I²C 硬件模块数据表 I2CHW V 1.1

Copyright © 2008-2010 Cypress Semiconductor Corporation. All Rights Reserved.

资源	PSoC® 模块				API 存储器（字节）		每个传感器所使用的引脚数
	CapSense®	I2C/SPI	定时器	比较器	闪存	RAM	
CY8C20x66、CY8C20x36、CY8C20336AN、CY8C20436AN、CY8C20636AN、CY8C20x46、CY8C20x96、CY7C60413、CY7C645xx、CY7C643/4/5xx、CY7C60424、CY7C6053x、CYONS2010、CYONS2011、CYONSFN2051、CYONSFN2053、CYONSFN2061、CYONSFN2151、CYONSFN2161、CYONSFN2162、CY8CTST200、CY8CTMG2xx、CY8CTMG30xx							
仅限从器件		x			279-440	8	2
增强型从器件（CY8C20x66、CYONS2xxx、CYONSFN2xxx、CYONSTB2xxx）		x			170-200	1	2

功能和概述

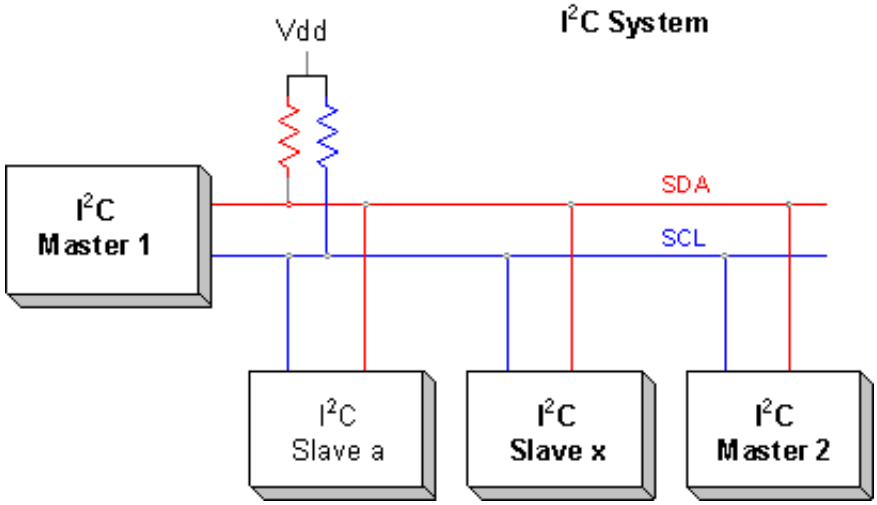
- 行业标准 Philips I²C 总线兼容接口
- 仅限从器件操作
- 仅有两个引脚（SDA 和 SCL）需要与 I²C 总线连接
- 标准数据速率为 100/400 kbits/s，同时支持 50 kbits/s
- 高级 API 只需少量用户编程
- 7-bit 寻址模式，支持 10-bit 寻址

增强型从器件的功能

- 支持 7-bit 硬件寻址比较
- 灵活的数据缓冲方案
- “ 无总线停顿 ” 工作模式
- 低功耗总线监控模式

I²C 硬件用户模块在固件中采用 I²C 从器件。 I²C 总线是 Philips® 开发的一种符合行业标准的两线硬件接口。 主控启动 I²C 总线上的所有通信，并为所有从器件提供时钟。 I2CHW 用户模块支持速率高达 400 kbits/s 的标准模式，并且与同一总线上的其他从器件兼容。 I2CHW 用户模块支持资源表中所列器件提供的增强型从器件功能，以及其他 PSoC 器件支持的从器件功能。

Figure 1. I²C 框图



功能说明

该用户模块支持 I²C 硬件资源。 当 CPU 时钟配置为以 12 MHz 运行时，其传输数据的速度可为 50/100/400 kbits/s。 可以使用较低的 CPU 时钟速率，但是在不使用增强型从器件功能的情况下，这样做可能导致在寻址或数据处理期间总线存在或多或少的停顿现象。 当使用增强型从器件功能时，总线不会出现停顿现象。 I²C 规范允许主控在时钟速率从 100 kHz 直至 DC 的情况下运行。 对于提供直接访问硬件资源的 SDA （串行数据）和 SCL （串行时钟），存在两种不同的选择。 在提供的 API 中支持七位寻址模式。 通过用户扩展为 API 设置支持 10-bit 寻址模式。

有关 I²C 总线和资源部署的详细介绍，请参见器件数据表和访问 Internet。

I²C 从器件

I²C 资源支持逐字节数据传输。 在每次寻址或数据传输 / 接收结束时，报告状态并且可能触发专用的中断。 状态报告和中断生成取决于硬件检测时数据传输的方向和 I²C 总线的状况。 可以将中断配置为在完成字节和检测到总线错误时发生。

每次的 I²C 数据操作由启动、寻址、R/W 方向、数据和终止构成。 该用户模块使用的 I²C 资源只能作为 I²C 从器件运行。 该用户模块提供基于中断的缓冲传输机制。 通信通过前台功能调用启动。 在完成消息的每个字节的传输时，触发中断，I²C 总线停顿，所提供的中断服务子程序 (ISR) 对允许继续通信的总线采取相应的操作，这取决于所执行的初始化。 未确认地址的从器件不会再次中断，直到接收到下一个地址。 从器件必须通过 “ 确认 (ACK)” 或 “ 否认 (NAK)” 对每个地址做出响应。

忽略主控和从器件之间的差异，通常存在两种情况，接收器的差异和发送器的差异。 对于 I²C 接收器，中断在输入数据的 8th 位后发生。 此时，无论输入字节是地址还是数据，接收器件必须对其做出 “ 确认 ” 还是 “ 否认 ” 的判断。 然后，接收器件将相应的控制位写入到 I2C_SCR 寄存器，向 I²C 资源通知 ACK 或 NAK 状态。 通过解除总线停顿，在总线上提供 ACK 或 NAK 状态并将下一个数据字节移入，写入 I2C_SCR 寄存器的操作对总线上的数据流进行步进操作。 对于另一种情况的发送器，中断在外部接收设备提供 ACK 或 NAK 后发生。 可以读取 I2C_SCR 以确定此位的状态。 对于发送器，数据加载到 I2C_DR 寄存器中，并且再次写入到 I2C_SCR 寄存器，以触发下一部分的数据传输。

使用缓冲读取和写入子程序 （bWriteBytes()、bWriteCBytes()、fReadBytes()） 时，没有必要使用任何缓冲区初始化函数 （InitWrite()、InitRamRead()、InitFlashRead()）。 这些函数作为函数调用的一部分被调用，以启动缓冲读取或写入 （bWriteBytes()、bWriteCBytes()、fReadBytes()）。

[+] Feedback

I²C 增强型从器件

增强型从器件为具有硬件寻址比较 （无总线停顿）功能的硬件添加 API 支持。 还包含 32-byte RAM 缓冲区，可以通过 I²C 模块直接对其执行读取或写入。 该缓冲区是 I²C 模块的组成部分，独立于传统的 SRAM，但是可以被 I2CHW 模块或 CPU 访问。 低功耗总线监控器功能允许增强型 I²C 从器件模块在 PSoC 睡眠时监控总线。 通过所有上述功能，可以在无总线停顿的情况下实现灵活的 I²C 通信，在使用 I²C 总线时，提高低功耗的性能。

I²C 增强型从器件功能为从器件提供 32-byte RAM 缓冲区接口。 外部主控控制针对 RAM 缓冲区的读写地址。 CPU 读写独立于主控的读写，从而使 RAM 缓冲区类似于一个双端口 RAM 缓冲区。 因为随时可以获取数据，所以绝不会发生总线停顿。

有四个寄存器用作缓冲区的地址指针。 其中两个由外部主控进行控制。 一个是基本指针 (I2C_BP))，另一个是当前指针 (I2C_CP).)。当主控写入一个或多个字节时，两个指针获取设置。 当主控写入时，发送的第一个字节始终为基本指针的地址和当前指针的初始设置地址。 第二个字节 （数据的第一个字节）位于当前指针的地址中，当前指针自动递增，以设置第三个字节 （数据的第二个字节）的地址。 每个字节写入到缓冲区后，当前指针递增。 主控执行读操作时，当前指针设置值与基本指针相同，在连续从缓冲区读取各个字节后，当前指针递增。 基本指针仅在发生主控写入时设置。 如果主控尝试读取规定范围之外的数据时，会返回无效数据。 如果主控尝试在规定范围之外写入时，数据将被丢弃，并且不会影响任何 RAM 区域。

其他两个寄存器由固件控制，还可用作相同 32-byte RAM 缓冲区的地址指针。 其中一个是基本指针 (CPU_BP))，另一个是当前指针 (CPU_CP).)。这两种指针的功能类似于先前提到的地址指针 （I2C_BP 和 I2C_CP).)。当执行 CPU 读写时，还用到另一个寄存器 (I2C_BUF))。 当写入该寄存器时，由当前指针 (CPU_CP).) 指定位置处的数据传输至缓冲区。当读取该寄存器时，返回当前指针指定位置处的数据。 如果 RAM 内容未初始化，则无有效值返回。

设计注意事项

从器件维护可供主控访问的缓冲区空间的内部计数。 变量命名为 I2CHW_Read_Count 和 I2CHW_Write_Count （其中 I2CHW 由 PSoC Designer 中用户模块的实例名称替换）。 变量为全局变量，可以通过包含相应的 “extern” 声明来使用 C 访问。 通过从计数变量的初始值减去当前值，可以确定主控读取或写入的字节数。 对于从器件 （仅限）用户模块，可以使用函数 I2CHW_InitWrite、I2CHW_InitRamRead、I2CHW_InitFlashRead 设置计数变量的初始大小。 为 MultiMasterSlave 用户模块中使用的可选从器件设置计数值的函数调用是 I2CHW_InitSlaveWrite、I2CHW_InitSlaveRamRead 和 I2CHW_InitSlaveFlashRead.。

缓冲区用于在 I²C 从器件 API 中读取和写入数据。 必须在启用 I²C 从器件之前，初始化相应的缓冲区。 通过 I²C 主控启动读写后，在 I2CHW_Status 字节中设置相应的状态位。 从器件的前台处理进程对写缓冲区中存放的数据或者从读缓冲区中提取的数据进行处理。 进入中断服务子程序 (ISR) 后，从器件数据传输中断服务子程序 (ISR) 不允许访问超出其定义长度的缓冲区。 对缓冲区的读写可以采用以下方式处理：

- 如果 I²C 主控尝试读取的数据超出缓冲区包含的数据，则转发最后一个字节，直至 I²C 主控停止读取操作。 （I²C 协议未针对 I²C 从器件定义停止主控读取操作的方法。）
- 当 I²C 主控接收数据或向 I²C 从器件写入数据时，如果确定没有可用的存储空间，则从器件接收到最后一个字节时，会生成 NAK。 如果 I²C 主控继续写入数据，则从器件连续发出 NAK。 生成第一个 NAK 后 （数据存储在最后可用的位置），不会进一步存储数据。
- 如果缓冲区定义为零长度，则否认写入到 I²C 从器件的数据，且不会存储。 启用直接从闪存读取数据功能还允许使用 RAM 或闪存缓冲区。 如果数据传输 ISR 使用闪存 /ROM 或 RAM 中的读缓冲区，则可以使用提供的 API 对其进行配置。

[+] Feedback

动态重新配置

赛普拉斯强烈建议不要将 I2CHW 资源与动态加载 / 卸载层整合。 只将 I2CHW 资源作为基本配置的组成部分。 根据运行要求说明修改 I2CHW 模块的运行，但是尝试移除作为 动态重新配置组成部分的资源，可以会对外部 I^2C 器件产生不利的影响。

I^2C 寻址

I^2C 地址包含于读写数据操作的第一个字节的头 7 位中。 该字节由 I^2C 主控使用，以对从器件进行寻址。 有效的选择项为 0-127 （十进制）。 字节的 LSb 包含 R/~W 位。 如果该位为零，则写入地址，如果 LSb 为 1，则寻址的从器件从其中读取数据。

在内部，用户模块获取输入地址，对其进行移位，使其与读 / 写位整合，从而构建一个完整的地址字节。

示例

0x48 的地址作为参数传递，或者定义为从器件的地址。 传递包含读 / 写信息的单个参数。 I^2C 主控发送 0x90 字节 (8-bits) 以向从器件写入数据，以及发送 0x91 字节以从从器件读取数据。

因为从器件模块的地址参数接受十进制数值输入，所以采用十进制键入 7-bit 地址 （十进制数 72）。

直流和交流电气特性

请参考 PSoC 设备的数据表，了解 I^2C 接口的电气特性。

根据框图说明，I^2C 总线要求外部上拉电阻。 上拉电阻 (R_P) 由供电电压、时钟速率和总线电容决定。 对于输出阶段，在 V_{OLmax} = 0.4V 的情况下，任何器件 （主控或从器件）的最小灌电流不得小于 3 mA。 对于 5V 的系统，这样可以将最小上拉电阻值限制到大约为 1.5 kΩ。 R_P 的最大值取决于总线电容和时钟速率。 对于总线电容为 150 pF 的 5V 系统，上拉电阻不得大于 6 kΩ。 有关 I^2C 总线规范的更多信息，请参见 www.nxp.com 的 NXP 网页。

Note 从赛普拉斯或其获得分许可的其中一个联营公司处购买 I^2C 组件，即可根据 Philips I^2C 专利权获得一份许可，以便在 I^2C 系统中使用这些组件，但前提是该系统符合 Philips 定义的 I^2C 标准规范。

放置

I2CHW 用户模块为 SCL 和 SDA 提供两种选择。 一种选择是使用 P1[7] 作为 SCL 并使用 P1[5] 作为 SDA。另一种选择是使用 P1[1] 作为 SCL 并使用 P1[0] 作为 SDA。尽可能使用 P1[5]/P1[7] 引脚，因为 P1[0]/P1[1] 可以与 ISSP 和 ECO 共用。 不存在放置限制。 I^2C 模块不能有多种放置，因为 I^2C 模块使用专用的 PSoC 资源模块和中断。

参数和资源

所有缓冲区都由 I^2C 主控根据其使用进行命名。 例如，名称或描述为 “ 读取 ” 的缓冲区由 I^2C 主控执行读取。

Slave_Addr

Slave_Addr 选择由 I^2C 主控使用的 7-bit 从器件地址对从器件进行寻址。 有效的选择项为 0-127 （十进制）。 增强型从器件不使用该参数。

[+] Feedback

I2C_Clock

指定 I^2C 接口运行所用的时钟速率。 存在三种可能的时钟速率：

- 50 K 标准
- 100 K 标准
- 400K 快速

I2C_Pin

从端口 1 选择用于 I^2C 信号的引脚。 对于这些引脚无需选择相应的驱动模式，因为 PSoC Designer 会自动选择。

Read_Buffer_Types

选择数据读取所支持的缓冲区的类型。 有 2 种选择项可用： “ 仅 RAM” 或 “RAM 或闪存 ”。 “ 仅 RAM” 选项会删除支持直接闪存 ROM 读取所要求的代码和变量。 “RAM 或闪存 ” 选项可以提供代码和变量支持，以便读取 RAM 缓冲区或闪存向主控传输数据。 如果选择 “RAM 或闪存 ”，则使用 I2CHW_InitRamRead() 或 I2CHW_InitFlashRead() API 以选择缓冲区类型。 增强型从器件不使用该参数。

Communication_Service_Type

此项参数让用户能够在基于中断的数据处理策略和轮询策略之间做出选择。 增强型从器件不使用该参数。 在基于中断的策略中，数据传输针对预先定义的缓冲区而启动。 数据随后在后台以最快程度移入或移出缓冲区。 其中还包含一个用于处理数据移动的中断服务子程序 (ISR)。 在选择轮询数据处理策略时，用户对何时执行数据移动操作拥有控制权。 为了实现轮询策略，用户可以定期调用函数 I2CHW_Poll() （请参见 I2CHWslave.h 文件获取准确的实例名称）。 每次调用轮询函数时，将传输单个字节。 其他 I^2C 函数的使用完全相同。 轮询通信策略可以在中断延迟具有关键重要性 （而且异步通信中断可能导致问题）的状况下应用。 另一种使用方式是在用户要求对何时进行数据传输拥有绝对控制权的情况下。 轮询的一个缺点是，当 I^2C 状态机启用时，总线将会在每个字节后自动停顿，直到轮询函数被调用时为止。

应用程序编程接口

应用编程接口 (API) 固件提供了支持多字节传输中的发送和接收操作的高级命令。 可以在 RAM 或闪存中创建读取缓冲区。 写缓冲区只能在 RAM 内存中设置。

Note 在这里，与所有用户模块 API 中的情况相同，调用 API 函数会改变 A 和 X 寄存器的值。 如果在调用后需要 A 和 X 的值，则调用函数负责在调用前保留 A 和 X 的值。 使用这种 “ 寄存器易变 ” 策略是出于效率原因，该策略是从 PSoC Designer 1.0 版本开始实施的。 C 语言编译器将自动处理此项要求。 汇编语言编程人员也必须确保其代码遵守这一策略。 虽然一些用户模块 API 函数可以保留 A 和 X 不变，但是无法保证它们将来也会如此。

常用函数

以下函数为用户模块的从器件和增强型从器件的常用函数。

I2CHW_Start

说明：

无任何操作。 仅用于保持接口一致性。 使用 I2CHW_EnableSlave() 启动从器件和 I2CHW_EnableInt() 以启用中断。

C 原型：

```
void  I2CHW_Start(void);
```


[+] Feedback

汇编程序：

```
lcall  I2CHW_Start
```

参数：

无

返回值：

无

副作用：

A 和 X 寄存器可以由此函数的本次执行或以后执行而被修改。 这适用于大型存储器模型中的所有 RAM 页指针寄存器。 在必要时，调用函数有责任将调用前后的数值保存在 fastcall16 函数内。

I2CHW_Stop

说明：

禁用 I2CHW （通过禁用 I^2C 中断）。

C 原型：

```
void   I2CHW_Stop(void);
```

汇编程序：

```
lcall  I2CHW_Stop
```

参数：

无

返回值：

无

副作用：

请参见 API 开始部分的介绍。

I2CHW_EnableInt

说明：

启用 I^2C 中断以便实现启动条件检测。 请牢记要通过使用以下这个宏来调用这个全局中断启用函数: M8C_EnableGInt.

C 原型：

```
void   I2CHW_EnableInt(void);
```

汇编程序：

```
lcall  I2CHW_EnableInt
```

参数：

无

返回值：

无

[+] Feedback

副作用：

请参见 API 开始部分的介绍。

I2CHW_DisableInt

说明：

通过禁用 SDA 中断来禁用 I^2C 从器件。 执行与 I2CHW_Stop. 相同的操作

C 原型：

```
void   I2CHW_DisableInt(void);
```

汇编程序：

```
lcall I2CHW_DisableInt
```

参数：

无

返回值：

无

副作用：

请参见 API 开始部分的介绍。

I2CHW_EnableSlave

说明：

通过设置 I2C_CFG 寄存器中的 “ 启用从器件 ” 位，为 I2C 硬件模块启用 I2C “ 从器件 ” 函数。

C 原型：

```
void   I2CHW_EnableSlave(void);
```

汇编程序：

```
lcall  I2CHW_EnableSlave
```

参数：

无

返回值：

无

副作用：

请参见 API 开始部分的介绍。

I2CHW_DisableSlave

说明：

通过清除 I2C_CFG 寄存器中的 “ 启用从器件 ” 位，禁用 I2C “ 从器件 ” 函数。

C 原型：

```
void   I2CHW_DisableSlave(void);
```


[+] Feedback

汇编程序：

```
lcall  I2CHW_DisableSlave
```

参数：

无

返回值：

无

副作用：

请参见 API 开始部分的介绍。

从器件函数

这些函数不可用于增强型从器件。

I2CHW_Poll

说明：

在 Communication_ Service_Type 参数设置为 “ 轮询 ” 时使用。 此函数提供了一个由用户控制的进入 I/O 处理子程序的入口。 如果将 Communication_Service_Type 参数设置为 “ 中断 ”，则此函数不做任何操作。

注： 调用 I2CHW_Poll 释放 I2C 总线。 这让主控可以在从器件固件完成处理先前的请求之前读取或写入数据。

C 原型：

```
void   I2CHW_Poll(void);
```

汇编程序：

```
lcall I2CHW_Poll
```

参数：

无

返回值：

无

副作用：

每次调用该子程序时，将处理一个 I^2C 事件，而状态变量将得到更新。 事件的构成可以是一个故障状况、一个 I/O 字节，或在某些特定情况下，一个停止状况。 调用该子程序可能有三种结果：

- 如果没有事件，则无任何操作。
- 如果有一个可用地址或数据字节，则对此地址或数据进行接收或发送。
- 如果外部主控已完成其写入操作，则处理停止事件。

在写入操作结束阶段处理停止状态时仅更新状态变量。 如果停止状态得到处理时 I^2C 字节处于待处理状态，再次调用 I2CHW_Poll 函数可以对其进行处理。 I2CHW_Poll() 函数在 Communication_Service_Type 设置为 “ 中断 ” 时不会产生任何作用。 在总线上检测到启动 / 重启条件和地址时，总线将处于停顿状态，直至 I2CHW_Poll() 函数被调用。 如果地址被否认，则为该数据操作发送的后续字节将会被忽略，直到检测到另一个启动 / 重启和地址，否则，I^2C 总线上的每个数据字节都将处于停顿状态，直至 I2CHW_Poll() 函数被调用。

[+] Feedback

I²C 总线由 I²C 硬件引起停顿，直到在 Communication_Service_Type 设置为 “ 轮询 ” 的情况下调用此函数。 对于已接收的数据，总线将在字节末尾并在生成 ACK/NAK （通过将 SCL （时钟）线保持在低位）之前停顿。 对于已发送的数据，总线将在 ACK/NAK 位在外部生成之后立即停顿。

I2CHW_bReadI2CStatus

说明：

返回 “ 控制 / 状态 ” 寄存器中的状态位。

C 原型：

```
BYTE  I2CHW_bReadI2CStatus(void);
```

汇编程序：

```
lcall  I2CHW_bReadI2CStatus ; Accumulator contains status
```

参数：

无

返回值：

```
BYTE  I2CHW_RsrcStatus
```

请参见表。

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

常量	值	说明
I2CHW_RD_NOERR	01h	主控读取数据，正常 ISR 退出
I2CHW_RD_ OVERFLOW	02h	主控所读取的数据字节超出了可用字节数量
I2CHW_RD_ COMPLETE	04h	一项读取操作已启动且已完成。
I2CHW_READFLASH	08h	下一个读取操作将来自某个闪存位置
I2CHW_WR_NOERR	10h	主控成功写入了数据
I2CHW_WR_OVERFLOW	20h	主控在写入缓冲区写入了过多字节
I2CHW_WR_COMPLETE	40h	主控写入操作根据新地址完成或停止
I2CHW_ISR_ACTIVE	80h	I²C ISR 尚未退出并处于活动状态

[+] Feedback

I2CHW_ClrRdStatus

说明：

清除 I2CHW_RsrcStatus 寄存器中的读取状态位。 不会影响其他位。

C 原型：

```
void  I2CHW_ClrRdStatus (void);
```

汇编程序：

```
lcall  I2CHW_ClrRdStatus
```

参数：

无

返回值：

无

副作用：

请参见 API 开始部分的介绍。

I2CHW_ClrWrStatus

说明：

清除 I2CHW_RsrcStatus 寄存器中的写入状态位。 不会影响其他位。

C 原型：

```
void  I2CHW_ClrWrStatus (void);
```

汇编程序：

```
lcall  I2CHW_ClrWrStatus
```

参数：

无

返回值：

无

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW_InitWrite

说明：

初始化一个数据缓冲区指针，让从器件用来存放数据，并初始化相同缓冲区计数字节的值。 将计数初始化为所提供的最大缓冲区长度。 在下一个主控写入实例中，数据将放置到该函数定义的地址中。

[+] Feedback

C 原型：

```
void   I2CHW_InitWrite(BYTE * pWriteBuf, BYTE bBufLen);
```

汇编程序：

```
AREA    bss (RAM, REL)
abWriteBuf    blk 10h

AREA    text (ROM,REL)
    push X                          ; save registers
    push A
    add  SP, 3
    mov  X, SP
    dec  X                          ; X points at data SP points at next
                                    ; empty stack location
    mov  [X], abWriteBuf            ; place the buffer address
                                    ; (page 0)on the stack at [X]
    mov  [X-2], 10                  ; place the count at [x-2]
                                    ; don't care what [X-1] is
                                    ; the compiler would assign 0 as the
                                    ; MSB of the Ramtbl addr
    lcall  I2CHW_InitWrite
    add   SP, -3                    ; restore the stack
    pop   A                         ; restore registers
    pop   X
```

参数：

pWriteBuf： 指向 RAM 缓冲区位置的指针。

写入缓冲区的长度。

返回值：

无

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW_InitRamRead

说明：

初始化主控用于检索数据的 RAM 数据缓冲区指针，并初始化相同缓冲区计数字节的值。 将 I2CHW_SlaveStatus 标记 I2CHW_READFLASH 清除为零 （仅限读取状态位），使下一次尝试从 RAM 中先前设置的缓冲区位置读取。

C 原型：

```
void   I2CHW_InitRamRead(BYTE * pReadBuf, BYTE bBufLen);
```

汇编程序：

```
AREA bss (RAM,REL)
abReadBuf:    blk 10h

AREA text (ROM,REL)
    push  X                         ; save registers
```


[+] Feedback

```
    push  A
    add   SP, 3
    mov   X, SP
    dec   X                         ; X points at data SP points at next
                                    ; empty stack location
    mov   [X], abReadBuf            ; place the read buffer address
                                    ; (page0)on the stack at [X]
    mov   [X-2], 10                 ; place the count at [x-2]
                                    ; don't care what [X-1] is
                                    ; the compiler would assign 0 as
                                    ; the MSB of the Ramtbl addr
    lcall  I2CHW_InitRamRead
    add   SP, -3                    ; back up the stack (subtract 3)
    pop   A                         ; restore registers
    pop   X
```

参数：

_ReadBuf： 指向 RAM 缓冲区位置的指针。bBufLen： 读取缓冲区的长度。

返回值：

无

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW_InitFlashRead

说明：

初始化用于检索数据的闪存数据缓冲区指针。将 I2CHW_SlaveStatus 标记 I2CHW_READFLASH 设置为 1，使下一次尝试从闪存中先前设置的缓冲区位置读取。

C 原型：

```
void   I2CHW_InitFlashRead(const BYTE * pFlashBuf, WORD wBufLen);
```

汇编程序：

```
area table(ROM,ABS)
org 0x1015

abFlashBuf:

      db 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88
      db 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff


area text(ROM,REL)


    push  X                         ; save registers
    push  A
    add   SP, 4
    mov   X, SP
    dec   X                         ; X points at data SP points at next
                                    ; empty stack location
```


[+] Feedback

```
    mov   [X], <abFlashBuf          ; place the LSB of rom
                                    ; address on the stack at [X]
    mov   [X-1], >abFlashBuf        ; place the MSB of the rom address
                                    ; at [x-1] variable
    mov   [X-2], 0x0F               ; place the LSB of length
                                    ; at [x-2]
    mov   [X-3], 0x00               ; place the MSB of length
                                    ; at [x-3]
    lcall  I2CHW_InitFlashRead
    add   SP, -4                    ; adjust the stack (subtract 4)
    pop   A                         ; restore registers
    pop   X
```

参数：

pFlashBuf： 指向闪存 /ROM 缓冲区位置的指针。wBufLen： 缓冲区的长度。

返回值：

无

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

增强型从器件函数

下列函数仅用于增强型从器件版本的用户模块。

I2CHW_StartBufferedSlave

说明：

启用 I2C 缓冲区模式。 增强型从器件 API 支持使用 32-byte 硬件缓冲区通信，从而实现向后兼容。

C 原型：

```
void  I2CHW_StartBufferedSlave(void);
```

汇编程序：

```
lcall  I2CHW_StartBufferedSlave
```

参数：

无

返回值：

无

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW_StopBufferedSlave

说明：

禁用 I2C 缓冲区模式。 这样返回增强型从器件，以与 I^2CHW 从器件 API 兼容。

C 原型：

```
void  I2CHW_StopBufferedSlave (void);
```


[+] Feedback

汇编程序：

```
lcall  I2CHW_StopBufferedSlave
```

参数：

无

返回值：

无

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW_EnableNoBCInt

说明：

关闭字节完成中断。I2CHW_StartBufferedSlave() 和 I2CHW_SetHWAddr() 必须在调用此函数之前调用。

C 原型：

```
void  I2CHW_EnableNoBCInt(void);
```

汇编程序：

```
lcall  I2CHW_EnableNoBCInt
```

参数：

无

返回值：

无

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW_DisableNoBCInt

说明：

启动字节完成中断。I2CHW_StartBufferedSlave() 和 I2CHW_SetHWAddr() 必须在调用此函数之前调用。

C 原型：

```
void  I2CHW_DisableNoBCInt (void);
```

汇编程序：

```
lcall  I2CHW_DisableNoBCInt
```

参数：

无

返回值：

无

[+] Feedback

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW_EnableStopIE

说明：

启动停止中断。I2CHW_StartBufferedSlave() 和 I2CHW_SetHWAddr() 必须在调用此函数之前调用。

C 原型：

```
void  I2CHW_EnableStopIE(void);
```

汇编程序：

```
lcall  I2CHW_EnableStopIE
```

参数：

无

返回值：

无

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW_DisableStopIE

说明：

关闭停止中断。I2CHW_StartBufferedSlave() 和 I2CHW_SetHWAddr() 必须在调用此函数之前调用。

C 原型：

```
void  I2CHW_DisableStopIE (void);
```

汇编程序：

```
lcall  I2CHW_DisableStopIE
```

参数：

无

返回值：

无

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW_ GetMasterWriteData

说明：

通过检索 I^2C 基本和当前指针地址（I2C_BP 和 I2C_CP 寄存器）获取最新写入的数据。 确定缓冲区计数，通过 CPU 基本和当前指针实现。 检索数据，将数据放入指定的缓冲区中。

C 原型：

```
void   I2CHW_GetMasterWriteData(BYTE * pbReadMasterBuf);
```


[+] Feedback

汇编程序：

```
   mov  A,> pbReadMasterBuf ; Load MSB part of pointer to RAM based null
                            ; terminated string.
   mov  X,< pbReadMasterBuf ; Load LSB part of pointer to RAM based null
                            ; terminated string.
   lcall I2CHW_GetMasterWriteData ; lcall function to send string out TX8 port
```

参数：

_pbReadMasterBuf: Pointer to a RAM buffer location.

返回值：

无

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW _SetHWAddr

说明：

设置硬件从器件地址。

C 原型：

```
void   I2CHW SetHWAddr (BYTE bHWAddr);
```

汇编程序：

```
mov  A, 0x07          ; Load address of 0x07
lcall I2CHW_SetHWAddr ; The address will be set to 0x07
```

参数：

_ bHWAddr： 要设置的硬件从器件地址。 (0 - 7Fh)

返回值：

无

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW _bMonitorBufAccess

说明：

返回 I2CHW_XSTAT 寄存器 SLAVE_BUSY 位的状态。 这样可以确定从器件何时忙碌，以及何时在访问缓冲区模块。

C 原型：

```
BYTE   I2CHW bMonitorBufAccess (void);
```

汇编程序：

```
lcall I2CHW bMonitorBufAccess ;lcall function to get status of buffer module
                              ;Status is returned in the Accumulator
```

参数：

无

[+] Feedback

返回值：

_ status： SLAVE_BUSY 位的状态

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW _WriteI2CBuf

说明：

该函数用于将 **pbWriteBuf** 指定的 SRAM 缓冲区的 bWriteCount 字节复制到起始于 **bStartAddr** 的 32-byte I2C 缓冲区中。

C 原型：

```
void   I2CHW_WriteI2CBuf(BYTE bStartAddr, BYTE * pbWriteBuf, BYTE bWriteCount);
```

汇编程序：

```
mov A, 32          ;Load buffer count
push A
mov A, >pbWriteBuf ;Load MSB part of pointer to RAM array
push A
mov A, <pbWriteBuf ;Load LSB part of pointer to RAM array
push A
mov A, 0           ;Load write start address (0-1Fh)
push A
lcall I2CHW_WriteI2CBuf
```

参数：

_ bStartAddr： CPU_BP （基本指针）的起始地址

_ pbWriteBuf： 指向要写入数据的 RAM 缓冲区位置的指针。

_ bWriteCount： 要写入字节的计数

返回值：

无

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW _ReadI2CBuf

说明：

该函数将起始于 **bStartAddr** 的 32-byte I2C 缓冲区的 bReadCount 字节复制到 **pbReadBuf** 指定的 SRAM 缓冲区中。

C 原型：

```
void   I2CHW ReadI2CBuf (BYTE bStartAddr, BYTE * pbReadBuf, BYTE bReadCount);
```

汇编程序：

```
mov A, 32         ;Load # bytes to read
push A
mov A, >pbReadBuf ;Load MSB part of pointer to RAM array
push A
mov A, <pbReadBuf ;Load LSB part of pointer to RAM array
```


[+] Feedback

```
push A
mov A, 0          ;Load read start address (0-1Fh)
push A
lcall I2CHW_ReadI2CBuf
```

参数：

_ bStartAddr： CPU_BP （基本指针）的起始地址

_ pbReadBuf： 指向存储要读取数据的 RAM 缓冲区位置的指针。

_ bReadCount： 要读取字节的计数

返回值：

无

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW _ wGetI2CPointers

说明：

读取 I^2C 基本指针 (I2C_BP)) 和当前指针 (I2C_CP)) 寄存器的值。 I2C_BP 包含在返回字的 MSB 中。 I2C_CP 包含在返回字的 LSB 中。

C 原型：

```
WORD   I2CHW wGetI2CPointers (void);
```

汇编程序：

```
lcall I2CHW wGetI2CPointers ;Returns LSB in A and MSB in X
; X contains I2C_BP and A contains I2C_CP
```

参数：

无

返回值：

WORD wI2Cptrs （X 包含 I2C_BP，A 包含 I2C_CP))

副作用：

请参见 API 开始部分的介绍。 目前，只有 CUR_PP 页面指针寄存器被修改。

I2CHW _Sleep

说明：

管理 I2C 运行的浅睡眠。 该 API 函数仅适用于 CY8CTMA3xx、CY8CTMG3xx 和 CY8CTST3xx 器件。

C 原型：

```
void   I2CHW_Sleep (void);
```

汇编程序：

```
call I2CHW_Sleep
```

参数：

无

[+] Feedback

返回值：

无

副作用：

请参见 API 开始部分的介绍。

固件源代码示例

以下是 I2CHW 从器件用户模块的应用示例：

```
/*************************************************************/
/* This sample code echoes bytes received from a master */
/* The master sends and requests up to 64 bytes.          */
/*                                                          */
/* The instance name of the I2CHWs User Module is defined */
/* as I2CHW.                                               */
/*                                                          */
/*  NOTE I2CHW does not depend upon the CPU clock          */
/*************************************************************/
#include <m8c.h>
#include <i2CHWCommon.h>

/* setup a 64 byte buffer */
BYTE    abBuffer[64];
BYTE    status;

void EchoData()
{
/* Start the slave and wait for the master */
    I2CHW_Start();
    I2CHW_EnableSlave();
/* Enable the global and local interrupts */
    M8C_EnableGInt;
    I2CHW_EnableInt();

    /* Setup the Read and Write Buffer - set to the same buffer */
    I2CHW_InitRamRead(abBuffer,64);
    I2CHW_InitWrite(abBuffer,64);


    /* Echo forever */
    while( 1 )
    {
        status = I2CHW_bReadI2CStatus();
        /* Wait to read data from the master */
        if( status & I2CHW_WR_COMPLETE )
        {
        /* Data received - clear the Write status */
        I2CHW_ClrWrStatus();
        /* Reset the pointer for the next read data */
        I2CHW_InitWrite(abBuffer,64);
        }
        /* wait until data is echoed */
        /* want to know if RD_NOERR is SET AND RD_COMPLETE is SET */
        if( status & I2CHW_RD_COMPLETE )
```


[+] Feedback

```
        {
        /* Data echoed - clear the read status */
        I2CHW_ClrRdStatus();
        /* Reset the pointer for the next data to echo */
        I2CHW_InitRamRead(abBuffer,64);
        }
    }
}
void main(void)
{
    EchoData();
}
```

以下是将 I2CHW 用户模块配置为使用汇编代码编写的从器件的应用示例：

```
;-----------------------------------------------------------
; this assembly assumes small memory model
;-----------------------------------------------------------
include "m8c.inc"
include "I2CHW_1Common.inc"
export rec_cnt
export i2c_addr
;export master_state

BUFFERSIZE: equ 64

export abBuffer
export rec_cnt
export status

area bss(RAM)

rec_cnt: blk 1
i2c_addr: blk 1
abBuffer: blk BUFFERSIZE
status:   blk 1
;master_state: blk 1


area text(ROM, REL)

export _main

_main:
    ;enable the I2C slave as an ISR based process

    lcall I2CHW_1_EnableSlave
call I2CHW_1_EnableInt
M8C_EnableGInt

Init:
mov A, BUFFERSIZE
push A
push A
```

```
mov A, <abBuffer
push A
mov X, sp
dec X
call I2CHW_1_InitWrite
;
;
;everything should still be initialized on the stack to call initRamRead
call I2CHW_1_InitRamRead
add SP, -3

checkI2CStatus:
    call I2CHW_1_bReadI2CStatus
    and A, I2CHW_WR_NOERR
    jnz writeHappened
    and A, I2CHW_RD_NOERR
    jnz readHappened

    jmp checkI2CStatus



readHappened:
    ;call I2CHW_3_ClrRdStatus
    ;calculate bytes read
    mov A, BUFFERSIZE
    sub A, [I2CHW_1_Read_Count]
    inc A
    cmp A, BUFFERSIZE
    jnz checkI2CStatus
    jmp _main

writeHappened:
    ;call I2CHW_3_ClrWrStatus
    ;calculate bytes read
    mov A, BUFFERSIZE
    sub A, [I2CHW_1_Write_Count]
    inc A
    cmp A, BUFFERSIZE
    jnz checkI2CStatus
    jmp Init


;end _main
```

以下是将 I2CHW 用户模块配置为使用汇编代码编写的主控的应用示例：

```
;-----------------------------------------------------------------------------
; Assembly main line
;-----------------------------------------------------------------------------

include "m8c.inc"       ; part specific constants and macros
include "memory.inc"    ; Constants & macros for SMM/LMM and Compiler
include "PSoCAPI.inc"   ; PSoC API definitions for all User Modules
```

```
export _main

area bss(RAM)
sTXData::   blk 10
sRXData::   blk 10


area text(ROM,REL)

_main:


     mov [sTXData+0], 00h
     mov [sTXData+1], 01h
     mov [sTXData+2], 02h
     mov [sTXData+3], 03h
     mov [sTXData+4], 04h
     mov [sTXData+5], 05h
     mov [sTXData+6], 06h
     mov [sTXData+7], 07h
     mov [sTXData+8], 08h
     mov [sTXData+9], 09h



     M8C_EnableGInt;Enable Global Interrupts
     call I2CHW_1_EnableInt;Enable I2C Interrupts
     call I2CHW_1_Start;Start the I2C Block
     call I2CHW_1_EnableMstr;Enable Master Mode for I2C

   .loop:

          ; Load data to send to slave at address 0x68
          mov A,I2CHW_1_CompleteXfer
          push A
          mov A, 0x05;Send 9 Bytes
          push A
          mov A, >sTXData
          push A
          mov A, <sTXData
          push A
          mov A, 0x68;Slave Address
          push A
          call _I2CHW_1_bWriteBytes
          add sp, -5

          ;Wait for the TX to complete
    .TxCompleteLoop:
          call I2CHW_1_bReadI2CStatus;Check the I2C Status
          and A, I2CHW_WR_COMPLETE;Check to see if the write is complete
          jz .TxCompleteLoop;If not try again
          call I2CHW_1_ClrWrStatus;If complete clear status

          ;Get read to read data from slave at address 0x68
```

```
          mov A, I2CHW_1_CompleteXfer
          push A
          mov A, 0x05;Read 9 Bytes
          push A
          mov A, >sRXData
          push A
          mov A, <sRXData
          push A
          mov A, 0x68;Slave Address
          push A
          call _I2CHW_1_fReadBytes
          add sp, -5

          ;Wait for the RX to Complete
    .RxCompleteLoop:
          call I2CHW_1_bReadI2CStatus;Check I2C Status
          and A, I2CHW_RD_COMPLETE;Check to see if read is complete
          jz .RxCompleteLoop;If not try again
          call I2CHW_1_ClrRdStatus;If complete clear status

          jmp .loop ;Send and Receive Again


.terminate:
    jmp .terminate
```

以下是使用 C 语言编写的 I2CHW 用户模块的增强型从器件应用示例：

```
/*-----------------------------------------------------------
* This sample code does a CPU write and read to and from
* the buffer.  Then waits for the buffer to get accessed
* again  and checks to see what was last written by the
* master.
*
* The instance name of the I2CHWs User Module is defined
* as I2CHW.
-----------------------------------------------------------*/

#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules

void main(void)
{
    BYTE abWriteBuf[] = "Hello World! 32byte string here";
    BYTE abReadBuf [32];
    I2CHW_EnableSlave();
    I2CHW_StartBufferedSlave(); //Enable buffered mode
```


[+] Feedback

```
    I2CHW_SetHWAddr(5); //Set slave address to 5

    while (I2CHW_bMonitorBufAccess());  //Wait until buffer is free
    I2CHW_WriteI2CBuf(0, abWriteBuf, 32); //write data

    while (I2CHW_bMonitorBufAccess());
    I2CHW_ReadI2CBuf(0, abReadBuf, 32);  //read data

    while (1)
    {
    //wait for Master to write data
if (I2CHW_bMonitorBufAccess())
{
            while (I2CHW_bMonitorBufAccess());
            //see if master wrote data
            I2CHW_GetMasterWriteData(abReadBuf);
}
    }
}
```

配置寄存器

此部分描述了 I2CHW 用户模块所使用或修改的 PSoC 资源寄存器。

Table 1. 资源 I2C_CFG：组 0 reg[D6] 配置寄存器

位	7	6	5	4	3	2	1	0
值	Reserved	PinSelect	Reserved	Stop IE	Clock Rate[1:0]		Reserved	启用

引脚选择：清除时，SDA 位于 P1[5] 上而 SCL 位于 P1[7] 上。设置时，SDA 位于 P1[0] 上而 SCL 位于 P1[1] 上。

启用停止错误中断：在 I^2C 停止条件下启用 I^2C 中断。

时钟频率 [1,0]：从三个有效时钟频率 50- 100- 400 kbps 中选择。

00b = 100 kHz

01b = 400 kHz

10b = 50k

11b = 保留

启用：启用 I^2C 硬件模块。

Table 2. 资源 I2C_SCR：组 0 reg[D7] 状态控制寄存器

位	7	6	5	4	3	2	1	0
值	Bus Error	Reserved	Stop Status	ACK out	Address	Transmit	Last Recd Bit (LRB)	Byte Complete

Bus Error：表示检测到总线错误状况。

Stop Status：表示检测到 I^2C 停止状况。

ACK out：指示 I^2C 模块对所收到的字节进行确认 (1) 或否认 (0)。

[+] Feedback

Address： 所接收或所发送的字节是一个地址。

Transmit： 该位用于设置后续字节传输的移位器方向。 该移位器始终从 I^2C 总线移入数据，但写入 “1” 则启用移位器输出，以驱动 SDA 输出线。 因为向该寄存器写入将启动下一次传输，所以在写入该位之前，必须将数据写入到数据寄存器。 在接收模式下 (0)，执行该写入之前，必须从数据寄存器读取先前已接收的数据。 固件会从已接收的从器件地址的 RW 位判断相应方向。 该方向控制仅对数据传输有效。 地址字节的方向由硬件决定。

Last Received Bit (LRB)： 在发送序列中最后一个收到位 （第 9 位）的值，来自目标器件的确认 / 否认状态。

Byte Complete： 已收到 8 个数据位。 对于接收模式，总线在等待确认 / 否认应答时停顿。 对于发送模式，同样接收 Ack Nak （参见 LRB），并且总线停顿，等待下一次操作。

Table 3. 资源 I2C_DR： 组 0 reg[D8] 数据寄存器

位	7	6	5	4	3	2	1	0
值	Data							

所接收或所发送的数据。 要发送数据，必须在向 I2C_SCR 寄存器写入前加载该寄存器。 所接收的数据从此寄存器中读取。 寄存器中可能包含地址或数据。

Table 4. 资源 I2C_XCFG： 组 0 reg[C8] I2C 扩展配置寄存器

位	7	6	5	4	3	2	1	0
值	Reserved	Reserved	Reserved	Reserved	No BC Int	Reserved	Buffer Mode	HW Addr EN

No BC Int： 在兼容模式中，每个已接收或发送的字节会生成 “ 字节完成中断 ”。

Buffer Mode： 决定增强型缓冲区模块的运行模式。

HW Addr EN： 启用硬件地址比较。 对于不同配置，请参考下表：

HW Addr EN	Buffer Mode	NoBCInt	字节完成中断	时钟 (SCL) 停顿
启用	启用	启用	无中断	无停顿
		禁用	对于每个字节生成中断。	
启用	禁用	启用	对于每个字节生成中断。[a]	SCL 在每个字节停顿。
		禁用	对于每个字节生成中断。	
禁用	启用	启用	仅在地址字节生成。[b]	SCL 仅在地址字节停顿。
		禁用	对于每个字节生成中断。	
禁用	禁用	启用	对于每个字节生成中断。[c]	SCL 在每个字节停顿。
		禁用	对于每个字节生成中断。	

a. 启用 HWAddr 且禁用缓冲区模式时，启用 NoBCInt 没有任何影响，甚至对于地址字节也如此。 因为接收 / 发送位由 CPU 设置。 对于发送操作，要发送的字节必须加载到 I2C_Data 寄存器中。

[+] Feedback

b. 在此配置中，启用 NoBCInt 仅对地址字节无影响。因为 CPU 必须向 I2C_SCR 寄存器的 ACK 位写入，以确认／否认地址字节。

c. 在兼容模式下，启用 NoBCInt 没有任何影响。

Table 5. 资源 I2C_XSTAT：组 0 reg[C9] I2C 扩展状态寄存器

位	7	6	5	4	3	2	1	0
值	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Dir	Slave Busy

Dir：指出当前缓冲区传输的方向。“1” - 表示主控读取。“0” - 表示主控写入。仅在将 Slave Busy 位设置为“1”时有效。

Slave Busy：这是在硬件比较中设置，通过以下停止信号复位。轮询该位用于确定从器件何时忙碌，以及何时在访问缓冲区模块。

Table 6. 资源 I2C_ADDR：组 0 reg[CA] I2C 地址寄存器

位	7	6	5	4	3	2	1	0
值	Reserved	Slave Address						

这七位为从器件所属的设备地址保留

Table 7. 资源 I2C_BP：组 0 reg[CB] I2C 基本地址指针寄存器

位	7	6	5	4	3	2	1	0
值	Reserved	Reserved	Reserved	I^2C Base Pointer				

该寄存器的值仅在每次开始 I^2C 写入数据操作时修改。在指定的写入数据操作中，I^2C 主控必须始终在从器件地址之后的第一个数据字节中提供该寄存器的值。执行读取操作时，主控不必设置该寄存器的值。该寄存器的当前值也可直接读取。

Table 8. 资源 I2C_CP：组 0 reg[CC] I2C 当前地址指针寄存器

位	7	6	5	4	3	2	1	0
值	Reserved	Reserved	Reserved	I^2C Current Pointer				

设置 I2C_BP 寄存器的同时，也会设置该寄存器，并且两者的设置值相同。在每次完成当前 I^2C 数据操作的数据字节后，该寄存器的值按 1 递增。该寄存器的值始终用于决定数据的读取位置或写入位置。

Table 9. 资源 CPU_BP：组 0 reg[CD] CPU 基本地址指针寄存器

位	7	6	5	4	3	2	1	0
值	Reserved	Reserved	Reserved	CPU Base Pointer				

该寄存器值完全通过 CPU 控制的 IO 写入来控制。写入该寄存器时，也使用相同的值更新当前的地址指针 CPU_CP。第一次从 I2C_ BUF 寄存器读取或写入的操作起始于该地址。固件确保从器件始终具有有效的数据，或者在覆盖之前读取数据。

[+] Feedback

Table 10. 资源 CPU_CP: 组 0 reg[CE] CPU 当前地址指针寄存器

位	7	6	5	4	3	2	1	0
值	Reserved	Reserved	Reserved	CPU Current Pointer				

设置 CPU_BP 寄存器的同时，也会设置该寄存器，并且两者的设置值相同。 无论何时对 I2C_BUF 寄存器执行写入或读取，CPU_CP 都会自动递增。

Table 11. 资源 I2C_BUF: 组 0 reg[CF] I2C 数据缓冲区寄存器

位	7	6	5	4	3	2	1	0
值	Data Buffer							

I^2C 数据缓冲区寄存器 (I2C_BUF)) 是 CPU 到数据缓冲区的读取和写入接口。 无论何时读取该寄存器，都会返回 CPU 当前指针 (CPU_CP)) 所指向位置处的数据。 同样，无论何时写入该寄存器，数据都会传输到缓冲区，并在 CPU 当前指针 (CPU_CP).) 所指定的位置处写入数据。在未初始化 RAM 内容的情况 下，无论何时通过 I^2C 或 CPU 接口读取该寄存器，都不会返回有效值。

[+] Feedback

版本历史记录

版本	创作者	说明
1.1	DHA	清除 “I2C 引脚 ” 参数的默认引脚值，因为它会破坏由其他用户模块初始设置的相应引脚的驱动模式。 对启动函数进行以下更改： 1. 将 UM 引脚的 Initial Open-Drain Low 驱动模式更改为 HI-Z 模拟。 2. 启用 I^2C 模块。 3. 给定延时 5 秒的 NOP 指令。 4. 恢复初始的 I^2C 引脚驱动模式。 增加通过影子寄存器更新输出引脚的功能。

Note PSoC Designer 5.1 在所有用户模块数据表中提供了版本历史记录，详细介绍了当前用户模块和之前用户模块之间的区别。

Copyright © 2008-2010 Cypress Semiconductor Corporation. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC Designer™ and Programmable System-on-Chip™ are trademarks and PSoC® is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.