PMC232 系列数据手册

带 12 位ADC、采用FPPA™技术
双核心 8 位单片机

第 0.03 版
2013 年 08 月 08 日

10F-2, No. 1, Sec. 2, Dong-Da Road, Hsin-Chu 300, Taiwan, R.O.C.

TEL: 886-3-532-7598 www.padauk.com.tw

重要声明

应广科技保留权利在任何时候变更或终止产品，建议客户在使用或下单前与应广科技或代理商联系以取得最新、最正确的产品信息。

应广科技不担保本产品适用于保障生命安全或紧急安全的应用，应广科技不为此类应用产品承担任何责任。关键应用产品包括，但不仅限于，可能涉及的潜在风险的死亡，人身伤害，火灾或严重财产损失。

应广科技不承担任何责任来自于因客户的产品设计所造成的任何损失。在应广科技所保障的规格范围内，客户应设计和验证他们的产品。为了尽量减少风险，客户设计产品时，应保留适当的产品工作范围安全保障。

提供本文档的中文简体版是为了便于了解，请勿忽视文中英文的部份，因为其中提供有关产品性能以及产品使用的有用信息，应广科技暨代理商对于文中可能存在的差错不承担任何责任，建议参考本文件英文版。

目 录

修订历史：

修 订	日 期	描 述
0.01	2012/11/30	初版
0.02	2013/03/31	1. 修订 page 7 : PMC232 和 P232 差异表 2. 修订page 14 : f_{IHRC} -> IHRC频率* (校准后)数值 3. 修订 page 30 : 5.7.1 项 ： 总频率漂移约为±8% 4. 修订所有 LVD 为 LVR 5. 修订 Band-Gap 参考电压生成器 6. 修订第 6.14~ 6.22 项初始值 8'hx00 为 0x00 7. 统一位分辨率为 12bit 8. Page47 : 当 EOSC 被选用当系统时钟后，快速换醒就自动关闭 9. 移除封装信息
0.03	2013/08/08	1. 增加输入电压(Input voltage) 与脚位的引入电流(Injected current on pin) 于直流/交流电气特性表中 2. 增加封装类型 SOP16 于项目 1.2 及项目 3. 3. 增加 SOP18/DIP18 脚位图 4. 修改项目 5.16.2 :ADC 的分辨率为 12 位 5. 移除 PMC232 和 P232 差异表的"快速复原 (Fast recover) " 项目并修改 P232C 和 PMC232 主要差异表 6. 增加 SOP16- A, B 封装方式 (购买信息)

P232C 和 PMC232 主要差异表

P232C 与 PMC232 主要差异列举如下：

项目	功能	P232C	PMC232
1	IO 输出电流	12mA@5.0V	10mA@5.0V
2	SRAM	200 bytes	160 bytes
3	Band-gap	+/- 200mV(@1.20V)	+/- 30mV(@1.20V) 校准后
4	LVR	4 段 LVR 设定	8 段 LVR 设定
5	单一处理器模式	支持	不支持
6	LCD VDD/2 偏置电压	没有	有
7	ADC 参考高电压	VDD 与 PB1	VDD
8	ADC 分辨率	8bit 到 12bit 可供选择	只有 12bit
9	端口数字/模拟输入编译寄存器	***padidr*** , ***pbdidr***, ***pcdidr***	***padier*** , ***pbdier***
10	IHRC 选择性指令	.ADJUST_OTP_IHRCR	.ADJUST_IIC
11	看门狗定时器溢时	512 ILRC 时钟周期	4 个周期可供选择
12	硬件比较器	有	没有

P232C 转 PMC232 程序

关于 P232C 转 PMC232，请参考下列流程：

1. 将 PMC232 的规格书和使用手册阅览一遍。
2. 用 Source Code 在工程文件“.pre”中将“.chip P232CXXX”直接改为“.chip PMC232”。
3. 按下“Build”键，之后 IDE 会出现错误和警告。
4. 逐一修改相关 Source Code 直到错误”error”讯息不再出现。
5. 烧录 realchip 在 PCB 上作功能的详细测试。
6. 如有需要，修改程序后回到第 3 步骤重新操作。
7. 如果您仍有任何问题，请联络我们的 FAE : fae@padauk.com.tw

1. 单片机特点

1.1. 高性能RISC CPU 架构

- 工作模式：2 个FPPA™处理单元运作模式或传统单一处理单元运作模式
- 2Kx16 bits OTP 程序存储器
- 160 Bytes 数据存储器
- 提供 100 条指令
- 大部份指令都是单周期（1T）指令
- 弹性化的堆栈深度，可程序设定
- 提供数据与指令的直接、间接寻址模式
- 所有的数据存储器都可当数据指针（index pointer）
- 独立的 IO 地址以及存储地址，方便程序开发

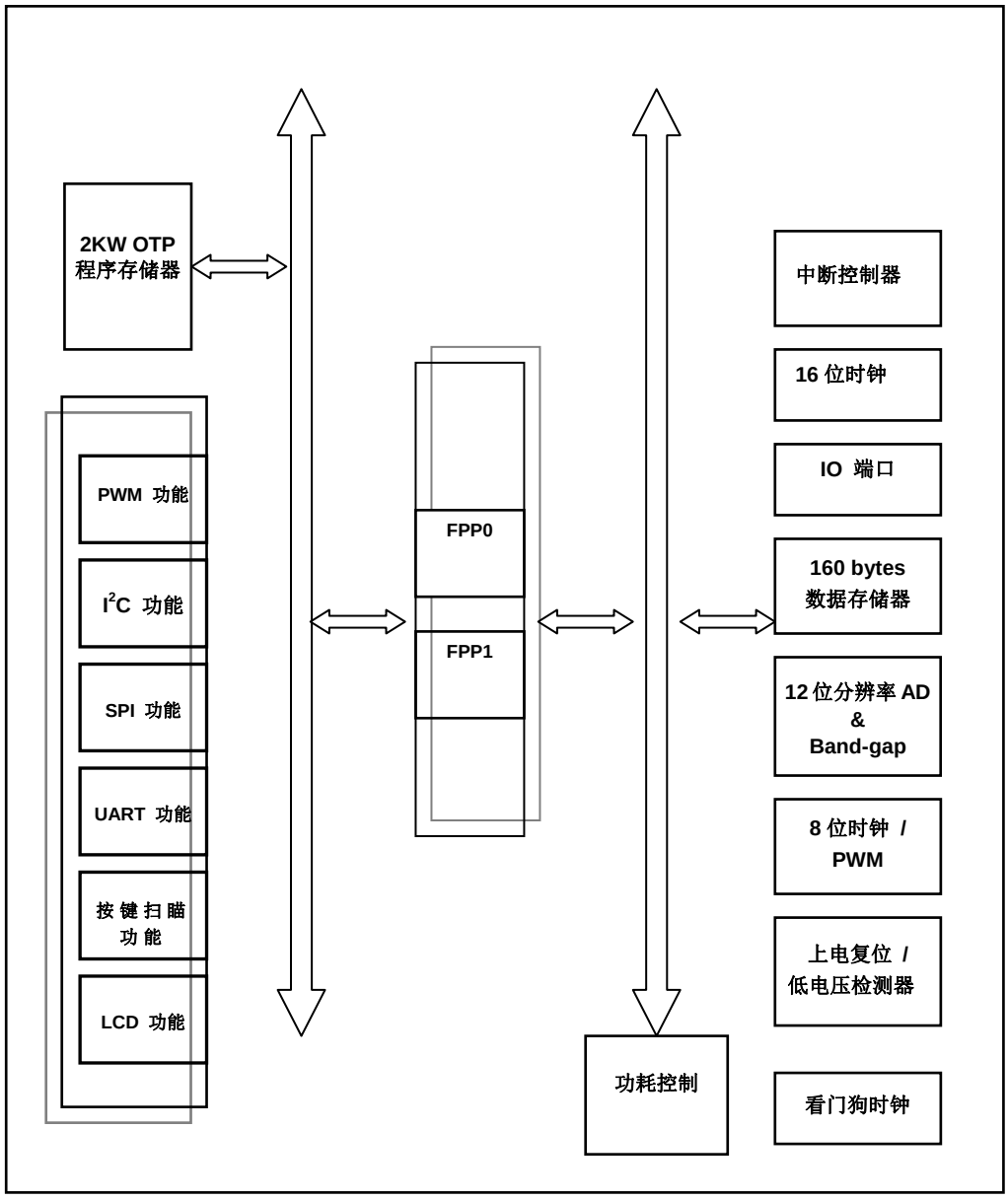
1.2. 系统功能

- 时钟源：内部高频 RC 振荡器（IHRC）、内部低频 RC 振荡器（ILRC）、外部晶振
- 内置 Band-gap 硬件模块输出 1.20V 参考电压
- 内置一个硬件 16 位定时器
- 内置一个硬件 8 位定时器并可提供 PWM 模式输出
- 内置一个 10 通道 12 位分辨率 A/D 转换器，其中 1 通道是 Band-gap 参考电压输入
- 内置 VDD/2 偏置电压产生器供液晶显示应用
- 最多提供 4x13 点 LCD 显示
- 提供快速唤醒模式
- 8 段 LVR 设定~ 4.1V, 3.6V, 3.1V, 2.8V, 2.5V, 2.2V, 2.0V, 1.8V
- 18 个 IO 引脚，每一 IO 引脚具有 10mA 电流驱动能力
- 两个外部中断引脚
- 每一 IO 引脚都可以单独设置系统唤醒功能
- 工作电压：2.2V ~ 5.5V
- 工作温度：-40°C ~ 85°C
- 工作频率（合并 2 个 FPP 处理单元）
  DC ~ 8MHz@VDD≧3.3V; DC ~ 4MHz@VDD≧2.5V; DC ~ 2MHz@VDD≧2.2V
- 功耗特性：
  $I_{operating}$ ~ 1.7mA@1MIPS, VDD=5.0V;　　$I_{operating}$ ~ 15uA@VDD=3.3V, ILRC ~ 12kHz
  $I_{powerdown}$ ~ 1uA@VDD=5.0V;　　$I_{powerdown}$ ~ 0.5uA@VDD=3.3V
- 购买信息：
  PMC232-S20: SOP20 (300mil);　　PMC232-S14: SOP14 (150mil);
  PMC232-D20: DIP20 (300mil);　　PMC232-D14: DIP14 (300mil);
  PMC232-S16A: SOP16 Type A (150mil)　　PMC232-S16B: SOP16 Type B (150mil)
  PMC232-S18: SOP18 (300mil);　　PMC232-D18: DIP18 (300mil);

2. 系统概述和方框图

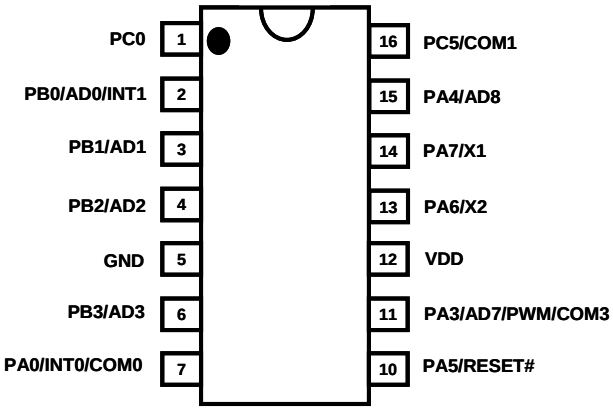
PMC232 系列是一个带 ADC、并行处理、完全静态，以 OTP 为程序存储基础的处理器，此处理器具有两个处理单元。它基于 RISC 架构，获得（Field Programmable Processor Array 现场可编程处理器阵列）技术专利，大多数的指令执行时间都是一个指令周期。

在 PMC232 内部有 2K X 16bit OTP 程序存储器以及 160 Bytes 数据存储器供两个 FPP 处理单元运算使用，芯片内部还设置有 10 通道 12 位分辨率 A/D 转换器，其中 1 通道为内置的 Band-gap 参考电压生成器，它可以提供于绝对电压的测量；另外，PMC232 提供 2 组硬件时钟，一个为 16 位时钟，第二个为 8 位时钟并且可产生 PWM 波形。



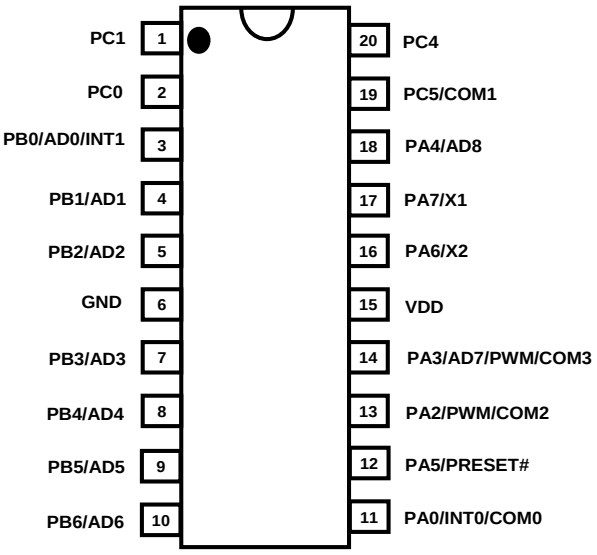
3. PMC232 系列引脚功能描述

PMC232-S14 (SOP14-150mil)
PMC232-D14 (DIP14-300mil)



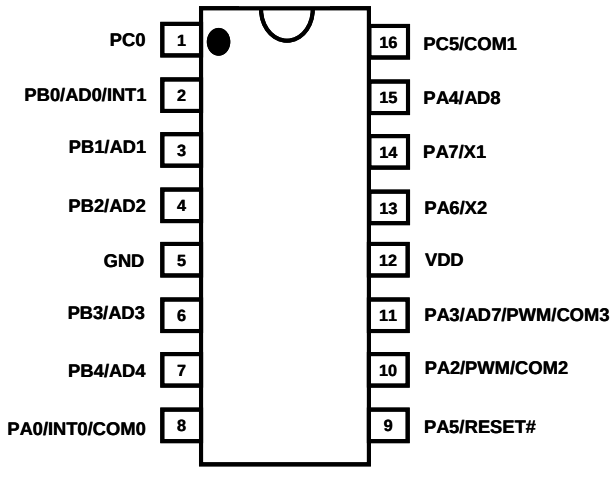
PMC232-S14 (SOP14-150mil)
PMC232-D14 (DIP14-300mil)

PMC232-S20 (SOP20-300mil)
PMC232-D20 (DIP20-300mil)



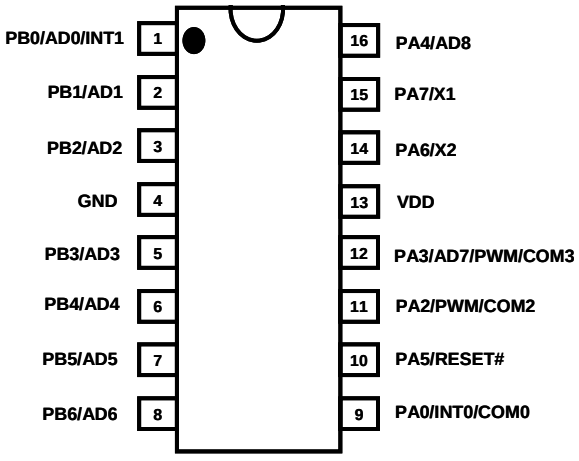
PMC232-S20 (SOP20-300mil)
PMC232-D20 (DIP20-300mil)

PMC232-S16A (SOP16-150mil)



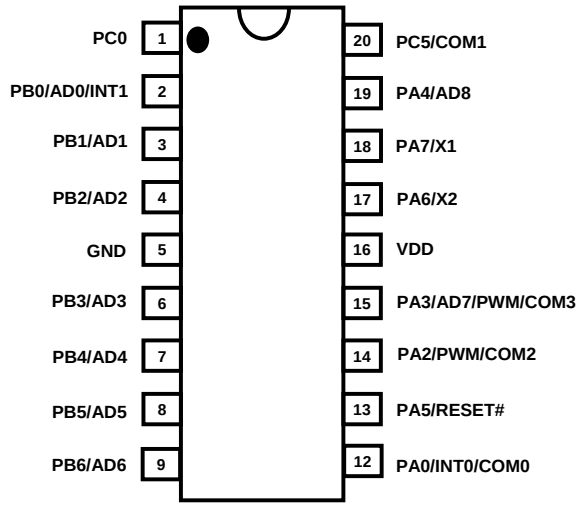
PMC232-S16A (SOP16-150mil)

PMC232-S16B (SOP16-150mil)



PMC232-S16B (SOP16-150mil)

PMC232-S18 (SOP18-300mil)
PMC232-D18 (DIP18-300mil)



PMC232-S18 (SOP18-300mil)
PMC232-D18 (DIP18-300mil)

引脚功能说明

引脚名称	电器型态	功能描述
PA7/X1	IO ST / CMOS	此引脚可用作： 1. 当使用内部 IHRC 振荡器时，它可以当 Port A 位 7，并可编程设定为数字输入/输出，弱上拉电阻。 2. 使用晶体振荡器时，作 X1 用。 当此引脚设定为晶体振荡功能时，请用寄存器 *padier* 位 7 关闭（”0”）此引脚的数字输入以减少漏电流。当此引脚设定禁用数字输入，在掉电模式的唤醒功能将同时被禁用。
PA6/X2	IO ST / CMOS	此引脚可用作： 1. 当使用内部振荡器 IHRC 或 ILRC 时，它可以当 Port A 位 6，可编程设定为数字输入/输出，弱上拉电阻。 2. 使用晶体振荡器时，作 X2 用。 当此引脚设定为晶体振荡功能时，请用寄存器 *padier* 位 6 关闭（”0”）此引脚的数字输入以减少漏电流。当此引脚设定禁用数字输入，在掉电模式的唤醒功能将同时被禁用。
PA5/ RESET#	IO (OC) ST / CMOS	此引脚可用作： 1. 当单片机的硬件外部复位。 2. 当 Port A 位 5；此引脚没有上拉或下拉电阻，当设定为输出时，只能输出低电位（开漏输出 open drain），输出高电位需要外加上拉电阻。 如果此引脚没有用的话，请外加上拉电阻（输入模式）或将它驱动低（输出），以防止漏电流。另外，可以用寄存器*padier*位 5 来关闭（”0”）此引脚在掉电时的唤醒功能。 当做为输入时，请在靠近I/O端串接 33Ω电阻用来抗干扰

引脚名称	电器型态	功能描述
PA4/ AD8	IO ST / CMOS / Analog	此引脚可用作： 1. Port A 位 4，这个引脚可编程设定为数字输入/输出，弱上拉电阻。 2. ADC 模拟输入通道 8。 当此引脚设定为模拟输入时，请用寄存器*padier*位 4 关闭（”0”）此引脚的数字输入以减少漏电流。当此引脚设定禁用数字输入，在掉电模式的唤醒功能将同时被禁用。
PA3/ AD7/ PWM/ COM3	IO ST / CMOS / Analog	此引脚可用作： 1. Port A 位 3，这个引脚可编程设定为数字输入/输出，弱上拉电阻。 2. ADC 模拟输入通道 7 3. Timer2 的 PWM 输出 4. 产生 COM3 的(VDD/2)偏置电压供给 LCD 应用。 当此引脚设定为模拟输入时，请用寄存器*padier*位 3 关闭（”0”）此引脚的数字输入以减少漏电流。当此引脚设定禁用数字输入，在掉电模式的唤醒功能将同时被禁用。
PA2/ PWM/ COM2	IO ST / CMOS	此引脚可用作： 1. Port A 位 2，这个引脚可编程设定为数字输入/输出，弱上拉电阻。 2. Timer2 的 PWM 输出。 3. 产生 COM2 的(VDD/2)偏置电压给 LCD 应用。 另外，可以用寄存器 *padier* 位 2 来关闭（”0”）此引脚在掉电时的唤醒功能。
PA0/ INT0/ COM0	IO ST / CMOS	此引脚可用作： 1. Port A 位 0，这个引脚可编程设定为数字输入/输出，弱上拉电阻。 2. 外部中断输入，中断服务可发生在上升沿或下降沿。 3. 产生 COM0 的(VDD/2)偏置电压给 LCD 应用。 另外，可以用寄存器 *padier* 位 0 来关闭（”0”）此引脚在掉电时的唤醒功能。
PB6/AD6 PB5/AD5 PB4/AD4 PB3/AD3 PB2/AD2 PB1/AD1	IO ST / CMOS / Analog	此引脚可用作： 1. Port B 位 6 ~ 1，这 6 个引脚可以编程设定为数字输入、高低电位输出，弱上拉电阻也可独立设定。 2. ADC 模拟输入通道 6 ~ 1。 当此 6 个引脚设定为模拟输入时，请用寄存器*pbdier*关闭（”0”）此引脚的数字输入以减少漏电流。当引脚设定禁用数字输入，在掉电模式的唤醒功能将同时被禁用。 当 PB2 做为 ADC 输入时，请加一个 0.1uF 电容在上面。
PB0/ AD0/ INT1	IO ST / CMOS / Analog	此引脚可用作： 1. Port B 位 0，这个引脚可以编程设定为数字输入、高低电位输出，弱上拉电阻也可独立设定。 2. ADC 模拟输入通道 0 3. 外部中断输入，中断服务可靠寄存器设定选择在上升沿或下降沿。 当此引脚设定为模拟输入时，请用寄存器*pbdier*关闭（”0”）此引脚的数字输入以减少漏电流。当此引脚设定禁用数字输入，在掉电模式的唤醒功能将同时被禁用。
PC5/ COM1	IO ST / CMOS	此引脚可用作： 1. Port C 位 5，这个引脚可以编程设定为数字输入、高低电位输出，弱上拉电阻也可独立设定。 2. 产生 COM1 的(VDD/2)偏置电压给 LCD 应用。

引脚名称	电器型态	功能描述
PC4 PC1	IO ST / CMOS	Port C 位 4, 1。这 2 个引脚可以编程设定为数字输入、高低电位输出，弱上拉电阻也可独立设定。
PC0	IO ST / CMOS	此引脚可用作： Port C 位 0，这个引脚可以编程设定为数字输入、高低电位输出，弱上拉电阻也可独立设定。
VDD		正电源
GND		地

注意： IO: 输入/输出; ST: 施密特触发；OC: 开漏输出； Analog: 模拟输入 CMOS: CMOS 电压准位

4. 器件电气特性

4.1. 直流/交流特性

下列所有数据除特别列明外，皆于Ta = -40 °C ~ 85 °C, Vdd=5.0V, f_{SYS}=2MHz之条件下获得。

符 号	特 性	最小值	典型值	最大值	单位	条 件 （Ta=25℃）
V_{DD}	工作电压	2.2	5.0	5.5	V	
f_{SYS}	系统时钟* IHRC/2 IHRC/4 IHRC/8 ILRC	 0 0 0 	 24K	 8M 4M 2M 	Hz	Under_20ms_Vdd_ok**= Y/N V_{DD}≧2.5V / V_{DD}≧3.1V V_{DD}≧2.2V / V_{DD}≧2.5V V_{DD}≧2.2V / V_{DD}≧2.2V V_{DD} = 5.0V
I_{OP}	工作电流		1.7 15		mA uA	f_{SYS}=IHRC/16=1MIPS@5.0V f_{SYS}=ILRC=12kHz@3.3V
I_{PD}	掉电电流 (使用 ***stopsy***s 指令)		1.0 0.5		uA uA	f_{SYS}= 0Hz,VDD=5.0V f_{SYS}= 0Hz,VDD=3.3V
I_{PS}	省电电流 (使用 ***stopexe*** 指令)		0.3		mA	VDD=5.0V; Band-gap, LVR, IHRC, ILRC, Timer16 硬件模块启用.
V_{IL}	输入低电压	0		$0.3V_{DD}$	V	
V_{IH}	输入高电压	$0.7 V_{DD}$		V_{DD}	V	
I_{OL}	IO 引脚灌电流	7	10	13	mA	V_{DD}=5.0V, V_{OL}=0.5V
I_{OH}	IO 引脚驱动电流	-5	-7	-9	mA	V_{DD}=5.0V, V_{OH}=4.5V
V_{IN}	输入电压	-0.3		VDD+0.3	V	
$I_{INJ (PIN)}$	脚位的引入电流			1	mA	VDD+0.3≧V_{IN}≧ -0.3
R_{PH}	上拉阻抗		62 100 210		KΩ	V_{DD}=5.0V V_{DD}=3.3V V_{DD}=2.2V
V_{LVR}	低电压复位电压	3.86 3.35 2.84 2.61 2.37 2.04 1.86 1.67	4.15 3.60 3.05 2.80 2.55 2.20 2.00 1.80	4.44 3.85 3.26 3.00 2.73 2.35 2.14 1.93	V	
V_{BG}	Band-gap参考电压 (校准前)	1.12	1.20	1.28	V	VDD=5V, 25°C
	Band-gap参考电压 (校准后)	1.17*	1.200*	1.23*		VDD=2.2V ~ 5.5V, -40°C <Ta<85°C*

符 号	特 性	最小值	典型值	最大值	单位	条 件 （Ta=25℃）
f_{IHRC}	IHRC 频率* (校准后)	15.76*	16*	16.24*	MHz	25°C, VDD=2.2V~5.5V
		14.72*	16*	17.28*		VDD=2.2V~5.5V, -40°C<Ta<85°C*
		15.04*	16*	16.96*		VDD=2.2V~5.5V, 0°C <Ta<70°C*
f_{ILRC}	ILRC 频率 *	20.4*	24*	27.6*	kHz	VDD=5.0V, Ta=25°C
		15.6*	24*	32.4*		VDD=5.0V, -40°C <Ta<85°C*
		10.2*	12*	13.8*		VDD=3.3V, Ta=25°C
		4.55*	7*	9.45*		VDD=2.2V, -40°C <Ta<85°C*
t_{INT}	中断脉冲宽度	30			ns	VDD= 5.0V
V_{ADC}	ADC的可工作电压	2.5		5.0	V	
V_{AD}	AD 输入电压	0		VDD	V	
ADrs	ADC 分辨率			12	bit	
ADcs	ADC 消耗电流		0.9 0.8		mA	@5V @3V
ADclk	ADC 工作时钟周期		2		us	2.5V ~ 5.5V
t_{ADCONV}	ADC 转换时间 (T_{ADCLK}是选定AD转换时钟周期)		13 14 15 16 17		T_{ADCLK}	8 位分辨率 9 位分辨率 10 位分辨率 11 位分辨率 12 位分辨率
AD DNL	AD 微分非线性		±2*		LSB	
AD INL	AD 积分非线性		±4*		LSB	
ADos	AD失调电压 (offset)		3		mV	
V_{DR}	数据存储器数据维持电压*	1.5			V	PMC232 在待机模式下
$R_{(VDD/2)}$	(VDD/2) 拉高/拉低阻抗	2.5	5	10	KΩ	
$\Delta V_{(VDD/2)}$	(VDD/2) 输出电压误差		±1%	±3%		@VDD=5.0V
t_{WDT}	看门狗定时器超时溢出时间		2048		ILR 时钟周期	misc[1:0]=00 (默认)
			4096			misc[1:0]=01
			16384			misc[1:0]=10
			256			misc[1:0]=11

符 号	特 性	最小值	典型值	最大值	单位	条 件 （Ta=25℃）
t_{WUP}	系统唤醒时间					
	STOPEXE 省电模式下，切换 IO 引脚的快速唤醒		128		T_{SYS}	T_{SYS}是系统时钟周期
	STOPSYS 掉电模式下，切换 IO 引脚的快速唤醒。系统时钟为 IHRC。		128 T_{SYS} + T_{SIHRC}			T_{SIHRC}是IHRC从上电后的稳定时间，在 5V下约 5us.
	STOPSYS 掉电模式下，切换 IO 引脚的快速唤醒。系统时钟为 ILRC。		128 T_{SYS} + T_{SILRC}			T_{SILRC}是ILRC从上电后的稳定时间，在 5V下约 43ms
	STOPEXE 省电模式和 STOPSYS 掉电模式下，切换 IO 引脚的普通唤醒		1024		T_{ILRC}	T_{ILRC}是ILRC时钟周期
t_{SBP}	系统开机时间 (从开启电源算起)		1024		T_{ILRC}	T_{ILRC}是ILRC时钟周期
t_{RST}	外部复位脉冲宽度	120			us	@VDD=5V

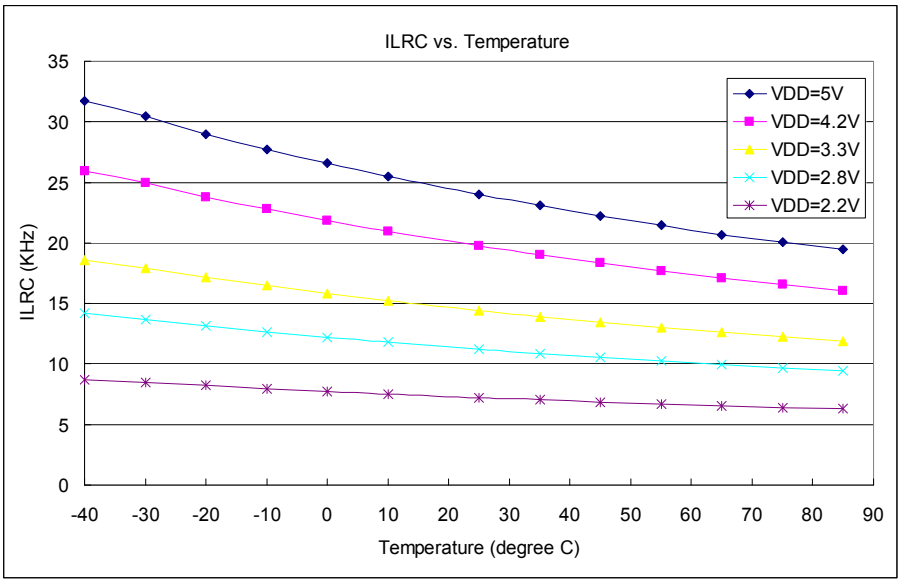
*****这些参数是设计参考值，并不是每个芯片测试。

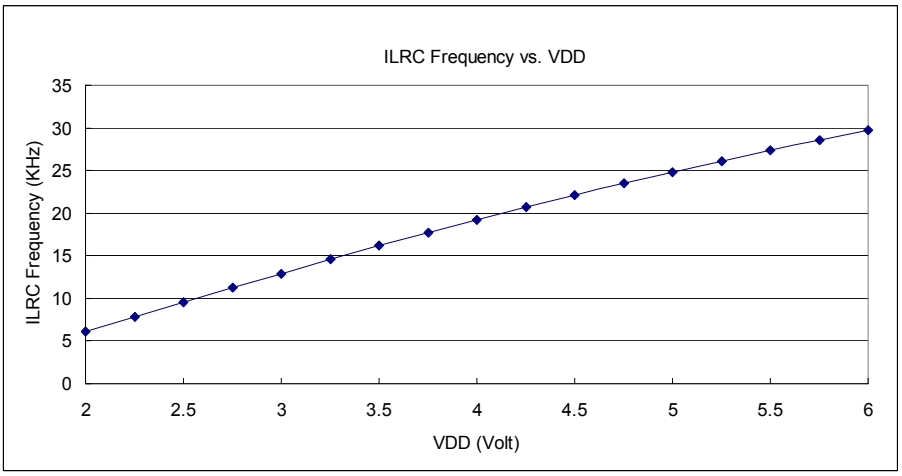
** Under_20ms_Vdd_Ok 为对 Vdd 能否于 20ms 内从 0V 上升到指定电压的一个检查条件。

4.2. 最大范围

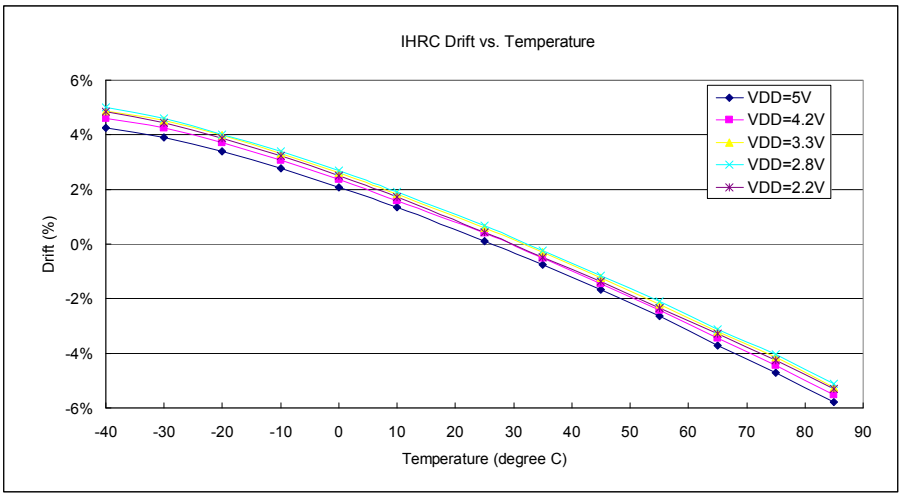
- 电源电压 ……..... 2.2V ~ 5.5V
- 输入电压……..... -0.3V ~ VDD + 0.3V
- 工作温度 ……..…… -40°C ~ 85°C
- 节点温度……..….. 150°C
- 储藏温度 ……..…… -50°C ~ 125°C

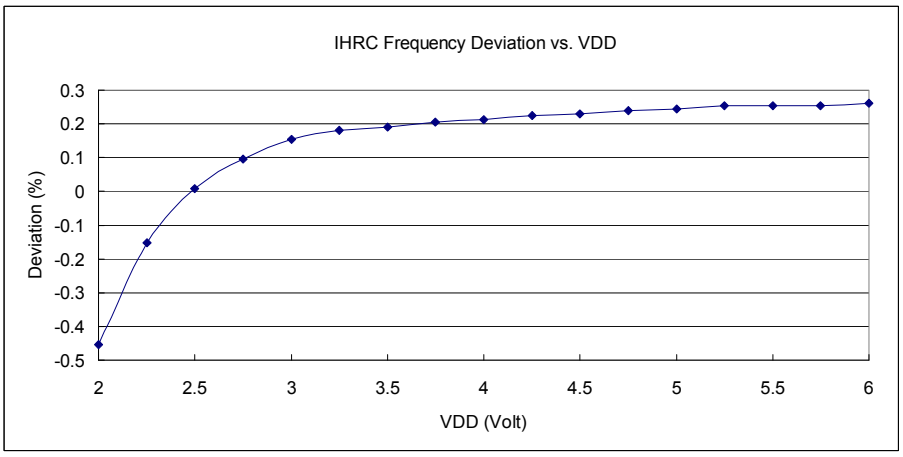
4.3. ILRC频率与VDD、温度关系的曲线图





4.4. IHRC频率与VDD、温度关系的曲线图

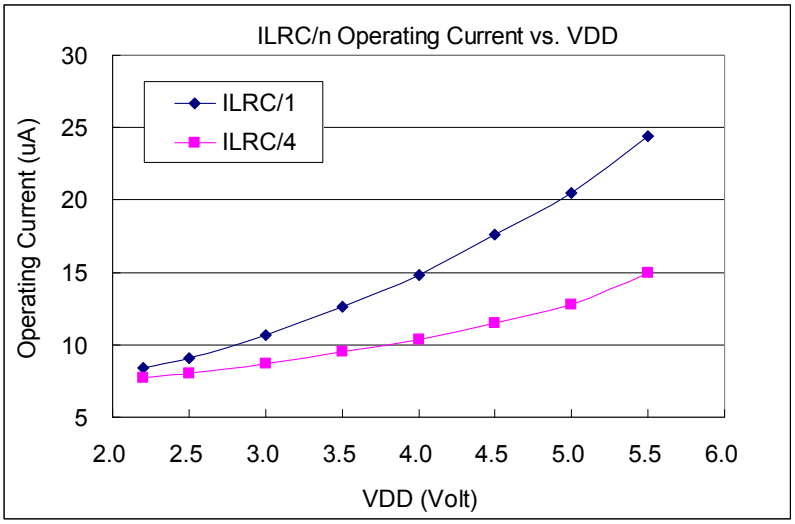




注意：IHRC 校准到 16MHz

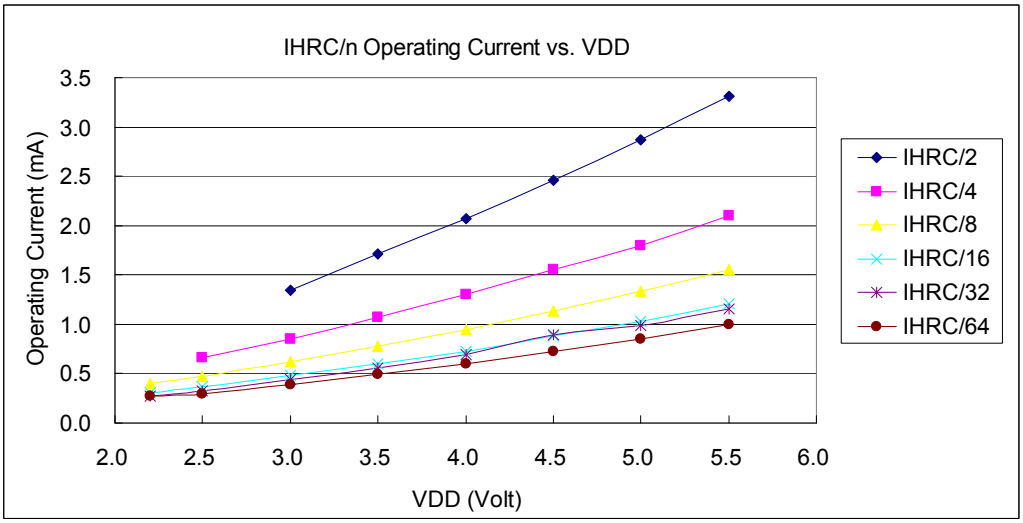
4.5. 工作电流量测值 @系统时钟= ILRC÷n

量测条件：
2-FPPA (FPPA0: 切换 PA0, FPPA1: 闲置)
启用: ILRC;
禁用: Band-gap, LVR, IHRC, EOSC, T16, TM2, ADC 等模块;
IO 引脚: PA0:0.5Hz 输出切换而且没负载, 其它脚位: 输入而且不浮接。



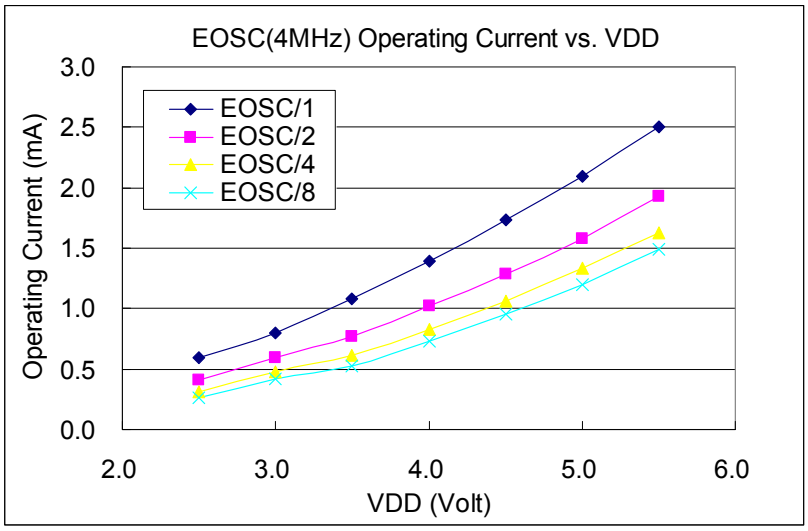
4.6. 工作电流量测值 @系统时钟= IHRC÷n

量测条件：
2-FPPA (FPPA0: 切换 PA0, FPPA1: 闲置)
启用: Band-gap, LVR, IHRC;
禁用: ILRC, EOSC, T16, TM2, ADC 等模块;
IO 引脚: PA0:0.5Hz 输出切换而且没负载, 其它脚位: 输入而且不浮接



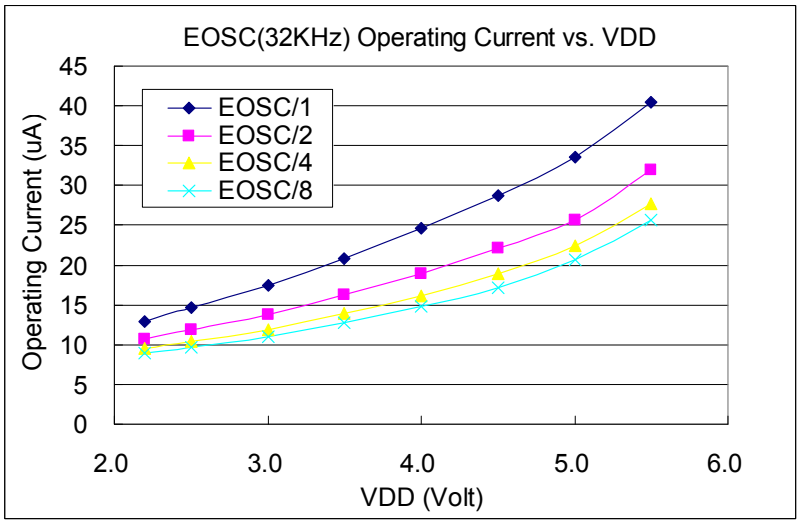
4.7. 工作电流量测值 @系统时钟= 4MHz晶振EOSC÷n

量测条件：
2-FPPA (FPPA0: 切换 PA0, FPPA1: 闲置)
启用: EOSC, MISC.6 = 1;
禁用: Band-gap, LVR, IHRC, ILRC, T16, TM2, ADC,等模块;
IO 引脚: PA0:0.5Hz 输出切换而且没负载, 其它脚位: 输入而且不浮接。

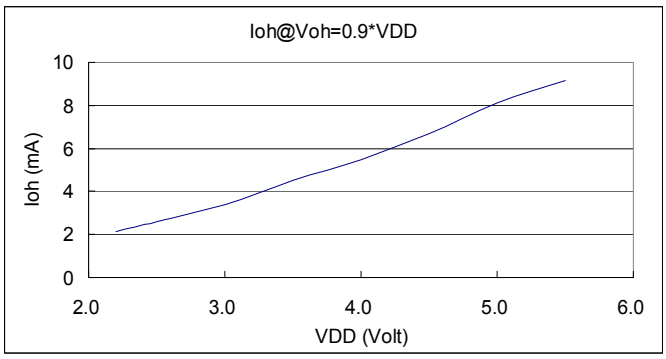


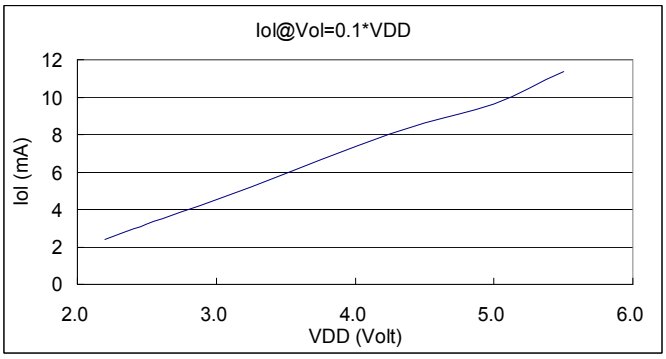
4.8. 工作电流量测值 @系统时钟= 32kHz晶振EOSC÷n

量测条件：
2-FPPA (FPPA0: 切换 PA0, FPPA1: 闲置)
启用: EOSC, MISC.6 = 1;
禁用: Band-gap, LVR, IHRC, ILRC, T16, TM2, ADC 等模块;
IO 引脚: PA0:0.5Hz 输出切换而且没负载, 其它脚位: 输入而且不浮接。

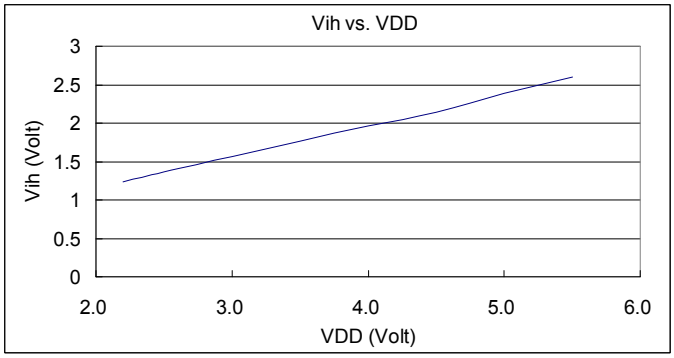


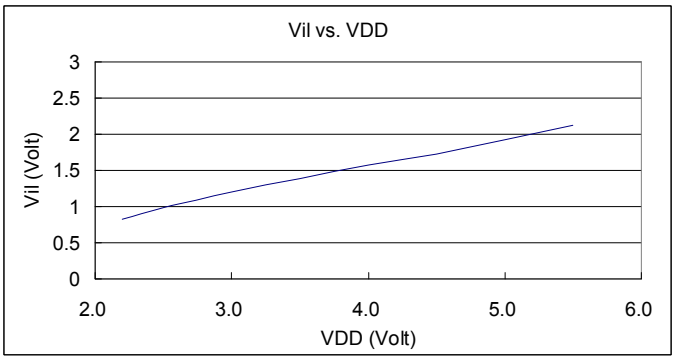
4.9. IO引脚输出驱动电流(I_{OH})和灌电流(I_{OL})曲线图



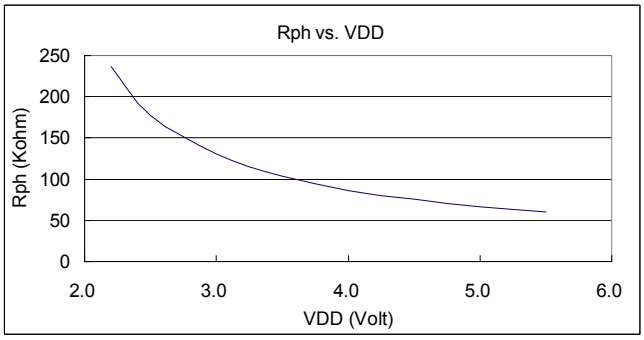


4.10. 测量的IO输入阈值电压(V_{IH}/V_{IL})

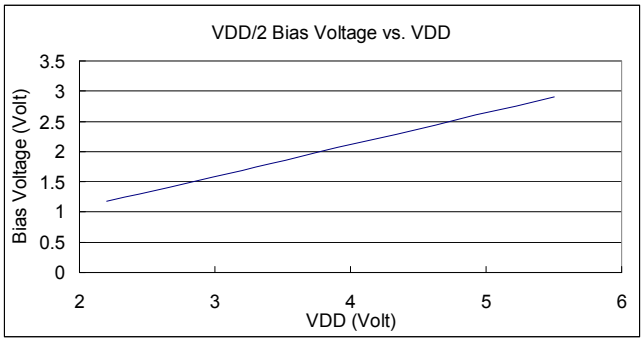




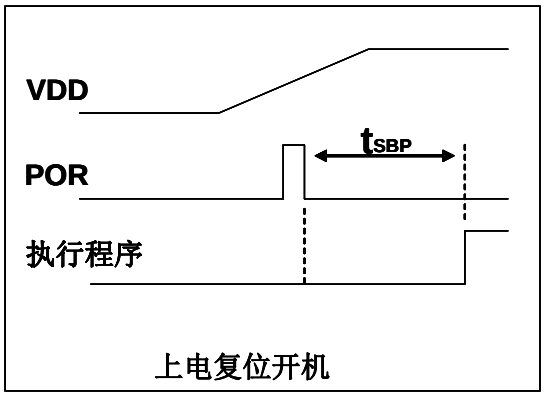
4.11. IO 引脚拉高阻抗曲线图

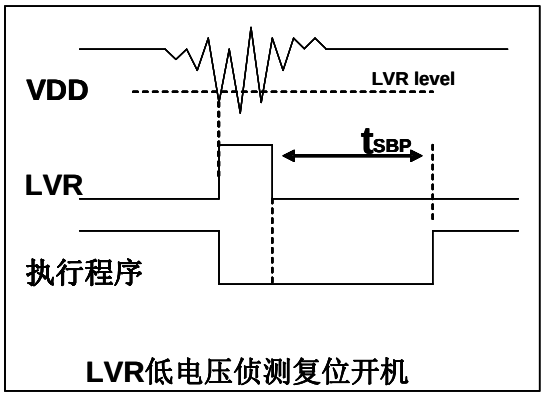


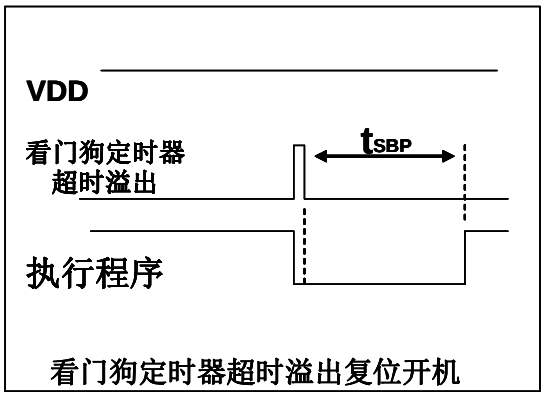
4.12. 输出(VDD/2)偏置电压与VDD关系的曲线图

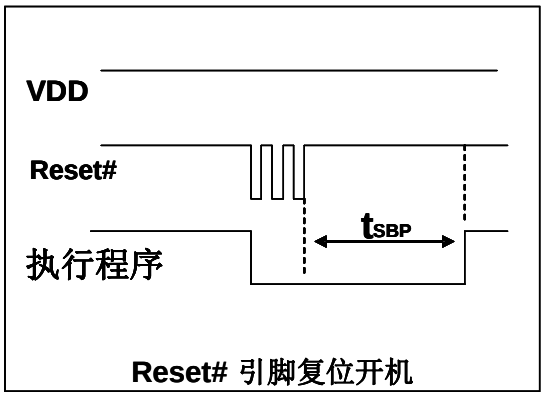


4.13. 开机时序图









5. 功能概述

5.1. 处理单元

PMC232 内有两个处理单元：FPP0 和 FPP1，在每一个处理单元中包括：（1）其本身的程序计数器来控制程序执行的顺序（2）自己的堆栈指针用来存储或恢复程序计数器的程序执行（3）自己的累加器（4）状态标志以记录程序执行的状态。在上电复位后只有 FPP0 是启用的，系统初始化将从 FPP0 开始，而 FPP1 可以由使用者的程序来决定是否使用。FPP0 和 FPP1 都有自己的程序计数器和累加器用以执行程序，标志寄存器以记录程序状态，堆栈指针做为跳跃操作。基于这样的架构，FPP0 和 FPP1 可以独立执行自己程序，达到并行处理效能。

FPP0 和 FPP1 共享 2Kx16 bits OTP 程序存储器，160 bytes 数据 SRAM 以及所有的 IO 口，这两个 FPP 单元是各自独立运作在相斥的时钟周期，以避免干扰。芯片内部有一个工作切换硬件模块以决定 FPP0 和 FPP1 相对应的周期。图 1 所示为 FPP0 和 FPP1 硬件框图以及基本时序图。对于 FPP0 而言，其程序将按顺序每两个系统时钟执行一次，如图：FPP0 在第（M-1），第 M 和第（M+1）时钟周期执行程序。对于 FPP1 而言，其程序将按顺序每两个系统时钟执行一次，如图：FPP1 在第（N-1），第 N 和第（N+1）时钟周期执行程序。

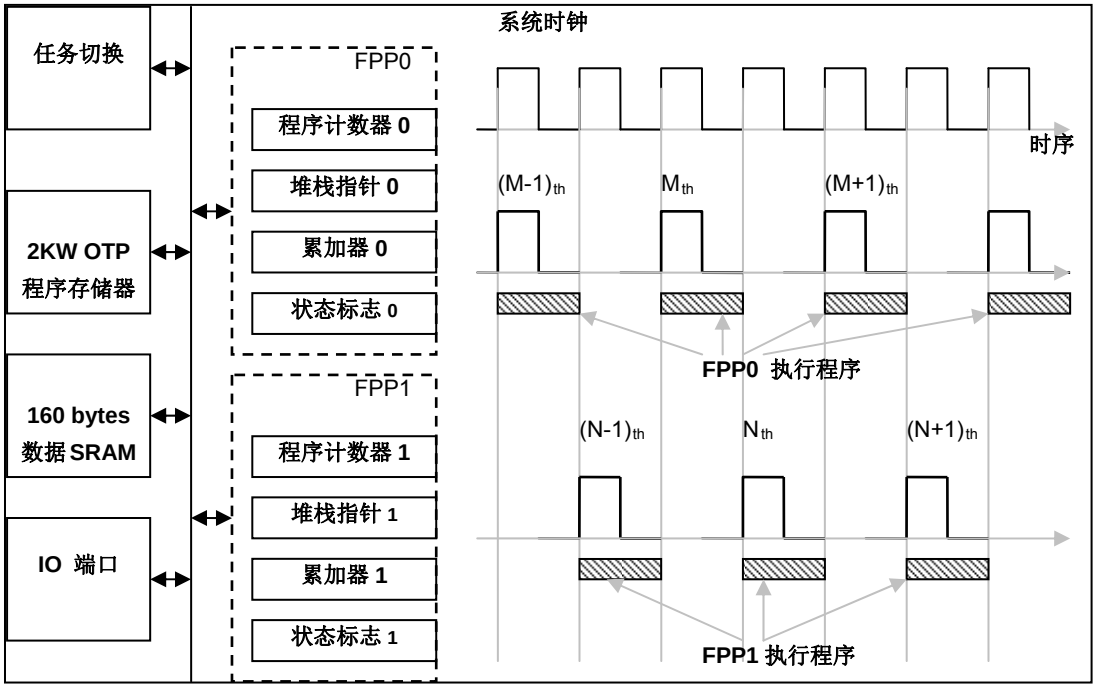


图 1：FPP 单元架构以及基本时序

每个 FPP 单元具有整个系统一半的计算能力，例如，如果系统时钟为 8MHz，FPP0 和 FPP1 将分别在 4MHz 时钟下工作。FPP 单元可以通过允许寄存器编程来启用或禁用；上电复位后，只有 FPP0 是被启用的。系统初始化将从 FPP0 开始，FPP1 可以由用户的程序来决定是否启用。FPP0 和 FPP1 可以被 FPP0 或 FPP1 中任一个禁用，包括禁用本身这一 FPP 单元。

5.1.1. 程序计数器

程序计数器（PC）记录下一个执行指令的地址，在每个指令周期后程序计数器会自动递增，以便指令码按顺序从程序存储器取出。某些指令，如分支指令和子程序调用都会改变顺序并放入一个新值到程序计数器。PMC232 程序计数器的位长度是 12 。在硬件复位后，FPP0 的程序计数器为 0、FPP1 为 1。当中断发生时，程序计数器会跳转到' h10 的中断服务程序处。FPP0 和 FPP1 都具有各自独立的程序计数器来控制其程序执行顺序。

5.1.2. 堆栈指针

在每个处理单元的堆栈指针是用来指引堆栈存储器的顶部，该处是用来存储子程序的局部变量和参数的地方；堆栈指针寄存器（SP）的地址是 IO 0x02h。堆栈指针的位数是 8 位，堆栈存储器是与数据 SRAM 共享，所以堆栈存储器的使用从地址 0x00h 开始，并在 208 字节以内。FPP0 和 FPP1 使用的堆栈存储器都可以由用户通过指定堆栈指针寄存器来调整，意味着 FPP0 和 FPP1 的堆栈指针单位深度是可调的，以优化系统性能。下面的示例显示了如何在 ASM 汇编语言下定义堆栈：

```
    .ROMADR     0
    GOTO        FPPA0
    GOTO        FPPA1
    ...
    .RAMADR     0               // 地址必需小于 0x100
    WORD        Stack0 [1]      // 1 个 WORD
    WORD        Stack1 [2]      // 2 个 WORD
    ...
FPPA0:
    SP  =   Stack0;             // 指定 Stack0 给 FPPA0 使用,
                                // 只能有一层呼叫，因为 Stack0[1]
    ...
    call    function1
    ...
FPPA1:
    SP  =   Stack1;             // 指定 Stack1 给 FPPA1 使用,
                                // 可以有 2 层呼叫，因为 Stack1[2]
    ...
    call    function2
    ...
```

在使用 Mini-C 汇编语言下，由系统软件计算堆栈的深度,使用者不需特别花时间计算，主程序如下：

```
    void    FPPA0 (void)
    {
  ...
    }
```

使用者可以在程序分解的窗口里检查堆栈的设定，图 2 表示在 FPP0 执行前的堆栈状态，系统计算出所需的堆栈空间，并保留该空间给程序使用。

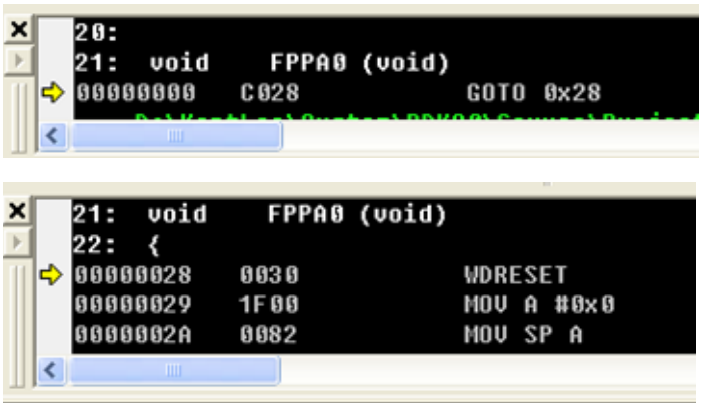


图 2：使用 Mini-C 的堆栈设定

5.1.3. 一个处理单元工作模式

传统的单片机使用者如果不需要有并行处理能力的单片机，PMC232 除了具有平行处理能力的双处理单元工作模式外，还提供单处理单元工作模式，它的表现就如同传统的单片机。当一个处理单元工作模式被选中后，FPP1 始终禁用，只有 FPP0 是使能的。图 3 显示了每个 FPP 单元的时序图，FPP1 总是禁用，只 FPP0 活跃。请注意在一个处理单元工作模式下，是不支持等待(*wait*)和延时(*delay*)指令。

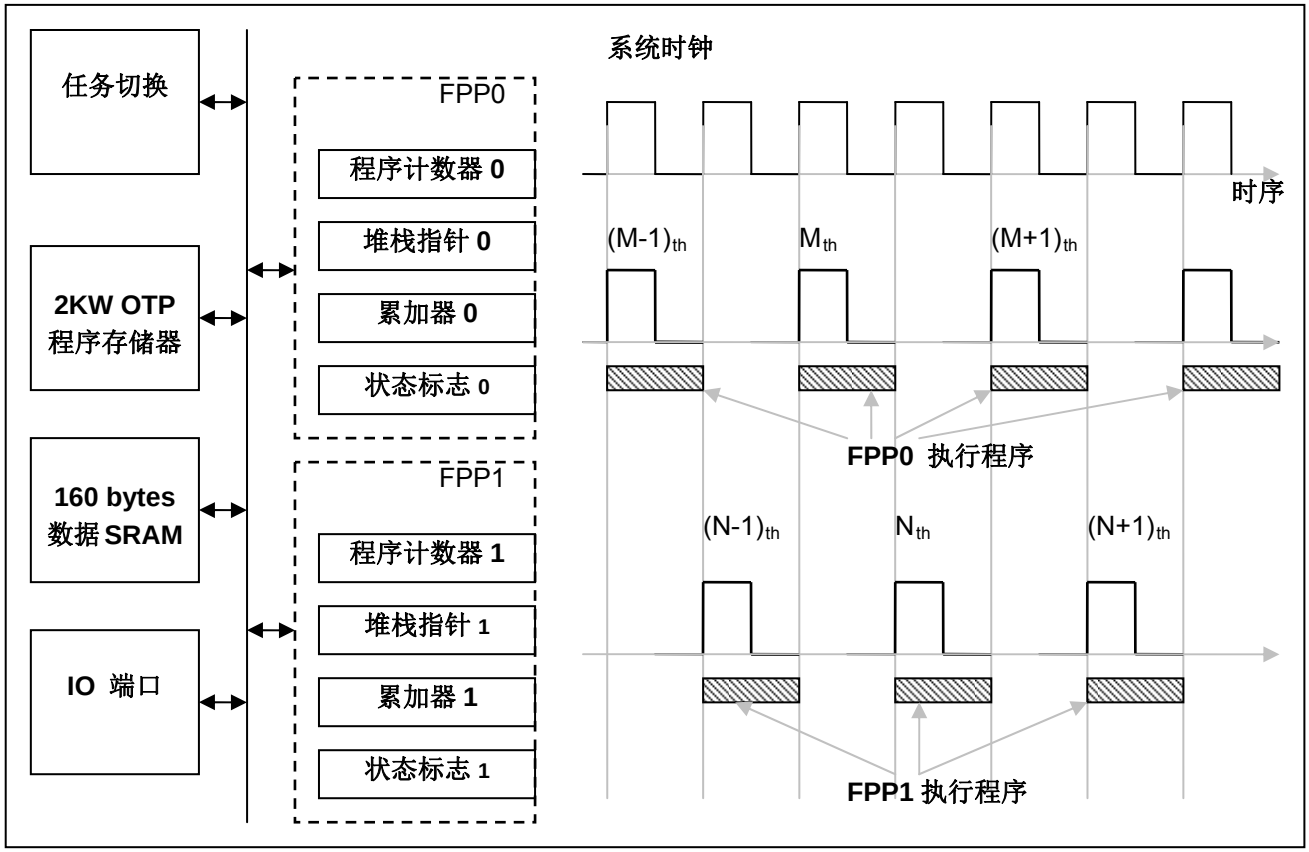


图 3：一个处理单元工作模式下的时序

5.2. OTP程序存储器

5.2.1. 程序存储器分配

OTP（一次性可编程）程序存储器用来存放要执行的程序指令。FPP0 和 FPP1 的所有程序代码都存储在这个 OTP 存储器。OTP 程序存储器可以储存数据，包含：数据，表格和中断入口。复位之后，FPP0 的初始地址为 0x0，FPP1 的初始地址为 0x1。中断入口是 0x10，只有 FPP0 能使用中断功；OTP 程序存储器最后 8 个地址空间是被保留给系统使用，如：校验，序列号等。PMC232 的 OTP 程序存储器结构是 2Kx16 位，如表 1 所示。 OTP 存储器从地址“hFF8~hFFF”供系统使用，从“h002 ~ h00F”和“h011~hFF7”地址空间是使用者的程序空间。地址 0x001 是 FPP1 的初始地址；另外，两个处理单元工作模式或一个处理单元工作模式，FPP0 的初始地址都是 0x000。

地址	功能
0x000	FPP0 起始地址 – *goto* 指令
0x001	FPP1 起始地址 – *goto* 指令
0x002	使用者程序区
•	•
0x00F	使用者程序区
0x010	中断入口地址
0x011	使用者程序区
•	•
0x7F7	使用者程序区
0x7F8	系统使用
•	•
0x7FF	系统使用

表 1：PMC232 程序存储器结构

5.2.2. 两个处理单元工作模式下程序存储器分配例子

表 2 显示了一个例子，使用两个处理单元工作模式下，程序存储器分配情形：

地址	功能
000	FPP0 起始地址 – *goto* 指令(goto 0x020)
001	FPP1 程序开始
•	•
00F	goto 0x1A1 继续 FPP1 程序
010	中断入口地址(只给 FPP0)
•	•
01F	中断程序结束
020	FPP0 程序开始
•	•
1A0	FPP0 程序结束
1A1	继续 FPP1 程序
•	•
7F7	FPP1 程序结束
7F8	系统使用
•	•
7FF	系统使用

表 2：两个处理单元工作模式之程序存储器分配案例

5.2.3. 一个处理单元工作模式下程序存储器分配例子

表 3 显示了一个例子，使用一个处理单元工作模式下，程序存储器分配情形，整个使用者程序存储器都可以被分配到 FPP0。

地址	功能
000	FPP0 起始地址
001	FPP0 程序开始
002	使用者程序区
•	•
00F	Goto 指令(goto 0x020)
010	中断入口地址
011	中断程序
•	•
01F	中断程序结束
020	使用者程序区
•	•
•	•
7F7	使用者程序区
7F8	系统使用
•	•
7FF	系统使用

表 3：一个处理单元工作模式之程序存储器分配案例

5.3. 程序结构

5.3.1. 两个处理单元工作模式下程序结构

开机后，FPP0 和 FPP1 的程序开始地址分别是 0x000 和 0x001。中断服务程序的入口地址是 0x010，而且只有 FPP0 才能接受中断服务。 PMC232 的基本软件结构如图 4 所示。两个 FPP 的处理单元的程序代码是被放置在同一个程序空间。除了初始地址和中断入口地址外，处理单元的程序代码可以放在程序存储器任何位置，并没有在特定的地址;开机后，将首先执行 fpp0Boot，其中将包括系统初始化和启用其它 FPP 的单元。

```
.romadr  0x00
// Program Begin
goto        fpp0Boot;
goto        fpp1Boot;
//------中断服务程序----------------
.romadr  0x010
            pushaf ;
            t0sn    intrq.0;   //PA.0 ISR
            goto    ISR_PA0;
            t0sn    intrq.1;    //PB.0 ISR
…
            goto    ISR_PB0;
//------中断服务程序结束--------
//------ FPP0程序开始----------
fpp0Boot :
//--- FPP0初始化…
…
fpp0Loop:
…
  goto fpp0Loop:
//------FPP0程序结束--------
//------FPP1程序开始----------
fpp1Boot :
//---FPP1初始化…
…
fpp1Loop:
…
  goto fpp1Loop:
//---------FPP1程序结束--------
```

图 4：两个处理单元工作模式之程序结构

5.3.2. 一个处理单元工作模式下程序结构

开机后，FPP0 的程序开始地址是 0x000，中断服务程序的入口地址是 0x010，一个处理单元工作模式下的程序结构与传统的单片机软件结构相同，开机后，程序将从地址 0x000 然后继续程序的顺序。

5.4. 启动程序

开机时，POR（上电复位）是用于复位PMC232；但是，上电后电源电压可能不太稳定，为确保单片机是工作在电压稳定的状态，在执行第一条指令之前，PMC232 会延迟 1024 个ILRC时钟周期，这时间就是t_{SBP}，如图 5 所示。

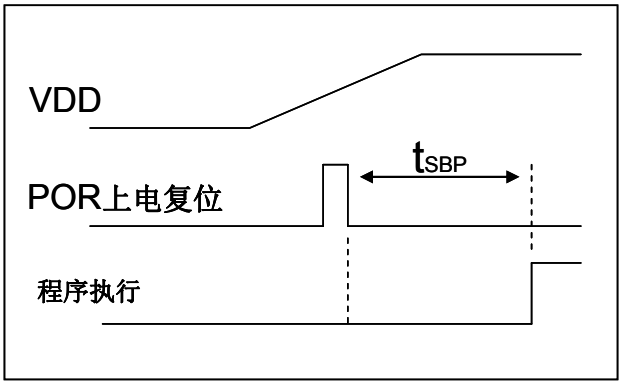


图 5：上电复位时序

开机后，使用者需要初始化系统，设定使用情形，图 6 显示的是典型开机流程。请注意，上电复位后 FPP1 是禁用，建议不要在 FPP0 以及系统初始化完成前，启用 FPP1。

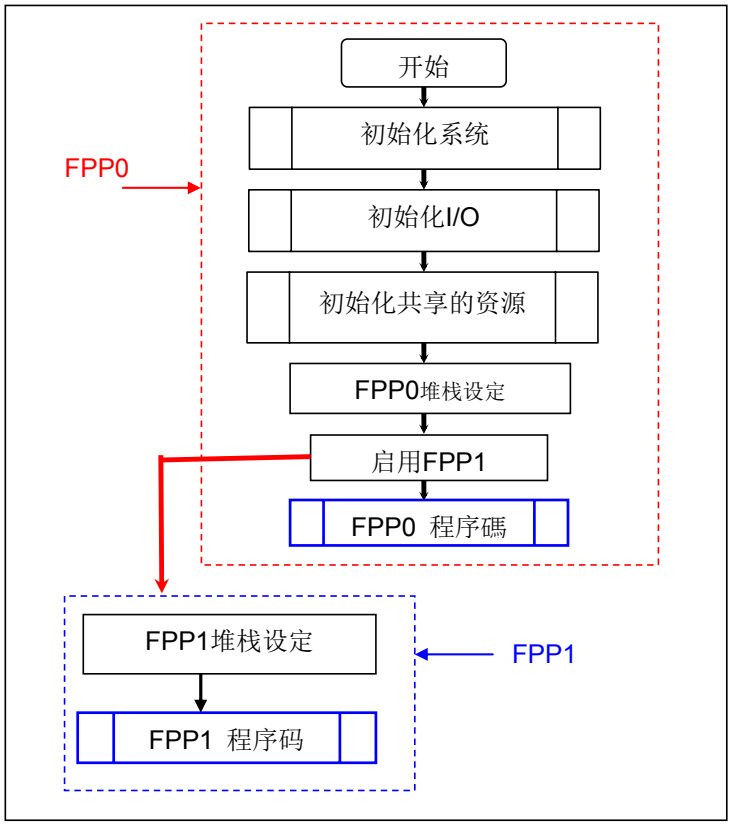


图 6：开机流程

5.5. 数据存储器

图 7 显示了 PMC232 内置 160 Bytes 数据存储器的结构以及使用，所有的 SRAM 数据存储器可以透过 FPP0 和 FPP1 在 1 个时钟周期内直接读取或写入，存取方式可以字节或位操作。此外 SRAM 数据存储器还充当间接存取方法的数据指针和 FPP0、FPP1 的堆栈记忆体。FPP0 和 FPP1 的堆栈记忆体使用是独立互不影响的，并定义在数据存储器中。FPP0 和 FPP1 处理单元的堆栈指针通过指针寄存器各自定义，FPP0 和 FPP1 所需要的存储器深度是由使用者来定，堆栈记忆体的调整可完全灵活安排，可以由用户动态调整。

对于间接存取指令而言，数据存储器用作数据指针来当数据地址，所有的数据存储器都可以当做数据指针，这对于间接存取指令是相当灵活和有用的。由于数据宽度为 8 位，间接存取记忆体大小必需在 256 字节以内，PMC232 内置的 160 个字节数据存储器都可以利用间接存取指令来存取。

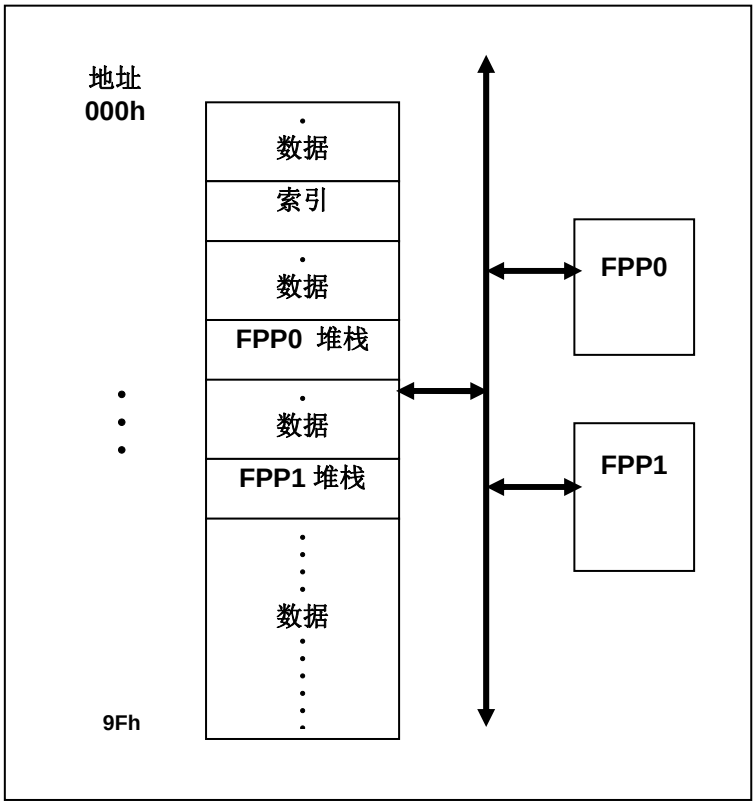


图 7：数据存储器结构和使用

5.6. 算术和逻辑单元

算术和逻辑单元（ALU）是用来作整数算术、逻辑、转移和其它特殊运算的单元。运算的数据来源可以从指令、累加器或 SRAM 数据存储器，计算结果可写入累加器或 SRAM。FPP0 和 FPP1 在其相应的操作周期分享 ALU 的使用。

5.7. 振荡器和时钟

PMC232 内置 3 个振荡器电路：晶体振荡器、内部高频RC振荡器（IHRC）和内部低频RC振荡器（ILRC），这 3 个振荡器电路可以分别透过寄存器eoscr.7，clkmd.4 以及clkmd.2 来启用或禁用。使用者可以选择不同的振荡器以及clkmd寄存器产生不同的系统频率，以满足不同的应用。

振荡器硬件模块	启用或禁用	开机后默认值
EOSC	**eoscr**.7	禁用
IHRC	**clkmd**.4	启用
ILRC	**clkmd**.2	启用

5.7.1. 内部高频振荡器（IHRC）和低频振荡器（ILRC）

开机后，IHRC是自动被启用的，可以通过***clkmd***寄存器位4 禁用它，IHRC的频率是可以透过***ihrcr***寄存器校准，通常它被校准至 16MHz以消除工艺生产所产生的变化，校准后的频率偏差，正常情况下可在 1%以内。IHRC频率校准是在用户程序编译时选择，并在芯片烧录OTP程序码时，一个个校准。IHRC的频率会因电源电压和工作温度而漂移，在VDD电压为 2.2V~5.5V以及温度 40°C~85°C条件下，总频率漂移约为±8%，请参考IHRC频率与VDD、温度关系的曲线图。

开机后，ILRC是自动被启用的，并可以通过***clkmd***寄存器位 2禁用它，ILRC的频率固定为 24kHz。但是，因为工厂生产的过程会有所不同，使用时电源电压和温度的差异等因素，都可能影响频率漂移。请参考直流电气特性规格数据。

5.7.2. 单片机校准

在芯片生产制造时，每一颗的IHRC频率和Bandgap参考电位可能都有稍微的不同，PMC232 提供了IHRC频率校准以及Bandgap参考电压校准，以消除芯片生产制造时的漂移，校准功能选项是在用户程序编译时选择，IDE软件在编译用户的程序时会自动插入用户程序，校准的命令如下：

.ADJUST_IC SYSCLK=IHRC/(**p1**), IHRC=(**p2**)MHz, VDD=(**p3**)V, Bandgap=(**p4**);

这里， **p1**=2, 4, 8, 16, 32; 提供系统时钟不同的频率。

p2=14 ~ 18; 提供芯片IHRC校准到不同的频率，通常选16MHz。

p3=2.5 ~ 5.5; 提供芯片在不同的电压校准。

p4= On或Off; Band-gap参考电压校准是On或Off。

5.7.3. IHRC频率校准和系统时钟

IHRC频率校准选项是在用户程序编译时选择，IDE软件在编译用户的程序时会自动插入用户程序，提供的选项是如表 4 所示：

SYSCLK	CLKMD	IHRCR	描述
○ Set IHRC / 2	= 34h (IHRC / 2)	有校准	IHRC校准到16MHz，系统时钟CLK=8MHz (IHRC/2)
○ Set IHRC / 4	= 14h (IHRC / 4)	有校准	IHRC校准到16MHz，系统时钟CLK=4MHz (IHRC/4)
○ Set IHRC / 8	= 3Ch (IHRC / 8)	有校准	IHRC校准到16MHz，系统时钟CLK=2MHz (IHRC/8)
○ Set IHRC / 16	= 1Ch (IHRC / 16)	有校准	IHRC校准到16MHz，系统时钟CLK=1MHz (IHRC/16)
○ Set IHRC / 32	= 7Ch (IHRC / 32)	有校准	IHRC 校准到16MHz，系统时钟 CLK=0.5MHz (IHRC/32)
○ Set ILRC	= E4h (ILRC / 1)	有校准	IHRC校准到16MHz，系统时钟CLK=ILRC
○ Disable	没改变	没改变	IHRC没有校准，系统时钟CLK也没有改变,Bandgap没有校准

表 4 IHRC频率校准选项

通常，.ADJUST_IC命令是摆在程序启动后的第 1 个动作，以便开机后能够设立所要的工作频率。IHRC频率校准只会进行一次，是在烧录OTP程序码时进行，烧录后就不会再重复执行了。假如使用者选择不同的频率校准选项，PMC232 在开机后的状态也将不同，下面所示为不同选项在开机后，PMC232 执行此命令后的状态：

(1) .ADJUST_IC SYSCLK=IHRC/2, IHRC=16MHz, VDD=5V, Bandgap=On

开机后，CLKMD = 0x34：

- IHRC 频率在 VDD=5V 下，校准到 16MHz 并且是启用的
- 系统时钟 CLK = IHRC/2 = 8MHz
- 看门狗定时器是禁用, ILRC 是启用的, PA5 引脚设为输入，Bandgap 校准到 1.2V

(2) .ADJUST_IC SYSCLK=IHRC/4, IHRC=16MHz, VDD=3.3V, Bandgap=On

开机后，CLKMD = 0x14：

- IHRC 频率在 VDD=3.3V 下，校准到 16MHz 并且是启用的
- 系统时钟 CLK = IHRC/4 = 4MHz
- 看门狗定时器是禁用, ILRC 是启用的, PA5 引脚设为输入，Bandgap 校准到 1.2V

(3) .ADJUST_IC SYSCLK=IHRC/8, IHRC=16MHz, VDD=2.5V, Bandgap=On

开机后，CLKMD = 0x3C：

- IHRC 频率在 VDD=2.5V 下，校准到 16MHz 并且是启用的
- 系统时钟 CLK = IHRC/8 = 2MHz
- 看门狗定时器是禁用, ILRC 是启用的, PA5 引脚设为输入，Bandgap 校准到 1.2V

(4) .ADJUST_IC SYSCLK=IHRC/16, IHRC=16MHz, VDD=2.5V, Bandgap=On

开机后，CLKMD = 0x1C：

- IHRC 频率在 VDD=2.5V 下，校准到 16MHz 并且是启用的
- 系统时钟 CLK = IHRC/16 = 1MHz
- 看门狗定时器是禁用, ILRC 是启用的, PA5 引脚设为输入，Bandgap 校准到 1.2V

(5) .ADJUST_IC SYSCLK=IHRC/32, IHRC=16MHz, VDD=5V, Bandgap=Off

开机后，CLKMD = 0x7C：

- IHRC 频率在 VDD=5V 下，校准到 16MHz 并且是启用的
- 系统时钟 CLK = IHRC/32 = 500kHz
- 看门狗定时器是禁用, ILRC 是启用的, PA5 引脚设为输入，Bandgap 没校准

(6) .ADJUST_IC SYSCLK=ILRC, IHRC=16MHz, VDD=5V, Bandgap=Off

开机后，CLKMD = 0XE4：

- IHRC频率在VDD=5V下，校准到 16MHz并且是禁用的
- 系统时钟 CLK = ILRC
- 看门狗定时器是禁用, ILRC 是启用的, PA5 引脚设为输入，Bandgap 没校准

(7) .ADJUST_IC DISABLE

开机后，CLKMD 寄存器没被改变（没任何动作）

- IHRC 频率没有校准并且是禁用的, Band-gap 没有校准
- 系统时钟 CLK = ILRC
- 看门狗定时器是启用, ILRC 是启用的, PA5 引脚设为输入

5.7.4. 晶体振荡器

如果使用晶体振荡器，X1 和 X2 之间需要晶体或谐振器。其应用线路如图 8 所示;晶体振荡器的工作频率可以从 32kHz 到 4MHz，超过 4MHz 是不支持的。寄存器 ***eoscr***（0x0b）位 7 是用来启用晶体振荡器，另外，寄存器 ***eoscr***（0x0b）位 6~5 提供不同的驱动电流能力，以配合不同的晶体振荡器频率：

- *eoscr*.[6:5]=01：低驱动电流，适用于较低频率，例如：32kHz 晶体振荡器
- *eoscr*.[6:5]=10：中驱动电流，适用于中间频率，例如：1MHz 晶体振荡器
- *eoscr*.[6:5]=11：高驱动电流，适用于较高频率，例如：4MHz 晶体振荡器

为了得到良好的正弦波形，外部电容 C1 和 C2 也需调整，表 5 显示不同的晶体或谐振器，C1 和 C2 的建议值以及在对应条件下所测量到的起振时间。因为晶体或谐振器都有其不同的特性，所需要的 C1、C2 以及起振时间也会因不同的晶体或谐振器而有些差异，使用时请参考晶体或谐振器规格并选择合适的 C1 和 C2。

频率	C1	C2	测量起振时间	条件
4MHz	4.7pF	4.7pF	6ms	(*eoscr*[6:5]=11, *misc*.6=0)
1MHz	10pF	10pF	11ms	(*eoscr*[6:5]=10, *misc*.6=0)
32kHz	22pF	22pF	450ms	(*eoscr*[6:5]=01, *misc*.6=0)

表 5、 不同的晶体或谐振器所需 C1 和 C2 的建议值

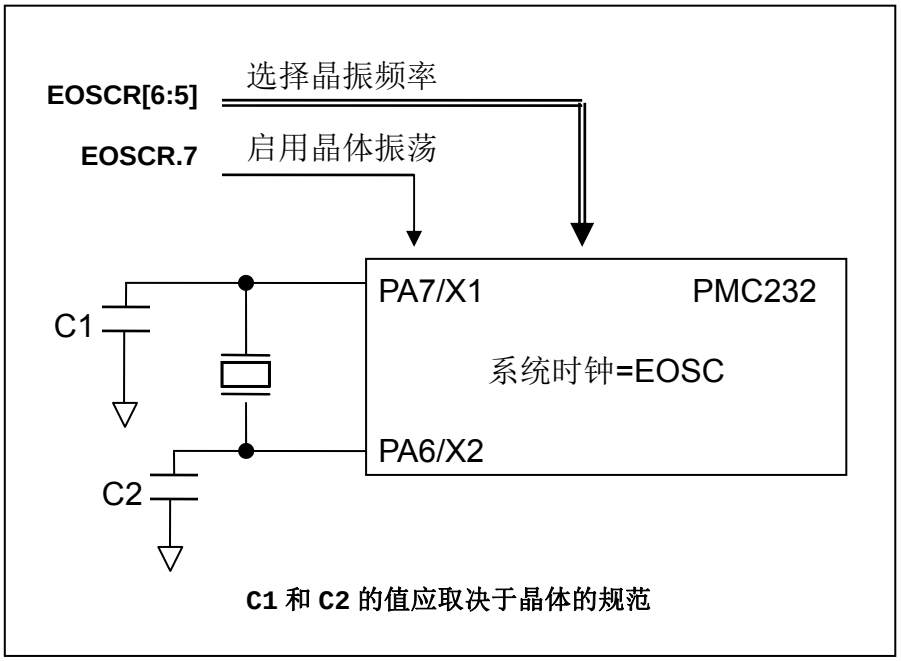


图 8：晶体振荡器使用接法

为了优化晶体振荡器的功耗和起振时间，***misc***寄存器第 6 位提供选项以提高晶体振荡器的电流驱动能力。在晶体振荡器开始启动时，启用***misc***寄存器的第 6 位以加快振荡；当振荡器稳定后，这位是可以被禁用的以达到节电的目的。当使用外部晶体振荡器时，***padier***寄存器的位 7 和位6 应设置为高，以防止漏电流。

另外，使用晶体振荡器要特别注意启用之后所需要的稳定时间，它会依频率、晶体或谐振器型号、外部电容、工作电压而不同，在将系统时钟源切换成晶体振荡器之前，必需确保晶体振荡器已经稳定，参考程序如下：

```
void   FPPA0 (void)
{
   . ADJUST_IC   SYSCLK=IHRC/16, IHRC=16MHz, VDD=5V, Bandgap=On
   // 如果 Bandgap 不需要校准，可以写成 ". ADJUST_IC  DISABLE" . . .
   ...
   $  EOSCREnable, 4MHz;     // EOSCR = 0b110_00000;
   $  T16M   EOSC, /1, BIT13;  // T16 收到 2^14=16384 个晶体振荡器时钟,
        // Intrq.T16 =>1,  晶体振荡器已经稳定
   WORD count   =  0;
   stt16       count;
   Intrq.T16  =  0;
   wait1    Intrq.T16;          //  计数从 0x0000 到 0x2000,然后设置 INTRQ.T16
   clkmd   =  0xA4;             //  将系统时钟切换成 EOSC;
   ...
```

进入掉电模式之前，请先将晶体振荡器关闭以避免不可预期的唤醒发生；假如使用 32kHz 晶体振荡器而且又需要非常的省电，当晶体振荡器稳定后，设置 *misc*.6 为 1 以降低电流。

5.7.5. 系统时钟和LVR水平

系统时钟可以来自 EOSC，IHRC 和 ILRC，图 9 显示为 PMC232 中的系统时钟选项的硬件框图。

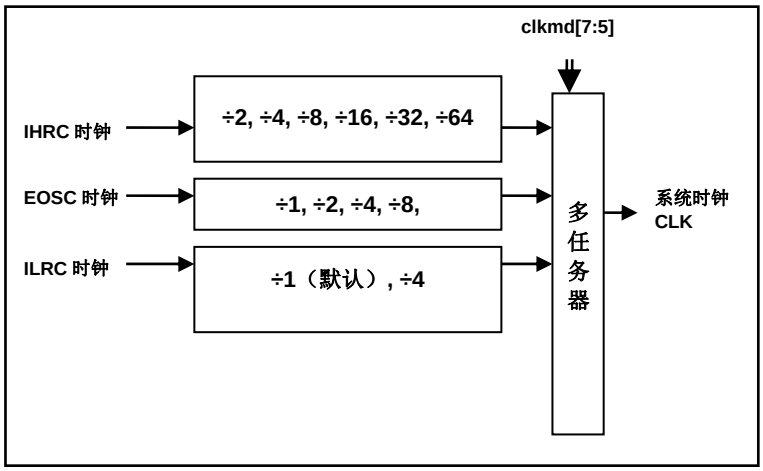


图 9：系统时钟选项

使用者可以依照不同的需求选择不同的工作系统时钟，选定的工作系统时钟应与电源电压和LVR水平结合起来，才能使系统稳定运作。低电压水平将在编译过程中选择，以下是工作频率和LVR水平的建议：

- 当系统时钟=8MHz，LVR=3.1V
- 当系统时钟=4MHz，LVR=2.5V
- 当系统时钟=2MHz，LVR=2.2V

5.7.6. 系统时钟切换

IHRC校准后，用户可能希望系统时钟切换到新的频率或可随时切换系统时钟来优化系统性能和功耗。基本上，PMC232 系统时钟可以随意在IHRC, ILRC和EOSC之间切换，只要透过***clkmd***寄存器设定，;系统时钟可以立即的转换成新的频率。请注意，在下命令给寄存器***clkmd***切换频率时，不能同时关闭原来的时钟模块。下面这些例子显示更多时钟切换需知道的信息，请参阅“求助”→“使用手册”→ “IC介绍”→ “缓存器介绍”→ “CLKMD”。

例 1: 系统时钟从ILRC切换到IHRC/2

```
...                     // 系统时钟为 ILRC
CLKMD    = 0x34;        // 切换为 IHRC/2，ILRC 不能在这里禁用
CLKMD.2  = 0;           // ILRC 可以在这里禁用
...
```

例 2: 系统时钟从ILRC切换到EOSC

```
...                     // 系统时钟为 ILRC
CLKMD    = 0xA6;        // 切换为IHRC，ILRC 不能在这里禁用
CLKMD.2  = 0;           // ILRC 可以在这里禁用
...
```

例 3: 系统时钟从IHRC/2 切换到ILRC

```
...                     // 系统时钟为 IHRC/2
CLKMD    = 0xF4;        // 切换为ILRC，IHRC 不能在这里禁用
CLKMD.4  = 0;           // IHRC 可以在这里禁用
...
```

例 4: 系统时钟从IHRC/2 切换到EOSC

```
...                     // 系统时钟为 IHRC/2
CLKMD    = 0XB0;        // 切换为EOSC，IHRC 不能在这里禁用
CLKMD.4  = 0;           // IHRC 可以在这里禁用
...
```

例 5: 系统时钟从IHRC/2 切换到IHRC/4

```
...                     // 系统时钟为 IHRC/2, ILRC 为启用
CLKMD    = 0X14;        // 切换为IHRC/4
...
```

例 6: 系统可能当机，如果同时切换时钟和关闭原来的振荡器

```
...                     // 系统时钟为ILRC
CLKMD    = 0x30;        // 不能从 ILRC 切换到 IHRC/2，同时又关闭 ILRC 振荡器
...
```

5.8. 16 位定时器 (Timer16)

PMC232 内含一个 16 位定时器，定时器时钟可来自于系统时钟（CLK）、外部晶体振荡器时钟、内部高频振荡时钟（IHRC）、内部低频振荡时钟（ILRC），或 Port A 位 0，1 个多任务器用于选择时钟输出的时钟来源，在送到 16 位定时器之前，1 个可软件编程的预分频器提供÷1、÷4、÷16、÷64 选择，让计数范围更大。

16 位定时器只能向上计数，定时器初始值可以使用 ***stt16*** 指令来设定，而定时器的数值也可以利用 ***ldt16*** 指令存储到 SRAM 数据存储器。可软件编程的选择器用于选择 timer16 的中断条件，当定时器溢出时，Timer16 可以触发中断。Timer16 模块框图如图 10。中断源是来自 16 位定时器的位 8 到位 15，中断类型可以上升沿触发或下降沿触发，定义在 ***intensr*** 寄存器位 4（IO 地址 0x0C）。

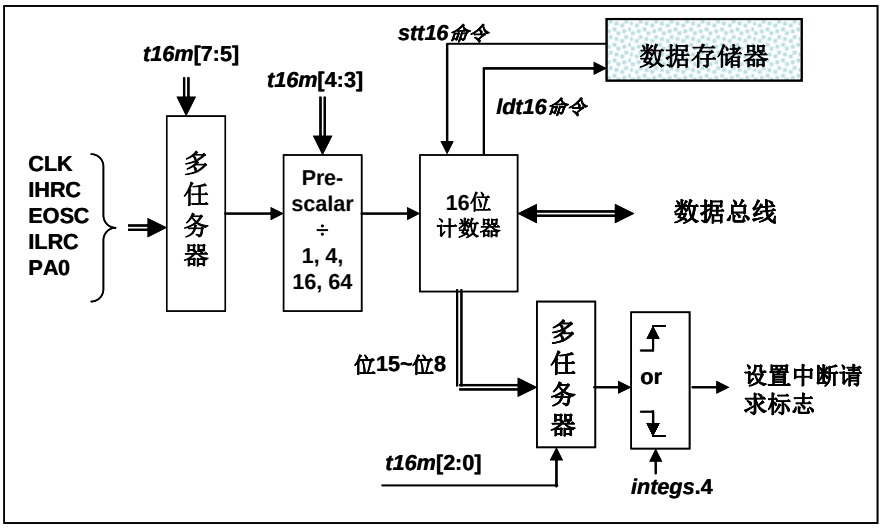


图 10：Timer16 模块框图

当使用 Timer16 时，Timer16 的使用语法已定义在.INC 文件中。有三个参数来定义 Timer16 的使用;第一参数是用来定义 Timer16 时钟源，第二参数是用来定义预分频器，最后一个是定义中断源。详细如下：

```
T16M        IO_RW       0x06
    $ 7~5:  STOP, SYSCLK, X, PA0_R, IHRC, EOSC, ILRC, PA0_F     // 第一参数
    $ 4~3:  /1, /4, /16, /64                                    // 第二参数
    $ 2~0:  BIT8, BIT9, BIT10, BIT11, BIT12, BIT13, BIT14, BIT15 // 第三参数
```

使用者可以依照系统的要求来定义 T16M 参数，例子如下：

```
$ T16M  SYSCLK, /64, BIT15;
    // 选择(SYSCLK/64) 当 Timer16 时钟源,每 2^16 个时钟周期产生一次 INTRQ.2=1
    // 假如系统时钟 System Clock = IHRC / 2 = 8 MHz
    // SYSCLK/64 = 8 MHz/64 = 8 uS,约每 524 mS 产生一次 INTRQ.2=1
```

$ T16M EOSC, /1, BIT13;

// 选择(EOSC/1) 当 Timer16 时钟源,每 2^14 个时钟周期产生一次 INTRQ.2=1

// 假如 EOSC=32768 Hz, 32768 Hz/(2^14) = 2Hz, 约每 0.5S 产生一次 INTRQ.2=1

$ T16M PA0_F, /1, BIT8;

// 选择 PA0 当 Timer16 时钟源, 每 2^9 个时钟周期产生一次 INTRQ.2=1

// 每接收 512 个 PA0 个时钟周期产生一次 INTRQ.2=1

$ T16M STOP;

// 停止 Timer16 计数

假如Timer16 是不受干扰的自由运行, 中断发生的频率可以用下列式子描述:

$$F_{INTRQ\_T16M} = F_{clock\ source} \div P \div 2^{n+1}$$

这里, F 是 Timer16 的时钟源频率,

P 是寄存器 ***t16m*** [4:3]的选择(可以为 1, 4, 16, 64),

N 是中断要求所选择的位, 例如: 选择位 10, n=10.

5.9. 8 位PWM定时器(Timer2)

PMC232 内置一个 8 位PWM硬件定时器，硬件框图请参考图 11，定时器的时钟源可能来自系统时钟（CLK），内部高频RC振荡器时钟（IHRC），内部低频RC振荡器时钟（ILRC），PA0，PA3，PA4，寄存器***tm2c***的位[7：4]用来选择定时器时钟。请注意，外部晶体振荡器是不能当做Timer2 的时钟，因为它可能有突波。另外，在执行仿真器（ICE）时，若内部高频RC振荡器时钟（IHRC）被选择当做Timer2 的时钟，当仿真器停住时，IHRC时钟仍继续送到Timer2，所以Timer2 在仿真器停住时仍然会继续计数。依据寄存器***tm2c***的设定，Timer2 的输出可以是PA2 或PA3.利用软件编程寄存器***tm2s***位[6:5]，时钟预分频器的模块提供了÷1，÷4，÷16 和÷64 的选择，另外，利用软件编程寄存器***tm2s***位[4:0]，时钟分频器的模块提供了÷1~÷31 的功能。在结合预分频器以及分频器，Timer2 时钟（TM2_CLK）频率可以广泛和灵活，以提供不同产品应用。 TM2_CLK也可以被选定为系统时钟，以提供特殊的系统时钟频率，请参阅***clkmd***寄存器。

8 位 PWM 定时器只能执行 8 位上升计数操作，经由寄存器 ***tm2ct***，定时器的值可以设置或读取。当 8 位定时器计数值达到上限寄存器设定的范围时，定时器将自动清除为零，上限寄存器用来定义定时器产生波形的周期或 PWM 占空比。8 位 PWM 定时器有两个工作模式：周期模式和 PWM 模式;周期模式用于输出固定周期波形或中断事件; PWM 模式是用来产生 PWM 输出波形，PWM 分辨率可以为 6 位或 8 位。图 12 显示出 Timer2 周期模式和 PWM 模式的时序图。

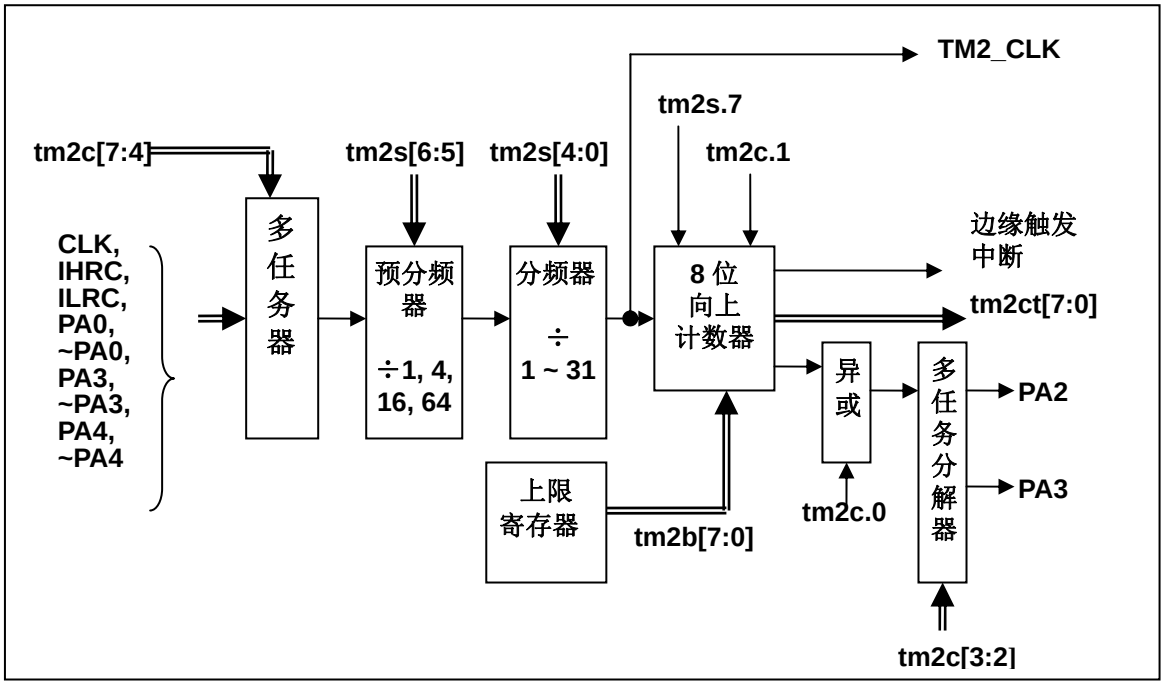


图 11. Timer2 模块框图

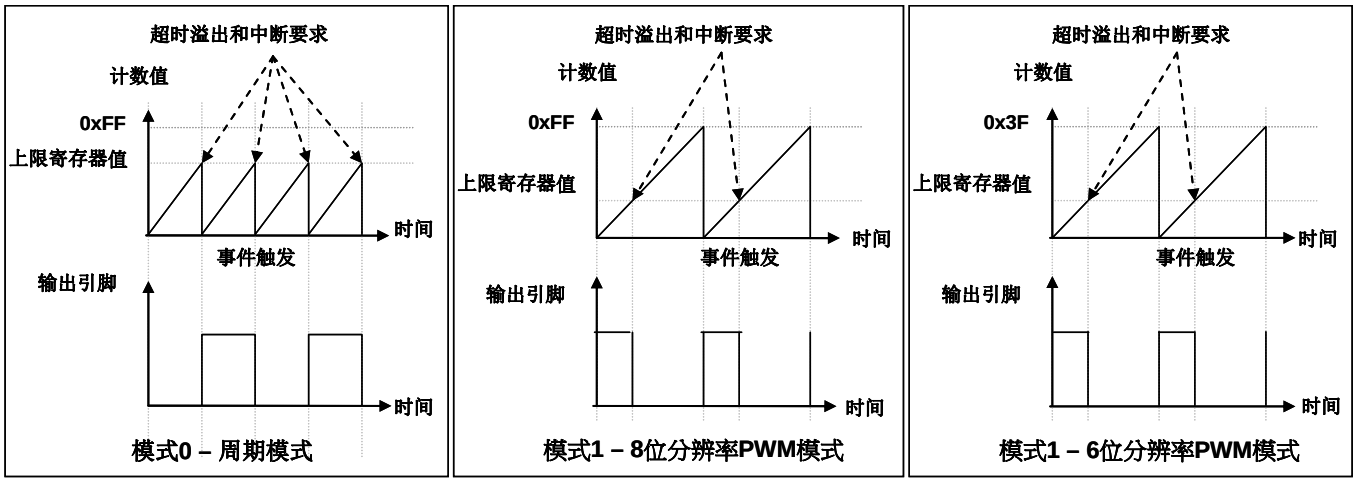


图 12. Timer2 周期模式和 PWM 模式的时序图

5.9.1. 使用Timer2 产生定期波形

如果选择周期模式的输出，输出波形的占空比总是 50%，其输出频率与寄存器设定，可以概括如下：

输出信号频率 = Y ÷ [2 × (K+1) × S1 × (S2+1)]

这里, Y = tm2c[7:4] : Timer2 所选择的时钟源频率
K = tm2b[7:0] : 上限寄存器设定的值(十进制)
S1 = tm2s[6:5] : 预分频器设定值 (1, 4, 16, 64)
S2 = tm2s[4:0] : 分频器值 (十进制，1 ~ 31)

例 1:

tm2c = 0b0001_1000, Y=8MHz
tm2b = 0b0111_1111, K=127
tm2s = 0b0_00_00000, S1=1, S2=0
➔ 输出信号频率 = 8MHz ÷ [2 × (127＋1) × 1 × (0＋1)] = 31.25kHz

例 2:

tm2c = 0b0001_1000, Y=8MHz
tm2b = 0b0111_1111, K=127
tm2s[7:0] = 0b0_11_11111, S1=64 , S2 = 31
➔ 输出信号频率 = 8MHz ÷ (2 × (127＋1) × 64 × (31＋1)) =15.25Hz

例 3:

tm2c = 0b0001_1000, Y=8MHz
tm2b = 0b0000_1111, K=15
tm2s = 0b0_00_00000, S1=1, S2=0
➔ 输出信号频率 = 8MHz ÷ (2 × (15＋1) × 1 × (0＋1)) = 250kHz

例 4:

tm2c = 0b0001_1000, Y=8MHz
tm2b = 0b0000_0001, K=1
tm2s = 0b0_00_00000, S1=1, S2=0
➔输出信号频率 = 8MHz ÷ (2 × (1＋1) × 1 × (0＋1)) =2MHz

使用Timer2 定时器产生定期波形的示例程序如下所示:

```
void FPPA0 (void)
{
    .ADJUST_OTP_IHRCR 8MIPS
    …
    tm2ct = 0x0;
    tm2b = 0x7f;
    tm2s = 0b0_00_00001; // 8 位 pwm，预分频 = 1，分频 = 2
    tm2c = 0b0001_01_0_0;  //系统时钟，输出 =PA2，周期模式
    while(1)
    {
       nop;
    }
}
```

5.9.2. 使用Timer2 产生 8 位PWM波形

如果选择 8 位PWM的模式，应设立tm2c [1] = 1，tm2s [7] = 0，输出波形的频率和占空比可以概括如下:

输出频率 = Y ÷ [256 × S1 × (S2+1)]

输出空占比 = (K＋1) ÷ 256

这里,

Y = tm2c[7:4] : Timer2 所选择的时钟源频率
K = tm2b[7:0] : 上限寄存器设定的值(十进制)
S1 = tm2s[6:5] : 预分频器设定值 (1, 4, 16, 64)
S2 = tm2s[4:0] : 分频器值 (十进制，1 ~ 31)

例 1:

tm2c = 0b0001_1010, Y=8MHz
tm2b = 0b0111_1111, K=127
tm2s = 0b0_00_00000, S1=1, S2=0
➔ 输出频率 = 8MHz ÷ (256 × 1 × (0+1)) = 31.25kHz
➔ 输出空占比 = [(127+1) ÷ 256] × 100% = 50%

例 2:

tm2c = 0b0001_1010, Y=8MHz
tm2b = 0b0111_1111, K=127
tm2s = 0b0_11_11111, S1=64, S2=31
➔ 输出频率 = 8MHz ÷ (256 × 64 × (31+1)) = 15.25Hz
➔ 输出空占比 = [(127+1) ÷ 256] × 100% = 50%

例 3:

tm2c = 0b0001_1010, Y=8MHz
tm2b = 0b1111_1111, K=255
tm2s = 0b0_00_00000, S1=1, S2=0
➔ 输出频率 = 8MHz ÷ (256 × 1 × (0+1)) = 31.25kHz
➔ 输出空占比 = [(255+1) ÷ 256] × 100% = 100%

例 4:

tm2c = 0b0001_1010, Y=8MHz
tm2b = 0b0000_1001, K = 9
tm2s = 0b0_00_00000, S1=1, S2=0
➔ 输出频率 = 8MHz ÷ (256 × 1 × (0+1)) = 31.25kHz
➔ 输出空占比 = [(9+1) ÷ 256] × 100% = 3.9%

使用Timer2 定时器产生PWM波形的示例程序如下所示:

```
voidFPPA0 (void)
{
    .ADJUST_OTP_IHRCR   8MIPS
    wdreset;
    tm2ct = 0x0;
    tm2b = 0x7f;
    tm2s = 0b0_00_00001;   //8 位 pwm，预分频 = 1，分频 = 2
    tm2c = 0b0001_01_1_0;  //系统时钟，输出 = PA2, PWM 模式
    while(1)
    {
    nop;
    }
}
```

5.9.3. 使用Timer2 产生 6 位PWM波形

如果选择 6 位PWM的模式，应设立tm2c [1] = 1，tm2s [7] = 1，输出波形的频率和占空比可以概括如下:

输出频率 = Y ÷ [64 × S1 × (S2+1)]
输出空占比 = [(K＋1) ÷ 64] × 100%

这里,

Y = tm2c[7:4] : Timer2 所选择的时钟源频率
K = tm2b[7:0] : 上限寄存器设定的值(十进制)
S1 = tm2s[6:5] : 预分频器设定值 (1, 4, 16, 64)
S2 = tm2s[4:0] : 分频器值 (十进制，1 ~ 31)

例 1:

tm2c = 0b0001_1010, Y=8MHz
tm2b = 0b0001_1111, K=31
tm2s = 0b1_00_00000, S1=1, S2=0
➔ 输出频率 = 8MHz ÷ (64 × 1 × (0+1)) = 125kHz
➔ 输出空占比 = [(31+1) ÷ 64] × 100% = 50%

例 2:

tm2c = 0b0001_1010, Y=8MHz
tm2b = 0b0001_1111, K=31
tm2s = 0b1_11_11111, S1=64, S2=31
➔ 输出频率 = 8MHz ÷ (64 × **64** × (31+1)) = 61.03 Hz
➔ 输出空占比 = [(31+1) ÷ 64] × 100% = 50%

例 3:

tm2c = 0b0001_1010, Y=8MHz
tm2b = 0b0011_1111, K=63
tm2s = 0b1_00_00000, S1=1, S2=0
➔ 输出频率 = 8MHz ÷ (64 × **1** × (0+1)) = 125kHz
➔ 输出空占比 = [(63+1) ÷ 64] × 100% = 100%

例 4:

tm2c = 0b0001_1010, Y=8MHz
tm2b = 0b0000_0000, K=0
tm2s = 0b1_00_00000, S1=1, S2=0
➔ 输出频率 = 8MHz ÷ (64 × **1** × (0+1)) = 125kHz
➔ 输出空占比 = [(0+1) ÷ 64] × 100% =1.5%

5.10. 看门狗定时器

看门狗定时器是一个定时器，其时钟源来自内部低频振荡器（ILRC），频率大约是 24kHz。利用 *misc* 寄存器的选择，可以设定四种不同的看门狗定时器超时时间，它是：

- 当 misc[1:0]=11 时：256 个 ILRC 时钟周期
- 当 misc[1:0]=10 时：16384 ILRC 时钟周期
- 当 misc[1:0]=01 时：4096 ILRC 时钟周期
- 当 misc[1:0]=01 时：2048 ILRC 时钟周期

为确保看门狗定时器在超时溢出周期之前被清零，在安全时间内，用指令“wdreset”清零看门狗定时器。在上电复位或任何时候使用 wdreset 指令，看门狗定时器都会被清零。当看门狗定时器超时溢出时， PMC232 将复位并重新运行程序。请特别注意，由于生产制程会引起 ILRC 频率相当大的漂移，上面的数据仅供设计参考用，还是需要以各个单片机测量到的数据为准。请注意：当启用快速唤醒时，看门狗时钟源会切换到系统时钟(例如：4MHz)，所以，建议打开快速唤醒功能时，在进入掉电模式前关闭看门狗定时器，等系统被唤醒后，在关闭快速唤醒之后再打开看门狗定时器。

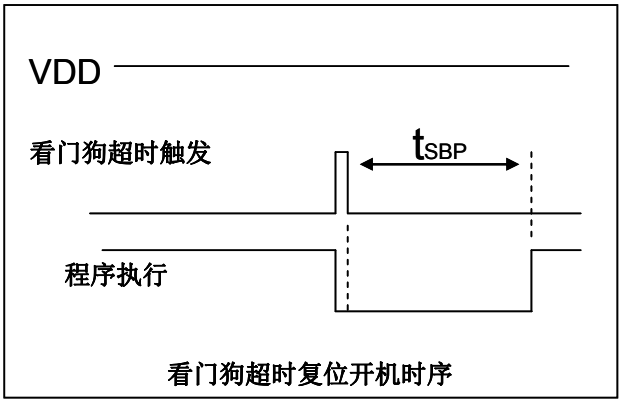


图 13：看门狗定时器超时溢出的相关时序

5.11. 中断

PMC232 有 5 个中断源：两个外部中断源（PA0，PB0，边缘触发形态可以由 ***integsr*** 寄存器选择），Timer16 中断源，Timer2 中断源，ADC 中断源。每个中断请求源都有自己的中断控制位启用或禁用它。硬件框图请参考图 14，所有的中断请求标志位是由硬件置位并且必须用软件清零的。所有的中断请求源最后都需由 ***engint*** 指令控制（启用全局中断）使中断运行，以及使用 ***disgint*** 指令（禁用全局中断）停用它。只 FPP0 可以接受中断请求，其它的 FPP 的单元不会受到中断干扰。中断堆栈是共享数据存储器，其地址由堆栈寄存器 *sp* 指定。由于程序计数器是 16 位宽度，堆栈寄存器 *sp* 位 0 应保持 0。此外，用户可以使用 *pushaf* 指令存储 ACC 和标志寄存器的值到堆栈，以及使用 *popaf* 指令将值从堆栈恢复到 ACC 和标志寄存器中。

当 PMC232 执行到中断入口地址处，全局中断会自动停止；到 ***reti*** 指令被执行时自动恢复启用。中断请求可以在任何时候接受，包括在中断服务程序被执行的过程中，中断嵌套的深度是由软件编程所决定的，因为每个 FPP 单位的 8 位堆栈指针寄存器都是可读写的。可由软件编程调整栈点在存储器里的位置，每个 FPP 单元堆栈指针的深度可以完全由用户指定，以实现最大的系统弹性。中断服务程序的入口地址都是 0x10，只属于 FPP0。

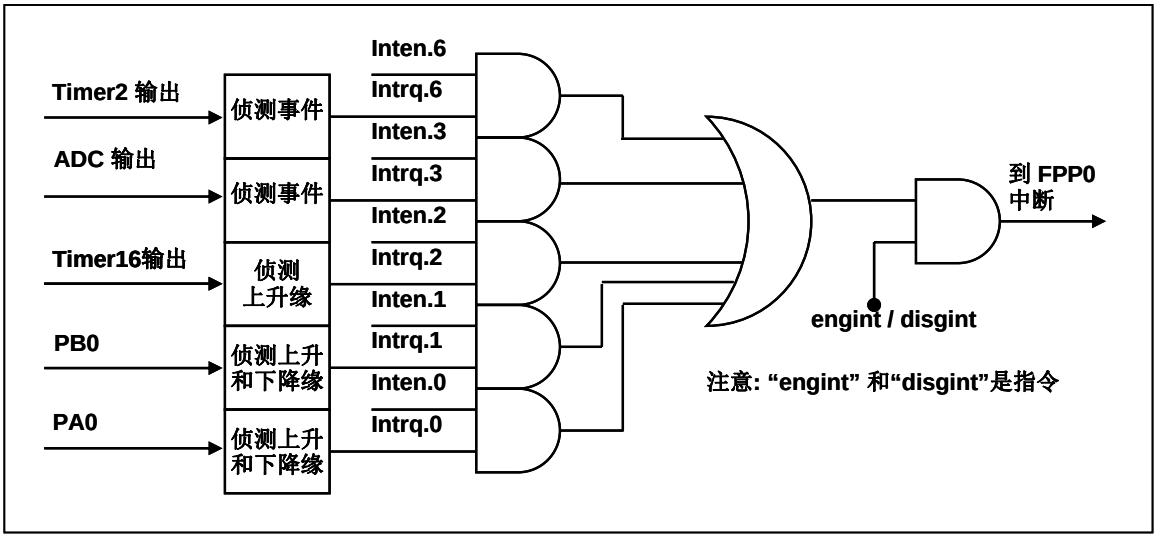


图 14 中断硬件框图

一旦发生中断，其工作流程将是：

- 程序计数器将自动存储到 *sp* 寄存器指定的堆栈存储器。
- 新的 *sp* 将被更新为 *sp*+2。
- 全局中断将自动被禁用。
- 将从地址 0x010 获取下一条指令。

在中断服务程序中，可以通过读寄存器 *intrq* 知道中断发生源。

中断服务程序完成后，发出 *reti* 指令返回既有的程序，其具体工作流程将是：

- 从 *sp* 寄存器指定的堆栈存储器自动恢复程序计数器。
- 新的 *sp* 将被更新为 *sp*-2。
- 全局中断将自动启用。
- 下一条指令将是中断前原来的指令。

使用者必须预留足够的堆栈存储器以存中断向量，一级中断需要两个字节，两级中断需要 4 个字节。下面的示例程序演示了如何处理中断，请注意，处理中断和 *pushaf* 是需要四个字节堆栈存储器。

```
void    FPPA0 (void)
{
    ...
    $   INTEN   PA0;            // INTEN =1;当 PA0 准位改变，产生中断请求
    INTRQ  =  0;                // 清除 INTRQ
    ENGINT                      // 启用全局中断
    ...
    DISGINT                     // 禁用全局中断
    ...
}

void    Interrupt  (void)       // 中断程序
{
  PUSHAF                // 存储 ALU 和 FLAG 寄存器
  If  (INTRQ.0)
  {                     // PA0 的中断程序
    INTRQ.0  =  0;
    ...
    }
    ...
  POPAF                 // 回复 ALU 和 FLAG 寄存器
 }
```

如果用户想要在执行中断服务程序时接受其它中断，使用***engint***再次启用中断服务。虽然它允许使用***engint***做中断嵌套(nesting interrupt)，但是请特别注意需要的堆栈，另外，Mini-C不能计算所需嵌套中断的堆栈。嵌套中断的示例如下所示：

```
void Interrupt  (void)
{
  PUSHAF ;
    ...
  ENGINT ;
  ...
  //  允许其它中断要求
    ...
    POPAF ;
  }
```

5.12. 掉电模式

PMC232 有三个由硬件定义的工作模式，分别为：正常工作模式，电源省电模式和掉电模式。正常工作模式是所有功能都正常运行的状态，省电模式（*stopexe*）是在降低工作电流而且 CPU 保持在随时可以继续工作的状态，掉电模式（*stopsys*）是用来深度的节省电力。因此，省电模式适合在偶尔需要唤醒的系统工作，掉电模式是在非常低消耗功率且很少需要唤醒的系统中使用。图 15 显示省电模式（*stopexe*）和掉电模式（*stopsys*）之间在振荡器模块的差异，没改变就是维持原状态。

STOPSYS 和 STOPEXE 模式下在振荡器的差异			
	IHRC	ILRC	EOSC
STOPSYS	停止	停止	停止
STOPEXE	没改变	没改变	没改变

图 15：省电模式和掉电模式在振荡器模块的差异

5.12.1. 省电模式（stopexe）

使用 *stopexe* 指令进入省电模式，只有系统时钟被禁用，其余所有的振荡器模块都仍继续工作。所以只有 CPU 是停止执行指令，对 Timer16 定时器而言，如果它的时钟源不是系统时钟，那 Timer16 仍然会保持计数。*stopexe* 的省电模式下，唤醒源可以是 IO 的切换，或者 Timer16 计数到设定值时(假如 Timer16 的时钟源是 IHRC、ILRC 或 EOSC 模块)。假如系统唤醒是因输入引脚切换，那可以视为单片机继续正常的运行，在 *stopexe* 指令之后最好加个 ***nop*** 指令，省电模式的详细信息如下所示：

- IHRC、ILRC 和 EOSC 振荡器模块：没有变化。如果它被启用，它仍然继续保持活跃。
- 系统时钟禁用。因此，CPU 停止执行。
- OTP 存储器被关闭。
- Timer16：停止计数，如果选择系统时钟或相应的振荡器模块被禁止，否则，仍然保持计数。
- 唤醒来源：IO 的切换或 Timer16

请注意在下“***stopexe***”命令前，必须先关闭看门狗时钟以避免发生复位，例子如下：

```
CLKMD.En_WatchDog = 0;    // 关闭看门狗时钟
stopexe;
nop;
....                      // 省电中
Wdreset;
CLKMD.En_WatchDog = 1;    // 开启看门狗时钟
```

另一个例子是利用 Timer16 来唤醒系统:

```
$ T16M IHRC, /1, BIT8          // Timer16 setting
...
WORD count  = 0;
STT16  count;
stopexe;
nop;
...
```

Timer16 的初始值为 0，在 Timer16 计数了 256 个 IHRC 时钟后，系统将被唤醒。

5.12.2. 掉电模式（stopsys）

掉电模式是深度省电的状态，所有的振荡器模块都会被关闭。使用 *stopsys* 指令就可以使 PMC232 芯片直接进入掉电模式。在进入掉电模式之前，必须启用内部低频振荡器（ILRC）以便唤醒系统时使用，也就是说在发出 *stopsys* 命令之前，*clkmd* 寄存器的位 2 必须设置为 1。下面显示发出 *stopsys* 命令后，PMC232 内部详细的状态：

- 所有的振荡器模块被关闭。
- 启用内部低频振荡器（设置寄存器 *clkmd* 位 2）。
- OTP 存储器被关闭。
- SRAM 和寄存器内容保持不变。
- 唤醒源：任何 IO 切换。
- 如果 PA 或 PB 是输入模式，并由 *padier* 或 *pbdier* 寄存器设置为模拟输入，那该引脚是不能被用来唤醒系统。

输入引脚的唤醒可以被视为正常运行的延续，为了降低功耗，进入掉电模式之前，所有的 I/O 引脚应仔细检查，避免悬空而漏电。断电参考示例程序如下所示：

```
CMKMD   = 0xF4;   // 系统时钟从 IHRC 变为 ILRC
CLKMD.4 = 0;      //  IHRC 禁用
...
while (1)
{
   STOPSYS;          // 进入断电模式
   if  (...)  break; // 假如发生唤醒而且检查 OK，就返回正常工作
                     // 否则，停留在断电模式。
}
CLKMD = 0x34;      // 系统时钟从 ILRC 变为 IHRC/2
```

5.12.3. 唤醒

进入掉电或省电模式后，PMC232 可以通过切换 IO 引脚恢复正常工作；而 Timer16 中断的唤醒只适用于省电模式。图 16 显示 *stopsysy* 掉电模式和 *stopexe* 省电模式在唤醒源的差异。

掉电模式和省电模式在唤醒源的差异		
	切换 IO 引脚	T16 中断
stopsys	是	否
stopexe	是	是

图 16：掉电模式和省电模式在唤醒源的差异

当使用 IO 引脚来唤醒 PMC232，寄存器 *padier* 和 *pbdier* 应正确设置，使每一个相应的引脚可以有唤醒功能。从唤醒事件发生后开始计数，正常的唤醒时间大约是 1024 个 ILRC 时钟周期；另外，PMC232 提供快速唤醒功能，透过 *misc* 寄存器选择快速唤醒可以降低唤醒时间。对快速唤醒而言，假如是在 *stopexe* 省电模式下，切换 IO 引脚的快速唤醒时间为 128 个系统时钟周期；假如是在 *stopsys* 掉电模式下，切换 IO 引脚的快速唤醒时间为 128 个系统时钟周期加上上电后振荡器(IHRC 或 ILRC)的稳定时间。振荡器的稳定时间是从上电后开始算起，视系统时钟是选择 IHRC 或 ILRC 而定。特别注意，当 EOSC 被选用当系统时钟后，快速唤醒就自动关闭。

模式	唤醒模式	系统时钟源	切换IO引脚的唤醒时间(t_{WUP})
STOPEXE 省电模式	快速唤醒	IHRC 或 ILRC	128 * $T_{SYS,}$；这里T_{SYS}是系统时钟周期
STOPSYS 掉电模式	快速唤醒	IHRC	128 T_{SYS} + T_{SIHRC}； 这里T_{SIHRC}是IHRC从上电到稳定的时间
STOPSYS 掉电模式	快速唤醒	ILRC	128 T_{SYS} + T_{SILRC}； 这里T_{SILRC}是ILRC从上电到稳定的时间
STOPSYS 或 STOPEXE 模式	快速唤醒	EOSC	1024 * T_{ILRC}；这里T_{ILRC}是ILRC时钟周期
STOPEXE 省电模式	普通唤醒	任一	1024 * T_{ILRC}；这里T_{ILRC}是ILRC时钟周期
STOPSYS 掉电模式	普通唤醒	任一	1024 * T_{ILRC}；这里T_{ILRC}是ILRC时钟周期

请注意：当启用快速唤醒时，看门狗时钟源会切换到系统时钟(例如：4MHz)，所以，建议要进入掉电模式前，打开快速唤醒之前要关闭看门狗定时器，等系统被唤醒后，在关闭快速唤醒之后再打开看门狗定时器。

5.13. IO 端口

对于数字功能而言，PMC232 所有的双向输入/输出端口都可以使用数据寄存器（pa, pb, pc），控制寄存器（pac, pbc, pcc）和弱上拉电阻（paph, pbph, pcph）独立配置成不同的功能；所有这些引脚设置有施密特触发输入缓冲器和 CMOS 输出驱动电位水平。当这些引脚设定为输出低电位或开漏模式下，弱上拉电阻会自动关闭。如果要读取端口上的电位状态，一定要先设置成输入模式；在输出模式下，读取到的数据是数据寄存器的值。图 17 显示了 IO 缓冲区硬件图。除 PA.5 外所有其它的 IO 口具有相同的结构。

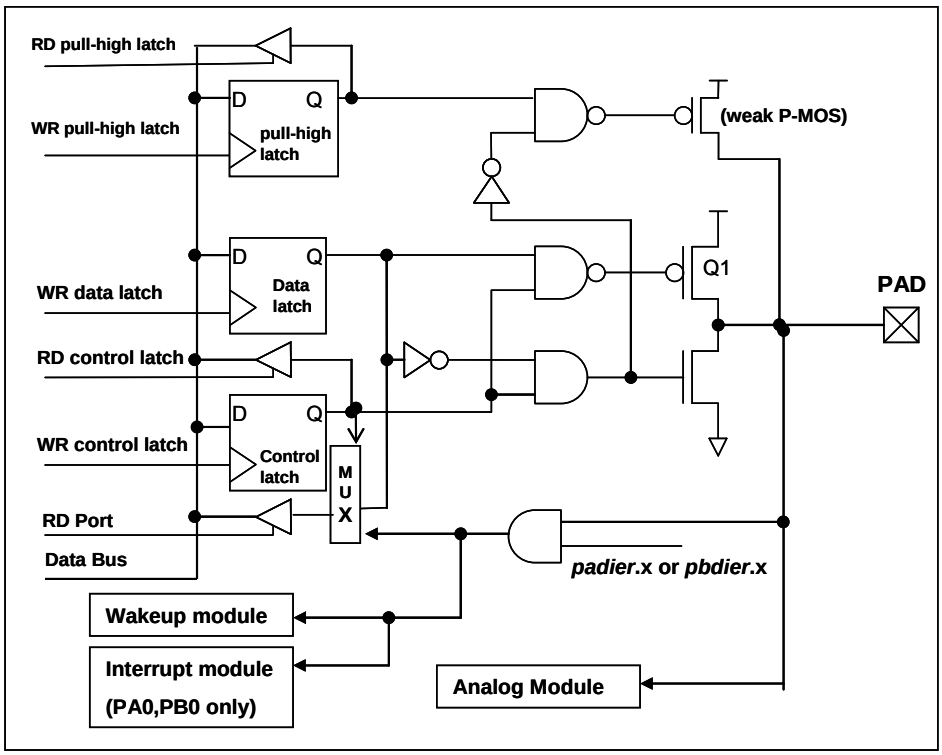


图 17 IO 缓冲区硬件图

使用 PA0 为例，表 6 显示了端口 A 位 0 的配置表。

pa.0	*pac.0*	*paph.0*	*paod.0*	功能描述
X	0	0	X	输入，没有弱上拉电阻
X	0	1	X	输入，有弱上拉电阻
0	1	X	0	输出低电位，没有弱上拉电阻（弱上拉电阻自动关闭）
1	1	0	0	输出高电位，没有弱上拉电阻
1	1	1	0	输出高电位，有弱上拉电阻

表 6：端口 A 位 0 的配置表

PA5 的输出只能是漏极开路模式（没有 Q1）。对于被选择为模拟功能的引脚，必须在寄存器 padier 以及 pbdier 相应位设置为低，以防止漏电流。当 PMC232 在掉电模式或者省电模式，每一个引脚都可以切换其状态来唤醒系统。对于需用来唤醒系统的引脚，必须设置为输入模式以及寄存器 padier 以及 pbdier 相应为高。同样的原因，当 PA0 或 PB0 用来作为外部中断引脚时，padier.0 或 pbdier.0 应设置为高。

5.14. 复位和LVR

5.14.1. 复位

复位 PMC232 有很多原因，包括：
（1）上电复位（POR）
（2）在正常运行下，PRST＃引脚活跃
（3）在正常运行下，WDT 定时器超时溢出
（4）电源电压下降突波 LVR

POR（Power-On-Reset 上电复位）是当上电时，把 PMC232 复位在初始状态；看门狗定时器超时溢出是执行软件下发生不正常的情况的复位，LVR 是用来侦测电源发生电压下降突波异常情况下的复位。一旦发生复位，大部份 PMC232 的寄存器将被设置为上电初始值，只有 ***gdio*** 寄存器（IO 地址 0x7）在看门狗超时是保存其内容不变。系统在出现异常情况应当重新启动，或跳跃程序计数器到 0x0 来解决的。当复位来自上电或 LVR 时，数据存储器处在不确定的状态，但若来自 PA5/PRST＃引脚复位或 WDT 超时溢位复位，内容将保持不变。

5.14.2. LVR

程序编译时，使用者可以选择 8 个不同级别的 LVR～4.1V，3.6V,3.1V,2.8V,2.5V,2.2V,2.0V,1.8V，通常情况下，使用者在选择 LVR 水平时，必须结合单片机工作频率和电源电压，以便让单片机稳定工作。

5.15. VDD/2 偏置电压

这项功能可以用寄存器 ***misc*** 位 4 来启用，PA3、PA2、PA1、PA0 这些引脚可以提供(VDD/2)电位输出，以做为驱动液晶显示器时 COM 的功能。当被选定的引脚希望有 VDD/2 电压功能时，使用者只需要将相对应的引脚设为输入模式，PMC232 将自动在该引脚产生 VDD/2 的电压。如果使用者想要输出高电位、VDD/2、GND 三个层次，只要分别设定为输出高电位产生 VDD、相应的引脚设定为输入并启用寄存器 misc 位 4 以产生 VDD/2、输出低电位产生 GND，即可产生三种相对应的电位。图 18 显示了如何使用此功能。

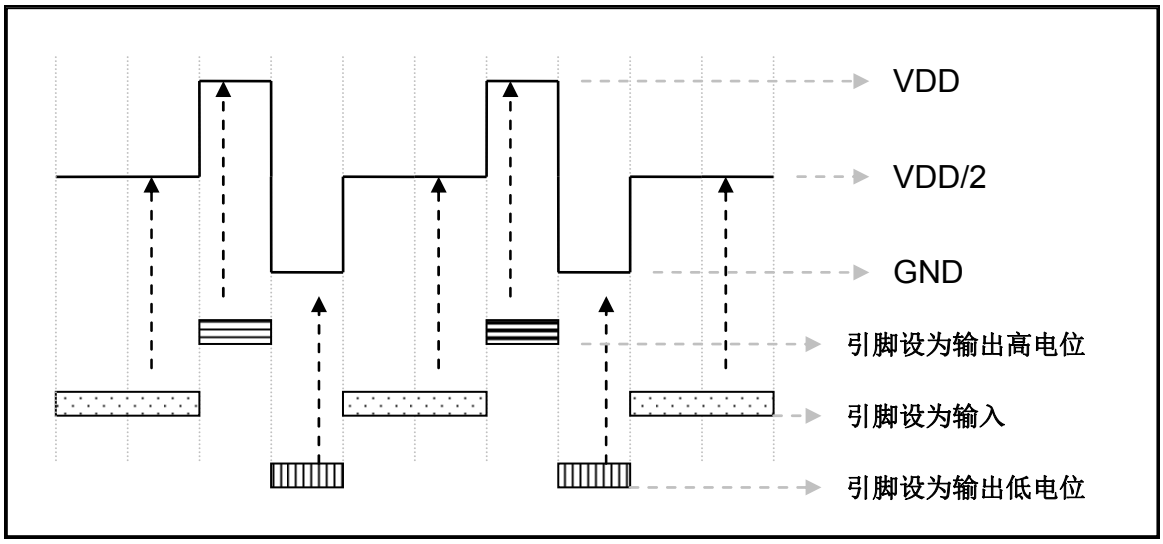


图 18 (VDD/2)偏置电压使用

5.16. 数字转换（ADC）模块

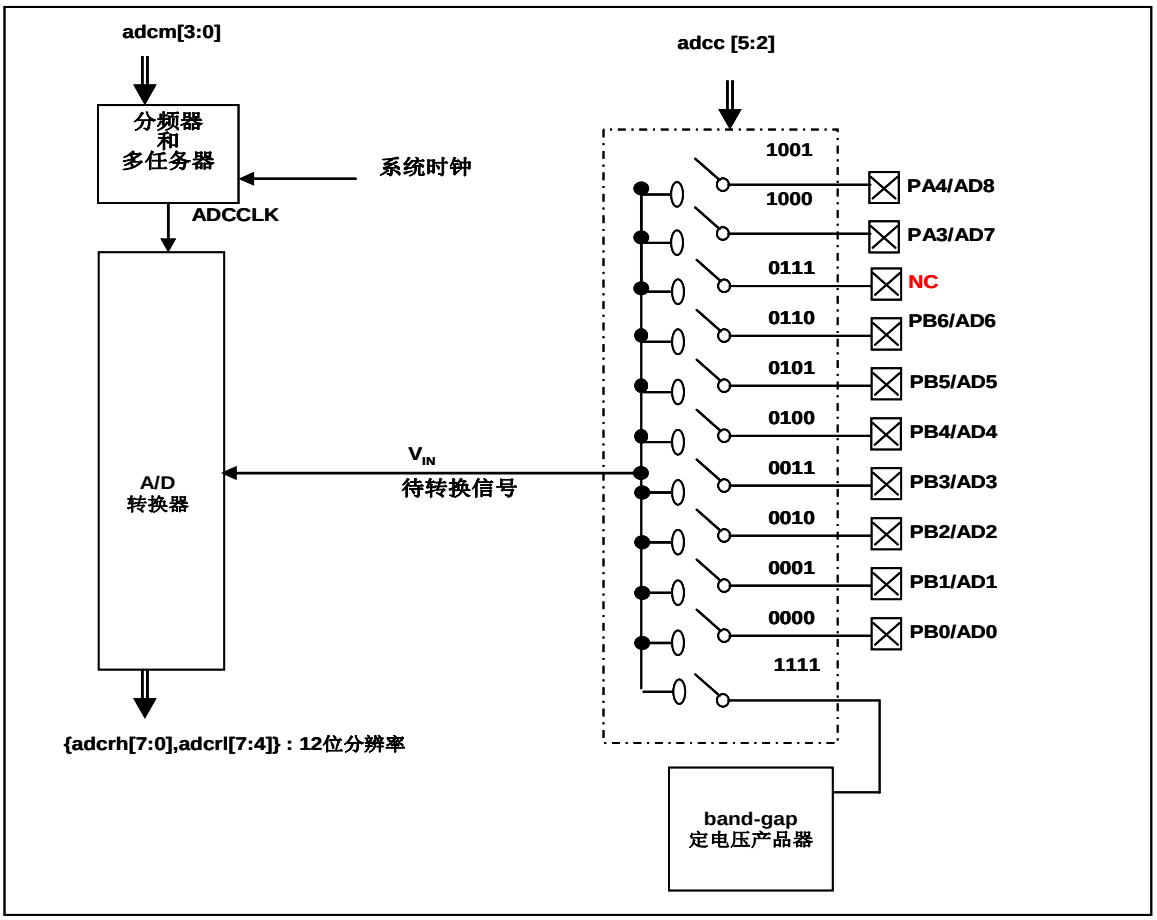


图 19 ADC 模块框图

ADC 模块有 6 个寄存器，分别是：

- ADC 控制寄存器 (***adcc***)
- ADC 模式控制寄存器 (***adcm***)
- ADC 数据高位/低位寄存器(***adcrh***, ***adcrl***)
- 端口 A/B 数字输入禁用寄存器 (***padier***, ***pbdier***)

做 AD 转换建议使用者遵守下面的步骤：

(1) ADC 模块的配置与设定：

- 利用 ***adcc*** 寄存器选择 ADC 输入通道
- 利用 ***adcm*** 寄存器配置 ADC 转换时钟以及分辨率
- 利用 ***padier***, ***pbdier*** 寄存器配置所选定的引脚作为模拟输入
- 利用 ***adcc*** 寄存器启用 ADC 模块

(2) 配置 ADC 模块的中断：（如果需要）

- 清零 ***intrq*** 寄存器位 3 的 ADC 中断请求标志
- 启用 ***inten*** 寄存器位 3 的 ADC 中断请求
- 利用 ***engint*** 指令启用全局中断

(3) 启动 ADC 转换：

- 利用 ***adcc*** 寄存器置位 ADC 转换过程控制位启动转换(***set1 adcc.6***)

(4) 等待完成 AD 转换标志位置位，方法可以用如下的任一种：

- 使用命令 ***wait1 addc.6*** 来等待完成的标志位置; 或
- 等待 ADC 的中断

(5) 读取 ADC 的数据寄存器

- 读取 ***adcrh***, ***adcrl*** 数据寄存器

(6) 下一个转换，依要求转到步骤 1 或第 2 步。

5.16.1. AD转换的输入要求

为了满足AD转换的准确性，电荷保持电容（C_{HOLD}）必须完全充电到参考高电压以及放电到参考低电压的水平。模拟输入电路模型图如图 20 所示，信号驱动源阻抗（R_S）和内部采样开关阻抗（R_{SS}）将影响到电荷保持电容C_{HOLD}充电所需要的时间。内部采样开关阻抗可能会因ADC的电源电压VDD有所变化，信号驱动源阻抗会影响到模拟输入信号的精度。用户必须确保在被测信号稳定时采样，因此，信号驱动源最大阻抗是与待量测信号频率有高度相关。建议在 500kHz输入频率和 10 位精确度条件下，模拟信号源的最高阻抗为 10KΩ；在 500Hz输入频率和 10 位精确度条件下，为 10MΩ。

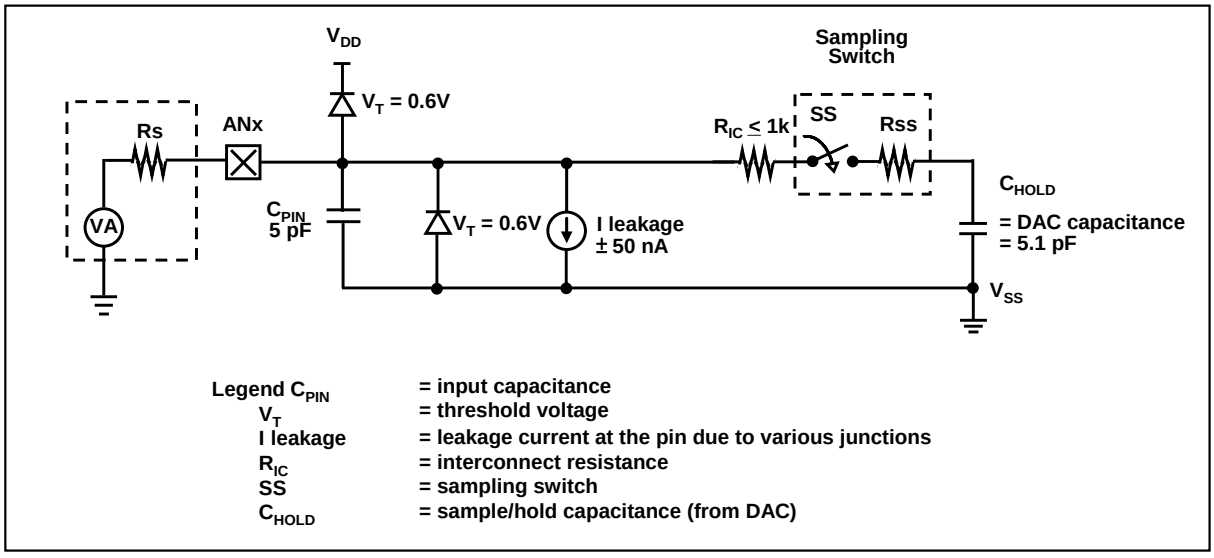


图 20. 模拟输入模型

在AD转换开始之前，必须确认所选模拟输入的信号采集时间应符合要求。PMC232 的ADC信号采集时间（T_{ACQ}）固定在一个ADCLK的时钟周期; ADCLK的选择必须满足最短信号采集时间。

5.16.2. ADC分辨率选择

ADC 的分辨率为 12 位。请在开始 AD 转换前先透过$ADCM 指令，把 adcm 寄存器的位[7：5]设置为“100“。高的分辨率可以检测小信号的变化，但是，它需要更多的时间把模拟信号转换为数字信号。位分辨率的选择可以通过 ***adcm*** 寄存器设置。ADC 的位分辨率设定应在 AD 转换开始之前配置。

5.16.3. ADC 时钟选择

ADC 模块的时钟（ADCLK）可以由设置 ***adcm*** 寄存器来选择，ADCLK 有 8 个可能的选择：从 CLK/1 到 CLK/128。由于信号采集时间 TACQ 是一个 ADCLK 时钟周期，所以该 ADCLK 必须满足这一要求。建议 ADC 模块时钟周期是 2us。

5.16.4. AD转换

AD转换的过程，从设置START/DONE（***adcc***位 6）为高开始，START/DONE的标志位内部将会自动清零，然后转换模拟信号将会一位一位的转换，当AD转换完成时，START/DONE将自动置高表示完成转换。当ADCLK被选定后，ADCLK的周期是T_{ADCLK}而AD转换的时间将是如下:

- 12 位分辨率: AD转换时间 = 17 T_{ADCLK}

5.16.5. 模拟引脚的配置

模数转换器的 10 个模拟输入信号与端口 PA3、PA4、PB[6:0]共享引脚，开机后这些引脚是设置在数字信号模式，为了避免在设置为数字电路时发生漏电流，这些引脚在当模拟输入时一定要利用寄存器 ***padier***, ***pbdier*** 设置为模拟输入。对于那些定义为模拟输入的引脚，当读 port A、B 时，其值将为 0。

ADC 的测量信号属于小信号，为避免测试信号在测量期间被干扰，被选定的引脚应该（1）被设置为输入模式（2）关闭弱上拉电阻高（3）利用 ***padier***, ***pbdier*** 寄存器配置所选定的引脚作为模拟输入。

5.16.6. 使用ADC

下面的示例演示使用 PB0～PB3 来当 ADC 输入引脚。

首先，定义所选择的引脚：

```
PBC     = 0B_XXXX_0000;  // PB0 ~ PB3 当输入
PBPH    = 0B_XXXX_0000;  // PB0 ~ PB3 没有弱上拉电阻
PBDIER  = 0B_XXXX_0000;  // PB0 ~ PB3 数据输入禁用
```

下一步，设定 ADCC 寄存器，示例如下：

```
$   ADCC   Enable, PB3;        // 设定 PB3 当 ADC 输入
$   ADCC   Enable, PB2;        // 设定 PB2 当 ADC 输入
$   ADCC   Enable, PB0;        // 设定 PB0 当 ADC 输入
```

下一步，设定 ***ADCM*** 寄存器，示例如下：

```
$   ADCM   12BIT, /16;         // 建议 /16 @系统时钟=8MHz
$   ADCM   12BIT, /8;          // 建议 /8 @系统时钟=4MHz
```

接着，开始 ADC 转换：

```
AD_START =  1;                 // 开始 ADC 转换
WAIT1       AD_DONE    ;       // 等待 ADC 转换结果
```

最后，当 AD_DONE 高电位时读取 ADC 结果：

```
WORD       Data;               // 2 字节结果: ADCRH 和 ADCRL
Data       =  (ADCRH << 8) | ADCRL;
```

ADC 也可以利用下面方法禁用：

```
$   ADCC   Disable;
```

或

```
ADCC     =  0;
```

6. IO寄存器

6.1. 算术逻辑状态寄存器（*flag*），IO 地址 = 0x00

位	初始值	读/写	描 述
7 - 4	-	-	保留。这 4 个位读值为“1”。
3	0	读/写	OV（溢出标志）。当数学运算溢出时，这一位会设置为 1。
2	0	读/写	AC（辅助进位标志）。两个条件下，此位设置为 1：（1）进行低半字节加法运算产生进位 （2）减法运算时，低半字节向高半字节借位。
1	0	读/写	C（进位标志）。有两个条件下，此位设置为 1：（1）加法运算产生进位（2）减法运算有借位。进位标志还受带进位标志的 shift 指令影响。
0	0	读/写	Z（零）。此位将被设置为 1，当算术或逻辑运算的结果是 0；否则将被清零。

6.2. FPP单元允许寄存器（*fppen*），IO地址 = 0x01

位	初始值	读/写	描 述
7 – 2	-	-	保留。
1	0	读/写	FPP1 启用。此位是用来启用 FPP1。 0/1：禁用/启用
0	1	读/写	FPP0 启用。此位是用来启用 FPP0。 0/1：禁用/启用

6.3. 堆栈指针寄存器（*sp*），IO地址 = 0x02

位	初始值	读/写	描 述
7 - 0	-	读/写	堆栈指针寄存器。读出当前堆栈指针，或写入以改变堆栈指针。

6.4. 时钟控制寄存器（*clkmd*），IO地址 = 0x03

位	初始值	读/写	描 述	
7 – 5	111	读/写	系统时钟选择	
			类型 0, clkmd[3]=0	类型 1, clkmd[3]=1
			000：内部高频 RC 振荡器时钟(IHRC) ÷ 4 001：内部高频 RC 振荡器时钟(IHRC) ÷ 2 010：内部高频 RC 振荡器时钟 (IHRC) 011：外部振荡器时钟(EOSC) ÷ 4 100：外部振荡器时钟(EOSC) ÷ 2 101：外部振荡器时钟(EOSC) 110：内部低频 RC 振荡器时钟(ILRC) ÷ 4 111：内部低频 RC 振荡器时钟(ILRC) (默认)	000：内部高频 RC 振荡器时钟(IHRC) ÷ 16 001：内部高频 RC 振荡器时钟(IHRC) ÷ 8 010：保留 011：内部高频 RC 振荡器时钟(IHRC)÷ 32 100：保留 101：外部振荡器时钟(EOSC) ÷ 8 11X：保留
4	1	读/写	内部高频 RC 振荡器功能。 0/1：禁用/启用	
3	0	读/写	时钟类型选择。此位用来选择开机后寄存器 clkmd 位[7:5]的时钟类型。 0 / 1: 类型 0 /类型 1	
2	1	读/写	内部低频 RC 振荡器功能。0/1：禁用/启用	
1	1	读/写	看门狗定时器功能。 0/1：禁用/启用	
0	0	读/写	引脚 PA5/PRST# 功能. 0 / 1: PA5 / PRST#.	

6.5. 中断允许寄存器（*inten*），IO地址 = 0x04

位	初始值	读/写	描 述
7	-	读/写	保留。
6	0	读/写	启用从 Timer2 的溢出中断。0/1：禁用/启用
5 : 4	-	-	保留。
3	0	读/写	启用从模数转换器的中断。 0/1：禁用/启用
2	0	读/写	启用从 Timer16 的溢出中断。0/1：禁用/启用
1	0	读/写	启用从 PB0 的中断。 0/1：禁用/启用
0	0	读/写	启用从 PA0 的中断。 0/1：禁用/启用

6.6. 中断请求寄存器（*intrq*），IO地址 = 0x05

位	初始值	读/写	描 述
7	-	读/写	保留。
6	-	读/写	Timer2 的中断请求，此位是由硬件置位并由软件清零。 0/1：不要求/请求
5 : 4	-	-	保留。
3	-	读/写	模拟数字转换器的中断请求，此位是由硬件置位并由软件清零。0/1：不要求/请求
2	-	读/写	Timer16 的中断请求，此位是由硬件置位并由软件清零。0/1：不要求/请求
1	-	读/写	PB0 的中断请求，此位是由硬件置位并由软件清零。0/1：不要求/请求
0	-	读/写	PA0 的中断请求，此位是由硬件置位并由软件清零。0/1：不要求/请求

6.7. Timer16 控制寄存器（*t16m*），IO地址 = 0x06

位	初始值	读/写	描 述
7 - 5	000	读/写	Timer16 时钟选择 000：禁用 001：系统时钟 010：保留 011：保留 100：IHRC 101：外部振荡器时钟 (EOSC) 110：内部低频 RC 振荡器时钟 111：PA0
4 – 3	00	读/写	Timer16 内部的时钟分频器 00： ÷1 01： ÷4 10： ÷16 11： ÷64
2 – 0	000	读/写	中断源选择。当选择位由低变高时，发生中断事件。 0 ： Timer16 位 8 1 ： Timer16 位 9 2 ： Timer16 位 10 3 ： Timer16 位 11 4 ： Timer16 位 12 5 ： Timer16 位 13 6 ： Timer16 位 14 7 ： Timer16 位 15

6.8. 通用数据输入/输出寄存器（*gdio*），IO地址 = 0x07

位	初始值	读/写	描 述
7 – 0	00	读/写	这个端口是IO空间的数据缓冲区，它只有在POR、LVR或引脚PRST＃作动行时被清零。看门狗超时复位时它的值不会被改变。它是在IO空间进行操作，如wait0 gdio.x，wait1 gdio.x和tog gdio.x用以取代记忆空间的指令（例如：wait1 mem; wait0 mem; tog mem）。

6.9. 外部晶体振荡器控制寄存器（eoscr），IO地址 = 0x0a

位	初始值	读/写	描 述
7	0	只写	启用外部晶体振荡器。 0/1：禁用/启用
6 – 5	00	只写	晶体振荡器的选择。 00：保留 01：低驱动电流。适用于较低频率晶体，例如：32kHz 10：中驱动电流。适用于中等频率晶体，例如：1MHz 11：高驱动电流。适用于较高频率晶体，例如：4MHz
4 – 1	-	-	保留。将来的兼容性，请保留 0。
0	0	只写	将 Band-gap 与 LVR 硬件模块掉电处理。0/1： 正常 / 掉电

6.10. 内部高频RC振荡器控制寄存器（ihrcr，只写），IO地址 = 0x0b

位	初始值	读/写	描 述
7 – 0	00	只写	内部高频 RC 振荡器的频率校准 这个寄存器是给系统频率校准用的，使用者请勿自行写值

6.11. 中断边沿选择寄存器（integs，只写），IO地址 = 0x0c

位	初始值	读/写	描 述
7 – 5	-	-	保留。
4	0	只写	Timer16 中断沿选择。 0： 被选定的位上升沿触发中断 1： 被选定的位下降沿触发中断
3 – 2	00	只写	PB0 中断沿选择。 00： 被选定的位上升沿和下降沿都触发中断 01： 被选定的位上升沿触发中断 10： 被选定的位下降沿触发中断 11： 保留
1 – 0	00	只写	PA0 中断沿选择。 00： 被选定的位上升沿和下降沿都触发中断 01： 被选定的位上升沿触发中断 10： 被选定的位下降沿触发中断 11： 保留

6.12. 端口A数字输入禁止寄存器（*padier*，只写），IO地址 = 0x0d

位	初始值	读/写	描 述
7	1	只写	禁用 PA7 数字输入以及唤醒功能。 1/0：启用/禁用。 当使用外部晶振时，此位应设为 0 用以防止漏电流。当选择禁用时，这个引脚的唤醒功能也被禁用。
6	1	只写	禁用 PA6 数字输入以及唤醒功能。 1/0：启用/禁用。 当使用外部晶振时，此位应设为 0 用以防止漏电流。当选择禁用时，这个引脚的唤醒功能也被禁用。
5	1	只写	禁用 PA5 唤醒功能。 1/0：启用/禁用。 此位设为 0 用来禁用 PA5 的唤醒功能。
4	1	只写	禁用 PA4 数字输入以及唤醒功能。 1/0：启用/禁用。 此位设为 0 用来防止引脚当模拟输入时漏电。当选择禁用时，从这个引脚的唤醒功能也被禁用。
3	1	只写	禁用 PA3 数字输入以及唤醒功能。 1/0：启用/禁用。 此位设为 0 用来防止引脚当模拟输入时漏电。当选择禁用时，从这个引脚的唤醒功能也被禁用。
2	1	只写	禁用 PA2 唤醒功能。 1/0：启用/禁用。 此位设为 0 用来禁用 PA2 的唤醒功能。
1	1	只写	保留
0	1	只写	禁用 PA0 唤醒功能。 1/0：启用/禁用。 此位设为 0 用来禁用 PA0 的唤醒功能和外部中断。

注意：
因为这个寄存器的控制极性在仿真板和 IC 上是相反的，为了保证在仿真和生产时程序的一致，请用下面的方法对这个寄存器进行写操作：

```
$PADIER  0xhh
```

例如：

```
$PADIER  0xF0;
```

使能 PA[7：4]的数字输入功能和唤醒功能，IDE 会自动识别仿真器和 IC。

6.13. 端口B数字输入禁止寄存器（*pbdier*，只写），IO地址 = 0x0e

位	初始值	读/写	描 述
7	1	只写	保留
6 - 0	0xEF	只写	禁用 PB6～PB0 数字输入，以防止引脚当模拟输入时漏电。当选择禁用时，从这个引脚的唤醒功能也被禁用。1/0：启用/禁用。

注意：
因为这个寄存器的控制极性在仿真板和 IC 上是相反的，为了保证在仿真和生产时程序的一致，请用下面的方法对这个寄存器进行写操作：

```
$PBDIER  0xhh
```

例如：

```
$PBDIER  0xF0;
```

使能 PA[6：4]的数字输入功能和唤醒功能，IDE 会自动识别仿真器和 IC。

6.14. 端口A数据寄存器（*pa*），IO地址 = 0x10

位	初始值	读/写	描 述
7 – 0	0x00	读/写	端口 A 数据寄存器

6.15. 端口A控制寄存器（*pac*），IO地址 = 0x11

位	初始值	读/写	描 述
7 - 0	0x00	读/写	端口 A 控制寄存器。这个寄存器是用来定义端口 A 每个相应的引脚的输入模式或输出模式。 0/1：输入/输出 请注意，PA5 可以当输入；但是当输出时，只能输出低；若输出高，它将是高阻抗状态（high impedance）。

6.16. 端口A上拉控制寄存器（*paph*），IO地址 = 0x12

位	初始值	读/写	描 述
7 - 0	0x00	读/写	端口 A 上拉控制寄存器。这个寄存器是用来控制端口 A 每个相应引脚的内部上拉功能。 0/1：禁用/启用 请注意：端口 A 位 5（PA5）没有上拉电阻。

6.17. 端口B数据寄存器（*pb*），IO地址 = 0x14

位	初始值	读/写	描 述
7 – 0	0x00	读/写	端口 B 数据寄存器。

6.18. 端口B控制寄存器（*pbc*），IO地址 = 0x15

位	初始值	读/写	描 述
7 – 0	0x00	读/写	端口 B 控制寄存器。这个寄存器是用来定义端口 B 每个相应的引脚的输入模式或输出模式。 0/1：输入/输出

6.19. 端口B上拉控制寄存器（*pbph*），IO地址 = 0x16

位	初始值	读/写	描 述
7 – 0	0x00	读/写	端口 B 上拉控制寄存器。这个寄存器是用来控制端口 B 每个相应引脚的内部上拉功能。 0/1：禁用/启用

6.20. 端口C数据寄存器（*pc*），IO地址 = 0x17

位	初始值	读/写	描 述
7 – 0	0x00	读/写	端口 C 数据寄存器。

6.21. 端口C控制寄存器（*pcc*），IO地址 = 0x18

位	初始值	读/写	描 述
7 – 0	0x00	读/写	端口 C 控制寄存器。这个寄存器是用来定义端口 C 每个相应的引脚的输入模式或输出模式。 0/1：输入/输出。

6.22. 端口C上拉控制寄存器（*pcph*），IO地址 = 0x19

位	初始值	读/写	描 述
7 – 0	0x00	读/写	端口 C 上拉控制寄存器。这个寄存器是用来控制端口 C 每个相应引脚的内部上拉功能。 0/1：禁用/启用

6.23. ADC 控制寄存器（*adcc*），IO地址 = 0x20

位	初始值	读/写	描 述
7	0	读/写	启用的 ADC 功能。0/1：禁用/启用
6	0	读/写	模数转换器过程控制位。 写“1”开始 AD 转换，同时自动清零完成标志。 读到“1”表示完成 AD 的转换。
5 – 2	0000	读/写	通道选择器。这 4 个位用于选择 AD 转换的输入信号。 0000: PB0/AD0, 0001: PB1/AD1, 0010: PB2/AD2, 0011: PB3/AD3, 0100: PB4/AD4, 0101: PB5/AD5, 0110: PB6/AD6, 0111: 保留 1000: PA3/AD7 1001: PA4/AD8 1111: bandgap 1.20 volt 参考电位 其它：保留。
1 – 0	-	-	保留。

6.24. ADC 模式控制寄存器（*adcm*，只写），IO地址 = 0x21

位	初始值	读/写	描 述
7 – 5	000	只写	位分辨率。 100:12-bit, AD 12-bit result [11:0] = { adcrh[7:0], adcrl[7:4] }. 其它：保留。
4	-	-	保留。
3 – 1	000	只写	ADC 时钟源的选择。 000：CLK÷1， 001：CLK÷2， 010：CLK÷4， 011：CLK÷8， 100：CLK÷16， 101：CLK÷32， 110：CLK÷64， 111：CLK÷128
0	-	-	保留。

6.25. ADC 数据高位寄存器（*adcrh*，只读），IO地址 = 0x22

位	初始值	读/写	描 述
7 – 0	-	只读	这 8 个只读位是 AD 转换的结果的位[11：4]。这个寄存器的位 7 是 ADC 转换结果的最高位。

6.26. ADC 数据低位寄存器（*adcrl*，只读），IO地址 = 0x23

位	初始值	读/写	描 述
7 – 4	-	只读	这 4 个只读位是 AD 转换的结果的位[3：0]。
3 – 0	-	-	保留。

6.27. 杂项寄存器 (*misc*)，IO地址 = 0X3b

位	初始值	读/写	描 述
7	-	-	保留。将来的兼容性，请保留 0。
6	0	只写	启用外部晶体振荡器的高驱动电流。0 / 1 ：启用/禁用。 当此位被启用，可以加速晶体振荡器稳定震荡，开机时间也可以加快；但是，功耗会变大。 当振荡器运行正常和稳定之后，为降低功耗，这一位可以被禁用。
5	0	WO	快唤醒功能。 0: 正常唤醒。唤醒时间为 1024 ILRC 时钟。 1: 快唤醒。 假如系统时钟用 IHRC：唤醒时间为 128 个系统时钟。 假如系统时钟用晶体振荡器：唤醒时间为 1024 个系统时钟+晶体振荡器稳定时间 请注意：当启用快速唤醒时，看门狗时钟源会切换到系统时钟(例如：4MHz)，所以，建议要进入掉电模式前，打开快速唤醒之前要关闭看门狗定时器，等系统被唤醒后，在关闭快速唤醒之后再打开看门狗定时器。
4	0	只写	启用 PA3, PA2, PA0, PC5 偏压在 VDD/2 假如这些引脚设成输入。 0 / 1 ：禁用/启用。
3	0	只写	LVR 时间 0：正常。LVR 开机时间为 1024 ILRC 时钟周期。 1：快速。LVR 开机时间为 64ILRC。
2	0	只写	LVR 功能控制 0/1 ： 启用/禁用
1-0	00	只写	看门狗溢出时间设置 00：2048 ILRC 时钟周期 01：4096 ILRC 时钟周期 10：16384 ILRC 时钟周期 11：256 ILRC 时钟周期

6.28. Timer2 控制寄存器 (tm2c), IO地址 = 0x3c

位	初始值	读/写	描 述
7 – 4	0000	读/写	Timer2 时钟选择。 0000： 禁用 0001： 系统时钟 0010 : IHRC 时钟 0011: 保留 0100 : ILRC 时钟 0101： 保留 011X： 保留 1000 : PA0 (上升沿) 1001 : ~PA0 (下降沿) 1010 : PA3 (上升沿) 1011 : ~PA3 (下降沿) 1100 : PA4 (上升沿) 1101 : ~PA4 (下降沿) 注意：在ICE模式且IHRC被选为Timer2 定时器时钟，当ICE停下时，发送到定时器的时钟是不停止，定时器仍然继续计数。
3 – 2	00	读/写	Timer2 输出选择。 00： 禁用 01 : PA2 10 : PA3 11： 保留
1	0	读/写	Timer2 模式选择。 0： 周期模式 1 : PWM 模式
0	0	读/写	启用 Timer2 反极性输出。 0 / 1 ： 禁用/启用。

6.29. Timer2 计数寄存器 (tm2ct), IO地址 = 0x3d

位	初始值	读/写	描 述
7 – 0	0x00	读/写	Timer2 定时器位[7:0]。

6.30. Timer2 分频器寄存器 (tm2s), IO 地址 = 0x37

位	初始值	读/写	描 述
7	0	只写	PWM 分辨率选择。 0 : 8 位 1 : 6 位
6 – 5	00	只写	Timer2 时钟预分频器。 00 : ÷ 1 01 : ÷ 4 10 : ÷ 16 11 : ÷ 64
4 – 0	00000	只写	Timer2 时钟分频器。

6.31. Timer2 上限寄存器 (tm2b), IO地址= 0x09

位	初始值	读/写	描 述
7 – 0	0x00	只写	Timer2 上限寄存器。 请注意此寄存器不能为 0。

7. 指令

符 号	描 述
ACC	累加器（ACC 是累加器的简写，用来避免与程序的 a 混淆)
a	累加器 (a 是程序使用累加器的符号)
sp	堆栈指针
flag	累加器状态标志寄存器
I	即时数据
&	逻辑 AND
|	逻辑 OR
←	移动
^	异或 OR
+	加
—	减
～	按位取反（逻辑补数，1 补数）
∓	负数 （2 补数）
OV	溢出（2 补数系统的运算结果超出范围）
Z	零（如果零运算单元操作的结果是 0，这位设置为 1）
C	进位（Carry）
AC	辅助进位标志（Auxiliary Carry）。
pc0	FPP0 的程序计数器
pc1	FPP1 的程序计数器

7.1. 数据传输类指令

mov **a, I**	移动即时数据到累加器 例如： *mov* a, 0x0f; 结果： a ← 0fh; 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
mov **M, a**	移动数据由累加器到存储器 例如： *mov* MEM, a; 结果： MEM ← a 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
mov **a, M**	移动数据由存储器到累加器 例如： *mov* a, MEM ; 结果： a ← MEM; 当 MEM 为零时，标志位 Z 会被置位。 受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』
mov **a, IO**	移动数据由IO到累加器 例如： *mov* a, pa ; 结果： a ← pa; 当 pa 为零时，标志位 Z 会被置位。 受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』

mov **IO, a**	移动数据由累加器到IO 例如： *mov* pb, a; 结果： pb ← a ; 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』

nmov **M, a**	取累加器的负逻辑（2 补数）并复制到存储器。
	例如： n*mov* MEM, a;
	结果： MEM ← a 的 2 补码
	受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
	应用范例：
	--- 	
	mov a, 0xf5 ; // ACC=0xf5
	nmov ram9, a; // ram9=0x0b，ACC=0xf5
	--- 	

nmov **a, M**	取存储器的负逻辑（2 补数）并复制到累加器。
	例如： n*mov* a, MEM ;
	结果： a ←MEM 的 2 补码; 当 MEM 的 2 补码为零时，标志位 Z 会被置位。
	受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』
	应用范例：
	--- 	
	mov a, 0xf5 ;
	mov ram9, a ; // ram9=0xf5
	nmov a, ram9 ; // ram9=0xf5, ACC=0x0b
	--- 	

ldtabh **index**	使用索引作为 OTP 的地址将 OTP 程序存储器的高字节数据读取并载入到累加器。需要 2T 时间执行这一指令。
	例如： *ldtabh* index;
	结果： a ← {bit 15~8 of OTP [index]};
	受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
	应用范例：
	--- 	
	word ROMptr ; // 在 RAM 定义 OTP 的指针
	…
	mov a, la@TableA ; // 指定 OTP TableA 指针 （LSB）
	mov lb@ROMptr, a ; // 将指针存到 RAM（LSB）
	mov a, ha@TableA ; // 指定 OTP TableA 指针(MSB)
	mov hb@ROMptr, a ; // 将指针存到 RAM (MSB)
	…
	ldtabh ROMptr ; // 读取数据并载入到累加器 (ACC=0X02)
	….
	TableA : dc 0x0234, 0x0042, 0x0024, 0x0018 ;
	--- 	

ldtabl **index**	使用索引作为 OTP 的地址并将 OTP 程序存储器的低字节数据读取并载入到累加器。需要 2T 时间执行这一指令。
	例如： *ldtabl* index;
	结果： a ← {bit7~0 of OTP [index]};
	受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
	应用范例：
	--- 	
	word ROMptr ; // 在 RAM 定义 OTP 的指针
	…
	mov a, la@TableA ; // 指定 OTP TableA 指针（LSB）
	mov lb@ROMptr, a ; // 将指针存到 RAM （LSB）
	mov a, ha@TableA ; // 指定 OTP TableA 指针 （MSB）
	mov hb@ROMptr, a ; // 将指针存到 RAM （MSB）
	…
	ldtabl ROMptr ; // 读取数据并载入到累加器 （ACC=0x34）
	….
	TableA : dc 0x0234, 0x0042, 0x0024, 0x0018 ;
	--- 	

ldt16 **word**	将 Timer16 的 16 位计算值复制到 RAM。
	例如： *ldt16* word;
	结果： word ← 16-bit timer
	受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
	应用范例：
	--- 	
	word T16val ; // 定义一个 RAM word
	…
	clear lb@T16val ; // 清零 T16val（LSB）
	clear hb@T16val ; // 清零 T16val（MSB）
	stt16 T16val ; // 设定 Timer16 的起始值为 0
	…
	set1 t16m.5 ; // 启用 Timer16
	…
	set0 t16m.5 ; // 禁用 Timer16
	ldt16 T16val ; // 将 Timer16 的 16 位计算值复制到 RAM T16val
	….
	--- 	

stt16 **word**	将放在 word 的 16 位 RAM 复制到 Timer16。 例如： *stt16* word; 结果： 16-bit timer ← word 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』

	应用范例：
	--- 	
	word T16val ; // 定义一个 RAM word
	…
	mov a, 0x34 ;
	mov lb@T16val, a ; // 将 0x34 搬到 T16val（LSB）
	mov a, 0x12 ;
	mov hb@T16val, a ; // 将 0x12 搬到 T16val（MSB）
	stt16 T16val ; // Timer16 初始化 0x1234
	…
	--- 	

xch **M**	累加器与 RAM 之间交换数据 例如： *xch* MEM ; 结果： MEM ← a , a ← MEM 受影响的标志位： Z：『不变』， C：『不变』， AC：『不变』， OV：『不变』

idxm **a, index**	使用索引作为 RAM 的地址并将 RAM 的数据读取并载入到累加器。需要 2T 时间执行这一指令。
	例如： *idxm* a, index;
	结果： a ← [index], index 是用 word 定义。
	受影响的标志位： Z：『不变』， C：『不变』， AC：『不变』， OV：『不变』
	应用范例：
	--- 	
	word RAMIndex ; // 定义一个 RAM 指针
	…
	mov a, 0x5B ; // 指定指针地址（LSB）
	mov lb@RAMIndex, a ; // 将指针存到 RAM（LSB）
	mov a, 0x00 ; // 指定指针地址为 0x00（MSB），
	mov hb@RAMIndex, a ; // 将指针存到 RAM（MSB）
	…
	idxm a, RAMIndex ; // 将 RAM 地址为 0x5B 的数据读取并载入累加器
	--- 	

Idxm **index, a**	使用索引作为 RAM 的地址并将累加器的数据读取并载入到 RAM。需要 2T 时间执行这一指令。
	例如：*idxm* index, a;
	结果： [index] ← a; index 是以 word 定义.
	受影响的标志位： Z：『不变』， C：『不变』， AC：『不变』， OV：『不变』
	应用范例：
	--- 	
	word RAMIndex ; // 定义一个 RAM 指针
	…
	mov a, 0x5B ; // 指定指针地址（LSB）
	mov lb@RAMIndex, a ; // 将指针存到 RAM（LSB）
	mov a, 0x00 ; // 指定指针地址为 0x00（MSB）
	mov hb@RAMIndex, a ; // 将指针存到 RAM（MSB）
	…
	mov a, 0xA5 ;
	idxm RAMIndex, a ; // 将累加器数据读取并载入地址为 0x5B 的 RAM
	--- 	

pushaf	将累加器和算术逻辑状态寄存器的数据存到堆栈指针指定的堆栈存储器
	例如：*pushaf*;
	结果：　　[sp] ← {flag, ACC};
	　　sp ← sp + 2 ;
	受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
	应用范例：
	--- 	
	.romadr 0x10 ; // 中断服务程序入口地址
	pushaf ; // 将累加器和算术逻辑状态寄存器的资料存到堆栈存储器
	… // 中断服务程序
	… // 中断服务程序
	popaf ; // 将堆栈存储器的资料回存到累加器和算术逻辑状态寄存器
	reti ;
	--- 	

popaf	将堆栈指针指定的堆栈存储器的数据回传到累加器和算术逻辑状态寄存器 例如：*popaf*; 结果：　　sp ← sp - 2 ; 　　{Flag, ACC} ← [sp] ; 受影响的标志位：Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』

7.2. 算术运算类指令

add **a, I**	将立即数据与累加器相加，然后把结果放入累加器 例如： *add* a, 0x0f ; 结果： a ← a + 0fh 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
add **a, M**	将 RAM 与累加器相加，然后把结果放入累加器 例如： *add* a, MEM ; 结果： a ← a + MEM 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
add **M, a**	将 RAM 与累加器相加，然后把结果放入 RAM 例如： *add* MEM, a; 结果： MEM ← a + MEM 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
addc **a, M**	将 RAM、累加器以及进位相加，然后把结果放入累加器 例如： *addc* a, MEM ; 结果： a ← a + MEM + C 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
addc **M, a**	将 RAM、累加器以及进位相加，然后把结果放入 RAM 例如： *addc* MEM, a ; 结果： MEM ← a + MEM + C 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
addc **a**	将累加器与进位相加，然后把结果放入累加器 例如： *addc* a ; 结果： a ← a + C 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』

addc **M**	将 RAM 与进位相加，然后把结果放入 RAM 例如： *addc* MEM ; 结果： MEM ← MEM + C 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
nadd **a, M**	将累加器的负逻辑（2 补码）与 RAM 相加，然后把结果放入累加器 例如： *nadd* a, MEM ; 结果： a ←a 的 2 补码 + MEM 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
nadd **M, a**	将 RAM 的负逻辑（2 补码）与累加器相加，然后把结果放入 RAM 例如： *nadd* MEM, a ; 结果： MEM ← MEM 的 2 补码 + a 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
sub **a, I**	累加器减立即数据，然后把结果放入累加器 例如： *sub* a, 0x0f; 结果： a ← a - 0fh (a + [2’s complement of 0fh]) 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
sub **a, M**	累加器减 RAM，然后把结果放入累加器 例如： *sub* a, MEM ; 结果： a ← a - MEM (a + [2’s complement of M]) 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
sub **M, a**	RAM 减累加器，然后把结果放入 RAM 例如： *sub* MEM, a; 结果： MEM ← MEM - a (MEM + [2’s complement of a]) 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
subc **a, M**	累加器减 RAM，再减进位，然后把结果放入累加器 例如： *subc* a, MEM; 结果： a ← a － MEM - C 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
subc **M, a**	RAM 减累加器，再减进位，然后把结果放入 RAM 例如： *subc* MEM, a ; 结果： MEM ← MEM － a - C 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
subc **a**	累加器减进位，然后把结果放入累加器 例如： *subc* a; 结果： a ← a - C 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
subc **M**	RAM 减进位，然后把结果放入 RAM 例如： *subc* MEM; 结果： MEM ← MEM - C 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
inc **M**	RAM 加 1 例如： *inc* MEM ; 结果： MEM ← MEM + 1 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
dec **M**	RAM 减 1 例如： *dec* MEM; 结果： MEM ← MEM - 1 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』

clear **M**	清除 RAM 为 0 例如： *clear* MEM ; 结果： MEM ← 0 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』

7.3. 移位运算类指令

sr **a**	累加器的位右移，位 7 移入值为 0 例如： *sr* a ; 结果： a (0,b7,b6,b5,b4,b3,b2,b1) ← a (b7,b6,b5,b4,b3,b2,b1,b0), C ← a(b0) 受影响的标志位： Z:『不变』， C:『受影响』， AC:『不变』， OV:『不变』
src **a**	累加器的位右移，位 7 移入进位标志位 例如： *src* a ; 结果： a (c,b7,b6,b5,b4,b3,b2,b1) ← a (b7,b6,b5,b4,b3,b2,b1,b0), C ← a(b0) 受影响的标志位： Z:『不变』， C:『受影响』， AC:『不变』， OV:『不变』
sr **M**	RAM 的位右移，位 7 移入值为 0 例如： *sr* MEM ; 结果： MEM(0,b7,b6,b5,b4,b3,b2,b1) ← MEM(b7,b6,b5,b4,b3,b2,b1,b0), C ← MEM(b0) 受影响的标志位： Z:『不变』， C:『受影响』， AC:『不变』， OV:『不变』
src **M**	RAM 的位右移，位 7 移入进位标志位 Example: *src* MEM ; 结果： MEM(c,b7,b6,b5,b4,b3,b2,b1) ← MEM (b7,b6,b5,b4,b3,b2,b1,b0), C ← MEM(b0) 受影响的标志位： Z:『不变』， C:『受影响』， AC:『不变』， OV:『不变』
sl **a**	累加器的位左移，位 0 移入值为 0 例如： *sl* a ; 结果： a (b6,b5,b4,b3,b2,b1,b0,0) ← a (b7,b6,b5,b4,b3,b2,b1,b0), C ← a (b7) 受影响的标志位： Z:『不变』， C:『受影响』， AC:『不变』， OV:『不变』
slc **a**	累加器的位左移，位 0 移入进位标志位 例如： *slc* a ; 结果： a (b6,b5,b4,b3,b2,b1,b0,c) ← a (b7,b6,b5,b4,b3,b2,b1,b0), C ← a(b7) 受影响的标志位： Z:『不变』， C:『受影响』， AC:『不变』， OV:『不变』
sl **M**	RAM 的位左移，位 0 移入值为 0 例如： *sl* MEM ; 结果： MEM (b6,b5,b4,b3,b2,b1,b0,0) ← MEM (b7,b6,b5,b4,b3,b2,b1,b0), C ← MEM(b7) 受影响的标志位： Z:『不变』， C:『受影响』， AC:『不变』， OV:『不变』
slc **M**	RAM 的位左移，位 0 移入进位标志位 Example: *slc* MEM ; 结果： MEM (b6,b5,b4,b3,b2,b1,b0,C) ← MEM (b7,b6,b5,b4,b3,b2,b1,b0), C ← MEM (b7) 受影响的标志位： Z:『不变』， C:『受影响』， AC:『不变』， OV:『不变』
swap **a**	累加器的高 4 位与低 4 位互换 例如：*swap* a ; 结果： a (b3,b2,b1,b0,b7,b6,b5,b4) ← a (b7,b6,b5,b4,b3,b2,b1,b0) 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』

swap **M**	RAM 的高 4 位与低 4 位互换 例如：*swap* MEM ; 结果： MEM (b3,b2,b1,b0,b7,b6,b5,b4) ← MEM (b7,b6,b5,b4,b3,b2,b1,b0) 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』

7.4. 逻辑运算类指令

and **a, I**	累加器和立即数据执行逻辑 AND，然后把结果保存到累加器 例如：*and* a, 0x0f; 结果： a ← a & 0fh 受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』
and **a, M**	累加器和 RAM 执行逻辑 AND，然后把结果保存到累加器 例如：*and* a, RAM10 ; 结果： a ← a & RAM10 受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』
and **M, a**	累加器和 RAM 执行逻辑 AND，然后把结果保存到 RAM 例如： *and* MEM, a ; 结果： MEM ← a & MEM 受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』
or **a, I**	累加器和立即数据执行逻辑 OR，然后把结果保存到累加器 例如： *or* a, 0x0f ; 结果： a ← a | 0fh 受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』
or **a, M**	累加器和 RAM 执行逻辑 OR，然后把结果保存到累加器 例如：*or* a, MEM ; 结果： a ← a | MEM 受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』
or **M, a**	累加器和 RAM 执行逻辑 OR，然后把结果保存到 RAM 例如： *or* MEM, a ; 结果： MEM ← a | MEM 受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』
xor **a, I**	累加器和立即数据执行逻辑 XOR，然后把结果保存到累加器 例如： *xor* a, 0x0f ; 结果： a ← a ^ 0fh 受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』

xor **a, IO**	累加器和 IO 寄存器执行逻辑 XOR，然后把结果保存到累加器
	例如： ***xor a, pa ;***
	结果： a ← a ^ pa ; // pa 是 A 端口的数据寄存器
	受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』

xor **IO, a**	累加器和 IO 寄存器执行逻辑 XOR，然后把结果保存到 IO 寄存器
	例如： ***xor pa, a ;***
	结果： pa ← a ^ pa ; // pa is the data register of port A
	受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』

xor **a, M**	累加器和 RAM 执行逻辑 XOR，然后把结果保存到累加器 Example: *xor* a, MEM ; 结果： a ← a ^ RAM10 受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』

xor **M, a**	累加器和 RAM 执行逻辑 XOR，然后把结果保存到 RAM 例如： *xor* MEM, a ; 结果： MEM ← a ^ MEM 受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』

not **a**	累加器执行 1 补码运算，结果放在累加器
	例如： *not* a ;
	结果： a ← ～a
	受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』
	应用范例：
	--- 	
	mov a, 0x38 ; // ACC=0X38
	not a ; // ACC=0XC7
	--- 	

not **M**	RAM 执行 1 补码运算，结果放在 RAM
	例如： *not* MEM ;
	结果： MEM ← ～MEM
	受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』
	应用范例：
	--- 	
	mov a, 0x38 ;
	mov mem, a ; // mem = 0x38
	not mem ; // mem = 0xC7
	--- 	

neg **a**	累加器执行 2 补码运算，结果放在累加器
	例如： *neg* a;
	结果： a ←a 的 2 补码
	受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』
	应用范例：
	--- 	
	mov a, 0x38 ; // ACC=0X38
	neg a ; // ACC=0XC8
	--- 	

neg **M**	RAM 执行 2 补码运算，结果放在 RAM
	例如： *neg* MEM;
	结果： MEM ← MEM 的 2 补码
	受影响的标志位： Z:『受影响』， C:『不变』， AC:『不变』， OV:『不变』
	应用范例：
	--- 	
	mov a, 0x38 ;
	mov mem, a ; // mem = 0x38
	neg mem ; // mem = 0xC8
	--- 	

comp **a, I**	累加器和立即数据比较运算，影响的是标志，标志的改变与（a - I）运算相同 例如： *comp* a, 0x55; 结果： 标志的改变与 （a - 0x55）运算相同 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』

	应用范例：
	--- 	
	mov a, 0x38 ;
	comp a, 0x38 ; // Z 标志位被设置为 1
	comp a, 0x42 ; // C 标志位被设置为 1
	comp a, 0x24 ; // C, Z 标志位被清除为 0
	comp a, 0x6a ; // C, AC 标志位被设置为 1
	--- 	

comp **a, M**	累加器和 RAM 比较运算，影响的是标志位，标志位的改变与（a - MEM）运算相同
	例如： *comp* a, MEM;
	结果：标志位的改变与（a - MEM）运算相同
	受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
	应用范例：
	--- 	
	mov a, 0x38 ;
	mov mem, a ;
	comp a, mem ; // Z 标志位被设置为 1
	mov a, 0x42 ;
	mov mem, a ;
	mov a, 0x38 ;
	comp a, mem ; // C 标志位被设置为 1
	--- 	

comp **M, a**	累加器和 RAM 比较运算，影响的是标志位，标志位的改变与（MEM - a）运算相同 例如： *comp* MEM, a; 结果：标志位的改变与（MEM - a）运算相同 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』

7.5. 位运算类指令

set0 **IO.n**	IO 的位 N 设为 0 例如： *set0* pa.5 ; 结果： PA5=0 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
set1 **IO.n**	IO 的位 N 设为 1 例如：*set1* pb.5 ; 结果： PB5=1 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
tog **IO.n**	IO 的位 N 改变为相反状态 例如： *tog* pa.5 ; 结果： PA5=>1 假如 PA5=0 ; PA5=>0 假如 PA5=1 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
set0 **M.n**	RAM 的位 N 设为 0 例如：*set0* MEM.5 ; 结果： MEM 位 5 为 0 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』

set1 **M.n**	RAM 的位 N 设为 1 例如： *set1* MEM.5 ; 结果： MEM 位 5 为 1 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』

swapc **IO.n**	IO 的第 n 位与进位标志位互换
	例如：*swapc* IO.0;
	结果： C ← IO.0 , IO.0 ← C
	当 IO.0 是输出脚位，进位标志 C 将被送到 IO.0 脚
	当 IO.0 是输入脚位，IO.0 脚的状态将被送到进位标志 C
	受影响的标志位： Z:『不变』， C:『受影响』， AC:『不变』， OV:『不变』
	应用范例 1： （串行输出）：
	--- 	
	…
	set1 pac.0 ; // PA.0 设为输出
	…
	set0 flag.1 ; // C=0
	swapc pa.0 ; // 将 C 传送到 PA.0， PA.0=0
	set1 flag.1 ; // C=1
	swapc pa.0 ; //将 C 传送到 PA.0， PA.0=1
	…
	--- 	
	应用范例 2： （串行输入）
	--- 	
	…
	set0 pac.0 ; // PA.0 设为输入
	…
	swapc pa.0 ; // 把 PA.0 读到 C
	src a ; // 将 C 移到累加器的位 7
	swapc pa.0 ; // 把 PA.0 读到 C
	src a ; // 将新的 C 移到累加器的位 7
	…
	--- 	

7.6. 条件运算类指令

ceqsn **a, I**	比较累加器与立即数据，如果是相同的，即跳过下一指令。标志位的改变与 (a ← a - I)相同 例如： *ceqsn* a, 0x55 ; *inc* MEM ; *goto* error ; 结果： 假如 a=0x55, then “goto error”; 否则, “inc MEM”. 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
ceqsn **a, M**	比较累加器与 RAM，如果是相同的，即跳过下一指令。标志位改变与 (a ← a - M)相同 例如：*ceqsn* a, MEM; 结果： 假如 a=MEM，跳过下一个指令 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』

ceqsn **M, a**	比较累加器与 RAM，如果是相同的，即跳过下一指令。标志位改变与 (M ← M - a)相同 例如： *ceqsn* MEM, a; 结果： 假如 a=MEM，跳过下一个指令 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
cneqsn **a, M**	比较累加器与 RAM，如果是不相同的，即跳过下一指令。标志位改变与 (a ← a - M)相同 例如：*cneqsn* a, MEM; 结果： 假如 a≠MEM，跳过下一个指令 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
cneqsn **a, I**	比较累加器与立即数据，如果是不相同的，即跳过下一指令。标志位的改变与 (a ← a - I)相同 例如： *cneqsn* a, 0x55 ; *inc* MEM ; *goto* error ; 结果： 假如 a≠0x55, then “goto error”；否则, “inc MEM”. 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
t0sn **IO.n**	如果 IO 的指定位是 0，跳过下一个指令。 例如：*t0sn* pa.5; 结果： 如果 PA5 是 0，跳过下一个指令。 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
t1sn **IO.n**	如果 IO 的指定位是 1，跳过下一个指令。 Example: *t1sn* pa.5 ; 结果：如果 PA5 是 1，跳过下一个指令。 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
t0sn **M.n**	如果 RAM 的指定位是 0，跳过下一个指令。 例如： *t0sn* MEM.5 ; 结果： 如果 MEM 的位 5 是 0，跳过下一个指令。 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
t1sn **M.n**	如果 RAM 的指定位是 1，跳过下一个指令。 例如： *t1sn* MEM.5 ; 结果： 如果 MEM 的位 5 是 1，跳过下一个指令。 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
izsn **a**	累加器加 1，若累加器新值是 0，跳过下一个指令。 例如： *izsn* a; 结果： a ← a + 1，若 a=0，跳过下一个指令。 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
dzsn **a**	累加器减 1，若累加器新值是 0，跳过下一个指令。 例如： *dzsn* a; 结果： a ← a – 1，若 a=0，跳过下一个指令。 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
izsn **M**	RAM 加 1，若 RAM 新值是 0，跳过下一个指令。 例如： *izsn* MEM; 结果： MEM ← MEM + 1，若 MEM=0，跳过下一个指令。 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』

dzsn **M**	RAM 减 1，若 RAM 新值是 0，跳过下一个指令。 例如： *dzsn* MEM; 结果： MEM ← MEM - 1，若 MEM=0，跳过下一个指令。 受影响的标志位： Z:『受影响』， C:『受影响』， AC:『受影响』， OV:『受影响』
wait0 **IO.n**	直到 IO 的 N 位为 0，才转到下一个指令；否则，在这里等候。 例如： *wait0* pa.5; 结果： 等候 PA5=0 才转到下一个指令 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
wait1 **IO.n**	直到 IO 的 N 位为 1，才转到下一个指令；否则，在这里等候。 例如： *wait1* pa.5; 结果： 等候 PA5=0 才转到下一个指令 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』

7.7. 系统控制类指令

call **label**	函数调用，地址可以是全部空间的任一地址 例如： *call* function1; 结果： [sp] ← pc + 1 pc ← function1 sp ← sp + 2 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
goto **label**	转到指定的地址，地址可以是全部空间的任一地址 例如： *goto* error; 结果： 跳到 error 并继续执行程序 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
delay **a**	延迟（N+1）周期，N 是由累加器所指定，时间周期是根据执行此指令的 FPP 单元的 1 个指令周期。指令执行后，累加器将为零。 例如： *delay* a; 结果：假如 ACC=0fh，在此延迟 16 个周期 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』 注意：由于ACC是指令计数时的暂时缓冲区，请确保执行此指令时不会被中断。否则，延迟时间可能不是所预期的。
delay **I**	延迟（N+1）周期，N 是立即指定的数据，时间周期是根据执行此指令的 FPP 单元的 1 个指令周期。指令执行后，累加器将为零。 例如： *delay* 0x05; 结果： 在此延迟 6 个周期 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』 注意：由于ACC是指令计数时的暂时缓冲区，请确保执行此指令时不会被中断。否则，延迟时间可能不是所预期的。

delay **M**	延迟（N+1）周期，N 是由 RAM 所指定，时间周期是根据执行此指令的 FPP 单元的 1 个指令周期。指令执行后，累加器将为零。 例如： *delay* M; 结果：假如 M=ffh，在此延迟 256 个周期 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』 注意：由于ACC是指令计数时的暂时缓冲区，请确保执行此指令时不会被中断。否则，延迟时间可能不是所预期的。
ret **I**	将立即数据复制到累加器，然后返回 例如： *ret* 0x55; 结果： A ← 55h ret ; 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
ret	从函数调用中返回原程序 例如： *ret;* 结果： sp ← sp - 2 pc ← [sp] 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
reti	从中断服务程序返回到原程序。在这指令执行之后，全部中断将自动启用。 例如： *reti*; 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
nop	没任何动作 例如： *nop*; 结果： 没任何改变 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』

pcadd **a**	目前的程序计数器加累加器成为下一个程序计数器。
	例如： *pcadd a*;
	结果： pc ← pc + a
	受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
	应用范例：
	--- 	
	…
	mov a, 0x02 ;
	pcadd a ; // PC <- PC+2
	goto err1 ;
	goto correct ; // 跳到这里
	goto err2 ;
	goto err3 ;
	…
	correct: // 跳到这里
	…
	--- 	

engint	允许全部中断。 例如： *engint*; 结果：中断要求可送到 FPP0，以便进行中断服务 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』

disgint	禁止全部中断。 例如： *disgint* ; 结果：送到 FPP0 的中断要求全部被挡住，无法进行中断服务 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
stopsys	系统停止。 例如：*stopsys*; 结果：停止系统时钟和关闭系统 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
stopexe	CPU 停止。 所有震荡器模块仍然继续工作并输出：但是系统时钟是被禁用以节省功耗。 例如： stopexe; 结果： 停住系统时钟，但是仍保持震荡器模块工作 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
reset	复位整个单片机，其运行将与硬件复位相同。 例如： *reset*; 结果：复位整个单片机 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』
wdreset	复位看门狗定时器 例如： *wdreset* ; 结果： 复位看门狗定时器 受影响的标志位： Z:『不变』， C:『不变』， AC:『不变』， OV:『不变』

7.8. 指令执行周期综述

如选用双核心 (FPP0 与 FPP1) 处理单元 :

2 个周期 (2T)	*ldtabh, ldtabl, idxm*
1 个周期 (1T)	Others

如选用单核心 (FPP0 或 FPP1) 处理单元 :

2 个周期 (2T)	*goto, call, ldtabh, ldtabl, idxm*
1 个周期+ 跳过 (1T+S)	*ceqsn, cneqsn, t0sn, t1sn, dzsn, izsn*
1 个周期 (1T)	Others

这里 :

1 个周期+ 跳过 : 如果跳过下条指令的条件成立的话, 就执行 2T, 否则就执行 1T 并且继续往下执行.

7.9. 指令影响标志的综述

Instruction	Z	C	AC	OV	Instruction	Z	C	AC	OV	Instruction	Z	C	AC	OV
mov a, I	-	-	-	-	*mov* M, a	-	-	-	-	*mov* a, M	Y	-	-	-
mov a, IO	Y	-	-	-	*mov* IO, a	-	-	-	-	*nmov* M, a	-	-	-	-
nmov a, M	Y	-	-	-	*ldtabh* index	-	-	-	-	*ldtabl* index	-	-	-	-
ldt16 word	-	-	-	-	*stt16* word	-	-	-	-	*xch* M	-	-	-	-
idxm a, index	-	-	-	-	*idxm* index, a	-	-	-	-	*pushaf*	-	-	-	-
popaf	-	-	-	-	*add* a, I	Y	Y	Y	Y	*add* a, M	Y	Y	Y	Y
add M, a	Y	Y	Y	Y	*addc* a, M	Y	Y	Y	Y	*addc* M, a	Y	Y	Y	Y
addc a	Y	Y	Y	Y	*addc* M	Y	Y	Y	Y	*nadd* a, M	Y	Y	Y	Y
nadd M, a	Y	Y	Y	Y	*sub* a, I	Y	Y	Y	Y	*sub* a, M	Y	Y	Y	Y
sub M, a	Y	Y	Y	Y	*subc* a, M	Y	Y	Y	Y	*subc* M, a	Y	Y	Y	Y
subc a	Y	Y	Y	Y	*subc* M	Y	Y	Y	Y	*inc* M	Y	Y	Y	Y
dec M	Y	Y	Y	Y	*clear* M	-	-	-	-	*sr* a	-	Y	-	-
src a	-	Y	-	-	*sr* M	-	Y	-	-	*src* M	-	Y	-	-
sl a	-	Y	-	-	*slc* a	-	Y	-	-	*sl* M	-	Y	-	-
slc M	-	Y	-	-	*swap* a	-	-	-	-	*swap* M	-	-	-	-
and a, I	Y	-	-	-	*and* a, M	Y	-	-	-	*and* M, a	Y	-	-	-
or a, I	Y	-	-	-	*or* a, M	Y	-	-	-	*or* M, a	Y	-	-	-
xor a, I	Y	-	-	-	*xor a, IO*	Y	-	-	-	*xor IO, a*	-	-	-	-
xor a, M	Y	-	-	-	*xor* M, a	Y	-	-	-	*not* a	Y	-	-	-
not M	Y	-	-	-	*neg* a	Y	-	-	-	*neg* M	Y	-	-	-
comp a, I	Y	Y	Y	Y	*comp* a, M	Y	Y	Y	Y	*comp* M, a	Y	Y	Y	Y
set0 IO.n	-	-	-	-	*set1* IO.n	-	-	-	-	*tog* IO.n	-	-	-	-
set0 M.n	-	-	-	-	*set1* M.n	-	-	-	-	*swapc* IO.n	-	Y	-	-
ceqsn a, I	Y	Y	Y	Y	*ceqsn* a, M	Y	Y	Y	Y	*ceqsn* M, a	Y	Y	Y	Y
cneqsn a, M	Y	Y	Y	Y	*cneqsn a, I*	Y	Y	Y	Y	*t0sn* IO.n	-	-	-	-
t1sn IO.n	-	-	-	-	*t0sn* M.n	-	-	-	-	*t1sn* M.n	-	-	-	-
izsn a	Y	Y	Y	Y	*dzsn* a	Y	Y	Y	Y	*izsn* M	Y	Y	Y	Y
dzsn M	Y	Y	Y	Y	*wait0* IO.n	-	-	-	-	*wait1* IO.n	-	-	-	-
call label	-	-	-	-	*goto* label	-	-	-	-	*delay* a	-	-	-	-
delay I	-	-	-	-	*delay* M	-	-	-	-	*ret* I	-	-	-	-
ret	-	-	-	-	*reti*	-	-	-	-	*nop*	-	-	-	-
pcadd a	-	-	-	-	*engint*	-	-	-	-	*disgint*	-	-	-	-
stopsys	-	-	-	-	*stopexe*	-	-	-	-	*reset*	-	-	-	-
wdreset	-	-	-	-										