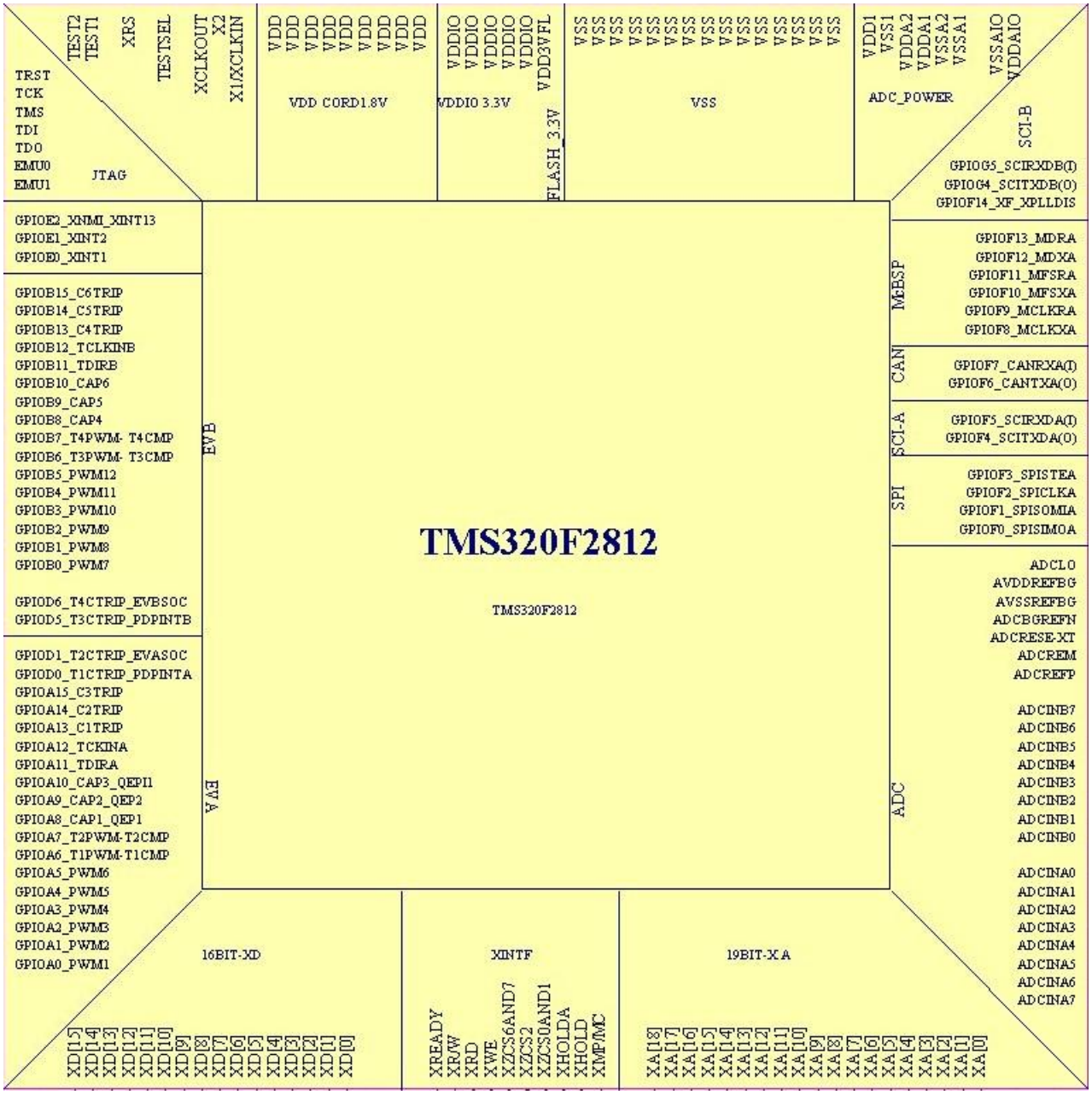
目录

TMS320F2812 引脚详细分析

0

推　荐



XINTF 信号

XA[0]~XA[18] --- 19 位地址总线

XD[0]~XD[15] --- 16 位数据总线

XMP/MC` --- 1 -- 微处理器模式 --- XINCNF7 有效

0 -- 微计算机模式 --- XINCNF7 无效

XHOLD` --- 外部 DMA 保持请求信号。XHOLD 为低电平时请求 XINTF 释放外部总线，并把所有的总线与选通端置为高阻态。当对总线的操作完成且没有即将对 XINTF 进行访问时，XINTF 释放总线。此信号是异步输入并与 XTIMCLK 同步

XHOLDA` --- 外部 DMA 保持确认信号。当 XINTF 响应 XHOLD 的请求时 XHOLDA 呈低电平，所有的 XINTF 总线和选通端呈高阻态。XHOLD 和 XHOLDA 信号同时发出。当 XHOLDA 有效(低)时外部器件只能使用外部总线

XZCS0AND1` --- XINTF 区域 0 和区域 1 的片选，当访 XINTF 区域 0 或 1 时有效(低)

XZCS2` --- XINTF 区域 2 的片选，当访 XINTF 区域 2 时有效(低)

XZCS6AND7` --- XINTF 区域 6 和区域 7 的片选，当访 XINTF 区域 6 或 7 时有效(低)

XWE` --- 写有效。有效时为低电平。写选通信号是每个区域操作的基础，由 XTIMINGX 寄存器的前一周期、当前周期和后一周期的值确定

XRD` --- 读有效。低电平读选通。读选通信号是每个区域操作的基础，由 xTIMINGX 寄存器的前一周期、当前周期和后一周期的值确定。注意：XRD`和 XWE`是互斥信号

XR/W` --- 通常为高电平，当为低电平时表示处于写周期，当为高电平时表示处于读周期

XREADY --- 数据准备输入，被置 1 表示外设已为访问做好准备。XREADY 可被设置为同步或异步输入。在同步模式中，XINTF 接口块在当前周期结束之前的一个 XTIMCLK 时钟周期内要求 XREADY 有效。在异步模式中，在当前的周期结束前 XINTF 接口块以 XTIMCLK 的周期作为周期对 XREADY 采样 3 次。以 XTIMCLK 频率对 XREADY 的采

样与 XCLKOUT 的模式无关

JTAG 和其他信号

X1/XCLKIN --- 振荡器输入 / 内部振荡器输入，该引脚也可以用来提供外部时钟。C28x 能够使用一个外部时钟源，条件是要在该引脚上提供适当的驱动电平，为了适应 1.8V 内核数字电源(VDD)，而不是 3.3V 的 I/O 电源(VLDIO)。可以使用一个嵌位二极管去嵌位时钟信号，以保证它的逻辑高电平不超过 VDD(1.8V 或 1.9V)或者去使用一个 1.8V 的振荡器

X2 --- 振荡器输出

XCLKOUT --- 源于 SYSCLKOUT 的单个时钟输出，用来产生片内和片外等待状态，作为通用时钟源。XCLKOU 丁与 SYSCLKOUT 的频率或者相等，或是它的 1/2，或是 l/4。复位时 XCLKOUT＝SYSCLKOUT/4

TESTSEL --- 测试引脚，为 TI 保留，必须接地

TEST1 --- 测试引脚，为 TI 保留，必须悬空

TEST2 --- 测试引脚，为 TI 保留，必须悬空

TMS --- JTAG 测试模式选择端，有内部上拉功能，在 TCK 的上升沿 TAP 控制器计数一系列的控制输入

TDI --- 带上拉功能的 JTAG 测试数据输入端，在 TCK 的上升沿，TDI 被锁存到选择寄存器、指令寄存器或数据寄存器中

TDO --- JTAG 扫描输出，测试数据输出。在 TCK 的下降沿将选择寄存器的内容从 TDO 移出

TCK --- JTAG 测试时钟，带有内部上拉功能

TRST` --- 有内部上拉的 JTAG 测试复位。当它为高电平时扫描系统控制器件的操作。若信号悬空或为低电平，器件以功能模式操作，测试复位信号被忽略

注意：TRST`上不要用上拉电阻。它内部有上拉部件。在强噪声的环境中需要使习附加上拉电阻，此电阻值根据调试器设计的驱动能力而定。一般取 22K 即能提供足够的保护。因为有了这种应用特性，所以使得调试器和应用目际板都有合适且有效的操作

EMU0 --- 带上拉功能的仿真器 I/O 口引脚 0，当 TGST`为高电平时，此引脚用作中断输入。该中断来自仿真系

统，并通过 JTAG 扫描定义为输入/输出

EMU1 --- 仿真器引脚 1，当 TGST`为高电平时，此引脚输出无效，用作中断输入。该中断来自仿真系统的输入，通过 JTAG 扫描定义为输入/输出

XRS` --- 器件复位(输入)及看门狗复位(输出)。器件复位，XRS 使器件终止运行，PC 指向地址 0x3FFFCO。当 XRS 为高电平时，程序从 PC 所指出的位置开始运行。当看门狗产生复位时,DSP 将该引脚驱动为低电平，在看门向复位期间，低电平将持续 512 个 XCLKIN 周期。该引脚的输出缓冲器是一个带有内部上拉(典型值 100mA)的开漏缓冲器，推荐该引脚应该由一个开漏设备去驱动

ADC 模拟输入信号

ADCINA7 ~ ADCINA0 --- 采样/保持 A 的 8 通道模拟输入。在器件未上电之前 ADC 引脚不会被驱动

ADCINB7 ~ ADCINB0 --- 采样/保持 B 的 8 通道模拟输入。在器件未上电之前 ADC 引脚不会被驱动

ADCREFP --- ADC 参考电压输出（2V）。需要在该引脚上接一个低 ESR(50m~1.5 欧姆)的 10uf 陶瓷旁路电容，另一端接至模拟地

ADCREFM --- ADC 参考电压输出（1V）。需要在该引脚上接一个低 ESR(50m~1.5 欧姆)的 10uf 陶瓷旁路电容，另一端接至模拟地

ADCRESE-XT --- ADC 外部偏置电阻(24.9K)

ADCBGREFN --- 测试引脚，为 TI 保留，必须悬空

AVDDREFBG --- ADC 模拟电源(3.3V)

AVSSREFBG --- ADC 模拟地

ADCLO --- 普通低侧模拟输入

VSS1 --- ADC 数字地

VSSA1、2 --- ADC 模拟地

VDD1 --- ADC 数字电源(1.8V)

VDDA1、2 --- ADC 模拟电源(3.3V)

VDDAIO --- I/O 模拟电源(3.3V)

VSSAIO --- I/O 模拟地

电源信号

VDD --- 1.8V 或 1.9V 核心数字电源

VSS --- 内核和数字 I/O 地

VDDAIO --- I/O 模拟电源(3.3V)

VDDIO --- I/O 数字电源(3.3V)

VSSAIO --- I/O 模拟地

VDD3VL --- flash 核电源(3.3V)，上电后所有时间内都应将该引脚接至 3.3V

GPIO 和外设共用的管脚

EV-A

PWM1--6

T1PWM_T1CMP --- 定时器 1 输出

T2PWM_T2CMP --- 定时器 2 输出

CAP1_QEP1 --- 捕获输入

CAP2_QEP2 --- 捕获输入

CAP3_QEP11 --- 捕获输入

TDIRA --- 计数器方向

TCKINA --- 计数器时钟输入

C1TRIP` --- 比较器 1 输出

C2TRIP` --- 比较器 2 输出

C3TRIP` --- 比较器 3 输出

T1CTRIP`_PDPINTA` --- 定时器 1 比较输出

T2CTRIP`/EVASOC` --- 定时器 2 比较输出或 EV-A 启动外部 AD 转换输出

EV-B

PWM7--12

T3PWM_T3CMP --- 定时器 1 输出

T4PWM_T4CMP --- 定时器 2 输出

CAP4_QEP12 --- 捕获输入

CAP5_QEP4 --- 捕获输入

CAP6_QEP3 --- 捕获输入

TDIRB --- 计数器方向

TCKINB --- 计数器时钟输入

C4TRIP` --- 比较器 4 输出

C5TRIP` --- 比较器 5 输出

C6TRIP` --- 比较器 6 输出

T3CTRIP`_PDPINTB` --- 定时器 3 比较输出

T4CTRIP`/EVBSOC` --- 定时器 4 比较输出或 EV-B 启动外部 AD 转换输出

中断信号

XINT_XBIO` --- XINT1 或 XBIO`核心输入

XINT2_ADCSOC --- XINT2 或开始 AD 转换

XINMI_XINT13 --- XNMI 或 XINT13

SPI

SPISIMOA --- SPI 从动输入，主动输出

SPISOMIA --- SPI 从动输出，主动输入

SPICLKA --- SPI 时钟

SPISTEA --- SPI 从动传送使能

SCI-A，SCI-B

SCITXDA --- SCI-A 发送

SCIRXDA --- SCI-A 接收

SCITXDB --- SCI-B 发送

SCIRXDB --- SCI-B 接收

CAN

CANTXA --- CAN 发送

CANRXA --- CAN 接收

MCBSP

MCLKXA --- 发送时钟

MCLKRA --- 接收时钟

MFSXA --- 发送帧同步信号

MSXRA --- 接收帧同步信号

MDXA --- 发送串行数据

MDRA --- 接收串行数据

XF——CPU 输出

XF_XPLLDIS` --- 引脚有 3 个功能：1、XF----通用输出引脚。2、XPLLDIS -- 复位期间此引脚被采样以检查锁相环 PLL 是否被使能，若该引脚采样为低，PLL 将被禁止。此时，不能使用 HALT 和 STANDBY 模式。3、GPIO --- 通用输入/输出功能。

HELLO 一:如何开始 DSP 的学习

---By eys417

摘录一些，分享下....

以下为各网友学习 DSP 的一些经验

fxw451：大家先大体上看一遍书，把大体的知识了解一下。其次就是看例子了，例子是关键，例子里有你学的所有的东西，这次你再拿出一本书来看，这次是有针对性的看，比如你做的 spi 的，你就直接看 spi 那张，一边看例子一边看书，这样你就可以把一些重要的寄存器给记住了。对于初学者来说，一直好奇的就是 ccs 的使用，拿我第一次使用 ccs 来说，当我把 ccs 和板子连载一起时，我相当高兴，成功感油然升起，接下来就是用 ccs 里的看自带的例子了，看完后你就会发现，这些是什么东东哦，什么都不会，这就对了，你要是看一开始看会了你就是神仙了，dsp 不像单片机那么容易上手，所以你要花费点功夫吃透它，好东西不是那么容易就可以搞定的。到了自己编程的时候了，这个时候不要要求自己能编一个什么样的程序，你要仿着例子里的东西全部搞定就可以了，这就是你编程的第一步，当然也是成功的一步，在这成功下，我相信你的积极性肯定被调到起来，对 dsp 越来越热爱了。

wsppike：DSP 相比于单片机，它的很多架构都是一个全新的概念，所以我们得花时间去熟悉它。

suary：1）把存储器映射结构搞清楚----说的具体点就是 dsp 内到底有那些存储器（ram，rom，flash，etc），这些存储器到底是如何分配的，这个可以参考相关的.cmd 文件的写法，它定义了存储器映射和输入输出段的位置

2）编译器的堆栈操作---有关这点我还是没有具体弄清楚，就是中断或是子程序调用时，系统自己的堆栈操作。2407 有一个 8 级硬件堆栈，而 2812 没有，这个区别比较大，所以在编一边针对堆栈操作的程序（eg. rtos)时就要特别注意了。

3)中断系统----每个 mcu 的中断系统搞清楚了，会给编程带来很大的便利，所以一定要对所用的 mcu 的中断过程了解的清清楚楚。

4）数据结构---设计好的，适合的数据结构会使自己的程序编写变得结构清楚而且"容易"。

dsp31:我的经验是：DSP 不管是软件和硬件开发一定要多思考，多比较。软件人员一定要会调试和定位硬件电路的问题点。硬件人员一定要考虑接口设计，电平转换，电源稳定性，还有 ESD 防护等等。

cysmwander:首先得对 C 熟悉，但会 C 语言并不能说明你会编程，编程有结构设计的问题，C 语言只是工具，看不懂的就好好去想，想通了就发现自己又上了一个台阶；在学习 DSP 之前，我觉得应该需要去明白 DSP 能干什么，所以你一定要先看看 DATASHEET，看看 DSP 的外设和资源，看看你能做啥。
开发环境建立完了你要熟悉开发流程，就是说你有一个 IDEA，怎么把这个 IDEA 在目标板上实现，先做什么后做什么，这个一定要想清楚。

这些都准备好了，你就可以开始干了，千万别犹豫，古人说：临渊慕鱼，不如退而结网。千万别怕。一定要立即动手，毫无顾忌，不要想着，这个东西这么复杂，我从哪里动手阿，万事开头难，你就从你认为对的开始做，做错了重头来。我和我们公司新来的同事都说：多动手，烧掉几块板子和几个 MCU 都是小事，关键是你要动手。

我不建议大家直接拿例程来做试验，因为那样你 MCU 的结构没有把握，你把例程跑得再好，那也不是你的东西，一定要自己写，例程只能作为参照，一定要一个字母一个字母地去写程序。

yangyansky：1、看书，这是必要的，也是基础。第一遍，不用太详细，不过要让自己大概知道书里都有些什么内容。

2、重点了解 2812 的片内资源的分配情况，尤其是存储器的映射，知道了这个你就明白了你写的哪部分内容是具体存到哪里去了。掌握 bootload 的工作原理，知道程序是在哪里，怎么启动的运行的。至于 spi、sci 这些外设，我觉得等到具体运用的时候再看不迟。结合具体的运用还比较容易理解。

3、看例程,在自己编写自己需要功能的代码时要看看已有的例程，看看例程里的各种寄存器是怎么配置的，配置时又是怎么实现的，根据看懂的编写自己想要的，然后跑跑自己的程序看和已有的例程有什么异同，是不是自己想要的。当然，调试环节是最难的一个环节，这个过程中会出现很多意想不到的问题，只能慢慢摸索慢慢前进了。

4、重点理解中断、定时器、系统时钟的工作过程，这对程序里的时间分配问题很重要。

5、我觉得写程序要规范，这会简便很多操作，使编译得以优化，比如相关的文件定义要放在相应的头文件里，全局变量等的定义最好放在 GlobalVariableDefs.c 里，相应的中断程序要放在 DefaultIsr.c 里，等等。"

最后是自己的一些感悟：开发板，仿真器是必要的，它能带给我们更加直观的感受，有空去 TI 官网转转，里面东西还是挺多的，另外坚持不懈的等待 TI--MM 拒绝我的芯片申请。。囧……资料书推荐清华的那个《tms320c28x 系列 dsp 的 cpu 与外设》可当工具书查，还就是一本关于例程详解的，两本配合看。另外一个好的程序模板也是很关键的，TI 官网上 2812 例程，结构严谨并且最大的发挥了 CCS 的功能，建议先搞懂后再依据此创建自己的模板程序.还有就是芯片的 DATASHEET，由于大多数都是英文的看起来比较痛苦，但确都是必须的....想我，四级都还没过来着。关键还是耐心...最后友情提点下谷歌金山词霸。。。我汗...～

HELLO 二:完整工程的构成

这个系列为响应 HELLODSP 的 2812 学习活动的个人笔记，HELLODSP 版权所有。转载请注明

---By eys417

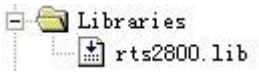
首先那些最基本的仿真驱动安装，CCS 配置等我就不在此累赘了...

一个完整工程文件的构成

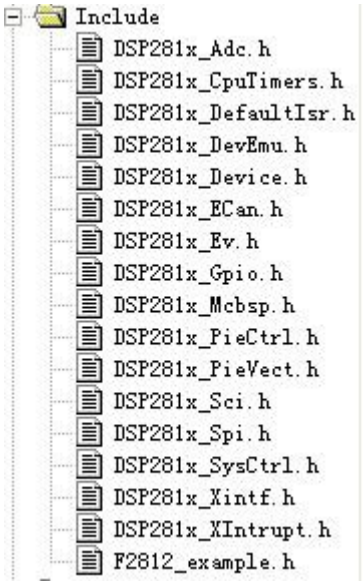
总的说一个完整的工程需要由库文件（.lib），头文件（.h），源文件（.c）和 CMD 文件(.CMD)组成，缺一不可。至于各文件内容将在以后中详细说明

2812 的库文件--文件夹地址 C:\CCStudio_v3.3\C2000\cgtools\lib\rts2800.lib

rts2800_ml.lib --- 大存储器模式

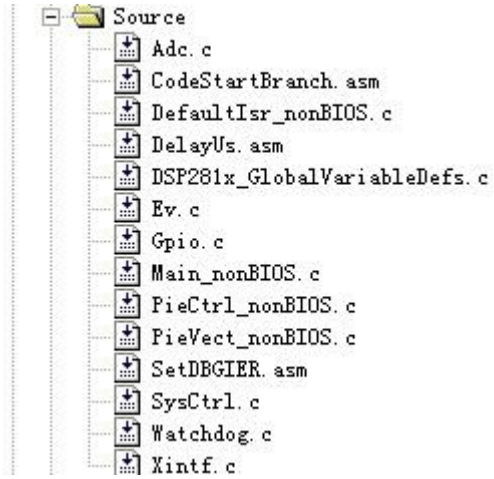


头文件



头文件的作用是定义了 2812 内部寄存器的数据结构。头文件一般情况下不需要修改，如果你需要定义一些在整个工程内都具有作用域的全局变量的时候，可以在头文件中定义这些变量，具体的方法我们以后在例程或项目实践中应该会有介绍。

C 文件(主函数)



ADC.C ——外设 AD 的初始化函数，与外设 AD 相关

CodeStartBranch.asm——引导过程中屏蔽看门狗定时器 。

DSP28_CpuTimers.C——CPU 定时器的初始化和配置函数，与 CPU 的定时器相关

DefaultIsr_nonBIOS.C——包含了 2812 所有的中断函数，写中断时，只要将程序写在对应的函数内就可以，大大保证了中断的成功率。

DelayUs.asm——延时微秒

DSP28x_GlobalVariableDefs.C——全局变量的定义，定义了 2812 的寄存器，中断向量表等内容。

Ev.C——外设 EV 的初始化函数，与外设 EV 相关。

Gpio.C——GPIO 的初始化函数，只和 GPIO 相关。

Main_nonBIOS.c——主函数

PieCtrl_nonBIOS.C——PIE 初始化函数，和中断相关，很重要。

PieVect_nonBIOS.C——PIE 中断向量表定义以及初始化，很重要。

SetDBGIER.asm——real time 仿真

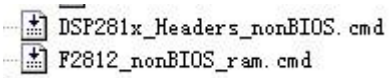
SysCtrl.C——系统初始化，主要对开门狗，时钟等模块进行初始化，以保证 2812 正常工作，非常重要。

Watchdog.C——看门狗初始化

Xintf.C——外部接口的初始化函数。

通过上面的分析我们可以看到几个文件非常重要，因此大家每次新建工程的时候，就把这些未编辑过的文件复制过来。其他的外设相关的文件，您这个工程中涉及到哪个外设，您就把这个外设相关的源文件复制过来，一起加入工程。

CMD 文件



以.CMD 为扩展名的文件，这个文件的作用是用来分配存储空间的。由于 DSP 编译器的编译结果是未定位的，DSP 也没有操作系统来定位执行代码，DSP 系统的配置需求也不尽相同，因此我们根据实际的需求，自己定义代码的存储位置。打个通俗的比喻，就是我们有一个仓库，现在需要把货物存放到仓库里面去，为了便于日后取用货物，我们将货物分门别类，然后把它们存放到指定的位置去。把哪些货物放到哪个位置的规则，就是我们的 CMD 文件的内容。

CMD 文件又分成两种。一种是分配 RAM 空间的--微计算机模式(仿真模式)---**XMP/MC`=0**，用来将程序

load 到 RAM 内进行调试，因为我们大部分时间都是在调试程序，所以多用这类 CMD，另一种是分配 FLASH 空间-----

微处理器模式---**XMP/MC`=1**，当程序调试完毕后，需要将其烧写到 FLASH 内部进行固化，这个时侯我们

就需要使用这类 CMD 文件了。

其中 DSP281x_Headers_nonBIOS.cmd---用于分配编译产生的各个段至存储器
F2812_EzDSP_RAM_lnk.cmd ---用于将 281x 的外设寄存器结构产生的数据段映射至对应的存储器空间
以上 2 个 CMD 文件均为仿真模式下无 BIOS 的 CMD 文件
CMD 文件内容将在以后详细解释

第二课课后

1. 什么是 GEL 文件？GEL 文件的作用是什么呢？

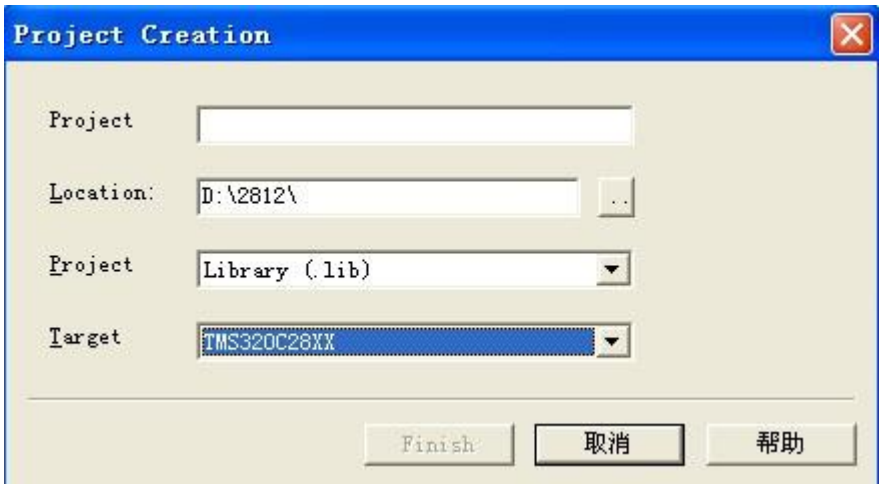
答：GEL 是通用扩展语言（General Extension Language）是一种解释语言，类似于 C 语言。GEL 函数可以用来扩展 CC 的功能，方便用户调试程序。CCS 提供丰富的内嵌 GEL 函数，用户也可以定义自己的 GEL 函数。在处理器属性的 GEL 文件窗内为每个处理器选择用户的 GEL 文件（扩展名为 .gel）

GEL 文件用于初始化 DSP。GEL 在 CCS 下有一个菜单，可以根据 DSP 的对象不同，可以用 gel 来调用一些菜单命令，对 DSP 的存储器进行配置，设置不同的初始化程序。

2. Lib 文件内部究竟是什么内容，我们自己能编辑 LIB 文件吗

答：.lib 是库文件，rts.lib 是 TI 提供的运行时支持库，如果是 C 代码写的源程序，必须要包含该库。该库由 TI 公司做好了，放在 CCS 的\cgtools\lib 中，源代码 TI 网站可以下载。但是我们无法查看.编辑 TI 所提供的库文件内容

我们也可以添加自己的库，可在 newproject 中新建



HELLO 三:CCS 的操作

1

推　荐

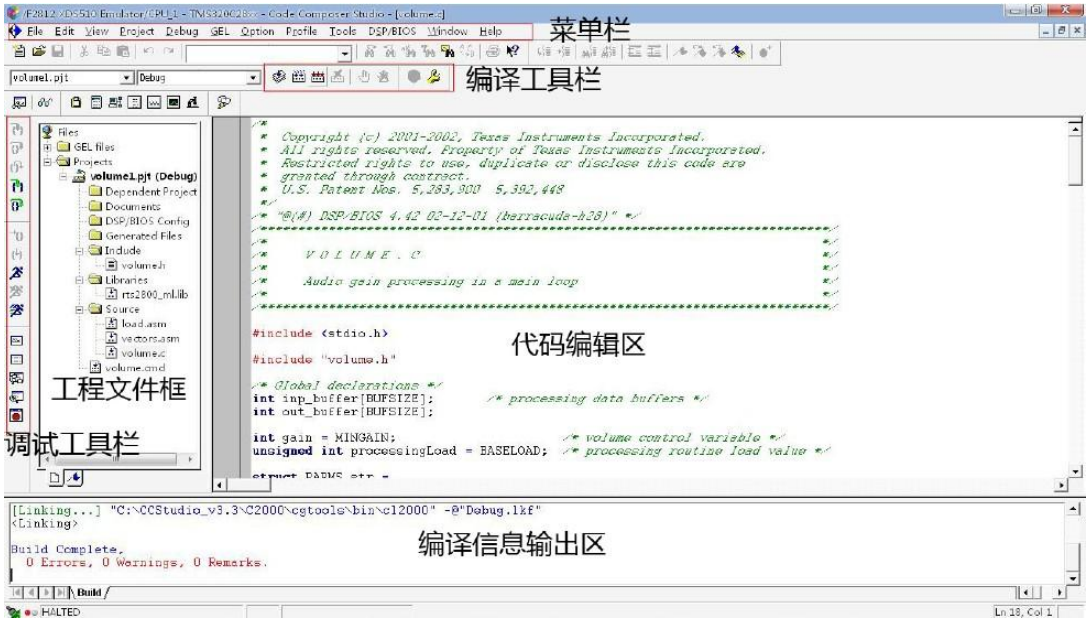
这个系列为响应 HELLODSP 的 2812 学习活动的个人笔记，HELLODSP 版权所有。转载请注明

---By eys417

工欲善其事，必先利其器

中国有句古话叫“磨刀不误砍柴工”，如果您将 CCS 这把常用的利刃磨锋利了，那么我相信为您的 DSP 开发过程节省不少的工夫。

1. CCS 的布局和结构



菜单栏——和 CCS 所有功能相关的菜单都在这里面。

编译工具栏——编译程序时常用的一些工具。

调试工具栏——调试程序时常用的一些工具。

工程文件框——打开的工程所有文件会按类别放在这里

代码编辑区——代码都是在这里编辑完成的了，最主要的工作区域。

编译信息输出区——编译时产生的信息会在这个区域内输出

值得一提的是 CC3.3 和 CCS2.2 工具栏的区别，CCS2.2 工具栏的图如下面所示，

我们将其和 CCS3.3 的工具栏比

对之后发现 CCS.2 中的探针工具已经不在 CCS3.3 的工具栏中了，这是因为 CCS3.3 中的断点就包含了探针功能

2. 开始调试程序

在编译完成之后，要来下载程序并进行功能调试。“File”,”Load Program”，在工程文件夹下面的 Debug 文件夹下，选中**.out 文件，点击打开，便开始下载程序了。将**.out 文件下载到目标板上 2812 的 RAM 中。

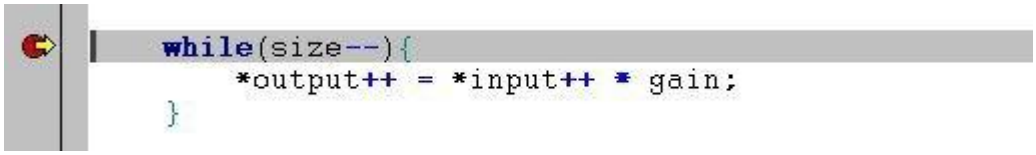
注意，这里是调试，所以将程序下载到 RAM。等到最后您要固化程序的时候，就得下载到 FLASH 了，因为断电之后，RAM 里面所有的数据都会消失。

调试程序常用的一些按钮：

Run 和 **Animate** 的区别，**Run** 是如果遇到断点的话它就停下来了。而 **Animate** 就算遇到断点时先停止 **DSP** 内核，刷新窗口，然后接着继续启动运行，常用来连续刷新变量窗口和生成 **graph** 图形等

1） 如何添加断点

我们调试代码时有时候想看看某一行或者某几行代码是否有执行，或者想看看执行前后变量的一些变化，那么我们就需要在这行或者这几行代码前加上断点了。加上断点的方法很简单，只要在该行代码前双击就行。双击之后，这行代码前面会出现一个红色圆块。另外一种添加断点的方法，就是在刚才的编译工具栏上，点一下那个小手图形的按钮，前提是你要把光标移动到想要设置断点的哪一行上。这时运行 **Run** 按钮，程序就会在断点处停下，黄色的小箭头又出现了。



那如何取消断点呢，在刚刚设置断点的那行再双击一下，代表断点的红色标记就消失了，断点也就被取消了。如果想要清除文件内的所有断点，那么我们可以按一下刚才小手按钮旁边的那个打了叉叉的小手按钮 **"Debug：Remove all breakpoints"**。

（2） 单步调试

让我们来了解一下 **CCS** 给我们提供的调试工具吧。调试工具栏上分两类，一类是用于在源代码中调试的，另一类是用于在汇编代码中调试的。

 **---Source-single step** 源代码单步调试了，就是按一下，走一步的模式。

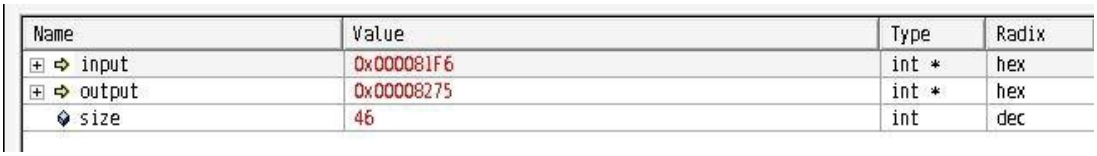
 **---Source-step over** 这个按钮是指在单步执行时，如果在函数内遇到子函数，那么不会进入子函数内单步执行，而是将子函数整个执行完再停止，也就是把子函数整个作为一步。

 **---Source-step out** 当单步执行到子函数内时，用 **step out** 就可以执行完子函数余下部分，并返回到上一层函数。

用于汇编调试的两个工具 **Assembly-single step** 和 **Assembly-step over** 含义和上面源文件调试的两个类似，就不再重复叙述了。

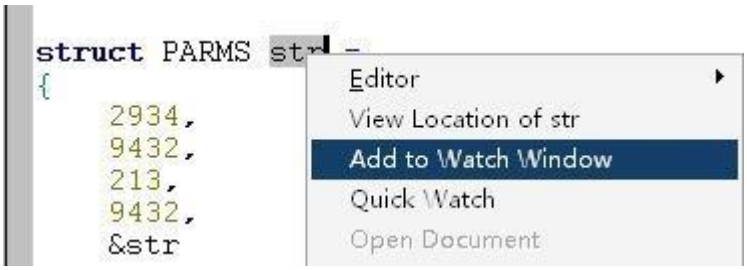
（3） 使用 watch window

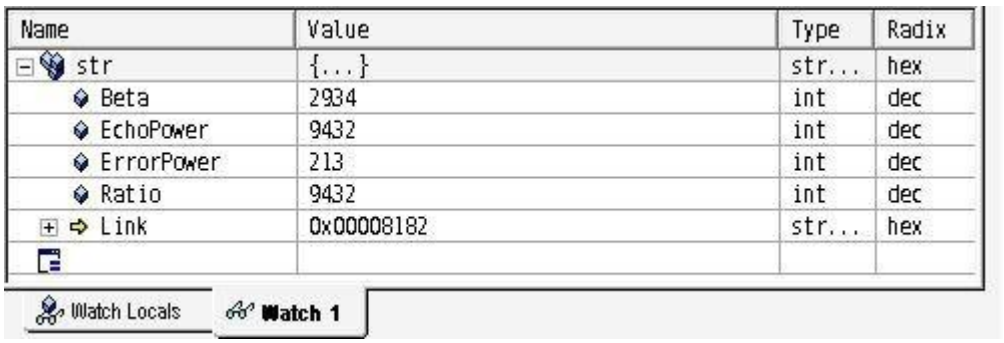
Watch window 的作用是来观察程序运行过程中的各个变量的值。调用 **watch window** 的方法是点击菜单栏的"View ","watch window"，这时 **watch window** 就会显示在 **CCS** 下方的信息区域，如下图所示：



Name	Value	Type	Radix
input	0x000081F6	int *	hex
output	0x00008275	int *	hex
size	46	int	dec

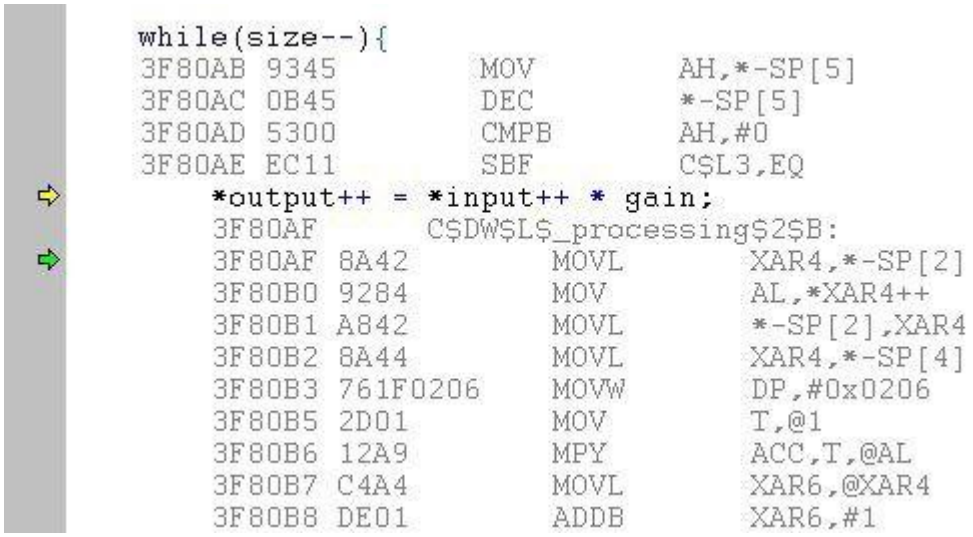
如果想观察某个特定的变量，在代码中选中这个变量，然后右键**"Add to watch window"**





（4）其他一些

我们在调试程序的时候经常想让程序从 **Main** 函数开使运行，点击**"Debug"--"Go main"**。 既能看到源文件中代码的执行情况，又能看到汇编指令的执行情况**----"View","Mixed Source/Asm"**



我们看到，每一行源代码下面就会有相应的汇编代码，黄色的指针指示源代码，绿色的指针指示汇编代码。如果要取消源代码和汇编代码在一个文件内的话，重复刚才的操作就可以了。

3. 统计代码运行时间

在 **CCS3.3** 中如何统计代码的运行时间，首先，将代码的阅览模式设置成前面的源码和汇编同时显示的模式。

点击**"Profile"--"Clock"--"Enable"**，来使能 **CLOCK** 功能。接下来点击**"Profile"--"Clock"--"View"**，在 **CCS** 最下面会出现一个类似于秒表的工具，旁边显示数字**"0"**。

我们关注 **main** 函数这一块，在代码**"int *input = &inp_buffer[0];"**前面设置断点，然后点击 **Run**，这样程序就会停在这一行代码前面，而且黄色指针指示源码，绿色指针指向汇编。

```
void main()
{
3F8092      main:
3F8092 FE04         ADDB        SP,#4
    int *input = &inp_buffer[0];
    3F8093 8F0081C0     MOVL        XAR4,#0x0081C0
    3F8095 A842         MOVL        *-SP[2],XAR4
    int *output = &out_buffer[0];
    3F8096 8F008240     MOVL        XAR4,#0x008240
    3F8098 A844         MOVL        *-SP[4],XAR4
```

这时，底下的 **CLOCK** 工具开始显示的是 **753**，当然不同的环境显示的数字应该是不一样的。这就是从开始执行到这一语句所花的时间了，那 **753** 的单位是**"CPU Cycles"**，**CPU** 的时钟周期。

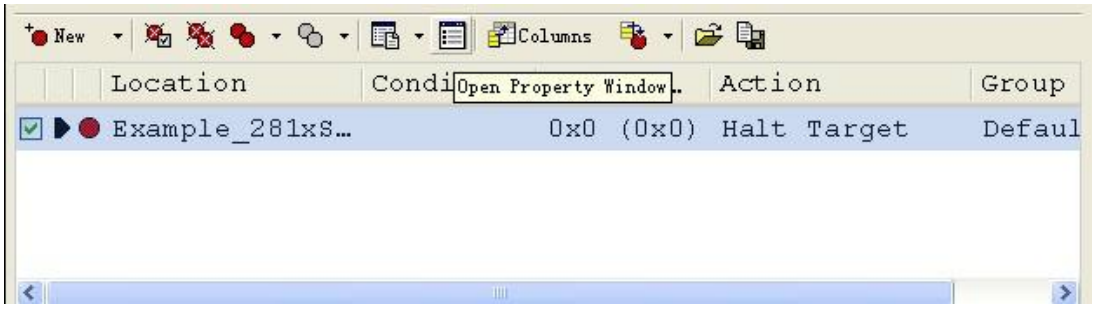
统计汇编指令的执行时间，点击一下 **Assembly-single step**。汇编指令下移一行，**CLOCK** 工具显示 **754**，也就是刚才这句代码执行了 **1** 个 **CC**。统计执行一段代码所花的时间，在需要统计的那段代码开始和结束的地方分别设置断点，如下图所示。

```
void main()
{
3F8092      main:
3F8092 FE04         ADDB        SP,#4
    int *input = &inp_buffer[0];
    3F8093 8F0081C0     MOVL        XAR4,#0x0081C0
    3F8095 A842         MOVL        *-SP[2],XAR4
    int *output = &out_buffer[0];
    3F8096 8F008240     MOVL        XAR4,#0x008240
    3F8098 A844         MOVL        *-SP[4],XAR4

#if defined(_TMS320C28X)
    Disable_WD();
    3F8099 767F8089     LCR         Disable_WD
#endif

    puts("hello dsp !\n");
    3F809B 8F008488     MOVL        XAR4,#0x008488
    3F809D 767F8185     LCR         puts
```

将两个地方的 **CLOCK** 工具显示的值相减就能得到这一段代码的执行时间了。

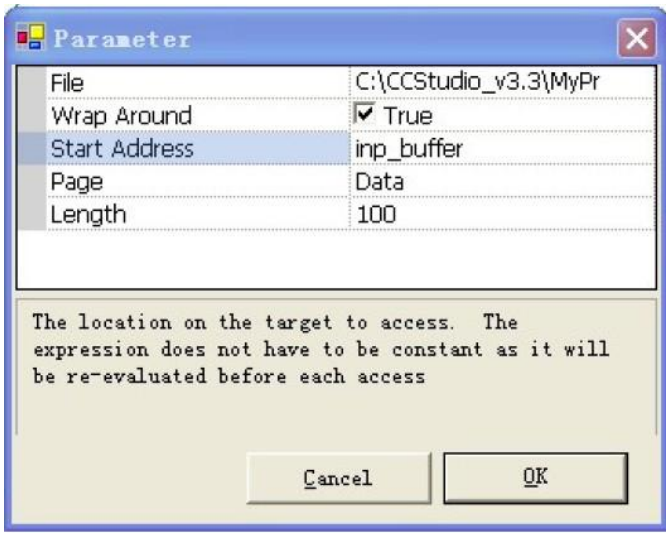
第三课课后：如何使用 CCS3.3 来显示图表？

CCS3.3 的探针功能使用：首先设置断点，点击 **breakpoint manager** 按钮**(**中间有个红色圆**)**，出现窗口



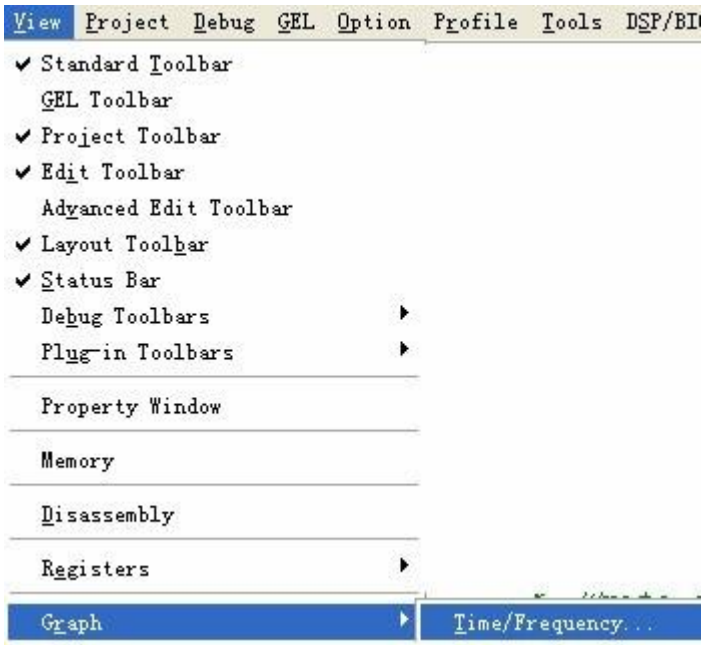
选择**"Action"**下的**"read from File"**

出现并设置以下窗口

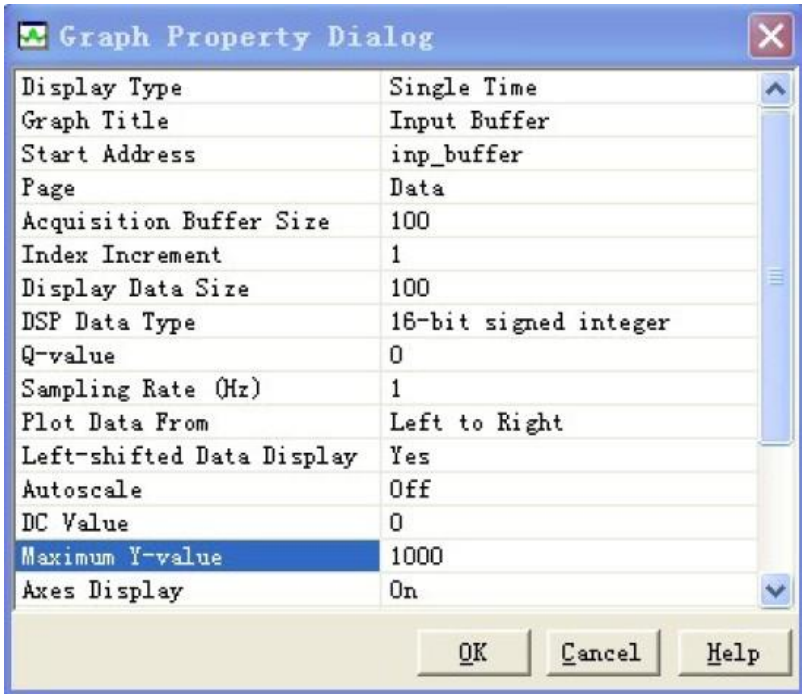


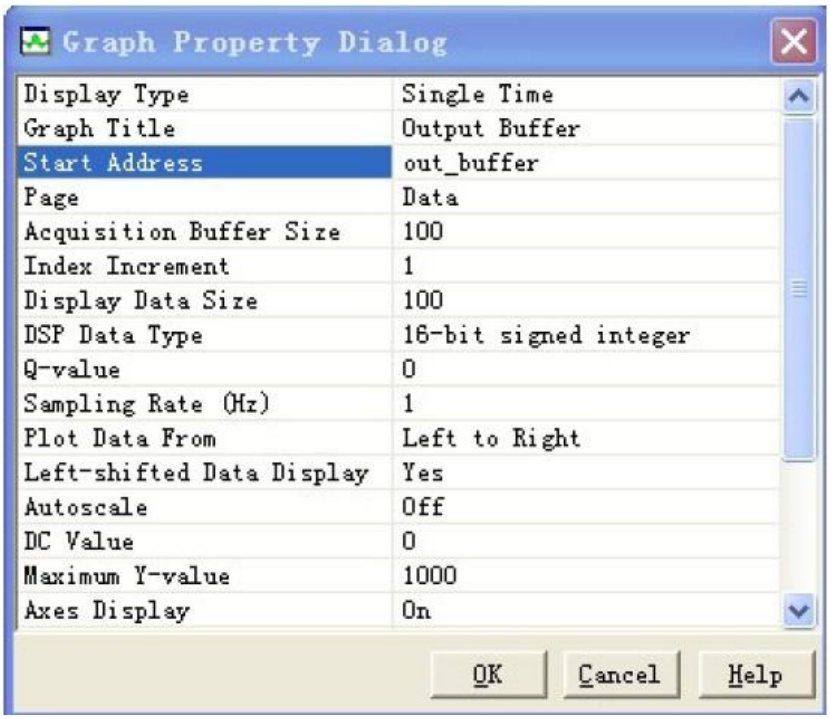
同时我们还可看到 

。这个是用来控制数据输入的停止和开始，便于观测正弦图形的输出然后就是选择图形菜单观察了

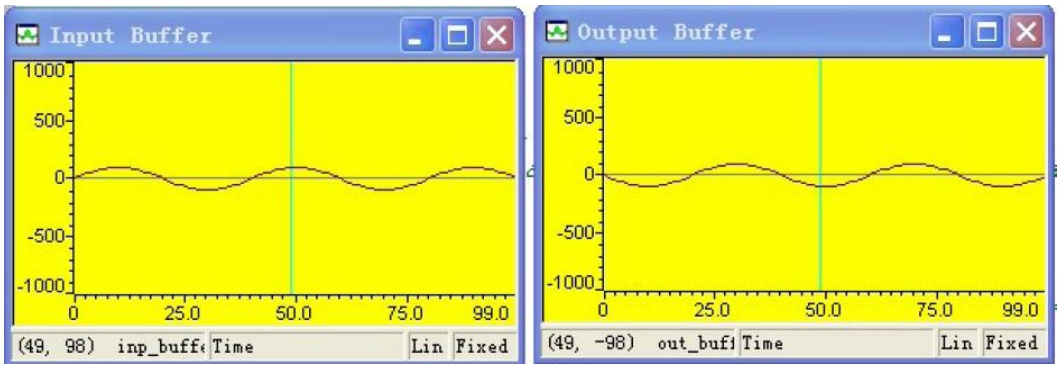


设置输入输出地址等参数

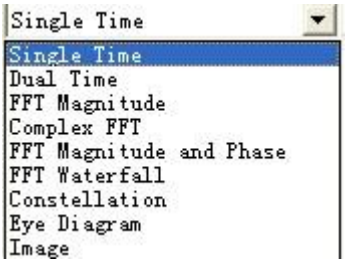




点击**"Animate"**开始仿真，就能看到我们想要的图像了



关于 **Graph** 的一些：



Single Time --- 单时域曲线 --- 显示 幅度-时间 曲线

Dual Time --- 双时域曲线 --- 在显示两条信号的 幅度-时间 曲线

FFT Mangnitude --- **FFT** 幅度 --- 进行 **FFT** 变换，显示 幅度-频率 曲线

Complex FFT --- 复数 **FFT** --- 对复数的实部和虚部分别进行 **FFT** 变换，显示两条信号的 幅度--频率 曲线

FFT Mangnitude and Phase --- **FFT** 幅度和相位 --- 在显示 幅度-频率 曲线和 相位-频率 曲线

FFT Waterfall --- **FFT** 多帧显示 --- 对数据(实数)进行 **FFT** 变换，其 幅度-频率 曲线构成一帧。这些帧按时间顺序构成 **FFT** 多帧显示图

Constellation --- 星座图 --- 显示信号的相位分布

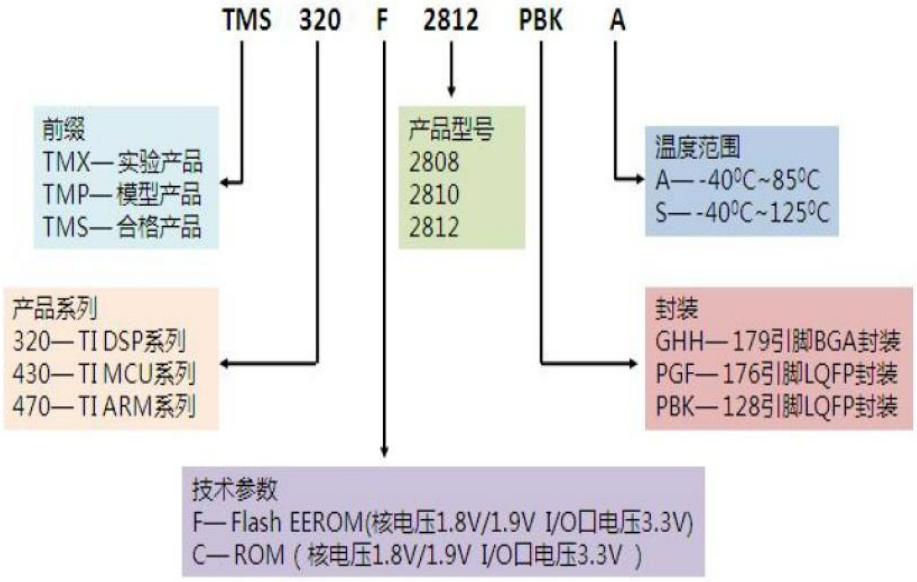
Eye Diagram --- 眼图 --- 显示信号间的干扰情况

Image --- 图像显示 --- 显示 **YUV** 或 **RGB** 图像

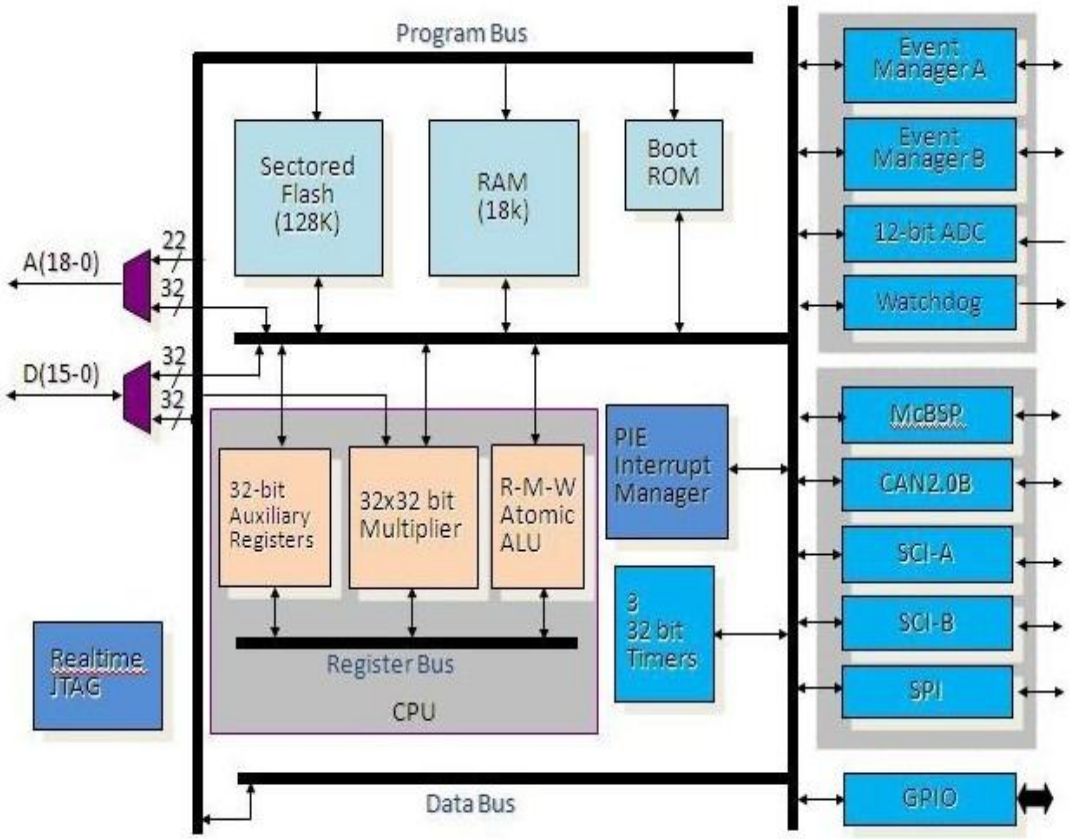
HELLO 四(一)：2812 片内资源

---By eys417

首先解释下 TI 的 DSP 型号中各个字母所表示的含义



TMS320F2812 是 32 位的定点 DSP，它既具有数字信号的处理能力，又具有强大的事件管理能力和嵌入式控制功能，特别适合用于需要大批量数据处理的测控领域，例如自动化控制、电力电子技术、智能化仪表、电机伺服控制。下面是 F2812 的内部资源框图。



2812 采用了高性能的静态 CMOS 技术，时钟频率可达 150MHZ（6.67ns），其核心电压为 1.8V/1.9V，I/O 口电压 3.3V，Flash 编程电压也为 3.3V，所以我们在设计 2812 电源部分的时候，需要将常用的 5V 电压转换成 1.8V 和 3.3V 的 电压之后，才能供给 2812。

F2812 的片内资源

2812 有 3 个 32 位的 CPU 定时器，支持动态的改变锁相环的频率，有片内振荡器和看门狗定时器模块。2812 具有 3 个外部中断，但是 2812 具有外部中断的扩展模块（PIE），它可支持 96 个外部中断，不过当前仅仅使用了 45 个外部中断，其他为保留。具有 128 位的密钥，用于保护 FLASH、OTP 和 L0、L1 中的内容不被盗读。

F2812 的片内外设

1. 2 个事件管理器 EVA、EVB
2. 2 个串行通信接口 SCI，标准的 UART(SCIA SCIB)。
3. 1 个串行外围接口 SPI。
4. 改进的 CAN 通信 ECAN。
5. 多通道缓冲串行接口 McBSP。
6. 12 位的 ADC，一共有 16 个通道，实现 AD 转换的功能
7. 最多有 56 个可独立编程的，多功能复用的 GPIO 引脚。
8. XINTF 外部扩展接口--异步，非复用的总线结构--用于扩展并口外设

关于地址总线和数据总线

地址总线，这类总线的作用就是来传送存储单元的地址的。

1. PAB （Program Address Bus）程序地址总线，它是一个 22 位的总线，用于传送程序空间的读写地址。程序运行的时候，假如执行到了某一个指令，那么需要去找到这段代码的地址，就是用 PAB 来传送。

2. DRAB（Data-Read Address Bus）数据读地址总线，它是个 32 位的总线，用于传送数据空

间的读地址。假如要读取数据空间某一个单元的内容，那么这个单元的地址就是通过 DRAB 来传送。

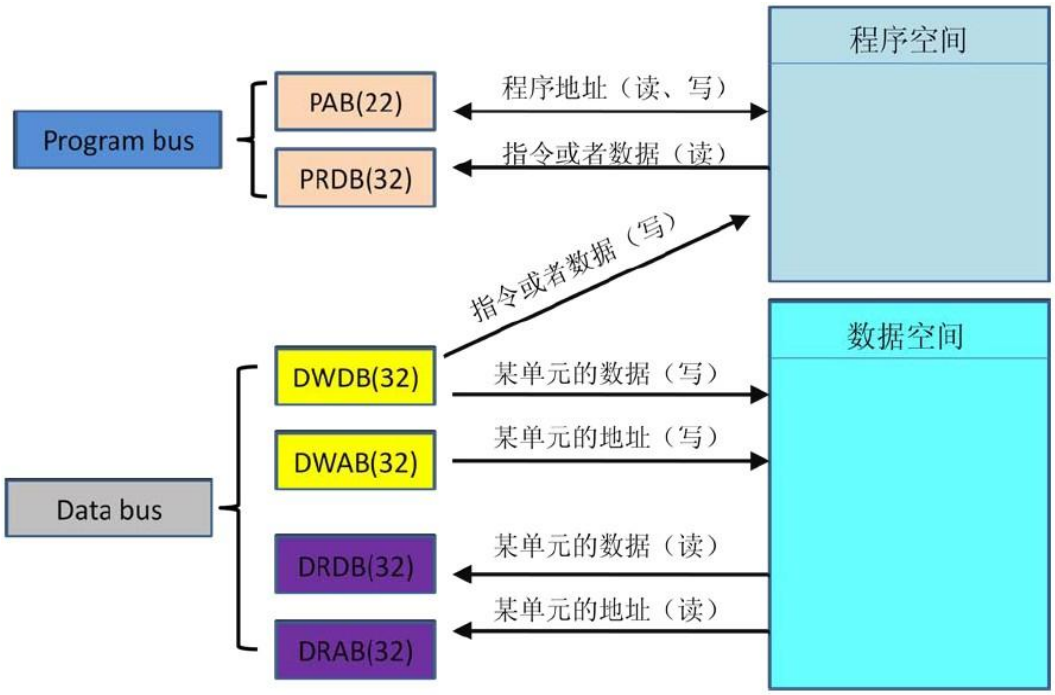
3.DWAB（Data-Write Address Bus）数据写地址总线，它也是个 32 位的总线，用于传送数据空间的写地址。类似的，如果我要对数据空间的某一个单元进行写操作，那么这个单元的地址就是通过 DWAB 来传送。

数据总线，这类总线传送的就是数据了，也就是单元内的具体内容

1. PRDW（Program-Read Data Bus）程序读数据总线，它是一个 32 位的总线，用于传送读取程序空间时的指令或者数据。我们在执行代码的时候，首先是通过 PAB 传送并找到了存放该指令的存储单元，但是这个存储单元下的具体内容就要由我们的 PRDW 来传送了。

2. DRDB（Data-Read Data Bus）数据读数据总线，它是一个 32 位的总线，在读取数据空间时用来传送数据。我们在进行读操作时，先通过 DRAB 总线确定了需要进行读操作的数据单元的地址，接下来传送这个数据单元下面的具体内容时就需要 DRDB 了。

3. DWDB（Data/Program-Write Data Bus）数据写数据总线，它是一个 32 位的总线，在进行写操作时，向数据空间/程序空间传送相应的数据。也就是假如我们要对数据空间的某一个单元进行写操作，我们通过 DWAB 传送了这一个单元的地址，同时我们需要 DWDB 来传送写入的内容。



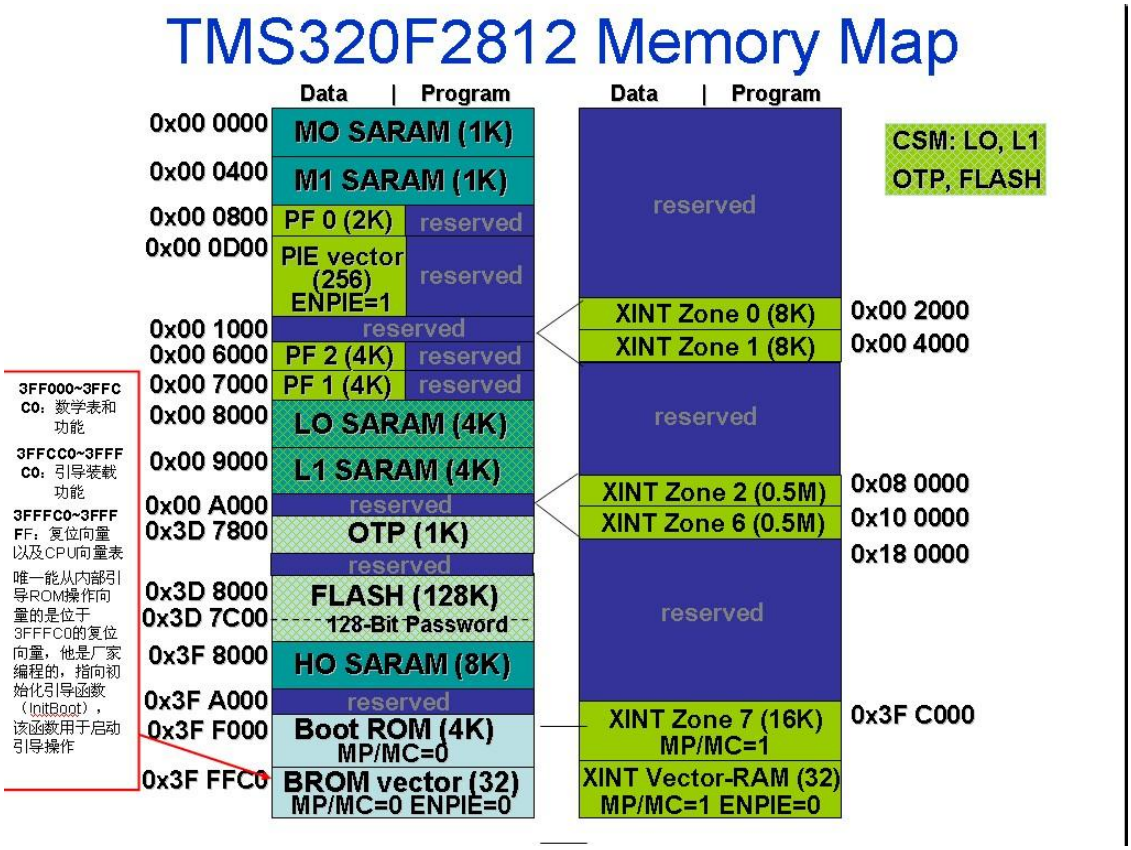
2812 内部的存储器资源

名称	大小
FLASH	128K*16 位
H0（RAM）	8K*16 位
L0（RAM）	4K*16 位
L1（RAM）	4K*16 位
M0（RAM）	1K*16 位
M1（RAM）	1K*16 位
Boot Rom	4K*16 位
OTP（One time Programmable ROM）	1K*16 位

HELLO 四(二)：2812 存储器映射及 CMD

2812 存储器映射

2812 具有 32 位的数据地址和 22 位的程序地址，总地址空间可以达到 4M 的数据空间和 4M 的程序空间。32 位的数据地址，就是能访问 2 的 32 次，是 4G，而 22 位的程序地址，就是能访问 2 的 22 次，是 4M。其实，2812 可寻址的数据空间最大是 4G，但是实际线性地址能达到的只有 4M，原因是 2812 的存储器分配采用的是分页机制，分页机制采用的是形如 0xXXXXXXX 的线性地址，所以数据空间能寻址的只有 4M。



2812 的存储器被划分成了下面的几个部分：

1. 程序空间和数据空间。2812 所具有的 RAM、ROM 和 FLASH 都被统一编址，映射到了程序空间和数据空间，这些空间的作用就是存放指令代码和数据变量。

2. 保留区。数据空间里面某些地址被保留了，作为 CPU 的仿真寄存器使用，这些地址是不向用户开放的。

3.CPU 中断向量。在程序空间里也保留了 64 个地址作为 CPU 的 32 个中断向量。通过 CPU 寄存器 ST1 中的 VMAP 位来将这一段地址映射到程序空间的底部或者顶部。

映射和空间的统一编址

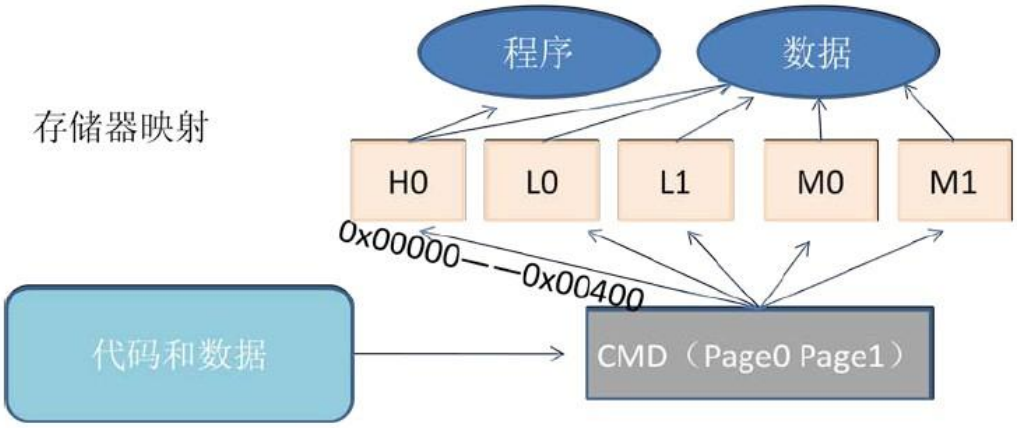


图 3 映射和空间统一编址的理解

F2812 内部的映射空间

低地址空间

低地址空间	
0x0000 0000	M0 矢量 RAM（VMAP=0）
0x0000 0040	M0 SARAM（1K*16）
0x0000 0400	M1 SARAM（1K*16）
0x0000 0800	外设帧 Frame0 （2K*16）
0x0000 0D00	PIE 向量（256*16，VMAP=1）
0x0000 1000	保留空间
0x0000 6000	外设帧 Frame2（4K*16）
0x0000 7000	外设帧 Frame1（4K*16）
0x0000 9000	L1 SARAM（4K*16）
0x0000 A000	保留空间

高地址空间

高地址空间	
0x003D 7800	OTP（1K*16 并保留 1K）
0x003D 8000-0x003F 7FF7	FLASH（128K*16）
0x003F 7FF8-0x003F 7FFF	128 位密钥
0x003F 8000	H0 SARAM（8K*16）
0x003F A000	保留空间
0x003F F000	Boot Rom(4K*16 MP/MC=0)
0x003F FFC0	BROM 向量（VMAP=1 MP/MC=0 ENPIE=0）

2812CMD 详解

CMD：command 命令，顾名思义就是命令文件指定存储区域的分配.2812 的 CMD 采用的是分页制，其中 PAGE0 用于存放程序空间，而 PAGE1 用于存放数据空间。

1.）#pragma ，CODE_SECTION 和 DATA_SECTION 伪指令

#pragma DATA_SECTION(funcA,"dataA"); ------ 函数外声明

将 funcA 数据块定位于用户自定义的段"dataA"中 ------ 需要在 CMD 中指定 dataA 段的物理地址

2.）MEMORY 和 SECTIONS 是命令文件中最常用的两伪指令。MEMORY 伪指令用来表示实际存在目标系统中的可以使用的存储器范围，在这里每个存储器都有自己的名字，起始地址和长度。SECTIONS 伪指令是用来描述输入端是如何组合到输出端内的。

以常用的 F2812_nonBIOS_RAM.cmd F2812_nonBIOS_Flash.cmd 和 DSP281x_Headers_nonBIOS.cmd 为例

F2812_nonBIOS_RAM.cmd ----- 用于仿真，无 BIOS 系统，片外 SRAM 配置

```
MEMORY
{
PAGE 0 :
   RAMM0       : origin = 0x000000, length = 0x000400
   BEGIN       : origin = 0x3F8000, length = 0x000002
   PRAMH0      : origin = 0x3F8002, length = 0x000FFE
   RESET       : origin = 0x3FFFC0, length = 0x000002

PAGE 1 :
   RAMM1     : origin = 0x000400, length = 0x000400
   DRAMH0    : origin = 0x3f9000, length = 0x001000
}

SECTIONS
{
   codestart          : > BEGIN,       PAGE = 0
   ramfuncs           : > PRAMH0       PAGE = 0
   .text              : > PRAMH0,      PAGE = 0
   .cinit             : > PRAMH0,      PAGE = 0
   .pinit             : > PRAMH0,      PAGE = 0
   .switch            : > RAMM0,       PAGE = 0
   .reset             : > RESET,       PAGE = 0, TYPE = DSECT /* not used, */

   .stack             : > RAMM1,       PAGE = 1
   .ebss              : > DRAMH0,      PAGE = 1
   .econst            : > DRAMH0,      PAGE = 1
   .esysmem           : > DRAMH0,      PAGE = 1
}
```

CMD 文件里有两个基本的段：初始化段和非初始化段。初始化段包含代码和常数等必须在 DSP 上电之后有效的数。故初始化块必须保存在如片内 FLASH 等非遗失性存储器中，非初始化段中含有在程序运行过程中才像变量内写数据进去，所以非初始化段必须链接到易失性存储器中如 RAM。

已初始化的段：.text,.cinit,.const,.econst,..pinit 和.switch..

.text：所有可以执行的代码和常量

.cinit:全局变量和静态变量的 C 初始化记录

.const：包含字符串常量和初始化的全局变量和静态变量（由 const）的初始化和说明

.econst：包含字符串常量和初始化的全局变量和静态变量（由 far const）的初始化和说明

.pinit：全局构造器（C++）程序列表

.switch：包含 switch 声明的列表

段	存储类型	页
.text	ROM OR RAM (FLASH)	0
.cinit	ROM OR RAM(FLASH)	0
.const	ROM OR RAM(FLASH)	1
.econst	ROM OR RAM(FLASH)	1
.pinit	ROM OR RAM(FLASH)	0
.switch	ROM OR RAM(FLASH)	0,1

非初始化的段：.bss,.ebss,.stack,.sysmem,和 esysmem.（更好的理解就是，这些段就是存储空间而已）

.bss: 为全局变量和局部变量保留的空间，在程序上电时.cinit 空间中的数据复制出来并存储在.bss 空间中。

.ebss：为使用大寄存器模式时的全局变量和静态变量预留的空间，在程序上电时，cinit 空间中的数据复制出来并存储在.ebss 中

.stack:为系统堆栈保留的空间，用于和函数传递变量或为局部变量分配空间。

.sysmem:为动态存储分配保留的空间。如果有宏函数，此空间被宏函数占用，如果没有的话，此空间保留为 0

.esysmem:为动态存储分配保留的空间。如果有 far 函数，此空间被相应的占用，如果没有的化，此空间保留为 0.

段	存储器类型	页
.const	RAM	1
.econst	RAM	1
.stack	RAM	1
.sysmem	RAM	1
.esysmem	RAM	1

F2812_nonBIOS_Flash.cmd --- 用于无 BIOS，从片内 FLASH 引导

```
MEMORY
{
PAGE 0:    /* Program Memory */
           /* Memory (RAM/FLASH/OTP) blocks can be moved to PAGE1 for data allocation */

   ZONE0       : origin = 0x002000, length = 0x002000     /* XINTF zone 0 */
   ZONE1       : origin = 0x004000, length = 0x002000     /* XINTF zone 1 */
   RAML0       : origin = 0x008000, length = 0x001000     /* on-chip RAM block L0 */
   ZONE2       : origin = 0x080000, length = 0x080000     /* XINTF zone 2 */
   ZONE6       : origin = 0x100000, length = 0x080000     /* XINTF zone 6 */
   OTP         : origin = 0x3D7800, length = 0x000800     /* on-chip OTP */
   FLASHJ      : origin = 0x3D8000, length = 0x002000     /* on-chip FLASH */
   FLASHI      : origin = 0x3DA000, length = 0x002000     /* on-chip FLASH */
   FLASHH      : origin = 0x3DC000, length = 0x004000     /* on-chip FLASH */
   FLASHG      : origin = 0x3E0000, length = 0x004000     /* on-chip FLASH */
   FLASHF      : origin = 0x3E4000, length = 0x004000     /* on-chip FLASH */
   FLASHE      : origin = 0x3E8000, length = 0x004000     /* on-chip FLASH */
   FLASHD      : origin = 0x3EC000, length = 0x004000     /* on-chip FLASH */
   FLASHC      : origin = 0x3F0000, length = 0x004000     /* on-chip FLASH */
   FLASHA      : origin = 0x3F6000, length = 0x001F80     /* on-chip FLASH */
   CSM_RSVD    : origin = 0x3F7F80, length = 0x000076     /* Part of FLASHA.  Program with all
   BEGIN       : origin = 0x3F7FF6, length = 0x000002     /* Part of FLASHA.  Used for "boot t
   CSM_PWL     : origin = 0x3F7FF8, length = 0x000008     /* Part of FLASHA.  CSM password loca

/* ZONE7      : origin = 0x3FC000, length = 0x003FC0     /* XINTF zone 7 available if MP/MCn=
   ROM         : origin = 0x3FF000, length = 0x000FC0     /* Boot ROM available if MP/MCn=0 */
   RESET       : origin = 0x3FFFC0, length = 0x000002     /* part of boot ROM (MP/MCn=0) or XIN
   VECTORS     : origin = 0x3FFFC2, length = 0x00003E     /* part of boot ROM (MP/MCn=0) or XIN

PAGE 1 :   /* Data Memory */
           /* Memory (RAM/FLASH/OTP) blocks can be moved to PAGE0 for program allocation */
           /* Registers remain on PAGE1                                                  */

   RAMM0       : origin = 0x000000, length = 0x000400     /* on-chip RAM block M0 */
   RAMM1       : origin = 0x000400, length = 0x000400     /* on-chip RAM block M1 */
   RAML1       : origin = 0x009000, length = 0x001000     /* on-chip RAM block L1 */
   FLASHB      : origin = 0x3F4000, length = 0x002000     /* on-chip FLASH */
   RAMH0       : origin = 0x3F8000, length = 0x002000     /* on-chip RAM block H0 */
}
```

```
SECTIONS
{

   /* Allocate program areas: */
   .cinit                : > FLASHA      PAGE = 0
   .pinit                : > FLASHA,     PAGE = 0
   .text                 : > FLASHA      PAGE = 0
   codestart             : > BEGIN       PAGE = 0
   ramfuncs              : LOAD = FLASHD,
                           RUN = RAML0,
                           LOAD_START(_RamfuncsLoadStart),
                           LOAD_END(_RamfuncsLoadEnd),
                           RUN_START(_RamfuncsRunStart),
                           PAGE = 0

   csmpasswds            : > CSM_PWL     PAGE = 0
   csm_rsvd              : > CSM_RSVD    PAGE = 0

   /* Allocate uninitalized data sections: */
   .stack                : > RAMM0       PAGE = 1
   .ebss                 : > RAML1       PAGE = 1
   .esysmem              : > RAMH0       PAGE = 1

   /* Initalized sections go in Flash */
   /* For SDFlash to program these, they must be allocated to page 0 */
   .econst               : > FLASHA      PAGE = 0
   .switch               : > FLASHA      PAGE = 0

   /* Allocate IQ math areas: */
   IQmath                : > FLASHC      PAGE = 0                    /* Math Code */
   IQmathTables          : > ROM         PAGE = 0, TYPE = NOLOAD     /* Math Tables In ROM */


   .reset                : > RESET,      PAGE = 0, TYPE = DSECT
   vectors               : > VECTORS     PAGE = 0, TYPE = DSECT

}
```

对于程序在 FLASH 中运行时，需要注意的： DSP 在 150M 时钟频率下，FLASH 中只能提供大约 120M 的时钟频率，所以有时候我们希望在 RAM 中运行时间敏感或计算量很大的子程序（比如 AD 采样）。但是我们所有代码都放在 FLASH 中，这就必须在上电后将 FLASH 中的这段敏感程序复制到 RAM 中运行，加快速度。这是在.CMD 文件就必须划分一段用来设置 RAM 的载入和运行地址。 程序代码如下：

```
SECTIONS {·····
 ramfuncs      :   LOAD = FLASHD,
                   RUN = RAML0,
                   LOAD_START(_RamfuncsLoadStart),
                   LOAD_END(_RamfuncsLoadEnd),
                   RUN_START(_RamfuncsRunStart),
                   PAGE = 0
 }
```

DSP281x_Headers_nonBIOS.cmd ----- 用于无 BIOS，外设寄存器产生的数据段映射至对应的存储器空间

```
MEMORY
{
 PAGE 0:    /* Program Memory */

 PAGE 1:    /* Data Memory */

   DEV_EMU      : origin = 0x000880, length = 0x000180     /* device emulation registers */
   PIE_VECT     : origin = 0x000D00, length = 0x000100     /* PIE Vector Table */
   FLASH_REGS   : origin = 0x000A80, length = 0x000060     /* FLASH registers */
   CSM          : origin = 0x000AE0, length = 0x000010     /* code security module registe
   XINTF        : origin = 0x000B20, length = 0x000020     /* external interface registers
   CPU_TIMER0   : origin = 0x000C00, length = 0x000008     /* CPU Timer0 registers */
   CPU_TIMER1   : origin = 0x000C08, length = 0x000008     /* CPU Timer1 registers */
   CPU_TIMER2   : origin = 0x000C10, length = 0x000008     /* CPU Timer2 registers */
   PIE_CTRL     : origin = 0x000CE0, length = 0x000020     /* PIE control registers */
   ECANA        : origin = 0x006000, length = 0x000040     /* eCAN control and status regis
   ECANA_LAM    : origin = 0x006040, length = 0x000040     /* eCAN local acceptance masks
   ECANA_MOTS   : origin = 0x006080, length = 0x000040     /* eCAN message object time stam
   ECANA_MOTO   : origin = 0x0060C0, length = 0x000040     /* eCAN object time-out registe
   ECANA_MBOX   : origin = 0x006100, length = 0x000100     /* eCAN mailboxes */
   SYSTEM       : origin = 0x007010, length = 0x000020     /* System control registers */
   SPIA         : origin = 0x007040, length = 0x000010     /* SPI registers */
   SCIA         : origin = 0x007050, length = 0x000010     /* SCI-A registers */
   XINTRUPT     : origin = 0x007070, length = 0x000010     /* external interrupt registers
   GPIOMUX      : origin = 0x0070C0, length = 0x000020     /* GPIO mux registers */
   GPIODAT      : origin = 0x0070E0, length = 0x000020     /* GPIO data registers */
   ADC          : origin = 0x007100, length = 0x000020     /* ADC registers */
   EVA          : origin = 0x007400, length = 0x000040     /* Event Manager A registers */
   EVB          : origin = 0x007500, length = 0x000040     /* Event Manager B registers */
   SCIB         : origin = 0x007750, length = 0x000010     /* SCI-B registers */
   MCBSPA       : origin = 0x007800, length = 0x000040     /* McBSP registers */
   CSM_PWL      : origin = 0x3F7FF8, length = 0x000008     /* Part of FLASHA.  CSM passwor
}
```

```
SECTIONS
{
   PieVectTableFile : > PIE_VECT,    PAGE = 1

/*** Peripheral Frame 0 Register Structures ***/
   DevEmuRegsFile     : > DEV_EMU,      PAGE = 1
   FlashRegsFile      : > FLASH_REGS,   PAGE = 1
   CsmRegsFile        : > CSM,          PAGE = 1
   XintfRegsFile      : > XINTF,        PAGE = 1
   CpuTimer0RegsFile  : > CPU_TIMER0,   PAGE = 1
   CpuTimer1RegsFile  : > CPU_TIMER1,   PAGE = 1
   CpuTimer2RegsFile  : > CPU_TIMER2,   PAGE = 1
   PieCtrlRegsFile    : > PIE_CTRL,     PAGE = 1

/*** Peripheral Frame 1 Register Structures ***/
   ECanaRegsFile      : > ECANA,        PAGE = 1
   ECanaLAMRegsFile   : > ECANA_LAM     PAGE = 1
   ECanaMboxesFile    : > ECANA_MBOX    PAGE = 1
   ECanaMOTSRegsFile  : > ECANA_MOTS    PAGE = 1
   ECanaMOTORegsFile  : > ECANA_MOTO    PAGE = 1

/*** Peripheral Frame 2 Register Structures ***/
   SysCtrlRegsFile    : > SYSTEM,       PAGE = 1
   SpiaRegsFile       : > SPIA,         PAGE = 1
   SciaRegsFile       : > SCIA,         PAGE = 1
   XIntruptRegsFile   : > XINTRUPT,     PAGE = 1
   GpioMuxRegsFile    : > GPIOMUX,      PAGE = 1
   GpioDataRegsFile   : > GPIODAT       PAGE = 1
   AdcRegsFile        : > ADC,          PAGE = 1
   EvaRegsFile        : > EVA,          PAGE = 1
   EvbRegsFile        : > EVB,          PAGE = 1
   ScibRegsFile       : > SCIB,         PAGE = 1
   McbspaRegsFile     : > MCBSPA,       PAGE = 1

/*** Code Security Module Register Structures ***/
   CsmPwlFile         : > CSM_PWL,      PAGE = 1
}
```

查看段的分配及使用情况.map 的链接器（存储器）分配映射文件，链接器的 map 文件描述以下内容： 通过 map 文件可以查看各段的分配情况，包括段的起始地址，使用的字节数等配合 cmd 文件的使用，可确定各个段的使用情况，从而保证程序的正常运行和最小的空间使用。

VisualLinker 可视化链接器 TI 公司出品的 DSP 软件开发环境 CCS 还提供了一种可视化生成存储器配置文件的工具：VisualLinker 可视化链接器。如果程序原来包含了一个链接器命令文件(.cmd 文件)，则当创建可视化链接文件的时候，原来 cmd 文件中的内存配置仍然会被使用。如果读者想修改内存配置，双击.rcp 文件就会在 CCS 中打开可视化链接器的图形界面，调整每个内存模块的大小，直到认为合适，然后只需要重新连编，程序即可生成新的输出文件，重复上面的步 骤，直到出现满意的结果。

HELLO 五(一)：2812 中断系统概述

推　荐

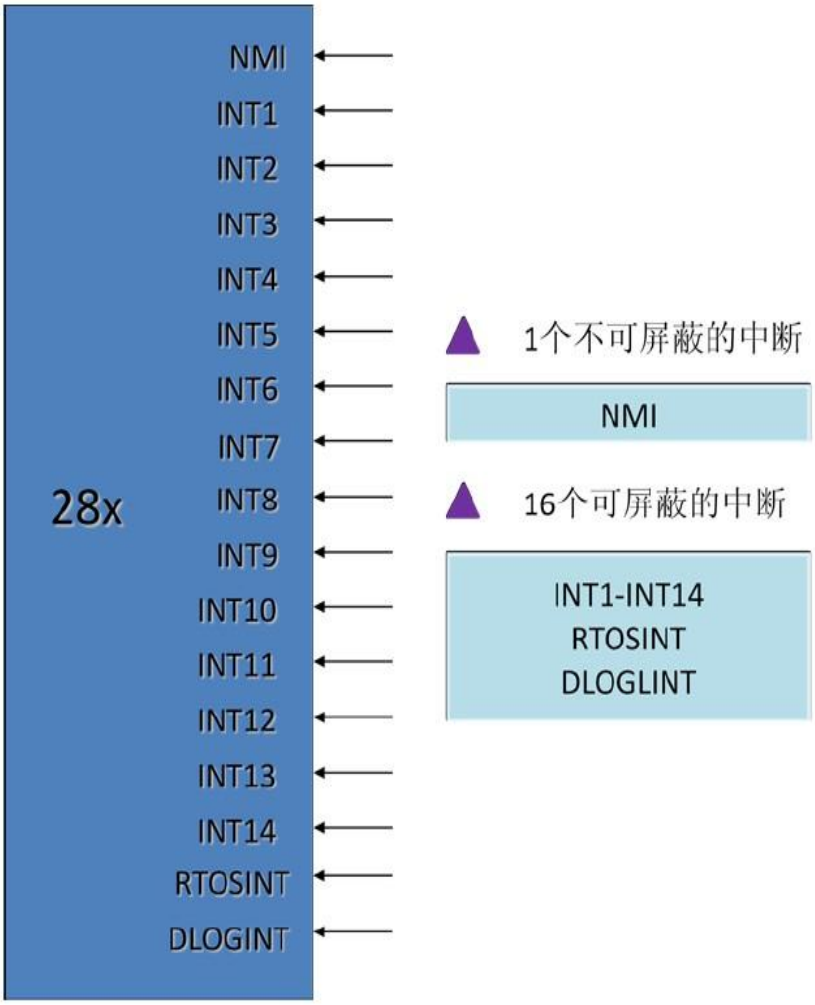
这个系列为响应 HELLODSP 的 2812 学习活动的个人笔记，HELLODSP 版权所有。转载请注明

---By eys417

什么是中断？

中断（Interrupt）是硬件和软件驱动事件，它使得 CPU 暂停当前的主程序，并转而去执行一个中断服务程序。

2812 的中断系统



2812 的 CPU 能够支持一个不可屏蔽中断 NMI 和 16 个可屏蔽的中断 INT1-INT14、RTOSINT 和 DLOGINT，2812 的 CPU 为了能够

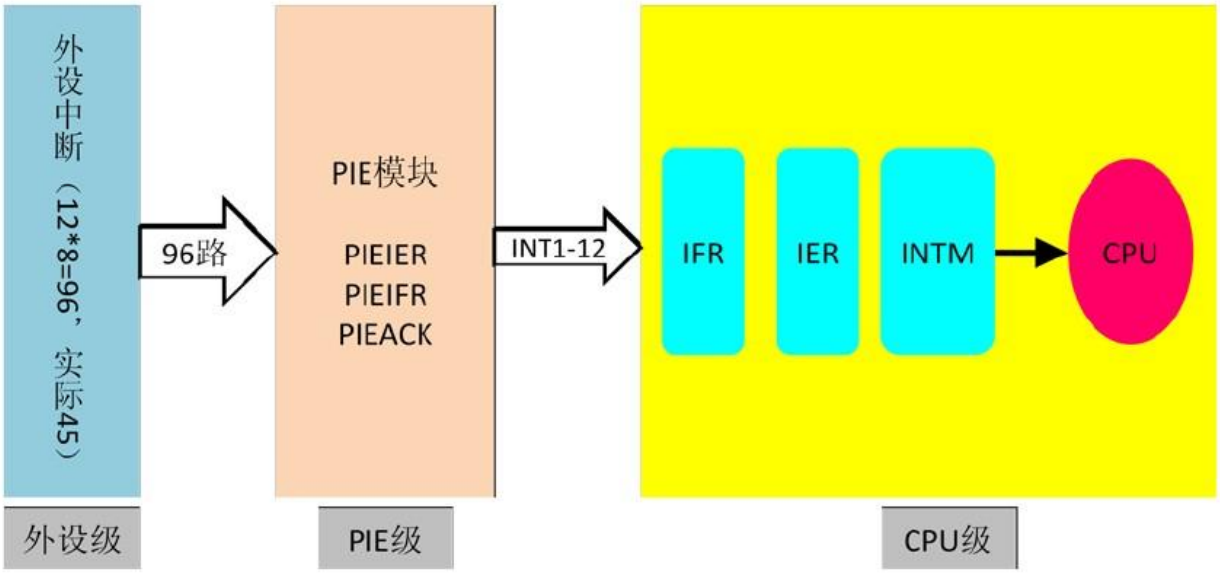
及时有效的处理好各个外设的中断请求，设计了一个专门处理外设中断的扩展模块（the Peripheral Interrupt Expansion block），叫做外设中断控制器 PIE，它能够对各种中断请求源（例如来自于外设或者其他外部引脚的请求）做出判断以及相应的决策。PIE 可以支持 96 个不同的中断，这些中断分成了 12 个组，每个组有 8 个中断，而且每个组都被反馈到 CPU 内核的 12 条中断线中的某一条上（INT1-INT12）。PIE 目前只使用了 96 个终端中的 45 个，其他的等待将来的功能扩展。

2812 的 PIE 内部的中断分布图

	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1	WAKEINT	TINT0	ADCINT	XINT2	XINT1		PDPINTB	PDPINTA
INT2		T1OFINT	T1UFINT	T1CINT	T1PINT	CMP3INT	CMP2INT	CMP1INT
INT3		CAPINT3	CAPINT2	CAPINT1	T2OFINT	T2UFINT	T2CINT	T2PINT
INT4		T3OFINT	T3UFINT	T3CINT	T3PINT	CMP6INT	CMP5INT	CMP4INT
INT5		CAPINT6	CAPINT5	CAPINT4	T4OFINT	T4UFINT	T4CINT	T4PINT
INT6			MXINT	MRINT			SPITXINTA	SPIRXINTA
INT7								
INT8								
INT9			ECAN1INT	ECAN0INT	SCITXINTB	SCIRXINTB	SCITXINTA	SCIRXINTA
INT10								
INT11								
INT12								

PIE 内部的中断 8 列 12 行，总共有 96 个中断，黄色部分表示已经使用的中断，例如：查看事件管理器 EVA 中定时器 T1 的周期中断 T1PINT-------T1PINT 在行号为 INT2，列号为 INTx.4 的位置，也就是说 T1IPNT 对应于 INT2，是 INT2 中的第四个中断。

2812 的 3 级中断机制



2812 的中断是 3 级中断机制，分别是外设级，PIE 级以及 CPU 级，对于某一个具体的外设中断请求，任意一级的不许可，CPU 最终都不会执行该外设中断。下面我们将以 2812 的外设 EVA 中定时器 T1 的周期中断 T1PINT 为例。

（1）.外设级

假如在程序的执行过程中，某一个外设产生了一个中断事件，那么在这个外设的某个寄存器中与该中断事件相关的中断标志位（IF=Interrupt Flag）被置为 1。此时，如果该中断相应的中断使能（IE=Interrupt Flag）已经被置位为 1，外设就会向 PIE 控制器发出一个中断请求。相反的，如果虽然中断事件产生了，相应的中断标志位也被置 1 了，但是该中断没有被使能（相应的使能位为 0），那么外设就不会向 PIE 发出中断请求，但是值得一提的是，相应的中断标志位会一直保持置位状态，直到用程序清楚它为止。当然，在中断标志位保持在 1 的时候，一旦该中断被使能了，那么外设立马会向 PIE 发出中断申请。 我们用具体的 T1PINT 来进行进一步的说明。当定时器 T1 的计数器寄存器 T1CNT 计数到和 T1 周期寄存器 T1PINT 的值匹配时（相等时），就产生了一个 T1PINT 事件，即 T1 的周期中断。这时候，事件管理器 EVA 的中断标志寄存器 A（EVAIFRA）中的第 7 位 T1PINT FLAG 被置为 1，这时候如果 EVA 的中断屏蔽寄存器 A（EVAIMRA）中的第 7 位 T1PINT 的使能位是 1，则 EVA 就会向 PIE 发出中断请求，当然，如果该位的值是 0，也就是该中断未被使能（被屏蔽），则 EVA 不会向 PIE 发出中断请求，而且 EVAIFRA 中 T1PINT FLAG 位将一直保持为 1，除非通过程序将其清除。需要注意的是，不管在什么情况下，外设寄存器中的中断标志位都必须手工清除。

中断外设级总结：

外设中断的屏蔽，需要将与该中断相关的外设寄存中的中断使能位置 0；
外设中断标志位的清除，需要将与该中断相关的外设寄存中的中断标志位置 1；

```
清除 T1PINT 标志位：  EvaRegs.EVAIFRA.bit.T1PINT = 1;
中断屏蔽位使能 ：     EvaRegs.EVAIMRA.bit.T1PINT = 1;
```

（2）.PIE 级

当外设产生中断事件，相关中断标志位置位，中断使能位使能之后，外设就会把中断请求提交给我们的 PIE 模块。PIE 模块将 96 个外设和外部引脚的中断进行了分组，每 8 个中断为 1 组，一共是 12 组，分别是 PIE1-PIE12。每个组的中断被多路汇集进入 1 个 CPU 中断，例如 PDPINDA,PDPINDB,XINT1,XINT2,ADCINT,TINT0,WAKEINT 这 7 个中断都在 PIE1 组内，这些中断

都汇集到 CPU 中断的 INT1。 和外设级类似的，PIE 控制器中的每个组都会有一个中断标志寄存器 PIEIFRx 和和中断使能寄存器 PIEIERx，当然 x=1.....12。每个寄存器的低 8 位对应于 8 个外设中断，高 8 位保留。例如 T1PINT 对应于 PIEIFR2 的第 4 位和 PIEIER2 的第 4 位。PIE 除了每组具有刚才的 PIEIERx，PIEIFRx 寄存器之外，还有一个 PIEACK 寄存器，它的低 12 位分别对应着 12 个组，即 INT1-INT12，高位保留。假如 T1 的周期中断被响应了，则 PIEACK 寄存器的第 2 位（对应于 INT2）就会被置位，并且一直保持直到手动清除这个标志位。当 CPU 在响应 T1PNT 的时候，PIEACK 的第 2 位一直是 1，这时候如果 PIE2 组内发生其他的外设中断，则暂时不会被 PIE 响应送给 CPU，必须等到 PIEACK 的第 2 位被复位之后，如果该中断请求还存在，那么立马由 PIE 控制块将中断请求送至 CPU。所以，每个外设中断被响应之后，一定要对 PIEACK 的相关位进行手动服务，否则同组内的其他中断都不会被响应。

中断 PIE 级总结：

PIE 中断的使能。就得将其相应组的使能寄存器 PIEIERx 的相应位进行置位；
PIE 中断的屏蔽。这是和使能相反的操作；
PIE 应答寄存器 PIEACK 相关位的清除，以使得 CPU 能够响应同组的其他中断。

将 PIE 级的中断和外设级的中断相比较之后发现，外设中断的中断标志位是需要手工清除的，而 PIE 级的中断标志位都是自动置位或者清除的。但是 PIE 多了一个 PIEACK 寄存器，同一时间只能放一个中断过去，只有等到这个中断被响应，给 PIEACK 置位，才能让同组的下一个中断过去，被 CPU 响应

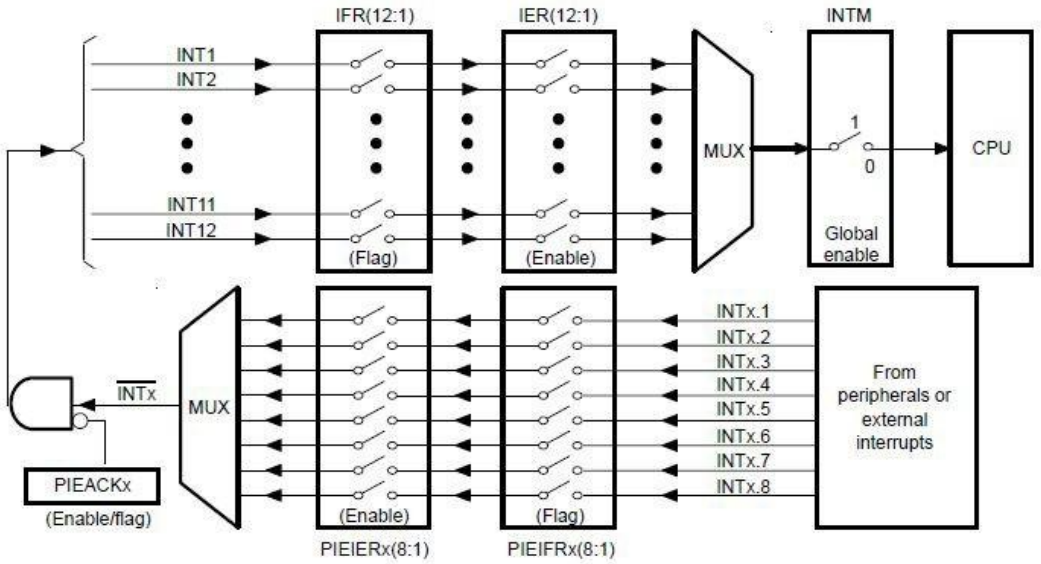
清除 PIE 中与 T1PINT 相关的应答位： PieCtrl.PIEACK.bit.ACK2=1;
使能 PIE 中断 INT2.4(T1PINT 中断)： PieCtrlRegs.PIEIER2.all=M_INT4; //M_INT4=0x0008

（3）CPU 级

CPU 也有标志寄存器 IFR 和使能寄存器 IER。当某一个外设中断请求通过 PIE 发送到 CPU 时，CPU 级中与 INTx 相关的中断标志位就会被置位。例如，T1 的周期中断 T1PINT 的请求到达 CPU 这边时，与其相关的 INT2 的标志位就会被置位。这时候，该标志位就会被所存在 IFR 中，这时候，CPU 不会马上去执行相应的中断，而是等待 CPU 使能 IER 寄存器的相关位，并且对 CPU 寄存器 ST1 中的全局中断屏蔽位做适当的使能。如果 IER 中的相关位被置位了，并且 INTM 的值为 0，则中断就会被 CPU 响应。在 T1PINT 里，当 IER 的第 2 位即 INT2 被置位，INTM 为 0，则 CPU 就会响应定时器 T1 的周期中断。 CPU 接到了终端的请求，就得暂停正在执行的程序，转而去响应中断程序，但是此时，它必须得做一些准备工作，以便于执行完中断程序之后回过头来还能找到原来的地方和原来的状态。CPU 会将相应的 IER 和 IFR 位进行清除，EALLOW 也被清除，INTM 被置位，就是不能响应其他中断了，CPU 向其他中断发出了通知，正在忙，没空来处理你们的请求了，得等到处理完手上的中断之后才能再来处理你们的请求。然后，CPU 会存储返回地址并自动保存相关的信息，例如将正在处理的数据放入堆栈等等，做好这些准备工作之后，CPU 会从 PIE 块中取出对应的中断向量 ISR，从而转去执行中断子程序。

中断 CPU 级总结：

CPU 级的操作都是自动的，不管是中断标志位（IFR），还是中断的使能位（IER）。



HELLO 五(二)：2812 中断系统程序


```
// $        Date:  24/10/2009      整理：  eyes417     $
//###########################################################################
//
//  FILE :  Example_281x_Interrupt.c
//
//  TITLE:  2812 中断函数写法，格式 I
//
//                       ASSUMPTIONS:
//
//  As supplied, this project is configured for "boot to H0" operation.
```

```
// Other then boot mode pin configuration, no other hardware configuration is requir
ed.
//
//###########################################################################
```

1).先定义中断函数，再给相应的 PIE 中断赋地址。写在在 main.c 中

```
interrupt void eva_timer1_isr(void);   //中断函数声明
void main(void)
{
   InitSysCtrl();

//禁止和清除所有 CPU 中断
   DINT;
   IER = 0x0000;
    IFR = 0x0000;

    InitPieCtrl();            //初始化 PIE 控制寄存器
   InitPieVectTable();   //初始化 PIE 中断向量表

// 赋予地址,中断发生时,自动跳转
   EALLOW;
   PieVectTable.T1PINT = &eva_timer1_isr;
   EDIS;

// InitPeripherals();   //初始化所有外设
   init_eva_timer1();

//使能 PIE 中断 INT2.4(T1PINT 中断)
   PieCtrlRegs.PIEIER2.all = M_INT4;
```

```
    IER |= M_INT2        //开 CPU 中断


    EINT;      // Enable Global interrupt INTM
    ERTM ;   // Enable Global realtime interrupt DBGM
    for(;;);
}

//EV-A 定时器 1 中断服务函数
interrupt void eva_timer1_isr(void)
{
      .........              //中断内容
     //清除定时器标志位
    EvaRegs.EVAIMRA.bit.T1PINT = 1;
   EvaRegs.EVAIFRA.all = BIT7;     //BIT7---0x0080
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP2;
}
```

2).写在 DSP281x_DefaultIsr.c 中。

```
void main(void)
{
    InitSysCtrl();


//禁止和清除所有 CPU 中断
    DINT;
    IER = 0x0000;
    IFR = 0x0000;


    InitPieCtrl();  //初始化 PIE 控制寄存器
```

```
    IInitPieVectTable();   //初始化 PIE 中断向量表

// InitPeripherals();   //初始化所有外设
    init_eva_timer1();

//使能 PIE 中断 INT2.4(T1PINT 中断)
    PieCtrlRegs.PIEIER2.all = M_INT4;
    IER |= M_INT2          //开 CPU 中断

    EINT;      // Enable Global interrupt INTM
    ERTM ;   // Enable Global realtime interrupt DBGM
    for(;;);
}
```

在 DSP281x_DefaultIsr.c 中

```
interrupt void T1PINT_ISR(void)
 {
    ..........
    EvaRegs.EVAIMRA.bit.T1PINT = 1;          //中断屏蔽位
    EvaRegs.EVAIFRA.bit.T1PINT=1;           //清除中断标志位
    PieCtrlRegs.PIEACK.bit.ACK2=1;           //响应同组中断
    EINT;                                  //开全局中断
}
```

HELLO 六：2812 的时钟系统

2

推　荐

这个系列为响应 **HELLODSP** 的 **2812** 学习活动的个人笔记，**HELLODSP** 版权所有。转载请注明

---By eys417

1. 振荡器 OSC 和锁相环 PLL

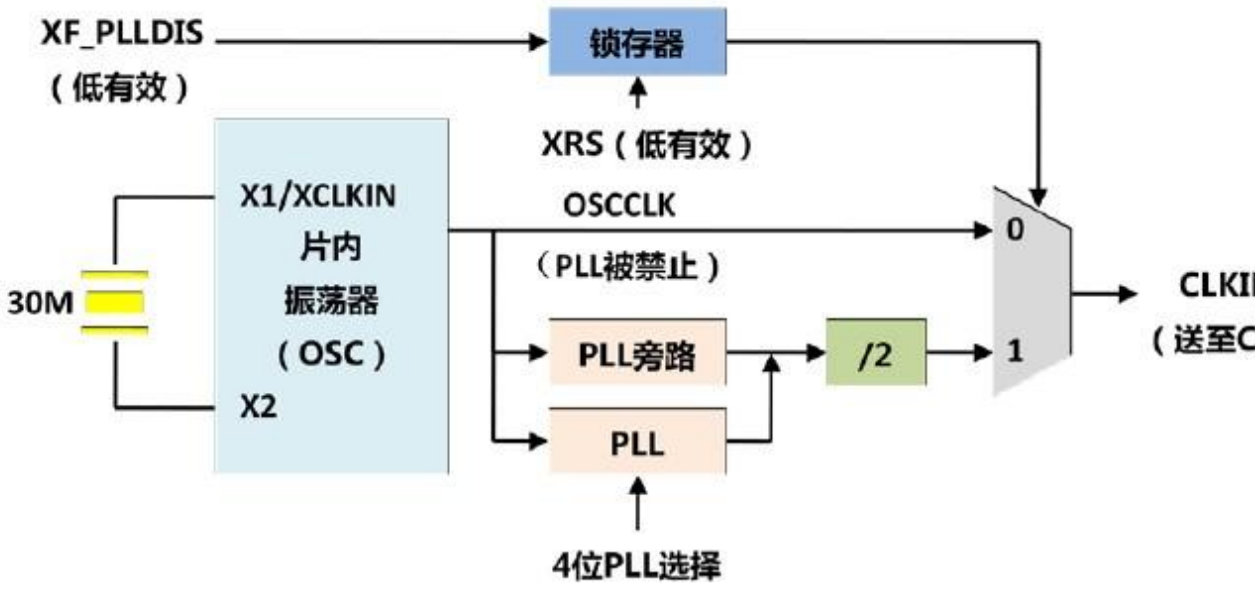


图 1　2812 芯片内的 OSC 和 PLL 模块

2812 基于 PLL 的时钟模块可以采用两种模式，一种是 PLL 未被禁止的情况下（旁路或使能），使用外部晶振给 2812 提供时钟信号，使用 X1/CLKIN 引脚和 X2 引脚；另外一种 PLL 被禁止的情况下，旁路片内振荡器，由外部时钟源提供时钟信号，即将外部振荡器的信号输入到 X1/XCLKN 引脚，X2 引脚不使用。通常情况下，采用第一种方式，由外部晶振通过片内 OSC 来产生时钟信号。

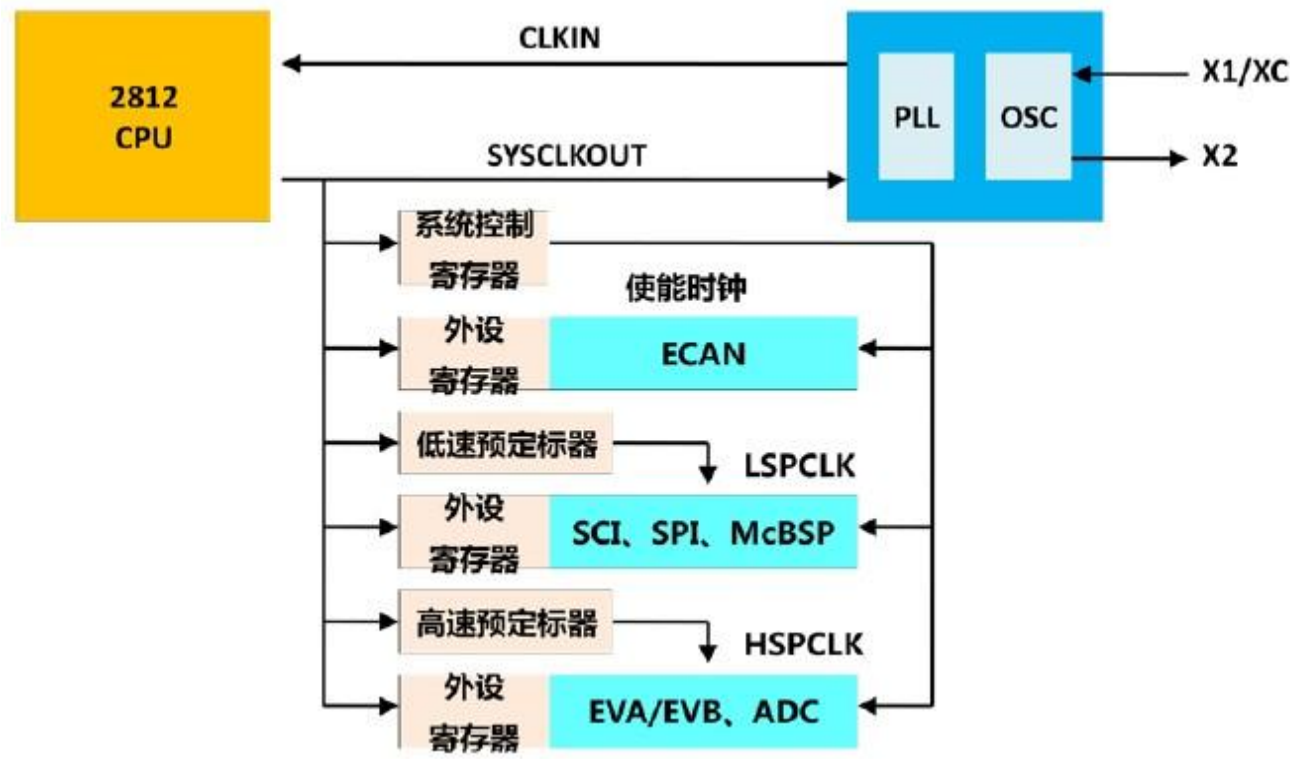
PLL 模式	说明	
PLL 禁止	上电复位时通过将 XPLLDSIS（低电平有效）引脚置低来进入该模式，PLL 模块完全不使能。此时，输入 CPU 的时钟是由外部振荡器直接通过 X1/XCLKIN 引脚输入的信号。	
PLL 旁路	如果 PLL 未处于不使能的状态，上电默认的 PLL 配置（PLLCR 的值为 0）。PLL 自身被旁路，从 X1/XCLKIN 引脚输入的时钟信号先被/2，然后再送去 CPU。	
PLL 使能	通过给 PLLCR 寄存器写一个不为 0 的值来实现 PLL 的使能，时钟信号需要进入 PLL 模块进行 n 倍频，然后再被/2，最后送至 CPU。	

平常使用的是第 3 种方式，即 PLL 使能

晶振为 30M，PLLCR 的 DIV 位被设置成 1010 时的时钟频率

CLKIN=（OSCLKIN*10）/2=(XCLKIN*10)/2=(30M*10)/2=150M Hz

2.2812 芯片中各种时钟信号的产生情况



和时钟使能相关的寄存器是外设时钟控制寄存器 PCLKCR。使能外设时钟

SysCtrlRegs.PCLKCR.bit.SCIENCLKA=1;

SysCtrlRegs.PCLKCR.bit.EVAENCLK=1;

SysCtrlRegs.PCLKCR.bit.ADCENCLK=1;

低速时钟外设 ---- SCIA、SCIB、SPI、McBSP

LSPCLK 计算公式

LOSPCP=0,LSPCLK=SYSCLKOUT

LOSPCP=1—7,LSPCLK=SYSCLKOUT/（2*LOSPCP）

注：LOSPCP 表示的是 LOSPCP 寄存器中位 2-0 的值

高速时钟外设 ---- EVA、EVB 和 ADC

HSPCLK 计算公式

HISPCP=0,HSPCLK=SYSCLKOUT

HISPCP=1—7,HSPCLK=SYSCLKOUT/（2*HISPCP）

注：HISPCP 表示的是 HISPCP 寄存器中位 2-0 的值

SYSOUTCLK 组:CPU 定时器,eCAN 总线

OSCCLK 组:看门狗电路

3.看门狗（Watch Dog）

看门狗,又叫 watchdog timer,是一个定时器电路，一般有一个输入,叫喂狗(kicking the dog or service the dog),一个输出到 MCU 的 RST 端,MCU 正常工作的时候,每隔一端时间输出一个信号到喂狗端,给 WDT 清零,如果超过规定的时间不喂狗,(一般在程序跑飞时),WDT 定时超过,就回给出一个复位信号到 MCU,是 MCU 复位．防止 MCU 死机．看门狗的作用就是防止程序发生死循环，或者说程序跑飞。

2812 的看门狗电路有一个 8 位的看门狗加法计数器 WDCNTR，无论什么时候，如果 WDCNTR 计数到最大值时，看门狗模块就会产生一个输出脉冲，脉冲宽度为 512 个振荡器时钟宽度。为了防止看门狗加法计数器 WDCNTR 溢出，我们通常可以采用两种方法：一种是禁止看门狗，即使得计数器 WDCNTR 无效；另一种就是定期的“喂狗”，通过软件向负责复位看门狗计数器的看门狗密钥寄存器（8 位的 WDKEY）周期性的写入 0x55+0xAA，紧跟着 0x55 写入 0xAA 能够清除 WDCNTR。写任何其他的值都会使看门狗立即复位

4.系统初始化函数

系统初始化函数 DSP281x_SysCtrl.c 文件。

```
void InitSysCtrl(void)
{

/*  EALLOW;

//固定格式，在 TMX 采样时，为了能够使得片内 RAM 模块 M0/M1/L0/L1LH0 能够获得最好的性能，
控制寄存器的位必须使能，这些位在设备硬件仿真寄存器内
   DevEmuRegs.M0RAMDFT = 0x0300;
   DevEmuRegs.M1RAMDFT = 0x0300;
   DevEmuRegs.L0RAMDFT = 0x0300;
   DevEmuRegs.L1RAMDFT = 0x0300;
   DevEmuRegs.H0RAMDFT = 0x0300;

   EDIS;
*/
  DisableDog();              // Disable the watchdog

   InitPll(0xA);               // Initialize the PLLCR to 0xA---30M*10/2

   InitPeripheralClocks();     // Initialize the peripheral clocks

}

//禁止看门狗

void DisableDog(void)
{
    EALLOW;
    SysCtrlRegs.WDCR= 0x0068;
    EDIS;
}

//喂狗函数

void KickDog(void)
{
    EALLOW;
    SysCtrlRegs.WDKEY = 0x0055;
    SysCtrlRegs.WDKEY = 0x00AA;
```

```
    EDIS;
}

//PLL 初始化，30M*val/2

void InitPll(Uint16 val)
{
   volatile Uint16 iVol;

   if (SysCtrlRegs.PLLCR.bit.DIV != val)
   {

      EALLOW;
      SysCtrlRegs.PLLCR.bit.DIV = val;
      EDIS;

      DisableDog();

      for(iVol= 0; iVol< ( (131072/2)/12 ); iVol++)
      {

      }
   }
}

//初始化个外设时钟--时能 OR 禁止

void InitPeripheralClocks(void)
{
   EALLOW;

// HISPCP/LOSPCP 高低速时钟设置
   SysCtrlRegs.HISPCP.all = 0x0001;
   SysCtrlRegs.LOSPCP.all = 0x0002;

//初始化个外设时钟--使能 OR 禁止
   SysCtrlRegs.PCLKCR.bit.EVAENCLK=1;
   SysCtrlRegs.PCLKCR.bit.EVBENCLK=1;
   SysCtrlRegs.PCLKCR.bit.SCIAENCLK=1;
   SysCtrlRegs.PCLKCR.bit.SCIBENCLK=1;
   SysCtrlRegs.PCLKCR.bit.MCBSPENCLK=1;
   SysCtrlRegs.PCLKCR.bit.SPIENCLK=1;
   SysCtrlRegs.PCLKCR.bit.ECANENCLK=1;
   SysCtrlRegs.PCLKCR.bit.ADCENCLK=1;
```

```
   EDIS;
}
```

HELLO 七：2812 的 IO 口控制--LED 点亮程序

复用控制寄存器 --- GPxMUX　　（0---数字 IO，1---专用外设功能）

方向控制寄存器 --- GPxDIR　　（0---输入，1---输出）

量化控制寄存器 --- GPxQUAL　　（0---无量化，1---量化范围 0x00--0xff）

I/O 数据寄存器 --- GPxDAT　　（0---输出--引脚置低，1---输出--引脚置高）

I/O 置位寄存器 --- GPxSET　　（0---无变化，1---引脚置为高）

I/O 清零寄存器 --- GPxCLEAR　　（0---无变化，1---引脚置为低）

单独触发寄存器 --- GPxTOGGLE　　（0---无变化，1---引脚置电平跳变一次）

```
//IO 口初始化
void InitGpio(void)
{
 EALLOW;                    // Enable EALLOW protected register access

//--- Group A pins
 GpioMuxRegs.GPADIR.all=0x0000;     //若设为数字 I/O，可配置为输入(0)/输出(1)
 GpioMuxRegs.GPAQUAL.all=0x0000; // 0x00--无量化，0xff--510 个 SYSCLKOUT 周
期量化过程

//---选择引脚作为专用外设或通用 I/O
```

```
GpioMuxRegs.GPAMUX.bit.C3TRIP_GPIOA15  = 0;  // 0: select GPIO
GpioMuxRegs.GPAMUX.bit.C2TRIP_GPIOA14  = 0;  // 0: select GPIO
GpioMuxRegs.GPAMUX.bit.C1TRIP_GPIOA13  = 0;  // 0: select GPIO
GpioMuxRegs.GPAMUX.bit.TCLKINA_GPIOA12 = 0;  // 0: select GPIO
GpioMuxRegs.GPAMUX.bit.TDIRA_GPIOA11   = 0;  // 0: select GPIO
GpioMuxRegs.GPAMUX.bit.CAP3QI1_GPIOA10 = 0;  // 0: select GPIO
GpioMuxRegs.GPAMUX.bit.CAP2Q2_GPIOA9   = 0;  // 0: select GPIO
GpioMuxRegs.GPAMUX.bit.CAP1Q1_GPIOA8   = 1;  // 1: select periph
GpioMuxRegs.GPAMUX.bit.T2PWM_GPIOA7    = 0;  // 0: select GPIO
GpioMuxRegs.GPAMUX.bit.T1PWM_GPIOA6    = 0;  // 0: select GPIO
GpioMuxRegs.GPAMUX.bit.PWM6_GPIOA5     = 0;  // 0: select GPIO
GpioMuxRegs.GPAMUX.bit.PWM5_GPIOA4     = 0;  // 0: select GPIO
GpioMuxRegs.GPAMUX.bit.PWM4_GPIOA3     = 0;  // 0: select GPIO
GpioMuxRegs.GPAMUX.bit.PWM3_GPIOA2     = 0;  // 0: select GPIO
GpioMuxRegs.GPAMUX.bit.PWM2_GPIOA1     = 0;  // 0: select GPIO
GpioMuxRegs.GPAMUX.bit.PWM1_GPIOA0     = 1;  // 1: select periph

EDIS;     // Disable EALLOW protected register access

}
```

HELLO 八(一) 2812EV 模块---通用定时器

2

推　荐

事件管理器的功能

2812 具有两个事件管理器模块 EVA 和 EVB，每个 EV 模块都具有 2 个通用定时器、3 个比较单元、3 个捕获单元以及 1 个正交编码电路。

（表格中蓝色的字表示该信号是低电平有效）

	EVA		EVB	
事件管理器模块	模块	信号引脚	模块	信号引脚
通用定时器	定时器 1	T1PWM_T1CMP	定时器 3	T3PWM_T3CMP
	定时器 2	T2PWM_T2CMP	定时器 4	T4PWM_T4CMP
比较单元	比较器 1	PWM1/2	比较器 4	PWM7/8
	比较器 2	PWM3/4	比较器 5	PWM9/10
	比较器 3	PWM5/6	比较器 6	PWM11/12
捕获单元	捕获器 1	CAP1_QEP1	捕获器 4	CAP4_QEP3
	捕获器 2	CAP2_QEP2	捕获器 5	CAP5_QEP4
	捕获器 3	CAP3_QEPI1	捕获器 6	CAP6_QEPI2
QEP 电路	QEP1	CAP1_QEP1	QEP3	CAP4_QEP3
	QEP2	CAP2_QEP2	QEP4	CAP5_QEP4
	QEP11	CAP3_QEPI1	QEP12	CAP6_QEPI2
外部定时器输入	计数方向	TDIRA	计数方向	TDIRB
	外部时钟	TCLKINA	外部时钟	TCLKINB
External compare-output trip inputs		C1TRIP		C4TRIP
		C2TRIP		C5TRIP
		C3TRIP		C6TRIP
External timer-compare trip inputs		T1CTRIP_PDPINTA		T3TRIP_PDPINTB
		T2CTRIP/EVASOC		T4CTRIP/EVBSOC
External trip inputs		T1CTRIP_PDPINTA		T3TRIP_PDPINTB
外部 AD 转换启动信号		T2CTRIP/EVASOC		T4CTRIP/EVBSOC

通用定时器用来计时的，而且每个定时器还能产生 1 路独立的 PWM 波形；

比较单元主要功能就是用来生成 PWM 波形的，EVA 具有 3 个比较单元，每个单元可以生成一对（两路）互补的 PWM 波形，生成的 6 路 PWM 波形正好可以驱动一个三相桥电路。

捕获单元的功能是捕捉外部输入脉冲波形的上升沿或者下降沿，可以统计脉冲的间隔，也可以统计脉冲的个数。

正交编码电路可以对输入的正交脉冲进行编码和计数，它和光电编码器相连可以获得旋转机械部件的位置和速率等信息。

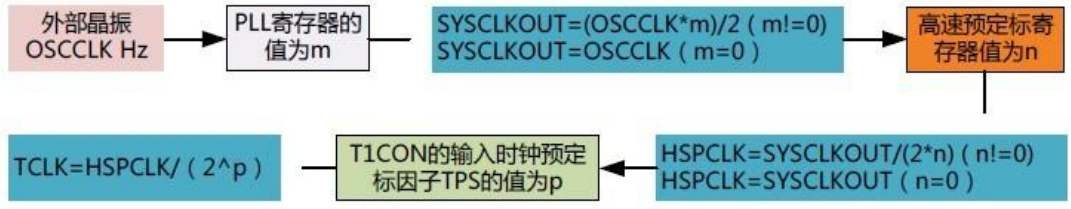
（1） External compare-output trip inputs—我们可以理解为切断比较输出的外部控制输入，以 C1TRIP 为例，当比较单元 1 工作时，其两个引脚 PWM1 和 PWM2 正在不断的输出 PWM 波形，这时候，如果 C1TRIP 信号变为低电平，则此时 PWM1 和 PWM2 引脚被置成高阻态，不会再有 PWM 波形输出，也就是在这个引脚上输入低电平，则比较输出就会被切断。

（2） External timer-compare trip inputs—我们可以理解为切断定时器比较输出的外部控制输入，以 T1PWM_1CMP 为例，当定时器 1 的比较功能在运行，并且 T1PWM 引脚输出 PWM 波形的时候，这时候如果 T1CTRIP 引脚信号变为低电平，则该引脚状态被置成高电平，也不会再有 PWM 波形输出。

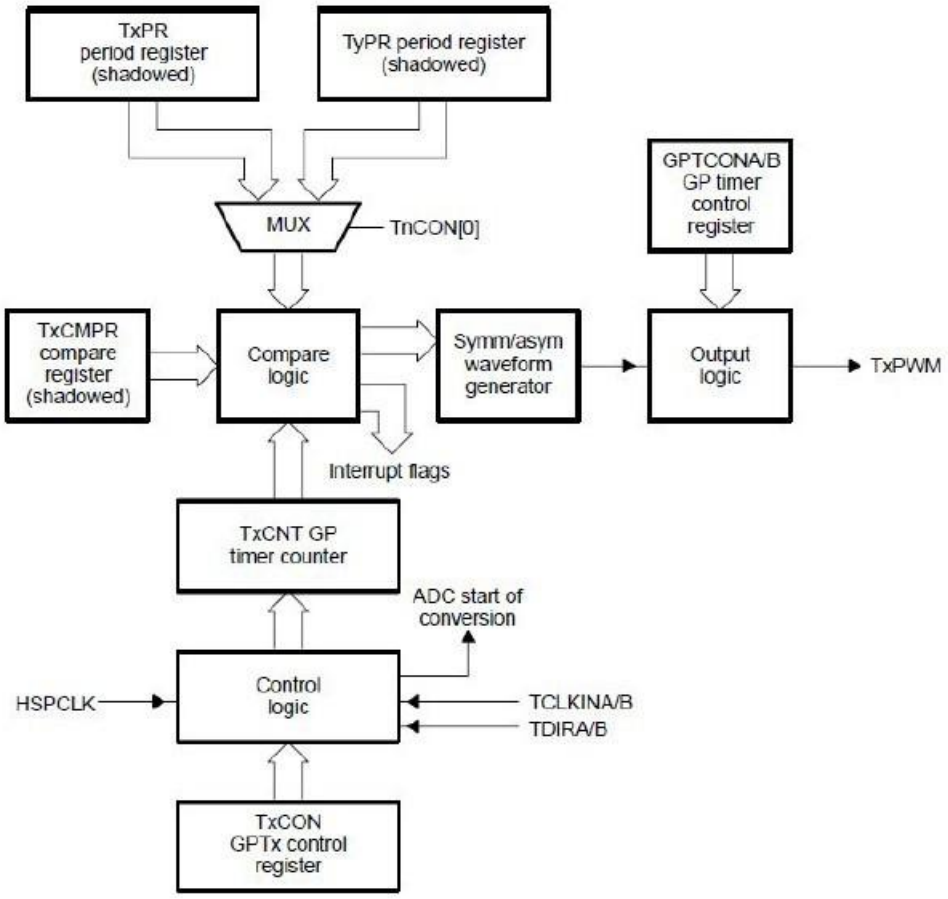
（3） External trip inputs 的 PDPINTx（x=A 或者 B）其实是个功率驱动保护，它为系统的安全提供了保护，例如如果当电路中出现电压、电流或者温度急剧上升的时候，如果 PDPINTx 的中断没有被屏蔽，当 PDPINTx 的引脚变为低电平时，2812 所有的 PWM 输出引脚都会变为高阻态，从而阻止了电路进一步损坏，达到保护系统的目的。当然 PDPINTx 在电路设计时就要考虑到给它配一个监视电路状态的信号。

1. 通用定时器---以通用定时器 1 为例

EV 事件管理器时钟模块



通用定时器模块结构



和 T1 相关的常用寄存器

T1 周期寄存器 ---- T1PR 1 6 位
T1 比较寄存器 ---- T1CMPR 1 6 位
T1 计数寄存器 ---- T1CNT 1 6 位
T1 控制寄存器 ---- T1CON 1 6 位
全局定时器控制寄存器 A ---- GPTCONA 1 6 位

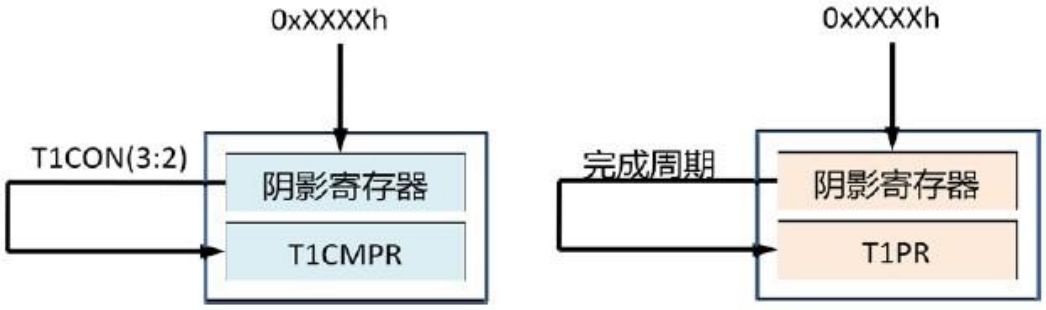
T1 的常见输入信号

来自于 CPU 的内部时钟
外部时钟输入 TCLKINA，最大频率为器件自身时钟的 1/4，也就是 1/4*150M
TDIRA/B，用于定时器的增/减计数模式
复位信号 RESET

T1 的输出信号

定时器的比较输出 T1PWM_T1CMP
送给 ADC 模块的 AD 转换启动信号
下溢、上溢、比较匹配和周期匹配信号
计数方向指示

阴影寄存器作用---重载条件



可以在一个周期的任何时刻向 T1CMPR 或者 T1PR 写入新的数值，假设我们要向 T1CMPR 写入新的数值 0xXXXXh，首先将这个数值写入 T1CMPR 的阴影寄存器，当 T1CON 中第 3 位 TCLD1 和第 2 位 TCLD0 所指定的特定事件发生时，阴影寄存器的数据就会被写入 T1CMPR 的工作寄存器。

向 T1PR 写入新的数据 0xXXXXh，数据也会被立即写入阴影寄存器，只有当 T1CNT 的完成这个周期的计数，值为 0 的时候，阴影寄存器中的内容才会被载入到工作寄存器中，从而改变 T1PR 的值。

定时器比较寄存器重载条件

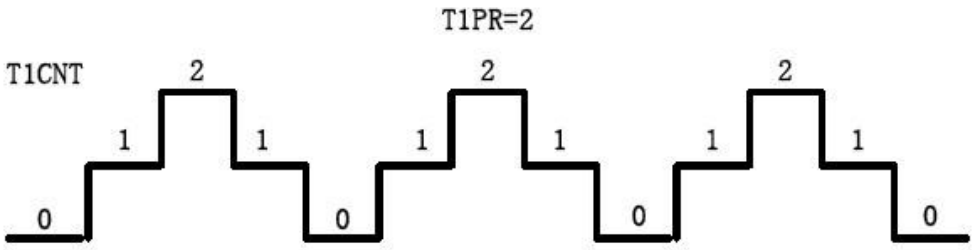
TCLD1	TCLD0	
0	0	当计数器 T1CNT 值为 0
0	1	当计数器 T1CNT 值为 0 或者等于周期寄存器
1	0	立即载入
1	1	保留

定时器的计数方式---由 T1CON 第 12 位 • 11 位决定

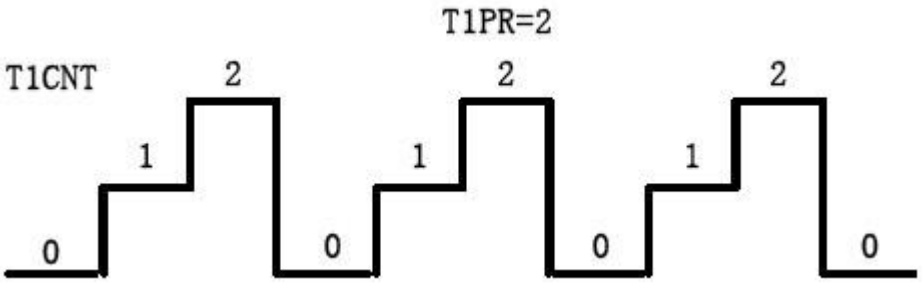
TMODE1	TMODE0	
0	0	停止/保持
0	1	连续增/减模式

1	0	连续增模式
1	1	定向增/减计数模式（directional up/down count mode）

连续增/减技术模式-----实际的计数周期为 2*T1PR

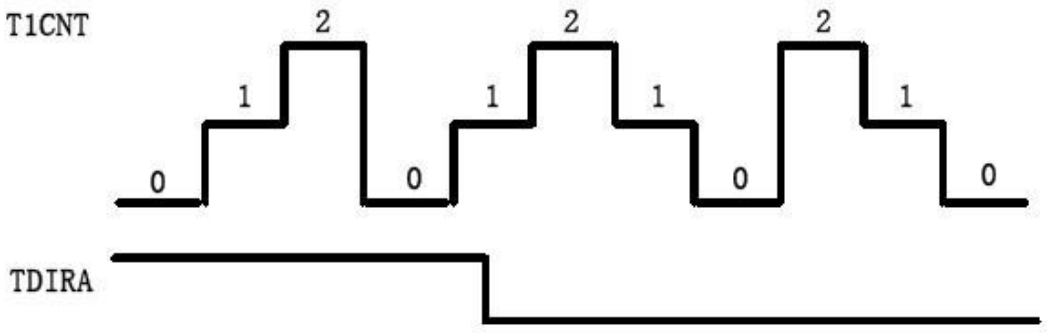


连续增模式-----实际的计数周期为 T1PR+1



定向增或者减计数模式

T1CNT 进行增计数或者是减计数，取决于引脚 TDIRA 的电平，如果 TDIRA 为高电平，则 T1CNT 进行增计数；如果 TDIRA 为低电平，则 T1CNT 进行减计数。如果是在计数过程中 TDIRA 电平发生了变化，那么必须在完成当前计数周期后的下一个 CPU 时钟周期时，计数方向发生改变。



T1 相关的中断

上溢中断 T1OFINT、下溢中断 T1UFINT、比较中断 T1CINT、周期中断 T1PINT

（1）当 T1CNT 的值为 0xFFFFh 的时候，发生定时器 T1 的上溢中断。当上溢事件发生后，再过 1 个 CPU 时钟周期，则上溢中断的标志位被置位。

（2）当 T1CNT 的值为 0x0000h 的时候，发生定时器 T1 的下溢中断。当下溢事件发生后，再过 1 个 CPU 时钟周期，则下溢中断的标志位被置位。

（3）当 T1CNT 的值和 T1 比较寄存器 T1CMPR 的值相等时，发生定时器 T1 的比较中断。当发生比较匹配时，再过 1 个 CPU 时钟周期，则比较中断的标志位被置位。

（4）当 T1CNT 的值和 T1 周期寄存器 T1PR 的值相等时，发生定时器 T1 的周期中断。当发生周期事件时，再过 1 个 CPU 时钟周期，则周期中断的标志位被置位。

当某个中断的标志位被置位，如果该中断已经使能，则会像 PIE 模块发送中断申请。退出中断的时候，一定要手动清除外设中断标志位。在 EV 中，和上述中断相关的寄存器是 EVAIFRA、EVAIMRA 、EVAIFRB、EVAIMRB

上述事件除了能够产生中断以外，还能产生一个 ADSOC 信号，就是启动 AD 转换的信号，这样可以周期性的去启动 AD 转换。依据寄存器 GPTCONA 的第 8 和第 7 位，这个功能的优点就在于允许在 CPU 不干涉的情况下使通用定时器的事件和 ADC 启动转换同步进行

GPTCONA 中 T1 启动 AD 转换的信号的相关位（T1TOADC）

Bit8	bit7	
0	0	不启动 ADC
0	1	下溢中断启动 ADC

1　　0　　周期中断启动 ADC

1　　1　　比较中断启动 ADC

定时器的同步

T2 可以使用 T1 的周期寄存器而忽略自身的周期寄存器，也可以使用 T1 的使能位来启动 T2 计数，这样的功能保证了 T1 和 T2 能够实现同步计数

1．将 T2CON 的 T2SWT1 置 1，实现由 T1CON 的 TENABLE 位来启动通用定时器 2 的计数，这样，两个计数器（T1、T2）就能同时启动计数。

2．对 T1CNT 和 T2CNT 进行不同值的初始化。

3．将 T2CON 的 SELT1PR 置 1，指定定时器 2 将定时器 1 的周期寄存器作为自己的周期寄存器。

2812---通用定时器 1 初始化程序(启动 ADC)

通用定时器 1 初始化程序(启动 ADC)

```
// $        Date:   4/11/2009       整理：  eyes417     $
//###########################################################################
//
//   FILE :  Example_281xEvTimerPeriod.c
//
//  TITLE:  事件管理器 GP 定时器--周期中断启动 ADC 转换
//
//                       ASSUMPTIONS:
//
```

```
//  As supplied, this project is configured for "boot to H0" operation.
//  Other then boot mode pin configuration, no other hardware configuration is required.
//
//###########################################################################

 #i nclude "DSP281x_Device.h"
 #i nclude "DSP281x_Examples.h"

interrupt void eva_timer1_isr(void);
void init_eva_timer1(void);

void main(void)
{
  InitSysCtrl();
  InitGpio();       //IO 口初始化

  DINT;                 //关 CPU 总中断
  InitPieCtrl();      //初始化 PIE 控制寄存器
  IER = 0x0000;
   IFR = 0x0000;
   InitPieVectTable();   //初始化 PIE 中断向量表

  EALLOW;
//将相应的向量指向中断服务程序，中断发生时,自动跳转
  PieVectTable.T1PINT = &eva_timer1_isr;
   EDIS;
```

```
   init_eva_timer1();           //初始化 EV-A 定时器 1

//使能 PIE 中断 INT2.4(T1PINT 中断)
  PieCtrlRegs.PIEIER2.all = M_INT4;
   IER |= M_INT2 ;

   EINT;         // 使能 INTM(全局中断)
  ERTM;          // Enable Global realtime interrupt DBGM

  for(;;);

}

//EV-A 定时器 1 初始化
void init_eva_timer1(void)
{
   EvaRegs.GPTCONA.all = 0;
   EvaRegs.T1PR = 0x1200;       // Period---周期值
  EvaRegs.T1CMPR = 0x0000;     // Compare Reg--比较值

 //清除周期 EV-A 定时器 1 中断位
  EvaRegs.EVAIMRA.bit.T1PINT = 1;
   EvaRegs.EVAIFRA.bit.T1PINT = 1;

   EvaRegs.T1CNT = 0x0000;             //计数器初值
```

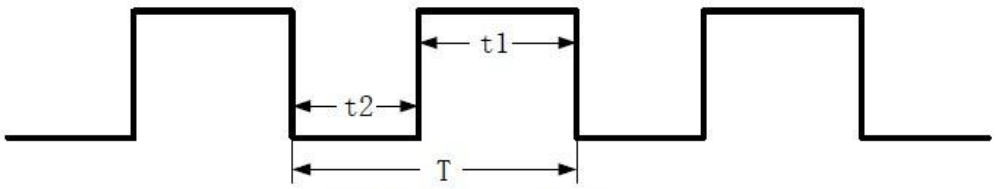
```
//递增模式，x/1 分频，内部时钟，使能比较，使用自己的周期，立即启动定时器计数
  EvaRegs.T1CON.all = 0x1042;

//启动由 EVA 定时器 1 周期中断产生的 ADC 转换
  EvaRegs.GPTCONA.bit.T1TOADC = 2;
}

//EV-A 定时器 1 中断服务函数
interrupt void eva_timer1_isr(void)
{
  ......          //中断服务内容程序

//清除定时器中断位
  EvaRegs.EVAIMRA.bit.T1PINT = 1;
   EvaRegs.EVAIFRA.all = BIT7;       //BIT7---0x0080

//响应中断,从而使 INT2 中断组继续接收中断
  PieCtrlRegs.PIEACK.all = PIEACK_GROUP2;

}
```

HELLO 八(二) 2812EV 模块---PWM

PWM

PWM---Pulse Width Modulation---脉宽调制,简单的描述就是一些矩形脉冲波形，PWM 波形最重要的三个参数是周期、频率和占空比。PWM 是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术，广泛应用在从测量、通信到功率控制与变换的许多领域中。

PWM 波形图



PWM周期：T=t1+t2

PWM频率：F=1/T

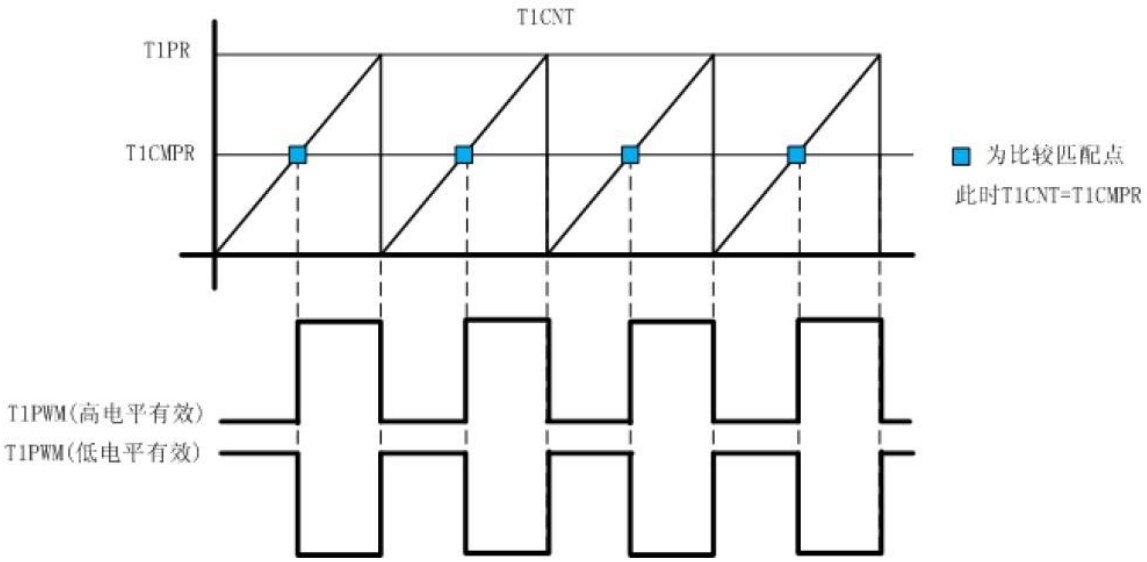
PWM占空比：D=t1/(t1+t2)=t1/T

EVA 的两个通用定时器能够产生 2 路独立的 PWM 波形—T1PWM 和 T2PWM，三个比较单元每一个都能产生一对互补的 PWM 波形，比较单元 1 产生 PWM1 和 PWM2，比较单元 2 产生 PWM3 和 PWM4，比较单元 3 产生 PWM5 和 PWM6。这样，EVA 一共能产生 8 路 PWM 波形。

通用定时器产生的 PWM 波

T1 和 T2 分别能够产生 1 路独立的 PWM，以 T1 为例。当 T1 计数寄存器 T1CNT 的值和 T1CMPR 的值相等时，就会发生比较匹配事件，这时如果 PWM 的功能使能，则 T1PWM 引脚便可以输出 PWM 波形。T1 能够产生两种类型的 PWM，一种是不对称的 PWM 波形，一种是对称的 PWM 波形，取决于 T1CNT 的计数方式。

（1）当 T1CNT 为连续增计数时 ----- 不对称的 PWM 波形。



定时器 T1 工作于连续增模式。当 T1CNT 的值计数到和 T1CMPR 的值相等时，发生比较匹配事件。如果 T1CON 的第 1 位定时器比较使能为 TECMPR 为 1，即定时器比较操作被使能，且 GPTCONA 的第 6 位比较输出使能位 TCMPOE 为 1，同时 GPTCONA 下的 T1PIN 引脚输出极性为高电平或者低电平的话，T1PWM 就会输出不对称的 PWM 波形

T1 连续增模式，定时器的周期 T=（T1PR+1）*tc，其中 tc 为 T1CNT 每计数 1 次所需的时间

TCLK 为定时器时钟频率。分频系数决定

T1PWM 的周期为：$\frac{T1PR+1}{TCLK}$，单位为 s；

T1PWM 的频率为：$\frac{TCLK}{T1PR+1}$，单位为 Hz；

T1PWM 的占空比要分 GPTCONA 中 T1PIN 的输出极性，

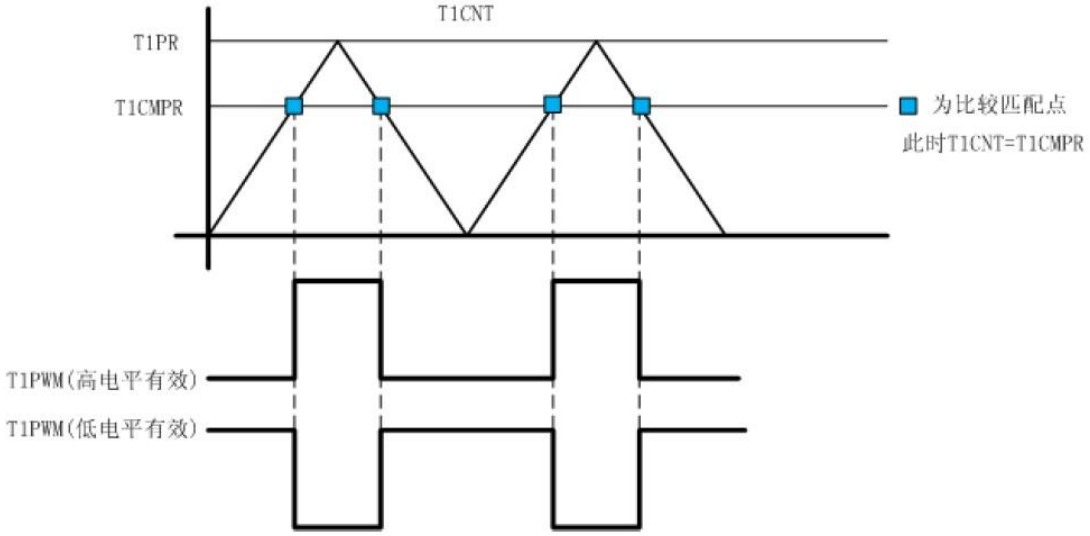
$$D=\frac{T1PR+1-T1CMPR}{T1PR+1}$$

当 T1PIN 为高电平有效时，则占空比为：

$$D=\frac{T1CMPR}{T1PR+1}$$

当 T1PIN 为低电平有效时，PWM 波形的占空比为：

2）当 T1CNT 为连续增/减计数时 ----- 对称的 PWM 波形。



当定时器 T1 工作于连续增/减计数模式。当 T1CNT 的值计数到和 T1CMPR 的值相等时，发生比较匹配事件。如果 T1CON 的第 1 位定时器比较使能为 TECMPR 为 1，即定时器比较操作被使能，且 GPTCONA 的第 6 位比较输出使能位 TCMPOE 为 1，同时 GPTCONA 下的 T1PIN 引脚输出极性为高电平或者低电平的话，T1PWM 就会输出对称的 PWM 波形。

当 T1 工作于连续增/减计数模式时，T1CNT 一个周期 T=（2*T1PR）*tc，其中 tc 是 T1CNT 计一次数所花的时间，

T1PWM 的周期为：$\frac{2*T1PR}{TCLK}$，单位为 s；

T1PWM 的频率为：$\frac{TCLK}{2*T1PR}$，单位为 Hz；

T1PWM 的占空比要分 GPTCONA 中 T1PIN 的输出极性，

当 T1PIN 为高电平有效时，则占空比为：

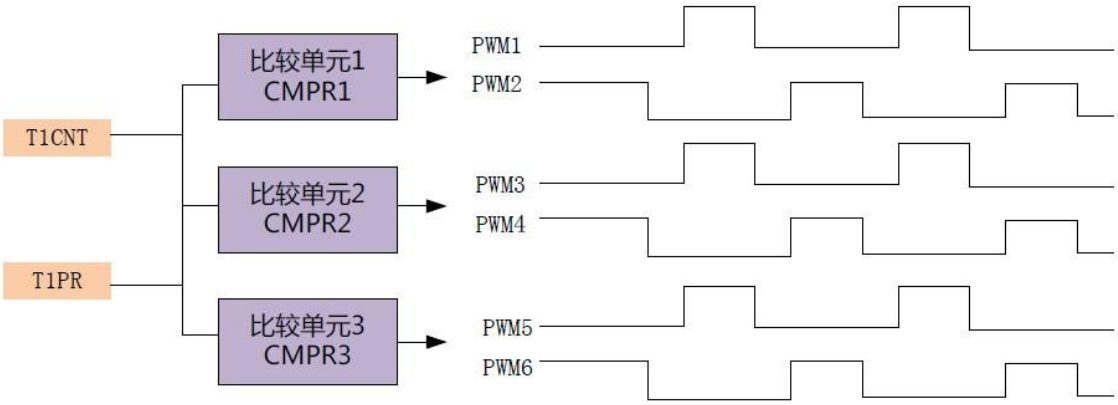
$$D=\frac{T1PR-T1CMPR}{T1PR}$$

当 T1PIN 为低电平有效时，PWM 波形的占空比为：

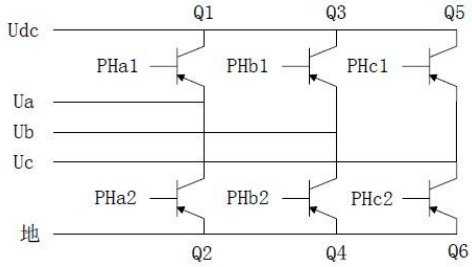
$$D=\frac{T1CMPR}{T1PR}$$

（3）比较单元产生的可带有死区的 PWM 波形

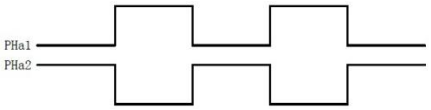
2812 的 EV 还为我们提供了 3 个全比较单元，分别是比较单元 1，比较单元 2 和比较单元 3。这 3 个全比较单元每一个都能产生一对互补的 PWM 波形，也可以通过相应的寄存器设置死区时间。这样，使得 EVA 和 EVB 都有能力去驱动一个三相全桥电路。



三相全桥电路



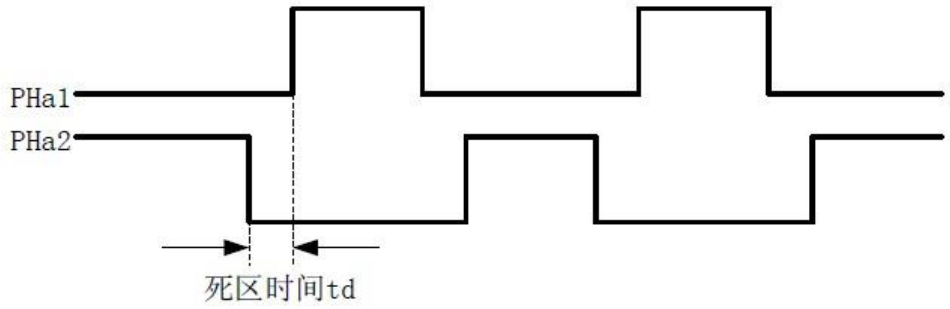
桥理想驱动波形



该电路由 6 个开关管组成，上下两个开关管组成 1 个桥壁。任何一个开关管在输入的 PWM 波形处于高电平时导通，处于低电平时关断。同一桥壁上的上下两个开关管不能同时导通，因为如果同时导通，电源和地就会短接，也就是会发生短路。因此，PHa1 和 PHa2，PHb1 和 PHb2，PHc1 和 PHc2 必须都是互补的，

以 PHa1 和 PHa2 为例，理想情况下当 PHa1 为高电平时，PHa2 为低电平；当 PHa1 为低电平时，PHa2 为高电平 PHa1 为高电平时 Q1 导通，此时 PHa2 为低电平，Q2 关闭，当 PHa1 从高电平转变为低电平时，Q1 由导通变为关断，而此时 Q2 由关断变为导通，实际上开关管从导通转为关断的时候，总会有延时，这样，就会有一小段时间里面其实 Q1 和 Q2 都处于导通状态，这样是非常危险的。为了解决这个问题，我们通常要求上下管输出的驱动波形要具有一定的死区时间，上下桥壁中任何一个开关管从关断到导通都要经过 1 个死区时间的延时

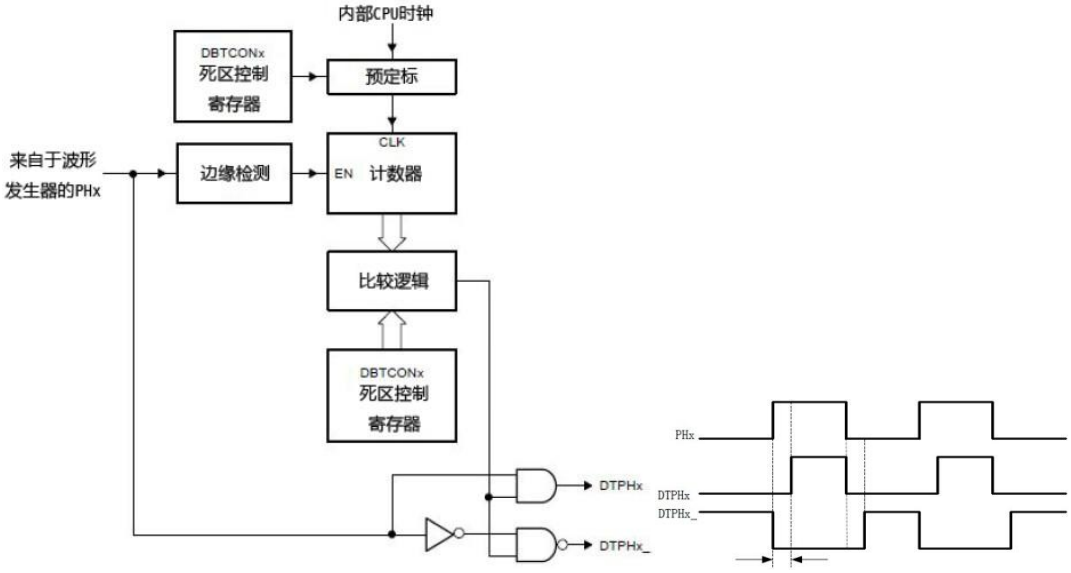
带死区 PWM 波形



比较单元产生 PWM 的波形和定时器通过比较功能产生 PWM 波形的原理是类似的。只不过定时器中的比较寄存器 T1CMPR，变成了比较单元的比较寄存器 CMPR1。三个比较单元都是类似的，以比较单元 1 为例。比较单元产生 PWM 时，所相关的寄存器有 T1PR,T1CNT,CMPR1,比较控制寄存器 COMCONA 和比较行为控制寄存器 ACTRA。

比较单元的时基是由 T1 来提供的，因此我们用到的是 T1PR 和 T1CNT，当 T1CNT 中的值和 CMPR1 中的值相等时，就发生了比较匹配。这时候，如果 COMCONA 的 CENABLE 为 1，即比较操作被使能，FCMPOE 为 1，比较输出时各路 PWM 波形都由相应的比较逻辑来驱动，同时如果 ACTRA 中 CMP1 和 CMP2 的极性为低电平或者高电平有效的时候，就会产生两路互补的 PWM 波形，PWM1 和 PWM2。 和 T1 产生 PWM 一样，当 T1 工作于连续增计数模式时，比较单元 1 输出不对称的 PWM 波形，而当 T1 工作于连续增减计数模式时，比较单元 1 输出对称的 PWM 波形。

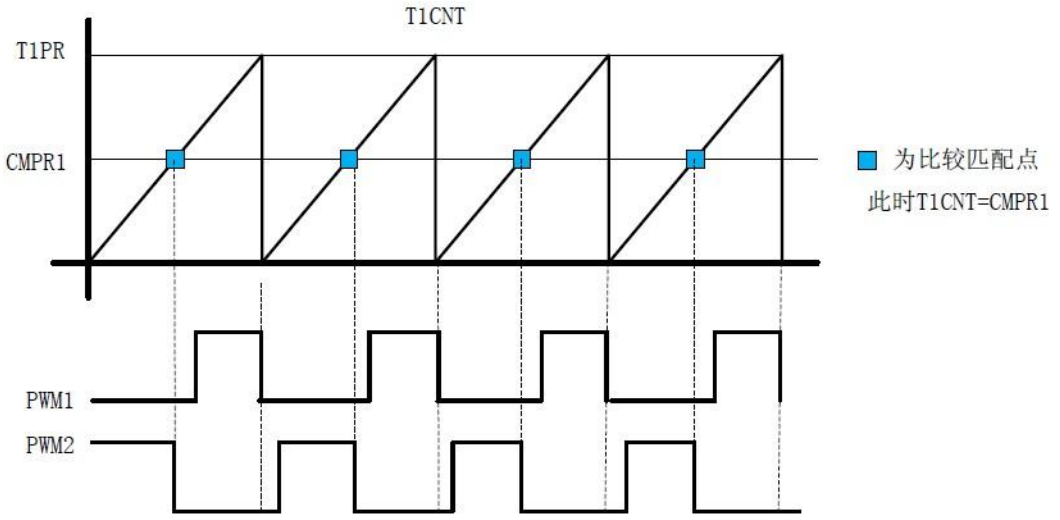
死区单元的模块图（x=1,2,3）



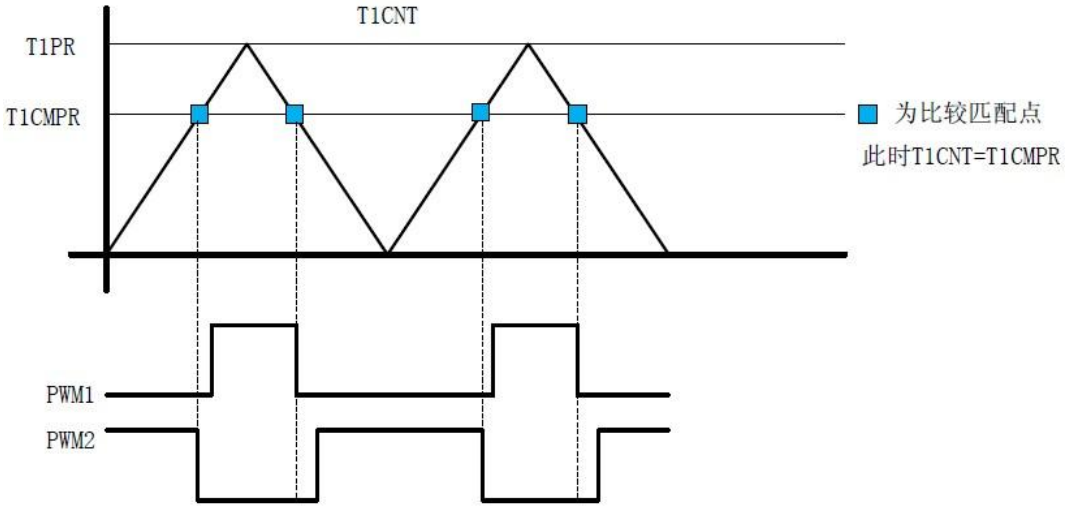
当比较单元的比较操作被使能，就会产生波形 PHx。PHx 经过死区单元，就会输出两路互补的带有死区的 PWM 波形 DTPHx 和 DTPHx_。PHx、DTPHx、DTPHx_之间的关系，如果没有死区，那么 DTPHx 和 DTPHx_应该是完全互补的。DTPHx 的导通时刻是在 PHx 的基础上延时了 1 个死区时间，而关闭时刻未变。DTPHx_是在 PHx 取反的基础上，也将导通时间延迟了 1 个死区时间，而关断的时间没有发生改变。

死区时间的控制，死区控制寄存器 DBTCONx 的[11~8]死区定时器周期和 DBTCON 的[4~2]位死区定时器预定标因子。如果死区定时器周期为 m，死区定时器预定标因子 x/p，则死区的值就为（p*m）个 CPU 时钟周期。

比较单元 1 产生的不对称 PWM 波形---带死区



比较单元 2 产生的对称 PWM 波形---带死区



DSP---PWM 波形源码

推　荐

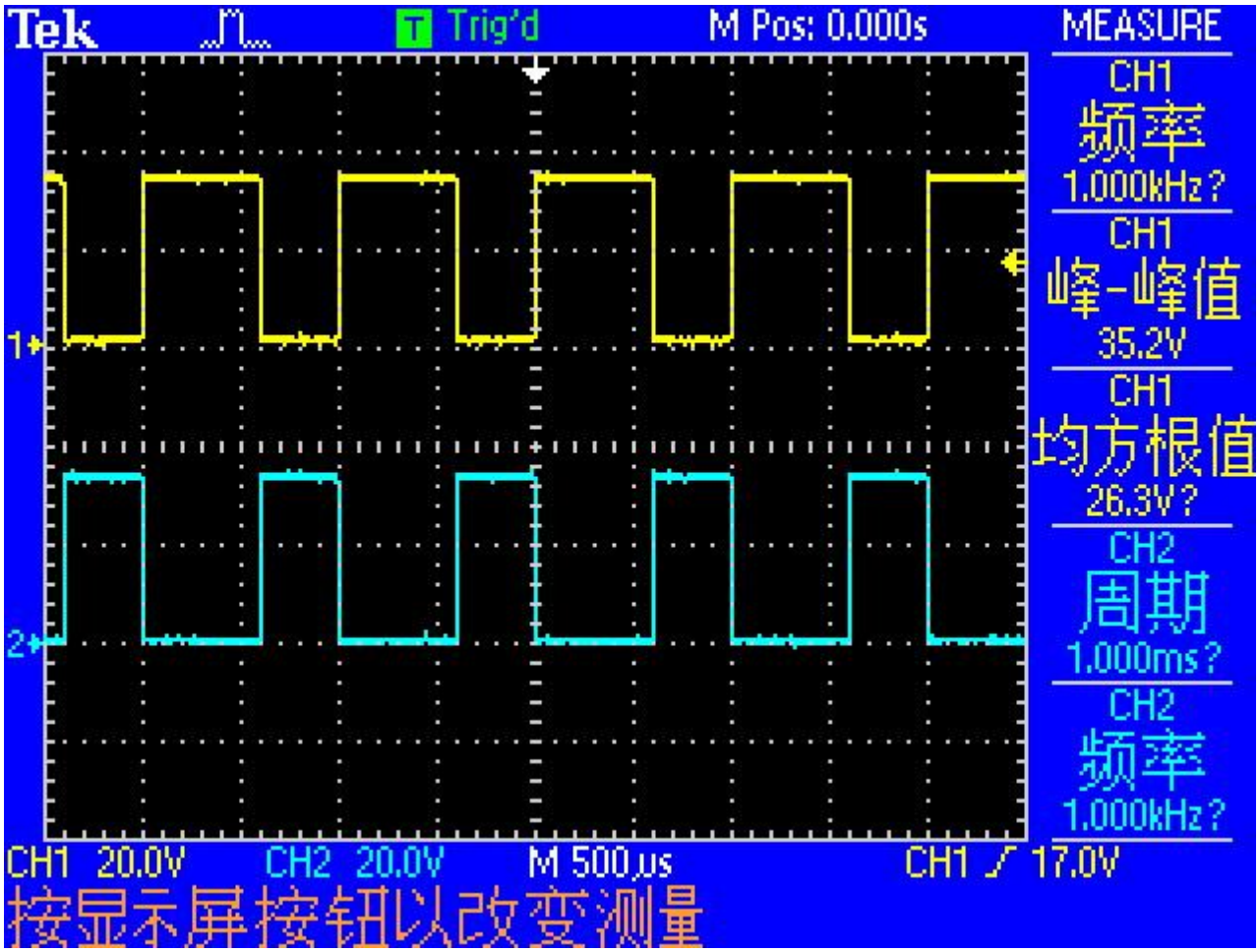
DSP2812---PWM 波形

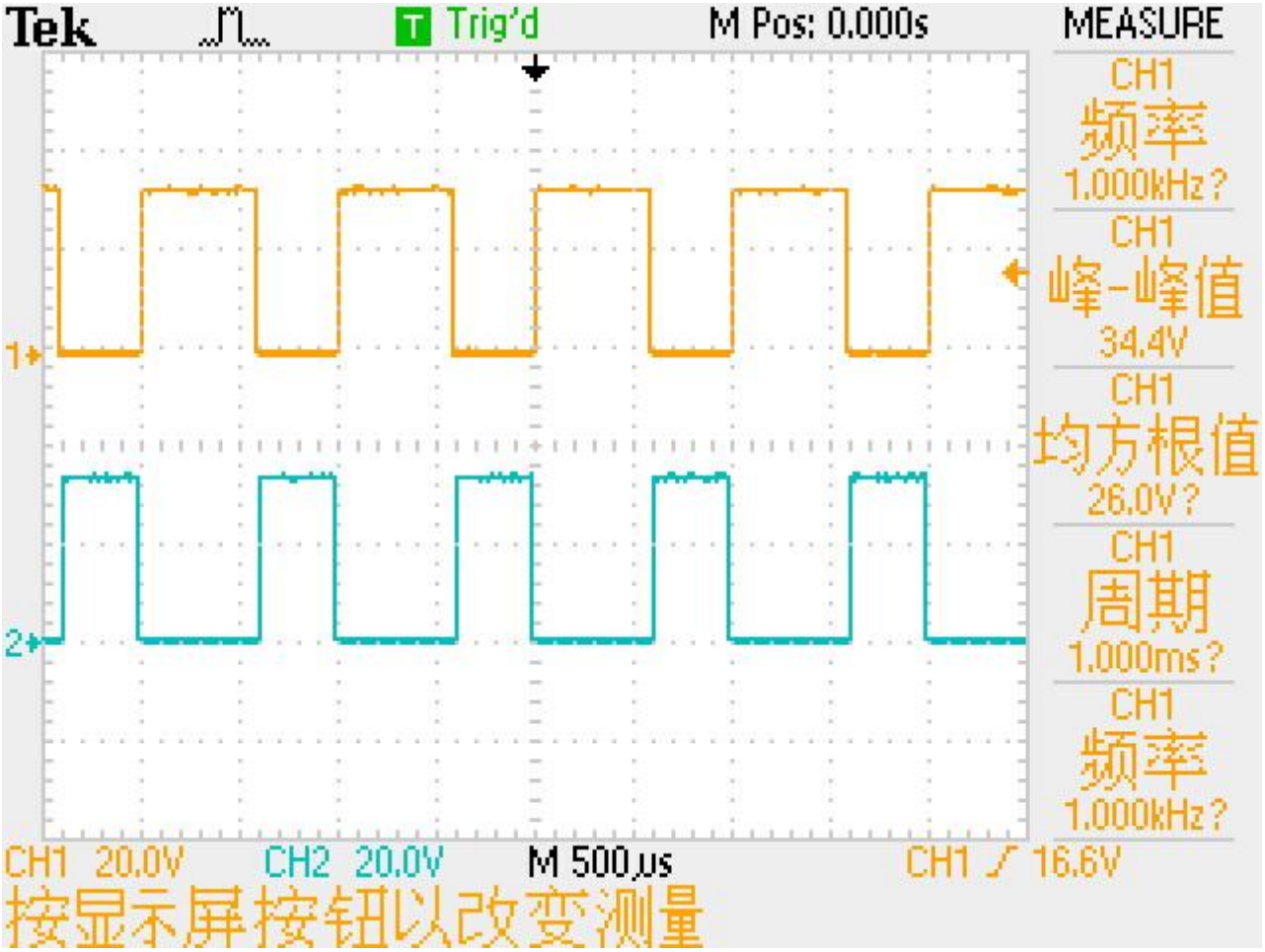
```
// $        Date:  6/11/2009      整理：  eyes417     $
//###########################################################################
//
//   FILE :  Example_281xEvPwm.c
//
//  TITLE:  EV-A  ---  PWM 波形程序
//
//                        ASSUMPTIONS:
//
//  As supplied, this project is configured for "boot to H0" operation.
//  Other then boot mode pin configuration, no other hardware configuration is required.
//
//###########################################################################


# include "DSP281x_Device.h"                    // DSP281x Headerfile Include File
# include "DSP281x_Examples.h"                  // DSP281x Examples Include File

void init_eva(void);

void  main(void)
{
  InitSysCtrl();

// InitGpio(); 配置 IO 口功能为 PWM 模式
   EALLOW;
  GpioMuxRegs.GPAMUX.all = 0x00FF;       // EVA PWM 1-6  pins
  EDIS;
```

```
  DINT;                                  //关 CPU 总中断
  InitPieCtrl();                         //初始化 PIE 控制寄存器
   IER = 0x0000;
    IFR = 0x0000;
    InitPieVectTable();          //初始化 PIE 中断向量表

   init_eva();                           //初始化 EV-A
  EvaRegs.T1CON.bit.TENABLE=1;    //手工启动定时器

   EINT;      // 使能 INTM(全局中断)
   ERTM;    // Enable Global realtime interrupt DBGM

  for(;;);

 }


//EV-A 初始化
void init_eva(void)
{
  EvaRegs.T1PR = 37500;    //周期值--连续增减时，PWM 频率=TCLK/(2*T1PR)---频率设为 1K, PWM=75M/（2
*37500）
  EvaRegs.T1CMPR = 0x3C00;           // Compare Reg--比较值
 EvaRegs.T1CNT = 0x0000;             //计数器初值


 //连续增/减模式，x/1 分频，内部时钟，使能比较，使用自己的周期,禁止定时器启动(等
 初始化全部完成后手工启动)
   EvaRegs.T1CON.all = 0x0802;
```

```
  EvaRegs.GPTCONA.bit.TCMPOE = 1;      //通过逻辑产生 T1 PWM
 EvaRegs.GPTCONA.bit.T1PIN = 1;       //GP 定时器 1 比较时低有效

//使能比较产生 1--6 PWM 波  1 个比较单元控制 2 路互补的 PWM 输出，控制 PWM 占空比
//连续增减--低有效时：PWM 占空比=CMPR1/T1PR,高有效时：PWM 占空比=(T1PR-CMPR1)/T1
PR
  EvaRegs.CMPR1 = 15000;              //第一路 PWM 占空比设为 0.4，0.4=15000/37500
  EvaRegs.CMPR2 = 0x3C00;
   EvaRegs.CMPR3 = 0xFC00;

// output pin 1 CMPR1 - 高有效，output pin 2 CMPR1 - 低有效
// output pin 3 CMPR2 - 高有效，output pin 4 CMPR2 - 低有效
// output pin 5 CMPR3 - 高有效，output pin 6 CMPR3 - 低有效
  EvaRegs.ACTRA.all = 0x0666;           //比较方式控制寄存器，控制 PWM 引脚的 高/
低 有效
  EvaRegs.DBTCONA.all = 0x0000;         //静止死区
  EvaRegs.COMCONA.all = 0xA600;         //比较控制寄存器--禁止空间矢量 PWM 模式

}
```

用示波器观察到的 PWM 波形


Tek
Trig'd
M Pos: 0.000s
MEASURE
CH1
频率
1.000kHz?
CH1
峰-峰值
35.2V
CH1
均方根值
26.3V?
CH2
周期
1.000ms?
CH2
频率
1.000kHz?
1
2
CH1 20.0V
CH2 20.0V
M 500µs
CH1 17.0V
按显示屏按钮以改变测量



总结：PWM 波形产生流程

1）：将 I/O 口设置为 PWM 引脚模式

2）：设置装载 TxCON，决定计数方式，启动比较操作

3）：设置装载 TxPR，决定 PWM 波形周期

4）：初始化 EvaRegs.CMPR1--3 的值,每个比较单元控制 2 路互补的 PWM 输出，控制 PWM 占空比

5）：EvaRegs.ACTRA 比较方式控制寄存器，控制 PWM 引脚的 高/低 有效

6）：EvaRegs.DBTCONA 死区时间的设置

7）：EvaRegs.COMCONA 设置比较控制寄存器

附：

```
/*****************************死区寄存器配置*********************************/
EvaRegs.DBTCONA.bit.DBT=5;                    //死区定时器周期为 5
EvaRegs.DBTCONA.bit.EDBT1=1;                  //死区定时器 1 使能
EvaRegs.DBTCONA.bit.DBTPS=3;                  //死区定时器预定标因子，死区时钟为 HSPCLK/8
```

HELLO 九：2812--SCI 模块

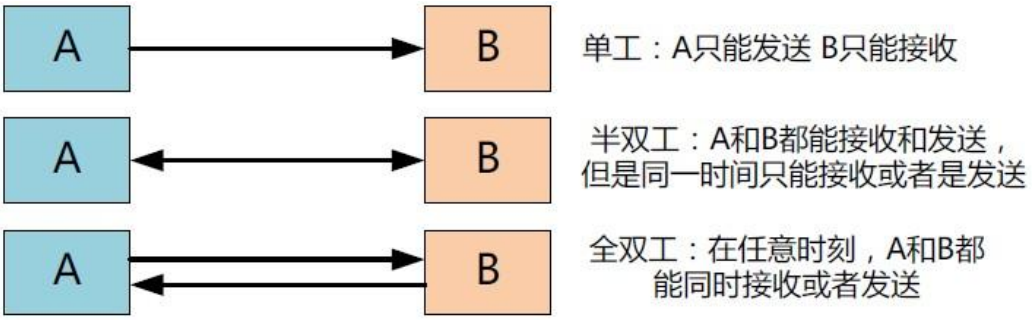
1

推　荐

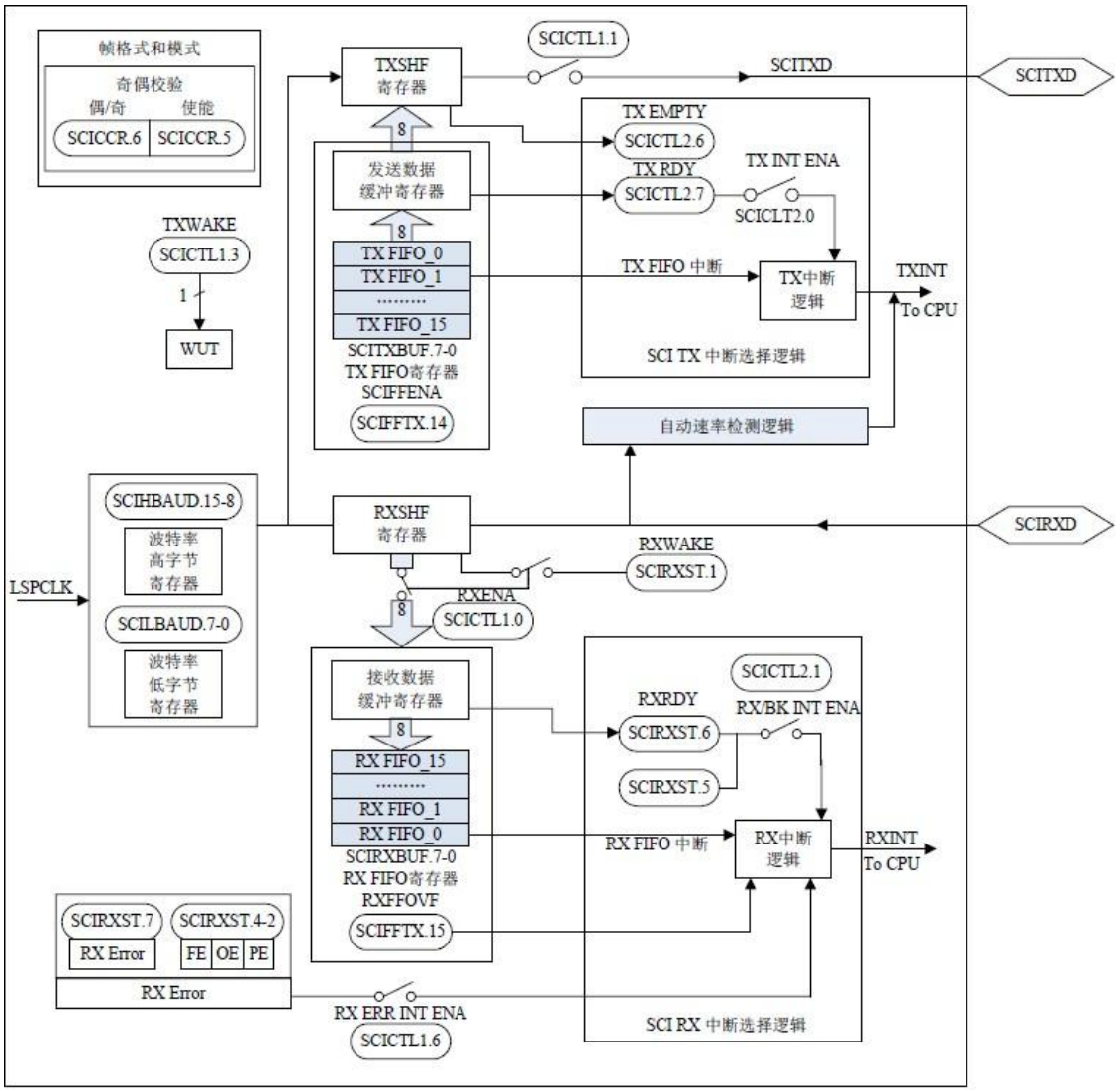
SCI（Serial Communication Interface），即串行通信接口，是一个双线的异步串口，即具有接收和发送两根信号线的异步串口，一般可以看作是 UART（通用异步接收/发送装置）。

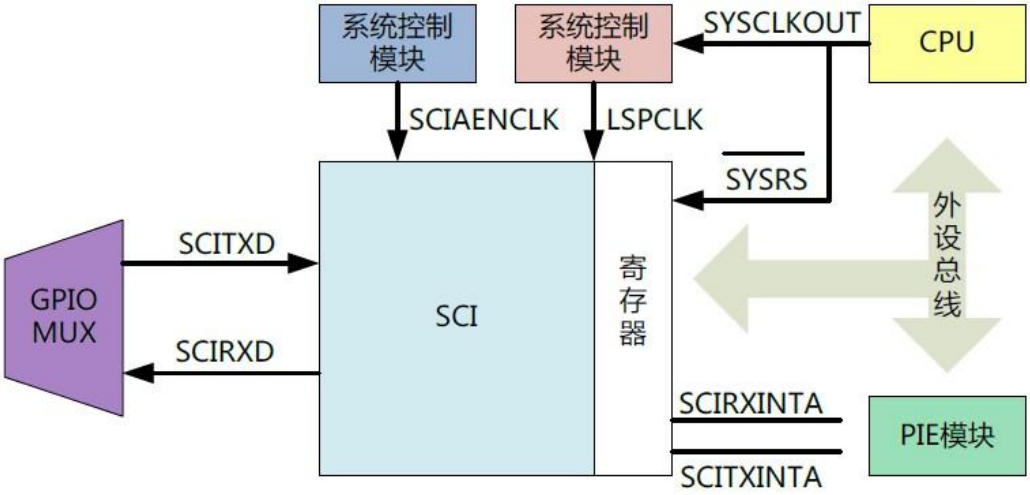
2812 的 SCI 模块支持 CPU 与采用 NRZ（non-return-to-zero 不归零）标准格式的异步外围设备之间进行数字通信。如果设计时我们的 SCI 使用的是 RS232 串行接口，那么，2812 就能和其他使用 RS232 接口的设备进行通信。例如 2812 内部的两个 SCI 之间，或者 2812 的 SCI 和其他 DSP 的 SCI 之间均能实现通信。

2812 内部具有两个相同的 SCI 模块，SCIA 和 SCIB，每一个 SCI 模块都各有一个接收器和发送器。SCI 的接收器和发送器各具有一个 16 级深度的 FIFO（First in fist out 先入先出）队列，它们还都有自己独立的使能位和中断位，可以在半双工通信中进行独立的操作，或者在全双工通信中同时进行操作。


A
B
单工：A只能发送 B只能接收
A
B
半双工：A和B都能接收和发送，
但是同一时间只能接收或者是发送
A
B
全双工：在任意时刻，A和B都
能同时接收或者发送

一. 2812-SCI 模块


帧格式和模式
奇偶校验
偶/奇
使能
SCICCR.6
SCICCR.5
TXWAKE
SCICTL1.3
1
WUT
TXSHF
寄存器
SCICTL1.1
SCITXD
SCITXD
8
发送数据
缓冲寄存器
8
TX FIFO_0
TX FIFO_1
.....
TX FIFO_15
SCITXBUF.7-0
TX FIFO寄存器
SCIFFENA
SCIFFTX.14
TX EMPTY
SCICTL2.6
TX RDY
SCICTL2.7
TX INT ENA
SCICLT2.0
TX FIFO 中断
TX中断
逻辑
TXINT
To CPU
SCI TX 中断选择逻辑
自动速率检测逻辑
SCIHBAUD.15-8
波特率
高字节
寄存器
LSPCLK
SCILBAUD.7-0
波特率
低字节
寄存器
RXSHF
寄存器
SCIRXD
RXWAKE
SCIRXST.1
RXENA
8
SCICTL1.0
接收数据
缓冲寄存器
8
RX FIFO_15
.....
RX FIFO_1
RX FIFO_0
SCIRXBUF.7-0
RX FIFO寄存器
RXFFOVF
SCIFFTX.15
SCICTL2.1
RXRDY
RX/BK INT ENA
SCIRXST.6
SCIRXST.5
RX FIFO 中断
RX中断
逻辑
RXINT
To CPU
SCIRXST.7
RX Error
SCIRXST.4-2
FE
OE
PE
RX Error
RX ERR INT ENA
SCICTL1.6
SCI RX 中断选择逻辑



SCI 模块具有两个引脚，SCITXDA 和 SCIRXDA，分别实现发送数据和接收数据的功能，这两个引脚对应于 GPIOF 模块的第 4 和第 5 位，在编程初始化的时候，需要将 GPIOFMUX 寄存器的第 4 和第 5 位置为 1，才能使得这两个引脚具有发送和接收的功能。SCIA 可以产生两个中断，SCIRXINTA 和 SCITXINTA，即发送中断和接收中断。

二. SCI 模块的特点

1 -- 具有 4 个错误检测标志：极性（parity）、溢出（overrun）、帧（framing）、中断（break）检测。

2 -- 多处理器模式下具有两种唤醒方式：空闲线方式和地址位方式。通常使用的时候很少遇到多处理器模式，我们采用的是空闲线方式。

3 -- 通信工作于半双工或者全双工模式。

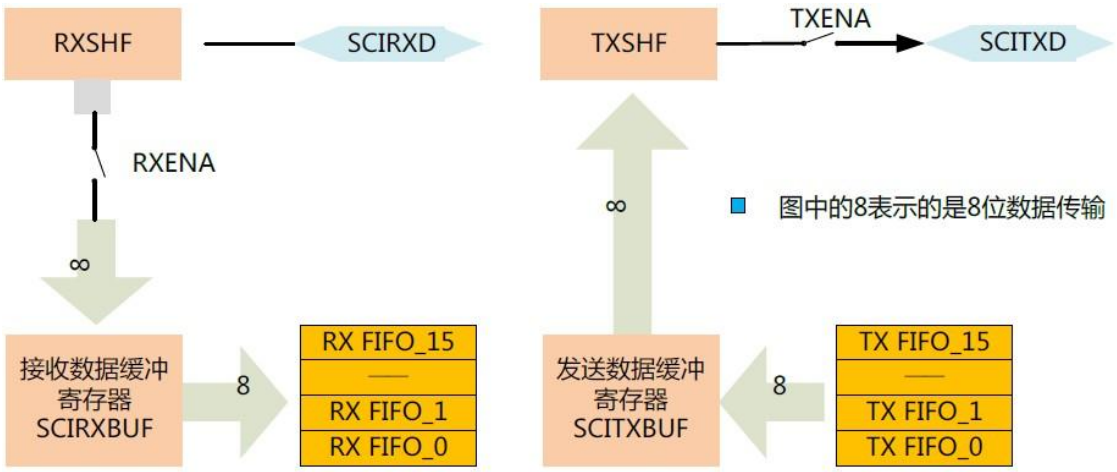
4 -- 具有双缓冲接收和发送功能，接收缓冲寄存器为 SCIRXBUF，发送缓冲寄存器为 SCITXBUF。

5 -- 发送和接收可以通过中断方式来实现，也可以通过查询方式来实现。

6 -- 具有独立的发送中断使能位和接收中断使能位。

7 -- SCIA 模块具有 13 个控制寄存器，值得注意的是，这些寄存器都是 8 位的寄存器，当某个寄存器被访问时，数据位于低 8 位，高 8 位为 0，因此，把数据写入高 8 位将是无效的。

三. SCI 模块发送和接收数据的工作原理



SCI 有独立的数据发送器和数据接收器，这样能够保证 SCI 既能够同时进行，也能够独立进行发送和接收的操作。

SCI 发送数据的过程如下：如图右半部分所示，在 FIFO 功能使能的情况下，首先，发送数据缓冲寄存器 SCITXBUF 从 TX FIFO 中获取由 CPU 加载的需要发送的数据，然后 SCITXBUF 将数据传输给发送移位寄存器 TXSHF，如果 SCI 的发送功能使能，TXSHF 则将接收到的数据逐位逐位的移到 SCITXD 引脚上。

SCI 接收数据的过程如下：如图的左半部分所示，首先，接收移位寄存器 RXSHF 逐位逐位的接收来自于 SCIRXD 引脚的数据，如果 SCI 的接收功能使能，RXSHF 将这些数据传输给接收缓冲寄存器 SCIRXBUF，CPU 就能从 SCIRXBUF 读取外部发送来的数据。当然，如果 FIFO 功能使能的话，SCIRXBUF 会将数据加载到 RX FIFO 的队列中，CPU 再从 FIFO 的队列读取数据

四. SCI 数据格式

在进行通信的时候，一般都会涉及到协议，所谓协议就是通信双方预先约定好的数据格式，以及数据的具体含义。这种事先约定好的规则，我们就把它叫做通信协议。

在 SCI 中，通信协议体现在 SCI 的数据格式上。通常将 SCI 的数据格式称之为可编程的数据格式，原因就是可以通过 SCI 的通信控制寄存器 SCICCR 来进行设置，规定通信过程中所使用的数据格式。SCI 使用的是 NRZ 的数据格式。

NRZ 数据格式

1 -- 1 个起始位

2 -- 1—8 个数据位

3 -- 1 个奇/偶/非极性位

4 -- 1—2 个结束位

5 -- 在地址位模式下，有 1 个用于区别数据或者地址的特殊位（仅用于多处理器通信）

真正的数据内容是 1—8 位，1 个字符的长度。我们通常将带有格式信息的每一个数据字符叫做一帧，在通信中常常是以帧为单位的。SCI 有空闲线模式和地址位模式，而在平常使用的时候，我们一般都是两个处理器之间的通信，例如 2812 和 PC 机或者 2812 和 2812 之间通信，这时候，更适合使用空闲线模式，而地址位模式一般用于多处理器之间的通信。在空闲线模式下，SCI 发送或者接收一帧的数据格式如图示，其中 LSB 是数据的最低位，MSB 是数据的最高位。

空闲线模式下 SCI 一帧的数据格式---具体的定义这些数据格式的寄存器是通信控制寄存器 SCICCR

起始位	LSB	2	3	4	5	6	7	MSB	奇/偶/无极性	结束位

五. SCI 通信波特率设置

所谓的波特率就是指每秒所能发送的位数。2812 的每个 SCI 都具有两个 8 位的波特率寄存器，SCIHBAUD 和 SCILBAUD，通过编程，可以实现达到 64K 不同的速率。

波特率的计算公式 ：$$BRR = \frac{LSPCLK}{SCI \quad Asynchronous \quad Baud * 8} - 1$$ （ 1 ≤ BRR ≤65535 ）

$$SCI \quad Asynchronous \quad Baud = \frac{LSPCLK}{16}$$ （ BRR=0 ）

BRR=波特率选择寄存器中的值，从十进制转换成十六进制后，高 8 位赋值给 SCIHBAUD，低 8 位赋值给 SCILBAUD

LSPCLK=37.5M 时，SCI 常见的波特率

理想波特率	BRR（十进制）	SCIHBAUD	SCILBAUD	精确波特率	误差（%）
2400	1952	0x7A	0	2400	0
4800	976	0x3D	0	4798	-0.04
9600	487	0x1	0xE7	9606	-0.06
19200	243	0	0xF3	19211	0.06
38400	121	0	0x79	38422	0.06

在进行通信的时候，双方都必须以相同的数据格式和波特率进行通信，否则通信会失败。例如 2812 和 PC 机上的串口调试软件进行通信时，2812 采用了什么样的数据格式和波特率，那么串口调试软件也需要设定成相同的数据格式和波特率，反之也一样。

六. SCI 发送和接收数据的机制

通常使用的有两种方式：一种是查询方式，另一种是中断方式。

查询方式：就是程序不断去查询状态标志位，看看 SCI 是不是已经做好了数据发送或者接收的准备。当数据发送时，需要查询的是位于 SCI 控制寄存器 2（SCICTL2）的第 7 为 TXREADY，发送器缓冲寄存器就绪标志。当这个位为 1 的时候，表明发送数据缓冲寄存器 SCITXBUF 已经准备好开始接收并发送下一个数据了。当数据写入 SCITXBUF，TXREADY 自动会清零，如果 TX ENA 使能了，发送移位寄存器 TXSHF 就会把 SCITXBUF 里面的数据发送出去。当数据接收时，需要查询的是 SCI 接收状态寄存器（SCIRXST）中的 RXRDY，接收器就绪标志。当从 SCIRXBU F 寄存器中已经准备好一个字符的数据，等待 CPU 去读时，RXRDY 位就会置 1。当数据被 CPU 从 SCIRXBUF 读出后，或者系统复位，都可以使 RXRDY 清 0。

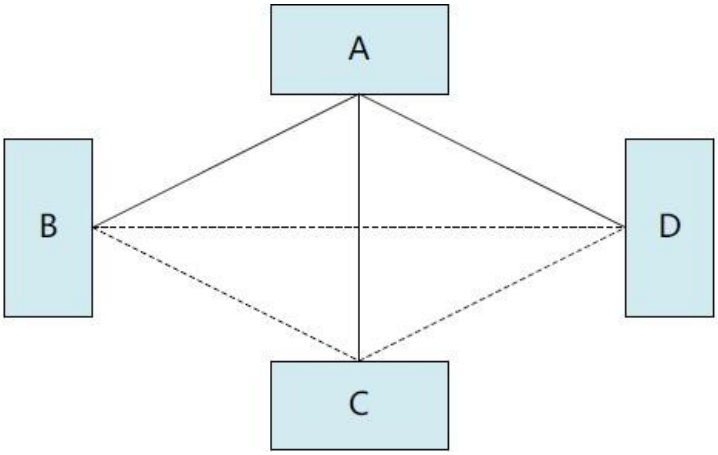
中断方式：如果需要使用中断，必须使能外设自己的中断、PIE 中断和 CPU 中断。SCIA 的发送和接收中断分别位于 PIE 模块第 9 组的第 1 和第 2 位，同时对应于 CPU 中断的 INT9。当前面所述的 TXRDY 也是个中断标志位，当该位置 1 时，就会产生发送中断事件，如果各级中断都已经使能，则会响应 SCI 的发送中断函数。当接收中断标志位 RXRDY 置 1 时，就会产生接收中断标志。如果各级中断已经使能，则会响应 SCI 的接收中断。值得提出来的是，我们在讲中断这一章内容的时候讲过，外设中断的标志位一定要手动复位，在这里 SCI 是个例外，原因如下：当发送器缓冲寄存器 SCITXBUF 做好准备发送数据时，TXRDY 置 1，但是当 CPU 将数据写入 SCITXBUF 的时候，TXRDY 自动会清 0。而当接收器缓冲寄存器已经准备好数据等待 CPU 去读取时，RXRDY 置 1，当 CPU 将数据从 SCIRXBUF 读出时，RXRDY 也会自动清 0。这样也就是说发送和接收这两个中断标志位都是可以自动清零的，所以无需手动复位。这是和其他外设中断不一样的地方，注意

查询函数位于主函数的 for 循环内，这样才能不断的去查询 TXRDY 和 RXRDY 的状态，但是很明显程序运行的效率比较低。但是查询方式较中断方式而言，比较简单。因此，可以建议使用如下的组合，数据接收采用中断方式，而数据发送采用查询方式。

七. 多处理器通信

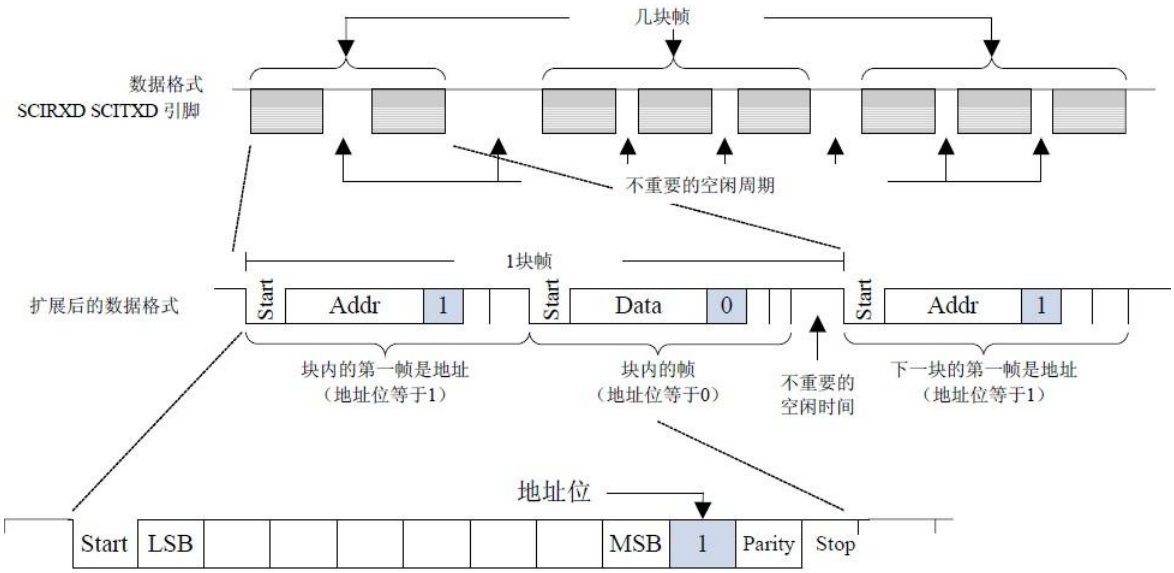
多处理器通信，顾名思义，就是多个处理器之间实现数据通信。一个简单的多处理器通信示意图如图 6 所示。在图中 A、B、C、D 之间都可以实现通信，实现表示处理器 A 和处理器 B、处理器 C、处理器 D 之间的通信。处理器 A 在同一时刻，只能和 B、C、D 之中的一个实现数据传输。当处理器 A 需要给 B、C、D 中的某一个处理器发送数据时，A-B，A-C，A-D 这 3 条通路上都会出现相同的数据，如何确保这些数据被正确的处理器接收呢？

如果我们给 A、B、C、D 事先分配好地址，然后 A 发送出去的信息里含有地址信息，B 或者 C 或者 D，在接收到这个数据信息的时候，先进行地址的核对，如果地址不符合，则不予响应。如果地址符合，则立即读取数据。这就是多处理器通信的基本原理。



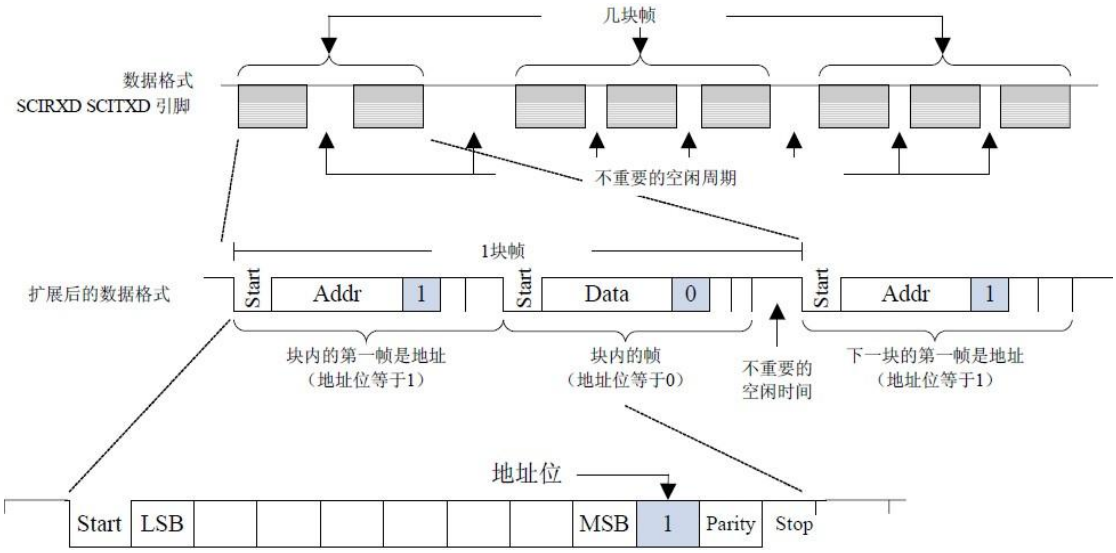
根据地址信息识别方法的不同，多处理器通信方式分为空闲线模式和地址位模式。

地址位多处理器通信格式



地址位多处理器通信 --- 当处理器 A 发出一串数据时，我们将其叫做数据块，由一个一个的帧构成。帧就是带有格式信息的字符数据。从图可以看到，某一个数据块中的第一帧是地址信息，第二帧是数据信息，然后留有一些空余空间之后，又有一个数据块，第一帧也是地址信息，后面是数据信息。我们可以看到，第一帧地址信息后面的一个位是 1，而第 2 帧数据信息后面的一个位是 0。我们把这一位称之为地址位，用于表示这个帧的数据时地址信息还是数据信息。像这样，在通信格式中加入地址位来判断信息是数据还是地址的方式叫做多处理器通信的地址位方式。

空闲线方式多处理器通信方式



空闲线方式多处理器通信 --- 块与块之间有一段比较长的空闲时间，这段时间要明显长于快内帧与帧之间的空闲时间。如果某个帧之后有一段 10 个位或者更长的空闲时间，那就表明新的数据块开始了。在某一个数据块中，第一帧代表地址信息，后面的帧为数据信息。也就是说，地址信息还是数据信息是通过帧与帧之间的空闲间隔来判断的。当帧与帧之间的空闲间隔超过 10 个位的时候，就表示新的数据块开始，而且其第一帧为地址信息。

空闲线模式中数据格式里没有额外的地址位，在处理 10 个字节以上的数据块时比地址位模式更为有效，被应用于典型的非多处理器 SCI 通信场合。而地址位模式由于有专门的位来进行识别地址信息，所以数据块之间不需要空闲时间等待，所以这种模式在处理一些小的数据块的时候更为有效，当然，我们也得看到，当传输数据的速度比较快，而程序执行速度不够快时，很容易中间会产生 10 位以上的空闲，这样其优势就更加不明显了。

SCI 查询方式

281x-SCI 和 PC 间的通讯

```
// $        Date:  30/11/2009      整理：  eyes417     $
//###########################################################################
//
//   FILE :  281x-SCI_PC.c
//
//  TITLE:  281x-SCI 和 PC 间的通讯  ---  查询方式，使用 FIFO
//           PC 发送至串口，DSP 再回送至 PC                    //
//
//                 ASSUMPTIONS:
//
//  As supplied, this project is configured for "boot to H0" operation.
//  Other then boot mode pin configuration, no other hardware configuration is required.
//
//###########################################################################


# include "DSP281x_Device.h"     // DSP281x Headerfile Include File
# include "DSP281x_Examples.h"   // DSP281x Examples Include File

void scia_echoback_init(void);
 void scia_fifo_init(void);
 void scia_xmit(int a);                    //串口发送--数据
void scia_msg(char *msg);                 //串口发送--字符串

Uint16 LoopCount;


void main(void)
 {
```

```
  Uint16 ReceivedChar;
    char *msg;

// Step 1. Initialize System Control:

  InitSysCtrl();

// Step 2. Initalize GPIO:
 // InitGpio();  //配置 IO 口功能为 SCIA 和 SCIB 模式

  EALLOW;
    GpioMuxRegs.GPFMUX.bit.SCITXDA_GPIOF4 = 1;
    GpioMuxRegs.GPFMUX.bit.SCIRXDA_GPIOF5 = 1;
    GpioMuxRegs.GPGMUX.bit.SCITXDB_GPIOG4 = 1;
    GpioMuxRegs.GPGMUX.bit.SCIRXDB_GPIOG5 = 1;
    EDIS;

// Step 3. Clear all interrupts and initialize PIE vector table:

  DINT;                    // Disable CPU interrupts

    InitPieCtrl();          //初始化 PIE 控制寄存器

  IER = 0x0000;

  IFR = 0x0000;

  InitPieVectTable();    //初始化 PIE 中断向量表

// Step 4. Initialize all the Device Peripherals:

  // InitPeripherals();    //初始化所有外设

// Step 5. User specific code:

  scia_fifo_init();

  scia_echoback_init();

//从 PC 上的串口调试助手输入要发送的数据至 DSP

  msg = "\r\nYou will enter a character, and the DSP will echo it back! \n\0";
    scia_msg(msg);

  for(;;)
    {
```

```
        msg = "\r\nEnter a character: \0";
        scia_msg(msg);

    // 查询方式--接收--使用 FIFO
    while(SciaRegs.SCIFFRX.bit.RXFIFST!=1) { }       // wait for XRDY =1 for empty state
                                //XRDY =1 (空状态)等待数据发送
    // 接收数据
    ReceivedChar = SciaRegs.SCIRXBUF.all;

    // Echo character back
    msg = "  You sent: \0";
        scia_msg(msg);
        scia_xmit(ReceivedChar);      //串口发送函数

     LoopCount++;

    }  //end for

} //end main



//SCI-A 初始化
void scia_echoback_init()
{
//通信控制寄存器,1 个停止位,无奇偶校验,自测试禁止,空闲线模式,字符长度 8 位
 SciaRegs.SCICCR.all =0x0007;

//禁止接收错误中断,禁止休眠,使能发送，接收
 SciaRegs.SCICTL1.all =0x0003;
  SciaRegs.SCICTL2.all =0x0003;
  SciaRegs.SCICTL2.bit.TXINTENA =1;    //使能 TXRDY 中断
 SciaRegs.SCICTL2.bit.RXBKINTENA =1; //接收缓冲器中断使能

//波特率设置 SCI-Band=LSPCLK/((BRR+1)*8)---9600K
 SciaRegs.SCIHBAUD = 0x0001;
  SciaRegs.SCILBAUD = 0x00E7;

 SciaRegs.SCICTL1.all =0x0023;  // SCI 初始化完成退出复位重启状态

}

//串口发送--变量
void scia_xmit(int a)
```

```
{
    while (SciaRegs.SCIFFTX.bit.TXFFST != 0) {}
    SciaRegs.SCITXBUF=a;

}

//串口发送--字符或字符串
 void scia_msg(char * msg)
 {
    int16 i;
    i = 0;
    while(msg[i] != '\0')
    {
        scia_xmit(msg[i]);
        i++;
    }
 }

// 初始化 SCI--FIFO
 void scia_fifo_init()
 {
    SciaRegs.SCIFFTX.all=0xE040;  //允许接收，使能 FIFO，清除 TXFIFINT
   SciaRegs.SCIFFRX.all=0x204f;  //使能 FIFO 接收，清除 RXFFINT,16 级 FIFO
   SciaRegs.SCIFFCT.all=0x0000;  //禁止波特率校验

}

//===========================================================================
// No more.
//===========================================================================
```