028 for backward compatible.	0x0000_0000
USB_CFGP0	USB_BA+0x50C	R/W	Endpoint 0 Set Stall and Clear In/Out Ready Control Register. It is also mirrored to USB_BA+0x02C for backward compatible.	0x0000_0000

下面为大家介绍这些寄存器的详细用法。

1) 中断使能寄存器 USB_INTEN

一般我们会使能 BUS_IE（USB 总线中断）、USB_IE（USB 事件中断）、FLDET_IE（热插拔中断）

USB Interrupt Enable Register (USB_INTEN)

Register	Offset	R/W	Description	Reset Value
USB_INTEN	USB_BA+0x000	R/W	USB Interrupt Enable Register	0x0000_0000

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
INNAK_EN	Reserved						WAKEUP_EN
7	6	5	4	3	2	1	0
Reserved				WAKEUP_IE	FLDET_IE	USB_IE	BUS_IE

2) 中断状态寄存器 USB_INTSTS

发生中断时，先读中断状态寄存器，查看bit[3:0]

- 如果 BUS_STS 被置，说明发生了 USB 总线事件，然后再看 USB_ATTR 寄存器 bit[3:0 查看具体发生什么事件
- 如果 USB_STS 被置，说明发生 USB 事件，再看 bit[31]和 bit[23:16]。如果 bit[31]被置说明发生 SETUP 事件，读 SETUP buffer 查看主机发送了什么 USB 命令；如果 bit[23:16]被置，说明对应端点发生中断，再查看 USB_EPSTS 寄存器具体发生何种事件
- 如果 FLDET_STS 被置，再看 USB_FLDET 寄存器，查看是插入还是拔出

USB Interrupt Event Status Register (USB_INTSTS)

This register is USB Interrupt Event Status register; clear by writing '1' to the corresponding bit.

Register	Offset	R/W	Description	Reset Value
USB_INTSTS	USB_BA+0x004	R/W	USB Interrupt Event Status Register	0x0000_0000

31	30	29	28	27	26	25	24
SETUP	Reserved						
23	22	21	20	19	18	17	16
EPEVT7	EPEVT6	EPEVT5	EPEVT4	EPEVT3	EPEVT2	EPEVT1	EPEVT0
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
Reserved				WAKEUP_STS	FLDET_STS	USB_STS	BUS_STS

3) 设备地址寄存器 USB_FADDR

主机分配给该设备的 USB 设备地址要写到该寄存器里面，这样主机发下来的包 USB IP 才知道是给自己的

Register	Offset	R/W	Description	Reset Value
USB_FADDR	USB_BA+0x008	R/W	USB Device Function Address Register	0x0000_0000

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
Reserved	FADDR						

4) 端点状态寄存器 USB_EPSTS

每个端点的状态包括：IN ACK、IN NACK、OUT DATA0 ACK、OUT DATA1 ACK、SETUP ACK、等时传输结束

- IN ACK：表示主机发送 IN 令牌，从机回数据给主机，主机回 ACK 给从机，之后会发生 IN ACK 中断

- IN NACK：表示主机发送 IN 令牌，从机没有准备好数据回 NACK，之后会发生 IN NACK 中断
- OUT DATA0 ACK：表示主机发送 OUT 令牌，DATA0 令牌，从机接收主机发来的数据并回 ACK 给主机，之后会发生 OUT DATA0 ACK 中断
- OUT DATA1 ACK：表示主机发送 OUT 令牌，DATA1 令牌，从机接收主机发来的数据并回 ACK 给主机，之后会发生 OUT DATA0 ACK 中断
- SETUP ACK：表示主机发送 SETUP 令牌，将命令发送给从机，从机回 ACK 给主机，之后会发生 SETUP ACK 中断
- 等时传输结束：表示等时传输一包传完

Register	Offset	R/W	Description	Reset Value
USB_EPSTS	USB_BA+0x00C	R	USB Endpoint Status Register	0x0000_0000

31	30	29	28	27	26	25	24
EPSTS7[2:1]			EPSTS6[2:1]			EPSTS5[2:1]	
23	22	21	20	19	18	17	16
EPSTS5[0]	EPSTS4[2:0]			EPSTS3[2:0]			EPSTS2[2]
15	14	13	12	11	10	9	8
EPSTS2[1:0]		EPSTS1[2:0]			EPSTS0[2:0]		
7	6	5	4	3	2	1	0
OVERRUN	Reserved						

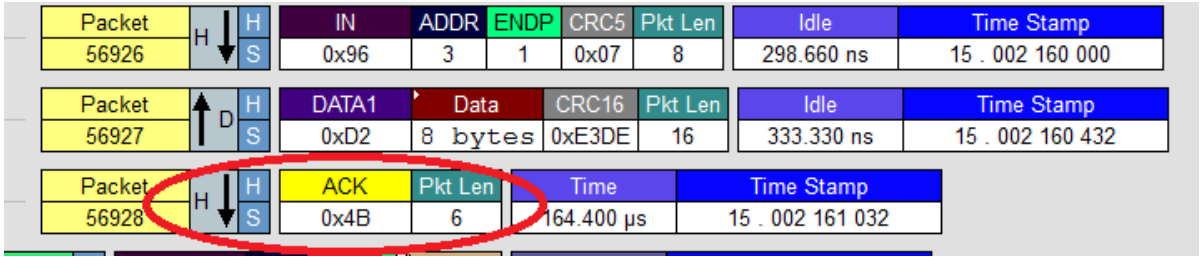
[31:29]	EPSTS7	**Endpoint 7 Bus Status** These bits are used to indicate the current status of this endpoint. 000 = In ACK 001 = In NAK 010 = Out Packet Data0 ACK 110 = Out Packet Data1 ACK 011 = Setup ACK 111 = Isochronous transfer end

5) 属性寄存器 USB_ATTR
   - Bit[10]：字节模式/字模式访问 USB SRAM。字节模式访问 USB SRAM，就只能一个一个字节读/写 USB SRAM；字模式就只能一个一个字(4 字节)读/写 USB SRAM
   - Bit[9]：PHY 电路上电
   - Bit[8]：使能 Pull-up 电阻
   - Bit[7]：使能 USB 功能
   - Bit[5]：使能 USB 进入 K 状态，用于远程唤醒 USB 主机

➢ Bit[7]：使能 PHY 功能

剩下的bit[3:0]标志USB总线的状态，发生USB BUS中断的时候，查看这几位可以得知发生何种USB 总线事件

Bit[3]是 Timeout 事件，该事件只是在 IN Token 才有用。当数据传给 Host 之后，红色圈起来的 ACK 经过 18 个 12M clock 都没有收到，就会产生 Timeout 事件



Register	Offset	R/W	Description	Reset Value
USB_ATTR	USB_BA+0x010	R/W	USB Bus Status and Attribution Register	0x0000_0040

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved					BYTEM	PWRDN	DPPU_EN
7	6	5	4	3	2	1	0
USB_EN	Reserved	RWAKEUP	PHY_EN	TIME-OUT	RESUME	SUSPEND	USBRST

6) SETUP SRAM 偏移寄存器 USB_BUFSEG

该寄存器指明 SETUP buffer 在 USB SRAM 中的偏移

Register	Offset	R/W	Description	Reset Value
USB_BUFSEG	USB_BA+0x018	R/W	Setup Token Buffer Segmentation Register	0x0000_0000

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							BUFSEG[8]
7	6	5	4	3	2	1	0
BUFSEG[7:3]					Reserved		

7) 端点 SRAM 偏移地址 USB_BUFSEGx

这些寄存器指明 IN/OUT buffer 在 USB SRAM 中的偏移

Register	Offset	R/W	Description	Reset Value
USB_BUFSEG0	USB_BA+0x500	R/W	Endpoint 0 Buffer Segmentation Register. It is also mirrored to USB_BA+0x020 for backward compatible.	0x0000_0000
USB_BUFSEG1	USB_BA+0x510	R/W	Endpoint 1 Buffer Segmentation Register. It is also mirrored to USB_BA+0x030 for backward compatible.	0x0000_0000
USB_BUFSEG2	USB_BA+0x520	R/W	Endpoint 2 Buffer Segmentation Register. It is also mirrored to USB_BA+0x040 for backward compatible.	0x0000_0000
USB_BUFSEG3	USB_BA+0x530	R/W	Endpoint 3 Buffer Segmentation Register. It is also mirrored to USB_BA+0x050 for backward compatible.	0x0000_0000
USB_BUFSEG4	USB_BA+0x540	R/W	Endpoint 4 Buffer Segmentation Register. It is also mirrored to USB_BA+0x060 for backward compatible.	0x0000_0000
USB_BUFSEG5	USB_BA+0x550	R/W	Endpoint 5 Buffer Segmentation Register. It is also mirrored to USB_BA+0x070 for backward compatible.	0x0000_0000
USB_BUFSEG6	USB_BA+0x560	R/W	Endpoint 6 Buffer Segmentation Register	0x0000_0000
USB_BUFSEG7	USB_BA+0x570	R/W	Endpoint 7 Buffer Segmentation Register	0x0000_0000

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							BUFSEG[8]x
7	6	5	4	3	2	1	0
BUFSEG[7:3]x					Reserved		

Buffer 分配示例图如下，假设控制端点最大包大小为 64B，其它端点最大包大小也是 64B：

Setup SRAM
EP0 SRAM
EP1 SRAM
EP2 SRAM
……

从偏移 0 开始依次分配给 SETUP buffer、控制端点 IN buffer、控制端点 OUT buffer，然后是其它端点。同一个端点地址 IN 和 OUT 可以用同一块 SRAM。BUFSEGx 寄存器里面只要填偏移地址就好。

USB SRAM 基地址为 0x40060100，这是 MCU 访问 USB SRAM 的地址。例如：控制端点 IN 数据，软件写 0x40060108，数据就会发出去；控制端点 OUT 数据，软件读 0x40060148，就会得到主机发来的数据。

8) 最大包大小寄存器 USB_MXPLDx

这个寄存器是收/发触发寄存器，写该寄存器 USB IP 就知道软件已经准备好收/发数据了

- **IN token**
  - 该寄存器填 USB SRAM 中将发送到主机的数据长度
- **OUT token**
  - 写该寄存器填希望收到的最大包长度
  - 读该寄存器得到实际从主机收到的包长度
  - 如果收到的包长度大于希望收到的值，OVERRUN flag 将被置，但是数据还是会被收进来

Register	Offset	R/W	Description	Reset Value
USB_MXPLD0	USB_BA+0x504	R/W	Endpoint 0 Maximal Payload Register. It is also mirrored to USB_BA+0x024 for backward compatible.	0x0000_0000
USB_MXPLD1	USB_BA+0x514	R/W	Endpoint 1 Maximal Payload Register. It is also mirrored to USB_BA+0x034 for backward compatible.	0x0000_0000
USB_MXPLD2	USB_BA+0x524	R/W	Endpoint 2 Maximal Payload Register. It is also mirrored to USB_BA+0x044 for backward compatible.	0x0000_0000
USB_MXPLD3	USB_BA+0x534	R/W	Endpoint 3 Maximal Payload Register. It is also mirrored to USB_BA+0x054 for backward compatible.	0x0000_0000
USB_MXPLD4	USB_BA+0x544	R/W	Endpoint 4 Maximal Payload Register. It is also mirrored to USB_BA+0x064 for backward compatible.	0x0000_0000
USB_MXPLD5	USB_BA+0x554	R/W	Endpoint 5 Maximal Payload Register. It is also mirrored to USB_BA+0x074 for backward compatible.	0x0000_0000
USB_MXPLD6	USB_BA+0x564	R/W	Endpoint 6 Maximal Payload Register	0x0000_0000
USB_MXPLD7	USB_BA+0x574	R/W	Endpoint 7 Maximal Payload Register	0x0000_0000

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							MXPLD[8]
7	6	5	4	3	2	1	0
MXPLD[7:0]							

9) 端点配置寄存器 USB_CFGx

USB_CFG7	USB_BA+0x578	R/W	Endpoint 7 Configuration Register	0x0000_0000

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved						CSTALL	Reserved
7	6	5	4	3	2	1	0
DSQ_SYNC	STATE		ISOCH	EP_NUM			

Bits	Description	
[31:10]	Reserved	Reserved
[9]	CSTALL	**Clear STALL Response** 1 = Clear the device to respond STALL handshake in the setup stage. 0 = Device Disabled to clear the STALL handshake in the setup stage.
[8]	Reserved	Reserved
[7]	DSQ_SYNC	**Data Sequence Synchronization** 1 = DATA1 PID 0 = DATA0 PID It is used to specify the DATA0 or DATA1 PID in the following IN token transaction. H/W will toggle automatically in IN token base on the bit.
[6:5]	STATE	**Endpoint STATE** 00 = Endpoint Disabled 01 = Out endpoint 10 = IN endpoint 11 = Undefined
[4]	ISOCH	**Isochronous Endpoint** This bit is used to set the endpoint as Isochronous endpoint, no handshake. 1 = Isochronous endpoint 0 = No Isochronous endpoint
[3:0]	EP_NUM	**Endpoint Number** These bits are used to define the endpoint number of the current endpoint.

- CSTALL 和 CFGPx 寄存器的 SSTALL 一起控制回 STALL 给主机，只用于控制端点。软件设置 SSTALL 之后，Device 会一直回 STALL 给主机一直等软件将 STALL 清为 0。如果希望回一次 STALL 就自动将 STALL bit 清除，就需要将 CSTALL 置为 1。

- IN token 的时候，数据 PID 为 DATA0 还是 DATA1 就由 DSQ_SYNC 决定，所以修改这个寄存器的时候，一定当心无意中修改到 DSQ_SYNC 的值
- 该端点是 IN 还是 OUT 由 STATE 决定 bit[6:5]
- Bit[3:0]决定该端点的地址，该地址要和端点描述符里面的一致。例如：端点描述符里面配置端点地址为 1，则某个端点配置寄存器的端点地址也要填 1，这样 IP 收到端点地址为 1 的包才知道填入哪个 buffer 里面

大家可能疑惑没有配置端点是控制、块、中断的地方，这些传输类型不用配置，它们对 Device HW来说没有什么区别

10) 控制寄存器 USB_CFGPx

USB_CFGP7	USB_BA+0x57C	R/W	Endpoint 7 Set Stall and Clear In/Out Ready Control Register	0x0000_0000

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
Reserved						SSTALL	CLRRDY

Bits	Description	
[31:2]	Reserved	Reserved
[1]	SSTALL	Set STALL 1 = Set the device to respond STALL automatically. 0 = Device Disabled to respond STALL.
[0]	CLRRDY	Clear Ready When the MXPLD register is set by user, it means that the endpoint is ready to

SSTALL：回 STALL 给主机

CLRRDY：写 MXPLD 寄存器之后，内部有个 Ready 标志会置位，该 bit 用于清除 Ready 标志

11) SE0 控制寄存器 USB_DRVSE0

SE0 状态就是 USB Host 接口 idle 状态，写这个寄存器可以由软件模拟插拔的动作。DRVSE0 = 1 主机会认为设备拔掉，等待一会主机的反应时间，再将 DRVSE0 = 0，主机就会重新枚举设备。

这个功能可以用于 2 个应用程序跳转时，如果两个都用到 USB 功能，可以使能 SE0 迫使主机重新枚举；或者同一个应用程序想分别用作两个 USB 设备，也可以通过 SE0 实现

Register	Offset	R/W	Description	Reset Value
USB_DRVSE0	USB_BA+0x090	R/W	USB Drive SE0 Control Register	0x0000_0001

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
Reserved							DRVSE0

[0]	DRVSE0	**Drive Single Ended Zero in USB Bus** The Single Ended Zero (SE0) is when both lines (USB_DP and USB_DM) are being pulled low. 1 = Force USB PHY transceiver to drive SE0 0 = None

4.2.3 代码分析

了解USB协议和新唐全速USB IP用法之后，下面的例子就比较容易理解了。

我们以HID transfer为例说明USB API的用法。该设备通过中断端点和主机收/发数据。

描述符范例在4.2.1.3节USB的描述符中。

4.2.3.1 中断处理函数

给大家讲解一下中断处理函数，了解整个USB的处理流程

```
void USBD_IRQHandler(void)
{
    uint32_t u32IntSts = USBD_GET_INT_FLAG();
    uint32_t u32State = USBD_GET_BUS_STATE();
```

```
    if(u32IntSts & USBD_INTSTS_FLDET)/*热插拔中断*/
    {
        // 清除中断标志
        USBD_CLR_INT_FLAG(USBD_INTSTS_FLDET);

        if(USBD_IS_ATTACHED())
        {
            /* USB 插入，使能USB和PHY */
            USBD_ENABLE_USB();
        }
        else
        {
            /* USB 拔除，关闭USB功能 */
            USBD_DISABLE_USB();
        }
    }

    if(u32IntSts & USBD_INTSTS_BUS)/*USB总线中断*/
    {
        /* 清除中断标志 */
        USBD_CLR_INT_FLAG(USBD_INTSTS_BUS);

        if(u32State & USBD_STATE_USBRST)
        {
            /* 总线复位，一些状态需要复位 */
            USBD_ENABLE_USB();
            USBD_SwReset();
        }
        if(u32State & USBD_STATE_SUSPEND)
        {
            /* 使能 USB ，关闭 PHY */
            USBD_DISABLE_PHY();
        }
        if(u32State & USBD_STATE_RESUME)
        {
            /* 使能 USB 和 PHY */
            USBD_ENABLE_USB();
        }
    }

    if(u32IntSts & USBD_INTSTS_USB)/*USB事件中断*/
```

```
    {
        // SETUP事件
        if(u32IntSts & USBD_INTSTS_SETUP)
        {
            /* 清除SETUP中断标志 */
            USBD_CLR_INT_FLAG(USBD_INTSTS_SETUP);

            /* 清除控制端点的Ready标志 */
            USBD_STOP_TRANSACTION(EP0);
            USBD_STOP_TRANSACTION(EP1);
            /*分析SETUP包*/
            USBD_ProcessSetupPacket();
        }

        // 端点事件
        if(u32IntSts & USBD_INTSTS_EP0)/*EP0被配置为控制端点IN*/
        {
            /* 清除中断标志 */
            USBD_CLR_INT_FLAG(USBD_INTSTS_EP0);
            // 处理控制端点发送数据给主机完成中断
            USBD_CtrlIn();
        }

        if(u32IntSts & USBD_INTSTS_EP1) /*EP1被配置为控制端点OUT*/
        {
            /*清除中断标志*/
            USBD_CLR_INT_FLAG(USBD_INTSTS_EP1);

            //处理主机通过控制端点发送给Device的数据
            USBD_CtrlOut();
        }

        if(u32IntSts & USBD_INTSTS_EP2) /*EP2被配置为中断IN*/
        {
            /*清除中断标志*/
            USBD_CLR_INT_FLAG(USBD_INTSTS_EP2);
            // 中断IN处理函数
            EP2_Handler();
        }

        if(u32IntSts & USBD_INTSTS_EP3) /*EP3被配置为中断OUT*/
```

```
        {
            /*清除中断标志*/
            USBD_CLR_INT_FLAG(USBD_INTSTS_EP3);
            // 中断OUT处理函数
            EP3_Handler();
        }
        /*其它端点都没有使用，简单清除标志就好*/
        if(u32IntSts & USBD_INTSTS_EP4)
        {
            /*清除中断标志*/
            USBD_CLR_INT_FLAG(USBD_INTSTS_EP4);
        }

        if(u32IntSts & USBD_INTSTS_EP5)
        {
            /*清除中断标志*/
            USBD_CLR_INT_FLAG(USBD_INTSTS_EP5);
        }

        if(u32IntSts & USBD_INTSTS_EP6)
        {
            /*清除中断标志*/
            USBD_CLR_INT_FLAG(USBD_INTSTS_EP6);
        }

        if(u32IntSts & USBD_INTSTS_EP7)
        {
            /*清除中断标志*/
            USBD_CLR_INT_FLAG(USBD_INTSTS_EP7);
        }
    }
    /*清除中断标志*/
    USBD_CLR_INT_FLAG(u32IntSts);
}
```

EP2被配置为中断IN，EP3被配置为中断OUT，所以EP2_Handler用于处理要发送到主机的数据， EP3_Handler用于接收从主机发来的数据。要发送到主机的数据可以提前通过下面的代码写到EP2 IN Buffer里面，如果要发送的数据大于最大包大小，剩下的就在EP2_Handler里面发送，每次最多填MAX_PACKET_SIZE到IN Buffer里面。

```
/*填中断IN Buffer*/
ptr = (uint8_t *)(USBD_BUF_BASE + USBD_GET_EP_BUF_ADDR(EP2));/*取得EP2 IN Bufer指针*/
```

```
USBD_MemCopy(ptr, (void *)&g_u8PageBuff[0], EP2_MAX_PKT_SIZE);/*将数据复制到中断IN */
USBD_SET_PAYLOAD_LEN(EP2, EP2_MAX_PKT_SIZE);/*触发MXPLD寄存器*/
```

EP3被配置为中断OUT，每收到一包Host发来的数据就会进入一次函数EP3_Handler

```
/* 中断 IN 处理函数，每调用一次表示发送一包到主机成功 */
void EP2_Handler(void)
{
    HID_SetInReport();
}
static uint8_t  g_u8PageBuff[256],g_u32BytesInPageBuf;
/* 中断 OUT 处理函数，每调用一次表示从主机收到一包数据*/
void EP3_Handler(void)
{
    uint8_t *ptr;
    /*取得EP2 OUT Bufer指针*/
    ptr = (uint8_t *)(USBD_BUF_BASE + USBD_GET_EP_BUF_ADDR(EP3));
    /*将数据复制到缓存里面*/
    USBD_MemCopy(&g_u8PageBuff[g_u32BytesInPageBuf], ptr, USBD_GET_PAYLOAD_LEN(EP3));
    g_u32BytesInPageBuf += USBD_GET_PAYLOAD_LEN(EP3);
    /*触发OUT 端点，准备接收下一包*/
    USBD_SET_PAYLOAD_LEN(EP3, EP3_MAX_PKT_SIZE);
}
```

了解上面的代码之后，通过中断IN/OUT收/发数据的流程就清楚了

4.2.3.2 HID报告描述符分析

下面是HID报告描述符，这个是Mouse的描述符，返回给Host共4个字节：

- 第一个字节说明 3 个按键哪个按下
- 第二个字节说明滑轮的值
- 第三个和第四个报告 X/Y 的坐标

  Report Size 表示每几个 bit 为一组，Report Count 表示有几组。下面红框中的 Report Size * Report Count 相加就是 4Bytes

```
const __align(4) uint8_t g_HID_au8MouseReportDescriptor[] = {
      0x05, 0x01,  /* Usage Page(Generic Desktop Controls) */
      0x09, 0x02,  /* Usage(Mouse) */
      0xA1, 0x01,  /* Collection(Application) */
      0x09, 0x01,  /* Usage(Pointer) */
      0xA1, 0x00,  /* Collection(Physical) */
```

```
    0x05, 0x09,  /* Usage Page(Button) */
    0x19, 0x01,  /* Usage Minimum(0x1) */
    0x29, 0x03,  /* Usage Maximum(0x3) */
    0x15, 0x00,  /* Logical Minimum(0x0) */
    0x25, 0x01,  /* Logical Maximum(0x1) */
    0x75, 0x01,  /* Report Size(0x1) */
    0x95, 0x03,  /* Report Count(0x3) */
    0x81, 0x02,  /* Input(3 button bit) */
    0x75, 0x05,  /* Report Size(0x5) */
    0x95, 0x01,  /* Report Count(0x1) */
    0x81, 0x01,  /* Input(5 bit padding) */
    0x05, 0x01,  /* Usage Page(Generic Desktop Controls) */
    0x09, 0x38,  /* Usage(Wheel) */
    0x15, 0x81,  /* Logical Minimum(0x81)(-127) */
    0x25, 0x7F,  /* Logical Maximum(0x7F)(127) */
    0x75, 0x08,  /* Report Size(0x8) */
    0x95, 0x01,  /* Report Count(0x1) */
    0x81, 0x06,  /* Input(1 byte wheel) */
    0x75, 0x0C,  /* Report Size(0xC) */
    0x95, 0x02,  /* Report Count(0x2) */
    0x09, 0x30,  /* Usage(X) */
    0x09, 0x31,  /* Usage(Y) */
    0x16, 0x01, 0xF8, /* Logical Minimum(0xF801) */
    0x26, 0xFF, 0x07, /* Logical Maximum(0x7FF) */
    0x81, 0x06,       /* Input */
    0xC0,         /* End Collection */
    0xC0,         /* End Collection */
};
```

下面也是HID 报告描述符，这个是HID transfer设备的报告描述符，返回给Host的字节数就是最大包大小，主机给Device的数据每包也是最大包大小。如果主机希望发送SetFeature命令，这里可以说明Feature特性，不然可以拿掉。

```
0x05, 0x01, // USAGE_PAGE (Generic Desktop)
0x09, 0x00, // USAGE (0)
0xA1, 0x01, // COLLECTION (Application)
0x15, 0x00, // LOGICAL_MINIMUM (0)
0x25, 0xFF, // LOGICAL_MAXIMUM (255)
0x19, 0x01, //  USAGE_MINIMUM (1)
0x29, 0x08, //  USAGE_MAXIMUM (8)
```

```
0x95, MAX_PACKET_SIZE_HID_INT,
0x75, 0x08, //  REPORT_SIZE (8)
0x81, 0x02, //  INPUT (Data,Var,Abs)
0x19, 0x01, //  USAGE_MINIMUM (1)
0x29, 0x08, //  USAGE_MAXIMUM (8)
0x91, 0x02, //  OUTPUT (Data,Var,Abs)
0x19, 0x01, //  USAGE_MINIMUM (1)
0x29, 0x08, //  USAGE_MAXIMUM (8)
0xB1, 0x03, //  Feature (Const,Var,Abs)
0xC0        // END_COLLECTION
```

2

上面的单个红框分别对应函数HidD_GetInputReport、HidD_SetOutputReport、HidD_SetFeature。它们是Windows C++中的函数，用来发送SET_REPORT/GET_REPORT命令，通过控制传输收/发数据的。如果报告描述符中没有红框②的内容，函数HidD_SetOutputReport就会出错。如果通过中断IN/OUT端点收/发数据，使用ReadFile/WriteFile就行了。

4.2.3.3 HID设备数据传输

HID设备有两种传输数据的方式

- **控制端点**
  - Get_Report
  - Set_Report
- **中断端点**
  - IN
  - OUT

可以通过控制端点收/发数据，也可以通过中断端点收/发数据。通过控制端点收/发数据也可以用其他命令，例如：Vendor命令。

1) Get_Report 命令 **0xA1 0x01 0x00 0x01 0x00 0x00 0x40 0x00**
   - 1010-0001 class 命令，最高位为‘1’，表示 input
   - 0x01 Get_Report 命令
2) Set_Report 命令 **0x21 0x09 0x00 0x02 0x00 0x00 0x40 0x00**
   - 00100001 class 命令，最高位为‘0’，表示 output
   - 0x09 Set_Report 命令

4.2.3.4 HID Transfer设备枚举

枚举过程的处理函数主要是**USBD_ProcessSetupPacket();USBD_CtrlIn();和USBD_CtrlOut();**

枚举过程主要是主机发送命令，从机通过函数USBD_ProcessSetupPacket分析命令，然后通过

USBD_CtrlIn和USBD_CtlOut收/发数据。

数据的收/发过程和中断IN/OUT(EP2_Hander/EP3_Hander) 是一样的，唯一不同的是控制端点有个状态阶段

```
void USBD_ProcessSetupPacket(void)
{
    /* 从SETUP Buffer中取得SETUP包 */
    USBD_MemCopy(g_usbd_SetupPacket, (uint8_t *)USBD_BUF_BASE, 8);
    /* 检查命令类型 */
    switch(g_usbd_SetupPacket[0] & 0x60)
    {
        case REQ_STANDARD:   // 标准USB命令
        {
            USBD_StandardRequest();
            break;
        }
        case REQ_CLASS:   // USB Class命令
        {
            if(g_usbd_pfnClassRequest != NULL)
            {
                g_usbd_pfnClassRequest();
            }
            break;
        }
        case REQ_VENDOR:   // Vendor命令
        {
            if(g_usbd_pfnVendorRequest != NULL)
            {
                g_usbd_pfnVendorRequest();
            }
            break;
        }
        default:   // reserved
        {
            /* Setup error, stall the device */
            USBD_SET_EP_STALL(EP0);
            USBD_SET_EP_STALL(EP1);
            break;
        }
    }
}
```

标准USB命令的处理函数有点复杂，其实也就是根据不同的USB命令进行不同的处理。下面的代码并不完整，感兴趣的可以到BSP里面usbd.c中有完整版。给大家分析几个典型的命令，其它的命令处理方式都是一样的。

```
void USBD_StandardRequest(void)
{
    /* 清除全局变量 */
    g_usbd_CtrlInPointer = 0;
    g_usbd_CtrlInSize = 0;

    if(g_usbd_SetupPacket[0] & 0x80)    /* 该命令数据阶段的方向是Device - > Host */
    {
        // Device to host
        switch(g_usbd_SetupPacket[1])
        {
            case GET_CONFIGURATION:
            {
                // 返回当前配置设定，将返回数据写入控制端点IN Buffer中
                M8(USBD_BUF_BASE + USBD_GET_EP_BUF_ADDR(EP0)) = g_usbd_UsbConfig;
                /* 状态阶段 */
                USBD_SET_DATA1(EP1); /*使用DATA1令牌*/
                USBD_SET_PAYLOAD_LEN(EP1, 0);/*数据长度为0*/
                /* 数据阶段 */
                USBD_SET_DATA1(EP0); /*使用DATA1令牌*/
                USBD_SET_PAYLOAD_LEN(EP0, 1); /*数据长度为1*/
                break;
            }
            case GET_DESCRIPTOR:/*取得各类描述符*/
            {
                USBD_GetDescriptor();
                break;
            }
            ……
        }
    }
    else/* 该命令数据阶段的方向是Host - > Device */
    {
        // Host to device
        switch(g_usbd_SetupPacket[1])
```

```
        {
            case SET_ADDRESS:/*该命令没有数据阶段*/
            {
                g_usbd_UsbAddr = g_usbd_SetupPacket[2];
                DBG_PRINTF("Set addr to %d\n", g_usbd_UsbAddr);

                /* 状态阶段 */
                USBD_SET_DATA1(EP0); /*使用DATA1令牌*/
                USBD_SET_PAYLOAD_LEN(EP0, 0); /*数据长度为0*/

                break;
            }
            case SET_CONFIGURATION: /*该命令没有数据阶段*/
            {
                g_usbd_UsbConfig = g_usbd_SetupPacket[2];

                if(g_usbd_pfnSetConfigCallback)
                    g_usbd_pfnSetConfigCallback();

                /* 状态阶段 */
                USBD_SET_DATA1(EP0); /*使用DATA1令牌*/
                USBD_SET_PAYLOAD_LEN(EP0, 0); /*数据长度为0*/
                break;
            }
            ……
        }
    }
}
```

收到SET_CONFIGURATION命令表示主机枚举成功。

下面着重介绍USBD_GetDescriptor()和USBD_CtrlIn和USBD_CtlOut函数，枚举过程拿各种描述符是关键。而描述符就要通过USBD_CtrlIn传给Host。

描述符分为设备描述符，配置描述符和字符串描述符；另外还有HID Class特有的HID 报告描述符

```
void USBD_GetDescriptor(void)
{
    uint32_t u32Len;
    /*取得控制读数据阶段数据的长度*/
    u32Len = 0;
```

```
    u32Len = g_usbd_SetupPacket[7];
    u32Len <<= 8;
    u32Len += g_usbd_SetupPacket[6];

    switch(g_usbd_SetupPacket[3])
    {
        // 取得设备描述符
        case DESC_DEVICE:
        {
            /*设备描述符字节数和希望返回的字节数之间取一个最小值*/
            u32Len = Minimum(u32Len, LEN_DEVICE);
            DBG_PRINTF("Get device desc, %d\n", u32Len);
            /*返回设备描述符，如果数据长度大于MAX_PACKET_SIZE，该函数会自动分包传输*/
            USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo->gu8DevDesc, u32Len);
            USBD_PrepareCtrlOut(0, 0);/*准备状态阶段*/
            break;
        }
        // 取得配置描述符
        case DESC_CONFIG:
        {
            uint32_t u32TotalLen;

            u32TotalLen = g_usbd_sInfo->gu8ConfigDesc[3];
            u32TotalLen = g_usbd_sInfo->gu8ConfigDesc[2] + (u32TotalLen << 8);

            /*配置描述符字节数和希望返回的字节数之间取一个最小值*/
            u32Len = Minimum(u32Len, u32TotalLen);
            DBG_PRINTF("Minimum len %d\n", u32Len);
            /*返回配置描述符，如果数据长度大于MAX_PACKET_SIZE，该函数会自动分包传输*/
            USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo->gu8ConfigDesc, u32Len);
            USBD_PrepareCtrlOut(0, 0);/*准备状态阶段*/
            break;
        }
        // 取得HID描述符
        case DESC_HID:
        {
            u32Len = Minimum(u32Len, LEN_HID);
            DBG_PRINTF("Get HID desc, %d\n", u32Len);

            USBD_PrepareCtrlIn((uint8_t *)&g_usbd_sInfo->gu8ConfigDesc[LEN_CONFIG +
LEN_INTERFACE], u32Len);
```

```
            USBD_PrepareCtrlOut(0, 0);/*准备状态阶段*/
            break;
        }
        // 取得HID报告描述符
        case DESC_HID_RPT:
        {
            uint32_t u32TotalLen;
            uint32_t u32RptDescLen;

            /* 取得报告描述符的长度.*/
            u32RptDescLen = g_usbd_sInfo->gu8ConfigDesc[u32TotalLen - LEN_ENDPOINT -
1];
            u32RptDescLen = g_usbd_sInfo->gu8ConfigDesc[u32TotalLen - LEN_ENDPOINT - 2]
+ (u32TotalLen << 8);
            /*报告描述符字节数和希望返回的字节数之间取一个最小值*/
            u32Len = Minimum(u32Len, u32RptDescLen);

            USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo->gu8HidReportDesc, u32Len);
            USBD_PrepareCtrlOut(0, 0);/*准备状态阶段*/
            break;
        }
        // 取得字符串描述符
        case DESC_STRING:
        {
            // Get String Descriptor
            if(g_usbd_SetupPacket[2] < 4)
            {
                /*字符串描述符字节数和希望返回的字节数之间取一个最小值*/
                u32Len = Minimum(u32Len, g_usbd_sInfo-
>gu8StringDesc[g_usbd_SetupPacket[2]][0]);
                /*返回字符串描述符，如果数据长度大于MAX_PACKET_SIZE，该函数会自动分包传输*/
                USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo-
>gu8StringDesc[g_usbd_SetupPacket[2]], u32Len);
                USBD_PrepareCtrlOut(0, 0);/*准备状态阶段*/

                break;
            }
            else
            {
                // Not support. Reply STALL.
                USBD_SET_EP_STALL(EP0);
                USBD_SET_EP_STALL(EP1);
```

```
                break;
            }
        }
        default:
            // Not support. Reply STALL.
            USBD_SET_EP_STALL(EP0);
            USBD_SET_EP_STALL(EP1);

            DBG_PRINTF("Unsupported get desc type. stall ctrl pipe\n");

            break;
    }
}
```

准备状态阶段一般提前准备比较好，因为状态阶段一定不能回NACK，等数据发完再触发状态阶段，可能来不及。为了保险起见在准备数据阶段的时候，就把状态阶段准备好。

下面是控制传输IN ACK中断处理函数，如果要发给主机的数据大于控制传输最大包大小，剩下的就在这个函数里面传输

```
void USBD_CtrlIn(void)
{
    if(g_usbd_CtrlInSize)/*如果还有数据要发送给主机*/
    {
        // 处理剩下的数据
        if(g_usbd_CtrlInSize > g_usbd_CtrlMaxPktSize)
        {
            // 剩下的数据 > MXPLD，复制MXPLD个字节到控制端点IN Buffer
            USBD_MemCopy((uint8_t *)USBD_BUF_BASE + USBD_GET_EP_BUF_ADDR(EP0), (uint8_t
*)g_usbd_CtrlInPointer, g_usbd_CtrlMaxPktSize);
            /*触发MXPLD寄存器*/
            USBD_SET_PAYLOAD_LEN(EP0, g_usbd_CtrlMaxPktSize);
            g_usbd_CtrlInPointer += g_usbd_CtrlMaxPktSize;
            g_usbd_CtrlInSize -= g_usbd_CtrlMaxPktSize;
        }
        else
        {
            //剩下的数据 <= MXPLD，将剩下的数据全部复制到控制端点IN Buffer
            USBD_MemCopy((uint8_t *)USBD_BUF_BASE + USBD_GET_EP_BUF_ADDR(EP0), (uint8_t
*)g_usbd_CtrlInPointer, g_usbd_CtrlInSize);
            USBD_SET_PAYLOAD_LEN(EP0, g_usbd_CtrlInSize); /*触发MXPLD寄存器*/
```

```
            g_usbd_CtrlInPointer = 0;
            g_usbd_CtrlInSize = 0;
        }
    }
    else
    {
        // Set address命令状态阶段结束
        if((g_usbd_SetupPacket[0] == 0) && (g_usbd_SetupPacket[1] == 5))
        {
            if((USBD_GET_ADDR() != g_usbd_UsbAddr) && (USBD_GET_ADDR() == 0))
            {
                USBD_SET_ADDR(g_usbd_UsbAddr);/*将设备地址写到寄存器中*/
            }
        }

        // 没有数据要发送了，准备状态阶段
        USBD_PrepareCtrlOut(0, 0);
        DBG_PRINTF("Ctrl In done. Prepare OUT 0\n")
    }
}
```

主机发数据到控制端点，就会调到这个函数

```
void USBD_CtrlOut(void)
{
    uint32_t u32Size;
    /*希望接收的数据还没有接收完*/
    if(g_usbd_CtrlOutSize < g_usbd_CtrlOutSizeLimit)
    {
        u32Size = USBD_GET_PAYLOAD_LEN(EP1);
        /*将接收到的数据复制到缓存中*/
        USBD_MemCopy((uint8_t *)g_usbd_CtrlOutPointer, (uint8_t *)USBD_BUF_BASE +
USBD_GET_EP_BUF_ADDR(EP1), u32Size);
        g_usbd_CtrlOutPointer += u32Size;
        g_usbd_CtrlOutSize += u32Size;

        if(g_usbd_CtrlOutSize < g_usbd_CtrlOutSizeLimit)
            USBD_SET_PAYLOAD_LEN(EP1, g_usbd_CtrlMaxPktSize);/*触发下一次接收*/
    }
}
```

4.2.4 新唐 USB2.0 IP 特点

我们以NUC442/NUC472中的USB IP为例，说明USB 2.0 IP的特点

- 与 USB 2.0 High-Speed specification 兼容.
- 提供 1 个中断向量(NUC472/NUC442 是 88，NUC505 是 9).
- 支持 4 种传输类型
  - Control
  - Bulk
  - Interrupt
  - Isochronous
- 提供 12 个 endpoints，可以配置成除了控制端点之外的 Bulk/Interrupt/Isochronous 传输类型
- 控制端点不占用 USB IP 的 12 个端点，IP 中已经内嵌，不需要配置
- 包含 4092bytes 内部 SRAM 作为 USB buffer。这些 buffer 就是用作 IN buffer 和 OUT buffer 的，用来和主机之间收/发数据。SETUP buffer 有自己专门的 buffer
- USB buffer 不能通过 AHB 或者 APB 总线直接访问，只能通过特定寄存器或者 DMA 访问
- 提供远程唤醒功能，可以将处于休眠状态的主机唤醒.

所有的寄存器列表如下，上半部分是系统设定部分，下半部分是每个端点都有这样的一组寄存器

USBD Base Address: USBD_BA = 0x4001_9000				
USBD_GINTSTS	USBD_BA+0x000	R	Interrupt Status Low Register	0x0000_0000
USBD_GINTEN	USBD_BA+0x008	R/W	Interrupt Enable Low Register	0x0000_0001
USBD_BUSINTSTS	USBD_BA+0x010	R/W	USB Bus Interrupt Status Register	0x0000_0000
USBD_BUSINTEN	USBD_BA+0x014	R/W	USB Bus Interrupt Enable Register	0x0000_0040
USBD_OPER	USBD_BA+0x018	R/W	USB Operational Register	0x0000_0002
USBD_FRAMECNT	USBD_BA+0x01C	R	USB Frame Count Register	0x0000_0000
USBD_FADDR	USBD_BA+0x020	R/W	USB Function Address Register	0x0000_0000
USBD_TEST	USBD_BA+0x024	R/W	USB Test Mode Register	0x0000_0000
USBD_CEPDAT	USBD_BA+0x028	R/W	Control-Endpoint Data Buffer	0x0000_0000
USBD_CEPCTL	USBD_BA+0x02C	R/W	Control-Endpoint Control and Status	0x0000_0000
USBD_CEPINTEN	USBD_BA+0x030	R/W	Control-Endpoint Interrupt Enable	0x0000_0000
USBD_CEPINTSTS	USBD_BA+0x034	R/W	Control-Endpoint Interrupt Status	0x0000_1800
USBD_CEPTXCNT	USBD_BA+0x038	R/W	Control-Endpoint In-transfer Data Count	0x0000_0000
USBD_CEPRXCNT	USBD_BA+0x03C	R	Control-Endpoint Out-transfer Data Count	0x0000_0000
USBD_CEPDATCNT	USBD_BA+0x040	R	Control-Endpoint data count	0x0000_0000
USBD_SETUP1_0	USBD_BA+0x044	R	Setup1 & Setup0 bytes	0x0000_0000
USBD_SETUP3_2	USBD_BA+0x048	R	Setup3 & Setup2 Bytes	0x0000_0000
USBD_SETUP5_4	USBD_BA+0x04C	R	Setup5 & Setup4 Bytes	0x0000_0000
USBD_SETUP7_6	USBD_BA+0x050	R	Setup7 & Setup6 Bytes	0x0000_0000
USBD_CEPBUFSTART	USBD_BA+0x054	R/W	Control Endpoint RAM Start Address Register	0x0000_0000
USBD_CEPBUFEND	USBD_BA+0x058	R/W	Control Endpoint RAM End Address Register	0x0000_0000
USBD_DMACTL	USBD_BA+0x05C	R/W	DMA Control Status Register	0x0000_0000
USBD_DMACNT	USBD_BA+0x060	R/W	DMA Count Register	0x0000_0000
USBD_EPBDAT	USBD_BA+0x08C	R/W	Endpoint B Data Register	0x0000_0000
USBD_EPBINTSTS	USBD_BA+0x090	R/W	Endpoint B Interrupt Status Register	0x0000_0003
USBD_EPBINTEN	USBD_BA+0x094	R/W	Endpoint B Interrupt Enable Register	0x0000_0000
USBD_EPBDATCNT	USBD_BA+0x098	R	Endpoint B Data Available Count Register	0x0000_0000
USBD_EPBRSPCTL	USBD_BA+0x09C	R/W	Endpoint B Response Control Register	0x0000_0000
USBD_EPBMPS	USBD_BA+0x0A0	R/W	Endpoint B Maximum Packet Size Register	0x0000_0000
USBD_EPBTXCNT	USBD_BA+0x0A4	R/W	Endpoint B Transfer Count Register	0x0000_0000
USBD_EPBCFG	USBD_BA+0x0A8	R/W	Endpoint B Configuration Register	0x0000_0022
USBD_EPBBUFSTART	USBD_BA+0x0AC	R/W	Endpoint B RAM Start Address Register	0x0000_0000
USBD_EPBBUFEND	USBD_BA+0x0B0	R/W	Endpoint B RAM End Address Register	0x0000_0000

4.2.4.1 和USB1.1 IP的不同点

- USB1.1 的 USB Buffer MCU 可以直接访问，USB2.0 只能通过寄存器或者 DMA 访问
- 控制端点 USB2.0 IP 已经内嵌，不需要软件配置，USB1.1 的需要配置
- 控制传输状态阶段，USB2.0 写 NAKCLR 就行了，USB1.1 需要软件设定 DATA1，并触发 Zero 长度的包

- USB2.0 IP 的 DATA0/DATA1 都不需要软件切，USB1.1 IP 状态阶段 DATA1 需要软件帮忙切
- Buffer 设定寄存器 USB2.0 IP 需要设定起始和结束地址，USB1.1 IP 只需要设定起始地址就好。这是因为，USB2.0 传输太快，可以多分几个 MPS Buffer 给某个端点，这样只要该端点的 Buffer 中有空间，数据就会接收进来，主要 OUT 传输比较有用

后面会详细介绍 USB2.0 IP 的寄存器，看过寄存器介绍，更容易理解 USB2.0 IP 的用法

4.2.4.2 Buffer和Memory之间传输数据

Buffer就是USB IP内部的缓存，Memory就是系统的SRAM。

这个USB的Buffer不能由CPU直接访问，只能通过寄存器或者DMA的方式访问。

1) 通过寄存器的方式
   - 控制端点的 Buffer 通过数据寄存器 CEPDAT 访问。
     - IN 数据时，将数据依次写到 CEPDAT 寄存器之后，将数据个数写到 CEPTXCNT 寄存器
     - OUT 数据时，从 CEPDAT 读出数据，数据个数 CEPRXCNT
   - 其它端点的 Buffer 通过数据寄存器 EPxDAT
     - IN 数据时，将数据依次写到 EPxDAT 寄存器之后，将数据个数写到 EPxTXCNT 寄存器
     - OUT 数据时，从 EPxDAT 读出数据，数据个数 EPxDATCNT
2) 通过 DMA 的方式
   - 将 Memory 地址填入 DMAADDR 寄存器，地址要 4 对齐
   - 将长度字节填到 DMACNT 寄存器
   - 写 DMACTL 寄存器，将要操作的端点地址和 R/W 写到该寄存器里，然后使能 DMAEN，之后 DMA 会进行指定操作。将 DMACNT 个字节 IN/OUT 成功之后，会发生 DMADONE 中断

   注：DMACNT <= 4096

4.2.4.3 中断寄存器

1) 中断使能和中断状态寄存器 GINTEN/GINTSTS

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved		EPLIEN	EPKIEN	EPJIEN	EPIIEN	EPHIEN	EPGIENNN
7	6	5	4	3	2	1	0
EPFIEN	EPEIEN	EPDIEN	EPCIEN	EPBIEN	EPAIEN	CEPIEN	USBIEN

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved		EPLIF	EPKIF	EPJIF	EPIIF	EPHIF	EPGIF
7	6	5	4	3	2	1	0
EPFIF	EPEIF	EPDIF	EPCIF	EPBIF	EPAIF	CEPIF	USBIF

这两个寄存器的作用一目了然，就是使能USB中断和控制端点以及其它端点的中断，和指示中断的状态。

2) USB Bus 中断使能和中断状态寄存器 BUSINTEN/BUSINTSTS

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							VBUSDETIEN
7	6	5	4	3	2	1	0
Reserved	PHYCLKVLDIEN	DMADONEIEN	HISPDIEN	SUSPENDIEN	RESUMEIEN	RSTIEN	SOFIEN

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							VBUSDETIF
7	6	5	4	3	2	1	0
Reserved	PHYCLKVLDIF	DMADONEIF	HISPDIF	SUSPENDIF	RESUMEIF	RSTIF	SOFIF

- SOF：接收到帧起始标志发生中断
- RST：接收到总线复位信号发生中断
- RESUME：接收到总线 Resume 信号发生中断

- SUSPEND：接收到总线 Suspend 信号发生中断
- HISPD：High Speed Settle 中断，高速 USB
- DMADONE：DMA 完成中断
- PHYCLKVLD：phy 中 480M/48M 时钟稳定中断
- VBUSDET：热插拔中断

4.2.4.4 USBD_OPER寄存器

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
Reserved					CURSPD	HISPDEN	RESUMEEN

- 远程唤醒时（PHYCTL[24]），用来发送 Resume 给 Host
- HISPDEN 用来设定 USB IP 工作在 High-speed 还是 full-speed
- CURSPD：设备当前速度是全速还是高速

4.2.4.5 USBD_FADDR寄存器

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
Reserved	FADDR						

这个寄存器和USB 1.1 IP的寄存器一样，用来记录USB设备地址

4.2.4.6 控制端点寄存器

1) 数据寄存器 USBD_CEPDAT

31	30	29	28	27	26	25	24
DAT							
23	22	21	20	19	18	17	16
DAT							
15	14	13	12	11	10	9	8
DAT							
7	6	5	4	3	2	1	0
DAT							

控制端点数据收/发寄存器

2) 控制寄存器 USBD_CEPCTL

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
Reserved				FLUSH	ZEROLEN	STALLEN	NAKCLR

- STALLEN 和 NAKCLR 一起动作，00：控制传输状态阶段使能，准备好状态阶段；10：回 STALL，该位自动清 0
- ZEROLEN：回 0 长度数据包给主机
- FLUSH：清除 Buffer 中的数据同时将 CEPDATCNT 寄存器清 0

3) 中断使能和状态寄存器 CEPINTEN/CEPINTSTS

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved			BUFEMPTYIEN	BUFFULLIEN	STSDONEIEN	ERRIEN	STALLIEN
7	6	5	4	3	2	1	0
NAKIEN	RXPKIEN	TXPKIEN	PINGIEN	INTKIEN	OUTTKIEN	SETUPPKIEN	SETUPTKIEN

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved			BUFEMPTYIF	BUFFULLIF	STSDONEIF	ERRIF	STALLIF
7	6	5	4	3	2	1	0
NAKIF	RXPKIF	TXPKIF	PINGIF	INTKIF	OUTTKIF	SETUPPKIF	SETUPTKIF

这两个寄存器一个使能中断，一个表示中断的状态

- SETUPTK：收到 SETUP 令牌发生中断
- SETUPPK：收到 SETUP 包发生中断
- OUTTK：收到 OUT 令牌发生中断
- INTK：收到 IN 令牌发生中断
- PING：收到 PING 令牌发生中断
- TXPK：IN 数据之后收到 ACK 发生中断，表示发送完成
- RXPK：收到 OUT 数据包发生中断
- NAK：回 MAK 给主机发生中断
- STALL：回 STALL 给主机发生中断
- ERR：
- STSDONE：状态阶段完成发生中断
- BUFFULL：CEP Buffer 满发生中断
- BUFEMPTY：CEP Buffer 空发生中断

4) IN 传输数据个数寄存器 USBD_CEPTXCNT

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
TXCNT							

对于 IN 令牌，将要发送的字节数写到这个寄存器

5) OUT 传输数据个数寄存器

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
RXCNT							

OUT 传输中收到的字节数

6) 数据个数寄存器

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
DATCNT							
7	6	5	4	3	2	1	0
DATCNT							

该寄存器是只读的，显示控制端点 Buffer 中的字节数

7) SETUP1_0/SETUP3_2/SETUP5_4/SETUP7_6 寄存器

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
SETUP1							
7	6	5	4	3	2	1	0
SETUP0							

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
SETUP3							
7	6	5	4	3	2	1	0
SETUP2							

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
SETUP5							
7	6	5	4	3	2	1	0
SETUP4							

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
SETUP7							
7	6	5	4	3	2	1	0
SETUP6							

这 4 个寄存器用于存放 SETUP 命令，共 8 个字节

8) Buffer 起始地址和结束地址寄存器

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved				SADDR			
7	6	5	4	3	2	1	0
SADDR							

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved				EADDR			
7	6	5	4	3	2	1	0
EADDR							

控制端点起始地址和结束地址寄存器

4.2.4.7 DMA寄存器

1) DMACTL 寄存器

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
DMARST	SGEN	DMAEN	DMARD	EPNUM			

- EPNUM：操作的端点地址
- DMARD：读 Memory 还是写 Memory
- DMAEN：触发 DMA 开始 DMA 操作
- SGEN：使能 DMA Scatter Gather 功能
- DMARST：复位 DMA，DMA 一旦使能就不能停止，可以通过该 bit 让它停止

2) DMACNT 寄存器

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved				DMACNT			
15	14	13	12	11	10	9	8
DMACNT							
7	6	5	4	3	2	1	0
DMACNT							

DMA 要读/写的字节数

3) DMAADDR 寄存器

31	30	29	28	27	26	25	24
DMAADDR							
23	22	21	20	19	18	17	16
DMAADDR							
15	14	13	12	11	10	9	8
DMAADDR							
7	6	5	4	3	2	1	0
DMAADDR							

Memory 地址寄存器，DMA 将 Memory 中数据传到 USB 内部 Buffer 或者将内部 Buffer 的数据传到外部 Memory 中

4.2.4.8 端点寄存器

下面的寄存器每个端点都有一份

1) DAT 寄存器

31	30	29	28	27	26	25	24
EPDAT							
23	22	21	20	19	18	17	16
EPDAT							
15	14	13	12	11	10	9	8
EPDAT							
7	6	5	4	3	2	1	0
EPDAT							

端点数据收/发寄存器，不用 DMA 时就用这个寄存器收/发

2) 中断使能和状态寄存器

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved			SHORTRXIF	ERRIF	NYETIF	STALLIF	NAKIF
7	6	5	4	3	2	1	0
PINGIF	INTKIF	OUTTKIF	RXPKIF	TXPKIF	SHORTTXIF	BUFEMPTYIF	BUFFULLIF

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved			SHORTRXIEN	ERRIEN	NYETIEN	STALLIEN	NAKIEN
7	6	5	4	3	2	1	0
PINGIEN	INTKIEN	OUTTKIEN	RXPKIEN	TXPKIEN	SHORTTXIEN	BUFEMPTYIEN	BUFFULLIEN

这两个寄存器一个使能中断，一个表示中断的状态

- BUFFULL：端点 Buffer 满发生中断
- BUFEMPTY：端点 Buffer 空发生中断

- SHORTTX：发送短包结束发生中断，SHORTTXEN(EPxRSPCTL[6])使能短包发送。用于 Auto-Validate 模式
- TXPK：IN 数据之后收到 ACK 发生中断，表示发送完成
- RXPK：收到 OUT 数据包发生中断
- OUTTK：收到 OUT 令牌发生中断
- INTK：收到 IN 令牌发生中断
- PING：收到 PING 令牌发生中断
- NAK：回 MAK 给主机发生中断
- STALL：回 STALL 给主机发生中断
- NYET：该端点剩下的 Buffer 不够放剩下的 OUT 数据就会发生 NYET 中断
- ERR：
- SHORTRX：接收到 BULK-out 短包（<MPS 的包）发生中断，DISBUF(EPxRSPCTL[7])使能短包接收。

3) DATCNT 寄存器

31	30	29	28	27	26	25	24
Reserved	DMALOOP						
23	22	21	20	19	18	17	16
DMALOOP							
15	14	13	12	11	10	9	8
DATCNT							
7	6	5	4	3	2	1	0
DATCNT							

该寄存器是只读的，IN 传输时返回 Buffer 中等该发送的有效字节数，OUT 传输时返回 Buffer 中收到的有效字节数。

4) RSPCTL 寄存器

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
DISBUF	SHORTTXEN	ZEROLEN	HALT	TOGGLE	MODE		FLUSH

- FLUSH：清除 Buffer 中的数据，同时将 DATCNT 清 0

- MODE：用于设定 IN 传输的模式，收到 IN 令牌时
  - Auto ：满 Max Packet Size ，HW 自动帮忙送，否则回 NAK
  - Manual_Validate：手动写 TXCNT 才发送
  - Fly：buffer 中有多少数据就发多少数据

5) MPS 寄存器

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved					EPMPS		
7	6	5	4	3	2	1	0
EPMPS							

定义该端点最大包大小

6) TXCNT 寄存器

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
					TXCNT		
7	6	5	4	3	2	1	0
TXCNT							

Manual 模式下，要发送的字节数。只用于 IN 传输。

7) CFG 寄存器

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved						Reserved	
7	6	5	4	3	2	1	0
EPNUM				EPDIR	EPTYPE		EPEN

- EPEN：使能端点
- EPTYPE：设定端点为 BULK、中断、等时传输类型
- EPDIR：设定端点传输方向 IN/OUT
- EPNUM：设定端点号

8) RAM 起始地址和结束地址寄存器 EPxBUFSTART 和 EPxBUFEND

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved				SADDR			
7	6	5	4	3	2	1	0
SADDR							

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved				EADDR			
7	6	5	4	3	2	1	0
EADDR							

设定该端点Buffer起始地址和结束地址

4.2.5 代码分析

了解USB协议和新唐高速USB IP用法之后，下面的例子比较容易理解。

该代码也是HID transfer的例子。传输不使用DMA，而用DAT寄存器。DMA传输一般用于BULK比较有用，所以在USB Mass Storage的例子里面会看到DMA的用法。

4.2.5.1 中断处理函数

给大家讲解一下中断处理函数，了解整个USB的处理流程

```
void USBD_IRQHandler(void)
{
    __IO uint32_t IrqStL, IrqSt;
```

```
    IrqStL = USBD->GINTSTS & USBD->GINTEN;    /* get interrupt status */

    if (!IrqStL)    return;

    /* USB 中断 */
    if (IrqStL & USBD_GINTSTS_USBIF_Msk) {
        IrqSt = USBD->BUSINTSTS & USBD->BUSINTEN;

        if (IrqSt & USBD_BUSINTSTS_SOFIF_Msk)
            USBD_CLR_BUS_INT_FLAG(USBD_BUSINTSTS_SOFIF_Msk);
        /*收到USB总线复位信号*/
        if (IrqSt & USBD_BUSINTSTS_RSTIF_Msk) {
            USBD_SwReset();
            /*复位DMA*/
            USBD_ResetDMA();
            /*清除EPA和EPB的buffer*/
            USBD->EPARSPCTL = USBD_EPRSPCTL_FLUSH_Msk;
            USBD->EPBRSPCTL = USBD_EPRSPCTL_FLUSH_Msk;

            if (USBD->OPER & 0x04)  /* 高速设备 */
                HID_InitForHighSpeed();
            else                    /* 全速设备 */
                HID_InitForFullSpeed();
            /*使能SETUP包中断*/
            USBD_ENABLE_CEP_INT(USBD_CEPINTEN_SETUPPKIEN_Msk);
            USBD_SET_ADDR(0);

            USBD_ENABLE_BUS_INT(USBD_BUSINTEN_RSTIEN_Msk|USBD_BUSINTEN_RESUMEIEN_Msk|USBD_
            BUSINTEN_SUSPENDIEN_Msk);
            USBD_CLR_BUS_INT_FLAG(USBD_BUSINTSTS_RSTIF_Msk);
            USBD_CLR_CEP_INT_FLAG(0x1ffc);
        }

        if (IrqSt & USBD_BUSINTSTS_RESUMEIF_Msk) {
            USBD_ENABLE_BUS_INT(USBD_BUSINTEN_RSTIEN_Msk|USBD_BUSINTEN_SUSPENDIEN_Msk);
            USBD_CLR_BUS_INT_FLAG(USBD_BUSINTSTS_RESUMEIF_Msk);
        }

        if (IrqSt & USBD_BUSINTSTS_SUSPENDIF_Msk) {
            USBD_ENABLE_BUS_INT(USBD_BUSINTEN_RSTIEN_Msk | USBD_BUSINTEN_RESUMEIEN_Msk);
            USBD_CLR_BUS_INT_FLAG(USBD_BUSINTSTS_SUSPENDIF_Msk);
```

```
        }

        if (IrqSt & USBD_BUSINTSTS_HISPDIF_Msk) {
            USBD_ENABLE_CEP_INT(USBD_CEPINTEN_SETUPPKIEN_Msk);
            USBD_CLR_BUS_INT_FLAG(USBD_BUSINTSTS_HISPDIF_Msk);
        }
        /*DMA传输完成中断，因为没有使能DMA，所以这里并不会走到*/
        if (IrqSt & USBD_BUSINTSTS_DMADONEIF_Msk) {
            g_usbd_DmaDone = 1;
            printf("Read command - Complete\n");
            USBD_CLR_BUS_INT_FLAG(USBD_BUSINTSTS_DMADONEIF_Msk);
            /*读Memory中的数据，用于IN传输*/
            if (USBD->DMACTL & USBD_DMACTL_DMARD_Msk) {
                if (g_usbd_ShortPacket == 1) {/*最后一包<MPS*/
                    USBD->EPARSPCTL = USB_EP_RSPCTL_SHORTTXEN;    // 使能短包发送
                    g_usbd_ShortPacket = 0;
                }
            }
        }

        if (IrqSt & USBD_BUSINTSTS_PHYCLKVLDIF_Msk)
            USBD_CLR_BUS_INT_FLAG(USBD_BUSINTSTS_PHYCLKVLDIF_Msk);
        /*热插拔中断*/
        if (IrqSt & USBD_BUSINTSTS_VBUSDETIF_Msk) {
            if (USBD_IS_ATTACHED()) {
                /* USB Plug In */
                USBD_ENABLE_USB();
            } else {
                /* USB Un-plug */
                USBD_DISABLE_USB();
            }
            USBD_CLR_BUS_INT_FLAG(USBD_BUSINTSTS_VBUSDETIF_Msk);
        }
    }
    /*控制端点中断*/
    if (IrqStL & USBD_GINTSTS_CEPIF_Msk) {
        IrqSt = USBD->CEPINTSTS & USBD->CEPINTEN;
        /*收到SETUP令牌中断*/
        if (IrqSt & USBD_CEPINTSTS_SETUPTKIF_Msk) {
            USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_SETUPTKIF_Msk);
            return;
```

```
        }
        /*收到SETUP包中断*/
        if (IrqSt & USBD_CEPINTSTS_SETUPPKIF_Msk) {
            USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_SETUPPKIF_Msk);
            USBD_ProcessSetupPacket();
            return;
        }
        /*收到OUT 令牌中断*/
        if (IrqSt & USBD_CEPINTSTS_OUTTKIF_Msk) {
            USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_OUTTKIF_Msk);
            USBD_ENABLE_CEP_INT(USBD_CEPINTEN_STSDONEIEN_Msk);
            return;
        }
        /*收到IN 令牌中断*/
        if (IrqSt & USBD_CEPINTSTS_INTKIF_Msk) {
            USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_INTKIF_Msk);
            if (!(IrqSt & USBD_CEPINTSTS_STSDONEIF_Msk)) {
                USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_TXPKIF_Msk);
                USBD_ENABLE_CEP_INT(USBD_CEPINTEN_TXPKIEN_Msk);/*使能发送完成中断*/
                USBD_CtrlIn();
            } else {
                USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_TXPKIF_Msk);
                /*使能发送完成和状态阶段完成中断*/
                 USBD_ENABLE_CEP_INT(USBD_CEPINTEN_TXPKIEN_Msk|USBD_CEPINTEN_STSDONEIEN_Msk
                 );
            }
            return;
        }

        if (IrqSt & USBD_CEPINTSTS_PINGIF_Msk) {
            USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_PINGIF_Msk);
            return;
        }
        /*发送完成中断*/
        if (IrqSt & USBD_CEPINTSTS_TXPKIF_Msk) {
            USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_STSDONEIF_Msk);
            USBD_SET_CEP_STATE(USB_CEPCTL_NAKCLR);/*触发状态阶段*/
            if (g_usbd_CtrlInSize) {/*如果还有数据要发送*/
                USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_INTKIF_Msk);
                USBD_ENABLE_CEP_INT(USBD_CEPINTEN_INTKIEN_Msk);/*使能IN令牌中断*/
            } else {
```

```
                USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_STSDONEIF_Msk);

                USBD_ENABLE_CEP_INT(USBD_CEPINTEN_SETUPPKIEN_Msk|USBD_CEPINTEN_STSDONEIEN_
                Msk);
            }
            USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_TXPKIF_Msk);
            return;
        }
        /*接收到数据包中断*/
        if (IrqSt & USBD_CEPINTSTS_RXPKIF_Msk) {
            USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_RXPKIF_Msk);
            USBD_SET_CEP_STATE(USB_CEPCTL_NAKCLR);

            USBD_ENABLE_CEP_INT(USBD_CEPINTEN_SETUPPKIEN_Msk|USBD_CEPINTEN_STSDONEIEN_Msk)
            ;
            return;
        }

        if (IrqSt & USBD_CEPINTSTS_NAKIF_Msk) {
            USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_NAKIF_Msk);
            return;
        }

        if (IrqSt & USBD_CEPINTSTS_STALLIF_Msk) {
            USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_STALLIF_Msk);
            return;
        }

        if (IrqSt & USBD_CEPINTSTS_ERRIF_Msk) {
            USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_ERRIF_Msk);
            return;
        }
        /*状态阶段完成中断*/
        if (IrqSt & USBD_CEPINTSTS_STSDONEIF_Msk) {
            USBD_UpdateDeviceState();
            USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_STSDONEIF_Msk);
            USBD_ENABLE_CEP_INT(USBD_CEPINTEN_SETUPPKIEN_Msk);
            return;
        }

        if (IrqSt & USBD_CEPINTSTS_BUFFULLIF_Msk) {
            USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_BUFFULLIF_Msk);
```

```
            return;
        }

        if (IrqSt & USBD_CEPINTSTS_BUFEMPTYIF_Msk) {
            USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_BUFEMPTYIF_Msk);
            return;
        }
    }

    /* 中断IN传输 */
    if (IrqStL & USBD_GINTSTS_EPAIF_Msk) {
        IrqSt = USBD->EPAINTSTS & USBD->EPAINTEN;
        if (USBD->EPAINTSTS & 0x02)
            EPA_Handler();
        USBD_CLR_EP_INT_FLAG(EPA, IrqSt);
    }
    /* 中断OUT传输 */
    if (IrqStL & USBD_GINTSTS_EPBIF_Msk) {
        IrqSt = USBD->EPBINTSTS & USBD->EPBINTEN;
        if (USBD->EPBINTSTS & 0x01)
            EPB_Handler();
        USBD_CLR_EP_INT_FLAG(EPB, IrqSt);
    }

    if (IrqStL & USBD_GINTSTS_EPCIF_Msk) {
        IrqSt = USBD->EPCINTSTS & USBD->EPCINTEN;
        USBD_CLR_EP_INT_FLAG(EPC, IrqSt);
    }

    if (IrqStL & USBD_GINTSTS_EPDIF_Msk) {
        IrqSt = USBD->EPDINTSTS & USBD->EPDINTEN;
        USBD_CLR_EP_INT_FLAG(EPD, IrqSt);
    }

    if (IrqStL & USBD_GINTSTS_EPEIF_Msk) {
        IrqSt = USBD->EPEINTSTS & USBD->EPEINTEN;
        USBD_CLR_EP_INT_FLAG(EPE, IrqSt);
    }

    if (IrqStL & USBD_GINTSTS_EPFIF_Msk) {
        IrqSt = USBD->EPFINTSTS & USBD->EPFINTEN;
```

```
        USBD_CLR_EP_INT_FLAG(EPF, IrqSt);
    }

    if (IrqStL & USBD_GINTSTS_EPGIF_Msk) {
        IrqSt = USBD->EPGINTSTS & USBD->EPGINTEN;
        USBD_CLR_EP_INT_FLAG(EPG, IrqSt);
    }

    if (IrqStL & USBD_GINTSTS_EPHIF_Msk) {
        IrqSt = USBD->EPHINTSTS & USBD->EPHINTEN;
        USBD_CLR_EP_INT_FLAG(EPH, IrqSt);
    }

    if (IrqStL & USBD_GINTSTS_EPIIF_Msk) {
        IrqSt = USBD->EPIINTSTS & USBD->EPIINTEN;
        USBD_CLR_EP_INT_FLAG(EPI, IrqSt);
    }

    if (IrqStL & USBD_GINTSTS_EPJIF_Msk) {
        IrqSt = USBD->EPJINTSTS & USBD->EPJINTEN;
        USBD_CLR_EP_INT_FLAG(EPJ, IrqSt);
    }

    if (IrqStL & USBD_GINTSTS_EPKIF_Msk) {
        IrqSt = USBD->EPKINTSTS & USBD->EPKINTEN;
        USBD_CLR_EP_INT_FLAG(EPK, IrqSt);
    }

    if (IrqStL & USBD_GINTSTS_EPLIF_Msk) {
        IrqSt = USBD->EPLINTSTS & USBD->EPLINTEN;
        USBD_CLR_EP_INT_FLAG(EPL, IrqSt);
    }
}
```

这个中断处理函数比较长，处理的中断也很多，大家只看注释有点难度。

控制端点处理流程如下：

1) 默认使能 SETUP 包中断
2) 发生 SETUP 包中断时，如果需要 IN 传输使能 IN 令牌中断，调用函数 USBD_ProcessSetupPacket()
3) 发生 IN 令牌中断时，调用 USBD_CtrlIn();填数据，使能发送一包结束中断 TXPK

4) 发生 TXPK 中断时，触发状态阶段，检测是否还有数据要发送，如果有就再次使能 IN 令牌中断，重复步骤 3)；如果没有，再次使能 SETUP 包中断

上面步骤2）如果没有数据阶段，直接就到状态阶段，则触发状态阶段，并使能状态阶段完成中断。

中断IN端点处理流程：

1) 将数据填入 IN Buffer 中，使能 IN 令牌中断
2) 发生 EPA 中断，调用 EPA_Handler

中断OUT端点处理流程：

1) 默认使能接收到数据中断 RXPK
2) 发生 EPB 中断，调用 EPB_Handler

EPA被配置为中断IN，EPB被配置为中断OUT，所以EPA_Handler用于处理要发送到主机的数据， EPB_Handler用于接收从主机发来的数据。要发送到主机的数据可以提前通过下面的代码写到EPA IN Buffer里面，如果要发送的数据大于最大包大小，剩下的就在EPA_Handler里面发送，每次最多填MAX_PACKET_SIZE到IN Buffer里面。

```
/*填中断IN Buffer*/
for (i=0; i<len; i++)
    USBD->EPADAT_BYTE = g_u8PageBuff[i];/*将数据写到EPA Buffer中*/
USBD->EPATXCNT = len;/*发送长度*/
USBD_ENABLE_EP_INT(EPA, USBD_EPINTEN_INTKIEN_Msk);/*使能IN令牌中断*/
```

EPB被配置为中断OUT，每收到一包Host发来的数据就会进入一次函数EPB_Handler

```
/* 中断 IN 处理函数，每调用一次表示发送一包到主机成功 */
void EPA_Handler(void)
{
    HID_SetInReport();
}
static uint8_t  g_u8PageBuff[256],g_u32BytesInPageBuf;
/* 中断 OUT 处理函数，每调用一次表示从主机收到一包数据*/
void EPB_Handler(void)
{
    uint32_t len, i;
    /*接收到的数据长度*/
    len = USBD->EPBDATCNT & 0xffff;
    /*将数据读出来*/
    for (i=0; i<len; i++)
        g_u8OutBuff[i] = USBD->EPBDAT_BYTE;
    /*处理收到的数据*/
```

```
    HID_GetOutReport(g_u8OutBuff, len);
}
```

了解上面的代码之后，通过中断IN/OUT收/发数据的流程就清楚了。

4.2.5.2 HID Transfer 设备枚举

HID报告描述符和HID设备数据传输就不讲了，大家可以参考HID报告描述符分析 和HID设备数据传输两节。

枚举过程的处理函数主要是**USBD_ProcessSetupPacket();USBD_CtrlIn();和USBD_CtrlOut();**

枚举过程主要是主机发送命令，从机通过函数USBD_ProcessSetupPacket分析命令，然后通过USBD_CtrlIn和USBD_CtlOut收/发数据。

数据的收/发过程和中断IN/OUT(EPA_Hander/EPB_Hander) 是一样的，唯一不同的是控制端点有个状态阶段

```
void USBD_ProcessSetupPacket(void)
{
    /*取得SETUP包 */
    gUsbCmd.bmRequestType = (uint8_t)(USBD->SETUP1_0 & 0xff);
    gUsbCmd.bRequest = (int8_t)(USBD->SETUP1_0 >> 8) & 0xff;
    gUsbCmd.wValue = (uint16_t)USBD->SETUP3_2;
    gUsbCmd.wIndex = (uint16_t)USBD->SETUP5_4;
    gUsbCmd.wLength = (uint16_t)USBD->SETUP7_6;
    /* 检查命令类型 */
    switch(g_usbd_SetupPacket[0] & 0x60)
    {
        case REQ_STANDARD:   // 标准USB命令
        {
            USBD_StandardRequest();
            break;
        }
        case REQ_CLASS:   // USB Class命令
        {
            if(g_usbd_pfnClassRequest != NULL)
            {
                g_usbd_pfnClassRequest();
            }
            break;
        }
        case REQ_VENDOR:   // Vendor命令
```

```
        {
            if(g_usbd_pfnVendorRequest != NULL)
            {
                g_usbd_pfnVendorRequest();
            }
            break;
        }
        default:   // reserved
        {
            /* Setup error, stall the device */
            USBD_SET_CEP_STATE(USBD_CEPCTL_STALLEN_Msk);
            break;
        }
    }
}
```

标准USB命令的处理函数有点复杂，其实也就是根据不同的USB命令进行不同的处理。下面的代码并不完整，感兴趣的可以到BSP里面usbd.c中有完整版。给大家分析几个典型的命令，其它的命令处理方式都是一样的。

```
void USBD_StandardRequest(void)
{
    /* 清除全局变量 */
    g_usbd_CtrlInPointer = 0;
    g_usbd_CtrlInSize = 0;

    if(gUsbCmd.bmRequestType & 0x80)    /* 该命令数据阶段的方向是Device - > Host */
    {
        // Device to host
        switch(gUsbCmd.bRequest)
        {
            case GET_CONFIGURATION:
            {
                // 返回当前配置设定
                USBD_PrepareCtrlIn((uint8_t *)&g_usbd_UsbConfig, 1);
                /*使能IN令牌中断，安全起见，先清除一下。发生IN令牌中断时会调用USBD_CtrlIn，
                  将返回数据写入控制端点IN Buffer中，然后会使能TXPK中断，发生TXPK中断时触发状
                  态阶段，并使能状态完成中断*/
                USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_INTKIF_Msk);
                USBD_ENABLE_CEP_INT(USBD_CEPINTEN_INTKIEN_Msk);
                break;
```

```
            }
            case GET_DESCRIPTOR:/*取得各类描述符*/
            {
                if (!USBD_GetDescriptor()) {/*返回0说明有数据阶段*/
                    USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_INTKIF_Msk);
                    USBD_ENABLE_CEP_INT(USBD_CEPINTEN_INTKIEN_Msk);/*使能IN令牌中断*/
                }
                break;
            }
            ……
        }
    }
    else/* 该命令数据阶段的方向是Host - > Device */
    {
        // Host to device
        switch(g_usbd_SetupPacket[1])
        {
            case SET_ADDRESS:/*该命令没有数据阶段*/
            {
                g_usbd_UsbAddr = (uint8_t)gUsbCmd.wValue;

                /* 状态阶段 */
                USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_STSDONEIF_Msk);
                USBD_SET_CEP_STATE(USB_CEPCTL_NAKCLR);//触发状态阶段
                USBD_ENABLE_CEP_INT(USBD_CEPINTEN_STSDONEIEN_Msk);//使能状态完成中断

                break;
            }
            case SET_CONFIGURATION: /*该命令没有数据阶段*/
            {
                g_usbd_UsbConfig = (uint8_t)gUsbCmd.wValue;
                g_usbd_Configured = 1;
                /* 状态阶段 */
                USBD_CLR_CEP_INT_FLAG(USBD_CEPINTSTS_STSDONEIF_Msk);
                USBD_SET_CEP_STATE(USB_CEPCTL_NAKCLR); //触发状态阶段
                USBD_ENABLE_CEP_INT(USBD_CEPINTEN_STSDONEIEN_Msk);//使能状态完成中断
                break;
            }
            ……
        }
    }
```

```
}
```

收到SET_CONFIGURATION命令表示主机枚举成功。

下面着重介绍USBD_GetDescriptor()和USBD_CtrlIn和USBD_CtlOut函数，枚举过程拿各种描述符是关键。而描述符就要通过USBD_CtrlIn传给Host。

描述符分为设备描述符，配置描述符和字符串描述符；另外还有HID Class特有的HID 报告描述符

```
int USBD_GetDescriptor(void)
{
    uint32_t u32Len;

    u32Len = gUsbCmd.wLength;
    g_usbd_CtrlZero = 0;

    switch ((gUsbCmd.wValue & 0xff00) >> 8) {
    // 取得设备描述符
    case DESC_DEVICE: {
        u32Len = Minimum(u32Len, LEN_DEVICE);
        USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo->gu8DevDesc, u32Len);
        break;
    }
    // 取得配置描述符
    case DESC_CONFIG: {
        uint32_t u32TotalLen;

        if (USBD->OPER & 0x04) { /* high speed */
            u32TotalLen = g_usbd_sInfo->gu8ConfigDesc[3];
            u32TotalLen = g_usbd_sInfo->gu8ConfigDesc[2] + (u32TotalLen << 8);

            u32Len = Minimum(u32Len, u32TotalLen);
            if ((u32Len % g_usbd_CtrlMaxPktSize) == 0)
                g_usbd_CtrlZero = 1;

            USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo->gu8ConfigDesc, u32Len);
        } else { /* full speed */
            u32TotalLen = g_usbd_sInfo->gu8OtherConfigDesc[3];
            u32TotalLen = g_usbd_sInfo->gu8OtherConfigDesc[2] + (u32TotalLen << 8);

            u32Len = Minimum(u32Len, u32TotalLen);
```

```
            if ((u32Len % g_usbd_CtrlMaxPktSize) == 0)
                g_usbd_CtrlZero = 1;

            USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo->gu8OtherConfigDesc, u32Len);
        }
        break;
    }
    /* 取得设备 Qualifier 描述符，同时支持全速与高速的设备，必须同时提供这个描述符和
       Other_Speed_configuration描述符。用来报告当前没有在使用的速度信息。当设备速度改变时，
       某些字段可以改变
     */
    case DESC_QUALIFIER: {
        u32Len = Minimum(u32Len, LEN_QUALIFIER);
        USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo->gu8QualDesc, u32Len);
        break;
    }
    /* 取得Other Speed Descriptor，同时支持全速与高速的设备，必须同时提供这个描述符和
       Qualifier描述符。用来报告当前没有在使用的速度信息。当设备速度改变时，某些字段可以改变
     */
    case DESC_OTHERSPEED: {
        uint32_t u32TotalLen;

        u32TotalLen = g_usbd_sInfo->gu8OtherConfigDesc[3];
        u32TotalLen = g_usbd_sInfo->gu8OtherConfigDesc[2] + (u32TotalLen << 8);

        u32Len = Minimum(u32Len, u32TotalLen);
        if ((u32Len % g_usbd_CtrlMaxPktSize) == 0)
            g_usbd_CtrlZero = 1;

        USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo->gu8OtherConfigDesc, u32Len);
        break;
    }
    // 取得HID 描述符
    case DESC_HID: {
        u32Len = Minimum(u32Len, LEN_HID);
        USBD_MemCopy(g_usbd_buf, (uint8_t *)&g_usbd_sInfo-
>gu8ConfigDesc[LEN_CONFIG+LEN_INTERFACE], u32Len);
        USBD_PrepareCtrlIn(g_usbd_buf, u32Len);
        break;
    }
    // 取得HID Report 描述符
```

```
    case DESC_HID_RPT: {
        if ((u32Len % g_usbd_CtrlMaxPktSize) == 0)
            g_usbd_CtrlZero = 1;

        switch (gUsbCmd.wIndex & 0xff) {
        case 0: {
            u32Len = Minimum(u32Len, g_usbd_sInfo->gu32HidReportSize[0]);
            USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo->gu8HidReportDesc[0], u32Len);
            break;
        }
        case 1: {
            u32Len = Minimum(u32Len, g_usbd_sInfo->gu32HidReportSize[1]);
            USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo->gu8HidReportDesc[1], u32Len);
            break;
        }
        case 2: {
            u32Len = Minimum(u32Len, g_usbd_sInfo->gu32HidReportSize[2]);
            USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo->gu8HidReportDesc[2], u32Len);
            break;
        }
        }
        break;
    }
    // 取得字符串描述符
    case DESC_STRING: {
        // Get Language
        if ((gUsbCmd.wValue & 0xff) == 0) {
            u32Len = Minimum(u32Len, 4);
            USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo->gu8StrLangDesc, u32Len);
        } else {
            // Get String Descriptor
            switch (gUsbCmd.wValue & 0xff) {
            case 1: {
                u32Len = Minimum(u32Len, g_usbd_sInfo->gu8StrVendorDesc[0]);
                if ((u32Len % g_usbd_CtrlMaxPktSize) == 0)
                    g_usbd_CtrlZero = 1;

                USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo->gu8StrVendorDesc, u32Len);
                break;
            }
            case 2: {
```

```
                u32Len = Minimum(u32Len, g_usbd_sInfo->gu8StrProductDesc[0]);
                if ((u32Len % g_usbd_CtrlMaxPktSize) == 0)
                    g_usbd_CtrlZero = 1;

                USBD_PrepareCtrlIn((uint8_t *)g_usbd_sInfo->gu8StrProductDesc, u32Len);
                break;
            }
            default:
                // Not support. Reply STALL.
                USBD_SET_CEP_STATE(USBD_CEPCTL_STALLEN_Msk);
                return 1;
            }
        }
        break;
    }
    default:
        // Not support. Reply STALL.
        USBD_SET_CEP_STATE(USBD_CEPCTL_STALLEN_Msk);
        return 1;
    }
    return 0;
}
```

准备状态阶段一般提前准备比较好，因为状态阶段一定不能回NACK，等数据发完再触发状态阶段，可能来不及。为了保险起见在准备数据阶段的时候，就把状态阶段准备好。这个代码是在TXPK时触发状态阶段的，如果接下来还有数据要发肯定没有问题，如果MCU比较忙可能会有问题?!

下面是控制传输IN ACK中断处理函数，如果要发给主机的数据大于控制传输最大包大小，剩下的就在这个函数里面传输。

```
void USBD_CtrlIn(void)
{
    int volatile i;
    uint32_t volatile count;

    // 处理剩下的数据
    if(g_usbd_CtrlInSize >= g_usbd_CtrlMaxPktSize) {
        // 剩下的数据 >= MXPLD
        for (i=0; i<(g_usbd_CtrlMaxPktSize >> 2); i++, g_usbd_CtrlInPointer+=4)
            USBD->CEPDAT = *(uint32_t *)g_usbd_CtrlInPointer;
```

```
        USBD_START_CEP_IN(g_usbd_CtrlMaxPktSize);/*填TXCNT寄存器*/
        g_usbd_CtrlInSize -= g_usbd_CtrlMaxPktSize;
    } else {
        // 剩下的数据 < MXPLD
        for (i=0; i<(g_usbd_CtrlInSize >> 2); i++, g_usbd_CtrlInPointer+=4)
            USBD->CEPDAT = *(uint32_t *)g_usbd_CtrlInPointer;

        count = g_usbd_CtrlInSize % 4;
        for (i=0; i<count; i++)
            USBD->CEPDAT_BYTE = *(uint8_t *)(g_usbd_CtrlInPointer + i);

        USBD_START_CEP_IN(g_usbd_CtrlInSize); /*填TXCNT寄存器*/
        g_usbd_CtrlInPointer = 0;
        g_usbd_CtrlInSize = 0;
    }
}
```

主机发数据到控制端点，就会调到这个函数

```
void USBD_CtrlOut(uint8_t *pu8Buf, uint32_t u32Size)
{
    int volatile i;

    while(1) {
        /**/
        if (USBD->CEPINTSTS & USBD_CEPINTSTS_RXPKIF_Msk) {
            /*将数据读出*/
            for (i=0; i<u32Size; i++)
                *(uint8_t *)(pu8Buf + i) = USBD->CEPDAT_BYTE;
            USBD->CEPINTSTS = USBD_CEPINTSTS_RXPKIF_Msk;
            break;
        }
    }
}
```

HID Transfer 代码中没有用到USBD_CtrlOut，它是用中断IN/OUT传输数据的。也就是在EPA_Handler和EPB_Handler里面处理的。

4.2.6 总结

1) USB 设备不能自己发送数据给主机。不管数据收还是发都是由主机控制的
2) 控制传输有 3 个阶段，要小心处理状态阶段

3) USB1.1 IP 控制传输的状态阶段由软件切 DATA1，另外 SETUP 使用 DATA0，所以第一包 IN 的数据也要切 DATA1。之后 HW 会负责 DATA0/DATA1 轮流切
4) USB2.0 的 DATA0/DATA1 不用软件介入

4.3 ISO7816

新唐的很多芯片都带ISO7816接口，目前最多带6个SmartCard接口。与ISO7816-3和EMV2000兼容。有DATA、PWR、CLK、RST 共4根脚。CLK脚的频率可以编程。

有发送/接收FIFO，中断触发阀值可以编程。

SC接口可以当作UART口使用。可以设置UART波特率、停止位、校验位，以及支持收/发FIFO。

4.3.1 ISO7816 协议简介

IC卡的使用流程如下，首先上电然后拿到ATR，之后就可以收发命令。详细如下：

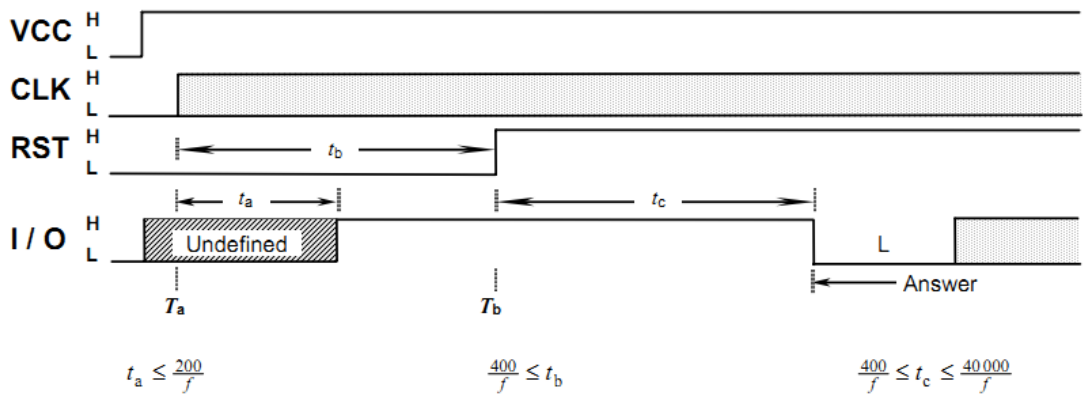
- Activation 然后 IC 返回 ATR
- 分析 ATR 得到初始化参数
- 收/发命令
- Deactivation

4.3.1.1 Activation

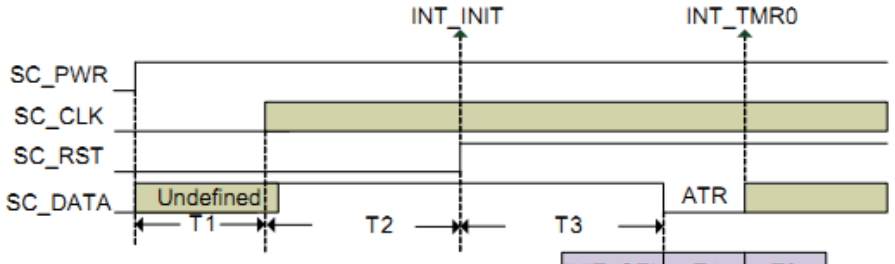
Activation时序如下，

- EMV2000
  - 根据 EMV2000 的 spec， VCC 拉 high 的时间不能超过 1us。
  - CLK 输出之后，200 个 CLK 时钟之内 IO 要拉 high
  - IO 拉 high 之后要等个 40000 个 CLK 时钟，才能将 RST 拉 high
  - RST 拉 high 之后最多等 42000 个 CLK 时钟，等待卡返回 ATR
- ISO7816-3

下面的时序图是IS7816-3的规定，但是只要符合EMV2000的spec，就符合ISO7816的



新唐的SC支持硬件Activation， 时序如下：



Time	Comment
T1	SC_PWR to SC_CLK Start
T2	SC_CLK Start to SC_RST Assert
T3	SC_CLK Start to ART Appear

INIT_SEL	T1	T2
00	85	489
01	133	537
10	165	569
11	165	42060

Unit : SC Clock

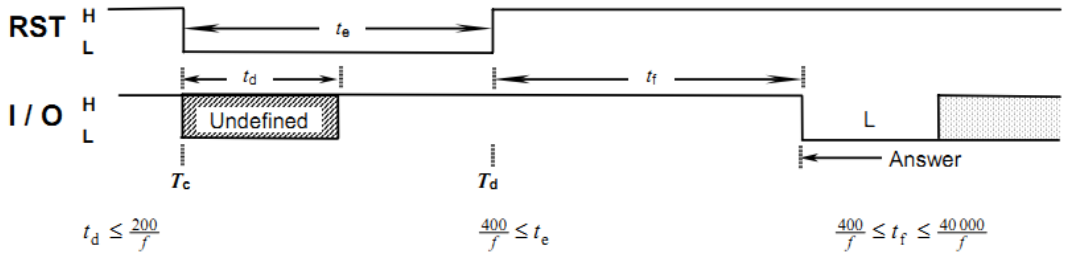
Note : These value are measured by chip IO pin, the real value will depended on system design

上图中有几个关键的时间：T1、T2、T3。T1的时间最长165 SC clock，如果你的系统PWR需要很久才能上电，硬件Activation就不太适用，只能软件定时了。

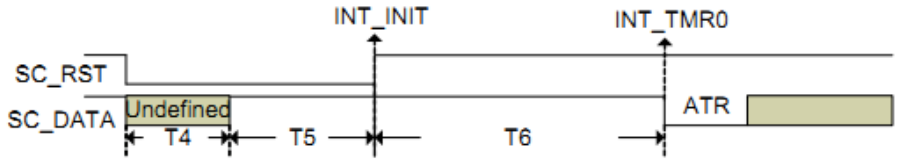
如果你的卡不是符合上述规范的，则VCC拉higi到CLK 输出，CLK输出到RST拉high，CLK输出到IO拉high，这之间的时序只能你自己修改了。

4.3.1.2 Warm Reset

系统第一次上电进行的Reset叫Cold Reset，以后进行的叫Warm Reset。时序和Activation一样，只是没有VCC和CLK信号



下面是新唐硬件Warm Reset的时序，T4/T5可以设定



Time	Comment
T4	SC_RST to SC_DATA Reception Mode
T5	SC_DATA Reception Mode to SC_RST Assert
T6	SC_CLK Start to ATR Appear

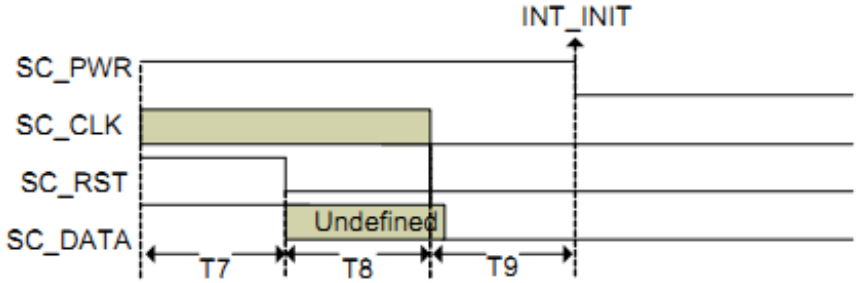
INIT_SEL	T4	T5
00	81	483
01	129	531
10	161	563
11	161	42106

Unit:SC Clock

Note: This value is measured by chip IO pin, the real value will depended on system design

4.3.1.3 Deactivation

下面是新唐硬件Deactivation的时序，T7/T8/T9可以设定



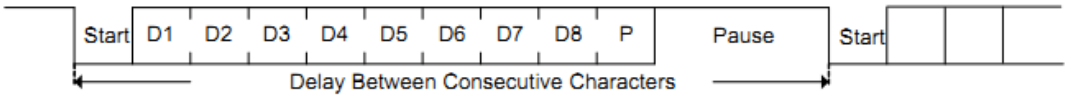
Time	Comment
T7	Deactivation Trigger to SC_RST Low
T8	SMC_RST Low to Stop SC_CLK
T9	Stop SC_CLK to Stop SC_PWR

INIT_SEL	T7	T8	T9
00	97	83	87
01	145	131	135
10	177	163	167
11	177	163	167

Unit: SC Clock

4.3.1.4 数据格式

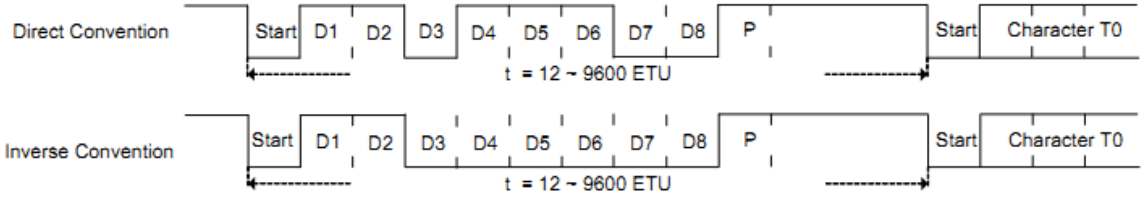
ISO7816每个字符的格式如下



SC数据格式有点像UART，1-bit START + 8-bit数据 + 1-bit 偶校验 + 2-bit 停止位，共12个bit。***每个bit的时间叫ETU***。

ATR返回的第一个字节叫TS，它决定了以后通讯选择Direct Convention(0x3B)还是Inverse Convention(0x3F)

- Direct Convention：’H’表示’1’，’L’表示’0’，LSB 优先
- Inverse Convention：’H’表示’0’，’L’表示’1’，MSB 优先

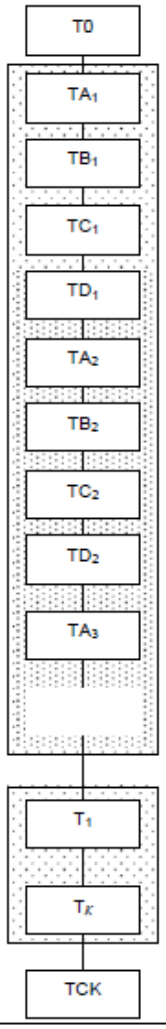


Direct Convention	0_1101_1100_1 (0x3B)
Inverse Convention	0_1100_0000_1 (0x3F)

4.3.1.5 ATR格式

Activation之后IC卡返回的字符串，第一个字节是TS，后面就是ATR，格式如下：ATR的字符依次为：T0 、TA1、 TB1、 TC1、 TD1、 TA2、 TB2、 TC2、 TD2 ….然后是历史字节。Tax/TBx/TCx/TDx不一定存在，要看T0和TDx的值。

Table 6 — Answer-to-Reset



Format byte T0 (mandatory)		Encodes Y_1 and K
Interface bytes (optional)		
TA_1	Global, encodes Fi and Di	
TB_1	Global, deprecated	
TC_1	Global, encodes N	
TD_1	Structural, encodes Y_2 and T	
TA_2	Global, specific mode byte	
TB_2	Global, deprecated	
TC_2	Specific to T=0, see 10.2	
TD_2	Structural, encodes Y_3 and T	
For $i > 2$,		
TD_{i-1}	Structural, encodes Y_i and T	
TA_i	— Specific to T after T from 0 to 14 in TD_{i-1}	
TB_i	— Global after T=15 in TD_{i-1}	
TC_i		
TD_i	Structural, encodes Y_{i+1} and T	
Historical bytes (optional)		
T_1	See ISO/IEC 7816-4	
T_2		
...		
T_K		
Check byte TCK (conditional)		

1) T0 格式如下

Figure 13 — Coding of T0

K：定义历史字节数

Y1：Bit5 决定后面是否有 TA1 字节，Bit6 决定后面是否有 TB1 字节，Bit7 决定后面是否有 TC1 字节，Bit8 决定后面是否有 TD1 字节

2) TA1 决定 Fi 和 Di 的值，如下表，同时决定 CLK 引脚最大时钟频率 fmax。Bit[8:5]决定 Fi 和 fmax，Bit[4:1]决定 Di 的值

TA$_1$ encodes the indicated value of the clock rate conversion integer (*Fi*), the indicated value of the baud rate adjustment integer (*Di*) and the maximum value of the frequency supported by the card (*f* (max.)). The default values are *Fi* = 372, *Di* = 1 and *f* (max.) = 5 MHz. For the use of *Fi* and *Di*, see 7.1, TC$_1$ and TA$_2$ below, 9.2 and 10.2. For the use of *f* (max.), see 5.2.3.

— According to Table 7, bits 8 to 5 encode *Fi* and *f* (max.).

Table 7 — *Fi* and *f* (max.)

Bits 8 to 5	0000	0001	0010	0011	0100	0101	0110	0111
Fi	372	372	558	744	1116	1488	1860	RFU
f (max.) MHz	4	5	6	8	12	16	20	—

Bits 8 to 5	1000	1001	1010	1011	1100	1101	1110	1111
Fi	RFU	512	768	1024	1536	2048	RFU	RFU
f (max.) MHz	—	5	7,5	10	15	20	—	—

— According to Table 8, bits 4 to 1 encode *Di*.

Table 8 — *Di*

Bits 4 to 1	0000	0001	0010	0011	0100	0101	0110	0111
Di	RFU	1	2	4	8	16	32	64

Bits 4 to 1	1000	1001	1010	1011	1100	1101	1110	1111
Di	12	20	RFU	RFU	RFU	RFU	RFU	RFU

3) TB1 已经不用了，一般为 0x00
4) TC1 决定 GT 中 N 的值，GT 是相邻两个字符 leading edge 最小间隔，详细含义后面章节有讲

$$GT = 12\,\text{etu} + R \times \frac{N}{f}$$

5) TD1 格式如下，如果 ATR 中没有 TD1，默认就是 T=0 的卡

Bit 8 (msb)	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1 (lsb)
← Y_{i+1} →				← T →			

Figure 14 — Coding of TD$_i$

T：决定该卡支持的 T 类型，一般 IC 卡有 T=0 和 T=1 两种

Y：含义和 T0 中 Y 的含义相同，决定后面是否有 TA2、TB2、TC2、TD2

6) TA2 这个字节很少见到，据说决定 Fi 和 Di 的值用 TA1 中定义的还是用默认的 Fi=1，Di=1
7) TB2 已经不用了，一般为 0x00
8) TC2 用于 T=0，设定 WI 的值。WT = (WI x 960 x D) etu。WT 是相邻两个字符 leading edge 最大间隔
9) TD2 结构同 TD1，如果支持 T=1，TD2=x1h
10) TA3 用于 T=1，编码卡支持的 block 的最大字节数，有效范围[10h, FEh]
11) TB3 用于 T=1，Bit[8:5]用于定义 BWI，有效范围[0, 4]；Bit[4:1]用于定义 CWI，有效范围[0, 5]

其它的字节就不介绍了很少见到。

下面列举一个 T=0 only 的卡 ATR 的例子： 7d 94 00 00 43 03 01 82 21 55 00 81 4d 50 02

3b	7d	94	00	00	43	03	01	82	21	55	00	81	4d	50	02
TS	T0	TA1	TB1	TC1	历史字节										

T0 = 01111101，表示 TA1、TB1、TC1 存在，没有 TD1，有 13 个历史字节
TA1 = 10010100b，Fi = 1001b，Di=0100b，查上表得到 Fi = 512，Di = 8，fmax=5M。所以可以计算 ETU = 512/8 = 64，每个 bit 64 个 clock
TC1 = 0x00，N=0 表示不需要额外的 GT 时间，GT = 12etu，数据可以连续传输

$$GT = 12\,\text{etu} + R \times \frac{N}{f}$$

TD1 不存在，所以这张卡只支持 T=0

下面列举一个 T=1 only 的卡 ATR 的例子，TD1 存在，并且 T=1，所以这张卡只支持 T=1

Character	Value	Remarks
TS	'3B' or '3F'	Indicates direct or inverse convention
T0	'Ex'	TB1 to TD1 present; x indicates the number of historical bytes present
TB1	'00'	VPP not required
TC1	'00' to 'FF'	Indicates amount of extra guardtime required. Value 'FF' has special meaning (see section 8.3.3.3)
TD1	'81'	TA2, TB2, and TC2 absent; TD2 present; T=1 to be used
TD2	'31'	TA3 and TB3 present; TC3 and TD3 absent; T=1 to be used
TA3	'10' to 'FE'	Returns IFSI, which indicates initial value for information field size for the ICC and IFSC of 16–254 bytes
TB3	m.s. nibble '0' to '4' l.s. nibble '0' to '5'	BWI = 0 to 4 CWI = 0 to 5
TCK	See section 8.3.4	Check character

Table 16: Basic ATR for T=1 Only

4.3.1.6 ETU

ETU计算公式如下

$$1\,\text{etu} = \frac{F}{D} \times \frac{1}{f}$$

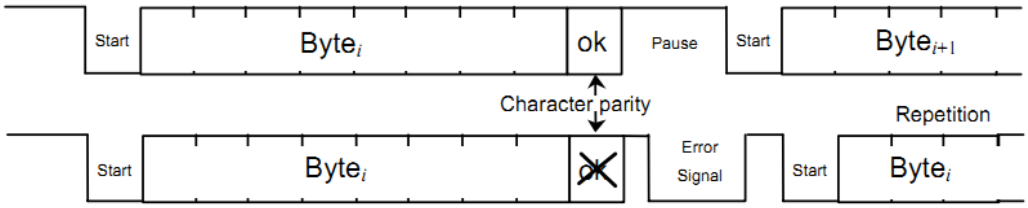
- F: clock rate conversion integer
- D: baud rate adjustment integer
- f：是CLK引脚的频率

默认 etu is 372 clock cycles

Fi和Di的值在TA1中定义，默认Bit[8:5]=1，Bit[4:1]=1

4.3.1.7 出错

如果发生校验错误，接收方会Parity bit之后拉1.5 ETU的low，接收方就会知道，接收方可以重传。



4.3.1.8 时间定义

默认：GT = 12etu、WT = 9600etu、ETU = 372

下面这些时间是EMV2000的定义，只要符合EMV2000的定义，就符合ISO7816-3的定义。大家主要记住下面两点：

- ✧ 字符之间的最小间隔，和字符之间的最大间隔
- ✧ GT 是最小间隔，WT 是最大间隔

记住上面两条，下面的定义就很好理解了

相关缩写定义如下：

- ➢ BWI：block waiting time integer。[0, 4]

- CWI：character waing time integer。[0, 5]
- WI：waiting time integer。由 TC2(8-bit)说明，如果没有 TC2，WI = 10
- BGT：Block guard time。相反方向连续两个字符 leading edge 之间的最小间隔。BGT = 22etu
- CGT：character guard time。相同方向两个字符 leading edge 之间的最小间隔
- GT：Guard time。两个字符 leading edge 之间的最小间隔。默认 12etu

$$GT = 12\,\text{etu} + R \times \frac{N}{f}$$

R/f = etu，所以GT = 12etu + Netu

- N = [0, 254] CGT= GT
- N = 255
  - T=0 CGT = 12etu
  - T=1 CGT = 11etu

- BWT：block waiting time。相反方向连续两个字符之间的最大间隔。BWT = 22etu

$$BWT = 11\,\text{etu} + 2^{BWI} \times 960 \times \frac{Fd}{f}$$

- CWT：character waiting time。相同方向两个字符 leading edge 之间的最大间隔。这个缩写是 ISO7816-3 里面的，EMV2000 里面没有 CWT，只有 WWT

$$CWT = (11 + 2^{CWI})\,\text{etu}$$

- WT：waiting time。两个字符 leading edge 之间的最大间隔。默认 9600etu

$$WT = WI \times 960 \times \frac{Fi}{f}$$

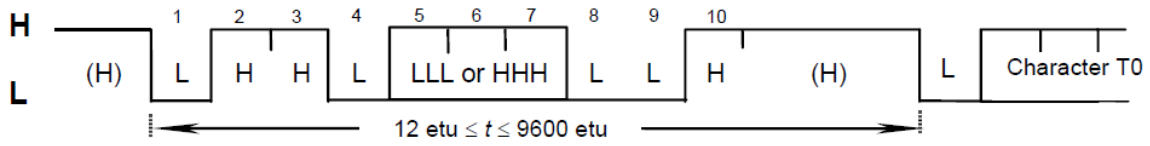


Figure 11 — Initial character TS

每个字符占用时间最小12etu，最大9600etu

通过下图能更好的理解CGT和CWT之间的关系，相同方向字符之间的最小和最大间隔

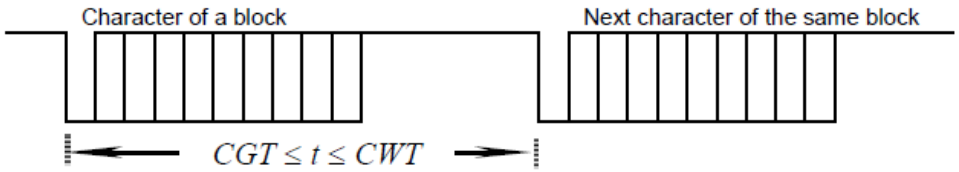
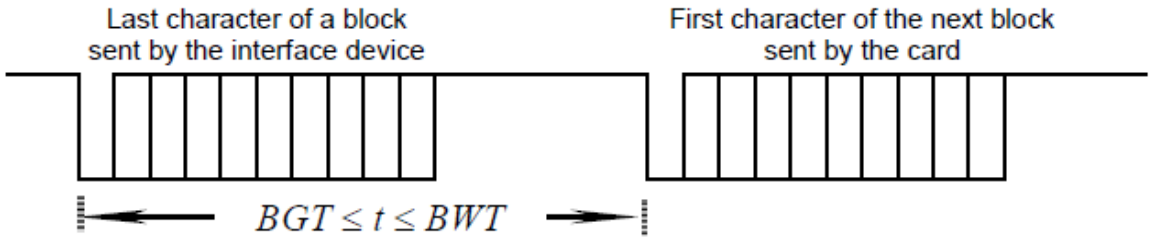


Figure 21 — Character timings within the block

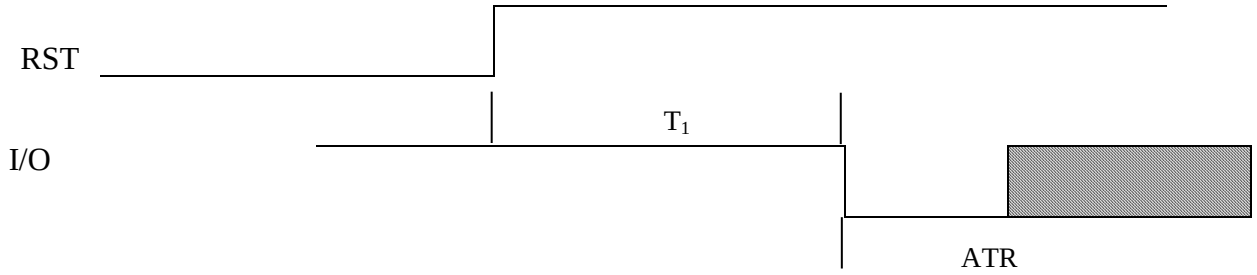
通过下图能更好的理解BGT和BWT之间的关系，相反方向字符之间的最小和最大间隔



- WWT，EMV2000 里面有个时间叫 WWT，它是定义 IC 卡发送字符和前一个字符（无论是接口设备发送还是 IC 卡发送）之间的最大间隔。WWT = (960xDxWI)etu （其实就是 ISO7816-3 里面的 WT 的定义）。接口设备接收完毕最大间隔(WWT + Dx480)etu 必须返回数据。

BWT、CWT、BGT、CGT这些值只用于T=1模式

4.3.1.9 ATR时序



T_1 <= 42000 clock ，RST拉high之后，最多T_1时间内ATR应该返回。

ATR字符：

- 每个字符 12 etu
- 字符 leading edge 之间最大间隔 9600etu
- 19200etu 时间内所有字符接收完毕

- 最多 32 个字节

4.3.1.10 命令

接口设备发给IC卡的命令格式如下：

CLA INS P1 P2	L_c field	Data field	L_e field

含义如下表：

Table 6 — Command APDU contents

Code	Name	Length	Description
CLA	Class	1	Class of instruction
INS	Instruction	1	Instruction code
P1	Parameter 1	1	Instruction parameter 1
P2	Parameter 2	1	Instruction parameter 2
L_c field	Length	variable 1 or 3	Number of bytes present in the data field of the command
Data field	Data	variable = L_c	String of bytes sent in the data field of the command
L_e field	Length	variable ≤ 3	Maximum number of bytes expected in the data field of the response to the command

CLA编码如下，0xD0往后都是私有的命令，8xh和9xh命令结构符合ISO7816但是命令的含义是私有的：

Table 8 — Coding and meaning of CLA

Value	Meaning
'0X'	Structure and coding of command and response according to this part of ISO/IEC 7816 (for coding of 'X', see table 9)
'10' to '7F'	RFU
'8X', '9X'	Structure of command and response according to this part of ISO/IEC 7816. Except for 'X' (for coding, see table 9), the coding and meaning of command and response are proprietary
'AX'	Unless otherwise specified by the application context, structure and coding of command and response according to this part of ISO/IEC 7816 (for coding of 'X', see table 9)
'B0' to 'CF'	Structure of command and response according to this part of ISO/IEC 7816
'D0' to 'FE'	Proprietary structure and coding of command and response
'FF'	Reserved for PTS

INS编码如下：

Table 4.1 — Commands in the alphabetic order

Command name	INS	See
ACTIVATE FILE	'44'	Part 9
APPEND RECORD	'E2'	7.3.7
CHANGE REFERENCE DATA	'24'	7.5.7
CREATE FILE	'E0'	Part 9
DEACTIVATE FILE	'04'	Part 9
DELETE FILE	'E4'	Part 9
DISABLE VERIFICATION REQUIREMENT	'26'	7.5.9
ENABLE VERIFICATION REQUIREMENT	'28'	7.5.8
ENVELOPE	'C2', 'C3'	7.6.2
ERASE BINARY	'0E', '0F'	7.2.7
ERASE RECORD (S)	'0C'	7.3.8
EXTERNAL (/ MUTUAL) AUTHENTICATE	'82'	7.5.4
GENERAL AUTHENTICATE	'86', '87'	7.5.5
GENERATE ASYMMETRIC KEY PAIR	'46'	Part 8
GET CHALLENGE	'84'	7.5.3
GET DATA	'CA', 'CB'	7.4.2
GET RESPONSE	'C0'	7.6.1
INTERNAL AUTHENTICATE	'88'	7.5.2
MANAGE CHANNEL	'70'	7.1.2
MANAGE SECURITY ENVIRONMENT	'22'	7.5.11
PERFORM SCQL OPERATION	'10'	Part 7
PERFORM SECURITY OPERATION	'2A'	Part 8
PERFORM TRANSACTION OPERATION	'12'	Part 7
PERFORM USER OPERATION	'14'	Part 7
PUT DATA	'DA', 'DB'	7.4.3
READ BINARY	'B0', 'B1'	7.2.3
READ RECORD (S)	'B2', 'B3'	7.3.3
RESET RETRY COUNTER	'2C'	7.5.10
SEARCH BINARY	'A0', 'A1'	7.2.6
SEARCH RECORD	'A2'	7.3.7
SELECT	'A4'	7.1.1
TERMINATE CARD USAGE	'FE'	Part 9
TERMINATE DF	'E6'	Part 9
TERMINATE EF	'E8'	Part 9
UPDATE BINARY	'D6', 'D7'	7.2.5
UPDATE RECORD	'DC', 'DD'	7.3.5
VERIFY	'20', '21'	7.5.6
WRITE BINARY	'D0', 'D1'	7.2.4
WRITE RECORD	'D2'	7.3.4

IC返回的数据和状态格式

Data	Data	SW1 SW2

SW1和SW2是状态字节

有了上面这些概念，新唐SC IP就比较容易理解了。

4.3.2 新唐 SC IP 特点

- **ISO 7816-3 兼容**
- **EMV2000 兼容**

- 传输时钟频率(SC_CLK 引脚的时钟)可编程
- 接收缓冲触发级别可编程
- guard time 可编程
- 支持 1 个 24-bit 和 2 个 8-bit 计数器，用于 ATR 和 wait time 过程
- 支持 auto convention
- 时钟停止功能可配置
- Tx/Rx 错误重试次数可配置
- 硬件支持 activation/ warm reset/ deactivation 序列
- 如果卡拔除，支持自动 deactivation
- 支持 UART 模式
  - 每个通道的波特率可编程
  - Rx FIFO 阈值可配置
  - 支持奇，偶和无校验
  - 支持 1 或者 2 停止位

4.3.2.1 定时器工作模式

SC IP带3个定时器，一个是24-bit的，还有2个8-bit的。这些定时器有10种工作模式：

One-shot 模式

- **0: CNTEN使能开始计数，超时时结束，相当于普通的one-shot模式**
- **1: DAT线上检测到第一个 START bit (ether Tx or Rx) 开始计数，超时时结束**
- **2:收到第一个START bit开始计数,超时时结束**
- **3: 只用于硬件Activation、warm reset。RST L->H 开始计数, 收到ATR 或者超时 (Timer 0 only)结束计数**

周期模式

- **4: 跟模式0相似, CNTEN使能开始计数，超时时结束。但是数到0之后重新开始计数，相当于普通周期模式**
- **5: 跟模式1相似, DAT线上检测到第一个 START bit (ether Tx or Rx) 开始计数，超时时结束。但是超时后计数器会重载，再次收到START时会重新开始计数**
- **6: 跟模式2相似, 收到第一个START bit开始计数,超时时结束。但是超时后计数器会重载，再次收到START时会重新开始计数**
- **7: DAT线上检测到START 开始计数(ether Tx or Rx), 每次检测到START会重新加载**
- **15: 由软件Start或者检测到START开始计数，再次检测到START会重新加载.**
- **8: 上数计数器，由软件enable/disable，计时值在disable时会存到TMRDATx寄存器，有点像Timer的连续计数模式**

上面这些计数器就是为了严格计数WT和GT而设计的。另外SC IP有独立的Timer，也免得用系统的。

下面详细介绍一下新唐SC IP的寄存器

4.3.2.2 接收/发送缓冲寄存器SC_DAT

该寄存器用于数据的收发

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
DAT							

读该寄存器从接收FIFO中得到收到的数据，写该寄存器数据将存到发送FIFO中

4.3.2.3 控制寄存器SC_CTL

31	30	29	28	27	26	25	24
Reserved	SYNC	Reserved			CDLV	CDDBSEL	
23	22	21	20	19	18	17	16
TXRTYEN	TXRTY			RXRTYEN	RXRTY		
15	14	13	12	11	10	9	8
NSB	TMRSEL		BGT				
7	6	5	4	3	2	1	0
RXTRGLV		CONSEL		AUTOCEN	TXOFF	RXOFF	SCEN

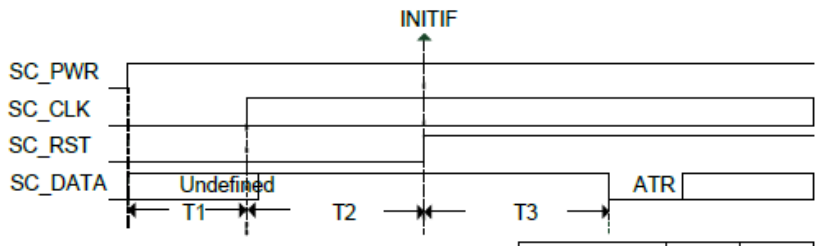
- SCEN：使能 SC engine
- RXOFF：关闭接收
- TXOFF：关闭发送
- AUTOCEN：自动检测 pattern 是 inverse pattern 还是 direct convention，并自动修改 bit[5:4]CONSEL
- CONSEL：选择 Convention
- RXTRGLV：设定接收缓冲中断阈值
- BGT：BGT 用于防止收变送, 或送变收后的传送端马上送出数据。照规范应该是两个方向都要防止。但是认证时, 卡过快回复, 还是要收得下来, 不能当 error 处理。

- TMRSEL：选择使能哪几个定时器，一般填 11b，3 个定时器都使能
- NSB：用于设定 STOP bit 长度
- RXTRY：接收校验错误时，设定允许的重试次数。超过次数之后出错的数据还是会被收下来
- RXTRYEN：使能接收重试功能
- TXTRY：发送发生校验错误时，设定允许的重试次数。
- TXTRYEN：使能发送重试功能
- CDDBSEL：Card Detect 引脚消抖功能选择
- CDLV：卡插入电平。设定卡插入时，CD 引脚是高电平还是低电平
- SYNC：修改 SC_CTL 寄存器时需要查看该 bit，确认上次写入的值已经有效

4.3.2.4 SC_ALTCTL寄存器

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
ACTSTS2	ACTSTS1	ACTSTS0	RXBGTEN	ADACEN	Reserved	INITSEL	
7	6	5	4	3	2	1	0
CNTEN2	CNTEN1	CNTEN0	WARSTEN	ACTEN	DACTEN	RXRST	TXRST

- TXRST：清除发送缓冲
- RXRST：清除接收缓冲
- DACTEN：使能硬件 Deactivation
- ACTEN：使能硬件 Activation
- WARSTEN：使能硬件 warm reset
- CNTEN0：Timer0 开始计数
- CNTEN1：Timer1 开始计数
- CNTEN2：Timer2 开始计数
- INITSEL：选择 Activation/warm-reset/deactivation 的时序。例如：如下图 T1/T2/T3 的时间



Time	Comment
T1	SC_PWR to SC_CLK Start
T2	SC_CLK Start to SC_RST Assert
T3	SC_CLK Start to ATR Appear

INIT_SEL (SC_ ALTCT[9:8])	T1	T2
00	85	489
01	133	537
10	165	569
11	165	42060

Unit: SC Clock

Note : The values are measured by chip I/O pin and the real value will depend on system design

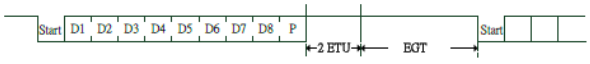
Figure 6.25-4 SC Activation Sequence

- ADACEN：当卡拔掉时自动进行 Deactivation 过程
- RXBGTEN：看要不要忽略接收时的 BGT (卡太快送过来). 理论上每个方向都要有 BGT, 但是认证时, 卡过快回复, 还是要收得下来, 不能当 error 处理, 所以一般不会用
- ACTSTS0：Timer0 Active 标志
- ACTSTS1：Timer1 Active 标志
- ACTSTS2：Timer2 Active 标志

4.3.2.5 SC_EGT寄存器

该寄存器设定的EGT的值如下图所示：相同方向字符之间最小间隔

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
EGT							

Bits	Description	
[31:8]	Reserved	Reserved.
[7:0]	EGT	**Extended Guard Time** This field indicates the extended guard timer value.  **Note:** The counter is ETU base and the real extended guard time is EGT.

4.3.2.6 接收超时寄存器SC_RXTOUT

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							RFTM
7	6	5	4	3	2	1	0
RFTM							

该寄存器类似于UART的RX timeout功能，RX timeout的单位是ETU

4.3.2.7 SC_ETUCTL寄存器

该寄存器用于设定ETU的值，ETURDIV默认值为371

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
CMPEN	Reserved			ETURDIV			
7	6	5	4	3	2	1	0
ETURDIV							

实际ETU为ETURDIV + 1，所以默认ETU=372 SC clock

CMPEN：使能补偿功能，硬件自动n、n-1个SC clock循环，n=ETURDIV

4.3.2.8 中断使能/状态寄存器SC_INTEN和SC_INTSTS

SC_INTEN寄存器用于使能各种中断，SC_INTSTS用来显示各种状态。中断不使能状态寄存器的状态也会改变的

SC_INTEN

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved					ACERRIEN	RXTOIF	INITIEN
7	6	5	4	3	2	1	0
CDIEN	BGTIEN	TMR2IEN	TMR1IEN	TMR0IEN	TERRIEN	TXBEIEN	RDAIEN

SC_INTSTS

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved					ACERRIF	RBTOIF	INITIF
7	6	5	4	3	2	1	0
CDIF	BGTIF	TMR2IF	TMR1IF	TMR0IF	TERRIF	TBEIF	RDAIF

- RDAIEN：接收字节数达到接收阈值中断
- TXBEIEN：发送缓冲空中断
- TERRIEN：发送错误中断，包括接收 break 错误，帧错误，校验错误，接收缓冲溢出（接收 FIFO 已满又收到数据），发送缓冲溢出（发送 FIFO 已满，又写数据到发送 FIFO）
- TMR0IEN：Timer0 中断
- TMR1IEN：Timer1 中断
- TMR2IEN：Timer2 中断
- BGTIEN：发送转接收时，接收数据 START-bit 时间短于 BGT 设定的值
- CDIEN：卡插拔中断
- INITIF：Activation/warm-reset/Deactivation 时，RST L->H 时将发生该中断
- RBTOIF：接收缓冲超时中断。就是接收 FIFO 中有数据，但是字节数没有达到接收阈值，并且间隔 RFTM 没有再收到数据，将发生该中断
- ACERRIEN：接收到的 pattern 值不是 0x3F 也不是 0x3B

4.3.2.9 状态寄存器SC_STATUS

这个寄存器是只读的，用于显示各种状态

31	30	29	28	27	26	25	24
TXACT	TXOVERR	TXRERR	Reserved		TXPOINT		
23	22	21	20	19	18	17	16
RXACT	RXOVERR	RXRERR	Reserved		RXPOINT		
15	14	13	12	11	10	9	8
Reserved		CDPINSTS	CINSERT	CREMOVE	TXFULL	TXEMPTY	TXOV
7	6	5	4	3	2	1	0
Reserved	BEF	FEF	PEF	Reserved	RXFULL	RXEMPTY	RXOV

- RXOV：接收 FIFO 溢出

- RXEMPTY：接收 FIFO 为空
- RXFULL：接收 FIFO 已满
- PEF：接收字符发生校验错误
- FEF：接收 FIFO 没有有效的 STOP-bit
- BEF：接收引脚在 low 的时间超过 start-bit + data-bits + parity + stop-bit 长度
- TXOV：发送 FIFO 溢出
- TXEMPTY：发送 FIFO 为空
- TXFULL：发送 FIFO 满
- CREMOVE：卡拔出
- CINSERT：卡插入
- CDPINSTS：SC_CD 引脚的电平状态
- RXPOINT：接收缓冲指针的值
- RXRERR：发生接收错误，开始重传
- RXOVERR：接收超过重试次数
- RXACT：正在接收
- TXPOINT：发送缓冲指针的值
- TXRERR：发生发送错误，开始重传
- TXOVERR：发送超过重试次数
- TXACT：正在发送

4.3.2.10 PIN控制寄存器SC_PINCTL

该寄存器用于控制SC_DAT、SC_CLK、SC_RST、SC_PWR引脚的状态

31	30	29	28	27	26	25	24
Reserved	SYNC	Reserved					
23	22	21	20	19	18	17	16
Reserved					RSTSTS	PWRSTS	DATSTS
15	14	13	12	11	10	9	8
Reserved				PWRINV	Reserved	SCDOOUT	Reserved
7	6	5	4	3	2	1	0
Reserved	CLKKEEP	Reserved				SCRST	PWREN

- PWREN：根据 PWRINV 的设置驱动 PWR 引脚到高电平还是低电平
- SCRST：驱动 SC_RST 引脚到高电平/低电平
- CLKKEEP：驱动 CLK 引脚输出时钟
- SCDOOUT：驱动 SC_DAT 引脚的电平
- PWRINV：用于设置 SC_PWR 引脚 low 供电还是 high 供电。如果 low 供电，PWREN=1 时，SC_PWR 输出低电平；如果 high 供电，PWREN=1 时，SC_PWR 输出高电平

- DATSTS：用于显示 SC_DAT 引脚的状态
- PWRSTS：用于显示 SC_PWR 引脚的状态
- RSTSTS：用于显示 SC_RST 引脚的状态
- SYNC：修改该寄存器之前要确认前面的值已经生效

4.3.2.11 Timer控制寄存器SC_TMRCTL0/SC_TMRCTL1/SC_TMRCTL2

该寄存器用于选择Timer的工作模式和设定超时时间

SC_TMRCTL0

31	30	29	28	27	26	25	24
Reserved				OPMODE			
23	22	21	20	19	18	17	16
CNT							
15	14	13	12	11	10	9	8
CNT							
7	6	5	4	3	2	1	0
CNT							

SC_TMRCTL1/SC_TMRCTL2

31	30	29	28	27	26	25	24
Reserved				OPMODE			
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
CNT							

- OPMODE：如 4.3.2.1 定时器工作模式所述的几种模式
- CNT：定时器的超时值，OPMODE=8 时该值不起作用

4.3.2.12 UART模式控制寄存器SC_UACTL

该寄存器用于SC接口用作UART时设定校验和数据长度的，停止位长度在SC_CTL寄存器NSB设定

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
OPE	PBOFF	WLS	Reserved				UARTEN

- UARTEN：使能 UART 功能
- WLS：设定数据位长度
- PBOFF：使能校验位
- OPE：设定奇偶校验

4.3.2.13 Timer当前计数值寄存器SC_TMRDAT0/SC_TMRDAT1_2

这些寄存器都是只读的，用于显示定时器当前的计数值

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
CNT0							
15	14	13	12	11	10	9	8
CNT0							
7	6	5	4	3	2	1	0
CNT0							

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
CNT2							
7	6	5	4	3	2	1	0
CNT1							

4.3.3 代码分析

下面的代码使用SC2接口取得ATR并打印ATR的数据，然后发送4个命令给IC卡，并打印Response。

新唐提供SC library ，该代码就是利用SC library实现的。SC library目前不支持LE！=0的命令，如果命令有返回数据，需要再用Get Response命令读取

```
/*SC2中断处理函数*/
void SC2_IRQHandler(void)
{
    // Please don't remove any of the function calls below
    if(SCLIB_CheckCDEvent(2))
        return; // 发生卡插/拔事件，不需要处理其它事件...
    SCLIB_CheckTimeOutEvent(2);/*检查超时事件*/
    SCLIB_CheckTxRxEvent(2);/*检查收/发事件*/
    SCLIB_CheckErrorEvent(2);/*检查出错事件*/

    return;
}
/*系统初始化*/
void SYS_Init(void)
{
    /* 初始化系统时钟 */
    /* 解锁保护寄存器 */
    SYS_UnlockReg();

    /* 使能外部晶振HXT (4~24 MHz) */
    CLK_EnableXtalRC(CLK_PWRCTL_HXTEN_Msk);

    /* 等待外部 12MHz 晶振稳定 */
    CLK_WaitClockReady( CLK_STATUS_HXTSTB_Msk);

    /* 使能PLL到84M*/
    CLK->PLLCTL |= CLK_PLLCTL_PD_Msk;
    CLK->PLLCTL = CLK_PLLCTL_84MHz_HXT;

    /* 等待PLL稳定 */
    CLK_WaitClockReady(CLK_STATUS_PLLSTB_Msk);

    /* HCLK选择PLL做时钟源 */
    CLK_SetHCLK(CLK_CLKSEL0_HCLKSEL_PLL,CLK_CLKDIV0_HCLK(1));

    /* 使能IP 时钟 */
    CLK_EnableModuleClock(UART5_MODULE);
    CLK_EnableModuleClock(SC2_MODULE);
```

```
    /* 选择 IP 时钟源，都选择外部晶振 */
    CLK_SetModuleClock(UART5_MODULE, CLK_CLKSEL1_UARTSEL_HXT, CLK_CLKDIV0_UART(1));
    CLK_SetModuleClock(SC2_MODULE, CLK_CLKSEL3_SC2SEL_HXT, CLK_CLKDIV1_SC2(3));  // 12/3
4M, SC2 clock 为 4MHz

    /* 重新计算SystemCoreClock的值. */
    SystemCoreClockUpdate();

    /*配置多功能引脚，PB10/PB11用作UART5 TX和RX；PA2/PA3/PA4/PA5用作SC_DAT/SC_CLK/SC_PWR/SC_RST*/
    SYS->GPB_MFPH = SYS_GPB_MFPH_PB10MFP_UART5_TXD | SYS_GPB_MFPH_PB11MFP_UART5_RXD ;
    SYS->GPA_MFPL = SYS_GPA_MFPL_PA2MFP_SC2_DAT | SYS_GPA_MFPL_PA3MFP_SC2_CLK |
             SYS_GPA_MFPL_PA4MFP_SC2_PWR | SYS_GPA_MFPL_PA5MFP_SC2_RST;

    /* 重新加锁 */
    SYS_LockReg();
}
/*select df01*/
uint8_t g_cmd1buf[] = {0x00, 0xA4, 0x00, 0x00, 0x02, 0xDF, 0x01};/*该命令返回61 2b*/
uint32_t g_cmd1len = 7;
/*get response*/
uint8_t g_cmd2buf[] = {0x00, 0xC0, 0x00, 0x00, 0x2b};/*返回2b个字符+ 90 00*/
uint32_t g_cmd2len = 5;
/*INIT SAM for purchase*/
uint8_t g_cmd3buf[] = {0x80, 0x70, 0x00, 0x00, 0x24, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
0x00, 0x00, 0x00, 0x00, 0x09,
                        0x20, 0x14, 0x09, 0x16, 0x21, 0x40, 0x18, 0x01, 0x00, 0x11, 0x22,
0x33, 0x44, 0x55, 0x66, 0x77,
                        0x88, 0xba, 0xfe, 0xc4, 0xcf, 0xba, 0xfe, 0xc4, 0xcf/*, 0x08*/};
uint32_t g_cmd3len = 41;
/*get response*/
uint8_t g_cmd4buf[] = {0x00, 0xC0, 0x00, 0x00, 0x08};/*返回8个字符 + 90 00*/
uint32_t g_cmd4len = 5;

uint8_t g_cmdrsp[64];
uint32_t g_cmd1rsplen;

int main(void)
{
    SCLIB_CARD_INFO_T s_info;
    int retval, i;
```

```
    /* 系统初始化 */
    SYS_Init();

    /* 初始化UART 为 115200-8n1用于printf */
    UART_Open(UART5, 115200);

    printf("\nThis sample code reads ATR from smartcard\n");

    // 打开SC接口2. 没有CD pin，PWR pin high raise VCC pin to card
    SC_Open(SC2, SC_PIN_STATE_IGNORE, SC_PIN_STATE_HIGH);
    NVIC_EnableIRQ(SC2_IRQn);/*使能SC2中断*/

    // 等待卡插入，如果没有CD 引脚，该函数返回TRUE
    while(SC_IsCardInserted(SC2) == FALSE);
    // 激活slot 2
    retval = SCLIB_ActivateDelay(2, FALSE, 33);

    if(retval == SCLIB_SUCCESS) {/*成功拿到ATR*/
        SCLIB_GetCardInfo(2, &s_info);
        printf("ATR: ");
        for(i = 0; i < s_info.ATR_Len; i++)
            printf("%02x ", s_info.ATR_Buf[i]);
        printf("\n");

        /*发送select命令到SC2*/
        SCLIB_StartTransmission(2, g_cmd1buf, g_cmd1len, g_cmdrsp, &g_cmd1rsplen);
        printf("get response\n");
         for(i = 0; i < g_cmd1rsplen; i++)
            printf("%02x ", g_cmdrsp[i]);
        printf("\n");
        /*发送Get Response命令*/
        SCLIB_StartTransmission(2, g_cmd2buf, g_cmd2len, g_cmdrsp, &g_cmd1rsplen);
        printf("get response\n");
         for(i = 0; i < g_cmd1rsplen; i++)
            printf("%02x ", g_cmdrsp[i]);
        printf("\n");

        /* 发送INIT SAM for purchase到SC2 */
        SCLIB_StartTransmission(2, g_cmd3buf, g_cmd3len, g_cmdrsp, &g_cmd1rsplen);
        printf("get response\n");
         for(i = 0; i < g_cmd1rsplen; i++)
```

```
            printf("%02x ", g_cmdrsp[i]);
        printf("\n");
        /*发送Get Response命令*/
        SCLIB_StartTransmission(2, g_cmd4buf, g_cmd4len, g_cmdrsp, &g_cmd1rsplen);
        printf("get response\n");
         for(i = 0; i < g_cmd1rsplen; i++)
            printf("%02x ", g_cmdrsp[i]);
        printf("\n");

    } else
        printf("Smartcard activate failed\n");

    while(1);
}
```

5 技巧篇

5.1 UART

5.1.1 收发效率

UART有收/发FIFO，如果要实现UART快速收发就要用到UART FIFO功能。而用到UART FIFO功能就不得不提一下超时功能。

UART 的FIFO，RX的阀值可以设定，如下：

RFITL	INTR_RDA Trigger Level (Bytes)
00	01
01	04
10	08
11	14

可以设定RX FIFO中接收的字节数超过多少个才发生中断。这样可以降低CPU的loading。如果设定RFITL为11b，就是收到14个字节才发生中断。一旦接收的数据不足14个字节，剩下的在RX FIFO中的字节怎么办呢？这就需要设定超时功能了。TOIC的单位为波特率。

UART TIME-OUT Register (UARTx_TMCTL)

Register	Offset	R/W	Description	Reset Value
UART_TMCTL x=0,1	UARTx_BA+0x20	R/W	UART Time-Out Control State Register.	0x0000_01FF

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
DLY							
15	14	13	12	11	10	9	8
Reserved							TOIC
7	6	5	4	3	2	1	0
TOIC							

例如TMCTL = 40，就是RX FIFO中有数据，但是间隔40个波特率时间没有再收到数据，就会发生超时中断。调用下面的函数使能接收FIFO满中断（RDA）和超时中断

UART_ENABLE_INT(UART0, (UART_INTEN_RDAIEN_Msk | UART_INTEN_RXTOIEN_Msk));

中断处理函数如下：

```
/*FIFO +超时 */
static VOID UART0_INT_HANDLE(void)
{
    UINT8 bInChar;

    if(UART0->INTSTS & (UART_INTSTS_RDAINT_Msk|UART_INTSTS_RXTOINT_Msk))//RDA or Timeout
    {
         /* Get all the input characters */
         //while(UART_IS_RX_READY(UART0)) {
         while(UART_GET_RX_EMPTY(UART0)==0) {
             /* Get the character from UART Buffer */
             bInChar = UART_READ(UART0);
         }
    }
}
```

上面的代码注意不能用红色的那行代码，只有RX FIFO中的字节数>= RFITL的设定，UART_IS_RX_READY才会为TRUE。就是说，一旦读出一个字节，它就会为FALSE了。

5.1.2 冲突处理

如果在收发UART数据的时候，妨碍了其它中断的处理。则可以用下面的函数

NVIC_SetPriority(TMR0_IRQn,0);

NVIC_SetPriority(UART0_IRQn,3);

将其它中断的优先级提高。如上，Timer0优先级为0，而UART0优先级为3。 0为最高优先级，3为最低优先级。M0/M4的中断是可以抢占的。如果正在处理UART0的中断，此时发生了Timer0中断，会先去处理Timer0的中断。

5.1.3 ROM 擦除时如何保证 UART 不丢数据

这里主要是指UART收到的数据。因为ROM擦除一个page少则需要3ms多则20ms，如果UART波特率为115200的话，86.8us一个字节，3ms可以收34个字节了。而我们UART一般接收FIFO是16Byte，会溢出。

如果擦除ROM的时候不需要处理任何中断，那就好办了。只要把擦除的代码放到SRAM中执行，然后利用轮询的方法查看是否有UART数据需要接收，就可以解决了。

```
/*FIFO +超时 */
```

5.2 SPI PDMA+FIFO 收发

SPI最快速的收发模式就是FIFO模式了。但是如果碰巧CPU要忙于其他事情的话，那么最理想的模式就是PDMA+FIFO模式了

```
/*SPI PDMA + FIFO收发 */
```

5.3 细说 I2C SI 位

I2C中如果SI没有清0，此时I2C_SCK是被拉低的

```
/*I2C */
```

Revision History

Date	Revision	Description
2015.8.4	1.00	1. Initially issued.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.