



PSoC[®] Creator[™]

组件创建指南

文档编号：001-95959 版本 **

赛普拉斯半导体公司
198 Champion Court
San Jose, CA 95134-1709
电话（美国）：800.858.1810
电话（国际）：408.943.2600
<http://www.cypress.com>

© 赛普拉斯半导体公司，2007-2015。此处所包含的信息可能会随时更改，恕不另行通知。除赛普拉斯产品内嵌的电路外，赛普拉斯半导体公司不对任何其他电路的使用承担任何责任。也不会以明示或暗示的方式授予任何专利许可或其他权利。除非与赛普拉斯签订明确的书面协议，否则赛普拉斯不保证产品能够用于或适用于医疗、生命支持、救生、关键控制或安全应用领域。此外，对于可能发生运转异常和故障并对用户造成严重伤害的生命支持系统，赛普拉斯不授权将其产品用作此类系统的关键组件。若将赛普拉斯产品用于生命支持系统，则表示制造商将承担因此类使用而招致的所有风险，并确保赛普拉斯免于因此而受到任何指控。

PSoC[®] 和 CapSense[®] 是赛普拉斯半导体公司的注册商标； PSoC Creator[™]、 Programmable System-on-Chip[™]（可编程系统片上）和 Warp[™] 是赛普拉斯半导体公司的商标。该处引用的所有其它商标或注册商标归其各自所有者所有。

所有源代码（软件和/或固件）均归赛普拉斯半导体公司（赛普拉斯）所有，并受全球专利法规（美国和美国以外的专利法规）、美国版权法以及国际条约规定的保护和约束。赛普拉斯据此向获许可者授予适用于个人的、非独占性、不可转让的许可，用以复制、使用、修改、创建赛普拉斯源代码的派生作品、编译赛普拉斯源代码和派生作品，并且其目的只能是创建自定义软件和/或固件，以支持获许可者仅将其获得的产品依照适用协议规定的方式与赛普拉斯集成电路配合使用。除上述指定的用途外，未经赛普拉斯明确的书面许可，不得对此类源代码进行任何复制、修改、转换、编译或演示。

免责声明：赛普拉斯不针对此材料提供任何类型的明示或暗示保证，包括（但不仅限于）针对特定用途的适销性和适用性的暗示保证。赛普拉斯保留在不做出通知的情况下对此处所述材料进行更改的权利。赛普拉斯不对此处所述之任何产品或电路的应用或使用承担任何责任。对于合理预计可能发生运转异常和故障，并对用户造成严重伤害的生命支持系统，赛普拉斯不授权将其产品用作此类系统的关键组件。若将赛普拉斯产品用于生命支持系统中，则表示制造商将承担因此类使用而招致的所有风险，并确保赛普拉斯免于因此而受到任何指控。

产品使用可能受适用的赛普拉斯软件许可协议限制。

目录



1. 简介	11
1.1 组件创建流程概述	11
1.2 规范	12
1.3 参考	12
1.4 修订记录	12
2. 创建项目和组件	13
2.1 赛普拉斯组件要求	13
2.1.1 文件名称	13
2.1.2 名称注意事项	13
2.1.3 文件名称长度限制	13
2.1.4 组件版本	14
2.1.4.1 项目打开时的组件更新	14
2.2 创建库项目	15
2.3 添加组件项目（符号）	16
2.3.1 创建空符号	16
2.3.2 使用向导创建符号	18
3. 定义符号信息	21
3.1 参数简介	21
3.1.1 正式参数与局部参数	22
3.1.2 内置参数	22
3.1.3 表达函数	23
3.1.4 用户定义的类型	24
3.2 定义符号参数	25
3.3 添加参数验证程序	26
3.4 添加用户定义类型	27
3.5 指定文档属性	28
3.5.1 创建外部组件	29
3.5.2 定义目录位置	29
3.6 定义格式形状属性	30
3.6.1 常用的形状属性	31
3.6.2 高级形状属性	31
4. 添加一个实现	33
4.1 使用原理图实现组件	35
4.1.1 添加一个原理图	35
4.1.2 完成原理图	36
4.1.2.1 设计范围资源（DWR）设置	36
4.2 创建原理图宏	37
4.2.1 添加一个原理图宏文档	37

4.2.2	定义宏	38
4.2.3	版本管理	38
4.2.4	组件更新工具	38
4.2.5	宏文件命名规则.....	39
4.2.5.1	宏和符号拥有相同的名称.....	39
4.2.6	文档属性	39
4.2.6.1	组件目录放置.....	39
4.2.6.2	摘要文本	39
4.2.6.3	隐藏属性	39
4.2.7	宏数据手册	39
4.2.8	宏的后期处理	39
4.2.9	示例.....	39
4.3	实现 UDB 组件.....	41
4.3.1	UDB 硬件的简介	41
4.3.1.1	UDB 概况.....	41
4.3.1.2	数据路径操作.....	42
4.3.2	使用 UDB 编辑器实现	45
4.3.2.1	UDB 编辑器的说明	45
4.3.2.2	添加 UDB 文档.....	47
4.3.2.3	完成 UDB 文档.....	47
4.3.2.4	版本和组件更新工具	48
4.3.2.5	UDB 编辑器语法.....	48
4.3.2.6	UDB 元素.....	49
4.3.2.7	API	60
4.3.3	使用 Verilog 实现硬件	61
4.3.3.1	Verilog 文件的要求	61
4.3.3.2	添加 Verilog 文件	62
4.3.3.3	完成 Verilog 文件	62
4.3.4	UDB 元素	62
4.3.4.1	时钟 / 使能规范	63
4.3.4.2	数据路径	63
4.3.4.3	控制寄存器	68
4.3.4.4	状态寄存器	69
4.3.4.5	Count7	71
4.3.5	固定模块	72
4.3.6	设计范围资源	72
4.3.7	何时使用赛普拉斯所提供的基元代替逻辑	72
4.3.8	用于创建组件的 Warp 性能	72
4.3.8.1	生成语句	72
4.4	使用软件实现组件.....	74
4.5	移除某个组件	75
5.	仿真硬件	77
5.1	仿真环境	77
5.2	模型位置	78
5.3	测试工作台开发	78
5.3.1	提供 CPU 时钟	78
5.3.1.1	CPU 时钟示例	78
5.3.2	寄存器访问任务.....	79
5.3.2.1	FIFO 写入	79
5.3.2.2	FIFO 读取	80
5.3.2.3	寄存器读取	80

5.3.2.4	寄存器写入.....	80
5.3.2.5	状态读取	80
5.3.2.6	控制器读取.....	80
6.	添加 API 文件	81
6.1	API 概况	81
6.1.1	API 生成	81
6.1.2	文件命名规范.....	81
6.1.3	API 模板扩展	81
6.1.3.1	参数	81
6.1.3.2	用户定义类型	82
6.1.4	有条件的 API 生成情况.....	83
6.1.5	Verilog 层次更换.....	83
6.1.6	合并区域.....	83
6.1.7	API 案例	83
6.2	将 API 文件添加到组件内	84
6.3	完成 .c 文件.....	84
6.4	完成 .h 文件.....	85
7.	自定义组件	87
7.1	使用源代码的自定义程序	87
7.1.1	保护使用了源代码的自定义程序.....	87
7.1.2	开发流程.....	87
7.1.3	添加源文件	88
7.1.4	在 “Custom” 中创建子目录.....	88
7.1.5	添加资源文件.....	89
7.1.6	命名类别 / 自定义程序.....	89
7.1.7	指定汇编参考.....	89
7.1.8	自定义程序缓存器	89
7.2	预编译组件自定义程序.....	90
7.3	使用指南	90
7.3.1	使用不同的命名空间.....	90
7.3.2	使用不同的外部依赖属性.....	91
7.3.3	使用通用的组件共享代码.....	91
7.4	自定义示例:	91
7.5	接口.....	91
7.5.1	由组件实现的接口	92
7.5.2	PSoC Creator 提供的接口	93
7.5.3	自定义程序中的时钟查询.....	94
7.5.3.1	ICyTerminalQuery_v1.....	94
7.5.3.2	ICyClockDataProvider_v1	94
7.5.4	时钟 API 支持	94
8.	添加调试支持	95
8.1	调试框架	95
8.2	架构.....	96
8.3	调试 API.....	96
8.3.1	LaunchTuner API（启动调试器 API）	96
8.3.2	通信 API（ICyTunerCommAPI_v1）	96
8.4	传输参数	97
8.5	组件调试器 DLL	97
8.6	通信设置	97

8.7	启动调试器.....	97
8.8	固件交警（Firmware Traffic Cop）.....	98
8.9	组件更改.....	98
8.9.1	通信数据.....	98
8.10	简单的调试器.....	99
9.	添加 Bootloader 支持	101
9.1	固件.....	101
9.1.1	保护.....	101
9.1.2	函数.....	101
9.1.2.1	void CyBtldrCommStart(void).....	102
9.1.2.2	void CyBtldrCommStop(void).....	102
9.1.2.3	void CyBtldrCommReset(void).....	102
9.1.2.4	cystatus CyBtldrCommWrite(uint8 *data, uint16 size, uint16 *count, uint8 timeOut).....	102
9.1.2.5	cystatus CyBtldrCommRead(uint8 *data, uint16 size, uint16 *count, uint8 timeOut).....	103
9.1.3	自定义 Bootloader 接口.....	103
10.	完成创建组件	105
10.1	添加 / 创建数据手册.....	105
10.2	添加控制文件.....	106
10.3	添加 / 创建 Debug XML（调试 XML）文件.....	107
10.3.1	XML 格式.....	107
10.3.2	示例 XML 文件.....	109
10.3.3	示例窗口.....	111
10.3.3.1	选择组件实例调试窗口.....	111
10.3.3.2	组件实例调试窗口.....	112
10.3.4	寄存器窗口.....	114
10.4	添加 / 创建 DMA 功能文件.....	115
10.4.1	将 DMA 功能文件添加到一个组件内：.....	115
10.4.2	修改组件头文件：.....	115
10.4.3	完成添加 / 创建 DMA 功能文件.....	116
10.4.3.1	类别名称.....	116
10.4.3.2	Enabled（使能）.....	118
10.4.3.3	Bytes In Burst 参数.....	119
10.4.3.4	“Bytes in Burst is Strict” 字段.....	120
10.4.3.5	多层并行访问路径的宽度.....	121
10.4.3.6	Inc Addr.....	121
10.4.3.7	“Each Burst Requires A Request”（每次突发需要一个请求）字段.....	122
10.4.3.8	位置名称.....	123
10.4.4	DMA 功能文件示例：.....	125
10.5	添加 / 创建 .cystate XML 文件.....	126
10.5.1	将 .cystate 文件添加到一个组件内.....	127
10.5.2	状态.....	127
10.5.3	状态信息.....	127
10.5.3.1	注释类型.....	127
10.5.3.2	默认信息.....	127
10.5.4	最佳实践.....	127
10.5.5	XML 格式.....	128
10.5.6	.cystate 文件的示例.....	129
10.6	添加静态库.....	130

10.6.1	最佳实践.....	131
10.7	添加依赖属性	132
10.7.1	添加用户依赖属性	132
10.7.2	添加默认依赖属性	134
10.8	编译项目	136
11.	最佳实践	137
11.1	时钟.....	137
11.1.1	UDB 时钟架构注意事项.....	137
11.1.2	组件时钟注意事项	137
11.1.3	UDB 到芯片资源的时钟的注意事项.....	138
11.1.4	UDB 到输入 / 输出时钟的注意事项.....	138
11.1.5	触发器的亚稳态	138
11.1.6	超出时钟域范围	139
11.1.7	长组合路径的注意事项	139
11.1.8	同步与异步时钟	139
11.1.9	使用 cy_psoc3_udb_clock_enable 基元.....	139
11.1.10	使用 cy_psoc3_sync 组件	141
11.1.11	路由、全局和外部时钟	141
11.1.12	时钟下降沿的隐患	141
11.1.13	常见的时钟规则	141
11.2	中断.....	142
11.2.1	状态寄存器	142
11.2.2	内部中断生成和屏蔽寄存器	143
11.2.3	睡眠间隔保持.....	143
11.2.4	FIFO 状态信号.....	144
11.2.5	缓冲区上溢	145
11.2.6	缓冲区下溢	145
11.3	DMA.....	146
11.3.1	用于数据传输的寄存器	147
11.3.2	状态寄存器	147
11.3.3	多层并行访问路径的数据带宽	148
11.3.4	FIFO 动态控制说明	149
11.3.5	数据路径条件 / 数据生成	149
11.3.6	UDB 局部总线配置接口.....	150
11.3.7	UDB 对寻址.....	150
11.3.7.1	工作寄存器地址空间	151
11.3.7.2	8 位工作寄存器访问	151
11.3.7.3	16 位工作寄存器地址空间.....	152
11.3.7.4	16 位工作寄存器地址的限制	152
11.3.8	DMA 总线的使用	153
11.3.9	DMA 通过到突发时间	153
11.3.10	组件 DMA 的功能	154
11.4	低功耗支持.....	154
11.4.1	功能的要求	154
11.4.2	设计注意事项.....	154
11.4.3	固件 / 应用编程接口要求	154
11.4.3.1	数据结构模板	154
11.4.3.2	保存 / 恢复方法	155
11.4.3.3	添加使能和停止功能	156
11.5	组件封装	156
11.5.1	层级设计	156

11.5.2	参数化	160
11.5.3	组件设计中的注意事项	160
11.5.3.1	使用资源	160
11.5.3.2	电源管理	160
11.5.3.3	组件开发	161
11.5.3.4	测试组件	162
11.6	Verilog	163
11.6.1	Warp: PSoC Creator 合成工具	163
11.6.2	可合成的编码指南	163
11.6.2.1	阻塞与无阻塞赋值	163
11.6.2.2	Case 语句	164
11.6.2.3	参数处理	166
11.6.2.4	锁存器	168
11.6.2.5	复位和设置	168
11.6.3	优化	168
11.6.3.1	性能优化	168
11.6.3.2	大小优化	169
11.6.4	资源选择	170
11.6.4.1	数据路径	171
11.6.4.2	PLD 逻辑	171
A.	表达式评估工具	173
A.1	评估上下文	173
A.2	数据类型	173
A.2.1	Bool	173
A.2.2	Error	173
A.2.3	Float	174
A.2.4	Int	174
A.2.5	String	174
A.3	数据类型转换	175
A.3.1	Bool	175
A.3.2	错误	175
A.3.3	浮点	175
A.3.4	整数	175
A.3.5	字符串	176
A.3.5.1	bool-ish 字符串	176
A.3.5.2	Float-ish 字符串	176
A.3.5.3	Int-ish 字符串	176
A.3.5.4	其他子类型的字符串	177
A.4	运算符	177
A.4.1	算术运算符 (+、-、*、/、%、unary + (一元加)、unary - (一元减))	177
A.4.2	数值比较运算符 (==、!=、<、>、<=、>=)	177
A.4.3	字符串比较运算符 (eq、ne、lt、gt、le、ge)	178
A.4.4	字符串串联运算符 (.)	178
A.4.5	三元运算符 (?)	178
A.4.6	Cast 运算符	178
A.5	字符串插值	178
A.6	用户自定义的数据类型 (枚举)	178
B.	数据路径配置工具	179
B.1	通用功能	179
B.2	框架	180

B.2.1	界面	180
B.2.2	菜单	181
B.2.2.1	File（文件）菜单	181
B.2.2.2	Edit（编辑）菜单	181
B.2.2.3	View（视图）菜单	181
B.2.2.4	Tools（工具）菜单	181
B.2.2.5	Help（帮助）菜单	181
B.3	常见任务	182
B.3.1	启动数据路径配置工具	182
B.3.2	打开 Verilog 文件	182
B.3.3	保存文件	182
B.4	使用位字段参数	183
B.4.1	将参数添加到枚举位字段内	183
B.4.2	将参数添加到掩码位字段内	183
B.4.3	位字段的依赖属性	184
B.5	使用各种配置	184
B.5.1	配置命名	184
B.5.2	编辑配置	185
B.5.3	复制、粘贴配置	185
B.5.4	复位配置	185
B.6	使用数据路径实例	186
B.6.1	创建新的数据路径实例	186
B.6.2	删除数据路径实例	186
B.6.3	设置初始寄存器值	186

1. 简介



该指南提供了有助于创建 PSoC Creator 组件的指导信息。高级用户使用该指南来创建复杂的组件，以便其他用户可以使用这些组件来与 PSoC Creator 进行交互。此外，该指南也提供了一些基本原理，这些内容对想要创建自己组件的初学者非常有用。该章节包括：

- 组件创建流程概述
- 规范
- 参考
- 修订记录

1.1 组件创建流程概述

创建组件的过程包括以下高级步骤。欲了解更多信息，请参考该指南中的各个章节。

- 创建库项目（[章节 2](#)）
- 创建组件 / 符号（[章节 2](#)）
- 定义符号信息（[章节 3](#)）
- 创建实现（[章节 4](#)）
- 仿真硬件（[章节 5](#)）
- 创建 API 文件（[章节 6](#)）
- 自定义组件（[章节 7](#)）
- 添加调试支持（高级）（[章节 8](#)）
- 添加 bootloader 支持（如需要）（[章节 9](#)）
- 添加 / 创建文档以及其他文件 / 文档（[章节 10](#)）
- 创建并测试组件（[章节 10](#)）

注意：这些章节提供了相关信息的逻辑组合，并介绍了创建组件的一种工作流程（有多种流程）。请参考[章节 11](#) 以了解创建组件时的各种最佳实践。

1.2 规范

下表列出了本指南使用的规范：

规范	使用说明
Courier New 字体	显示文件位置和源代码： C:\...cd\icc\, user entered text
斜体	显示文件名和参考文档： <i>sourcefile.hex</i>
[方括号、粗体]	显示程序中的键盘命令： [Enter] 或 [Ctrl] [C]
File > New Project	表示菜单路径： File（文件）> New Project（新建项目）> Clone（复制）
粗体	显示程序中的命令、菜单路径和选项以及图标名称： 单击 Debugger 图标，然后单击 Next 。
文本显示在灰色框中	显示仅针对 PSoC Creator 或 PSoC 器件的警告和功能。

1.3 参考

本指南是有关 PSoC Creator 的一系列文档之一。如需要，请参考以下文档：

- PSoC Creator 帮助
- PSoC Creator 自定义 API 参考指南
- PSoC Creator 调谐器 API 参考指南
- PSoC Creator 系统参考指南
- PSoC 3/5 技术参考手册（TRM）
- Warp™ Verilog 参考指南

1.4 修订记录

文档标题：PSoC Creator 组件创建者指南		
文档编号：文档编号：001-95959		
修订版	日期	变更说明
**	01/27/2015	本文档版本号为 Rev**，译自英文版 001-42697 Rev*N。

2. 创建项目和组件



该章节介绍了使用 PSoC Creator 来创建库项目以及添加符号的基本步骤。

2.1 赛普拉斯组件要求

应在 PSoC Creator 中进行所有的组件开发，但需要使用附加工具（如[位于第 179 页上的数据路径配置工具](#)）。要在指定的组件中进行所有代码和原理图编辑、符号创建、接口定义、文档创建等操作。

- 赛普拉斯所生产的所有组件都要配有更改日志。将作为单独组件的更改日志添加到组件内（类似于数据手册）。该日志文件应该是一个简单的文本文件。数据手册应该包含组件每个新版本的“更改内容”章节。
- 请确保组件名称中带有组件版本。此外，在显示的名称中不包含版本信息（目录放置），因为工具会为您添加版本信息。
- PSoC Creator 安装包含了组件的所有版本。因此，请确保在 PSoC Creator 的总结内容中提供了指定组件的所有版本。

2.1.1 文件名称

组件文件名称（包括任意版本编号）要与区分大小写的 C、UNIX 和 Verilog 兼容。

PSoC Creator 将保持每个组件的“givenName”（指定名称）和“storageName”（存储名称）。givenName 可能是大小写混合的字母；storageName 是 givenName 的小写字母版本。PSoC Creator 使用 GUI 中的 givenName，查找时将 givenName 转换为 thestorageName，并使用 storageName 将文件存储到磁盘内。用户可以更改某个组件的 givenName 中的字母。

2.1.2 名称注意事项

当创建组件时，组件的所有元素名称要与组件名称相同（除原理图宏、定制器源文件和 API 源文件外），因此建议选择合适的名称。

2.1.3 文件名称长度限制

组件名称不能超过 40 个字符，资源文件名称不能超过 20 个字符。当组件在最终用户设计中进行扩展时，较长的文件名称会导致路径名称的长度出现问题。

2.1.4 组件版本

赛普拉斯遵循以下组件版本规定（已在内部规范中详细说明）。外部客户创建自己的组件时，建议遵循以下规则。

PSoC Creator 支持组件版本和修改。组件的版本（主要和次要）完全是独立的。很少修改，除非它没有更改实际内容（或许只是文档更改）。

- **主要版本：**可能破坏兼容性的主要更改（如 API 更改、组件符号接口更改、性能更改等）
- **次要版本：**没有进行 API 更改。没有破坏兼容性的错误修复和附加性能
- **修改：**没有进行 API 更改。组件现有性能的文档、错误修复（没有破坏兼容性的）。注意，修改只是组件的一个方面，当用户安装更新后的组件时，修改也会自动更新。

通过将 “_v<major_num>_<minor_num>” 添加到组件名称内，组件名称可以包括版本信息。其中 <major_num> 和 <minor_num> 都是整数，分别表示主要版本和次要版本。例如，CyCounter8_v1_20 是 CyCounter8 组件的 1.20 版本。主编号和次编号本身都是整数，并且不能将其作为一个实数。例如，v1_1 与 v1_10 表示的意思不同，而且 v1_2 先于 v1_10 出现。

修改等级是组件的一方面，并被存储于组件符号内，因此组件名称不会反映修改等级。当组件发生了主要的更改时，如果组件名称包含了版本编号，那么会保护现有的设计。注意，主要版本和次要版本的命名都是由组件创建者决定的。

2.1.4.1 项目打开时的组件更新

PSoC Creator 提供了 <project>.cyversion 文件，以便在打开设计项目时可以检查组件更新情况。通过将 <project>.cyversion 文件添加到现有的库项目文件夹内，组件创建人员可以使用自动检查功能。对于新库项目，将自动生成 <project>.cyversion 文件。

<project>.cyversion 文件位于磁盘上，但不会显示在 Workspace Explorer 中。它和项目的基本名称相同。默认情况下，文件包含以下内容：

```
<project_version> </project_version>
```

为了确保现有的设计项目及时检查更新组件，每次组件的新版本可用时，组件创建者应更新 <project_version> 字段。字段中的文本是形式自由的。为了激活组件更新检查，必须更改该字段的值，并且该值必须与先前使用的值不同。例如：

```
<!-- e.g., version based text -->
<project_version>MyComponentLibrary v2.1</project_version>

<!-- e.g., hash based on name and version of included components -->
<project_version>8ebdf495798f830a62c3e9089764d6df</project_version>
```

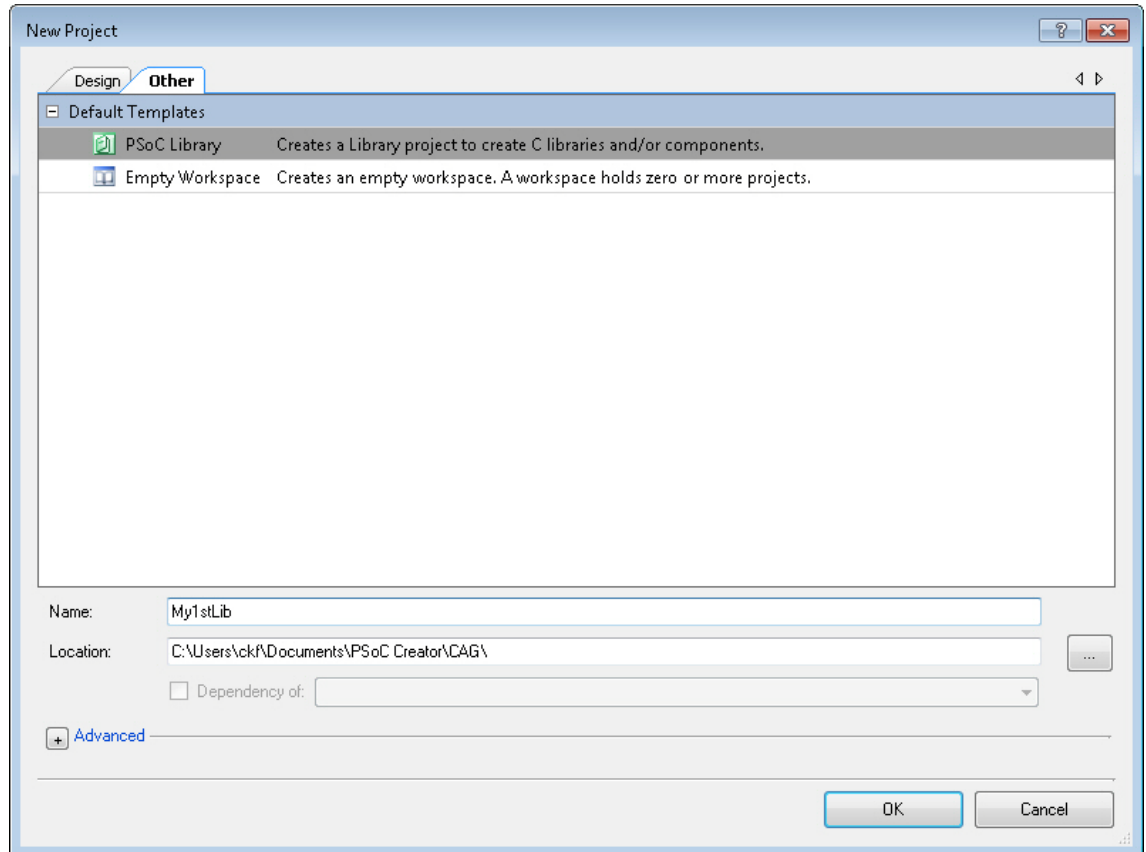
将该字段保留为空（默认值）或从项目文件夹中删除文件会导致 PSoC Creator 失去自动检查功能。此外，如果没有选择 **Tools > Options > Environment** 下的 **Check for up-to-date components...**（检查最新组件）选项，将忽略自动检查功能。

2.2 创建库项目

您可以在设计项目或库项目中创建一个或多个组件。它们的区别主要在于设计项目通常面向特殊器件和特殊设计目标，而库项目只是各种组件的组合。为了使用这些组件，需要添加一个项目，在该项目中，这些组件作为 **PSoC Creator** 的依赖属性。更多有关信息，请参见[位于第 132 页上的添加依赖属性](#)。

创建库项目步骤：

1. 依次点击 **File > New > Project** ，以打开新项目对话框。



2. 点击 **Other**（其他）选项卡，然后选择 **PSoC Library**（PSoC 库）项目模板。
3. 输入项目的 **Name**（名称），并点击省略符号（...）按钮以指定存储项目的 **Location**（位置）。
4. 点击 **OK**。

该项目显示在 **Source** 选项卡下的 Workspace Explorer 目录内。



2.3 添加组件项目（符号）

创建组件流程的第一步是将组件项几种类型中的一种添加到您的项目内（如符号、原理图或 Verilog 文件）。当添加组件项目时，您也间接创建了组件。添加任意组件项的流程都是相同的；但完成每个组件项的步骤却不一样。

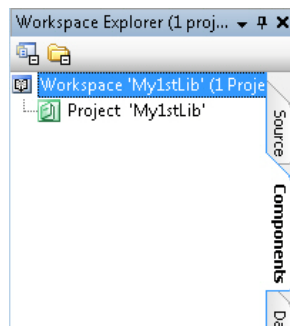
该章节将重点介绍创建一个符号作为第一个组件项目。有两种方法可以创建符号：空符号和符号向导。

注意：您也可以从原理图或 Verilog 中自动生成一个符号，但本章节没有对该流程进行说明。相关指导信息，请参考 PSoC Creator 帮助。

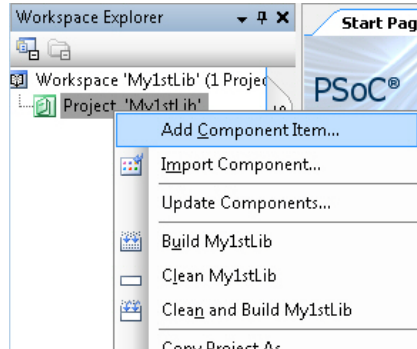
2.3.1 创建空符号

该章节对创建空符号以及添加形状和终端的流程进行了说明。（参考[位于第 18 页上的使用向导创建符号](#)，以了解创建符号的其他方法）。

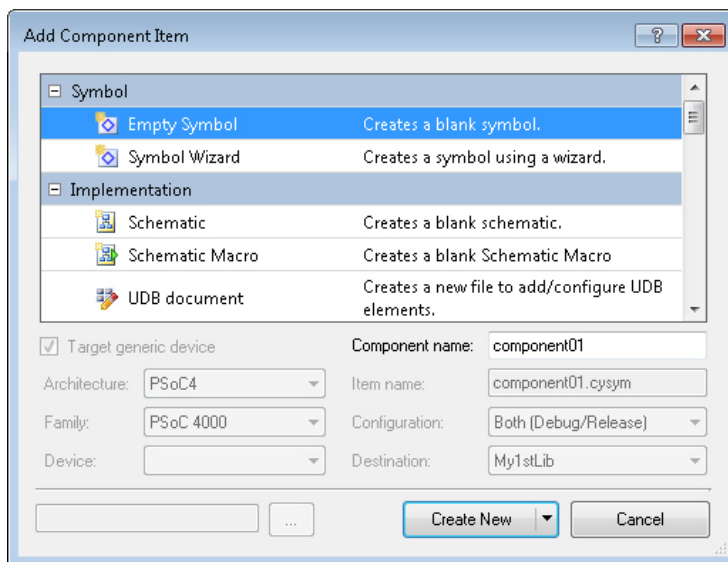
1. 如果需要，请打开相应 PSoC Creator 库项目。
2. 点击 Workplace Explorer 窗口中的 **Components**（组件）选项卡。



3. 右击项目，然后选择 **Add Component Item...**（添加组件项）



这时会出现 **Add Component Item** 对话框。



4. 选择 **Empty Symbol**（空符号）图标。
5. 输入包括了版本信息的 **Component name**（组件名称）。

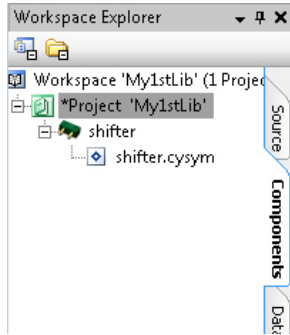
注意：符号和所有组件元素（原理图宏、定制器源文件和 API 源文件除外）将继续使用该组件名称。请参见[位于第 13 页上的名称注意事项](#)和[位于第 14 页上的组件版本](#)。

此外，一个组件中可能只有一个符号文件，并且该符号是通用的。因此，**Target**（目标）选项被禁用。

6. 点击 **Create New**（新建），使 PSoC Creator 创建一个新的符号文件。

注意：您也可以从下拉菜单中点击 **Add Existing**（添加现有），然后选择现有的符号文件，并将其添加到组件中。

Workspace Explorer 树内显示了该符号，其中符号文件（.cysym）作为唯一的节点。



符号编辑器也打开了 `<project_name>.cysym` 文件，因此您可以绘制或导入一个表示组件的图形。

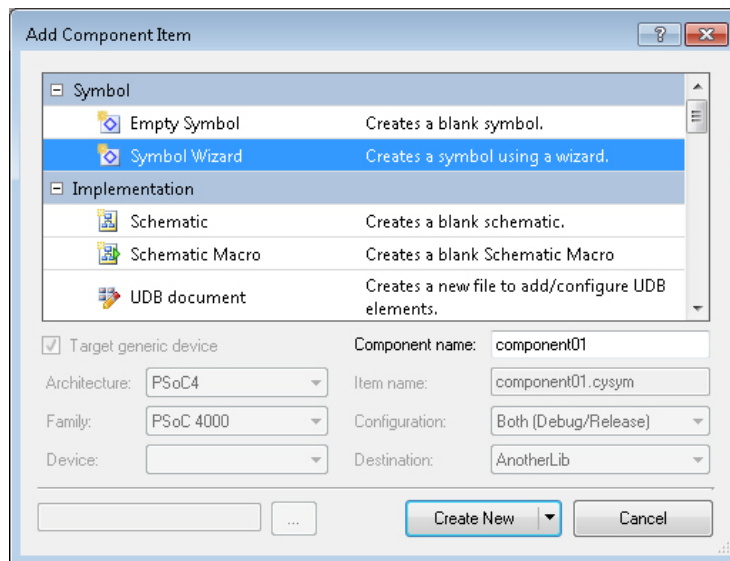
7. 绘制基本的形状和终端，以定义使用符号编辑器的符号。更多有关信息，请参见《PSoC Creator 帮助》部分的内容。

注意： 符号编辑器图纸中间的加号（或十字线）表示您原始的符号图纸。

8. 点击 **File > Save All**，保存您对项目 and 符号文件进行的变更。

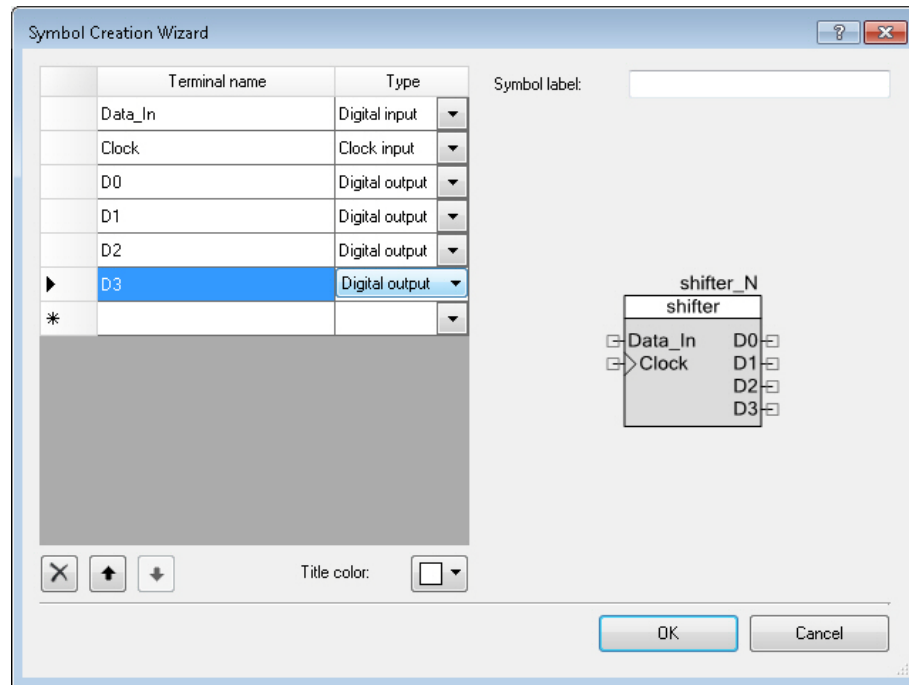
2.3.2 使用向导创建符号

在 Add Component Item（添加组件项）对话框中，除空符号图标外，还包含了符号向导图标。使用该向导模板的好处是：PSoC Creator 会为您创建基本的符号图纸和终端。



1. 依照位于第 16 页上的创建空符号 所列出的说明进行创建组件。
2. 选择符号向导图标而不是选择第 17 上的步骤 4 所列出的空符号图标。

点击添加组件项对话框中的 **OK** 后，会显示符号创建向导。



3. 在 **Add New Terminals** （添加新终端）中，输入您需要添加到符号内的终端的 **Name** （名称）和 **Type** （类型）。

Symbol Preview （符号预览）章节介绍了在符号编辑器图纸上您的符号的显示情况。

4. 如果需要，请使用 **Delete** （删除）、**Up** （上）和 **Down** （下）按键来转移和清除终端。
5. 此外，选择 **Title color** 并指定 **Symbol label** （符号标签）。
6. 点击 **OK**，以关闭符号创建向导。

创建空符号时，**Workspace Explorer** 树内显示了新组件，其中符号文件（.cysym）作为唯一的节点。但是，您的符号编辑器会显示由向导创建的符号，并将其放置在十字线中间。

7. 如果需要，请修改您的符号图纸。
8. 点击 **File > Save All** 以保存您项目和符号文件的变更。

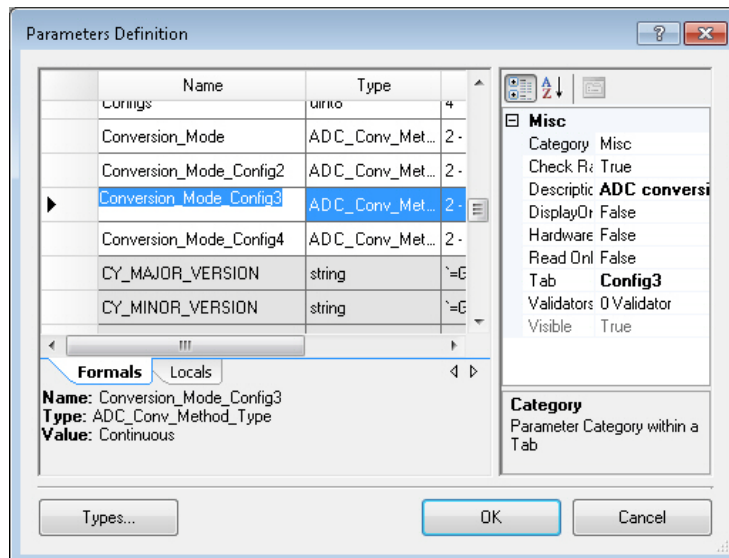
3. 定义符号信息



本章节介绍了定义各种符号信息（如参数、验证程序和属性）的过程。

3.1 参数简介

PSoC Creator 中的参数是一个或多个已命名的表达式集，用于定义某个组件的行为。您通过使用 Parameters Definition（参数定义）对话框定义某个符号的各个参数。



一个参数包含名称、类型、数值以及属性的相关列表。

- 您可根据需要进行命名；但要求保留前缀“CY_”给内置参数（请参见第 22 页上的内置参数）。新参数的名称不能以字符串“CY_”或任何其他形式开始。
- 类型项是一个下拉菜单，它列出了所有有效的数据类型。有关数据类型的说明，请参考第 173 页上的数据类型。
- 该数值是实例化组件的默认值。
- 每个参数的各种属性控制着参数显示以及性能的方式。有关这些属性的详细信息，请参见第 25 页上的定义符号参数。

3.1.1 正式参数与局部参数

符号有两种参数定义：正式和局部。正式参数是符号的主接口；组件的最终用户只能查看和配置正式参数。局部参数主要是用于为组件创建者提供便利；最终用户不能查看或编辑它们。

在一个符号中，正式参数具有固定的默认值，并且局部参数拥有带标识符的表达式，该标识符可以参考其他参数。局部参数表达式中的标识符是正式参数和局部参数的组合。实际上，参数集是一个评估上下文（针对局部参数）。

原理图文档有一个参数集。原理图在编辑器中打开时，它具有一个参数集，该参数集是其符号参数集的直接拷贝。在运行时完成该拷贝，并保证拷贝操作始终与符号同步。在 PSoC Creator 中不能直接看到这些参数，只能通过编辑符号才能看到它们。如果原理图没有符号，那么它便没有参数集。

一个组件的实例包含组件符号中的正式参数。将组件放在原理图中时，该实例会创建组件实例的正式符号参数的拷贝。用户可以将正式参数值修改为表达式。实例中的正式参数可以指的是原理图中存在的任何一个参数，而指的不是其他参数。正式参数是组件开发者将用户的配置值通过设计层级所进行传输的方式。

最终用户不能修改局部参数。局部参数的实例被评估在实例的上下文中。因此，它们可以参考实例中任意其他参数，但不能参考原理图中的任何参数。有关评估上下文以及它们使用情况的详细信息，请参见第 173 页上的表达式评估工具附录。

实例参数的名称与符号参数的匹配；类型无所谓。一个存储器中的实例都有一个活动参数集和单参数集。活动参数集是符号参数集中所包含的各个参数。单参数集是为实例而创建的，但它们的符号已被修改。这些参数不再有效。这样可以进行：最终用户修改它们的库搜索路径；切换为代替的实现或组件的版本；以及切换回它们的原来搜索路径，而不丢失任何数据。

这些单参数是临时的。如果相关符号被刷新，并且正式参数使用了这些名称，那么将激活单参数。原理图参数是母体实例中各实例参数的直接拷贝。

3.1.2 内置参数

内置参数是每个符号的默认定义。不能将内置参数定义从参数定义编辑器中移除。不可修改内置参数的名称和类型。目前，PSoC Creator 包括以下内置参数：

正式参数：

- **CY_MAJOR_VERSION** — 该参数包含了实例的主要版本号。
- **CY_MINOR_VERSION** — 该参数包含了实例的次要版本号。
- **CY_REMOVE** — 该参数是传输给 elaborator 的标志。该标志会通知给 elaborator 要删除网络列表中的实例。它与活动复用器结合使用，从而允许组件创建者能在单一的原理图中动态切换组件的两个不同实现。默认值为 false（表示保持实例的周围）。
- **CY_SUPPRESS_API_GEN** — 该参数用于表示一个特殊实例不应该具有为它而生成的 API（虽然，组件具有 API）。默认值为 “false”。
- **CY_VERSION** — 该参数显示了 PSoC Creator 组件的版本和编译信息。
- **INSTANCE_NAME** — 这是一个特别的参数。在编辑器中，它带有实例的短名称。这是用户想重新命名实例时需要编辑的参数。在复杂的设计中，可以通过它访问 “完全” 或 “分层” 的实例名称。这是在 API 文件中应所使用（通过 ``$INSTANCE_NAME``）的名称，用以确保不会发生任意冲突。换句话说，这个名称几乎总是组件创建者需要参考的名称，并且它有一个相应的值。

注意：在表达式评估工具和 API 与 HDL 生成中使用 `INSTANCE_NAME` 参数时，它会返回另一个值。对于表达式评估，它会返回实例的叶（leaf）名称；而适用于 API 和 HDL 生成时，它会返回实例的路径名称。

局部参数：

- **CY_COMPONENT_NAME** — 该参数包含了组件基本目录（包含了符号的目录）的文件系统路径。只有制定后，它才有效。制定前，该参数包含 “__UNELABORATED__” 值。
- **CY_CONTROL_FILE** — 该参数用于指定组件实例的控制文件。内部使用该参数为特定的组件定义固定的位置特性。如果使用了控制文件，则默认值（<:default:>）指的是通用级控制文件（即，<ComponentName>.ctl）。不要修改该值。
有关如何在设计中创建和使用控制文件的信息，请参考 PSoC Creator 帮助中的 “控制文件” 主题。可以查看 [第 106 页上的添加控制文件](#)。
- **CY_INSTANCE_SHORT_NAME** — 如果由于某种原因组件创建人员需要使用实例名称的缩写形式（甚至在已制定的设计中），它可以通过这个参数进行访问。最终用户永远看不到该参数。不能对它进行编辑。每当用户修改 `INSTANCE_NAME` 参数时，其值也会自动被更新。

3.1.3 表达函数

任何人都可以通过在一个表达式中使用下面各函数来访问它们。

- `string GetComponentName()` — 返回组件的名称。
- `string GetErrorText(error)` — 返回错误类型值中存在的错误信息。
- `string GetHierInstanceName()` — 返回层级实例名称。这是指定给某个实例的设计范围唯一名称。完整的层级实例名称不适用于原理图编辑器。在这种情况下，它会返回相同的值：`GetShortInstanceName()`。
- `int GetMajorVersion()` — 返回组件的主要版本编号。
- `string GetMarketingNameWithVersion()` — 返回市场全名和 PSoC Creator 的当前运行版本。
- `int GetMinorVersion()` — 返回组件的次要版本。
- `string GetShortInstanceName()` — 返回实例名称的缩写形式。这是用户在原理图中键入的名称。
- `string GetStateForDisplay()` — 返回组件 `cystate` 文件的字符串。返回值可以是一个空的字符串、“Prototype”（原型）或“Obsolete”（过时）。一个空的字符串可表示：组件处于生产状态；没有状态文件；在文件中存在错误条目；或目标芯片没有条目。
- `bool IsAssignableName(string)` — 如果具体的实例名称被合理地应用于组件实例，将返回 ‘true’ 值。它是重新命名组件实例时通常用于验证用户输入的函数。
- `bool IsError(anyType)` — 如果参数类型是错误的，将返回 ‘true’ 值；否则将返回 ‘false’ 值。
- `bool IsValidCCppIdentifierName(string)` — 如果给定的字符串是合法的 C 和 C++ 标识符名称，将返回 ‘true’；否则返回 ‘false’。
- `bool or error IsValidCCppIdentifierNameWithError(string)` — 如果给定的字符串为合法的 C 和 C++ 标识符名称，将返回布尔 ‘true’。如果标识符不是合法的 C 或 C++ 标识符，那么会返回错误类型值以及相应的信息。
- `GetStateForDisplay()` — 获取作为字符串的组件支持状态（比如原型、过时、非兼容）。
- `GetMarketingNameWithVersion()` — 获得应用的全名和版本（比如 “PSoC Creator 3.0”）。
- `GetSiliconRevision()` — 获得所选器件的版本。
- `GetDeviceFamily2()` — 获得所选器件的系列（比如，CY8C588）。该函数取代了 `GetDeviceFamily()`（建议不再使用）。

- `GetPartNumber()` — 获得所选器件的型号。
- `GetJtagId()` — 获得所选器件的 JTAG id。
- `GetArchMemberName2()` — 获得所选器件架构的内部名称（例如 PSoC3A、PSoC5LP 等）。该函数替代了 `GetArchMemberName()`（建议不再使用）。
- `GetArchitecture()` — 返回架构名称（例如 PSoC3、PSoC4、PSoC5 等）。


您可以创建包含了一个函数调用的表达式。为了在文本标签中使用某个函数，请使用格式 ``=IsError()'``。如果您想将一个参数值设置为这些函数中某一个的结果，那么只要将该值设置给那个函数便可。同样，这些函数可以用于验证程序的表达式。另请参阅[第 26 页上的添加参数验证程序](#)。

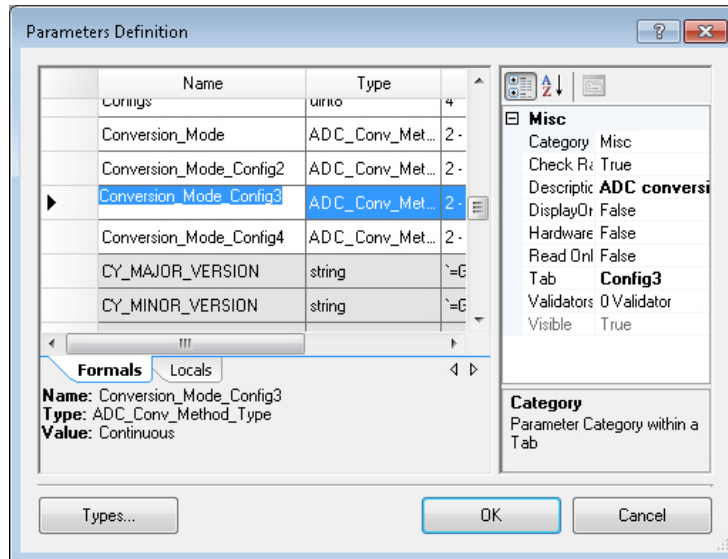
3.1.4 用户定义的类型

使用用户定义的类型（或枚举类型）来定义符号的参数，符号值来自枚举。可以“沿用”用户定义的类型。例如，您可以创建计数器的 UDB 实现，它是 Verilog 实现和一个符号。该符号和其他符号都被放置在顶层原理图中，便于创建固定的函数块。您可以重新定义所有枚举类型，打开可能性错误或者顶层组件可以使用（继承）较低级组件的枚举类型。请参见[第 25 页上的定义符号参数](#)和[第 27 页上的添加用户定义类型](#)。

3.2 定义符号参数

可以参照下列内容定义符号参数：

1. 通过点击符号编辑器图纸或符号文件选项卡，使符号文件有效。
2. 右键点击并选择 **Symbol Parameters...** （符号参数）打开 **Parameters Definition**（参数定义）对话框。



3. 分别点击 **Formals** 选项卡或 **Locals** 选项卡，将参数定义为正式参数或局部参数。相关详细信息，请参见第 22 页上的**正式参数与局部参数**。
4. 您可以为符号创建任意多个参数。请先在对话框左侧的参数表内，定义某个参数的 **Name**（名称）、**Type**（类型）和 **Value**（数值）项。
注意：参数的默认值只定义了第一次实例化组件时所使用的值。该值只是临时的值，可变的。如果随后组件的默认值被更改，那么该值不会被传输给任何先前被实例化的组件。
5. 在对话框右侧定义每个参数的下列属性：
 - **Category**（类别）— 该符号的类别名称会在 **Parameter Editor** 对话框中显示。这是将各个参数组合在一起的方法。
 - **Check Range**（检查范围）— 该范围为 **true** 时，会在每次进行评估后检查参数值是否处于相应类型的范围内。如果超出该范围，则 **PSoC Creator** 会生成一个表达式评估错误。有效范围如下：

类型	有效范围
枚举类型	枚举类型所定义的整数值
16 位有符号整数类型	-32768 .. x .. 32767
16 位无符号整数类型	0 .. x .. 65535
8 位有符号整数类型	-128 .. x .. 127
8 位无符号整数类型	0 .. x .. 255
布尔类型	true（真）和 false（假）
所有其他类型	所有值均有效

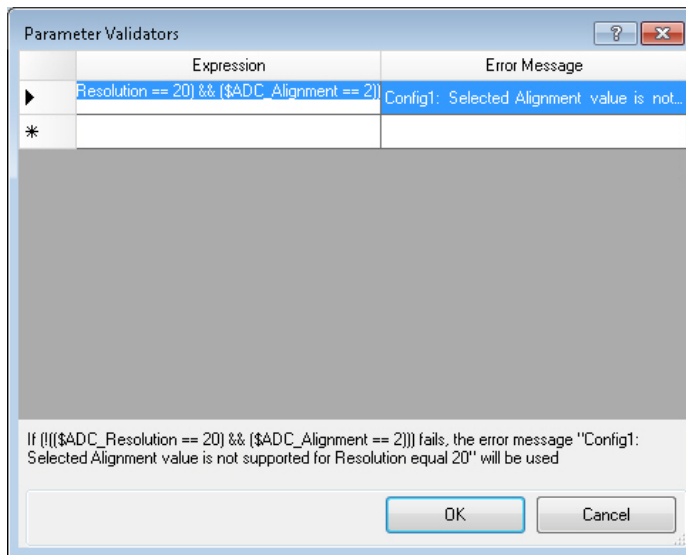
- **Description**（说明）— 是指给最终用户的参数说明；该说明会在 **Parameter Editor** 对话框中显示。
 - **DisplayOnHover**（鼠标悬停显示）— 指定将鼠标悬停在原理图编辑器的实例时是否显示该参数值。
 - **Hardware**（硬件）— 指定是否将该参数包含在生成的 Verilog 网表中。
 - **Read Only**（只读）— 指定最终用户是否能够修改该参数。
 - **Tab**（选项卡）— 该符号的选项卡名称会在原理图编辑器的 **Parameter Editor** 对话框中显示。
 - **Validators**（验证程序）— 是验证表达式的一个或多个参数。有关如何使用该字段的信息，请参见第 26 页上的[添加参数验证程序](#)；更多有关 CyExpressions 的信息，请参见第 173 页上的[表达式评估工具附录](#)。
 - **Visible**（可视）— 指定该参数值是否显示在 **Parameter Editor** 对话框中。
6. 当添加完参数时，请点击 **OK**，以关闭对话框。
- 注意：**通过右键点击，然后选择 **Copy Parameter Defintions (Ctrl + C)**（复制参数定义），您可以复制一个或多个参数定义。通过右键点击然后选择 **Paste Parameter Defintions (Ctrl + V)**（粘贴参数定义），可以粘贴它们。

3.3 添加参数验证程序

参数验证表达式被评估在实例上下文中。验证表达式指的是正式参数和局部参数。

要想添加一个参数验证程序，请进行下面的操作：

1. 在 **Parameters Definition** 对话框中，点击 **Validators**（验证程序）字段，然后点击省略号按钮。这时，将出现 **Parameter Validators** 对话框。



2. 请在 **Expression** 字段中，键入验证检查表达式。您可以使用 **\$** 符号参考参数名称。例如：

```
$param1 > 1 && $param < 16
($param1 == 8) || ($param1 == 16) || ($param1 == 24)
```

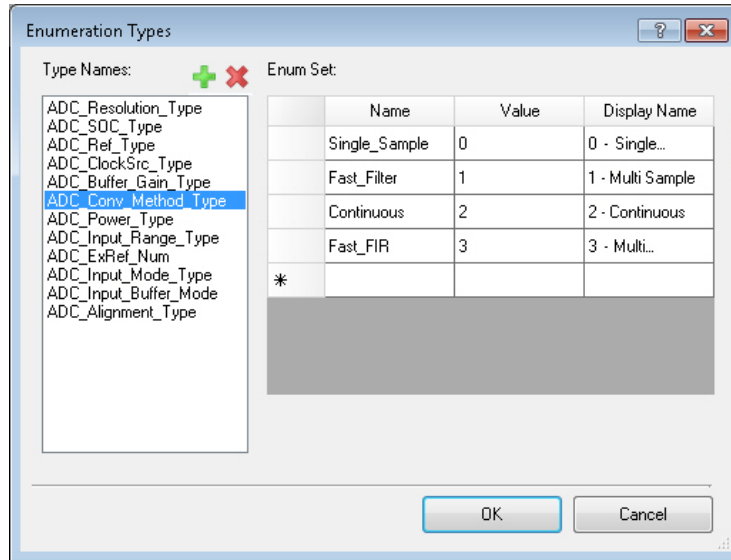
 相关详细信息，请参见第 173 页上的[表达式评估工具附录](#)。
3. 请在 **Error message**（错误信息）字段中，输入一条信息，以显示是否尚未满足验证检查。
4. 要添加其他验证程序，请点击 **Expression** 字段中的空行（标记为 >*），并输入合适的字段。

5. 要删除表达式，请在该行的第一列上点击 “>” 符号，进行删除，并按下 **[Delete]** 键。
6. 点击 **OK**，关闭该对话框。

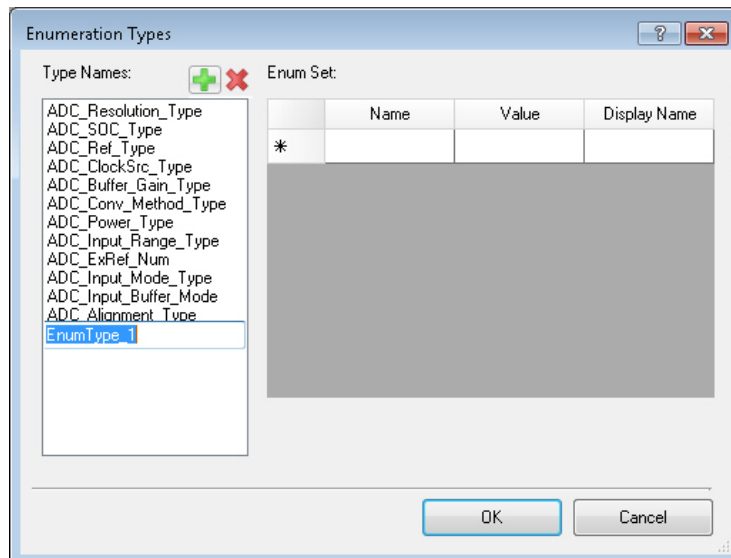
3.4 添加用户定义类型

创建用户定义类型，请参照以下各步骤：

1. 在 **Parameters Definition** 对话框中，点击 **Types...** 按键，以打开 **Enumeration Types**（枚举类型）对话框。



2. 点击 **Type Names** 下面的 **Add** 键，以创建新的枚举类型。



3. 使用您想要的名称替换 “EnumType_1”，然后按下 **[Enter]** 键。
4. 在 **Enum Sets** 下方，点击 **Name** 下的第一行，并键入枚举类型的第一个名称 / 值对的名称；然后在 **Value** 下面键入一个数值或接受默认的设置。

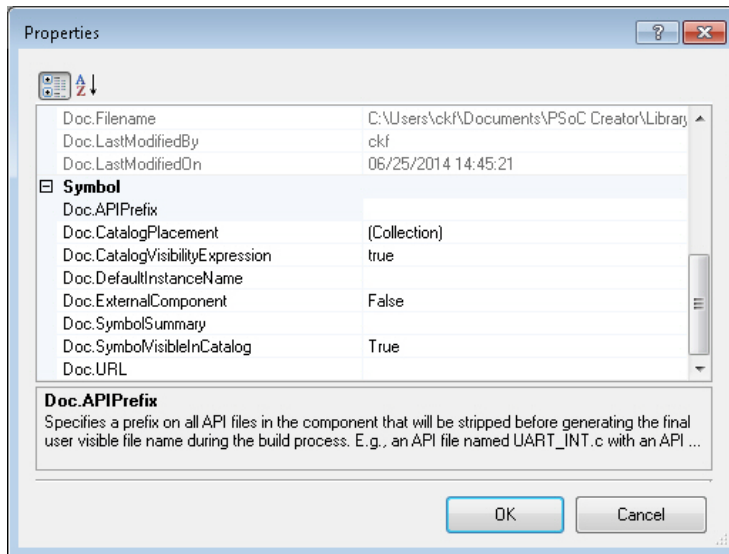
- 此外，可以在 **Display Name** 中输入一个字符串。这样，该字符串将显示在 **Parameters Definition** 对话框的 **Value** 下拉菜单中。

注意： 请不要使用可能被视为一个表达式的标点符号。例如，可以使用 “Clock with UpCnt & DwnCnt”，而不能使用 “Clock + UpCnt & DwnCnt。”

- 根据需要输入枚举组，然后点击 **OK**，以关闭 **Enumeration Types**（枚举类型）对话框。

3.5 指定文档属性

每个符号都包含了用于定义该符号不同内容的属性组。右击符号图标，并选择 **Properties**，以打开 **Properties** 对话框。



您可以为每个组件指定以下文档属性（如适用）：

■ 符号属性

- **Doc.APIPrefix** — 指定在生成代码过程中生成用户可见的最终文件名称前，会将前缀从组件的所有 API 文件上剥离掉。

例如，如果您向某个计数器组件输入 “Counter”（如上面所示），那么所生成的具有实例名称为 “Counter_1” 的组件的头文件将是 *Counter_1.h*。如果未在该属性中键入任何内容，那么生成文件将为 *Counter_1_Counter.h*。

注意： APIPrefix 仅适用于作为磁盘上项目一部分的 API 文件。它不适用于 API 定制器生成的 API 文件。

- **Doc.CatalogPlacement** — 定义了组件在组件目录中显示的方式。请参见第 29 页上的[定义目录位置](#)。
- **Doc.SymbolVisibilityExpression** — 用于输入一个表达式，这样能够在组件目录中显示符号。如果评估该表达式为 ‘false’（假），并且 “Show Hidden Components” 选项（Tools > Options > Design Entry > Component Catalog）未被使能，那么符号（或原理图宏）不会显示在组件目录中。
- **Doc.DefaultInstanceName** — 定义了组件的默认实例名称。如果该属性为空白，那么实例名称将默认为组件名称。

- ❑ **Doc.ExternalComponent** — 指定符号是否是一个外部组件：true（真）或 false（假）。请参考第 29 页上的创建外部组件。请注意，由于传统原因，标签可能被称为“注释”。
- ❑ **Doc.SymbolSummary** — 用于输入显示在组件目录中的简要描述。
- ❑ **Doc.SymbolVisibleInCatalog** — 指定组件是否显示在组件目录中：true 或 false。
- ❑ **Doc.URL** — 用于输入一个完整的网址或磁盘上的文件位置。如果在该字段中输入一个有效值，那么多种菜单项将有效，这样用户可以导航到相应的网页，或打开指定的文件。

3.5.1 创建外部组件

外部组件包含一个符号和各个参数，但没有一个实现。它用于记录设计原理图，通常通过片外器件和连线代表。赛普拉斯提供了多个库中的外部组件，包括：电阻器、电容器、二极管等。要指定将一个组件作为一个外部组件，请执行下面操作：

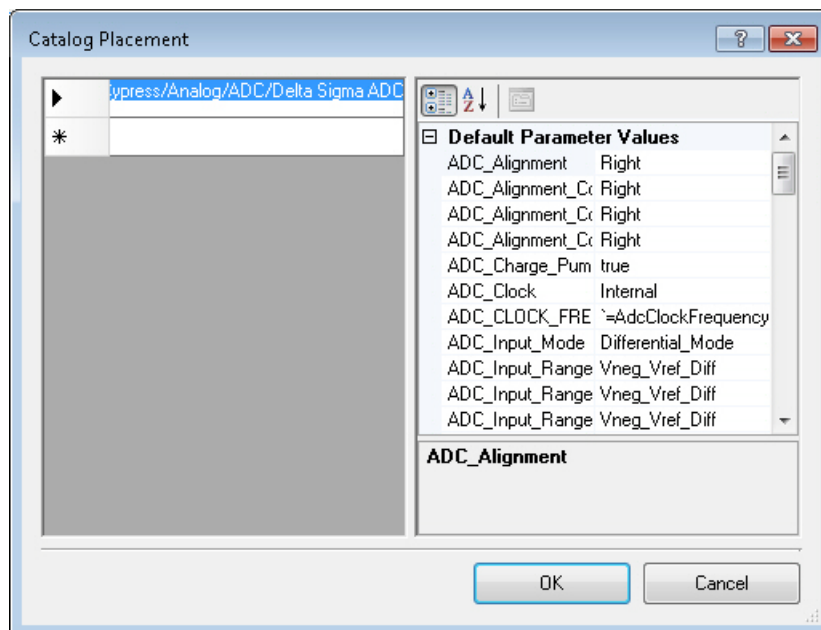
1. 将 Properties 对话框中的 **Doc.ExternalComponent** 属性设置为 true。
这样可以在制定过程中跳过组件，并可以使能数字或模拟线 / 终端被连接时的 DRC 错误。
2. 然后，使用设计元素控制板（DEP）的外部终端  来定义连接到片外资源的符号。

注意：指定为外部组件的组件不能带有非外部终端；但常规组件则可以。

如果外部组件包含非外部终端，那么它会生成 DRC。DRC 警告：组件不会被实现。同样，在外部组件中使用数字或模拟终端也会生成 DRC。该 DRC 会警告：终端未连接任何功能性资源。

3.5.2 定义目录位置

您可以使用 Catalog Placement（目录位置）对话框来定义符号在 Component Catalog 的各种树和选项卡中所显示形式。



如果您没有定义目录位置信息，那么符号会默认显示在 **Default > Components** 下。

1. 通过在 **Properties** 对话框的 **Doc.CatalogPlacement** 字段中点击省略号按钮，可以打开该对话框。
2. 请在对话框左侧的第一行表格内，使用以下语法输入放置定义：

t/x/x/x/x/d

t — 选项卡名称。显示在 **Symbol Catalog** 对话框中的选项卡，它们是按字母顺序（不分大小写）排列的。

x — 树中的节点。至少要有个节点。


d — 符号的显示名称（可选）。如果您未指定显示名称，则使用符号名称；但您必须使用语法：
t/x/。

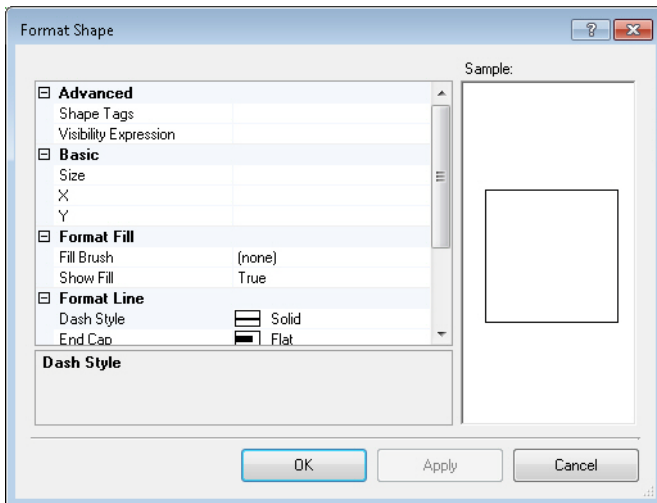
例如，在名称为“PGA”的组件的默认组件目录中，选项卡名称（t）是 **Cypress**，第一个节点（x）是 **Analog**，第二个节点（x）是 **Amplifiers**，因此显示名称（d）是 **PGA: Cypress/Analog/Amplifiers/PGA**。

3. 在对话框右侧的 **Default Parameter Values**（默认参数值）下面，请根据需要为该特定目录位置定义输入默认参数值。
注意：参数的默认值只定义了第一次实例化组件时所使用的值。该值只是临时的值，可变的。如果随后组件的默认值被更改，那么该值不会被传输给任何先前被实例化的组件。
4. 如果需要，请在表格左侧添加更多行，以便输入多个目录位置定义；为每个附加行设置相同或不同的默认参数值。
5. 点击 **OK**，关闭该对话框。

3.6 定义格式形状属性

Format Shape（格式形状）对话框用于定义形状的属性。可用的属性类型会根据所选定的（各）形状而发生改变。例如，文本提供了字体、颜色、大小等信息；直线提供了宽度、笔型、后盖等信息。

要打开该对话框，请选择一个或多个形状，并点击 **Format Shape**  按钮。



3.6.1 常用的形状属性

在该对话框中，几乎所有普通形状的属性都非常明显，它们与您在文字处理或绘图程序中看到的形状相同。对于大多数属性而言，请从下拉菜单中选择一个值。

3.6.2 高级形状属性

某些形状（如实例终端）具有以下各高级属性：

- **Default Expression**（默认表达式）— 将形状的默认值定义为 `<size>'b<value>`，其中 `<size>` = 位数，`<value>` = 二进制值。
- **Shape Tags**（形状标签）— 为一个或多个形状定义一个或多个标签，以用于形状自定义代码；请参考第 87 页上的[自定义组件](#)。
- **Visibility Expression**（可视化表达式）— 定义一个表达式，用于评估是否要显示已选的形状。例如，在 UART 组件中，该属性通过 `rts_n` 和 `cts_n` 终端的变量 `$FlowControl` 定义。如果 `$FlowControl` 被设置为 `true`，那么这些终端将显示在该原理图内；如果该变量被设置为 `false`，那么这些终端将被隐藏。

注意：当设置终端的可视化表达式时，要指定默认表达式。在该终端的可视化表达式被评估为 `false` 时，需要使用它。

4. 添加一个实现



一个组件实现指定了以下内容：

- 使用的以及不受支持的器件
- 如何编译组件
- 内部架构以及它的使用方式

您可以通过 **PSoC Creator** 使用某些设计策略按需要来创建组件。根据您的特殊应用的设计目的，设计策略会根据不同的实现选项而不同。这些选项包括原理图实现、原理图宏、使用通用数字模块（**UDB**）编辑器的硬件实现、使用 **Verilog** 的硬件实现以及软件实现。

原理图

使用原理图（请参考[第 35 页上的使用原理图实现组件](#)）是实现组件最简单和最直接的策略。通过原理图，您可以将各个现有组件放置在设计图纸上。该设计被包装在组件内，该组件具有输入 / 输出终端以及唯一的功能（可用于其它设计中）。如果您正在使用现有的模块来执行新功能，应该使用该方法。

该方法不允许您直接操作 **UDB** 模块，因此将限制 **UDB** 中可用的性能，如数据路径和高级的可编程逻辑器件（**PLD**）特性。如果基于 **UDB** 的单独组件可用，那么可以在基于原理图的组件中使用它，同组件目录中其他所有可用的组件一样。

原理图宏

类似于原理图的实现，可以创建一个包含设计的原理图宏，该设计包括预配置参数和设置以及已相互链接的组件。这样，可以在组件目录中使用该原理图宏。通过它您可以使用预配置设置而不需要担心组件的特殊配置。一个原理图宏通常用于实现具有复杂配置的组件。一般情况下，不会将它作为包含了多种组件的大型设计模板。更多有关实现原理图宏的信息，请参考[第 37 页上的创建原理图宏](#)。

基于 UDB 的设计

对于基于 **UDB** 的设计，请使用 **UDB** 编辑器或使用 **Verilog** 和数据路径配置工具。这两种方法都允许访问 **UDB** 元素。根据不同的情况，您的优选方法也有所不同。

- 通过使用 **UDB** 编辑器，能够以图形形式说明 **UDB** 中的数字功能。它不要求您了解 **Verilog** 或数据路径配置工具。因此，与 **Verilog** 方法相比，该方法可能更简单。**UDB** 编辑器通过在设计图纸上指定 **UDB** 模块中的参数来管理多项内部配置的详细信息。这些图形配置用于实时生成 **Verilog** 代码，也可用于观察各模块是如何被转换成代码的。但使用 **UDB** 编辑器的不足是该方法不支持 **UDB** 的某些高级特性。对于了解 **Verilog** 的用户，使用 **Verilog** 方法可以加快设计状态机的进程。请参见[第 45 页上的使用 UDB 编辑器实现](#)。

- 使用 Verilog 实现 UDB 组件是最通用也是最复杂的方法，因此，不建议初学者使用该方法。通过 Verilog，您可以设计比 UDB 编辑器更精确的状态机。另外，熟悉数字逻辑和 Verilog 的用户会更容易使用该方法。为了访问数据路径，您还需要使用数据配置工具（该工具提供了数据路径的所有功能）。这样，您可以使用 UDB 的所有性能来创建高效且复杂的逻辑。请参见[第 61 页上的使用 Verilog 实现硬件](#)。

软件

也只能通过软件实现。使用软件实现组件意味着设计中没有使用硬件。例如，在用作接口的组件中使用该方法，用以将某些代码链接在一起。更多信息，请参考[第 74 页上的使用软件实现组件](#)的内容。

排除组件

如果您的组件仅支持某些 PSoC 架构、器件系列或器件，那么也可以明确声明该组件仅支持这些特殊器件。更多有关排除组件的详细信息，请参考[第 75 页上的移除某个组件](#)。

实现优先级

请注意，对于使用硬件如原理图、UDB 编辑器或 Verilog 的实现，PSoC Creator 仅使用了这些硬件中的某一个来实现组件，剩余部分被忽略。这些组件的优先级如下。

1. Verilog — 拥有最高优先级。Creator 忽略了组件工作区中的 UDB 编辑器或原理图。
2. UDB 编辑器 — 比原理图的优先级更高，但如果组件工作区中有一个 Verilog 文件，它将被忽略。
3. 原理图 — 优先级最低，如果工作区中有 Verilog 或 UDB 编辑器文件，它将被忽略。

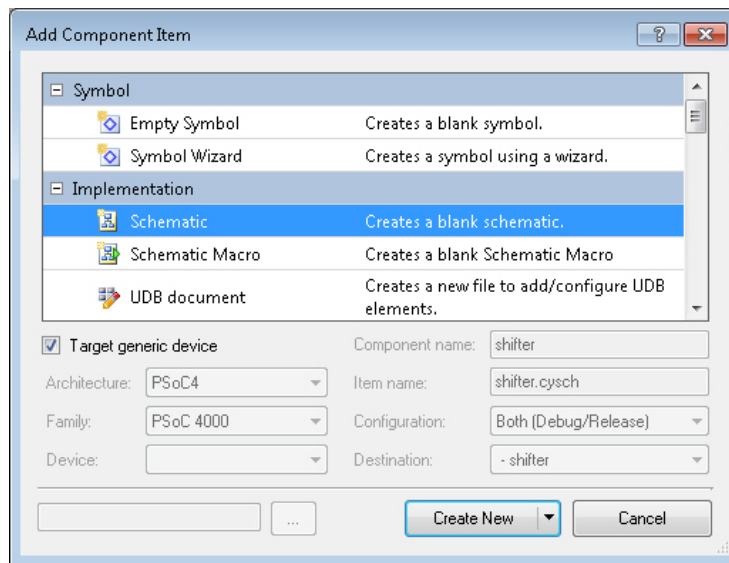
4.1 使用原理图实现组件

原理图是表示组件内部连接情况的图形框图。当您完成创建组件后，请在一个设计中进行初始化，然后对其进行编译（参考第 105 页上的完成创建组件），这样将生成所要文件来提供适用于 Warp 和滤波器的信息。

4.1.1 添加一个原理图

要想将一个原理图文件添加到您的组件中，请进行以下步骤：

1. 右击组件，然后选择 **Add Component Item**（添加组件项）。这时会出现 **Add Component Item** 对话框。



2. 选择原理图图标。
注意：该原理图会继续使用组件名称。
3. 在 **Target** 下面，使用下拉菜单或者 **Generic Device**（通用器件）来指定架构、器件系列和 / 或器件。
原理图可用于特定的架构、器件系列或器件，或适用于所有器件。因此，可以选择各 **Target** 项。
4. 请点击 **Create New**（创建新项）。

在 **Workspace Explorer**（工作区浏览器）树中，根据所指定的器件选项，原理图文件（.cysch）将出现在相应的目录中。

原理图编辑器将打开 .cysch 文件，这时您可以绘制代表组件连接的框图。

4.1.2 完成原理图

更多有关使用原理图编辑器的详细信息，请参考 PSoC Creator 帮助部分的 “使用设计输入工具 > 原理图编辑器” 一节。

4.1.2.1 设计范围资源 (DWR) 设置

DWR 设置被绑定到 DWR 组件实例中（时钟、中断、DMA 等）。因此在您的组件中，这些组件的命名规格可确保它们始终正确进行映射。

- 这些组件名称与自动生成的内部 ID 相关。
- 它们由层次实例名称组成。

如果您创建了一个包含 DWR 的组件，您可以随便重新对它命名。内部 ID 用于维持该组合。

如果您清除了组件中的 DWR，那么系统将使用分层实例名称。如果您这时添加了一个新的 DWR 组件，并给它使用了另外一个名称，那么 .cydwr 文件将丢失与用户的 DWR 组件相关联的所有设置。

4.2 创建原理图宏

原理图宏是一个微型的原理图，通过它您可以使用多个元素（如现有组件、引脚、时钟等）来实现组件。一个组件可以拥有多个宏。宏可以含有多个实例（包括用于该宏的组件）、终端和连线。

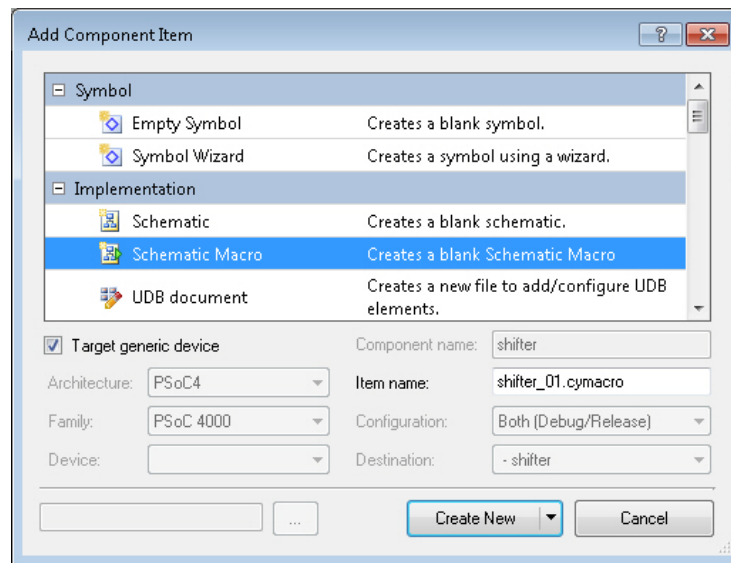
通常创建原理图宏，用于简化组件的使用情况。在原理图中，组件的典型用法得到准备并通过各个宏实现。最终用户会使用这些宏而不是标记符号。

4.2.1 添加一个原理图宏文档

将一个原理图宏添加到组件内与添加其他组件项的方法相同。要想将一个原理图宏文件添加到您的组件中，请进行以下步骤：

1. 右击组件，然后选择 **Add Component Item**（添加组件项）。

这时会出现 **Add Component Item** 对话框。



2. 选择原理图宏图标。
3. 在 **Target** 下面，使用下拉菜单或者选用 **Generic Device**（通用器件）来指定架构、器件系列和器件。
原理图宏“所属”组件。这些宏的等级可以是：架构级、器件级、器件系列级和普通级。有多个等级相同的宏。该特点与组件中其他表示方式（原理图、Verilog）不同。在一个组件中各种宏文件名称是唯一的。
4. 点击 **Create New**（创建新项）。

在工作区浏览器中，根据所指定的器件选项，将在相应的目录中列出原理图宏文件（.cymacro）。

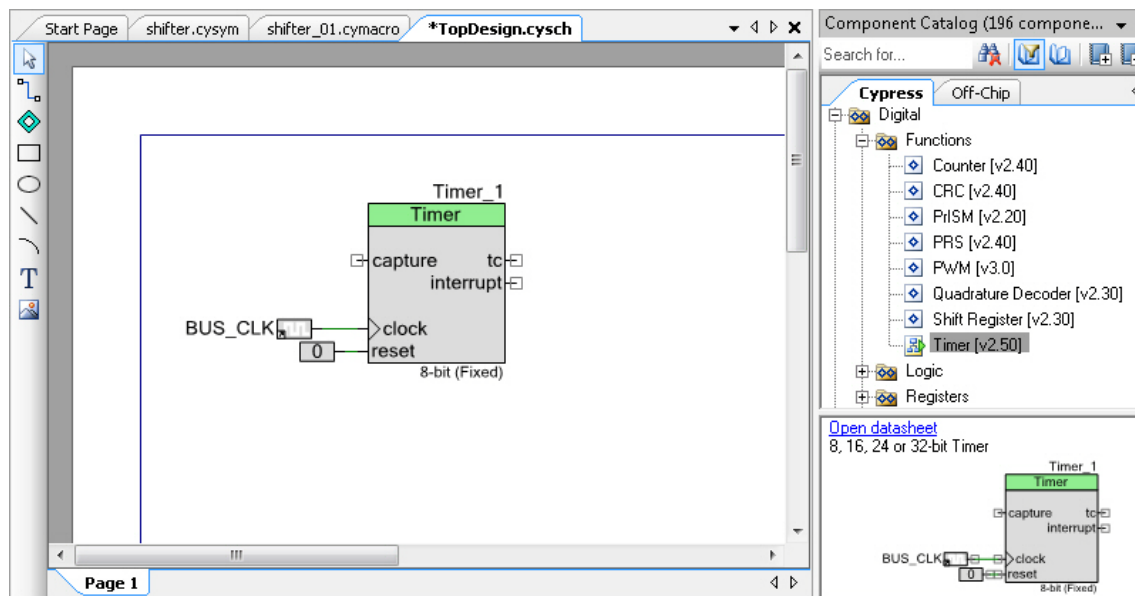
原理图宏编辑器打开了 .cymacro 文件。您可以添加组件，定义预定义设置（如默认时钟频率）以及覆盖每个组件的默认设置等。您也可以使用宏来预定义通信接口或模拟组件所需要的 I/O 引脚配置。

4.2.2 定义宏

下面显示的是创建定时器宏的一个示例：

1. 放置目录中的定时器组件。
若需要，您可以设置该组件的参数，以覆盖定时器的默认值。有关可用的各种参数，请参考组件数据手册。
2. 放置库中的某个时钟组件，并对其进行相应配置。
3. 将时钟输出连接至定时器组件的“时钟”输入终端上。
4. 将“逻辑低”组件放置在原理图上，并将其连接到定时器的“复位”输入端。
5. 右击原理图宏并从下拉菜单中选择 **Properties**（属性）。
根据需要，对组件目录放置情况和摘要进行相应设置。
6. 打开定时器的符号文件，在空白空间任意位置右击，并从下拉菜单中选择 **Properties**（属性）项。通过进行该设置从组件目录中隐藏该组件。

在一个设计项目中，您会看到以下图片：



7. 一旦您按照图示放置您的设计，请尝试清除逻辑低或更改时钟或更改定时器设置等。

4.2.3 版本管理

由于这些宏属于组件（请参考上述内容），因此它们通过组件版本化完成自己的版本化操作。该宏的名称不包含宏或组件的版本编号。

4.2.4 组件更新工具

在原理图上，通过使用组件更新工具可以更新组件的各个实例。当在原理图上放置了一个宏时，放置元素的“宏特性”将消失。现在，该宏的独立部分是单独的原理图元素。由于原理图上没有宏的“实例”，因此不需要对组件更新工具进行更新。

然而，原理图宏本身被定义为一个原理图。它可能包含了能够通过组件更新工具进行更新的其他组件实例。该宏定义原理图对组件更新工具是可见的，并且可以进行所有合适的更新。

4.2.5 宏文件命名规则

宏文件的扩展名为 “.cymacro”。默认的文件名称为 <component name>_<[0-9][0-9]>.cymacro。

4.2.5.1 宏和符号拥有相同的名称

我们不建议组件创建人员使用与符号相同的名称和放置目录，因为这样做会使终端用户混淆。

4.2.6 文档属性

原理图宏拥有许多与符号相同的属性。相关详细信息，请参见第 28 页上的指定文档属性。

4.2.6.1 组件目录放置

宏的目录放置属性与符号放置目录相同，但不具有与放置相关的参数。目录放置编辑器允许将该宏放置在组件目录中多个位置上。

原理图宏列出在组件目录中，同各个符号在目录中显示的方式一样。如果一个宏的显示名称在另一个组件中重复出现，那么在该名称的括号中会存在有一个索引，以表示有多个宏实例使用了相同的显示名称。

4.2.6.2 摘要文本

在组件目录中选择符号时，PSoC Creator 允许这些符号拥有显示在组件预览区内的相应“摘要文本”。各个宏将支持相同的特性。摘要文本字段在原理图宏编辑器中可用，当在组件目录中选择宏时，该文本将显示在预览区内（与符号的处理方式相同）。

4.2.6.3 隐藏属性

各个宏支持隐藏属性，同各个符号支持该属性的方法一样。在宏编辑器中，可以对该属性进行编辑。

4.2.7 宏数据手册

原理图宏没有单独的数据手册。但在该组件数据手册中的单独部分会为已给组件定义的所有宏进行说明。在组件目录中选择组件后，宏的预览区将显示到组件数据手册的链接。

4.2.8 宏的后期处理

将宏拖放到原理图上时，需要重新检查各个实例、终端和连线。在后期处理结束时，实例、终端和连线的名称不能与原理图中现有的名称产生冲突。

在添加到模型前，PSoC Creator 必须为宏中的每个实例、终端和连线分配一个非冲突名称。将每个实例、终端和连线添加到该原理图内，以确保用于该项目的 HDL 名称不被该原理图使用。这些项目的基本名称必须带有数字后缀，并一直递增，直到独立的 HDL 名称可用为止。

4.2.9 示例

假设某个宏带有一个实例（名为“i2c_master”）、三个终端（名为“sclk”、“sdata”和“clk”）以及三个连线（连线不被命名；它们的有效名称分别为 sclk、sdata 和 clk）。如果该原理图为空，并且宏被拖放在原理图上，那么各实例、终端和连线的名称如下：

- 实例：i2c_master

- 终端: sclk、sdata 和 clk
- 连线: sclk、sdata 和 clk

如果该宏被再次拖放, 那么这些名称显示如下:

- 实例: i2c_master_1
- 终端: sclk_1、sdata_1 和 clk_1
- 连线: sclk_1、sdata_1 和 clk_1

如果这些名称已被使用, 那么它的后缀将一直递增, 直到有可用的有效名称为止。

4.3 实现 UDB 组件

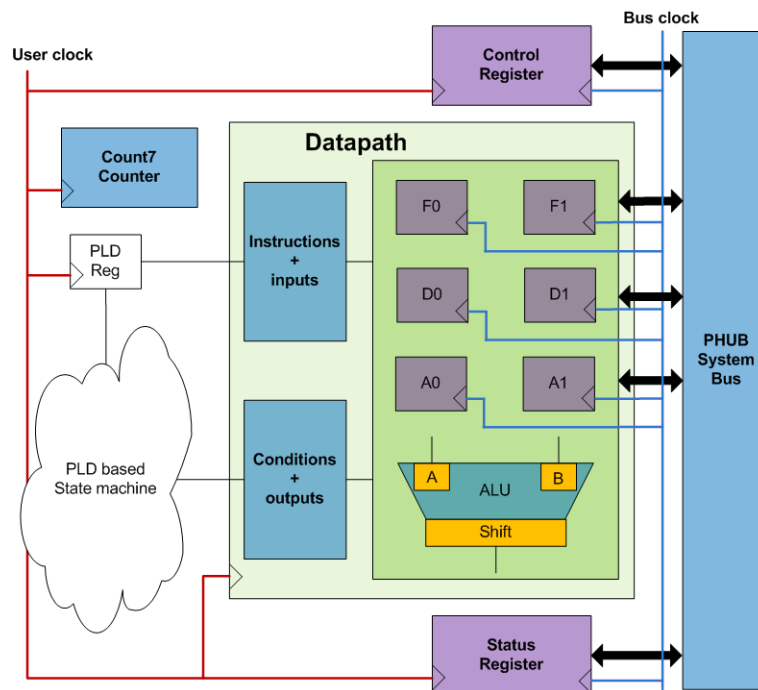
为了充分使用 UDB 中的资源，需要使用 UDB 编辑器或 Verilog 和数据路径配置工具来设计 UDB 组件。每个方法都有其自身的优点和缺点，但 Verilog 方法通常比 UDB 编辑器方法更高级。在本节中，我们说明了 UDB 的基础架构并列出了可用资源。然后，我们介绍了如何使用 UDB 编辑器和 Verilog 进行设计。

4.3.1 UDB 硬件的简介

UDB 是一个组合，包括未赋值的 PLD、结构处理器模型（数据路径）、控制和状态寄存器和一个 7 位计数器（count7）。通过灵活布线将这些元素连接起来。可以使用这些元素构成多种逻辑类型，也可以将这些元素连锁在一起构成一个大型设计。一个设计能够与 CPU、PSoC 器件中的其他硬件模块或两者进行通信。通过该灵活性，可将 UDB 构成一个逻辑单元，它可以链接您设计中的其他硬件或作为执行新功能的独立模块使用。

4.3.1.1 UDB 概况

下图显示的是 UDB 的高级框图。图中加亮显示了 UDB 中的主要模块以及它们的连接和控制方式。内部信号以及单独输入和输出不被显示。这些模块按照颜色译码，用于同其他类型区分开。紫红色表示一个寄存器，蓝色代表一个执行定义函数的固定模块，绿色表示数据路径，橙色表示的是算术逻辑单元（ALU）和移位器的输入和输出。白色为 PLD 逻辑。更多有关 UDB 架构的详细信息，请参考 TRM。



通过使用用户时钟和总线时钟可以驱动 UDB。总线时钟对数据路径中的寄存器以及控制 / 状态寄存器进行读写操作。这些数据字经过整个 PHUB 系统总线。用户时钟驱动 UDB 中的各个模块。通过布线 UDB 中的信号可以构成组件中的硬件输出，也可以使用这些信号驱动 UDB 中各结构模块的输入。

- **PLD** — 通常使用它来创建逻辑，以控制 UDB 中的其他结构资源。基于 PLD 的设计由组合逻辑和连续逻辑组成。连续逻辑单元通过用户时钟驱动。PLD 不仅是 UDB 中的最灵活的元素，它还是资源密集型器件。因此，当实现大型设计时，建议尽可能使用其他结构模块。
- **数据路径** — 数据路径是一个 8 位宽的处理器，用于对数据字进行简单的算术和按位运算。可将其链接构成 16、24 和 32 位宽的处理器。它可以拥有 8 个用户定义指令，通过使用 PLD 实现状态机可以驱动这些指令。数据路径构成了多种 UDB 设计的核心功能。当使用 8 位字或更宽的处理器时，（与 PLD 设计相比）应该使用这种设计。更多有关数据路径的信息，请参考[第 42 页上的数据路径操作](#)。
- **控制寄存器** — 用于在 UDB 和 CPU 间进行通信的控制寄存器。当使用控制寄存器时，CPU 可以直接将指令发送给 UDB 硬件。对 CPU 控制寄存器进行读和写操作都是在总线时钟上执行的，除非它处于使用用户定义时钟的同步或脉冲模式中。
- **状态寄存器** — 状态寄存器用于向 CPU 通知硬件信号的状态。CPU 读取状态寄存器的速率由总线时钟控制，而 UDB 写入该寄存器的过程则由用户时钟控制。状态寄存器也可以生成可屏蔽中断。要想实现该操作，需要将一个引脚用于中断输出并将其他 7 位用作该中断的可屏蔽触发器。
- **Count7 计数器** — UDB 包含了一个 7 位计数器，可以使用它来替代 PLD 或数据路径的计数器实现。如果所需要的计数器是 4 至 7 位的，这样做可以节省资源。这样，可以在您的整个设计中使用该计数器的终端计数。

4.3.1.2 数据路径操作

基于 UDB 的 PSoC 器件路径基本上是一个非常小的 8 位宽处理器，该处理器具有“动态配置”中定义的 8 种状态。可以将连续数据路径连接在一起，以便使用以下任意预定义模块使用更宽的数据带宽运行。以下内容详细说明了该数据路径以及它的用法。有关该数据路径的详细信息，请参考[技术参考手册 \(TRM\)](#)。

数据路径指令

数据路径分以下几个部分：

- **ALU** — ALU 能够在 8 位数据上执行以下操作。将多个数据路径互连起来构成 16、24 和 32 位时，各运算操作将在整个数据宽度上执行。
 - 通过 (Pass-Through)
 - 递增 (INC)
 - 递减 (DEC)
 - 加 (ADD)
 - 减 (SUB)
 - XOR
 - AND
 - OR
- **移位** — ALU 的输出被传输给能够执行下面操作的移位运算符。将多个数据路径互连起来构成 16、24 和 32 位时，各运算操作将在整个数据宽度上执行。
 - 通过 (Pass-Through)
 - 向左移位
 - 向右移位
 - 半字节交换
- **掩码** — 将移位运算符的输出传递给某个掩码运算符，这样能够屏蔽数据路径的全部 8 位。

- 寄存器 — 数据路径具有下面的寄存器；这些寄存器可用于硬件和 CPU，并具有在静态配置寄存器中所定义的各种配置选项。
 - 两个累加器寄存器：ACC0 和 ACC1
 - 两个数据寄存器：DATA0 和 DATA1
 - 两个大小为 4 字节的 FIFO：FIFO0 和 FIFO1 都能在多种模式下运行
- 比较运算符
 - 零检测：Z0 和 Z1 分别将 ACC0 和 ACC1 与零进行比较，并为互连逻辑输出二进制的真/假值，以便在需要时提供给硬件使用。
 - FF 检测：FF0 和 FF1 分别将 ACC0 和 ACC1 与 0xFF 进行比较，并为互连逻辑输出二进制的真/假值，以便在需要时提供给硬件使用。
 - 比较 0：
 - 相等比较（ce0）— 比较值（ACC0 和 Cmask0）等于 DATA0，并为互连逻辑生成二进制的真/假值，以便在需要时提供给硬件使用。（在静态配置寄存器中可以配置 Cmask0）。
 - 小于比较（cl0）— 比较值（ACC0 和 Cmask0）小于 DATA0，并为互连逻辑生成二进制的真/假值，以便在需要时提供给硬件使用。（在静态配置寄存器中可以配置 Cmask0）。
 - 比较 1：
 - 相等比较（ce1）— 比较值（（ACC0 或 ACC1）和 Cmask1）等于（DATA1 或 ACC0），并为互连逻辑生成二进制的真/假值，以便在需要时提供给硬件使用。（在静态配置寄存器中可以配置 Cmask1）。
 - 小于比较（cl1）— 比较值（ACC0 和 Cmask0）小于 DATA0，并为互连逻辑生成二进制的真/假值，以便在需要时提供给硬件使用。（在静态配置寄存器中可以配置 Cmask1）。
 - 溢出检测：表示通过将 ov_msb 输出作为二进制真/假值驱动到互连逻辑（以便需要时供硬件使用），可以溢出 msb。

通过数据路径，可以对大部分组件中常用的不同配置进行设计。可以通过将 Verilog 代码编程到 PLD 中实现数据路径内的多项功能。然而，PLD 会很快被用完，但数据路径则是固定的模块。总是需要在数据路径和有效的 PLD 数量之间进行权衡。设计者需要决定其中哪个是更珍贵的资源。请注意，某些功能（比如 FIFO）不能在 PLD 中实现。

数据路径寄存器

每个数据路径都包含了 6 个寄存器：A0、A1、D0、D1、F0 和 F1。这些寄存器具有多种功能和一定的限制，并可以通过多种方法使用这些寄存器进行设计。

- 累加器寄存器 A0 和 A1 通常作为 RAM 使用，这样可以保存临时输入输出 ALU 的值。这些都是最通用且访问最多的寄存器。
- 数据寄存器 D0 和 D1 不像累加器寄存器那样灵活。只有使用来自 FIFO 的数据和额外的 d0_load 或 d1_load 输入信号才能对这两个寄存器进行写操作。因此，在设计中通常将它们作为 ROM 使用。
- 通常将 4 字 FIFO F0 和 F1 作为数据路径的输入/输出缓冲器使用。这些 FIFO 不可直接给 ALU 提供数据，并且在 ALU 使用 FIFO 中的值前必须将该值加载到累加器寄存器中。更多有关 FIFO 配置的详细信息，请参考第 44 页上的 FIFO 模式。

数据路径输入 / 输出

不管数据宽度如何，数据路径最多可以包含 6 个输入位。在这 6 个输入位中，最多可以使用 3 个位在此时钟周期内控制数据路径指令。因此，在设计中只能使用 8 条数据路径指令。每一条指令能在同一个时钟周期内执行多项操作；这样可以进一步提高性能。

同样，不管数据的宽度如何，数据路径最多可包含 6 个输出位。通过使用这些输出位，可以将数据路径的状态信号发送到一个状态寄存器或设计中的其他模块（比如：状态机或 count7 计数器）内。这些状态信号是由数据路径中的比较和内部逻辑操作生成的，并且它不包括直接来自寄存器或 ALU 的数据位。但通过通过使用移位器连续将 ALU 中的位移出访问该信息。

FIFO 模式

可以使用辅助的控制寄存器将数据路径中的 4 字 FIFO 配置为几种模式。CPU/DMA 使用该寄存器，以便动态控制中断、计数器和 FIFO 操作。更多有关辅助控制寄存器的详细信息，请参考技术参考手册（TRM）。

可以将 FIFO 设置为单缓冲器模式或正常模式。

- **单缓冲器模式** — 该模式会将 FIFO 配置为 1 字的缓冲器，而不会将它配置为正常的 4 字 FIFO。被写入到 FIFO 中的所有值会立即覆盖掉 FIFO 的内容。如果仅需要一个 1 字寄存器的 FIFO 及其相应的 FIFO 总线状态信号和模块状态信号，那么可以使用这种模式。
- **正常模式** — 正常模式是用来填充最多 4 个数据字的标准 4 字 FIFO。

通过使用 FIFO 总线状态信号和模块状态信号，可以控制传入 / 传出 FIFO 的数据。这些是 FIFO 0/1 模块状态（f0_block_stat、f1_block_stat）信号和 FIFO 0/1 总线状态（f0_bus_stat、f1_bus_stat）信号。这些信号的行为取决于输入 / 输出模式和辅助的控制寄存器的设置情况。下表显示了可能使用的配置信息。

注意：该表仅供参考。更多有关 FIFO 配置的详细说明，请参考 TRM。

方向	级别模式	信号	状态	说明
输入	N/A	模块状态	空白	FIFO 中没有任何字节时被确认。
	正常速度	总线状态	未满	在 FIFO 中至少有一个字节大小的空间时将被确认。
	MID	总线状态	至少有一半为空	在 FIFO 中至少有两个字节的空间时将被确认。
输出	N/A	模块状态	已满	FIFO 已满时将被确认。
	正常速度	总线状态	非空	在 FIFO 中至少可以读取一个字节时将被确认。
	MID	总线状态	至少一半已满	在 FIFO 中至少可以读取两个字节时将被确认。

通过将辅助控制寄存器的 FIFO 级别模式设置为 NORMAL 或 MID，可以配置级别模式。

- **NORMAL FIFO 级别** — 准备读取或写入至少 1 字的数据（取决于 FIFO 方向）时，NORMAL FIFO 级别会允许总线状态信号被激活。
- **MID FIFO 级别** — 当准备读取或写入至少 2 字的数据（取决于 FIFO 方向）时，MID FIFO 级别会允许总线状态信号被激活。

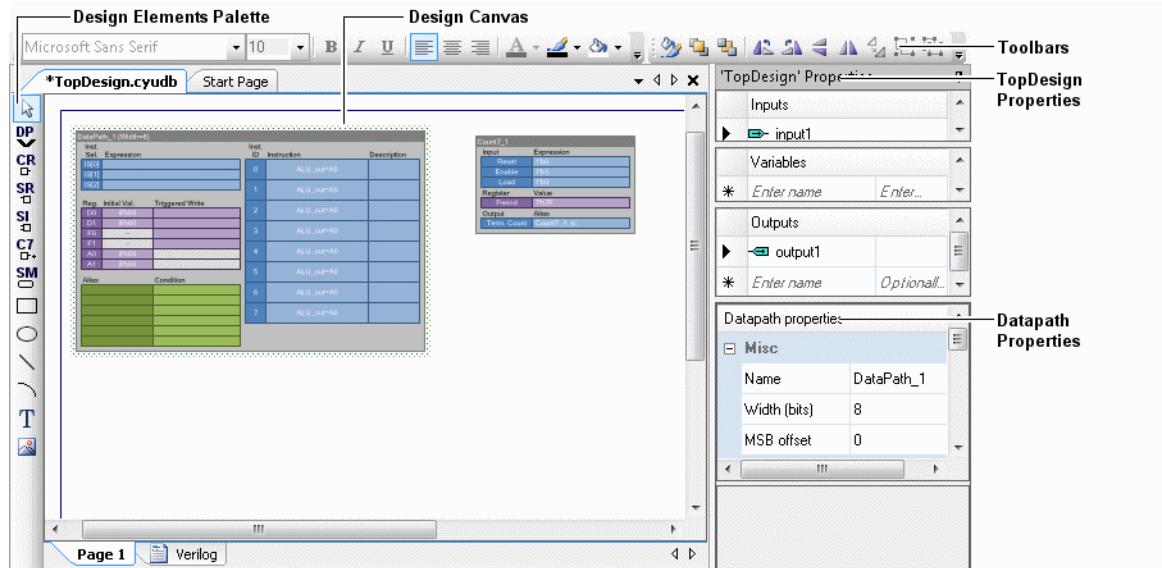
4.3.2 使用 UDB 编辑器实现

UDB 编辑器是一个图形化工具，用来构建基于 UDB 的设计，而不需要编写 Verilog 或不要使用更高级的数据路径配置工具。通过该方法可以访问 UDB 中的各种元素，包括：数据路径、控制寄存器、状态寄存器、状态中断寄存器、count7 计数器和 PLD；这些元素都以图形形式表示。因为 UDB 编辑器是一个图形化工具，所以它对 Verilog 知识和 UDB 的复杂细节要求较少。但这样会降低某些灵活和硬件的粒度控制，这是简单化抽象性的结果。它也不具有某些更高级的 UDB 功能，因此限制将其使用于复杂的设计中。

有关 UDB 元素的详细信息，请参见第 41 页上的 [UDB 硬件的简介](#)。本节介绍了这些 UDB 元素的目的是描述 UDB 编辑器中子模块的内部运行，并说明如何将子模块组合在一起，以便形成一个功能设计。

4.3.2.1 UDB 编辑器的说明

使用 UDB 编辑器，您在设计基于 UDB 的硬件时，使用较少的数字逻辑或 Verilog 的知识。对 UDB Editor 进行设计，这样您可以进行拖放，然后配置您的硬件，而无需编写代码。然后，该工具会将您的设计实时转换为 Verilog — 为您提供一次看到 UDB 模块如何转换为 HDL 的机会。



UDB 编辑器的结构如下：

- **Pages（页）** — 一旦您打开 UDB 编辑器文档，您会看到一个与图像页相同的可编辑页。这是您的设计图纸，用于放置和配置 UDB 元素。通过在 **Page 1** 选项卡中添加更多页，UDB 编辑器会为您提供页数量。然后，这些页互相转换（跟进行一个设计相同）。

- **Verilog** — 在 **Page** 选项卡的旁边，您可以找到 **Verilog** 选项卡。这是设计中被转换的硬件的只读视图。动态更新 **Verilog** 选项卡，以便在设计中发生变化时，它也会更新代码。该代码是不可编辑或删除的，所以您要在设计图纸中进行适当的改变，才能看到想要的结果。如果想要使用 **Verilog** 来编辑设计，您可以将该代码复制并粘贴到 **Verilog** 文件内。有关 **Verilog** 设计的更多信息，请参考 [第 61 页上的使用 Verilog 实现硬件](#)。
- **Design Properties**（设计属性）— 在设计图纸的右侧，您可以找到各种设计属性，通过这些属性您可以配置输入、输出以及设计中所使用的变量。设计完成时，输入和输出会变成硬件端口。当设计中含有数据路径时，设计属性窗口也适用于设置全局数据路径配置。
- **Design Elements Palette**（设计元素控制板）— 位于设计图纸的左侧。它是一个用于选择您设计中包含的 **UDB** 元素的菜单。这些 **UDB** 元素指的是数据路径（**DP**）、控制寄存器（**CR**）、状态寄存器（**SR**）、状态中断寄存器（**SI**）、**count7** 计数器（**C7**）以及状态机状态（**SM**）。有关 **UDB** 元素和 **UDB** 架构的详细信息，请参考 [第 41 页上的 UDB 硬件的简介](#)。

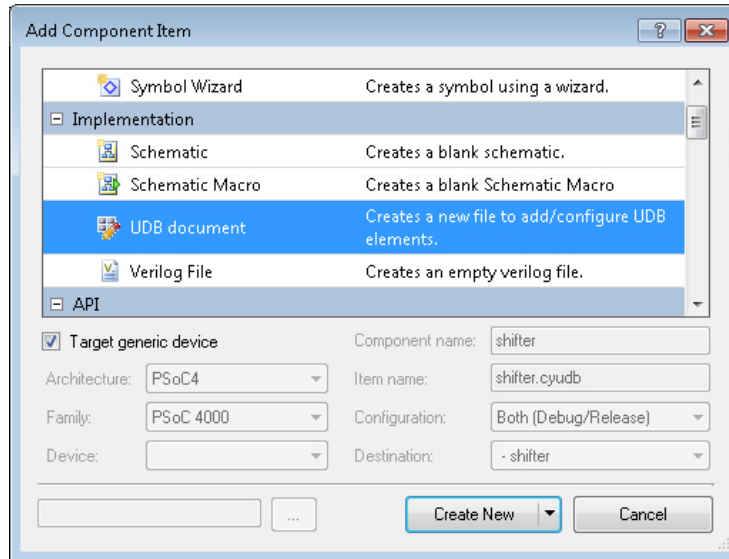
可以将 **UDB** 分为自由逻辑和结构化逻辑。结构化逻辑包含数据路径、控制寄存器、状态寄存器（状态中断寄存器（如果被使能））以及 **Count7** 计数器。通过控制寄存器和状态寄存器，**CPU** 可以控制这些结构化逻辑。另外，通过使用自由 **PLD** 逻辑可以设计状态机，该状态机可以生成这些模块的控制信号。

使用设计元素控制板中的 **SM** 模块可以构成状态机。这些模块通过状态转换被连接在一起，从而能够驱动设计中所使用的各种信号。有关创建状态机的更多信息，请参考 [第 57 页上的状态机](#)。

4.3.2.2 添加 UDB 文档

如何将一个 UDB 文档添加到您的组件：

1. 右键点击该组件，然后选择 **Add Component Item**（添加组件项）项。
这时会出现 **Add Component Item**（添加组件项）对话框。



2. 选择 UDB 文档的图标。
注意： UDB 文件会继承使用组件名称。
3. 在 **Target** 下面，使用下拉菜单或者选用 **Generic Device**（通用器件）来指定架构、器件系列和 / 或器件。
原理图可用于特定的架构、器件系列或器件，或适用于所有器件。因此，可以选择 **Target** 各项。
4. 点击 **Create New**。

在 **components** 选项卡下的工作区浏览器树中，根据所指定的器件选项，UDB 文档文件（.cyudb）将排列在相应目录中。使用 UDB 编辑器打开 '.cyudb' 文件，然后可以开始实现组件。

注意： 如果组件包含了一个 Verilog 文件和一个 UDB 编辑器文档，那么 UDB 编辑器文档会被忽略。如果组件包含一个 UDB 编辑器文档和一个原理图，则原理图会被忽略。

4.3.2.3 完成 UDB 文档

双击工作区中的 UDB 编辑器，打开它。使用设计元素控制板，拖动 UDB 元素，将其放置在设计图纸中。使用设计属性窗口确定组件的输入、输出以及设计中所使用的变量。当数据路径被选中时，该窗口也会用于为特定的数据路径实例设置全局配置。

UDB 编辑器至少包含两个选项卡。它们分别是设计图纸和结构化设计生成的 Verilog。**Verilog** 选项卡是只读的。如果需要编辑 Verilog，那么可以将代码复制并粘贴到 Verilog 文件内。这是从设计图纸中实时更新的功能，通过它您可以了解如何将设计映射到 Verilog 而不要进行构建。通过了解如何将每个模块转换为代码，可以将它作为一个用于了解 Verilog 的工具使用。另外，更大的设计有多种设计图纸。结合多种设计图纸可形成单一的 Verilog 输出。

更多有关使用 UDB 编辑器的详细信息，请参考 PSoC Creator 帮助一节中的“使用设计输入工具 > UDB 编辑器”。

注意：使用 UDB 编辑器实现的设计始终需要一个时钟终端。请注意，所有控制、输入、输出、变量、状态、数据路径、控制寄存器、状态寄存器以及 count7 计数器的名称在整个 UDB 编辑器文档中必须是唯一的。

4.3.2.4 版本和组件更新工具

使用 UDB 编辑器实现的设计使用子模块组件（状态寄存器、状态中断寄存器、控制寄存器以及 count7 计数器）的最新版本。如果旧设计使用的是这些模块的先前版本来实现的，则设计可以通过使用组件更新工具将其更新为当前版本。请注意，数据路径和状态机不会记录版本，因此不需要更新组件版本。

4.3.2.5 UDB 编辑器语法

UDB 编辑器使用 Verilog 表达方式来描述信号操作。其中一些 Verilog 表达方式类似于 C 表达方式。下表显示的是受支持的运算符。这些运算符可以使用于 UDB 元素的表达式字段中。通过使用 PLD 可以实现逻辑操作。

类别	表达式	说明	示例
算术运算符	*	乘法	A * B
	+	加法	A + B
	-	减法	A - B
	/	除法	A / B
	%	模算术	A % B
移位运算符	<<	向左移位	A << 1
	>>	向右移位	A >> 1
关系运算符	<	小于	A < B
	>	大于	A > B
	<=	小于或等于	A <= B
	>=	大于或等于	A >= B
等式运算符	==	等于	A == B
	!=	不等于	A != B
按位运算符	~	NOT（按位进行 NOT 运算）	~A
	&	AND（按位进行 AND 运算）	A & B
		OR（按位进行 OR 运算）	A B
	^	XOR（按位进行 XOR 运算）	A ^ B
	^~	XNOR（按位进行 XNOR 运算）	A ^~ B
	~^	XNOR（按位进行 XNOR 运算）	A ~^ B

类别	表达式	说明	示例
缩减运算符	&	AND（缩减一元 AND）	& A
	~&	NAND（缩减一元 NAND）	~& A
		OR（缩减一元 OR）	A
	~	NOR（缩减一元 NOR）	~ A
	^	XOR（缩减一元 XOR）	^ A
	^~	XNOR（缩减一元 XNOR）	^~ A
	~^	XNOR（缩减一元 XNOR）	~^ A
逻辑运算符	!	NOT（逻辑 NOT）	! A
	&&	AND（逻辑 AND）	A && B
		OR（逻辑 OR）	A B
条件运算符	?:	类似于 C 语言中的三元运算符。	(A) ? 1'b1 : 1'b0
串联	{ }	串联位	{ A, B }

4.3.2.6 UDB 元素

UDB 编辑器中有效的子模块或 UDB 元素包含了数据路径、控制寄存器、状态寄存器、状态寄存器中断以及 count7 计数器。通过使用 UDB 编辑器的 FSM 状态图能够访问 PLD。全部这些元素都存在于设计元素控制板中。

Datapath（数据路径）

数据路径是具有高达八个控制状态的可编程 8 位处理器，可将这些控制状态链接在一起，构成更大的 16/24/32 位处理器。数据路径可以用于执行移位和基本的算术运算符，并形成基于 UDB 的许多设计中的核心元素。每个 UDB 模块包含一个 8 位的数据路径，因此想链接这些数据路径或使用多数据路径会占用多个 UDB 模块。使用 UDB 编辑器进行设计会允许自动链接这些数据路径，因此除了选择数据宽度外，您不需要执行任意操作。

一个数据路径包含六个寄存器：A0、A1、D0、D1、F0 以及 F1。有关这些寄存器的详细信息，请参考第 43 页上的数据路径寄存器。使用 UDB 编辑器时，根据数据路径输入配置，FIFO 的输入 / 输出方向自动被检测。如果其中一个输入被设置为 FIFO 的负载触发器，那么该 FIFO 会被配置为输出。否则，它被设置为输入。

这六个有效输入位，其中最多有三个位可以用于控制该时钟周期的数据路径指令。如果未驱动全部三个指令位（意味着，不使用全部八个有效指令），那么会驱动未被使用的位，以得到 1'b0。有关有效指令的更多信息，请参考第 42 页上的数据路径指令。

下图显示的是在 UDB 编辑器的设计图纸中数据路径元素的实例。该数据路径元素包含了以蓝色显示的六个输入和六个输出、以紫色显示的六个寄存器以及以绿色显示的八个指令。

Datapath_1 (Width=8)					
In.	Selection	Expression	Inst. Addr.	Instruction	Comment
0	INSTR_ADDR[0]	1'b0	3'b000	ALUout=(A0)	
1	INSTR_ADDR[1]	1'b0			
2	INSTR_ADDR[2]	1'b0	3'b001	ALUout=(A0)	
3					
4					
5			3'b010	ALUout=(A0)	
Reg.	Load	Initial Value			
A0	Not supported	8'h00	3'b011	ALUout=(A0)	
A1	Not supported	8'h00			
D0	Unused	8'h00	3'b100	ALUout=(A0)	
D1	Unused	8'h00			
F0	Unused	Not supported	3'b101	ALUout=(A0)	
F1	Unused	Not supported			
Out.	Selection	Name			
0			3'b110	ALUout=(A0)	
1			3'b111	ALUout=(A0)	
2					
3					
4					
5					
			In.	Datapath inputs: 6 signals that control datapath instruction selection, shift in, and register loads.	
			Reg.	Datapath registers: 2 accumulators, 2 data registers, and 2 FIFOs.	
			Out.	Datapath outputs: 6 signals that provide access to signals from the datapath to the rest of the component.	
			Instructions	Datapaths support up to 8 pre-configured instructions, like addition and subtraction.	

点击数据路径也可显示 UDB 编辑器窗口中的各数据路径属性。这样能够确定全局数据路径的设置。双击数据路径元素的任意字段会打开一个 **Configure** 对话框，通过该对话框可以对具体区域进行指定配置。

数据路径属性

点击设计图纸中的数据路径可以访问该数据路径实例的全局数据路径属性。数据路径属性窗口会显示在组件属性窗口中。下表中列出了有效的配置属性。

类别	属性	说明
其他配置	名称	数据路径的示例名称。
	宽度（位）	数据路径的位宽度（8/16/24/32）。
	MSB 偏移	选择数据路径中的最高有效位。用于某些函数（这些函数使用了非 8 倍的字）。
移位普通配置	移出	选择将左移出还是右移出路由到数据路径输出端。
	默认移入	将移位配置 A/B 中的移位源设置为 Default （默认）时可确定移入值。
移位配置 A	移位方向	选择移位方向。
	移入源	选择使用已经路由的移入还是默认的移入表达式。
移位配置 B	移位方向	选择移位方向。
	移入源	选择使用已经路由的移入还是默认的移入表达式。

类别	属性	说明
可配置的比较器输入	配置 A	如果在某个指令中选择了比较配置 A ，那么该项则用于选择需要进行的比较类型。比较器 1 在每个指令周期内进行该项比较操作。
	配置 B	如果在某个指令中选择了比较配置 B ，那么该项将用于选择需要进行的比较类型。比较器 1 在每个指令周期内进行该项比较操作。
屏蔽模具	amask	应用于 ALU 输出的屏蔽值。只有勾选了 amask 复选框，才能使用该屏蔽。
	cmask0	应用于比较器 0 的输入端的屏蔽值。只有勾选了 cmask0 复选框，才能使用该屏蔽。
	cmask1	应用于比较器 1 的输入端的屏蔽值。只有勾选了 cmask1 复选框，才能使用该屏蔽。
FIFO	FIFO 同步模式	确定更新 FIFO 状态同步的方法。
	捕获模式	指定是否直接读取累加器寄存器，或是否对 FIFO 执行触发捕获操作。
	边沿模式	指定 FIFO 写触发器是电平敏感还是边沿敏感。
	快速模式	指定 FIFO 捕获的时钟源。可以使用数据路径时钟捕获或通过总线时钟得到更快的性能（但功耗也更高）。

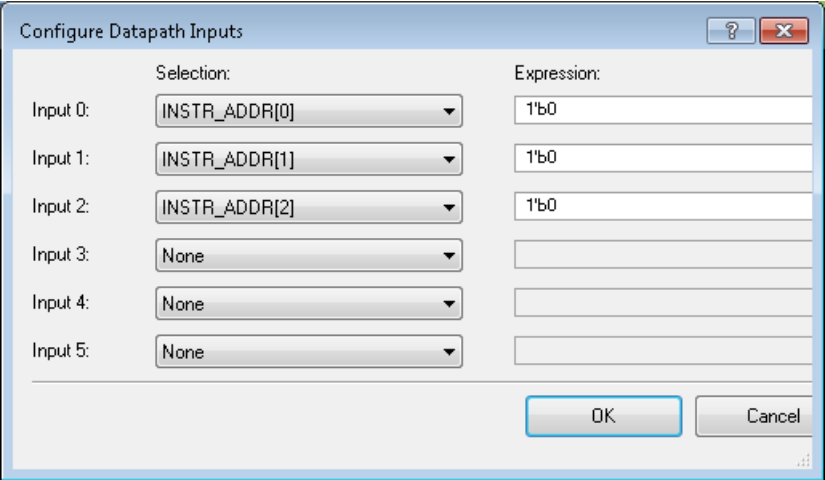
数据路径属性全局地影响着具体的数据路径实例。这些属性包括数据路径的位宽度大小、移位配置、比较器 1 的配置、掩码定义以及 FIFO 模式。一旦设置好了这些属性，那么数据路径的所有指定属性都应遵循这些全局设置。

输入

数据路径的输入用于控制数据寄存器和 FIFO 的负载。也可以将它们使用在链式数据路径中，但 UDB 编辑器会自动执行它们；因此，您不需要手动链接单独的数据路径。输入也使用于定义入移位器的串行输入，并用于控制数据路径中指令的状态。UDB 编辑器中全部有效输入的列表如下。

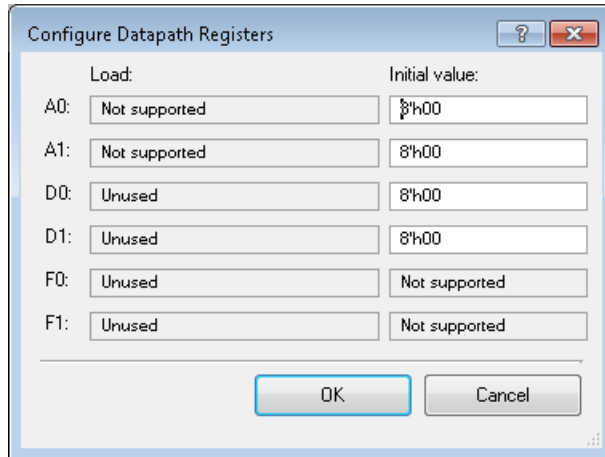
可选值	名称
INSTR_ADDR[0]	指令选择位的最低有效位
INSTR_ADDR[1]	指令选择位的中间位
INSTR_ADDR[2]	指令选择位的最高有效位
将 F0 加载到 D0 内	在时钟的上升沿上将 F0 的第一个元素加载到 D0 内
将 F1 加载到 D1 内	在时钟上升沿上将 F1 的第一个元素加载到 D1 内
将 A0 加载到 F0 内	在时钟上升沿上将 A0 的内容加载到 F0 内
将 A1 加载到 F0 内	在时钟上升沿上将 A1 的内容加载到 F1 内
将 ALUout 加载到 F0 内	在时钟上升沿上将 ALU 输出的内容加载到 F0 内
将 A0 加载到 F1 内	在时钟上升沿上将 A0 的内容加载到 F1 内
将 A1 加载到 F1 内	在时钟上升沿上将 A1 的内容加载到 F1 内
将 ALUout 加载到 F1 内	在时钟上升沿上将 ALU 输出的内容加载到 F1 内
移入	将移位配置 A/B 的移位源设置为 Routed Shift-in（已路由的移入）时所使用的表达式。

六个有效输入中最多有三个可用于决定接下来被执行的数据路径指令。这些输入默认显示在输入部分中 — INSTR_ADDR[2:0]。这些输入的表达式默认为 1'b0。对于被使用的每一个指令位，该位的表达式通常与状态机驱动的信号相应。例如，将在状态机中命名为 myInstr[1:0] 的变量分配到不同的值，以便控制数据路径中的指令。然后，应在其表达式字段中将 INSTR_ADDR[1:0] 分配给 myInstr[1:0]。同样，如果 “Load F0 with A0”（将 A0 加载到 F0）信号可用于数据路径中，并且用来控制该信号的信号被命名为 loadF0，那么 loadF0 应放在表达式字段中，并要求位于 “Load F0 with A0” 输入的旁边。更多有关信息，请参见 [第 57 页上的状态机](#)。



寄存器

双击寄存器会打开 **Configure** 对话框。通过该对话框可以设置启动时寄存器的初始值。如果寄存器不支持当前配置的初始值，那么它会变成灰色。该对话框也会显示每个寄存器的加载表达式（若有）作为参考。这是一个只读字段；必须通过 **Configure Datapath Inputs** 对话框对加载表达式进行编辑。更多有关信息，请参见 [第 43 页上的数据路径寄存器](#)。



The dialog box titled "Configure Datapath Registers" contains two columns: "Load:" and "Initial value:". It lists registers A0, A1, D0, D1, F0, and F1. A0 and A1 are marked "Not supported" in the Load column and have initial values of 8'h00. D0 and D1 are marked "Unused" and have initial values of 8'h00. F0 and F1 are marked "Unused" and are marked "Not supported" in the Initial value column. There are OK and Cancel buttons at the bottom.

Register	Load	Initial value
A0:	Not supported	8'h00
A1:	Not supported	8'h00
D0:	Unused	8'h00
D1:	Unused	8'h00
F0:	Unused	Not supported
F1:	Unused	Not supported

输出

数据路径的输出包含了多个比较器状态值、FIFO 状态值、移位输出以及多种溢出。下表列出了 UDB 编辑器中有效的输出。

可选值	名称
A0 == D0	A0 等于 D0 的状态（由比较器 0 执行）
A0 < D0	A0 小于 D0 的状态（由比较器 0 执行）
A0 == 0	比较 A0 是否等于 0。
A0 == 0xFF	比较 A0 是否等于 0xFF。
Config A: "equal" Config B: "equal"	比较器 1 的相等比较的状态。
Config A: "less than" Config B: "less than"	比较器 1 的小于比较的状态。
A1 == 0	比较 A1 是否等于 0。
A1 == 0xFF	比较 A1 是否等于 0xFF。
溢出	该选项用于监控上溢是否发生在指令周期内。
进位	算术运算的移出。
CRC MSB	CRC 反馈输出
移出	移出位。它既可以向左移出，也可以向右移出。
F0 总线状态（未滿）	FIFO 0 总线状态，用于指出 FIFO 0 是否已滿。
F0 模块状态（非空）	FIFO 0 模块状态，用于指出 FIFO 0 是否为空。
F1 总线状态（未滿）	FIFO 1 总线状态，用于指出 FIFO 1 是否已滿。
F1 模块状态（非空）	FIFO 1 模块状态，用于指出 FIFO 1 是否为空。

数据路径的输出可以用于状态机，以便控制转换（例如，对下一个周期的指令代码进行修改），或直接连接到其他硬件模块使用的输出终端。这些信号还可以被内部路由到其他硬件模块，比如：另一个数据路径、状态寄存器或 **count7** 计数器。请注意，每个数据路径最多可具有六个输出位。有关如何将状态机与数据路径交互的详细信息，请参考[第 57 页上的状态机](#)。

指令

通过双击数据路径实例的指令面板可以单独设置八个有效数据路径指令。每条指令被分为三个部分：ALU 运算、寄存器写操作和比较选项。欲了解运算的完整列表，请参见[第 42 页上的数据路径指令](#)。

ALU 运算用于确定该指令周期所进行的运算符或 Boolean 运算。

- ALU 接收来自两个寄存器源的数据，并对它们进行计算。第一个输入（A）被限制为 A0 或 A1，但第二个输入（B）可以接收 D0、D1，并可以内部提供数值 1 作为一个输入。然后，结果被传输到 ALUout。
- ALUout 值在输出前能够进行 1 位转移。如果对得到的结果进行 1 位转移，那么需要指定移位运算。移位属性由数据路径属性窗口中的全局移位配置控制。
- 完成功能和移位定义后，ALUout 的表达式会被显示出来。使用该表达式检查 ALU 运算是否正确。

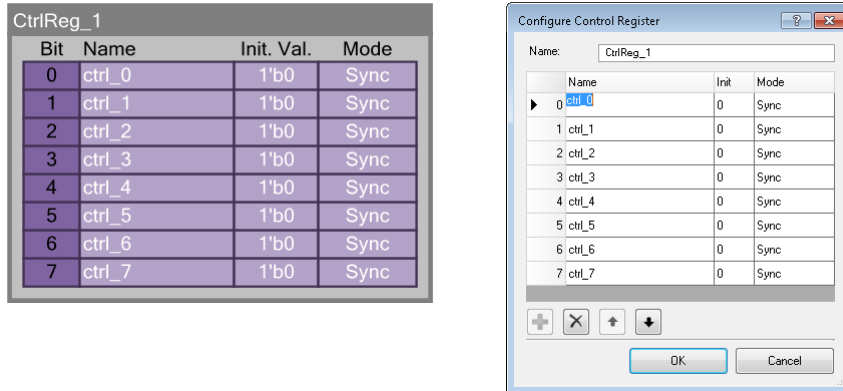
使用寄存器写操作将下一条指令周期所使用的数值加载到 A0 和 A1 内。可将它们作为 ALUout 的反馈，或用于接收数据寄存器或 FIFO 中的新数据。

- A0 可以保持初始值，或由 D0 或 F0 覆盖。它也可以由 ALUout 覆盖，以提供反馈。
- 同样，A1 可以保持初始值，或被 D1 或 F1 覆盖。它也可以被 ALUout 覆盖，以提供反馈。
- 如果一个指令周期完成后没有将 ALUout 保存到某个寄存器内，那么其结果会被下一个指令的结果覆盖。因此，建议将 ALUout 写入到某个累加器，或者使用数据路径输入一节中所提到的“负载 FIFO”信号将 ALUout 加载到 FIFO 内。

比较选项用于设置使用比较器 0 和比较器 1 建立的比较条件。比较器 0 始终对 A0 和 D0 进行比较，并且它显示在选项 0 内。另一方面，通过确定使用 Config A 还是 Config B（由数据路径属性窗口中的可配置比较器输入属性指定）可以选择比较器 1。然后，通过数据路径输出中的比较器输出可以确定比较结果。

控制寄存器

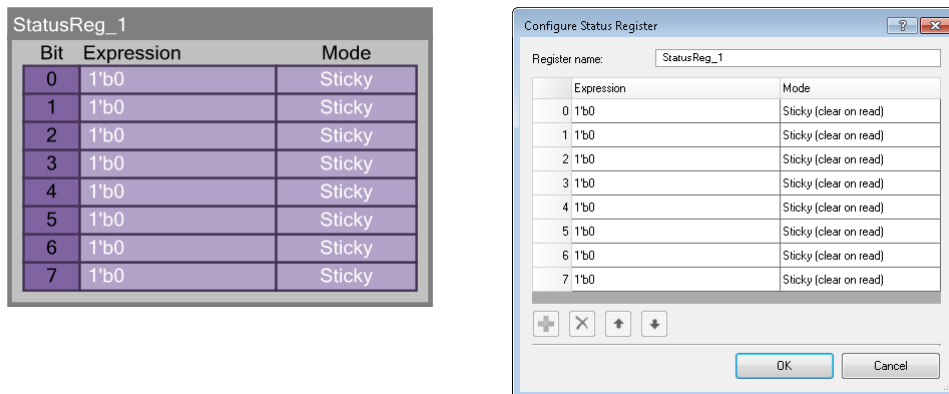
CPU 使用控制寄存器将各条指令发送给组件。每个控制寄存器共有八个可用位。可以在设计中使用这些位来控制组件操作的特性。将设计元素控制板中的控制寄存器子模块拖放到设计图纸上。双击它，以打开 **Configure** 对话框。



控制寄存器的每一位都有自己的名称。该名称是设计中所使用的信号名称。它的初始值可被设置为 1'b1 或 1'b0。可将控制寄存器中的每一位设置为 **Direct**、**Sync** 或 **Pulse** 等模式。有关更加详细的信息，请参考控制寄存器组件数据手册和 *数据参考手册 (TRM)*。

状态寄存器

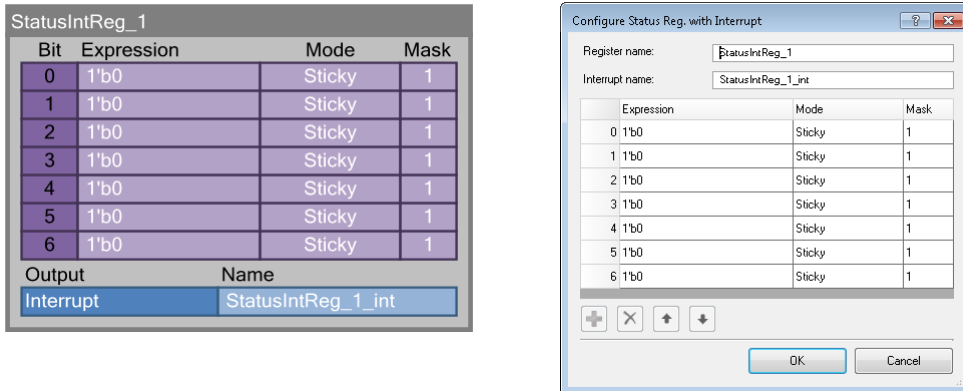
CPU 通过使用状态寄存器可以从组件中读取硬件信号。单状态寄存器中提供的八位是可用位，CPU 通过这些位能够查看组件中的信号。将设计元素控制板中的状态寄存器子模块拖放到设计图纸上。双击它，以打开 **Configure** 对话框。



通过状态寄存器位的表达式字段，一些信号可以在进入寄存器前经过 PLD 逻辑运算成一个信号。这样，即使不使用状态机，仍可采用 PLD 的某些性能。例如，数据路径输出的两个信号（又称“lessComp0”和“equalComp0”）在状态表达式字段中会组合在一起，形成“lessComp0 | equalComp0”信号。请注意，该表达式需要遵循 Verilog 语法。可以将状态位设置为 **Transparent**（透明）或 **Sticky**（粘滞）等模式。有关更详细的信息，请参考状态寄存器组件数据手册和 *数据参考手册 (TRM)*。

状态中断寄存器

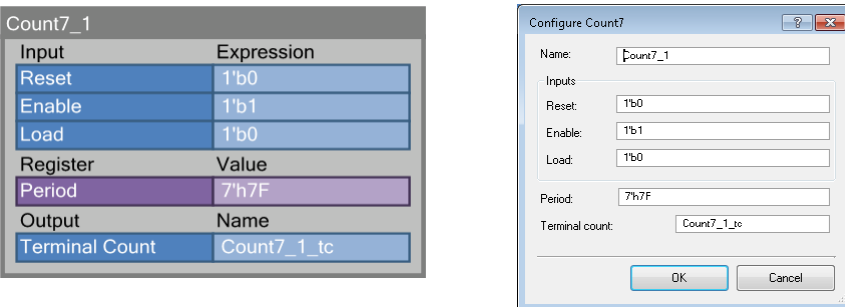
通过使用状态中断寄存器，可以生成状态位的可屏蔽中断。可将其中的七位作为输入使用，另外一位可作为中断输出使用。将 **Design Elements Palette**（设计元素控制板）中的状态中断寄存器子模块拖放到设计图纸上。双击它，以打开 **Configure** 对话框。



与状态寄存器相同，每一位都有一个可用于构成其逻辑的表达式字段。可以将各位设置为 **Transparent**（透明）或 **Sticky**（粘滞）等模式。掩码字段将决定该位是否被掩码，以生成中断。请注意，使用状态中断寄存器时，只能使用一个中断。有关更详细的信息，请参考状态寄存器组件数据手册和 *数据参考手册（TRM）*。

Count7

Count7 计数器是一个七位的递减计数器，当需要使用一个三到七位的计数器时，应该使用该计数器。与基于 **PLD** 或数据路径的计数器设计相比，该计数器可节省更多的资源。要想使用 **Count7** 计数器，请将设计元素控制板中的 **Count7** 计数器子模块拖放到设计图纸上。双击它，以打开 **Configure** 对话框。



Count7 计数器有 3 个输入：**Reset**（复位）、**Enable**（使能）和 **Load**（加载）。在计数器进行操作过程中，这些输入用于复位计数器、使能计数器并将周期值加载到该计数器内。默认情况下，将周期值设置为 **7'h7F**。该计数器具有一个输出（又称终端计数），计数器达到 0 时会将该输出驱动为高电平。有关更详细的信息，请参考 **Count7** 组件数据手册和 *技术参考手册（TRM）*。

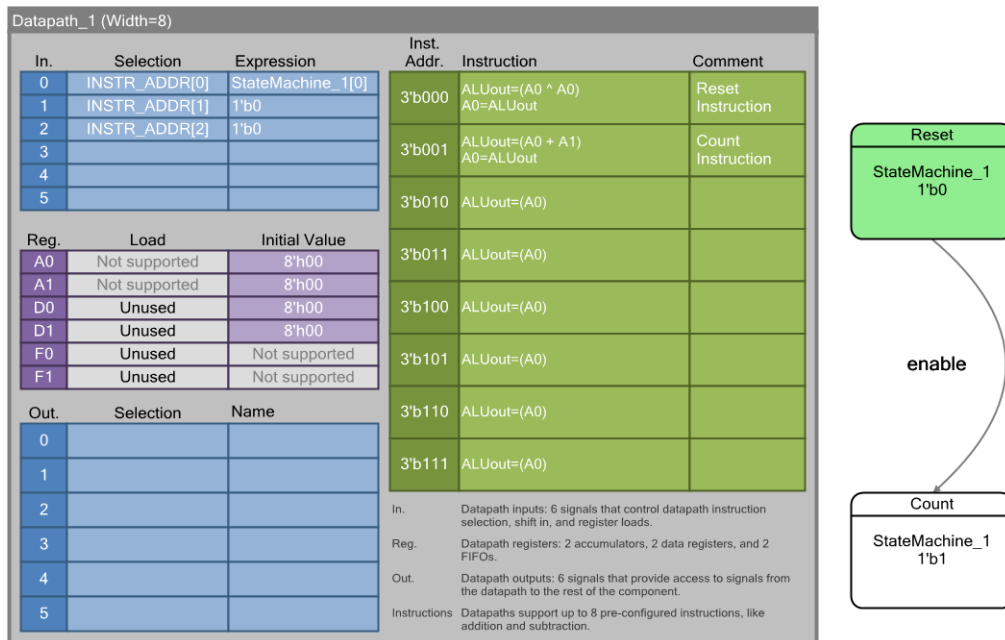
状态机

状态机通过使用 **PLD** 实现控制逻辑。通过该状态机，可以将控制信号发送到您设计中的元素内，并能够跟踪硬件中进行的各种操作。通过创建您状态机中不同的状态，可以实现上述的操作。一旦完成了某一状态中的子程序，或实现了某个条件时，该状态机会转换到状态转换所定义的另一种状态。

每个状态机都要求一个启动状态。它通常是闲置 / 复位状态，其中通过所生成的控制信号可使您设计中的其他 **UDB** 元素进入闲置或复位模式。然后，状态机将转到下一种状态，以启动某些操作。一般使用使能信号来触发该切换。进入该新状态后，根据所定义的切换条件，状态机会转换到另一种状态。如果在任意时间点都需要复位状态机，可以通过一个刺激（比如：**!enable**）使状态机返回到闲置 / 复位状态。该过程通常是通过下面所述的“复位条件”来实现的。

每一种状态都需要具有唯一的名称和状态编码。它同目标模块所执行的特定操作相对应。因此，某一状态下所生成的控制信号也要求是唯一的。此外还要定义退出该状态的条件。该条件可以通过变量分配来内部生成的，也可以是从外部提供并作为状态的输入。例如，内部生成的变量分配可以是一个基于 **PLD** 的计数器，一旦计数器达到特定值，它便转换到下一个状态。外部提供的输入可能是一个来自数据路径的状态信号，状态机可以使用该信号进行状态转换。

状态机一般用于控制数据路径指令的 **INSTR_ADDR** 位。在 *技术参考手册 (TRM)* 中，将这些位称为“动态配置”位。通过将各个信号映射到数据路径输入部分的 **INSTR_ADDR** 位，可以访问这些位。下图显示的是状态机如何控制数据路径。

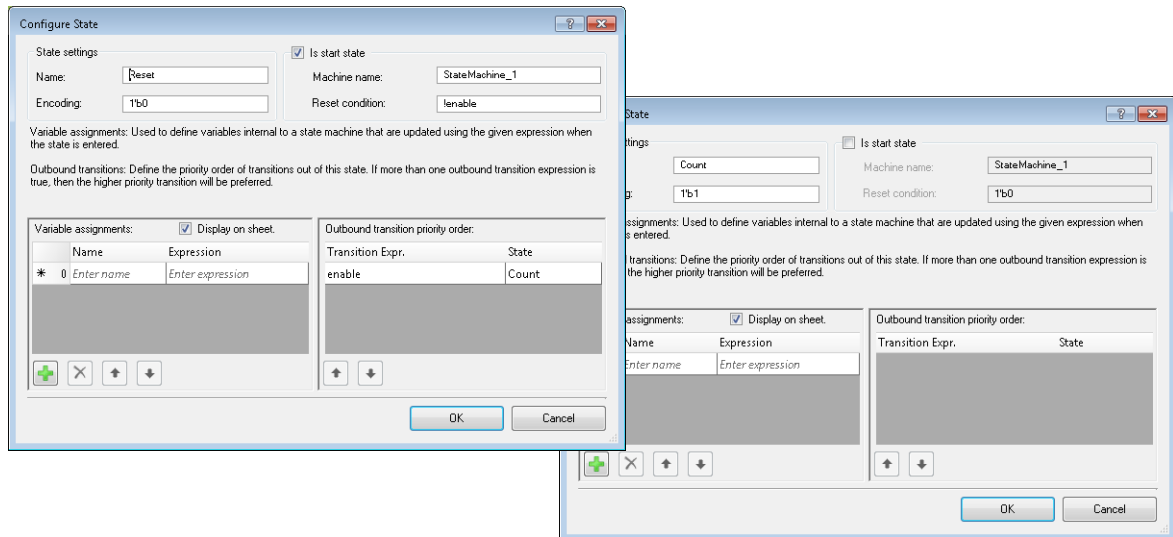


注意：本设计中没有使用 **D0**、**D1**、**F0** 和 **F1** 寄存器。此外，只分配 3 个可用 **INSTR_ADDR** 位中的最低有效位（**LSB**）。未分配的位被设为 **1'b0**。

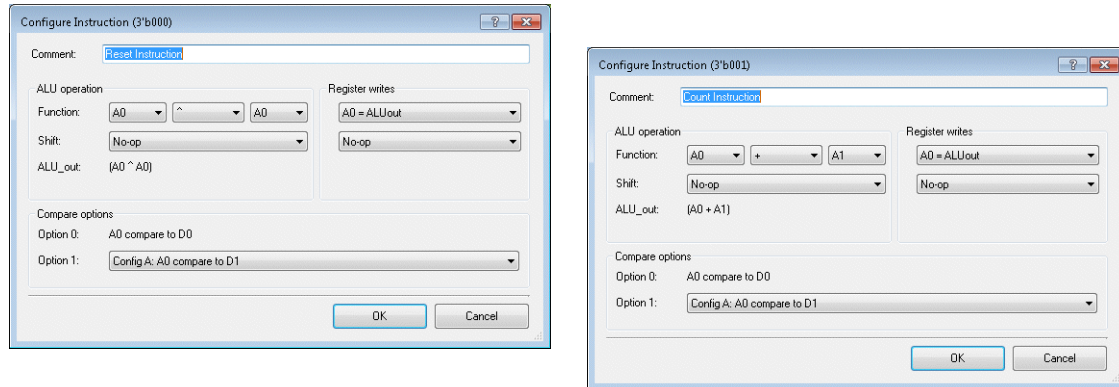
状态机用于控制数据路径实例 **Datapath_1**。

复位状态是起始状态，它将复位 **A0** 寄存器。当使能信号被设为高电平时，状态机将转换为计数（**Count**）状态，数据路径通过将保存在 **A1** 的值添加到 **A0** 来开始计数。如果使能信号为低电平，那么状态机将返回到复位状态，因此使数据路径处于复位状态。

状态机具有两种状态。起始状态被称为复位，计数状态被称为计数。复位状态和计数状态唯一的编码分别为 1'b0 和 1'b1。由于每一种状态必须是唯一的，因此在同一个 FSM 中不能复制这些值。该限制不适用于设计中的多个独立 FSM。

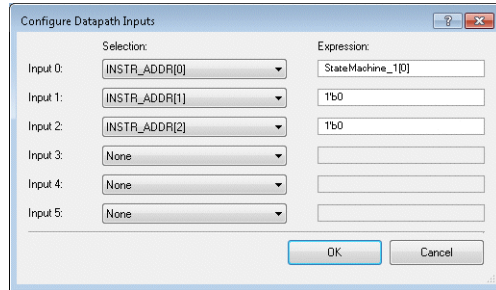


将数据路径设计为递增计数器，它会对存储在 A1 寄存器中的值的倍数进行计数。将 A0 作为累加器寄存器使用，用于保存所累加的结果。当“enable”（使能）信号为低电平时，累加器为空白。将“enable”信号驱动为高电平时，计数器会开始计数。通过使用两个数据路径指令，可以实现该操作。



- 指令 0：第一条是复位指令。在 ALU 中执行 $A0 \text{ XOR } A0$ ，以便在 ALUout 上输出 0。然后将该值存储在 A0 中，这样可清空 A0 寄存器。通过将 INSTR_ADDR 位设置为 3'b000，可以选择指令 0。
- 指令 1：该指令会将寄存器 A0 和 A1 的值加在一起。将该结果发送到 ALUout，然后输给 A0。通过将 INSTR_ADDR 位设置为 3'b001，可以选择指令 1。

要想将各条指令分配给数据路径，状态机必须先要将各个数值分配给 INSTR_ADDR 位。状态机名称本身是一个信号，可将它作为控制信号。由于本示例中只有两个编码为 1'b0 和 1'b1 的状态，我们将 LSB（StateMachine_1[0]）分配给 INSTR_ADDR[0]。其它 INSTR_ADDR 位未被使用，因此可以保持它们不变。



- 当“enable”信号为低电平时，状态机仍处于复位状态，并设置 StateMachine_1[0] = 1'b0。这样会使数据路径执行指令 0。
- 将“enable”信号驱动为高电平时，状态机会转到计数状态，并且 StateMachine_1[0] = 1'b1。现在，数据路径可以执行指令 1。
- 如果“enable”信号被驱动为低电平，那么状态机会返回复位状态，并且 StateMachine_1[0] = 1'b0。该转换是由复位条件（在此情况下为“!enable”）自动触发的。然后，数据路径将执行指令 0。

用户可以使用状态框图在 UDB 编辑器中构建状态机。各种状态可以链接在一起，构成一个基于 PLD 的状态机。为了开始设计状态机，请从设计元素控制板中拖放一个 FSM 状态。双击该状态，以打开配置对话框。

配置状态对话框由以下 4 个部分组成：状态设置、启动状态设置、变量分配以及输出转换的优先级顺序。根据状态操作和类型，可能不需要对所有部分进行定义。

- 状态设置用于指定状态名称及其相应的编码状态值。在当前的状态机中，状态名称及其相应的编码值必须是唯一的。
- 某一状态在 FSM 中可以是启动状态，也可以是其他状态。在状态配置对话框中，请选中“Is start state”（为启动状态）选框以将状态设置为启动状态，并将白色修改为绿色。如果某个状态为启动状态，那么应该为其定义唯一的名称。此外，也应该定义它的复位条件，在大部分情况下可通过使用“not enable”（未使能）或单独的“reset”（复位）输入信号触发该状态。发生复位条件时，状态机中的所有状态都能返回到启动状态。
- 变量分配用于定义状态机的内部变量，这些变量可用于控制 UDB 设计中的其它元素，如数据路径或 count7 计数器。可以在设计中任意位置读取该变量，但只能按照一个状态机写入它。不能从状态机写入设计范围的变量和输出（但可以通过状态机的变量获取它们的值）。
- 输出转换显示的是该状态可采用的输出转换。当多个条件为“真”时，它可用于设置转换的优先级。将一个转换表达式移动到更高的顺序时，它的优先级会比其他面的转换表达式的优先级高。但请注意，通常这是使转换表达式互斥的好方法。

通过点击并将某个状态拖动到另一个状态上，可以引起一个状态转换。两种状态放置在设计图纸上，并将其中一个配置为启动状态。通过点击并将启动状态拖放到下一种状态上，可引起一次转换。绘制转换后，会出现转换配置对话框。表达式字段可用于设置转换表达式。一旦该表达式为“真”，那么将在状态机中发生转换。如果多个转换条件为“真”，那么优先级更高的转换被执行。此外，通过转换配置对话框，还可以进行分配。这些分配与状态配置对话框中指定的变量分配相同，但它们只在特定转换发生后（而不在通过任何转换进入某个状态时）进行。

注意：将一个新的状态链接到现有的 FSM 会使现有的状态机名称得到扩展。另一方面，编码必须是唯一的，并且不能被扩展。

4.3.2.7 API

编译时，使用 UDB 编辑器进行的组件设计会生成一个名称为 `$INSTANCE_NAME`_defs.h` 的头文件，其中 `$INSTANCE_NAME`` 是您组件的实例名称。该文件包含了用于访问数据路径寄存器的定义以及用于执行数据路径 FIFO 配置工作的宏。

在软件中可以使用数据路径寄存器的定义对这些寄存器进行读写操作。通过使用下面的句法（其中，第一个是指向到寄存器的指针，第二个是寄存器），可以命名每个定义：

```
<COMPONENT INSTANCE NAME>_<DATAPATH INSTANCE NAME>_<REGISTER>_PTR
<COMPONENT INSTANCE NAME>_<DATAPATH INSTANCE NAME>_<REGISTER>_REG
```

可将指针定义用于 `cytypes.h` 文件所定义的 `CY_GET_REGn` 和 `CY_SET_REGn` 宏中，其中 ‘n’ 是与它们相关联的数据路径的宽度。通过这些宏，可以将数据写入到这些寄存器内，并从这些寄存器中读取数据。

用于数据路径 FIFO 配置工作的宏包括：清除数据路径 FIFO、设置 FIFO 级别模式、将 FIFO 设置为单缓冲区模式以及使 FIFO 返回正常模式。

- **Clear DP FIFOs**（清除数据路径 FIFO）— 通过 CPU，这些宏可清空设计中的特定 FIFO。进行操作后，FIFO 会返回到正常模式。
- **FIFO level mode**（FIFO 级别模式）— 这些宏可控制 4 字节 FIFO 确认总线状态的级别。可以设置为以下两种模式：NORMAL 和 MID。更多有关 FIFO 级别的信息，请参考第 44 页上的 [FIFO 模式](#)。
- **FIFO single buffer mode**（FIFO 单缓冲区模式）— 这些宏用于将特定 FIFO 设置为单缓冲区模式。通过单缓冲区模式，可以将 FIFO 作为 1 字的缓冲区（而不是 4 字 FIFO）使用。更多信息，请参考第 44 页上的 [FIFO 模式](#) 的内容。
- **Return to normal mode**（返回正常模式）— 这些宏将 FIFO 配置为正常的 4 字 FIFO。

这些宏通过访问辅助控制寄存器来设置这些配置。请注意，辅助控制寄存器中的多个位可用于不同的组件，因此应该使用关键区域 API 来稳定生成中断，以防止被损坏。下面示例显示的是清除实例名称为“MyComponent”的组件和“MyDatapath”数据路径的 FIFO 0。

```
uint8 interruptState;
/* Enter critical section */
interruptState = CyEnterCriticalSection();
/* Clears FIFO 0 */
MyComponent_MyDatapath_F0_CLEAR
/* Exit critical section */
CyExitCriticalSection(interruptState);
```

4.3.3 使用 Verilog 实现硬件

您可以通过使用 Verilog 来说明组件的功能。通过 PSoC Creator，您能够将 Verilog 文件添加到您的组件内，也可以编写和编辑该文件。请注意，如果选择使用 Verilog 实现，则要具备一定的条件，并且 PSoC Creator 只支持 Verilog 语言的可综合的子集；请参考 *Warp Verilog 参考指南*。您还需要使用数据路径配置工具来编辑数据路径的实例配置；请参考第 179 页上的数据路径配置工具。

此外，您同样可以使用 UDB 编辑器来实现 UDB 组件。请参见第 45 页上的使用 UDB 编辑器实现。

注意：Verilog 中的所有原始嵌入式组件都不会自动生成 API，将生成相应的 #define 值。比如 Verilog 中的 cy_psoc3_control 嵌入式寄存器不会导致生成 API（而通常将控制寄存器符号放置在原理图上时会自动生成这些 API）。

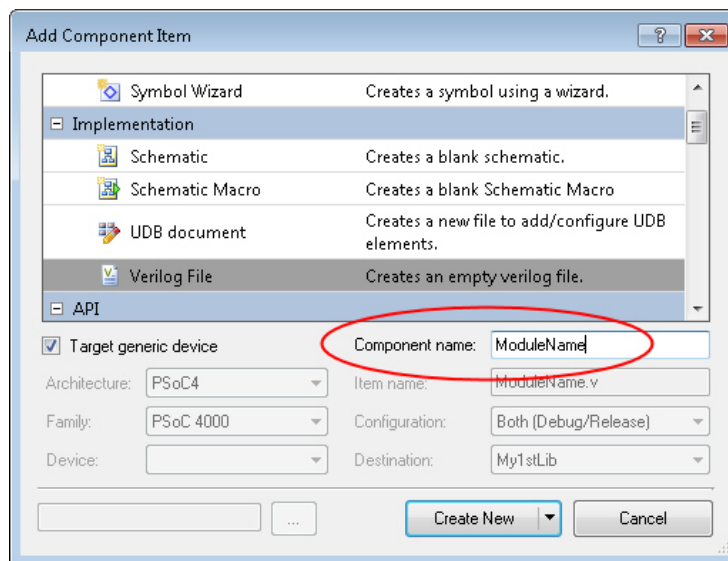
4.3.3.1 Verilog 文件的要求

PSoC Creator 允许每一组件只有一个 Verilog 文件，并且它只支持 Verilog 语言的可综合子集。

PSoC Creator 中使用符号所创建的组件名称必须与 Verilog 文件中定义的模块名称完全匹配。另外，PSoC Creator 所创建的 Verilog 文件名称与组件名称相同。设计工程师需要将该名称作为模块名称使用。

例如，PSoC Creator 中所创建的“ModuleName”组件会生成一个用以开发数字内容的 *ModuleName.v* 文件。该文件中的顶层模块必须为“ModuleName”（区分大小写），以便在 PSoC Creator 中能够正确地构建该模块。

在下面示例中，输入到 **Component name**（组件名称）文本框中的文本（“ModuleName”）必须与 Verilog 文件中使用的模块名称 [“module ModuleName(....)”] 相匹配。



当创建开始执行（如上述的示例所示）的符号时，请遵循命名规则。

赛普拉斯所提供的文件包含了组件 Verilog 文件中所需要的所有常量。这些常量位于 `cypress.v` 文件中。该文件包含在 PSoC Creator 中所有 Verilog 编译的搜索路径内。所有 Verilog 文件都使用了 `cypress.v` 文件，因此声名模块前必须有下面这一行：

```
`include "cypress.v"
```

4.3.3.2 添加 Verilog 文件

新建文件

如果没有现有的 Verilog 文件，那么可以使用 **Generate Verilog** 工具  创建基本的 Verilog 文件和相应的名称（“include “cypress.v”），以及基本模块的信息（具体取决于对符号的定义）。

现有文件

如果有现成的 Verilog 文件，您可以将它作为一个组件来添加。

1. 右击该组件，并选择 **Add Component Item**（添加组件项）项。
这时会出现 **Add Component Item**（添加组件项）对话框。
2. 选择 Verilog 图标。
注意： Verilog 文件会继承使用组件名称。请参见第 61 页上的 [Verilog 文件的要求](#)。
3. 如果需要，请将 **Create New**（创建新的文件）按钮切换为 **Add Existing**（添加现有的文件）。
4. 请点击省略号按钮 [...], 导航到文件所在的位置，然后点击 **Open**（打开）。
5. 点击 **Add Existing**。

这时会使用相应的文件名称将 Verilog 文件添加到 **Workspace Explorer**（工作区浏览器）内。

4.3.3.3 完成 Verilog 文件

双击 Verilog 文件以打开它，并完成该文件（若需要）。

注意： 使用 Verilog 实现的模块名称必须与它的组件名称相匹配，并且需要区分字母的大小写。

对于基于 UDB 的 PSoC 器件，Verilog 实现可应用于 Verilog 2005 的设计，并会编译成架构中可用的 PLD。但是，将优化一些其他可用模块的功能来避免使用有限的 PLD 资源。下面各节将对这些模块及其使用情况进行说明。

4.3.4 UDB 元素

PSoC 通用数字模块（UDB）包含了一些单独的可编程组件。进行设计时，可以通过 Verilog 访问它们。UDB 中包括下面各组件：数据路径、状态寄存器、状态中断寄存器、控制寄存器以及 count7 计数器。所采用的方法中需要时钟时，UDB 的各组件可以由特定的 Verilog 组件驱动。通过该 Verilog 组件，自定义程序能够将 UDB 中各个组件必须使用的时钟行为通知给调试器（fitter）。

注意： 虽然在该指南和组件名称中多次引用了“psoc3”，但这些引用都应用于基于 UDB 的所有 PSoC 器件，包括 PSoC 3 和 PSoC 5LP。

4.3.4.1 时钟/使能规范

对于使用时钟的 UDB 部分，可以指定所需的时钟类型（同步与异步）以及时钟的使能信号。调试器使用这些信息对时钟、使能所需的时钟类型进行检查，并对 UDB 中的时钟的布置/实现进行必要的更改，以确保它能够符合输出时钟的特性要求。

cy_psoc3_udb_clock_enable_v1_0

通过该基元，UDB 组件的时钟特性规范可以由该基元的输出驱动。该基元的输出只能驱动 UDB 元素。任何其他组件都会导致钳工中的 DRC 错误。该基元有一个参数：

- **sync_mode**: 它是用于指定所生成的时钟是同步（真）还是异步（假）的布尔（Boolean）参数。被默认设置为同步。

根据下面的示例对该元素进行实例化处理：

```
cy_psoc3_udb_clock_enable_v1_0 #(.sync_mode(`TRUE)) MyCompClockSpec (
    .enable(), /* Enable from interconnect */
    .clock_in(), /* Clock from interconnect */
    .clock_out() /* Clock to be used for UDB elements in this component */
);
```

4.3.4.2 数据路径

要想在您的设计中对一个或多个连续数据路径进行实例化处理，请在 Verilog 模块中使用下面的示例。

cy_psoc3_dp

它是可用的基本数据路径元素。选择该基本元素前，应该使用一些在该基元上构建的其他元素，因为这些元素上的所有 I/O 都列举在下面的实例中。但由于基于 UDB 的 PSoC 器件的 UDB 架构，会有一些限制，比如：许多信号在您的设计中能连接到的线数量是受限的。当需要单个数据路径时，最好应该使用 cy_psoc3_dp8 模块。

每个数据路径都有一些参数，可将它们作为已命名的参数发送给自己的模块。在 Verilog 文件中实现所有数据路径的各个参数时，推荐使用[数据路径配置工具](#)。通过该工具，可以读取 Verilog 文件、显示所有数据路径，并保存 Verilog 文件的正确信息，以供编译器（Warp）使用。

这些参数包括：

- **cy_dpconfig**: 是数据路径的配置，包括动态和静态配置。默认值为 {128'h0,32'hFF00FFFF,48'h0}
- **d0_init**: DATA0 寄存器的初始值。默认值为 8'b0
- **d1_init**: DATA1 寄存器的初始值。默认值为 8'b0
- **a0_init**: ACC0 寄存器的初始值。默认值为 8'b0
- **a1_init**: ACC1 寄存器的初始值。默认值为 8'b0

根据下面的示例对该数据路径进行实例化处理：

```
cy_psoc3_dp DatapathName(
    /* input          */ .clk(),           // Clock
    /* input    [02:00] */ .cs_addr(),      // Dynamic Configuration RAM address
    /* input          */ .route_si(),      // Shift in from routing
    /* input          */ .route_ci(),      // Carry in from routing
    /* input          */ .f0_load(),       // Load FIFO 0
    /* input          */ .f1_load(),       // Load FIFO 1
    /* input          */ .d0_load(),       // Load Data Register 0
    /* input          */ .d1_load(),       // Load Data Register 1
    /* output         */ .ce0(),           // Accumulator 0 = Data register 0
    /* output         */ .cl0(),           // Accumulator 0 < Data register 0
    /* output         */ .z0(),           // Accumulator 0 = 0
    /* output         */ .ff0(),          // Accumulator 0 = FF
    /* output         */ .ce1(),           // Accumulator [0]1] = Data register 1
    /* output         */ .cl1(),           // Accumulator [0]1] < Data register 1
    /* output         */ .z1(),           // Accumulator 1 = 0
    /* output         */ .ff1(),          // Accumulator 1 = FF
    /* output         */ .ov_msb(),       // Operation over flow
    /* output         */ .co_msb(),       // Carry out
    /* output         */ .cmsb(),         // Carry out
    /* output         */ .so(),           // Shift out
    /* output         */ .f0_bus_stat(),  // FIFO 0 status to uP
    /* output         */ .f0_blk_stat(),  // FIFO 0 status to DP
    /* output         */ .f1_bus_stat(),  // FIFO 1 status to uP
    /* output         */ .f1_blk_stat(),  // FIFO 1 status to DP
    /* input          */ .ci(),           // Carry in from previous stage
    /* output         */ .co(),           // Carry out to next stage
    /* input          */ .sir(),          // Shift in from right side
    /* output         */ .sor(),          // Shift out to right side
    /* input          */ .sil(),          // Shift in from left side
    /* output         */ .sol(),          // Shift out to left side
    /* input          */ .msbi(),         // MSB chain in
    /* output         */ .msbo(),         // MSB chain out
    /* input    [01:00] */ .cei(),         // Compare equal in from prev stage
    /* output    [01:00] */ .ceo(),         // Compare equal out to next stage
    /* input    [01:00] */ .cli(),         // Compare less than in from prv stage
    /* output    [01:00] */ .clo(),         // Compare less than out to next stage
    /* input    [01:00] */ .zi(),          // Zero detect in from previous stage
    /* output    [01:00] */ .zo(),         // Zero detect out to next stage
    /* input    [01:00] */ .fi(),          // 0xFF detect in from previous stage
    /* output    [01:00] */ .fo(),         // 0xFF detect out to next stage
    /* input          */ .cfbi(),         // CRC Feedback in from previous stage
    /* output         */ .cfbo(),         // CRC Feedback out to next stage
    /* input    [07:00] */ .pi(),          // Parallel data port
    /* output    [07:00] */ .po()          // Parallel data port
);
```

cy_psoc3_dp8

这是一个 8 位宽的数据路径元素。

与基本数据路径元素相同，该元素的每个数据路径都有相同的参数。将 “_a” 附加到所有参数，用以表示 LSB 数据路径（在下面的列表中，LSB 数据路径的 X = a）。

- cy_dpconfig_X,: 是数据路径的配置，包括动态和静态配置。默认值为 {128'h0,32'hFF00FFFF,48'h0}
- d0_init_X: DATA0 寄存器的初始值。默认值为 8'b0

- d1_init_X: DATA1 寄存器的初始值。默认值为 8'b0
- a0_init_X: ACC0 寄存器的初始值。默认值为 8'b0
- a1_init_X: ACC1 寄存器的初始值。默认值为 8'b0

根据下面的示例对该数据路径进行实例化处理：

```
cy_psoc3_dp8 DatapathName(
  /* input      */ /* .clk(),           // Clock
  /* input      [02:00] */ /* .cs_addr(),       // Dynamic COnfiguration RAM address
  /* input      */ /* .route_si(),      // Shift in from routing
  /* input      */ /* .route_ci(),      // Carry in from routing
  /* input      */ /* .f0_load(),       // Load FIFO 0
  /* input      */ /* .f1_load(),       // Load FIFO 1
  /* input      */ /* .d0_load(),       // Load Data Register 0
  /* input      */ /* .d1_load(),       // Load Data Register 1
  /* output     */ /* .ce0(),          // Accumulator 0 = Data register 0
  /* output     */ /* .cl0(),          // Accumulator 0 < Data register 0
  /* output     */ /* .z0(),           // Accumulator 0 = 0
  /* output     */ /* .ff0(),          // Accumulator 0 = FF
  /* output     */ /* .cel(),          // Accumulator [0|1] = Data register 1
  /* output     */ /* .cl1(),          // Accumulator [0|1] < Data register 1
  /* output     */ /* .z1(),           // Accumulator 1 = 0
  /* output     */ /* .ff1(),          // Accumulator 1 = FF
  /* output     */ /* .ov_msb(),       // Operation over flow
  /* output     */ /* .co_msb(),       // Carry out
  /* output     */ /* .cmsb(),        // Carry out
  /* output     */ /* .so(),           // Shift out
  /* output     */ /* .f0_bus_stat(), // FIFO 0 status to uP
  /* output     */ /* .f0_blk_stat(), // FIFO 0 status to DP
  /* output     */ /* .f1_bus_stat(), // FIFO 1 status to uP
  /* output     */ /* .f1_blk_stat() // FIFO 1 status to DP
);
```

cy_psoc3_dp16

这是两个 8 位宽的数据路径元素，它们被设置为 16 位模块的两个 8 位宽的连续数据路径。通过将各个单独数据路径间的转移、移入、移除和反馈等信号直接结合在一起，可以在整个 16 位的数据宽度上进行 ALU、转移和掩码操作。

与基本数据路径元素相同，该元素的每个数据路径都有同样的参数。将 “_a” 附加到所有参数内，表示数据路径的 LSB；另外将 “_b” 附加到所有参数内，则表示两个数据路径的 MSB（在下面列表中，LSB 数据路径的 X = a，MSB 数据路径的 X = b）。

- cy_dpconfig_X,: 是数据路径的配置情况，包括动态和静态配置。默认值为 {128'h0,32'hFFF0FFFF, 48'h0}。
- d0_init_X: DATA0 寄存器的初始值。默认值为 8'b0
- d1_init_X: DATA1 寄存器的初始值。默认值为 8'b0
- a0_init_X: ACC0 寄存器的初始值。默认值为 8'b0
- a1_init_X: ACC1 寄存器的初始值。默认值为 8'b0

根据下面的示例对该数据路径进行实例化处理：

```
cy_psoc3_dp16 DatapathName(
    /* input          */ .clk(),           // Clock
    /* input    [02:00] */ .cs_addr(),      // Dynamic Configuration RAM address
    /* input          */ .route_si(),      // Shift in from routing
    /* input          */ .route_ci(),      // Carry in from routing
    /* input          */ .f0_load(),       // Load FIFO 0
    /* input          */ .f1_load(),       // Load FIFO 1
    /* input          */ .d0_load(),       // Load Data Register 0
    /* input          */ .d1_load(),       // Load Data Register 1
    /* output [01:00] */ .ce0(),           // Accumulator 0 = Data register 0
    /* output [01:00] */ .cl0(),           // Accumulator 0 < Data register 0
    /* output [01:00] */ .z0(),           // Accumulator 0 = 0
    /* output [01:00] */ .ff0(),          // Accumulator 0 = FF
    /* output [01:00] */ .ce1(),           // Accumulator [0]1] = Data register 1
    /* output [01:00] */ .cl1(),           // Accumulator [0]1] < Data register 1
    /* output [01:00] */ .z1(),           // Accumulator 1 = 0
    /* output [01:00] */ .ff1(),          // Accumulator 1 = FF
    /* output [01:00] */ .ov_msb(),       // Operation over flow
    /* output [01:00] */ .co_msb(),       // Carry out
    /* output [01:00] */ .cmsb(),         // Carry out
    /* output [01:00] */ .so(),           // Shift out
    /* output [01:00] */ .f0_bus_stat(),  // FIFO 0 status to uP
    /* output [01:00] */ .f0_blk_stat(),  // FIFO 0 status to DP
    /* output [01:00] */ .f1_bus_stat(),  // FIFO 1 status to uP
    /* output [01:00] */ .f1_blk_stat()  // FIFO 1 status to DP
);
```

cy_psoc3_dp24

这是三个 8 位宽的数据路径元素，它们被设置为 24 位模块的三个 8 位宽的连续数据路径。通过将各个单独数据路径间的转移、移入、移除和反馈等信号直接结合在一起，可以在整个 24 位的数据宽度上进行 ALU、转移和掩码操作。

与基本数据路径元素相同，该元素的每个数据路径都有相同的参数。将 “_a” 附加到所有参数内，表示数据路径 LSB；附加 “_b” 表示中间的数据路径；附加 “_c” 表示 3 个数据路径的 MSB 数据路径（在下面列表中，LSB 数据路径、中间数据路径和 MSB 数据路径的 X 分别为 a、b 和 c）。

- **cy_dpconfig_X**：是数据路径的配置情况，包括动态和静态配置。默认值为 {128'h0, 32'hFF00FFFF, 48'h0}
- **d0_init_X**：DATA0 寄存器的初始值。默认值为 8'b0
- **d1_init_X**：DATA1 寄存器的初始值。默认值为 8'b0
- **a0_init_X**：ACC0 寄存器的初始值。默认值为 8'b0
- **a1_init_X**：ACC1 寄存器的初始值。默认值为 8'b0

根据下面的示例对该数据路径进行实例化处理：

```
cy_psoc3_dp24 DatapathName(
/* input      */ .clk(),           // Clock
/* input [02:00] */ .cs_addr(),     // Dynamic Configuration RAM address
/* input      */ .route_si(),      // Shift in from routing
/* input      */ .route_ci(),      // Carry in from routing
/* input      */ .f0_load(),       // Load FIFO 0
/* input      */ .f1_load(),       // Load FIFO 1
/* input      */ .d0_load(),       // Load Data Register 0
/* input      */ .d1_load(),       // Load Data Register 1
/* output [02:00] */ .ce0(),        // Accumulator 0 = Data register 0
/* output [02:00] */ .cl0(),        // Accumulator 0 < Data register 0
/* output [02:00] */ .z0(),        // Accumulator 0 = 0
/* output [02:00] */ .ff0(),       // Accumulator 0 = FF
/* output [02:00] */ .ce1(),       // Accumulator [0|1] = Data register 1
/* output [02:00] */ .cl1(),       // Accumulator [0|1] < Data register 1
/* output [02:00] */ .z1(),        // Accumulator 1 = 0
/* output [02:00] */ .ff1(),       // Accumulator 1 = FF
/* output [02:00] */ .ov_msb(),    // Operation over flow
/* output [02:00] */ .co_msb(),    // Carry out
/* output [02:00] */ .cmsb(),      // Carry out
/* output [02:00] */ .so(),       // Shift out
/* output [02:00] */ .f0_bus_stat(), // FIFO 0 status to uP
/* output [02:00] */ .f0_blk_stat(), // FIFO 0 status to DP
/* output [02:00] */ .f1_bus_stat(), // FIFO 1 status to uP
/* output [02:00] */ .f1_blk_stat() // FIFO 1 status to DP
);
```

cy_psoc3_dp32

这是四个 8 位宽的数据路径元素，它们被设置为 32 位模块的四个 8 位宽的连续数据路径。通过将各个单独数据路径间的转移、移入、移除和反馈等信号直接结合在一起，可以在整个 32 位的数据宽度上进行 ALU、转移和掩码操作。

与基本数据路径元素相同，该元素的每个数据路径都有相同的参数。将 “_a” 附加到所有参数表示数据路径 LSB；附加 “_b” 表示偏低的数据路径；附加 “_c” 表示偏高的数据路径、附加 “_d” 表示 4 个数据路径的 MSB 数据路径（在下面列表中，LSB 数据路径、偏低数据路径、偏高数据路径和 MSB 数据路径的 X 分别为 a、b、c 和 d）。

- cy_dpconfig_X: 是数据路径的配置情况，包括动态和静态配置。默认值为 {128'h0,32'hFFF0FFFF, 48'h0}
- d0_init_X: DATA0 寄存器的初始值。默认值为 8'b0
- d1_init_X: DATA1 寄存器的初始值。默认值为 8'b0
- a0_init_X: ACC0 寄存器的初始值。默认值为 8'b0
- a1_init_X: ACC1 寄存器的初始值。默认值为 8'b0

根据下面的示例对该数据路径进行实例化处理：

```
cy_psoc3_dp32 DatapathName(  
    /* input      */ .clk(),           // Clock  
    /* input      [02:00] */ .cs_addr(), // Dynamic Configuration RAM address  
    /* input      */ .route_si(),      // Shift in from routing  
    /* input      */ .route_ci(),      // Carry in from routing  
    /* input      */ .f0_load(),       // Load FIFO 0  
    /* input      */ .f1_load(),       // Load FIFO 1  
    /* input      */ .d0_load(),       // Load Data Register 0  
    /* input      */ .d1_load(),       // Load Data Register 1  
    /* output     [03:00] */ .ce0(),    // Accumulator 0 = Data register 0  
    /* output     [03:00] */ .cl0(),    // Accumulator 0 < Data register 0  
    /* output     [03:00] */ .z0(),     // Accumulator 0 = 0  
    /* output     [03:00] */ .ff0(),    // Accumulator 0 = FF  
    /* output     [03:00] */ .ce1(),    // Accumulator [0]1 = Data register 1  
    /* output     [03:00] */ .cl1(),    // Accumulator [0]1 < Data register 1  
    /* output     [03:00] */ .z1(),     // Accumulator 1 = 0  
    /* output     [03:00] */ .ff1(),    // Accumulator 1 = FF  
    /* output     [03:00] */ .ov_msb(), // Operation over flow  
    /* output     [03:00] */ .co_msb(), // Carry out  
    /* output     [03:00] */ .cmsb(),   // Carry out  
    /* output     [03:00] */ .so(),     // Shift out  
    /* output     [03:00] */ .f0_bus_stat(), // FIFO 0 status to uP  
    /* output     [03:00] */ .f0_blk_stat(), // FIFO 0 status to DP  
    /* output     [03:00] */ .f1_bus_stat(), // FIFO 1 status to uP  
    /* output     [03:00] */ .f1_blk_stat() // FIFO 1 status to DP  
);
```

4.3.4.3 控制寄存器

CPU 可以对控制寄存器进行写操作。在互连布线中，可使用它的每 8 位来控制 PLD 操作或数据通路的功能。可以在同一个设计中定义多个控制寄存器，但这些寄存器单独工作。

要想在您设计中对一个控制寄存器进行实例化处理，请在您的 Verilog 代码中使用下面的元素示例。

cy_psoc3_control

该 8 位控制寄存器具有下面各个可用参数：

- **cy_force_order**: 是一个 Boolean，如果在寄存器中不要求位顺序，那么编译器将使用该 Boolean 来提高路由器的能力。默认值为 **False**。通常，位的顺序非常重要，因此，应将该值设置为 **TRUE**。
- **cy_init_value**: 在配置芯片过程中加载寄存器的初始值。
- **cy_ctrl_mode_1**、**cy_ctrl_mode_0**（仅适用于 PSoc 3 ES3）：这两个参数都是可选的。它们都能控制一种操作模式，将该模式用于控制寄存器的每一位。请参考数据参考手册，了解有关每一种模式的详细信息：

cy_ctrl_mode_1	cy_ctrl_mode_0	说明
0	0	直接模式（默认设置）
0	1	同步模式
1	0	保留
1	1	脉冲模式

根据下面的示例对该控制寄存器进行实例化处理：

没有可选的模式参数：

```
cy_psoc3_control #(.cy_init_value (8'b00000000), .cy_force_order(`TRUE))
ControlRegName(
    /* output [07:00] */ .control()
);
```

有可选的模式参数：

```
cy_psoc3_control #(.cy_init_value (8'b00000000), .cy_force_order(`TRUE), .cy_ctrl_
mode_1(8'b00000000), .cy_ctrl_mode_0(8'b11111111)) ControlRegName(
    /* output [07:00] */ .control(), // Control bits
    /* input          */ .clock()    // Clock used for Sync or Pulse modes
);
```

4.3.4.4 状态寄存器

CPU 可读取状态寄存器。该寄存器的每 8 个位均支持互联布线，用以表明 PLD 或数据路径的状态。此外，这个 8 位的状态寄存器还能实现另一个功能。其中 7 位可作为状态位，另外，通过对 7 位的屏蔽输出进行“或”运算得到的第 8 位可用作中断源，如 `cy_psoc3_statusi` 模块中所示。可以在同一个设计中定义多个状态寄存器，但这些寄存器单独操作。

要想在您的设计中对某个状态寄存器进行实例化处理，请在您的 Verilog 代码中使用下列一种严格模块示例。

cy_psoc3_status

这是一个 8 位的状态寄存器。它具有下面各可用参数。应当将这些参数作为下面实例中所显示的已命名参数进行传送。

- **cy_force_order**：是一个 Boolean，如果在寄存器中不要求位顺序，那么编译器将使用该 Boolean 来提高路由器的能力。默认值为 **False**。通常，位的顺序非常重要，因此应将该值设置为 **TRUE**。
- **cy_md_select**：指的是寄存器中每个位的模式定义。这些位的模式可为 **Transparent**（透明模式）或 **Sticky**（粘滞模式）。每个位的默认模式均为 **Transparent**（透明模式）。

下面示例显示了该状态寄存器的实例化情况：

```
cy_psoc3_status #(.cy_force_order(`TRUE), .cy_md_select(8'b00000000)) StatusRegName (
    /* input [07:00] */ .status(), // Status Bits
    /* input          */ .reset(),  // Reset from interconnect
    /* input          */ .clock()   // Clock used for registering data
);
```

cy_psoc3_statusi

此模块是一个 7 位的状态寄存器，它所带有的中断输出是以这 7 个状态位为基础的。

对于 **statusi** 寄存器，可以使用硬件使能和软件使能。必须同时使能这两项才能生成中断。软件使能位于辅助控制寄存器内。请注意，辅助控制寄存器中的各个位可使用于不同的组件。因此，为了安全实现中断，请使用关键区域 **API**。请参考下面的示例：

```
#define MYSTATUSI_AUX_CTL (* (reg8 *) MyInstance__STATUS_AUX_CTL_REG)
uint8 interruptState;

/* Enter critical section */
interruptState = CyEnterCriticalSection();
/* Set the Interrupt Enable bit */
MYSTATUSI_AUX_CTL |= (1 << 4);
/* Exit critical section */
CyExitCriticalSection(interruptState);
```

该模块具有以下各个参数，应当将它们作为下面实例中已命名的参数进行传送：

- **cy_force_order**: 是一个 **Boolean**，如果在寄存器中不要求位顺序，那么编译器将使用该 **Boolean** 来提高路由器的能力。默认值为 **False**。通常，位的顺序非常重要，因此应将该值设置为 **TRUE**。
- **cy_md_select**: 指的是寄存器中每个位的模式定义。这些位的模式可是 **Transparent**（透明模式）或 **Sticky**（粘滞模式）。每个位的默认模式都是 **Transparent**（透明模式）。
- **cy_int_mask**: 指的是一个掩码寄存器，用于从该寄存器的 7 个位中选择一个生成中断。默认值为 0。这时，7 个位全部不能生成中断。

下面的示例显示了该状态寄存器的实例化情况：

```
cy_psoc3_statusi #(.cy_force_order(`TRUE), .cy_md_select(7'b0000000),
.cy_int_mask(7'b1111111)) StatusRegName (
    /* input  [06:00] */ .status(),    // Status Bits
    /* input          */ .reset(),     // Reset from interconnect
    /* input          */ .clock(),     // Clock used for registering data
    /* output         */ .interrupt()  // Interrupt signal (route to Int Ctrl)
);
```

4.3.4.5 Count7

通过使用一个简单的 7 位计数器，可以防止使用一条完整的 8 位数据路径或多个 PLD 实现同一个操作。该计数器采用了架构中的其它资源，用以避免珍贵的 PLD 和数据路径被用完。如果您的计数器是 3 到 7 位的，则该模块不会使用 PLD 或数据路径资源。该计数器在下面情况下非常有用：当您需要使用一个 4 位计数器绑定某个完整的数据路径或某个完整的 4 宏单元 PLD 时，您可以将该计数器作为通信接口的位计数器使用。

对于 count7 寄存器，可以使用硬件使能和软件使能。必须同时使能它们才可以使 count7 计数器工作。软件使能位于辅助控制寄存器内。请注意，辅助控制寄存器中的各个位可使用于不同的组件。因此，为了安全实现中断，请使用关键区域 API。请参考下面的示例：

```
#define MYCOUNT7_AUX_CTL (* (reg8 *) MyInstance__CONTROL_AUX_CTL_REG)
uint8 interruptState;

/* Enter critical section */
interruptState = CyEnterCriticalSection();

/* Set the Count Start bit */
MYCOUNT7_AUX_CTL |= (1 << 5);

/* Exit critical section */
CyExitCriticalSection(interruptState);
```

cy_psoc3_count7

该组件具有以下各参数，应当将它们作为下面实例中已命名的参数进行传送：

- **cy_period**: 是一个 7 位的周期值。其默认值为 7'b1111111。
- **cy_route_ld**: 是一个 Boolean 值，用于将路由到计数器的信号使能为加载信号。如果该值为 false，那么在到达终端计数时，作为一个参数进行传送的周期值将被重载到计数器内。如果该值为 true，则可以通过终端计数或加载信号将周期值加载到计数器内。默认值为 FALSE。
- **cy_route_en**: 是一个 Boolean 值，用于将路由到计数器的信号使能为一个使能信号。如果该值为 false，该计数器将始终处于有效状态；如果该值为 true，那么仅在使能输入信号为高电平时，该计数器才有效。默认值为 FALSE。

下面的示例显示了该计数器的实例化情况：

```
cy_psoc3_count7 #(.cy_period(7'b1111111), .cy_route_ld(`FALSE), .cy_route_en(`FALSE))
Counter7Name (
    /* input          */ .clock (), // Clock
    /* input          */ .reset(), // Reset
    /* input          */ .load(), // Load signal used if cy_route_ld = TRUE
    /* input          */ .enable(), // Enable signal used if cy_route_en = TRUE
    /* output [6:0]   */ .count(), // Counter value output
    /* output         */ .tc() // Terminal Count output
);
```

4.3.5 固定模块

对于所有固定模块，可以使用 `cy_registers` 的参数对所需模块的给定寄存器中的值进行明确设置。这些值将包含在为编程器件而生成的配置位流中。在调用 `main()` 函数前先要确定该位流。该参数的值是一个字符串，并且它按照下面的格式显示：

```
.cy_registers("reg_name=0x##[,reg_name=0x##]");
```

其中，`reg_name` 是模块中某个寄存器的名称（如技术参考手册中所列），例如，DSM 的 CR1 寄存器。“=” 后面是要分配给寄存器的十六进制值（前缀“0x”是可选的，但该值始终按十六进制数值识别）。如果所罗列的寄存器也由调试器配置，那么优先使用参数中所列出的值。

4.3.6 设计范围资源

有些组件被视为设计范围资源的一部分，并不包含在 Verilog 文件中，它们包括：

- 中断控制器
- DMA 请求

4.3.7 何时使用赛普拉斯所提供的基元代替逻辑

- 当需要通过 CPU 访问寄存器时
- 当进行大多数 ALU 操作，尤其是在它的宽度大于 4 位时
- 当进行移位操作，尤其是在它的宽度大于 4 位时
- 当进行掩码操作，尤其是在它的宽度大于 4 位时
- 当进行计数器操作，尤其是在它的宽度大于 4 位时

4.3.8 用于创建组件的 Warp 性能

PSoC Creator 组件可以使用 Verilog 文件来定义系统的数字硬件。该 Verilog 支持 Verilog 2001 中所使用的标准。

4.3.8.1 生成语句

在基于 UDB 的 PSoC 器件中开发组件时，能否使用生成语句非常重要。这是因为您不能将 ``ifdef` 语句和 Verilog 文件中已被参数化的值结合在一起使用。例如，如果用户所设置的参数要求使用一个 8 位宽的数据总线，那么您需要移除一条数据路径。通过使用有条件的生成语句，可以实现该操作。Warp 支持 Verilog 代码周边的有条件生成语句和 For-loop 生成语句。

注意：当尝试定义某个生成语句中的各个网时，请特别注意范围这一因素。

注意：在从数据路径、控制寄存器和状态寄存器传输到 API（位于已生成的 `cyfitter.h` 文件）的命名规范中，生成语句将占用一层。如果您使用的生成语句是“`begin : GenerateSlice`”，那么各组件将在它们的命名规范中下将一级。例如，生成语句中没有一条被命名为“`DatapathName`”的数据路径，那么在 `cyfitter.h` 文件中，该路径的变量如下定义：

```
#define ` $INSTANCE_NAME_DatapathName_u0_A0_REG 0
#define ` $INSTANCE_NAME_DatapathName_u0_A1_REG 0
#define ` $INSTANCE_NAME_DatapathName_u0_D0_REG 0
#define ` $INSTANCE_NAME_DatapathName_u0_D1_REG 0
```

但如果该数据路径位于 “GenerateSlice” 生成语句中，那么它的变量在 `cyfitter.h` 文件中将被定义为：

```
#define `$_INSTANCE_NAME_GenerateSlice_DatapathName_u0_A0_REG 0
#define `$_INSTANCE_NAME_GenerateSlice_DatapathName_u0_A1_REG 0
#define `$_INSTANCE_NAME_GenerateSlice_DatapathName_u0_D0_REG 0
#define `$_INSTANCE_NAME_GenerateSlice_DatapathName_u0_D1_REG 0
```

For Loop 生成语句示例

```
genvar i;
generate
for (i=0; i < 4; i=i+1) begin : GenerateSlice
    . . . Any Verilog Code using i
end
endgenerate
```

有条件生成语句的示例

```
generate
if(Condition==true) begin : GenerateSlice
    . . . Any Verilog Code
end
endgenerate
```

参数的使用情况

参数的位置在参数分配的右侧。该条件对组件的开发非常重要，尤其是在分配数据路径的配置信息的情况下。例如，在构建期间，SPI 组件要求能够动态设置数据位的数量。数据路径配置中 MSB 值的位字段要求 SPI 传输中基于数据宽度的另一个值。这时，可定义一个名称为 `MSBVal` 的参数，并且根据用于定义模块传输大小的现有参数 `NumBits` 分配该参数。更多有关实现上述操作的方法，请参考数据路径配置工具所介绍的内容。配置数据路径时，不应该手动编辑传输到数据路径的参数。但在其他参数定义中，您可以随意使用该方法。

localparam 参数的使用情况和已命名参数

根据 PSoC Creator 中各参数的可配置性能，各个组件广泛使用了这些参数。请小心保护您的参数，以阻止它们被意外的数值覆盖掉。在 Warp 的 PSoC Creator 版本中，有两个特性可帮助您解决这个问题。它们分别是 `localparam`（本地参数）的使用和已命名参数的传输。

使用 `Defparam` 参数时很容易出错，因此 Warp 添加了传输已命名参数的功能。该功能被广泛使用，如前一节定义的标准模块所示。已命名的参数包含在 `#(...)` 块中（如下面示例所示），并在实例声明时间内被传输：

```
cy_psoc3_status #(.cy_force_order(`TRUE)) StatusReg (
    . . . status register instantiation
```

请尽量使用本地参数保护您的参数。在 **SPI** 模块的上面示例中，应将 **MSBVal** 定义为本地参数，因为它根据参数 **NumBits** 设置的。用户不能以更高的级别设置该参数。该示例的代码如下：

```
module SPI(...)
parameter NumBits = 5'd8;
localparam MSBVal = (NumBits == 8 || NumBits == 16) ? 5'd8 :
                    (NumBits == 7 || NumBits == 15) ? 5'd7 :
                    (NumBits == 6 || NumBits == 14) ? 5'd6 :
                    (NumBits == 5 || NumBits == 13) ? 5'd5 :
                    (NumBits == 4 || NumBits == 12) ? 5'd4 :
                    (NumBits == 3 || NumBits == 11) ? 5'd3 :
                    (NumBits == 2 || NumBits == 10) ? 5'd2 :
                    (NumBits == 9) ? 5'd1;

endmodule
```

4.4 使用软件实现组件

使用软件实现组件指的是无需硬件干预的方法。您不需使用任何原理图或 **Verilog**，只要使用软件文件。这种实现过程的其它操作与其它组件类型的实现过程完全相同。

仅使用软件实现的组件示例是用来链接某些组件代码的接口。

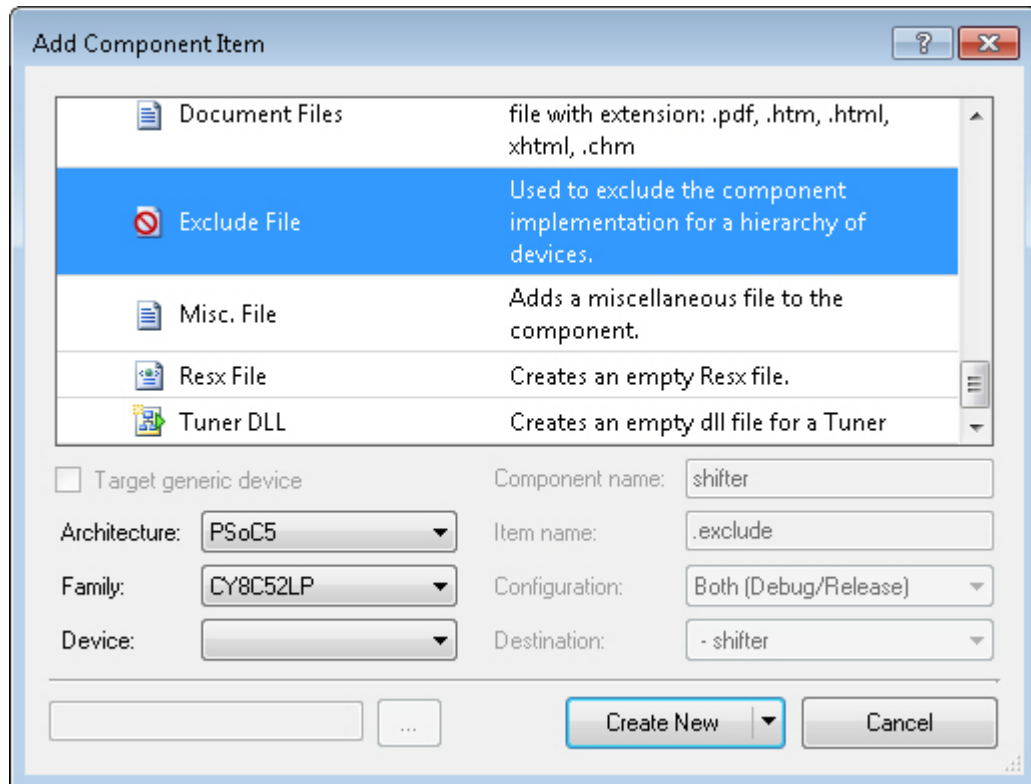
4.5 移除某个组件

您不但可以创建某个特定的架构、系列或器件的实现，而且还可以移除组件。移除某个组件指的是该组件不再显示在组件目录中，也不能将其用于所指定的架构、系列或器件。

注意： 某个特定实现会覆盖掉被移除的文件。例如，如果您移除了整个架构，但又创建了某个特定系列或器件的实现，那么该组件可用于该系列或器件。

要想移除某个组件，先按照下面步骤将一个移除文件添加到您的组件中：

1. 右键单击组件，然后选择 **Add Component Item** （添加组件项）项。
这时会出现 **Add Component Item** （添加组件项）对话框。



2. 向下滚动到 **Misc** 部分，然后单击 **Exclude File** （移除文件）图标。
3. 在 **Target** （目标）下方，指定需要移除的组件所在的架构、系列和 / 或器件。
4. 单击 **Create New** （创建新项）。

根据所指定的 **Target** 项，将在工作区浏览器的相应目录中列出被移除的组件。

当您完成该组件并尝试在某个设计中使用它时，请先选择您所添加的移除文件所在的架构、系列和 / 或器件，然后验证该组件是否已经失效。

5. 仿真硬件



赛普拉斯的 PSoC Creator 为组件开发人员提供了功能仿真模型，用于仿真和调试开发人员做出的设计。本章节向您介绍了 CPU 是如何访问（读和写）不同组件的。

5.1 仿真环境

通过 Warp，可以实现 Verilog 源设计的合成。赛普拉斯虽然目前尚未提供 Verilog 仿真器，但已经提供了第三方仿真器用于实现预合成仿真时所需要的系统文件。Warp 使用两个源文件来实现 Verilog 设计的合成操作：*lpm.v* 和 *rtl.v*。它们位于下列目录中：

```
$CYPRESS_DIR/lib/common/
```

在本章中，将 `$CYPRESS_DIR` 表示为 `$INSTALL_DIR/warp/`。

lpm.v 文件用于定义库中参数化模块库所使用的特定参数值。*rtl.v* 文件用于定义合成引擎所使用的不同基元，用于映射到目标器件的内部架构中。

该目录中还包含了 *cypress.v* 文件，该文件用于将设计链接到相应的 PSoC 模型。如果使用这些模型进行合成操作，那么用户的原始设计必须包含以下各行：

```
`include "cypress.v" //to use the PSoC-specific modules
`include "lpm.v"      //to use the LPM modules
`include "rtl.v"      //to use the RTL modules in the user's design
```

对于 Verilog 仿真器，请改用 `+incdir+`。例如：

```
+incdir+$CYPRESS_DIR/lib/common
```

或

```
+incdir+$CYPRESS_DIR/lib/sim/presynth/vlg
```

为方便使用并与 Verilog 预合成文件结构相兼容，Verilog 源文件也被映射到 `$CYPRESS_DIR/lib/sim/presynth/vlg` 中，这样便可以使用后面的示例。

一般情况下，通过下面的语句可以调用使用这些模块集的 Verilog 仿真器：

```
vlog +incdir+$CYPRESS_DIR/lib/common <test bench>.v <testfile>.v
```

为访问所需的模块，您必须使用合适的 ``include` 子语句。

5.2 模型位置

PSoC Creator（通过 Warp）提供的仿真模型位于目录结构中的 Warp 部分，具体路径如下：

```
$CYPRESS_DIR/lib/sim/presynth/vlg
```

5.3 测试工作台开发

在仿真环境中，必须为物理设备中 CPU 执行的某些功能提供测试工作台。为了对该功能进行仿真，测试工作台必须包含驱动器件模型内部信号的模块。

必须对器件模型所包含的组建进行计时、写入和 / 或读取操作。在器件模型中，通过提供与组件相关的内部信号（CPU 时钟）和操作任务（读 / 写），可以对各组件进行操作。下面各节介绍了该功能，并提供了几个示例。

5.3.1 提供 CPU 时钟

虽然状态寄存器、控制寄存器和 datapath（数据路径）没有连接到 CPU 时钟的引脚，但是仍要为它们提供时钟，用于通过 CPU 仿真芯片内发生的操作。为了对这些组件进行 CPU 访问，每个模型中均包含了一个内部 reg。受影响的模型分别为控制寄存器（cy_psoc3_control）、两种状态寄存器（cy_psoc3_status 和 cy_psoc3_statusi）以及 datapath（cy_psoc3_dp）。在每个组件中，都将信号命名为 cpu_clock。

5.3.1.1 CPU 时钟示例

该示例中包含了几个层次等级。底层（pattern_matcher）是 cy_psoc3_dp 的一个实例。因此，该模型中的 cpu_clock 寄存器由初始子句通过测试工作台进行操作。

```
// Build the CPU clock generator
reg CPUClock = 0;
localparam cycle = 10;
initial
begin
    while (!done)
    begin
        # (cycle / 2);
        CPUClock <= ~CPUClock;
        ela.matcher.pattern_matcher.cpu_clock = CPUClock;
    end
end
```

如果设计中具有多个控制寄存器、状态寄存器或 datapath，那么时钟发生器模块应包含多个分配语句（如下面示例所示）。

```
ela.matcher.pattern_matcher.cpu_clock = CPUClock;
ela.compress.compressor.cpu_clock    = CPUClock;
ela.trigger.PosEdgeReg.cpu_clock      = CPUClock;
ela.trigger.NegEdgeReg.cpu_clock      = CPUClock;
ela.ELAControl.cpu_clock              = CPUClock;
```

5.3.2 寄存器访问任务

还需要仿真 CPU 对上面三个组件中内部寄存器进行的访问。为了完成该操作，已向每个组件模型内添加了几个 task 函数。下面列出了这些任务的详细情况：

```
cy_psoc3_control: task control_write;
cy_psoc3_status: task status_read;
cy_psoc3_statusi: task status_read;
cy_psoc3_dp: task fifo0_write;
task fifo0_read;
task fifo1_write;
task fifo1_read;
task a0_write;
task a0_read;
task a1_write;
task a1_read;
task d0_write;
task d0_read;
task d1_write;
task d1_read;
```

这些任务用于读取和写入相应组件模型中不同的寄存器。调用任何一个任务时都要使用一个参数。

- 对于写入任务，该参数是将写入到寄存器内的 8 位数据。
- 对于读取任务，会返回这个 8 位寄存器数据，并将其分配给相应的参数。

一般通过下面格式调用这些任务：

```
// data (value) written to control register
<component_path>.control_write(value);

// data read from status register and stored in read_data
<component_path>.status_read(read_data);
```

下面内容显示的是使用情况的示例：

5.3.2.1 FIFO 写入

```
// Retrieve the data
reg[07:00] r_index;

always @(posedge rd_req or posedge new_cmd)
begin
    if (new_cmd)
        r_index = 0;
    if (rd_req)
    begin
        i2c_slave.data_dp.U0.fifo0_write (slave_mem[r_index]);
        r_index = r_index + 1;
        @(negedge rd_req);
    end
end
```

5.3.2.2 FIFO 读取

```
// Get the stuff from the FIFO
reg [07:00] read_data;

always @(posedge Clock)
begin
    if (ela.compress.DataAvailable)
    begin
        ela.compress.compressor.fifo0_read(read_data);
        $write ("%h ", read_data);
    end
end
```

5.3.2.3 寄存器读取

```
// Check the value of the CRC result
reg [15:00] seed_current;

always @(posedge prs.dcfg)
begin
    prs.PRSDp_a.a0_read(seed_current[07:00]);
    prs.PRSDp_a.a1_read(seed_current[15:08]);
end
```

5.3.2.4 寄存器写入

```
// Set up some initial conditions
initial
begin
    my_dp.U0.a0_write(8'hFF);
    my_dp.U0.d1_write(8'h57);
end
```

5.3.2.5 状态读取

```
// Check for the process being done.
reg [6:0] done;
always @(clock)
    param_timer.statusi.status_read(done);
```

5.3.2.6 控制器读取

```
// Do the testing
localparam POS_EDGE = 6'b001010;
localparam NEG_EDGE = 6'b001100;

initial
begin
    ela.trigger.PosEdgeReg.control_write(POS_EDGE);
    ela.trigger.NegEdgeReg.control_write(NEG_EDGE);
end
```

6. 添加 API 文件



本节介绍将应用编程接口（API）文件和代码添加到组件内的过程以及其他主题，如 API 生成和模板扩展。

6.1 API 概况

API 文件用于定义组件接口，该接口可为组件提供各种功能和参数。作为一个组件创建者，您要为组件创建 API 文件模板。这些模板是最终用户的特定实例。

6.1.1 API 生成

API 生成是一个过程，其中 PSoC Creator 创建了基于实例的代码，它包括生成用于定义硬件地址的符号常量。硬件地址是放置步骤引起的结果（该步骤是编译过程中调配阶段的一部分）。

6.1.2 文件命名规范

API 目录中的所有文件名称均被扩展为符合实例 _ 名称标准的文件，以进行编译。例如，将计数器 API 文件为 *Counter.c*、*Counter.h*、和 *CounterINT.c*。

如果有一个名为 *foo* 的顶级计数器组件，那么将生成 *foo_Counter.c*、*foo_Counter.h*、和 *foo_CounterINT.c* 文件。如果一个低级计数器实例的名称为 *bar_1_foo*，将生成 *bar_1_foo_Counter.c*、*bar_1_foo_Counter.h*、和 *bar_1_foo_CounterINT.c* 文件。

6.1.3 API 模板扩展

6.1.3.1 参数

创建 API 模板代码时，您可以通过以下语法使用内置、正式和本地参数（包括用户定义的参数），以扩展模板：

```
`@<parameter>`  
`$<parameter>`
```

这两种语法均有效，并能够扩展以提供指定的参数值。例如：

```
void `$INSTANCE_NAME`_Start(void);
```

对于名称为 *foo_1* 的计数器实例，该代码段将被扩展为：

```
void foo_1_counter_Start(void);
```

6.1.3.2 用户定义类型

下面的结构定义了用户定义类型的映射情况。

```
`#DECLARE_ENUM type_name`      — 为已指定的类型名称确定键名
`#DECLARE_ENUM_ALL`            — 为组件所使用的所有类型确定键名
```

注意：可以局部的（**PARITY**）或外部的（**UART__PARITY**）定义 **type_name**。

每一个目录都会生成 **#define** 语句的序列。对于实例特有的类型，扩展名称使用以下形式：

```
<path_to_instance>_<keyname>
```

对于假借的类型，扩展名称使用以下形式：

```
<path_to_instance>_<componentname>__<keyname>
```

请注意，实例名称后面是一个下划线（如 **API** 函数其他部分所示）；但如果组件名称被使用，那么它的后面将是一个双下划线（如系统其他部分所示）。这便表示局部定义的键值不能与寄存器名称发生冲突。

示例

```
Component:      Fox
Type:           Color (RED=1, WHITE=2, BLUE=3)
Also Uses:      Rabbit__Species
Component:      Rabbit
Type:           Species (JACK=1, COTTON=2, JACKALOPE=3, WHITE=4)
```

设计带有一个原理图，该原理图包含了一个实例名称为 **bob** 的 **Fox** 组件。在 **bob** 的 **API** 文件中，以下行：

```
`#declare_enum Color`
`#declare_enum Rabbit_Species`
```

被扩展为：

```
#define path_to_bob_RED 1
#define path_to_bob_WHITE 2
#define path_to_bob_BLUE 3
#define path_to_bob_Rabbit__JACK 1
#define path_to_bob_Rabbit__COTTON 2
#define path_to_bob_Rabbit__JACKALOPE 3
#define path_to_bob_Rabbit__WHITE 4
```

在 **bob** 的 **API** 文件中，以下行：

```
`#declare_enum_all`
```

被扩展为：

```
#define path_to_bob_RED 1
#define path_to_bob_WHITE 2
#define path_to_bob_BLUE 3
#define path_to_bob_Rabbit__JACK 1
#define path_to_bob_Rabbit__COTTON 2
#define path_to_bob_Rabbit__JACKALOPE 3
#define path_to_bob_Rabbit__WHITE 4
```

6.1.4 有条件的 API 生成情况

对于控制寄存器、状态寄存器或中断的特定场合，如果将设计映射到组件内可以确定是否正在使用组件（根据组件的连接情况），那么映射过程将从映射设计中移除组件。为了通知 API 生成系统不用再为组件添加代码，

```
#define `$_INSTANCE_NAME`__REMOVED 1
```

将在 *cyfitter.h* 文件中生成，以便可以有条件地将组件 API 编译到设计内。另外，为了使用 `#define`，需要创建所有代码以使用被移除的组件中的 API，因此不会访问移除组件。

6.1.5 Verilog 层次更换

如果您使用的是 Verilog 实现设计，您可能想要访问控制寄存器的多种调试常量。因此，PSoC Creator 为您提供了层次路径所生成的 '[...]' 更换机制，这样有助于您访问调试常量。例如：

```
foo_1`[uart_x,ctrl]`ADDRESS
```

该代码段可扩展为：

```
foo_1_uart_x_ctrl_ADDRESS
```

6.1.6 合并区域

合并区域用于定义最终用户编写代码的区域。这些代码将在组件或源代码的更新过程中得到保留。可以使用以下语句来定义该区域：

```
/* `#START <region name>` */  
/* `#END` */
```

最终用户在该区域中编写的所有代码内容将在随后进行的文件更新过程中得到保留。如果后面的文件版本没有包括相同名称的区域，那么将旧文件的全部区域复制到文件后面，并在注释内容中进行解释。

6.1.7 API 案例

您可以为组件指定 API 和原理图，或者仅指定其中一个（一个基元类型的组件可能不包括任何原理图和 API）。下面列出了是 API 可能发生的情况：

- 提供了原理图，但不存在 API。

在这种情况下，组件仅是其他组件的层次组合。在设计中进行实例化时，需要使该组合过程更顺畅。

请注意，向用户编辑组件项目的普通部分的设计输入文档时，不能将特定系列的基元放置在原理图中。同样，当用户编辑一个系列的设计输入文档时，只能向原理图内放置适用于此系列的基元。

- 存在 API，但没有提供原理图。

在这种情况下，组件将不会复用已存在的组件，并提供 API。只有使用了其他组件的纯软件组件时才会发生这种情况。

- 提供了原理图和 API。

这是一个典型的组件，设计中包括一个 API 和一个或多个基元以及 / 或其他组件。如果原理图包括其他已经实例化的组件（这些组件具有自己的 API），顶级设计的代码生成过程会变得更加复杂。例如，假设 **Count32** 组件是由两个 **Count16** 组件构成的，其中 **Count32** 普通原理图内包含了实例名称 **u1** 和 **u2**。当在顶级设计内对 **Count32** 组件进行实例化时，如果生成了 API，那么代码生成机制必须考虑层级场合。

在这种情况下，**Count32** 组件创建者需要使用模板代码上 **u1** 的 API，如

```
`$INSTANCE_NAME` `[u1]`Init()。
```

注意，如果 **Count16** 组件包括 **Count8** 组件，例如子结构相同，那么代码生成过程中它们的实例名称也会自然分级。

- 没有提供原理图，也没提供 API。

这是一种无效的配置（基元类型的组件除外）

6.2 将 API 文件添加到组件内

根据组件要求，通过使用 PSoC Creator 可将 API 文件添加到组件内：

1. 右击该组件，然后选择 **Add Component Item**（添加组件项）项。
这时会出现 **Add Component Item** 对话框。
2. 选择您想要添加的组件项的图标。
3. 选择 **Target**（目标）项。

注意：您可以将组件项指定为用于普通器件还是用于特定的系列和 / 或器件。

4. 在 **Item name**（项名称）中键入内容。
5. 点击 **Create New**。

组件项显示在 **Workspace Explorer** 内。根据您所指定的目标选项，组件项可能会位于子目录内。

6. 重新实现该过程，以添加其他 API 文件。

6.3 完成 .c 文件

.c 文件包含您组件的函数和参数定义。它也包含您添加到组件内的基本组件头文件的参照指令，如下面示例所示：

```
#include "`$INSTANCE_NAME`.h"
```

6.4 完成 .h 文件

.h 文件通常包含组件的函数原型。头文件应该包含 *cyfitter.h* 和 *cytypes.h* 文件的参照指令。这些文件是在编译过程中生成的。*cyfitter.h* 文件定义了代码生成步骤中计算出的所有实例特定地址。*cytypes.h* 提供了宏和定义，以使代码作为写入工具链，处理器（PSoC 3/ PSoC 5）并不并不清楚它。

下面显示的实例介绍了如何包含这些文件：

```
#include "cytypes.h"
#include "cyfitter.h"
```

生成头文件时，会使用到一些命名规范。对于工作寄存器（状态、控制、周期、掩码等等），将指定的字符附加到完整的实例名称（包括任何分级）内，如下表所示：

工作寄存器	名称（在每个工作寄存器名称前面添加了 < 实例名称 >_< 生成切片名称 >）
Status	_< 状态寄存器组件名称 >__STATUS_REG
Status Auxiliary Control	_< 状态辅助控制寄存器组件名称 >__STATUS_AUX_CTL_REG
Control	_< 控制寄存器组件名称 >__CONTROL_REG
Control Auxiliary Control	_< 控制辅助控制寄存器组件名称 >__CONTROL_AUX_CTL_REG
Mask	_< 屏蔽寄存器组件名称 >__MASK_REG
Period	_< 周期寄存器组件名称 >__PERIOD_REG
Accumulator 0	_<A0 寄存器组件名称 >__A0_REG
Accumulator 1	_<A1 寄存器组件名称 >__A1_REG
Data 0	_<D0 寄存器组件名称 >__D0_REG
Data 1	_<D1 寄存器组件名称 >__D1_REG
FIFO 0	_<F0 寄存器组件名称 >__F0_REG
FIFO 1	_<F1 寄存器组件名称 >__F1_REG
Datapath Auxiliary Control	_<DP 辅助控制寄存器组件名称 >__DP_AUX_CTL_REG

7. 自定义组件



自定义组件是指允许使用自定义代码（C#）来加强或代替 PSoC Creator 中某个实例化组件的默认行为机制。有时，该类代码被称为“customizers”（自定义程序），它可以：

- 自定义 **Configure**（配置）对话框
- 根据参数值自定义符号形状 / 显示界面
- 根据各参数，自定义符号终端名称、数量或配置
- 生成自定义 **Verilog** 代码
- 生成自定义的 **C/ 汇编语言** 代码
- 与时钟系统配合使用（适用于时钟和 **PWM** 组件）

本章节提供了用于自定义组件的示例，并提供了使用源代码进行自定义的引导，同时还列出并说明了可用于自定义组件的接口。*icyinstancecustomizer.cs* C# 源文件通过自定义程序接口为这些方法提供了已定义的参数和返回值。*cydsextensions* 项目包含了进行自定义时需要的源文件。该项目包含在 PSoC Creator 中，本文档位于 *PSoC Creator 自定义 API 参考指南* 内（*customizer_api.chm*，位于 *组件创建指南* 所在的相同目录下）。

7.1 使用源代码的自定义程序

PSoC Creator 可以接收使用 C# 开发的自定义程序。以下各节介绍了 C# 源代码的各方面内容：

7.1.1 保护使用了源代码的自定义程序

源代码形式的自定义程序可以依赖于外部程序集。所以如果希望避免以源代码形式发布自定义程序，您可以利用一个源代码存根，通过它加载外部程序集，并调用外部程序集中的方法。

7.1.2 开发流程

您需要为给定的项目提供 C# 源文件、资源文件和程序集参考。运行时，PSoC Creator 可以自动使用已给代码构建自定义 DLL 文件。当发生下面情况时，会自动构建 DLL 文件：

1. 打开了某个项目，它或它任意的依赖属性带有自定义程序源文件，但不存在有效的自定义程序 DLL 或先前创建的自定义程序 DLL 已经过时。
2. 创建了一个项目，该项目或它任意的依赖属性带有自定义程序源文件，但不存在有效的自定义程序 DLL 或先前创建的自定义程序 DLL 已经过时。
3. 组件创建者明确指出要求 PSoC Creator 创建新的自定义程序 DLL。

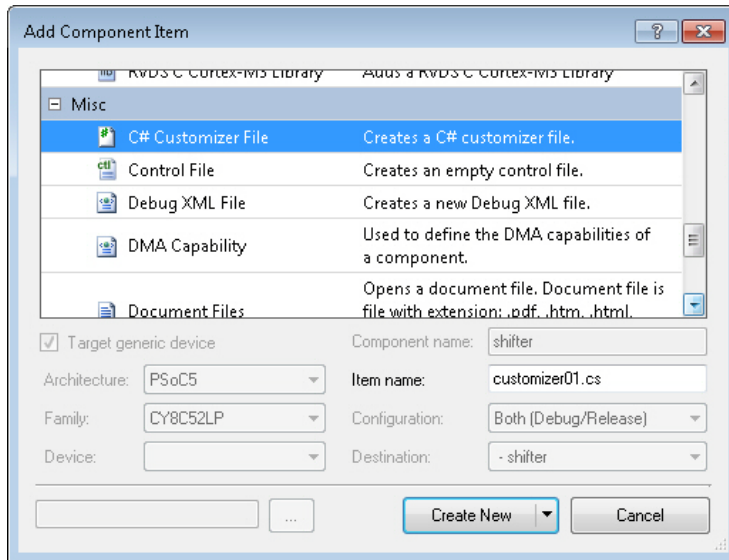
您可以指定在 “Debug”（调试）模式还是在 “Release”（发布）模式下创建自定义程序 DLL，也可以指定编译器的命令行选项。

创建过程中，如果存在任何错误 / 警告信息，它们将显示在 **Notice List**（提示列表）窗口中。由于在运行时创建 DLL，因此源文件控制中不需要保存所创建的自定义程序 DLL。

7.1.3 添加源文件

要想将源文件添加到您的组件内，请执行以下操作：

1. 右键单击组件，然后选择 **Add Component Item**（添加组件项）。这时会出现 **Add Component Item** 对话框。



2. 在 “Misc” 选项下，点击 **C#** 图标。
3. 在 **Target**（目标）选项下，勾选 **Generic Device**（通用器件）。
4. 在 **Item Name**（文件名称）项中键入适当的 **C#** 文件名称。
5. 点击 **Create New**（创建新文件）。

组件项将显示在 **Workspace Explorer** 树的 “Custom”（自定义）子目录内。

使用文本编辑器打开 ‘.cs’ 文件后，您可以对此文件进行编辑。

7.1.4 在 “Custom” 中创建子目录

要想添加一个子目录，请右键单击 “Custom” 目录，然后依次选择 **Add > New Folder**。

注意：创建的所有子目录都会在磁盘上创建。

7.1.5 添加资源文件

您可以将 .NET 资源文件（.resx）添加到自定义程序内。这些文件可以位于“Custom”目录下或所创建的子目录内。可以向每个组件添加多个资源文件。

要想添加资源文件，请执行以下操作：

1. 右键点击目录，然后依次选择 **Add > New Item**。
2. 在 **New Item** 对话框中，点击 **Resx** 文件的图标，然后键入该文件的名称，再点击 **OK**。

7.1.6 命名类别 / 自定义程序

只有一个类别能够实现所需的自定义程序的接口。必须将该类别命名为“CyCustomizer”。该类别所在的命名空间必须以自定义组件的名称结束。例如，如果组件名称为“my_comp”，那么正确的“CyCustomizer”所在命名空间应该为：

```
Foo.Bar.my_comp  
my_comp  
Some.Company.Name.my_comp
```

请参见[位于第 90 页上的使用指南](#)。

7.1.7 指定汇编参考

通过 PSoC Creator，您可以为自定义程序指定其他外部 DLL 的参考。将自动添加下面的程序集：

- "System.dll"
- "System.Data.dll"
- "System.Windows.Forms.dll"
- "System.Drawing.dll"
- "cydsextensions.dll"

要想指定 .NET 参考和其他用户参考，请浏览至程序集所在的目录，然后选中所需项。如果需要指定 .NET 参考，请浏览至：

```
%windir%\Microsoft.NET\Framework\v2.0.50727
```

您同样可以通过程序集引用对话框指定相关的路径。此外，还可以添加指向项目顶级目录（.cydsn）的程序集引用。您需要在相关路径内键入适当的引用。

7.1.8 自定义程序缓存器

自定义程序缓存器是所有编译的自定义程序 DLL 所在的目录。该目录的位置取决于编译 DLL 的可执行文件。例如，如果通过 PSoC Creator 对自定义程序 DLL 进行编译，那么该目录将位于：

```
Documents and Settings/user_name/Local Settings/Application Data/  
Cypress Semiconductor/PSoC Creator/<Release_Dir>/customizer_cache/
```

同样，如果 DLL 在 cydsfit 构建期间被编译，那么目录的位置将为：

```
Documents and Settings/user_name/Local Settings/Application Data/  
Cypress Semiconductor/cydsfit/<Release_Dir>/customizer_cache/
```

7.2 预编译组件自定义程序

预编译组件自定义程序是一项高级特性，大部分组件创建人员不需要使用它。但对于拥有多个组件自定义程序的大型静态库（例如，**PSoC Creator** 提供的库），该特性可以提高性能。

如果需要，可以在设计时重建组件自定义程序。要想检查组件自定义程序是否是最新的，需要检查所有组成文件。加载磁盘上的这些文件并检查它们可能需要花费一段时间。

要想跳过加载和检查这些文件，以及定义需要使用的组件自定义程序的汇编，可以对某个库的组件自定义程序进行预编译。这样会编译组件自定义程序的汇编，将其复制到库目录结构中，然后同库结合起来。预编译的组件自定义程序被称为 ‘_project_.dll’，并显示在项目顶部的 **Workspace Explorer** 中。

注意：由于存在预编译的组件自定义程序，因此不需要使用源文件。如果改动了源文件，那么组件自定义程序会逐步失效。另外，将自动打开预编译的自定义程序，这样可以防止重新命名项目或更新自定义程序。出于这些原因，仅在需要提高性能或在确保组件自定义程序不能改变的情况下，才会使用该特性。

为了避免混淆用户，如果项目中已经存在预编译的自定义程序汇编，那么 **PSoC Creator** 将不会创建新的自定义程序的程序集，因为它始终会优先使用预编译的自定义程序程序集。

要想为某个库创建预编译的组件自定义程序程序集，请执行以下操作：

1. 删除项目中全部现有的预编译的组件自定义程序汇编（请查看下面的步骤）。
2. 将 **Advanced Customizer**（高级自定义程序）工具栏添加到 **PSoC Creator** 中。
3. 点击 **Build and Attach customizer**（创建并添加自定义程序）按键。

要想移除库中预编译的组件自定义程序的程序集，请按照下面流程进行：

1. 从项目中移除名称为 ‘_project_.dll’ 的预编译组件自定义程序的程序集（而不是从磁盘上移除，原因该程序集已被打开，不会从磁盘删除它）。
2. 退出 **PSoC Creator**，然后删除项目文件夹中所有预编译的组件自定义程序的程序集。

7.3 使用指南

7.3.1 使用不同的命名空间

命名空间的 **leafname** 确定自定义程序源文件适合的组件。如果使用资源文件（.resx），它们将使用相同的名称来确定编译的资源文件。

当重新命名或导入组件时，自定义程序源文件内命名空间中的所有实例均被替换。

为了避免源代码中发生意外替换情况，并保证允许源代码共享（其中一个组件可以使用同一库中其他组件内定义的多个通用工具子程序），库中所有组件应该具有相同明显的前缀，例如：

```
Some.Company.Name.
```

7.3.2 使用不同的外部依赖属性

根据程序集名称（而不是根据文件位置），‘.NET’将解决外部程序集依赖属性。因此，如果两个程序集文件具有相同的名称，可能会引起混淆。为了避免外部依赖属性引起的问题，请确保每个外部程序集引用只具有一个特殊的名称。根据经验总结，‘.NET’能够更好地区分各个程序集。

7.3.3 使用通用的组件共享代码

如果两个自定义程序（例如，A 和 B）需要共享代码，请创建一个名称为 ‘Common’ 的新组件，用于保存共享代码。当然，如果您导入了 A 或者是 B，您还需要导入 Common 组件。

7.4 自定义示例：

请参见下面目录中的示例项目：

```
<Install Dir>\examples\customizers\SimpleDialogCustomizer.cydsn\
```

7.5 接口

通过自定义操作，可以定义以下两组接口：

- 由组件实现的自定义接口
- 由自定义接口使用的系统接口

通过将版本编号放在名称后面，对接口进行命名。这样可以升级接口，而不会破坏现有的自定义程序。

下面各节列出并总结了各种自定义接口。要想得到高级参考文档，请参考 *PSoC Creator 自定义 API 参考指南*（位于 *组件创建指南* 目录中的 *customizer_api.chm* 文件）。

7.5.1 由组件实现的接口

该部分介绍了由组件创建人员实现的自定义接口。

每次调用用户提供的自定义程序的实例时，都会创建一个新的实例。客户端不能使用该对象来连续进行调用信息。如果需要连续数据，可以将这些数据存储在实例中隐藏的参数内。

下表提供了自定义接口的简要总结。

表 7-1. 自定义接口总结

接口	说明
ICyAPICustomizeArgs_v2	通过该接口，用于执行 ICyAPICustomize_v2 的各个组件可以访问其他接口。
ICyAPIFitterQuery_v2	该接口将调试生成的信息（例如 :cyfitter.h 头文件中包含的信息）提供给组件。如果使用该接口，通过 ICyAPICustomize_v2 接口可以生成更好的代码。
ICyDesignEntryPrefs_v1	通过该接口，可以将 PSoC Creator 偏好信息（即 ICyInstQuery_v1、ICyDesignQuery_v1 和 ICyTerminalQuery_v1 接口的选择属性）提供给组件。例如，这样可以使组件与用户选择的“数字线”和“模拟线”的颜色设置相匹配。
ICyDeviceQuery_v1	该接口允许通过使用 ICyInstQuery_v1 接口的 DeviceQuery 属性来访问选定的器件（和相关信息）。
ICyDRCPProviderArgs_v1	通过该接口，用于执行 ICyDRCPProvider_v1 的各个组件可以访问其他接口。
ICyExprEvalArgs_v2	通过该接口，用于执行 ICyExprEval_v2 的各个组件可以访问其他接口。
ICyExprTypeConverter	通过该接口，执行 ICyExprEval_v1 或 ICyExprEval_v2 的组件可以使用表达式系统其余部分相同的规则和转换规则将表达式的 C# 函数中不透明的“对象”参考转换为具体的 C# 类型（如整数、浮点数和字符串）。
ICyInstEdit_v1	通过该接口，组件可以修改某个实例中正式参数内的表达式。该接口主要由执行 ICyParamEditHook_v1 的各个组件使用。这些组件同样不能使用局部参数。它们也不能添加 / 移除参数。
ICyInstQuery_v1	通过该接口，组件可以查询某个实例中参数内的值。将该接口作为参数提供给几乎所有自定义程序接口。
ICyInstValidate_v1	通过该接口，执行 ICyInstValidateHook_v1 的组件可以将表达式的验证错误通知给 PSoC Creator。
ICyResourceTypes_v1	通过该接口，可以为执行 ICyImplementationSelector_v1 的组件提供受多种自动实现选择系统支持的强制器件资源。
ICySymbolShapeEdit_v1	通过该接口，执行 ICyShapeCustomize_v1 接口的组件可以修改它们的图形外观（除通过 ICyTerminalEdit_v1 接口处理的终端外）。
ICySymbolShapeQuery_v1	通过该接口，用于执行 ICyShapeCustomize_v1 接口的各组件可以访问当前的图形外观。
ICyTerminalEdit_v1	通过该接口，执行 ICyShapeCustomize_v1 接口的组件可以添加和移除终端。
ICyTerminalQuery_v1	通过该接口，执行 ICyShapeCustomize_v1 的组件可以访问一组当前终端。此外，还可以访问终端信息（如连接到终端的时钟频率）。
ICyVoltageQuery_v1	通过该接口，组件可以查询用户设置在设计范围资源编辑器内的电压设置。可以使用 ICyInstQuery_v1 接口上的 VoltageQuery 属性访问该接口。

7.5.2 PSoC Creator 提供的接口

本部分介绍了 PSoC Creator 提供并且由自定义接口使用的系统接口。

PSoC Creator 实现了系统接口，允许使用本地 C# 和 .Net 类型访问 PSoC Creator 内部数据结构，从而提供稳定简单的接口。这些接口提供了用来读 / 写参数值、寻找实例信息以及发现连接信息的各种方法。

通常，有两种系统接口：query（查询）和 edit（编辑）。通过 Query 接口，可以查询 PSoC Creator 中的所有内容，但不能修改它们。通过 Edit 接口，自定义接口可以更新 / 修改组件的各方面内容。

下表包含了系统接口的简要总结。

表 7-2. 系统接口概要

接口	说明
ICyAPICustomize_v1	该接口已经由 ICyAPICustomize_v2 替换。通过该接口，组件可以完全控制固件 API 的生成。
ICyAPICustomize_v2	通过该接口，组件可以完全控制固件 API 的生成。
ICyClockDataProvider_v1	通过该接口，用于创建或处理时钟的组件可以为系统中其他组件提供与时钟相关的信息。
ICyDesignClient_v1	该接口已经由 ICyDesignClient_v2 替换。通过该接口，可以声明组件对 “设计范围” 信息（如时钟频率）十分敏感。PSoC Creator 通过该信息来决定以何种频率来更新原理图上的组件工具提示和外形。
ICyDesignClient_v2	通过该接口，可以表明组件对 “设计范围” 信息（如时钟频率）十分敏感。PSoC Creator 使用该信息来决定以何种频率来更新原理图上的组件工具提示和外形。
ICyDRCProvider_v1	通过该接口，组件可以自己执行设计规则检查（DRC）操作。PSoC Creator Notice List 窗口显示了所有已生成的 DRC 以及其他错误、警告和注意事项。
ICyExprEval_v1	该接口已被 ICyExprEval_v2 替换。通过该接口，组件可以实现 C# 语言函数。这些函数可用于各个表达式的参数、验证器和注释。因此，可以使用 C# 实现复杂或性能重要的表达形式。
ICyExprEval_v2	通过该接口，组件可以实现使用 C# 语言函数。这些函数可用于各个表达式的参数、验证器和注释。因此，可以使用 C# 实现复杂或性能重要的表达形式。
ICyImplementationSelector_v1	通过该接口，组件可以支持自动实现选择。组件要求使用受限制的资源（如固定功能定时器 - 计数器 - PWM 模块）。该请求可以是必须的，也可以是可选的。PSoC Creator 将优先为设计分配重要的资源，并通知组件已经完成的请求。
ICyInstValidateHook_v1	通过该接口，组件可以轻松验证它们的 C# 代码的参数值。除了基于验证器的传统表达式外，还运行了基于 C# 的验证器。

表 7-2. 系统接口概要 (续)

接口	说明
ICyParamEditHook_v1	通过该接口，组件可以完全控制 “Customize...” (自定义 ...) 操作。组件可以为 PSoC Creator 组件的默认配置对话框提供备用的控件，或者提供新的对话框。
ICyShapeCustomize_v1	通过该接口，组件可以完全控制它们在原理图上的外形。通过 ICySymbolShapeEdit_v1 和 ICyTermEdit_v1 接口，组件可以控制它们的图形外观。
ICyToolTipCustomize_v1	通过该接口，组件可以完全控制它们在工具上的文本提示。
ICyVerilogCustomize_v1	通过该接口，组件可以完全控制它们的 Verilog 执行情况。

7.5.3 自定义程序中的时钟查询

PSoC Creator 为组件创建者提供了一个强大的机制，它可以提供和查询与时钟相关的信息。该机制是通过一组自定义接口提供的。

7.5.3.1 ICyTerminalQuery_v1

包括两种名为 **GetClockData** 的方法。其中一种方法使用了两个参数（一个终端名称和一个索引），并返回驱动终端的时钟频率。另一种方法则使用了“实例路径”、一个终端名称和一个索引，并返回隐藏时钟的时钟频率。

7.5.3.2 ICyClockDataProvider_v1

生成或处理时钟的组件可以将这些时钟的频率信息提供给 PSoC Creator。而 **GetClockData** 方法可以自动访问这些信息。

7.5.4 时钟 API 支持

某些组件 API 需要访问初始的时钟配置，从而能够正确地进行操作（例如，I2C）。PSoC Creator 将通过 *cyfitter.h* 内的 **#defines** 列出以下的初始配置：

```
#define BCLK__BUS_CLK__HZ      value
#define BCLK__BUS_CLK__KZ      value
#define BCLK__BUS_CLK__MHZ     value
```

8. 添加调试支持



调试支持是用于少数组件（如 **CapSense®**）的高级功能。通过该功能，用户可以对设计中的组件进行调试。

进行调试时，固件组件要与主机应用进行通信。硬件将扫描值发送到 GUI，然后 GUI 再将已更新的调试参数发回固件。这是一个迭代过程，用以得到器件的最佳结果。

PSoC Creator 为组件创建者提供了一套可用于调试组件的灵活的 API。

8.1 调试框架

所有组件都能对这个通用调试框架进行设置，用来支持调试功能。该框架主要包括两个 API。一个 API 是由组件创建者执行的，作为组件调试器的启动点。另一个 API 由 PSoC Creator 执行，它提供了调试器与 PSoC 器件通信时所使用的通信机制。

通过最终用户设计中的 I²C 或 EZ I²C 组件，可以与器件进行通信。

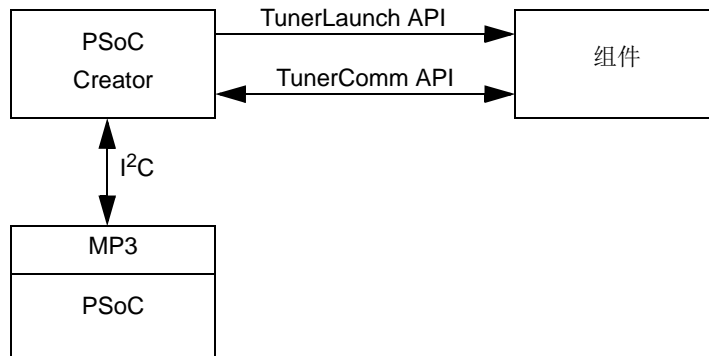
通常通过一个用于显示各个输入的扫描值，以及允许用户设置新的调试参数值的 GUI 可以实现调试操作。更改参数后，随即产生影响。

调试应用在一个安装了 PSoC Creator 的 PC 上运行。它显示了来自 PSoC 器件的各个值。根据用户要求，可将各个参数值从 PC 写入到芯片内。

用户需要通过使用支持调试框架和可调试组件的通信协议对这个双向的通信通道进行设置。PSoC Creator 不能自动设置用户设计的通信通道，因为它不能确定可用通道或者用户将通信组件使用于其他目的的方式。

8.2 架构

下面的框图显示了 PSoC Creator 的调试框架。PSoC Creator 使用 TunerLaunch API 启动组件调试。该组件的调试器使用 TuningComm API 对器件进行读 / 写操作。PSoC Creator 通过 I²C 和 MiniProg3 与 PSoC 器件进行通信。



8.3 调试 API

PSoC Creator 为组件创建者定义了两个用于调试的 API，分别为 LaunchTuner 和 TunerComm。

下面各节列出并总结了各种调试接口。要想查找深入分析文档，请参考 *PSoC Creator 调试 API 参考指南*（`tuner_api.chm` 文件，它与本 *组件创建指南* 位于同一个目录中）。

8.3.1 LaunchTuner API（启动调试器 API）

LaunchTuner API 由组件实现。当用户选中了组件实例中的 **Launch Tuner** 项时，PSoC Creator 将调用该 API。所有实现 LaunchTuner API 的组件均被视为可调试组件。PSoC Creator 会将实现 TunerComm API 的通信对象传输到调试器内。调试器使用该对象对器件执行读 / 写操作。

PSoC Creator 以非模式窗口格式启动调试器。这样在调试器运行期间，用户可以执行其他操作（如进行调试（debug））。

8.3.2 通信 API（ICyTunerCommAPI_v1）

通信 API 定义了一组函数。通过使用这些函数，调试器 GUI 不用知道正在使用的通信机制信息，但还能够与固件组件通信。启动调试器的应用针对所需通信协议分别执行每一个函数。

TunerComm API 由 PSoC Creator 实现。通过该函数，调试器可以使用一个受支持的调试通信通道与 PSoC 器件进行通信。

TunerComm API 作为调试器 GUI 和芯片上组件固件间的数据通信通道使用。PSoC Creator 并不知道用于通信的数据内容。组件调试器需要确保 GUI 和固件间传输的数据内容相互匹配。

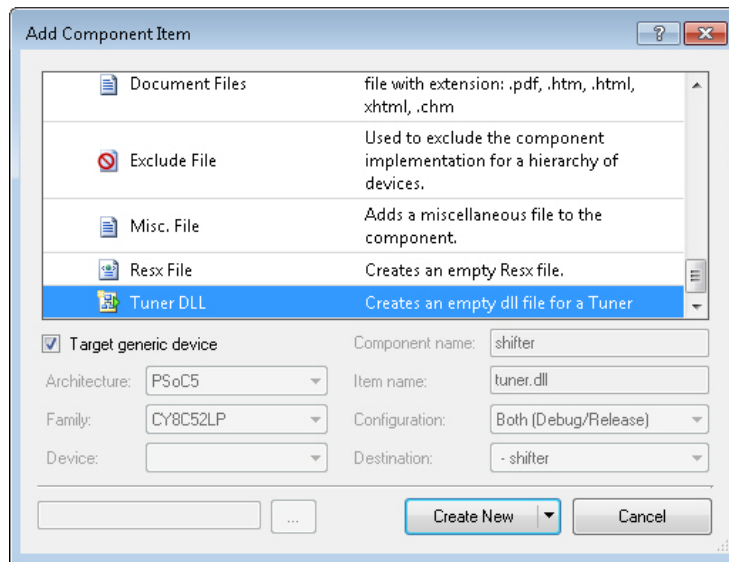
8.4 传输参数

通过使用 **CyTunerParams** 类型（即用于名称 / 数值对的一个简单包装类），可以在 PSoC Creator 和调试器间传输参数。

8.5 组件调试器 DLL

组件中应该包含作为调试器 DLL 的组件调试器代码。请使用 **Add Component Item**（添加组件项）对话框将一个“Tuner DLL”文件添加到组件内。

注意：您向一个组件内只能添加一个调试器 DLL 文件。



PSoC Creator 不支持基于源代码的调试器。因此，不能在工具中编译调试器，而是需要将它作为一个 DLL 使用。

8.6 通信设置

如本章节前面所述，调试器和器件之间的通信是由组件的最终用户设置的。组件调试器并不确定该通信的设置情况。因此，为了支持该功能，**TunerComm API** 提供了一个设置方法。组件调试器需要通过调用该方法来启动通信配置。

请参考 *PSoC Creator 调试 API 参考指南*（**tuner_api.chm** 文件，与 *组件创建指南* 位于同一个目录中）。

8.7 启动调试器

PSoC Creator 将提供一个用于启动组件实例中调试器的菜单项。该菜单项仅在组件实例的上下文菜单中有效。因为只有特定的可调试组件实例中，该菜单项才能发挥它的作用。

8.8 固件交警（Firmware Traffic Cop）

用户的设计中包含了一个用于实现用户固件设计的 *main.c* 文件。为支持调试操作，用户的 *main* 函数必须包含“Traffic Cop”代码，以便为 I²C 和可调式组件间的数据传输提供便利。下面的伪代码显示的是一个 *main.c* 示例。

```
main() {

    struct tuning_data {
        // this data is opaque to PSoC Creator but known to the component
    } tuning_data;

    I2C_Start();           // Start the communication component
    CapSense_Start();      // Start the tunable component

    I2C_SetBuffer(tuning_data); // Configure I2C to manage buffer

    for (;;) {

        // If there the data from the last push has been consumed
        // by the PC then go ahead and fill the buffer again.

        if (buffer read complete) {
            CapSense_FillBuffer(tuning_data);
        }

        // If the PC has completed writing new parameter values
        // then pass those along to the component for processing

        if (buffer write complete) {
            CapSense_ProcessParameters(tuning_data);
        }
    }
}
```

这个 *main* 函数需要通过某种方法使芯片和主机 PC 间进行的读和写操作实现同步。该方法可以是使用缓冲区中的某个数据字节来使能读和写操作。

8.9 组件更改

要想支持调试功能，必须创建组件的固件和各个 API。可能用户不想在他们的最终设计中增加调试支持的开销。因此，需要在组件配置对话框中为用户提供一个用于将组件置于调试模式的选项。组件进入调试模式后，必须构建用户设计并将其下载到器件内。

8.9.1 通信数据

通过一个共享的 C 结构对来自或者传输到芯片的数据进行通信。该 C 结构包含了扫描值和调试参数值等字段。它的正确结构由组件在构建时决定。在调试模式下所构建的组件会识别该结构，以便能够正确地处理各个参数。

8.10 简单的调试器

用于实现调试器的类别完全可以对该调试器的工作方式进行控制。下面介绍的是一个调试器的示例代码（更多信息，请参考 API 文档）：

```
public class MyTuner : CyTunerProviderBase
{
    MyForm m_form;    // a custom GUI

    public MyTuner() // Tuner constructor just creates the form
    {
        m_form = new MyForm();
    }

    public override void LaunchTuner(CyTunerParams params,
        ICyTunerComm comm, CyTunerGUIMode mode)
    {
        m_form.InitParams(params); // Set up the form
        m_form.SetComm(comm);

        // When this method exists, LaunchTuner returns to Creator
        m_form.ShowDialog();
    }

    // Creator calls this method to retrieve the new
    // values of all parameters
    public override CyTunerParams GetParameters()
    {
        return(m_form.GetParameters());
    }
}
```

在该代码中，调试器创建了一个自定义 GUI（MyForm），然后在该界面中调用一个显示对话框。由于该调试器正在执行自己的线程，因此它可以全面调用阻塞函数（即 ShowDialog()）而不会挂起 PSoC Creator 的操作。

PSoC Creator 将调用 LaunchTuner 方法，然后等待它返回。调用该方法被返回后，PSoC Creator 会立即调用 GetParameters()，以获取新的参数值，然后再将该值存储到组件实例中。

9. 添加 Bootloader 支持



该章节介绍了如何为组件提供 Bootloader 支持。有关 PSoC Creator Bootloader 系统的详细信息，请参考 *PSoC Creator 系统参考指南*。

为了给组件提供 Bootloader 支持，需要更改下面两个通用区域：

- 固件
- 自定义 Bootloader 接口

9.1 固件

对于支持引导加载的组件，需要执行节 9.1.2 所描述的五個函数。Bootloader 使用这些函数来设置通信接口，并为主机传输（发送 / 接收）数据包。

有关源代码和执行函数的详细信息，请参考第 81 页上的添加 API 文件章节。

9.1.1 保护

因为在设计中，多种 Bootloader 可以同时支持多个组件，因此每当执行所需 Bootloader 函数时，需要通过预处理的有条件指令对所有函数进行保护。该保护执行如下：

```
#if defined(CYDEV_BOOTLOADER_IO_COMP) &&
    (CYDEV_BOOTLOADER_IO_COMP == CyBtldr_`@INSTANCE_NAME`)

    //Bootloader code

#endif
```

9.1.2 函数

为了支持引导加载，要在组件内执行以下函数：

- CyBtldrCommStart
- CyBtldrCommStop
- CyBtldrCommReset
- CyBtldrCommWrite
- CyBtldrCommRead

以下章节提供了函数定义。

9.1.2.1 *void CyBtldrCommStart(void)*

说明： 该函数会启动已选中的通信组件。在多种情况下，仅需要调用现有的函数 ``@INSTANCE_NAME`_Start()`

参数： 无

返回值： 无

其他影响： 启动通信组件，并执行所需配置，这样才能使 PSoC 读取和 / 或写入数据。

9.1.2.2 *void CyBtldrCommStop(void)*

说明： 该函数会停止已选的通信组件。在多种情况下，只要调用现有的函数 ``@INSTANCE_NAME`_Stop()`

参数： 无

返回值： 无

其他影响： 停止通信组件，并执行所需的释放操作，以禁用通信组件。

9.1.2.3 *void CyBtldrCommReset(void)*

说明： 强制已选的通信组件删除过时数据。当指令被中断或损坏时，需要使用该函数来清除组件的当前状态，并重新启动组件。

参数： 无

返回值： 无

其他影响： 清除通信组件中所有的缓存数据，并设置组件的状态为读 / 写新指令。

9.1.2.4 *cystatus CyBtldrCommWrite(uint8 *data, uint16 size, uint16 *count, uint8 timeOut)*

说明： 要求提供从输入数据缓冲区写入到主机器件内的字节数。一旦完成写操作，被写入的字节数将被该数值更新掉。通过使用 **timeOut**（超时）参数来提供函数的最大可运行时间。如果写操作完成的比较早，那么该函数会尽快返回成功代码。如果在指定的时间内未完成写操作，则它将返回一条“错误”信息。

参数： **uint8 *data** — 指向缓冲区（包含被写入的数据）的指针
uint16 size — 从数据缓冲区写入的字节数
uint16 *count — 指针指向通信组件中实际写入的字节数所在的位置
uint8 timeOut — 指出通信超时前，通信组件需要等待的时间（一个单位表示 10 毫秒）

返回值： 如果成功写入了一个或多个字节，则返回 **CYRET_SUCCESS**。如果在 10 毫秒 * 超时量（单位为毫秒）的时间内主机控制器未响应写操作，则返回 **CYRET_TIMEOUT**。

其他影响： 无

9.1.2.5 *cystatus CyBtldrCommRead(uint8 *data, uint16 size, uint16 *count, uint8 timeOut)*

- 说明:** 要求提供从主机器件读取并被存储到已提供数据缓冲区内的字节数。一旦完成写操作，被写入的字节数将被该数值更新掉。通过使用 **timeOut**（超时）参数来提供函数的最大可运行时间。如果过早完成读操作，则该函数将尽快返回成功代码。如果在指定的时间内未完成读操作，它将返回一条“错误”信息。
- 参数:** **uint8 *data** — 指向缓冲区（该缓冲区用于存储来自主机控制器的数据）的指针
uint16 size — 在数据缓冲区中读取的字节数
uint16 *count — 指针指向通信组件将写入实际读取的字节数所在的位置
uint8 timeOut — 指出通信超时前，通信组件需要等待的时间（一个单位表示 10 毫秒）
- 返回值:** 如果成功地读取了一个或多个字节，则返回 **CYRET_SUCCESS**。如果在 10 毫秒 * 超时（单位为毫秒）的时间内主机控制器未响应读操作，则返回 **CYRET_TIMEOUT**。
- 其他影响:** 无

9.1.3 自定义 Bootloader 接口

Bootloader 需要使用配置通信组件，以便对 PSoC 器件执行传入或传出数据。对于可以配置为满足该要求的组件，存在一个由定制器执行的接口，该接口会通知 PSoC Creator，如下：

■ ICyBootLoaderSupport

在 *PSoC Creator 自定义 API 参考指南*（*customizer_api.chm* 文件，与 *组件创建指南* 位于同一个目录下）中的“通用接口”部分已经对该接口进行了详细的描述。它只包含一个函数，Bootloader 使用该函数来确定组件当前的配置是否与 Bootloader 相兼容。

```
public interface ICyBootLoaderSupport
{
    CyCustErr IsBootloaderReady(ICyInstQuery_v1 inst);
}
```

当组件与 Bootloader 相兼容时，放置在设计中的所有实例都将在设计范围资源系统编辑器中作为 Bootloader IO 组件的一个选项显示。在定制器中执行的单个函数只需要验证组件当前的配置情况，以确保设置同双向通信相兼容。

有关定制器的更多信息，请参考[第 87 页上的自定义组件章节](#)。

10. 完成创建组件



本章节将介绍有关完成创建组件过程的各个步骤，具体如下：

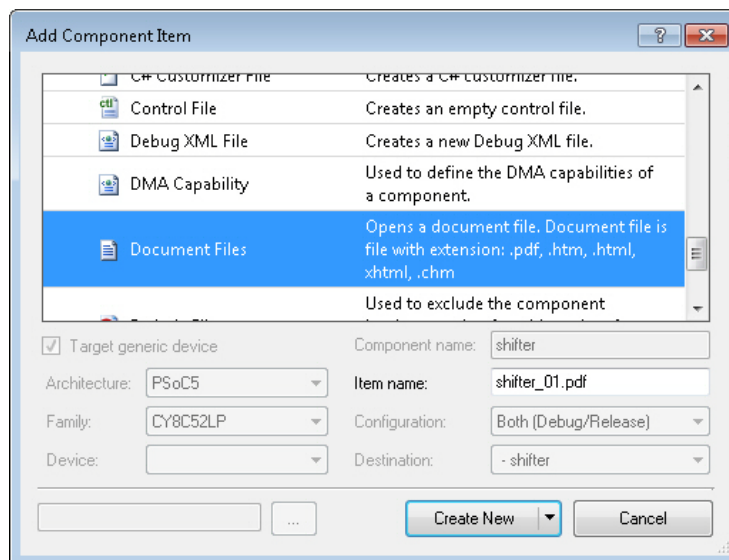
- 添加 / 创建数据手册
- 添加控制文件
- 添加 / 创建 Debug XML（调试 XML）文件
- 添加 / 创建 DMA 功能文件
- 添加 / 创建 .cystate XML 文件
- 添加静态库
- 添加依赖属性
- 编译项目

10.1 添加 / 创建数据手册

您可以将格式为 PDF、HTML 和 XML 的文档添加到您的组件内。不过，组件的数据手册必须是 PDF 格式的文件。当用户使用工具打开数据手册时，始终会打开与该组件相关的 PDF 文件。可以使用 MS Word、FrameMaker 或任何其它类型的文字处理软件来创建组件数据手册。并且，您可以将该文件转换为 PDF 格式，并将其添加到组件内。主要的要求是该 PDF 文件必须与组件同名，以便从组件目录中查看。

1. 右击该组件，然后选择 **Add Component Item**（添加组件项）项。

这时会出现 Add Component Item 对话框。

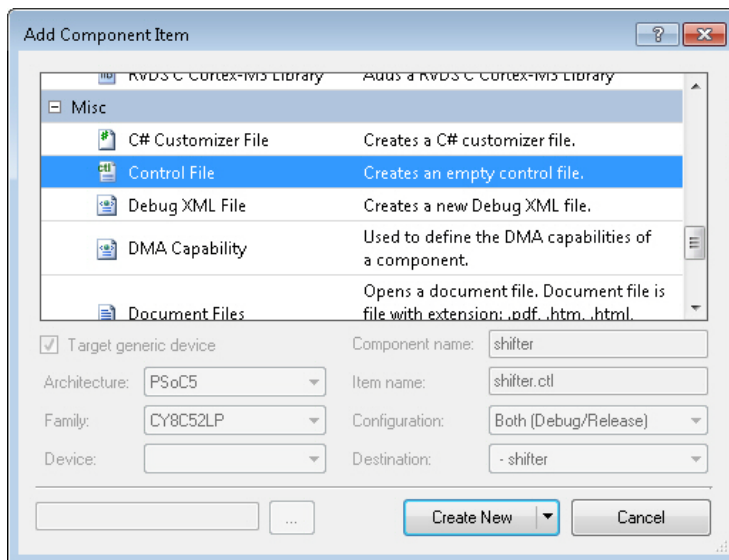


- 从模板的 **Misc** 类别中选择 **Document Files**（文档文件）图标，然后向 **Item name**（项名称）中键入 PDF 文件名称。
注意： 这个程序适用于需要添加到组件的任何其它文件。要想添加不同的文件类型，请点击不同的模板图标。
- 点击 **Create New**（创建新文件），使 PSoC Creator 能够创建一个虚拟文件；从下拉菜单中选择 **Add Existing**（添加现有文件），以选择现有文件并将其添加到组件内。
该项将显示在 **Workspace Explorer** 中，并在 PSoC Creator 外打开。
- 若适用，可以从项目中所有文件内复制任意的更新内容。

10.2 添加控制文件

控制文件用于定义组件示例的固定位置特性。通过使用该文件，组件创建者能够定义目标数据路径、PLD 和状态 / 控制寄存器资源。更多有关控制文件的信息，请参考“编译 PSoC Creator 项目”章节中的“PSoC Creator 帮助”部分。按照下面各步骤操作，以添加一个控制文件：

- 右击该组件，然后选择 **Add Component Item**（添加组件项）项。
这时会出现 **Add Component Item** 对话框。



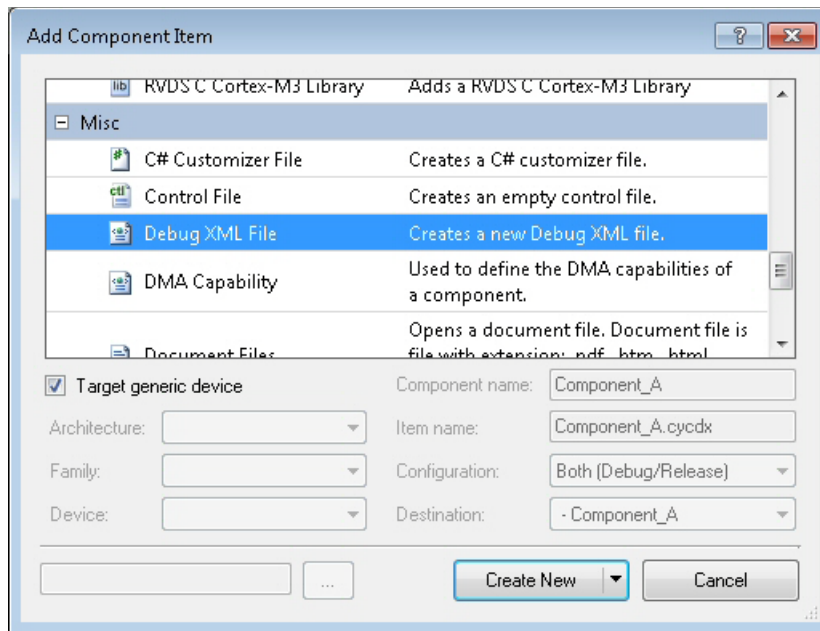
- 从模板的 **Misc** 类别中选择 **Control File**（控制文件）图标。其名称将与组件的名称相同。
- 点击 **Create New**（创建新文件），使 PSoC Creator 能够创建一个空白文件；从下拉菜单中选择 **Add Existing**（添加现有文件），以选择将现有文件添加到组件内。
该项将显示在 **Workspace Explorer** 中，并作为选项卡式的文档在工作区中打开。

当用户从 **Component Configure**（组件配置）对话框中使能了该特性，组件示例将被强制到特定的资源内。许多示例不能使用此特性，如果尝试使用它，将生成错误。

10.3 添加 / 创建 Debug XML（调试 XML）文件

PSoC Creator 提供了一个可选的机制为此组件创建新的调试器工具窗口。为了在特殊组件上使能该特性，您需要添加该组件的 XML 说明，说明内容包括已使用的任何存储器模块的信息或该组件的重要单独寄存器。该调试文件将使用于 PSoC Creator 组件调试窗口。欲了解如何使用 Component Debug（组件调试）窗口，请参考 PSoC Creator 帮助。

1. 右击该组件，然后选择 Add Component Item（添加组件项）项。
这时会出现 Add Component Item 对话框。
2. Component_A.cycdx 从模板的 Misc 类别中选择 Debug XML File（调试 XML 文件）图标；其名称将与该组件的名称相同，在这里它为 Component_A.cycdx。



3. 选择 Add Existing（添加现有文件）以选择现有文件并将其添加到组件内；点击 Create New（创建新文件）使 PSoC Creator 能够创建一个虚拟文件。

该项将显示在 Workspace Explorer 中，并作为选项卡式的文档在 PSoC Creator 代码编辑器中打开。

10.3.1 XML 格式

以下是 debug XML（调试 XML）文件的关键元素和属性。列在这里的项目包含了有助于创建调试文件的所有东西，而完整的架构定义可以在 PSoC Creator 安装的以下目录中找到：

```
<install location>\PSoC Creator\<Version#\PSoC Creator\tdt\cyblockregmap.xsd
```

同组件中的其他文件一样，调试 XML 文件能使用模板扩展（请参考节 6.1.3）在编译过程中评估参数。此外，地址必须是一个实际地址或属于 cydevice.h、cydevice_trm.h 或 cyfitter.h 的 #define。

注意：XML 文档中的各项名称不能包含空格。

模块

<block> 字段用于包装单个模块中的相关寄存器 / 存储器。一个模块可以是一个组件实例、组件部分（如 UDB）、子组件或相同寄存器 / 存储器的抽象集合。一个模块必须在层次结构中具有唯一的名称。它可以提供说明和模块的可视化表达式。

参数	类型	说明
name	字符串	定义模块名称。该字符将被添加到子项目内。
desc	字符串	提供了该模块的说明内容。
visible	字符串	指定是否显示模块中的内容（字符串评估为布尔值）。

存储器

<memory> 字段代表组件所使用的存储器中连续部分。存储器部分需要一个唯一的名称、存储器的起始地址和大小。必须将 **<memory>** 放置在 **<block>** 内。

参数	类型	说明
name	字符串	定义存储器部分的名称。
desc	字符串	提供了对存储器部分的说明内容。
BASE	字符串	存储器部分的起始地址（字符串评估为无符号整数值）。
SIZE	字符串	存储器部分中字节的数量（字符串评估为无符号整数值）。

寄存器

<register> 字段用于表示组件内部所使用的寄存器。必须将 **<register>** 放置在 **<block>** 内。

参数	类型	说明
name	字符串	定义寄存器的名称。
desc	字符串	提供了该寄存器的说明内容。
address	字符串	寄存器的地址（字符串评估为无符号整数值）。
size	字符串	寄存器中位的数量（字符串评估为无符号整数值）。
hidden	--	指定是否显示模块中的内容（字符串评估为布尔值）。

字段

<field> 用于定义拥有特殊意义的寄存器部分。它是 **<register>** 的子项目，因此请勿在寄存器外使用它。从组件调试窗口查看时，该定义允许为寄存器的特殊部分指定含义。

参数	类型	说明
name	字符串	定义字段的名称。

参数	类型	说明
desc	字符串	提供了该字段的说明。
from	字符串	字段的高序位（字符串评估为无符号整数值）。
to	字符串	字段的低序位（字符串评估为无符号整数值）。
access	字符串	为各种寄存器如只读（“R”）/只写（“W”）或只读和只写（“RW”）定义各种有效的访问类型。可能的值为： R — 只读 W — 只写 RW — 读 / 写 RCLR — 读取清除 RCLR W — 读取清除或写入 RWCLR — 读取清除或写入清除 RWSET — 读取或通过写入 1 设置 RWZCLR — 读取或通过写入 0 清除

数值

<value> 用于定义具有寄存器字段意义的特殊值。它是 <field> 的子项目。一个数值是可选助手的定义，当在组件调试窗口中打开带有 <field> 和 <value> 的 <register> 时，该定义将显示在工具提示内。通过该定义，可以快速查看特殊字段值的含义。

参数	类型	说明
name	字符串	定义该值的名称。
desc	字符串	提供了对该值的说明内容。
value	字符串	指定二进制字符串的数值，如 “100”、“0101”、“00”。

10.3.2 示例 XML 文件

下面是一个 debug XML（调试 XML）文件的示例。 请注意，必须将所有元素放置在 <block> 内，而且这些元素应该遵循上述限制。该示例包含了一个作为项目中的组件示例的高级模块。然后，通过使用 `FixedFunction` 组件参数，它将显示该组件的 UDB 实现或固定函数实现。

注意：显示在该示例中的地址不是实际地址；它们只作为演示地址使用。

```
<?xml version="1.0" encoding="us-ascii"?>

<deviceData version="1"
  xmlns="http://cypress.com/xsd/cydevicedata"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://cypress.com/xsd/cydevicedata
  cydevicedata.xsd">

  <block name="\$INSTANCE_NAME`" desc="Top level">
    <block name="UDB" desc="UDB registers" visible="!\$FixedFunction`">
      <register name="STATUS" address="\$INSTANCE_NAME`_bUDB__STATUS"
        bitWidth="8" desc="UDB status reg">
```

```

        <field name="GEN" from="7" to="2" access="RW" desc="General
status" />
        <field name="SPE" from="1" to="0" access="R" desc="Specific
status">
            <value name="VALID" value="01" desc="Denotes valid status" />
            <value name="INVALID" value="10" desc="Denotes invalid status"
/>
        </field>
    </register>
    <register name="PERIOD" address="\$INSTANCE_NAME`_bUDB__PERIOD"
bitWidth="16" desc="UDB Period value" />
    <memory name="Memory" address="\$INSTANCE_NAME`_bUDB__BUFFER"
size="32" desc="UDB buffer address" />
</block>
    <block name="FixedFunction" desc="Fixed Function registers"
visible="\$FixedFunction`">
        <register name="STATUS" address="\$INSTANCE_NAME`_bFF__STATUS"
bitWidth="8" desc="FF status reg">
            <field name="SPE" from="0" to="0" access="R" desc="Specific
status">
                <value name="VALID" value="0" desc="Denotes valid status" />
                <value name="INVALID" value="1" desc="Denotes invalid status"
/>
            </field>
        </register>
        <register name="PERIOD" address="\$INSTANCE_NAME`_bFF__PERIOD"
bitWidth="16" desc="FF Period value" />
        <memory name="Memory" address="\$INSTANCE_NAME`_bFF__BUFFER"
size="32" desc="FF buffer address" />
    </block>
</block>
</deviceData>

```

有关其他示例，请参考 PSoC Creator 中提供的组件（例如，I²C）。它们位于以下 PSoC Creator 安装目录中：

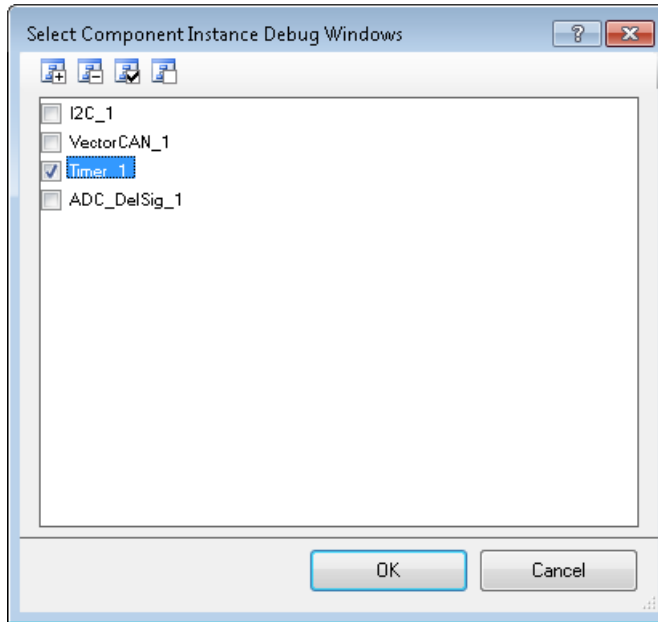
<install location>\PSoC Creator\<Version#>\PSoC Creator\psoc\content

10.3.3 示例窗口

以下部分显示的是可通过 XML 文件生成的不同调试窗口。对于每个显示窗口，XML 文件的相应部分也被显示以指出还需要进行哪个操作。这里的所有示例都来自 **Timer_v2_50** 组件。

10.3.3.1 选择组件实例调试窗口

该窗口是用户为特殊实例选择的调试窗口。



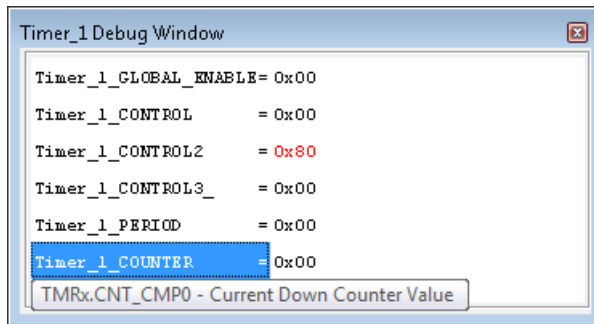
```
<?xml version="1.0" encoding="us-ascii"?>

<deviceData version="1"
  xmlns="http://cypress.com/xsd/cydevicedata"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://cypress.com/xsd/cydevicedata
  cydevicedata.xsd">

  <block name="`$INSTANCE_NAME`" desc="" visible="true">
    ...
  </block>
</deviceData>
```

10.3.3.2 组件实例调试窗口

该窗口显示的是特殊实例的调试信息。



```
<?xml version="1.0" encoding="us-ascii"?>

<deviceData version="1"
  xmlns="http://cypress.com/xsd/cydevicedata"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://cypress.com/xsd/cydevicedata
cydevicedata.xsd">

  <block name="`$INSTANCE_NAME`" desc="" visible="true">

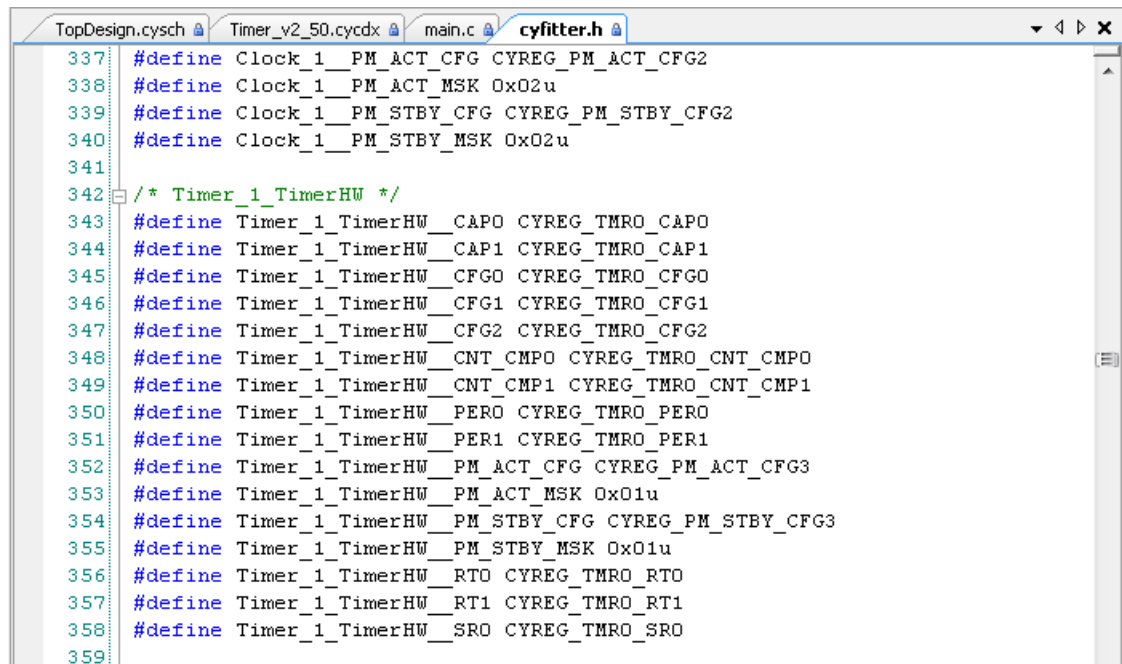
    <block name="`$INSTANCE_NAME`" desc="" visible="`$FF8`">
      <!-- Fixed Function Configuration Specific Registers -->
      <register name="CONTROL"
        address="`$INSTANCE_NAME`_TimerHW__CFG0" bitWidth="8"
        desc="TMRx.CFG0">
        ...
      </register>
      <register name="CONTROL2"
        address="`$INSTANCE_NAME`_TimerHW__CFG1" bitWidth="8"
        desc="TMRx.CFG1">
        ...
      </register>
      <register name="PERIOD"
        address="`$INSTANCE_NAME`_TimerHW__PER0" bitWidth="8"
        desc="TMRx.PER0 - Assigned Period">
      </register>
      <register name="COUNTER"
        address="`$INSTANCE_NAME`_TimerHW__CNT_CMP0" bitWidth="8"
        desc="TMRx.CNT_CMP0 - Current Down Counter Value">
      </register>
      <register name="GLOBAL_ENABLE"
        address="`$INSTANCE_NAME`_TimerHW__PM_ACT_CFG" bitWidth="8"
        desc="PM.ACT.CFG">
        <field name="en_timer" from="3" to="0" access="RW"
        desc="Enable timer/counters.">
        </field>
      </register>
    </block>
```

```

</block>
</deviceData>

```

对于地址字段，该组件使用赛普拉斯模板替换机制允许该地址字段适用于所有组件名称，并允许查找 *cydevice_trm.h* 或 *cyfitter.h* 中定义的任一值。在这种情况下，它使用 *cyfitter.h* 中的定义，因此它的功能将独立于工具完成放置定时器的位置。



```

TopDesign.cysch Timer_v2_50.cycdx main.c cyfitter.h
337 #define Clock_1_PM_ACT_CFG CYREG_PM_ACT_CFG2
338 #define Clock_1_PM_ACT_MSK 0x02u
339 #define Clock_1_PM_STBY_CFG CYREG_PM_STBY_CFG2
340 #define Clock_1_PM_STBY_MSK 0x02u
341
342 /* Timer_1_TimerHW */
343 #define Timer_1_TimerHW_CAPO CYREG_TMRO_CAPO
344 #define Timer_1_TimerHW_CAP1 CYREG_TMRO_CAP1
345 #define Timer_1_TimerHW_CFG0 CYREG_TMRO_CFG0
346 #define Timer_1_TimerHW_CFG1 CYREG_TMRO_CFG1
347 #define Timer_1_TimerHW_CFG2 CYREG_TMRO_CFG2
348 #define Timer_1_TimerHW_CNT_CMPO CYREG_TMRO_CNT_CMPO
349 #define Timer_1_TimerHW_CNT_CMP1 CYREG_TMRO_CNT_CMP1
350 #define Timer_1_TimerHW_PERO CYREG_TMRO_PERO
351 #define Timer_1_TimerHW_PER1 CYREG_TMRO_PER1
352 #define Timer_1_TimerHW_PM_ACT_CFG CYREG_PM_ACT_CFG3
353 #define Timer_1_TimerHW_PM_ACT_MSK 0x01u
354 #define Timer_1_TimerHW_PM_STBY_CFG CYREG_PM_STBY_CFG3
355 #define Timer_1_TimerHW_PM_STBY_MSK 0x01u
356 #define Timer_1_TimerHW_RTO CYREG_TMRO_RTO
357 #define Timer_1_TimerHW_RT1 CYREG_TMRO_RT1
358 #define Timer_1_TimerHW_SRO CYREG_TMRO_SRO
359

```

10.3.4 寄存器窗口

该窗口显示了所选寄存器的详细内容。

Bits	7	6	5	4	3	2	1	0
Access	RW	RW			RW	RW	RW	
Name	HW_EN		CMP_CFG		ROD	COD	TMR_CFG	
Value	0x0		0x0		Reset On Disable (ROD). Resets internal state of output logic			

Restore Commit Close

Bits	7	6	5	4	3	2	1	0
Access	RW	RW			RW	RW	RW	
Name	HW_EN		CMP_CFG		ROD	COD	TMR_CFG	
Value	0x0		0x0		0x0	0x0	0x0	

Restore Commit Close

0 = Equal - Compare Equal
1 = Less than - Compare Less Than
2 = Less than or equal - Compare Less Than or Equal
3 = Greater - Compare Greater Than
4 = Greater than or equal - Compare Greater Than or Equal

```

...
    <block name="\$INSTANCE_NAME\_CONTROL3" desc=""
visible="\$CONTROL3">
    <!-- UDB Parameter Specific Registers -->
    <register name=""
        address="\$INSTANCE_NAME\_TimerHW\_CFG2" bitWidth="8"
desc="TMRx.CFG2">
        <field name="TMR_CFG" from="1" to="0" access="RW"
desc="Timer configuration (MODE = 0): 000 = Continuous; 001 =
Pulsewidth; 010 = Period; 011 = Stop on IRQ">
            <value name="Continuous" value="0" desc="Timer runs
while EN bit of CFG0 register is set to '1'." />
            <value name="Pulsewidth" value="1" desc="Timer runs from
positive to negative edge of TIMEREN." />
            <value name="Period" value="10" desc="Timer runs from
positive to positive edge of TIMEREN." />
            <value name="Irq" value="11" desc="Timer runs until
IRQ." />
        </field>
        <field name="COD" from="2" to="2" access="RW" desc="Clear On
Disable (COD). Clears or gates outputs to zero.">
        </field>
        <field name="ROD" from="3" to="3" access="RW" desc="Reset On
Disable (ROD). Resets internal state of output logic">
        </field>
        <field name="CMP_CFG" from="6" to="4" access="RW"
desc="Comparator configurations">
            <value name="Equal" value="0" desc="Compare Equal " />
            <value name="Less than" value="1" desc="Compare Less
Than " />

```

```

        <value name="Less than or equal" value="10"
desc="Compare Less Than or Equal ." />
        <value name="Greater" value="11" desc="Compare Greater
Than ." />
        <value name="Greater than or equal" value="100"
desc="Compare Greater Than or Equal " />
    </field>
    <field name="HW_EN" from="7" to="7" access="RW" desc="When
set Timer Enable controls counting.">
    </field>
</register>
</block>
...

```

10.4 添加 / 创建 DMA 功能文件

您可以通过 DMA 功能文件使组件使用 PSoC Creator DMA 向导。DMA 向导简单化了 DMA 配置，从而能够更快且更可靠地在源地址和目标地址间传输数据。

10.4.1 将 DMA 功能文件添加到一个组件内：

1. 右击组件，然后选择 **Add Component Item**（添加组件项）项。
这时会出现 Add Component Item 对话框。
2. 从模板的 **Misc** 类别中选择 **DMA Capability**（DMA 功能文件）图标；其名称与该组件的名称相同。
3. 选择 **Add Existing**（添加现有文件），以选择将现有文件添加到组件内；点击 **Create New**（创建新文件），使 PSoC Creator 能够创建一个虚拟文件。
该项将显示在 Workspace Explorer 中，并作为选项卡式文档在 PSoC Creator 代码编辑器中打开。

10.4.2 修改组件头文件：

在组件头文件中生成了可读的源地址和目标地址。该步骤是可选的，但它会通过提供储存器位置的逻辑名称为用户带来更震撼的体验。例如，FIFO 0 储存了组件计算某些内容得到的结果。您可以使用 “<component>_long_name_F0_REG” 来定义在 *cyfitter.h* 文件中所提供的结果，或者您可以为它选择一个更有用的名称，如 “MyComponent_Result_PTR”。用户将在 DMA 向导中查看到 MyComponent_Result_PTR，从而能够更方便使用工具。

如果您拥有作为硬件组成部分的寄存器，那么可以在 *cyfitter.h* 文件中找到寄存器的通用定义。无论 PSoC Creator 将您的组件放置在何处，这些定义始终会更新正确的地址。要想查看您的可用组件，需要构建一个检测项目，将您的组件添加到项目内，然后打开已生成的 *cyfitter.h* 文件。下面是一个示例：

```

/* X_UDB_OffsetMix_1 */
#define X_UDB_OffsetMix_1_OffsetMixer_LSB_16BIT_DP_AUX_CTL_REG CYREG_B0_UDB10_11_ACTL
#define X_UDB_OffsetMix_1_OffsetMixer_LSB_16BIT_F0_REG CYREG_B0_UDB10_11_F0
#define X_UDB_OffsetMix_1_OffsetMixer_LSB_16BIT_F1_REG CYREG_B0_UDB10_11_F1
...

```

您可以使用在头文件中以红色所示的定义来将 X_UDB_OffsetMix_1_OffsetMixer_LSB__16BIT_F0_REG 摘要重新定义为更易懂的内容。例如：

头文件：

```
#define `INSTANCE_NAME`_OUTPUT_PTR ((reg16 *) `INSTANCE_NAME`_OffsetMixer_LSB
__16BIT_F0_REG)
#define `INSTANCE_NAME`_OUTPUT_LOW_PTR ((reg8 *) `INSTANCE_NAME`_OffsetMixer_LSB
__F0_REG)
#define `INSTANCE_NAME`_OUTPUT_HIGH_PTR ((reg8 *) `INSTANCE_NAME`_OffsetMixer_MSB
__F0_REG)
```

现在，您已经拥有了 `INSTANCE_NAME_OUTPUT_PTR` 定义，能够随便使用，不仅可以用于 DMA 向导，而且还可以用于 API 和组件的正常使用。

10.4.3 完成添加 / 创建 DMA 功能文件

DMA 功能文件在开头部分包括了大量注释内容，这些内容有助于理解该文件。

注意： DMA 功能文件中的注释使用 ‘<!--’ 和 ‘-->’ 隔开。例如：

```
<!-- Text between brackets on one line
or more
will be commented out -->
```

DMA 功能文件的开头部分如下：

```
<DMACapability>
  <Category name=""
    enabled=""
    bytes_in_burst=""
    bytes_in_burst_is_strict=""
    spoke_width=""
    inc_addr=""
    each_burst_req_request="">
    <Location name="" enabled="true" direction=""/>
  </Category>
</DMACapability>
```

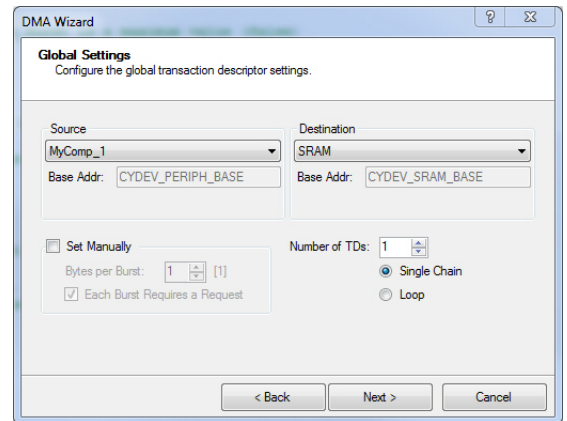
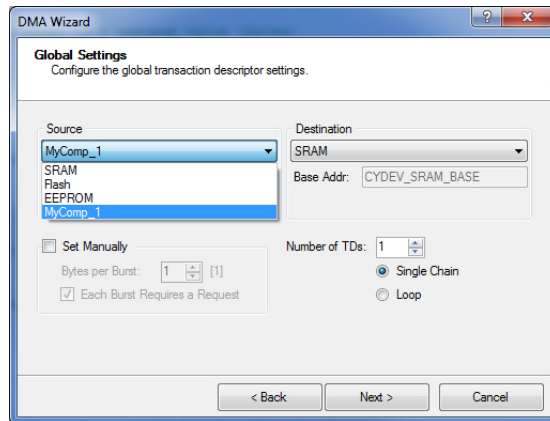
以下章节介绍了每个字段是如何影响到 DMA 向导的：

10.4.3.1 类别名称

您可以使用一个或多个类别。

- 如果您使用了某个类别，当选择源地址和目标地址时，DMA 向导仅会显示您的组件名称。

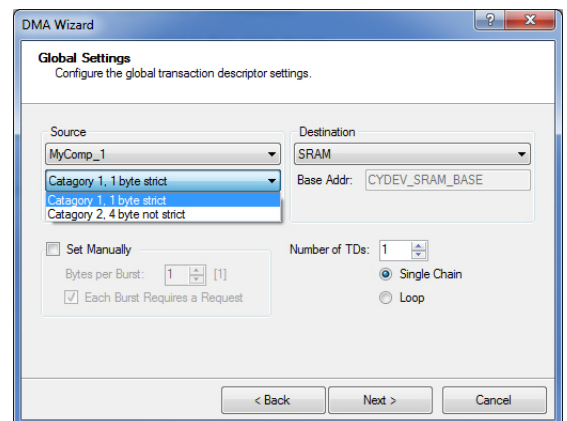
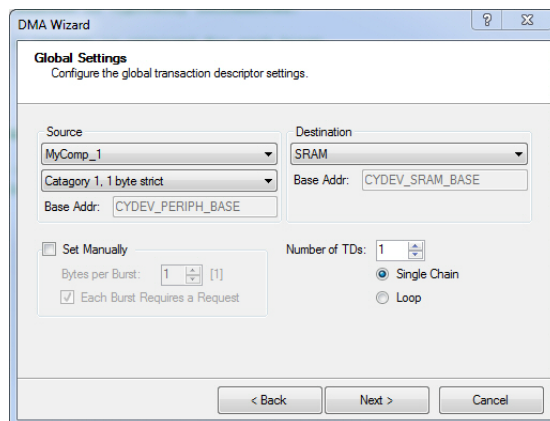
```
<DMACapability>
  <Category name="Category 1, 1 byte strict"
    enabled="true"
    bytes_in_burst="1"
    bytes_in_burst_is_strict="true"
    spoke_width="2"
    inc_addr="false"
    each_burst_req_request="true">
    <Location name="`INSTANCE_NAME`_location_1_cat_1" enabled="true"
      direction="source"/>
  </Category>
</DMACapability>
```



- 如果您使用了多个类别，那么在源地址和目标地址选择您的组件时，另一个下拉菜单将列出可用的类别。

```
<DMACapability>
  <Category name="Catagory 1, 1 byte strict"
    enabled="true"
    bytes_in_burst="1"
    bytes_in_burst_is_strict="true"
    spoke_width="2"
    inc_addr="false"
    each_burst_req_request="true">
    <Location name="`$INSTANCE_NAME`_location_1_cat_1" enabled="true"
      direction="source" />
  </Category>

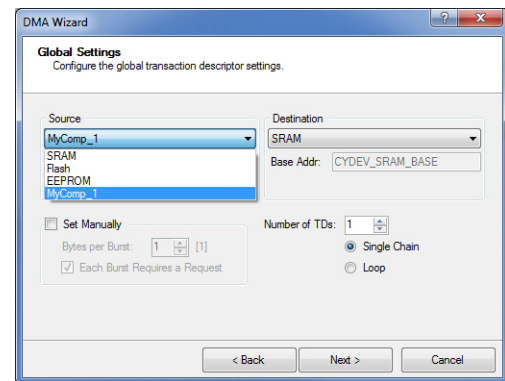
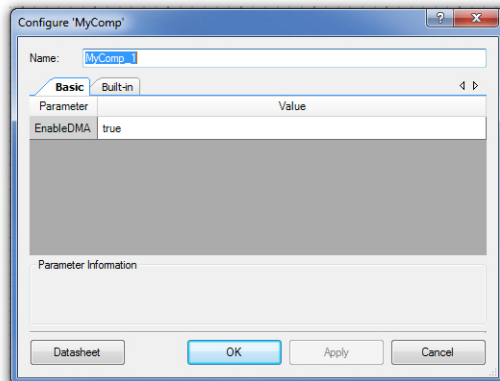
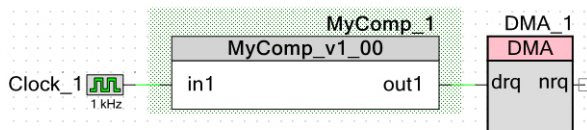
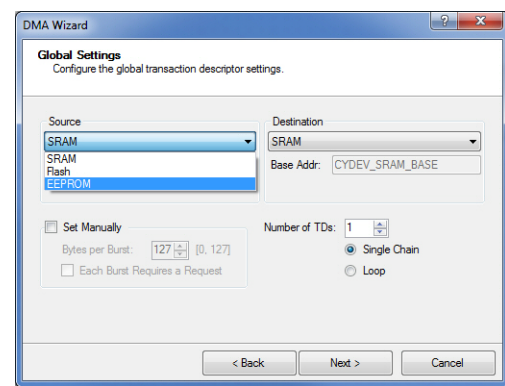
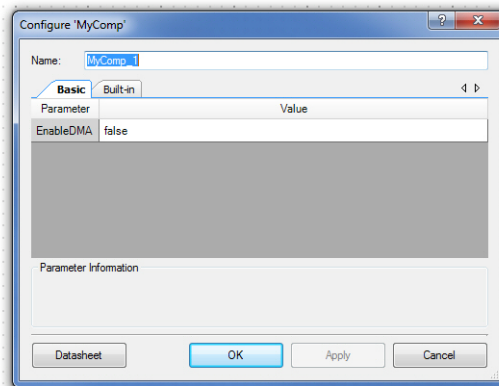
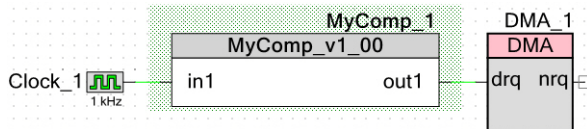
  <Category name="Catagory 2, 4 byte not strict"
    enabled="true"
    bytes_in_burst="4"
    bytes_in_burst_is_strict="false"
    spoke_width="2"
    inc_addr="true"
    each_burst_req_request="true">
    <Location name="`$INSTANCE_NAME`_location_1_cat_2" enabled="true"
      direction="source" />
  </Category>
</DMACapability>
```



10.4.3.2 Enabled（使能）

可将使能字段设置为“真”或“假”，或者可将其作为来自定制器的参数。这样可以根据在定制器中使能或禁用的特性来使能或禁用各类别。句法是指用于替换引号间的内容，该句法使用`=\$YourParameterName` [包括反引号`] 标识。

```
<DMACapability>
  <Category name="Catagory 1, 1 byte strict"
    enabled="`=$EnabledDMA`"
    bytes_in_burst="1"
    bytes_in_burst_is_strict="true"
    spoke_width="2"
    inc_addr="false"
    each_burst_req_request="true">
    <Location name="`$INSTANCE_NAME`_location_1_cat_1" enabled="true"
      direction="source"/>
  </Category>
</DMACapability>
```



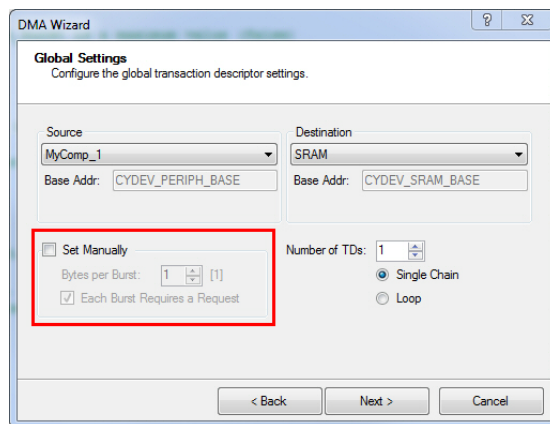
注意：通过使用相同的句法，DMA 功能文件中的任何一个字段都能够使用一个参数替换。您还可以在这些字段中采用 Boolean 表达式来生成更复杂的结果。例如：

```
enabled="`=$EnableExtraDest && $EnableExtraLocations`"
```

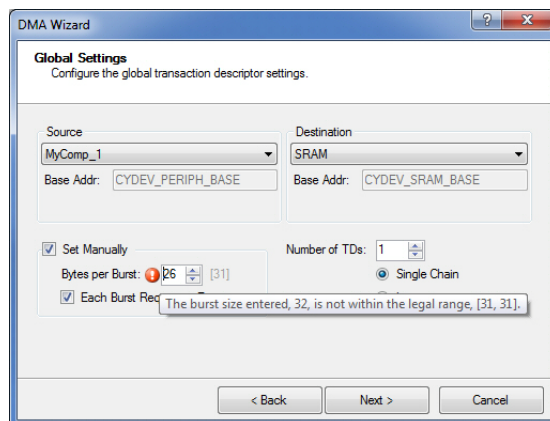
10.4.3.3 Bytes In Burst 参数

在 DMA 向导中，使用 `bytes_in_burst` 参数来设置 “Bytes per Burst”（每次突发的字节）框的初始值，其取值范围为 0 到 127。可将该参数值设置为 “严格要求”（每次突发的字节必须为指定值）或每次突发的最大字节数量。

```
<DMACapability>
  <Category name="Category 1, 1 byte strict"
    enabled="true"
    bytes_in_burst="31"
    bytes_in_burst_is_strict="true"
    spoke_width="2"
    inc_addr="false"
    each_burst_req_request="true">
    <Location name="`$INSTANCE_NAME`_location_1_cat_1" enabled="true"
      direction="source" />
  </Category>
</DMACapability>
```



通过勾选 **Set Manually**（手动设置）选框，用户可以覆盖该设置。但是，如果初始值超过了最大值或不在 “严格要求” 内，那么工具将显示错误。



该设置也可以是自定义程序的参数。使用自定义程序的验证集中的 “uint8” 类型将范围限制为 0 到 127。

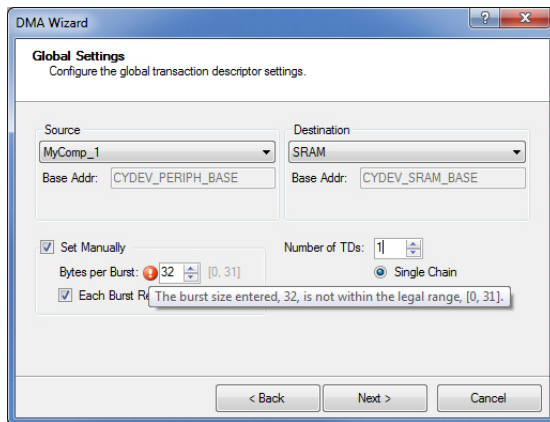
```
bytes_in_burst="`=$Bytes`"
```

10.4.3.4 “Bytes in Burst is Strict” 字段

当 “bytes_in_burst_is_strict” 字段被设置为 “真” 时，该字段将指定 “bytes_in_burst” 是否是所需的值（该值不能与已指定的值有偏差）。当 “bytes_in_burst_is_strict” 字段被设为 “假” 时，“bytes_in_burst” 值被设置为每次突发字节数的上限。

无论严格设置的值如何，用户始终都能够在 DMA 向导中选择 **Set Manually**（手动设置）并覆盖这些设置。但会发出一个警告。

```
<DMACapability>
  <Category name="Category 1, 1 byte strict"
    enabled="true"
    bytes_in_burst="31"
    bytes_in_burst_is_strict="true"
    spoke_width="2"
    inc_addr="false"
    each_burst_req_request="true">
    <Location name="`${INSTANCE_NAME}`_location_1_cat_1" enabled="true"
      direction="source"/>
  </Category>
</DMACapability>
```



10.4.3.5 多层并行访问路径的宽度

“spoke_width” 字段是源 / 目标寄存器所占用的多层并行访问路径的宽度（单位为字节）。建议设置与源 / 目标寄存器相关的多层并行访问路径的宽度。多层并行访问路径的宽度可以在标头 PHUB 和 DMAC 的 TRM 中查到。以下表格是从 TRM 中复制得到的：

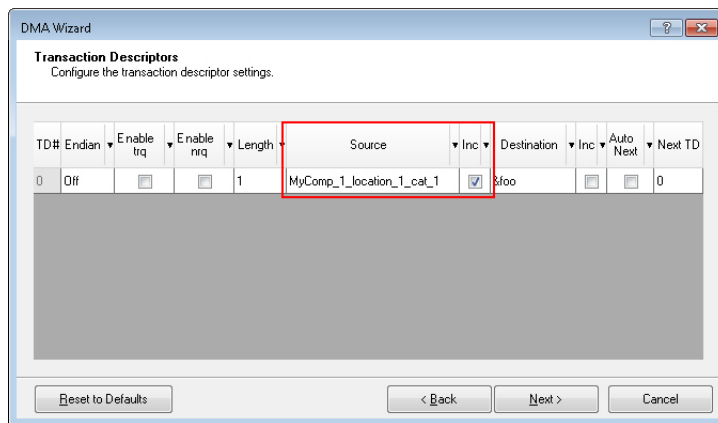
Spoke	Address Width (in bits)	Data Width (in bits)	Peripheral Names
0	14	32	SRAM
1	9	16	I/O interface, port interrupt control unit (PICU), external memory interface (EMIF)
2	19	32	PHUB local spoke, power management, clock, serial wire viewer (SWV), EEPROM
3	11	16	Delta-sigma ADC, analog interface
4	10	16	USB, CAN, fixed-function I ² C, fixed-function timers
5	11	32	Digital filter block (DFB)
6	17	16	UDB set 0 registers (including DSI, configuration, and control registers), UDB interface
7	17	16	UDB set 1 registers (including DSI, configuration, and control registers)

```
<DMACapability>
  <Category name="Category 1, 1 byte strict"
    enabled="true"
    bytes_in_burst="`=$Bytes`"
    bytes_in_burst_is_strict="false"
    spoke_width="2"
    inc_addr="false"
    each_burst_req_request="true">
    <Location name="`${INSTANCE_NAME}`_location_1_cat_1" enabled="true"
      direction="source" />
  </Category>
</DMACapability>
```

10.4.3.6 Inc Addr

通过 “inc_addr” 字段，可以在 DMA 向导的第二页中设置 “递增源地址” 或 “递增目标地址” 的初始选框状态（数据传输描述符或 TD 配置）。

```
<DMACapability>
  <Category name="Category 1, 1 byte strict"
    enabled="true"
    bytes_in_burst="1"
    bytes_in_burst_is_strict="true"
    spoke_width="2"
    inc_addr="true"
    each_burst_req_request="true">
    <Location name="`${INSTANCE_NAME}`_location_1_cat_1" enabled="true"
      direction="source" />
  </Category>
</DMACapability>
```

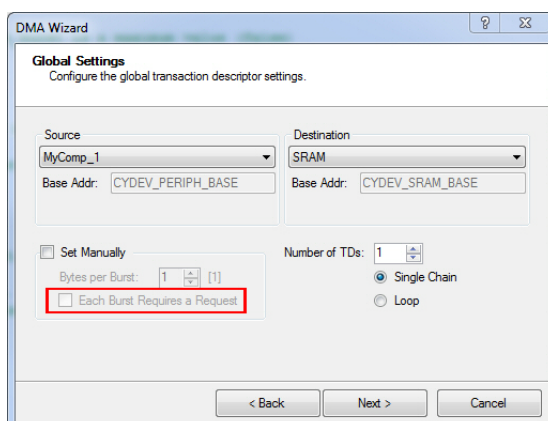


如果位置条目的“方向”字段（请参考[位于第 123 页上位置名称](#)）被设置为“目标地址”，那么当“inc_addr”字段被设为“真”时，请勾选 **Destination Inc** 的选框而不是 **Source Inc** 选框。

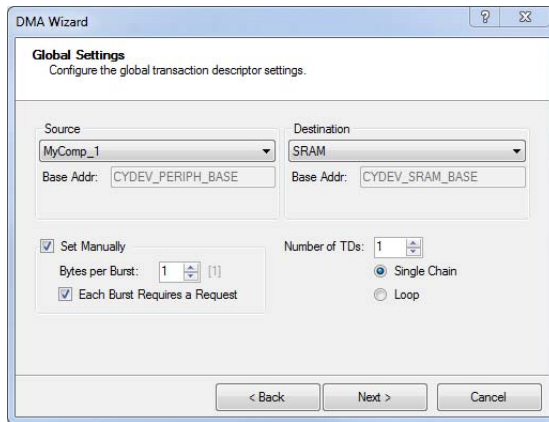
10.4.3.7 “Each Burst Requires A Request”（每次突发需要一个请求）字段

“each_burst_requires_a_request”字段指定了 DMA 向导中突发设置字段下 **Each Burst Requires a Request** 选框的初始值。

```
<DMACapability>
  <Category name="Category 1, 1 byte strict"
    enabled="true"
    bytes_in_burst="1"
    bytes_in_burst_is_strict="true"
    spoke_width="2"
    inc_addr="true"
    each_burst_req_request="false">
    <Location name="\${INSTANCE_NAME}_location_1_cat_1" enabled="true"
      direction="source"/>
  </Category>
</DMACapability>
```



与其它字段相同，无论 “each_burst_requires_a_request” 字段的值如何，用户都可以选择 **Set Manually** 选框，以便覆盖初始设置。



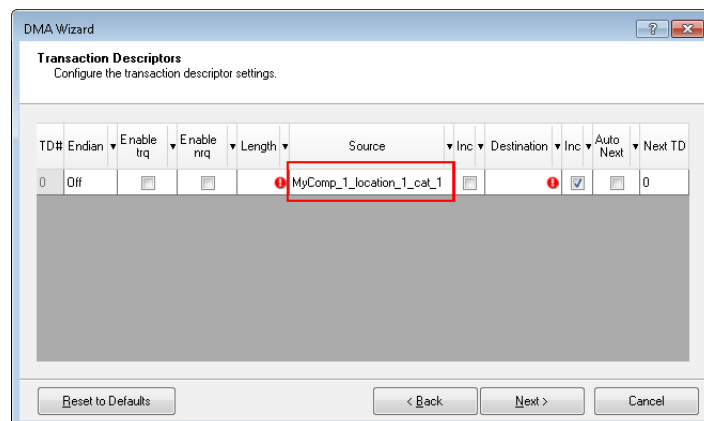
10.4.3.8 位置名称

“Location” 字段指定在 TD 配置窗口中所使用的实际地址。您可以在每一个类别中使用一个或多个 “Location” 字段。

注意： 寄存器名称的句法与参数的句法有点不同。

- 如果您在一个类别中仅有一个位置，那么该位置将被自动设置为 TD 配置窗口的相应源/目标地址。

```
<DMACapability>
  <Category name="Category 1, 1 byte strict"
    enabled="true"
    bytes_in_burst="1"
    bytes_in_burst_is_strict="true"
    spoke_width="2"
    inc_addr="true"
    each_burst_req_request="true">
    <Location name="\$INSTANCE_NAME\_location_1_cat_1" enabled="true"
      direction="source"/>
  </Category>
</DMACapability>
```



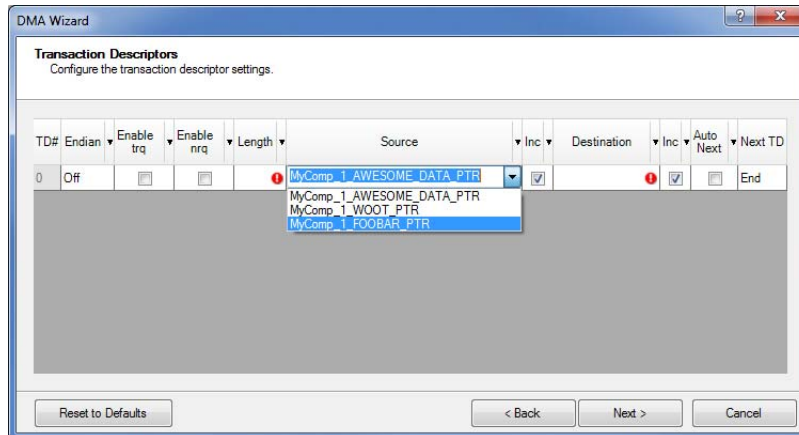
- 如果您在一个类别中使用了多个位置，那么下拉框中将显示所有位置以供选择。

```
<DMACapability>
  <Category name="Category 1, 1 byte strict"
    enabled="true"
```

```

bytes_in_burst="1"
bytes_in_burst_is_strict="true"
spoke_width="2"
inc_addr="true"
each_burst_req_request="true">
<Location name="\${INSTANCE_NAME}_AWESOME_DATA_PTR" enabled="true"
direction="source"/>
<Location name="\${INSTANCE_NAME}_WOOT_PTR" enabled="true" direction="source"/>
<Location name="\${INSTANCE_NAME}_FOOBAR_PTR" enabled="true" direction="source"/>
</Category>
</DMACapability>

```



使能字段

如果通过将“location”字段设置为“真”来使能该字段，那么位置条目将显示在 TD 配置窗口中。如果该字段被设置为“假”，它便不会显示在选项列表内。通过该字段，可以根据自定义程序中所设置的参数来选择性使能 / 禁用各选项。它的句法与其它字段的相同。

```
enabled="\${EnableLocationAwesome}"
```

方向字段

“direction”字段将位置设置为“源地址”和 / 或“目标地址”。该设置会影响类别在 DMA 向导中出现的位置，定位和递增地址的走向。

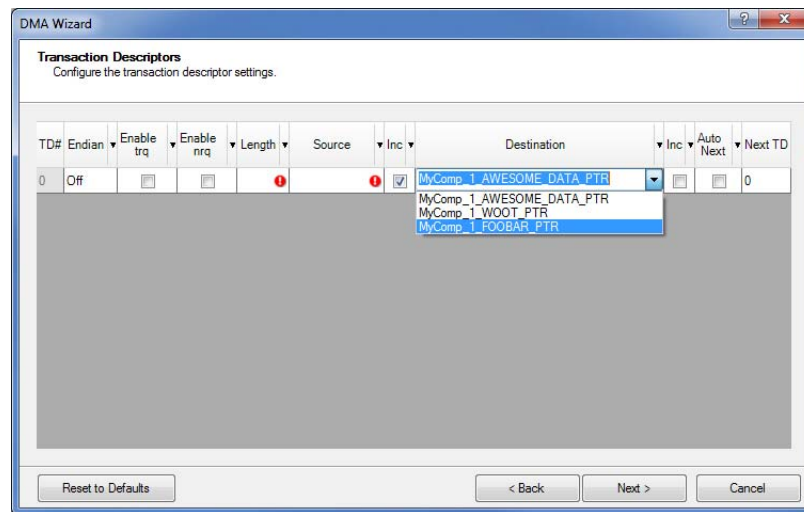
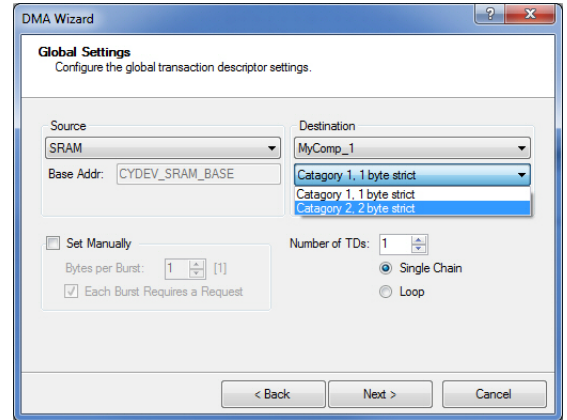
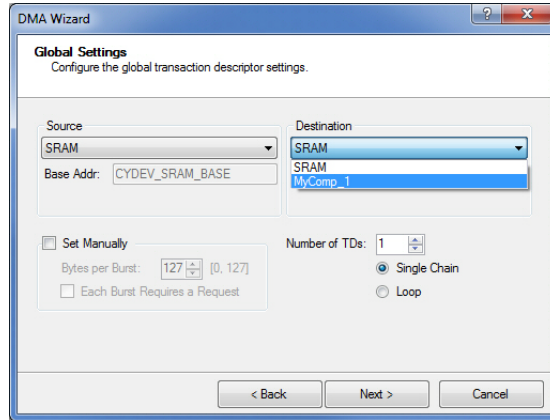
```

<DMACapability>
  <Category name="Catagory 1, 1 byte strict"
    enabled="true"
    bytes_in_burst="1"
    bytes_in_burst_is_strict="true"
    spoke_width="2"
    inc_addr="true"
    each_burst_req_request="true">
    <Location name="\${INSTANCE_NAME}_AWESOME_DATA_PTR" enabled="true"
    direction="destination"/>
    <Location name="\${INSTANCE_NAME}_WOOT_PTR" enabled="true" direction="destination"/>
    <Location name="\${INSTANCE_NAME}_FOOBAR_PTR" enabled="true" direction="destination"/>
  </Category>

  <Category name="Catagory 2, 2 byte strict"
    enabled="true"
    bytes_in_burst="2"
    bytes_in_burst_is_strict="true"
    spoke_width="2"

```

```
inc_addr="false"
each_burst_req_request="true">
  <Location name="`${INSTANCE_NAME}`_AWESOME_DATA_PTR" enabled="true"
  direction="destination"/>
  <Location name="`${INSTANCE_NAME}`_WOOT_PTR" enabled="true" direction="destination"/>
  <Location name="`${INSTANCE_NAME}`_FOOBAR_PTR" enabled="true" direction="destination"/
  >
</Category>
</DMACapability>
```



10.4.4 DMA 功能文件示例:

```
<!--
DMACapability needs to contain 1 or more Category tags. Category needs to contain 1 or more Location tags.
Category Attributes
=====
name: The name of the cataegory to display to the user in the DMA Wizard. (If only one category is entered
it will not be displayed as a sub-category in the wizard. Instead it will just be used when the
user selects its associated instance.)
enabled: [OPTIONAL] "true" or "false". If not provided it defaults to true. If false,
this category and its locations are not included in the DMA Wizard. Note: this value can be set
to an expression referencing parameters i.e. enabled="`${Your Expression here}`".
bytes_in_burst: Integer between 1 and 127. The number of bytes that can be sent/received in a single burst.
bytes_in_burst_is_strict: "true" or "false". Determines whether the bytes_in_burst is a maximum value (false)
or a specific value that must be used (true).
spoke_width: Integer between 1 and 4. The spoke width in bytes.
inc_addr: "true" or "false". Specifies whether or not the address is typically incremented.
each_busrtr_req_request: "true" or "false". Specifies whether or not a request is required for each burst.
Location Attributes
```

```

=====
name: The name of the location to display to the user in the DMA Wizard.
enabled: [OPTIONAL] "true" or "false". If not provided it defaults to true. If false, this
location is not included in the DMA Wizard. Note: this value can be set to an expression
referencing parameters by using: enabled="`=$Your Expression here`".
direction: "source", "destination", or "both".
-->
<DMACapability>
  <Category name="Catagory 1, variable byte strict"
    enabled="true"
    bytes_in_burst="`=$Bytes`"
    bytes_in_burst_is_strict="true"
    spoke_width="2"
    inc_addr="false"
    each_burst_req_request="true">
    <Location name="`$INSTANCE_NAME`_location_1_cat_1" enabled="true" direction="source"/>
    <Location name="`$INSTANCE_NAME`_location_2_cat_1" enabled="true" direction="source"/>
    <Location name="`$INSTANCE_NAME`_location_3_cat_1" enabled="true" direction="source"/>
  </Category>

<!-- blah blah blah -->

  <Category name="Catagory 4 optional, , 4 byte strict"
    enabled="`=$EnableExtraDest`"
    bytes_in_burst="4"
    bytes_in_burst_is_strict="true"
    spoke_width="2"
    inc_addr="true"
    each_burst_req_request="true">
    <Location name="`$INSTANCE_NAME`_location_1_cat_4" enabled="true" direction="destination"/>
    <Location name="`$INSTANCE_NAME`_location_2_cat_4" enabled="`=$EnableExtraDest && $EnableExtraLocations`"
direction="destination"/>
    <Location name="`$INSTANCE_NAME`_location_3_cat_4" enabled="`=$EnableExtraDest && $EnableExtraLocations`"
direction="destination"/>
  </Category>

  <Category name="Catagory 2, 4 byte not strict"
    enabled="true"
    bytes_in_burst="4"
    bytes_in_burst_is_strict="false"
    spoke_width="2"
    inc_addr="true"
    each_burst_req_request="true">
    <Location name="`$INSTANCE_NAME`_location_1_cat_2" enabled="true" direction="source"/>
    <Location name="`$INSTANCE_NAME`_location_2_cat_2" enabled="true" direction="source"/>
    <Location name="`$INSTANCE_NAME`_location_3_cat_2" enabled="true" direction="source"/>
  </Category>

  <Category name="Catagory 3, 2 byte strict"
    enabled="true"
    bytes_in_burst="2"
    bytes_in_burst_is_strict="true"
    spoke_width="2"
    inc_addr="true"
    each_burst_req_request="true">
    <Location name="`$INSTANCE_NAME`_location_1_cat_3" enabled="true" direction="destination"/>
    <Location name="`$INSTANCE_NAME`_location_2_cat_3" enabled="true" direction="destination"/>
    <Location name="`$INSTANCE_NAME`_location_3_cat_3" enabled="true" direction="both"/>
    <Location name="`$INSTANCE_NAME`_location_3_cat_3" enabled="true" direction="both"/>
  </Category>
</DMACapability>

```

10.5 添加 / 创建 .cystate XML 文件

通过添加 .cystate XML 文件，您可为组件提供组件状态的信息。该特性选项用于生成 DRC 错误、警告，以及有关组件是否符合量产质量或者组件是否适用于不兼容版本的芯片等内容的注释。如果您的组件不包含 .cystate 文件并且 / 或者目标器件没有条目，那么不会生成任何 DRC。

10.5.1 将 .cystate 文件添加到一个组件内

1. 右键点击组件，然后选择 **Add Component Item**（添加组件项）项。
这时会出现 **Add Component Item** 对话框。
2. 从模板的 **Misc** 类别中选择 **Misc. File**（Misc 文件）图标。
3. 它的名称与组件的相同，并且其扩展名为 **.cystate**（例如，**cy_clock_v1_50.cystate**）。
4. 点击 **Create New**（创建新文件）以创建一个空白文件；然后选择 **Add Existing**（添加现有文件），以选择想要添加到组件内的现有文件。
该项将显示在工作区浏览器中，并在 PSoC Creator 代码编辑器中作为选项卡式的文档被打开。

10.5.2 状态

通常，各组件可以处于下面三种状态：过时、生产或原型。

- **Obsolete**（过时状态）定义了不建议使用的组件。
- **Production**（生产状态）定义了完全经过测试并得到保证的组件，它们可用于生产设计。
- **Prototype**（原型状态）是指剩余的组件。该状态一般用于测试版或示例内容。

10.5.3 状态信息

.cystate 文件支持显示包含在注释列表窗口中的信息（包括注释类型和信息选项）。

10.5.3.1 注释类型

信息类型可以是以下三种的一种：

- **错误信息**终止编译操作
- **警告信息**它是一中很严重的信息，但不会终止编译操作
- **注释信息**它是风险较低的信息。

10.5.3.2 默认信息

PSoC Creator 具有原型状态和过时状态的默认信息。通过提供其自身字符串，组件可以覆盖这些默认信息。

- 对于**原型**组件，除非它被覆盖，否则工具将报告如下信息：
COMPONENT_NAME (INSTANCE_NAME) 是原型组件。该组件没有经过生产质量标准检测，生产时请务必小心。
- 对于**过时**组件，除非它被覆盖，否则工具将报告如下信息：
COMPONENT_NAME (INSTANCE_NAME) 是过时产品。建议不要再使用该组件，应当使用生产版本的组件（或备用实现版本）替换它。

10.5.4 最佳实践

下面是用于赛普拉斯组件的策略，以及赛普拉斯推荐使用于所有组件的信息。

- **量产**组件不应该生成任何信息。
- **过时**组件应当生成警告信息或错误信息。
这样是为了从用户设计中移除过时组件。当组件的状态为过时时，在首次发行版本中使用警告信息。然后在后续版本中将该信息升级为错误信息。这样可以确保您在没有警告的情况下不会破坏原始设计。

在大多情况下，建议覆盖默认信息，以便为用户提供更好的解决方法（既为引导他们使用特定的新版本或新组件）。

■ **原型**组件应生成注释信息或警告信息。

除非发生了已知的错误，否则原型组件要发出带有默认信息的注释。如果检测到某个错误，那么要将该信息升级为警告信息并覆盖默认信息，以说明该问题。

10.5.5 XML 格式

下面显示的是创建 .cystate 文件的格式。

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="ComponentStateRules" type="ComponentStateRulesType"/>

  <xs:complexType name="ComponentStateRulesType">
    <xs:sequence>
      <xs:element name="Rule" type="RuleType" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="Version" type="xs:int" use="required"/>
  </xs:complexType>

  <xs:complexType name="RuleType">
    <xs:sequence>
      <xs:element name="Pattern" type="PatternType" minOccurs="1" maxOccurs="1"/>
      <xs:element name="State" type="StateType" minOccurs="1" maxOccurs="1"/>
      <xs:element name="Severity" type="SeverityType" minOccurs="1" maxOccurs="1"/>
      <xs:element name="Message" type="xs:string" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="PatternType">
    <xs:sequence>
      <xs:element name="Architecture" type="xs:string"/>
      <xs:element name="Family" type="xs:string"/>
      <xs:element name="Revision" type="xs:string"/>
      <xs:element name="Device" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:simpleType name="StateType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Prototype"/>
      <xs:enumeration value="Production"/>
      <xs:enumeration value="Obsolete"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="SeverityType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="None"/>
      <xs:enumeration value="Note"/>
      <xs:enumeration value="Warning"/>
      <xs:enumeration value="Error"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

下面显示的是 .cystate 文件的关键元素。

ComponentStateRules

ComponentStateRules 是包括组件的一个或多个规则的根元素。该元素具有所需的 “Version” 属性，其唯一的合法值为 1。并且它有一个子要素，名为 “Rule”。

Rule

“Rule” 子要素定义了组件的规则。各个 “Rule” 子要素以优先级高低的顺序排列。一旦符合规则，将返回结果。每条规则支持以下要素集：

- **Pattern**（格式）— 这是需要匹配的格式名称。该要素的值由 **PatternType** 指定。它可以是任意架构、系列、器件和 / 或版本。这种格式是不分层的，它可留空以便查询所有条目。
- **State**（状态）— 它是状态的名称。该要素的值由 **StateType** 指定。它可以是量产、原型或过时等状态中的任何一个。
- **Severity**（严重程度）— 这是组件为该特殊规则所提供的信息类型。该要素的值由 **SeverityType** 指定，包括：
 - 无：表示符合要求或安全状态
 - 注释信息：报告信息
 - 警告信息：需要注意
 - 错误信息：危险或不合法
- **Message**（信息）— 它是信息的实际字符串。“Message” 子要素是可选的，如果未指定，那么工具将显示默认信息。

10.5.6 .cystate 文件的示例

以下示例显示的是一个 .cystate 文件，该文件将组件指定为 PSoC 5 器件的原型组件。如果器件是 PSoC 3 ES2，那么该组件已经过时，并且将提示用户更新为 ES3 版本。如果器件是 PSoC 3 ES1，那么用户将获得提示该组件已经过时的默认信息。

```
<?xml version="1.0" encoding="utf-8"?>
<ComponentStateRules Version="1">
  <Rule>
    <Pattern>
      <Architecture>PSoC5</Architecture>
      <Family>*</Family>
      <Revision>*</Revision>
      <Device>*</Device>
    </Pattern>
    <State>Prototype</State>
    <Severity>Warning</Severity>
    <Message></Message>
  </Rule>
  <Rule>
    <Pattern>
      <Architecture>PSoC3</Architecture>
      <Family>*</Family>
      <Revision>ES2</Revision>
    </Pattern>
    <State>Obsolete</State>
```

```

    <Severity>Error</Severity>
    <Message>Please update to ES3 to use this component</Message>
  </Rule>
</Rule>
<Pattern>
  <Architecture>PSOC3</Architecture>
  <Family>*</Family>
  <Revision>ES1</Revision>
  <Device>*</Device>
</Pattern>
<State>Obsolete</State>
<Severity>Error</Severity>
<Message></Message>
</Rule>
</ComponentStateRules>

```

10.6 添加静态库

一个组件可能需要附带用于软件堆栈的预编译库。由于编译器工具链和配置（DEBUG/RELEASE）不同，可能需要向一个组件实施添加不同的库。

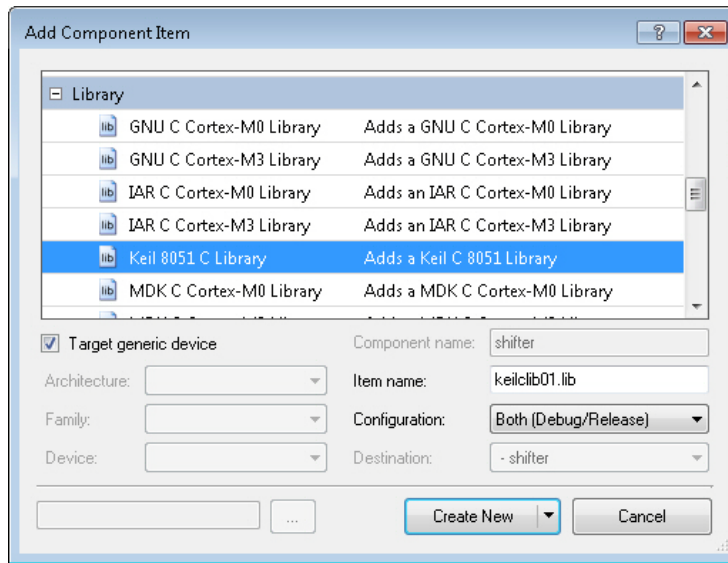
原因如下：

- **IP 保护** — 对于通信组件，一般需要隐藏软件堆栈的实施。通过使能组件的静态库，开发者无需提供整个软件封装的源代码也能使用标准的 PSoC Creator 组件开发流程（例如，没有提供额外库）。为了发挥该特性的优势，可以选择使用 CapSense。
- **编译性能** — 如果组件需要很多软件参与，势必会降低 PSoC Creator 的编译性能，因为它需要多次编译同一个代码。通过提供库中的非易失性源代码，组件创建者可以避免项目因使用了库内容而降低编译速度。

如何向组件添加库：

1. 右键点击 Workspace Explorer 中的组件，然后选择 **Add Component Item** 项。

这时会出现 **Add Component Item** 对话框。



1. 从模板中的 **Library** 目录下为所需工具链选择相应的库文件类型。根据所选类型，将显示相应的默认 **Item name**（项名称）。请参见[位于第 131 页上最佳实践](#)。
2. 当选中某个库项时，**Configuration** 字段将有效，用户可以选择所需的配置：
 - 调试
 - 释放
 - 两者
3. 通过点击 **Create New**（创建新文件），使 PSoC Creator 创建一个文件，或从下拉菜单中选择 **Add Existing**（添加现有文件），以选择现有的文件。

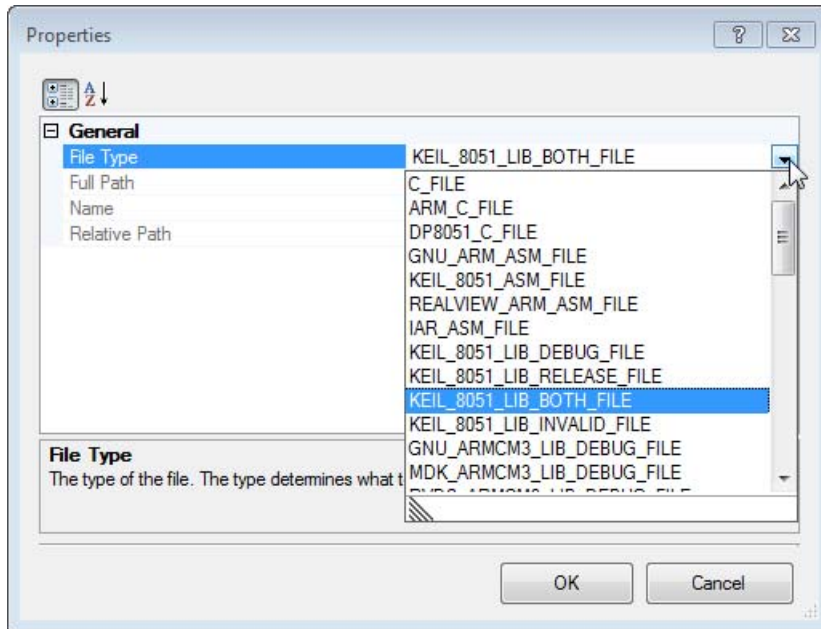
新建的文件被添加到 **Workspace Explorer** 中的“Library”目录下。

10.6.1 最佳实践

组件创建人员需要确保任何两个组件不能共享静态库代码。既为：**Comp1 API** 或库中的代码不能使用 **Comp2** 静态库中定义的符号。

使用一个组件中的静态库的最佳方法是使创建的组件只有唯一的静态库，并且将该组件添加到使用静态库的组件原理图内。这样，组件创建者可以更新顶层组件，并不用更新静态库，或进行方向操作。另外，组件创建人员还需要确保所有静态库均以架构级被添加到组件中。这样做可以确保组件创建人员将 **CM-0** 代码添加到基于 **CM-0** 的器件内，将 **CM-3** 代码添加到基于 **CM-3** 的器件内。

创建库过程中，如果组件创建人员选错了模板（选择 **GCC** 而不是 **MDK**），创建人员可以访问它们的组件，然后选择包含了错误模板的库，并从该库的右键菜单中选择属性项。在属性窗口中，使用 **File Type** 属性来选择库的相应模板。



10.7 添加依赖属性

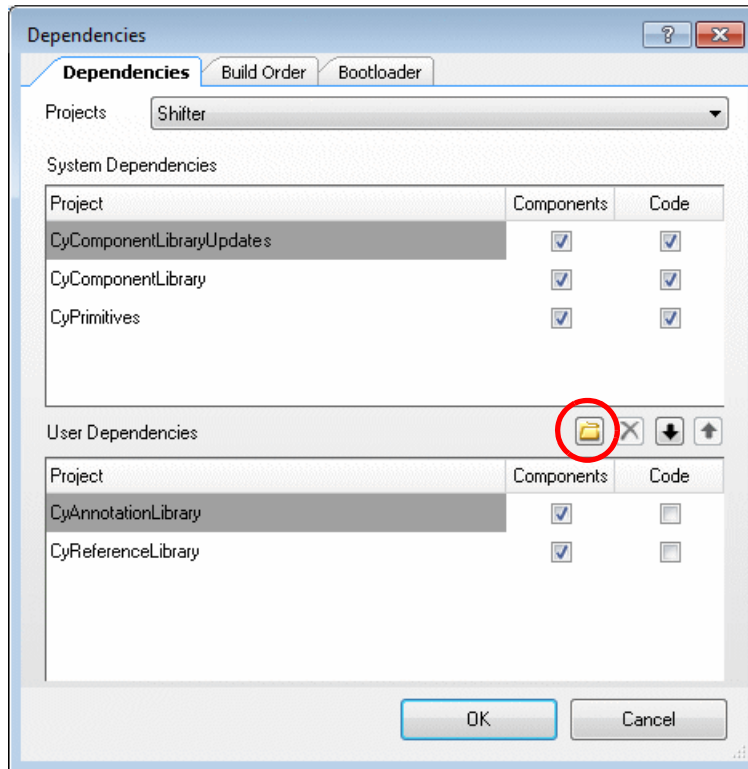
PSoC Creator 依赖于不同的系统依赖属性，这些属性包含了组件目录中由赛普拉斯提供的组件。一旦完成项目中的所有组件，您便可以给它添加用户依赖属性或默认依赖属性。

10.7.1 添加用户依赖属性

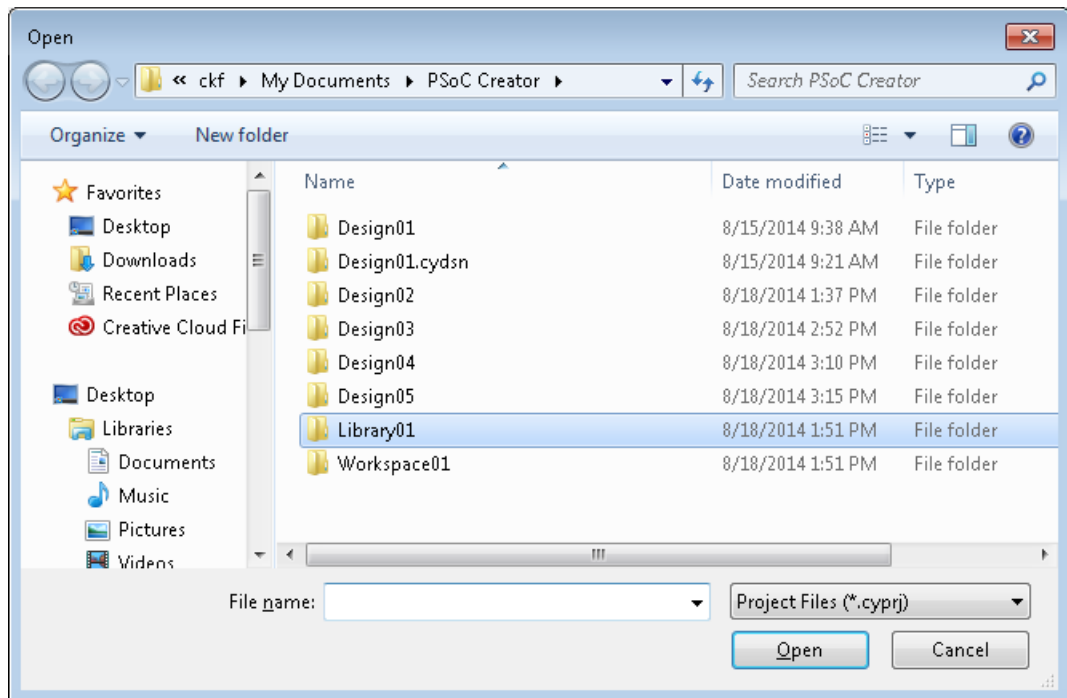
用户依赖属性使用于特定项目。

1. 打开或创建 PSoC Creator 中的设计项目。
2. 选择 **Project > Dependencies**。

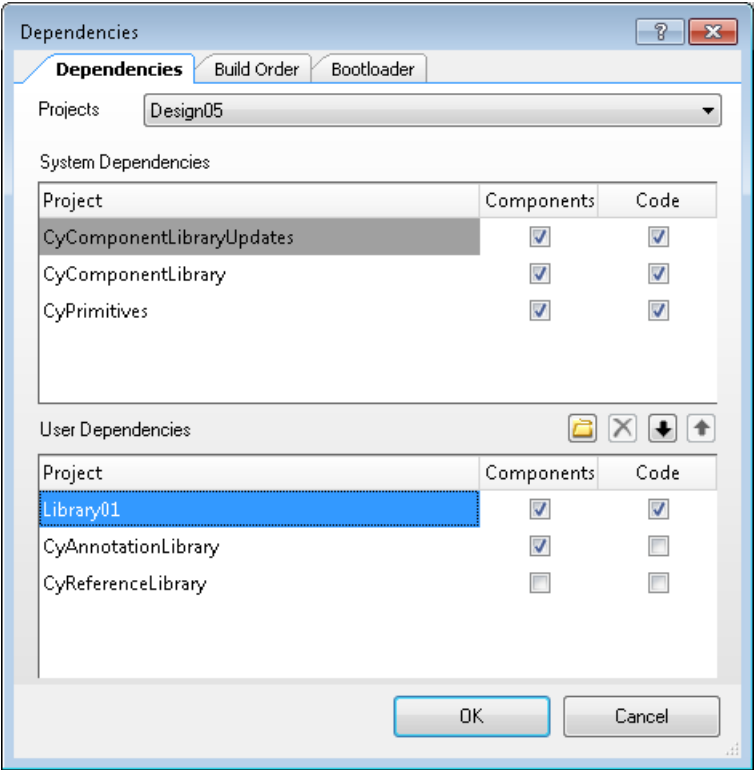
3. 点击 Dependencies 对话框中 **User Dependencies** 选项卡下的 **New Entry** 按钮。



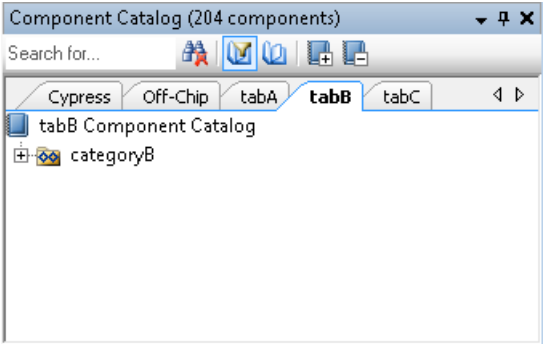
4. 在 Open 对话框中，找到项目所在的位置，然后选择它，并点击 **Open** 以关闭对话框。



5. 在 **Dependencies** 对话框中，您所选择的项目将被罗列在 **User Dependencies** 选项卡下。点击 **OK**，以关闭 **Dependencies** 对话框。



6. 在组件目录下，根据组件的配置，可能会有一个或多个新选项卡以及组件目录。

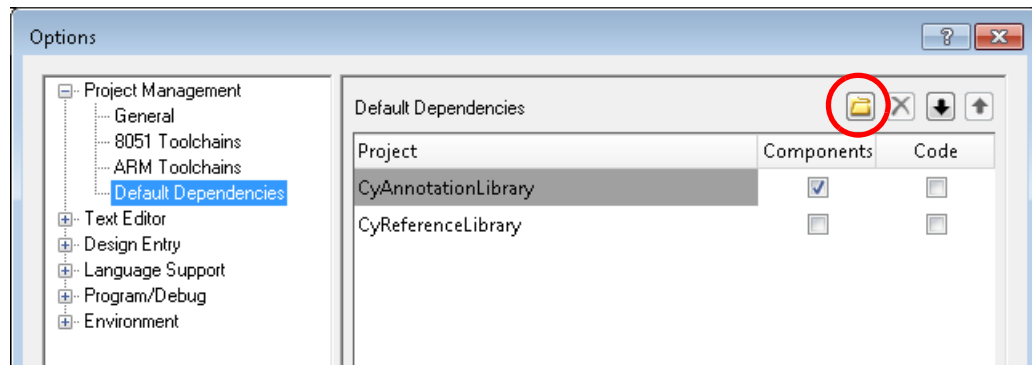


10.7.2 添加默认依赖属性

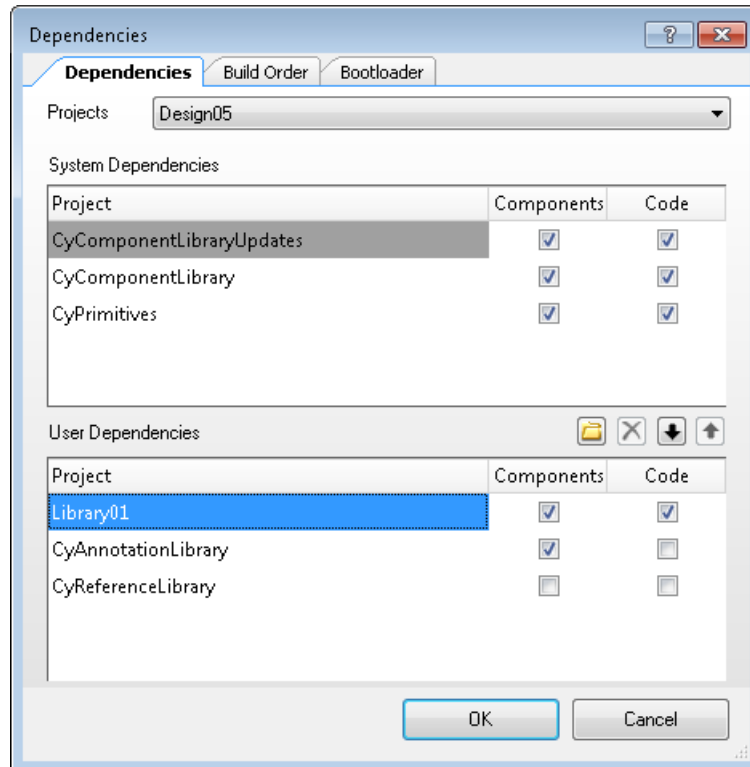
默认依赖属性被使用于所有 PSoC Creator 项目。

1. 如果在 PSoC Creator 中没有打开任何项目，那么请选择 **Tools > Options**，打开 **Options** 对话框。

2. 在该对话框中，展开 **Project Management** 目录并选择 **Default Dependencies**，然后点击 **New Entry** 按钮。

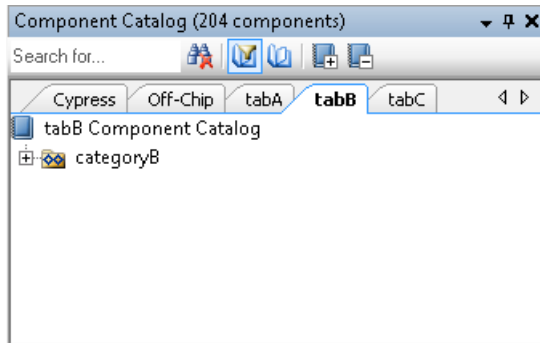


3. 在 **Open** 对话框中，找到库项目所在的位置，然后选择它，并点击 **Open** 以关闭对话框。
4. 在 **Options** 对话框中，您所选择的项目将被罗列在 **Default Dependencies** 选项卡下。点击 **OK**，关闭 **Options** 对话框。

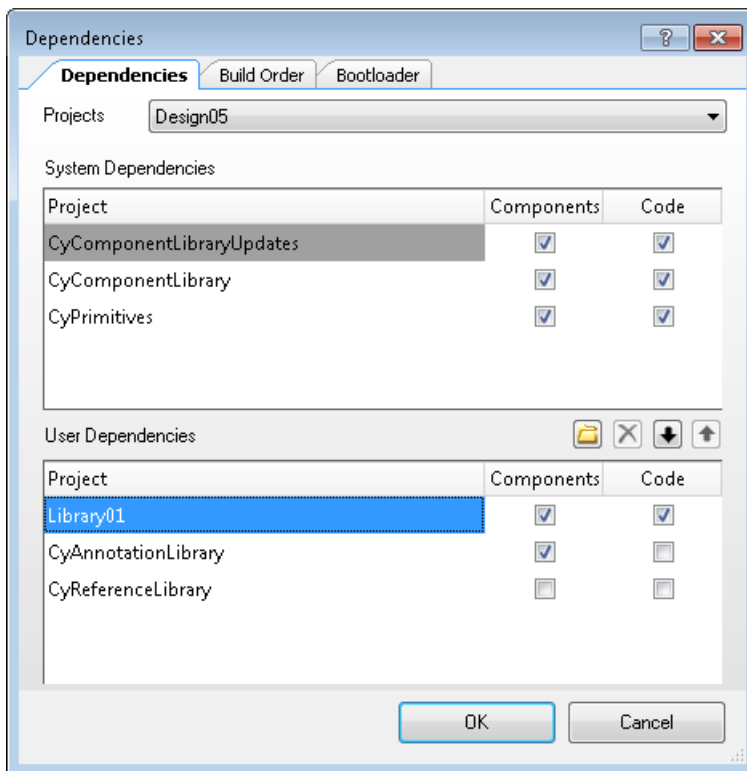


5. 打开或创建一个设计项目。

6. 在组件目录下，根据组件的配置，可能会有一个或多个新选项卡以及组件目录。



7. 如果您打开了 **Dependencies** 对话框，那么请注：意项目已经被包含在 **User Dependencies** 选项卡下。



10.8 编译项目

当已经添加并完成组件所需的所有组件项时，用户必须实例化设计中的组件，以便进行创建和检测该组件。

当编译一个组件项目时，将编译它的所有组成部分，包括系列特定的部分。编译后，编译目录会包含组件创建人员通过项目管理器所指定的组件的各个方面。

11. 最佳实践



本章节列出了创建组件时需要考虑的常见问题，包括：

- 时钟
- 中断
- DMA
- 低功耗支持
- 组件封装
- Verilog

11.1 时钟

当使用 PSoC 器件的内部时钟资源时，需要注意一些问题。对于数字系统，必须完全掌握时钟架构、设计限制和器件的特性。

PSoC 器件中多种可用数字资源（并不是所有资源）可由器件的内部时钟提供时钟源。有许多资源要通过片外时钟源提供。设计用于 PSoC 器件的组件时，均要考虑到这两种场合。

11.1.1 UDB 时钟架构注意事项

每个 UDB 中至少存在两个时钟域。第一个时钟域是总线时钟（bus_clk）。CPU 使用该时钟域访问 UDB 的内部寄存器，这里所说的内部寄存器包括状态、控制和数据路径。

第二个时钟域受“用户”时钟的控制。该时钟可能来自 PSoC 外部时钟结构。它是用于执行函数的主要时钟。

为了防止 UDB 中发生的亚稳态，每当信号超出时钟域的范围时需要使用同步触发器。有两种方法可通过 UDB 的内部资源实现。第一种方法是 PLD 宏单元作为同步器使用。第二种方法是将状态寄存器作为 4 位同步器使用。可以对状态寄存器进行相应配置，以便将 8 个内部触发器变成 4 个双触发器同步器。但是，使用这种方法，状态寄存器不再是 UDB 的内部资源。

11.1.2 组件时钟注意事项

在设计中，选择某个特殊功能的时钟源时，很可能需要使用时钟的两个边沿。在某些情况下，可以这样做，但是在其它情况下，这样做的风险很大。例如，如果所使用的时钟源的占空比不是 50%，那么可能违背了建立和保持时序，这样会导致意外或不预期的结果。为了避免发生这种情况，所有设计只应使用上升沿（Verilog 实例），并且只能将组件的时钟引脚连接至时钟的上升沿上。如果有某个事件必须在下降沿上发生，可以通过相应电路使所需时钟频率加倍。

11.1.3 UDB 到芯片资源的时钟的注意事项

UDB 阵列中进出的信号并不完全来自器件的 I/O 引脚。多数信号来自连接至 PSoC 的其它资源，如固定功能模块、DMA 或中断。有些模块具有内置的重新同步器，而有些模块则没有。一旦信号超出了 UDB 阵列的范围，都可能引发问题。必须对这些信号加以分析，以确保它们满足所需时序。

在所有场合中，对超出时钟域范围的信号进行时序分析非常有必要。另外，即使信号与内部时钟同步，也需要验证信号与其输入区域间的相位关系。发生偏移的同步信号可能满足不了目标电路中时序的要求。

11.1.4 UDB 到输入 / 输出时钟的注意事项

UDB 中适用于全局、bus_clk 和用户时钟的时钟结构并不完全适用于 PSoC 中其它架构。所以，要特别注意发送到 UDB 或 UDB 接收到的信号。

GPIO 寄存器只能访问 bus_clk，并且其时钟没有与某项功能使用的外部时钟对齐。因此，所有非 bus_clk 提供时钟源的输出寄存器在发送到输出端前必须使用 UDB 资源。这样，会使输出路径的时钟周期很长。

UDB 可以将任何信号作为外部时钟信号使用。不过，不能通过时钟树直接访问这些外部时钟。在该时钟能够进入时钟树前，需要使用长路径来路由它们。较长的时钟到达时间会加长建立时间，这样使得 I/O 时序问题更加复杂。

11.1.5 触发器的亚稳态

谈到数字电路的时钟，就必须了解时钟架构的影响，尤其是触发器亚稳态的定义和解释内容。

亚稳态指的是触发器输出不可预知或处于不稳定（亚稳定）状态的时长。最终，经过一段时间，该状态将转为稳定的‘1’或‘0’状态。然而，对于依赖于输出的电路，该处理速度不足以正确评估最终结果。

在组合电路中，这些输出会对驱动电路产生干扰。在后续电路中，这些干扰会影响寄存器的数据存储和状态机的决策过程。因此，当务之急是要避免发生亚稳态。

下面列出了引起亚稳态的一些条件：

- 信号超出了时钟域范围。
- 在一个时钟域的触发器间创建的组合路径过长。
- 一个时钟域中各个触发器间存在的时钟时滞

上述（或其它）条件中任何一项都能引起亚稳态。一般情况下，如果违背了触发器的建立时间（Tsu）或保持时间（Th），那么可能发生亚稳态。

11.1.6 超出时钟域范围

PSoC 3/5 UDB 具有一些内部存储单元。每个单元都可以通过 CPU bus_clk，又可以通过作为电路主时钟的其它时钟源（ClkIn）进行访问。可以通过下述各项选择该时钟源：1) CPU bus_clk；2) 全局时钟（SrcClk）；3) 外部时钟（ExtClk）。

如果 ClkIn 是 bus_clk，那么我们可以确保存在一个时钟域，这样便不用担心发生超出时钟域的情况。但是，仍存在过长时滞或长组合路径的可能性。

11.1.7 长组合路径的注意事项

在设计中使用较长的组合路径时，可能会违反随后存储单元的建立时间要求。为了防止发生这种情况，总延迟时间必须小于时钟周期。换句话说，便是必须满足下面的公式：

$$T_{co} + T_{comb} + T_{su} < T_{cycle} \quad \text{公式 1}$$

T_{co} 是驱动触发器的输出时间。 T_{comb} 是中间电路的组合延时。 T_{su} 是下一个触发器阶段的建立时间。

如果这些长路径完全位于同一个组件中，那么处理起来比较容易。可以通过一个触发器驱动组件的输出信号来避免这种情况。如果不能或不希望这样做，那么必须完全了解时钟到组件输出之间的时序。

11.1.8 同步与异步时钟

时钟架构一般包含多种时钟。有的与主设备系统时钟同步，有的则同它异步。为实现我们的目的，需要将主设备系统时钟定义为 MPU 的内核时钟（或它的派生时钟）。将该时钟作为 bus_clk。

为了有效防止发生亚稳态问题，连接至 MPU 的信号需要与 bus_clk 信号同步。如果它们是该时钟派生出的，那么设计者的任务便是确保不要发生长组合路径或过长的时滞问题（如上面所述）。

如果需要连接至 MPU 的信号由与 bus_clk 异步的时钟控制，那么到达接口前，需要对这些信号进行同步化处理。PSoC Creator 中帶有一些有助于实现该任务的基元组件。下面各节将对它们进行详细说明。

11.1.9 使用 cy_psoc3_udb_clock_enable 基元

PSoC Creator 中有一个可通过 Verilog 基元访问的工具。它能够解决时钟的复杂性，并自动处理时钟调节问题。该基元是 cy_psoc3_udb_clock_enable。使用该基元可以处理同步和异步时钟。

cy_psoc3_udb_clock_enable 具有使能（enable）和时钟（clock_in）信号输入、驱动 UDB 组件的时钟输出（clock_out），并且具有用于指定时钟结果（sync_mode）的同步行为的参数。

clock_in 信号可以是全局时钟或本地时钟，它与 bus_clk 可以是同步的，也可以是异步的。enable 信号与 bus_clk 也可以是同步或异步的。这两个信号被连接至基元，用户将选择其同步或异步模式。一

一旦完成，PSoC Creator 中的 Fitter 将确定获得 UDB 元素要求的时钟行为时所需的必要实现，并将相应信号连接至被映射的 UDB 结果。下表列出了将时钟 / 使能信号映射到 UDB 时所使用的规则集：

输入			UDB 转换		
clock_in	en	sync_mode	模式	Enable	Clock_out
Global Sync	同步	有	级别模式	En	ClkIn
Global Async	同步	有	不支持，合成中发生错误		
Local Sync	同步	有	边沿模式	ClkIn & En	SrcClk(ClkIn)
Local Async	同步	有	边沿模式	Sync(ClkIn & En)	bus_clk
Global Sync	异步	有	级别模式	Sync(En)	ClkIn
Global Async	异步	有	不支持，合成中发生错误		
Local Sync	异步	有	边沿模式	Sync(ClkIn & En)	SrcClk(ClkIn)
Local Async	异步	有	边沿模式	Sync(ClkIn & En)	bus_clk
Global Sync	同步	无	级别模式	En	ClkIn
Global Async	同步	无	级别模式	En	ClkIn
Local Sync	同步	无	级别模式	En	ExtClk(ClkIn)
Local Async	同步	无	级别模式	En	ExtClk(ClkIn)
Global Sync	异步	无	级别模式	Sync(En)	ClkIn
Global Async	异步	无	级别模式	Sync(En)	ClkIn
Local Sync	异步	无	级别模式	Sync(En)	ExtClk(ClkIn)
Local Async	异步	无	级别模式	Sync(En)	ExtClk(ClkIn)

在上表中，Sync() 函数表示使用双触发器对信号和 clock_out 进行同步化处理。可以通过 UDB 宏单元获得双寄存器。

cy_psoc3_udb_clock_enable 基元的典型实例如下：

```
cy_psoc3_udb_clock_enable_v1_0 #(.sync_mode ('TRUE')) My_Clock_Enable (
    .clock_in      (my_clock_input),
    .enable        (my_clock_enable),
    .clock_out     (my_clock_out));
```

11.1.10 使用 cy_psoc3_sync 组件

另外一个有用的工具是 **cy_psoc3_sync** 基元。它是一个双触发器，用于对信号和时钟进行同步化处理。这些双触发器是 **UDB** 状态寄存器的一个选项。即使该基元是单比特宽，它仍会使用整个状态寄存器。就算可能使用了四个基元，但仍只占用一个寄存器。

cy_psoc3_sync 基元的典型实例如下：

```
cy_psoc3_sync My_Sync (  
    .clock      (my_clock_input),  
    .sc_in      (my_raw_signal_in),  
    .sc_out     (my_synced_signal_out));
```

11.1.11 路由、全局和外部时钟

UDB 具有三个可用的时钟类型，包括：

- 8 个全局时钟 — 通过用户可选的时钟分频器获得
- **bus_clk** — 系统中频率最高的时钟
- 外部时钟 — 来自外部信号，作为时钟输入使用，以便支持直接时钟的功能（如 **SPI** 从设备）。

11.1.12 时钟下降沿的隐患

设计 **Verilog** 时，需要防止发生下降沿状态。该理论也适用于连接至实例化组件或原理图组件的时钟。

混合使用上升沿和下降沿触发通常可以限制信号横向经过其路径的时长。例如，如果某个触发器由时钟上升沿提供时钟，那么它的输出通过某个逻辑后进入由时钟下降沿触发的触发器，**Tcycle** 被剪短了一半。如果需要某个双边沿触发器，应该使时钟速度加倍，并且使用替代上升沿来触发。

11.1.13 常见的时钟规则

- 不要在组件中创建时钟。时钟信号应该是组件的输入，并且所有时钟逻辑都要由同一个时钟提供。例如，不要对时钟信号进行“**AND**”运算来得到门控时钟。
- 也不要使用全局时钟信号或创建包括该信号的组合信号。同样，不要从组件中输出全局时钟信号。全局时钟除了作为 **UDB** 元素的时钟输入外，在其他场合中，未定义它的时序。要想创建具有已定义时序的时钟信号，必须使用由其他时钟所提供的触发器输出。这样，生成的时钟输出速率为输入时钟的一半或更低。
- 驱动来自组件中计时部分（触发器）的所有输出，以便最小化时钟到输出的时长。
- 保密异步复位和预设。从时序角度来看，它们将导致从复位 / 预设信号到注册的输出的组合时序路径。
- 如果数据信号与时钟异步，那么需要使用同步器。
- 不能将全局时钟信号作为组件的输出信号。
- 防止在时钟的下降沿上提供脉冲。使用双倍速的时钟。

11.2 中断

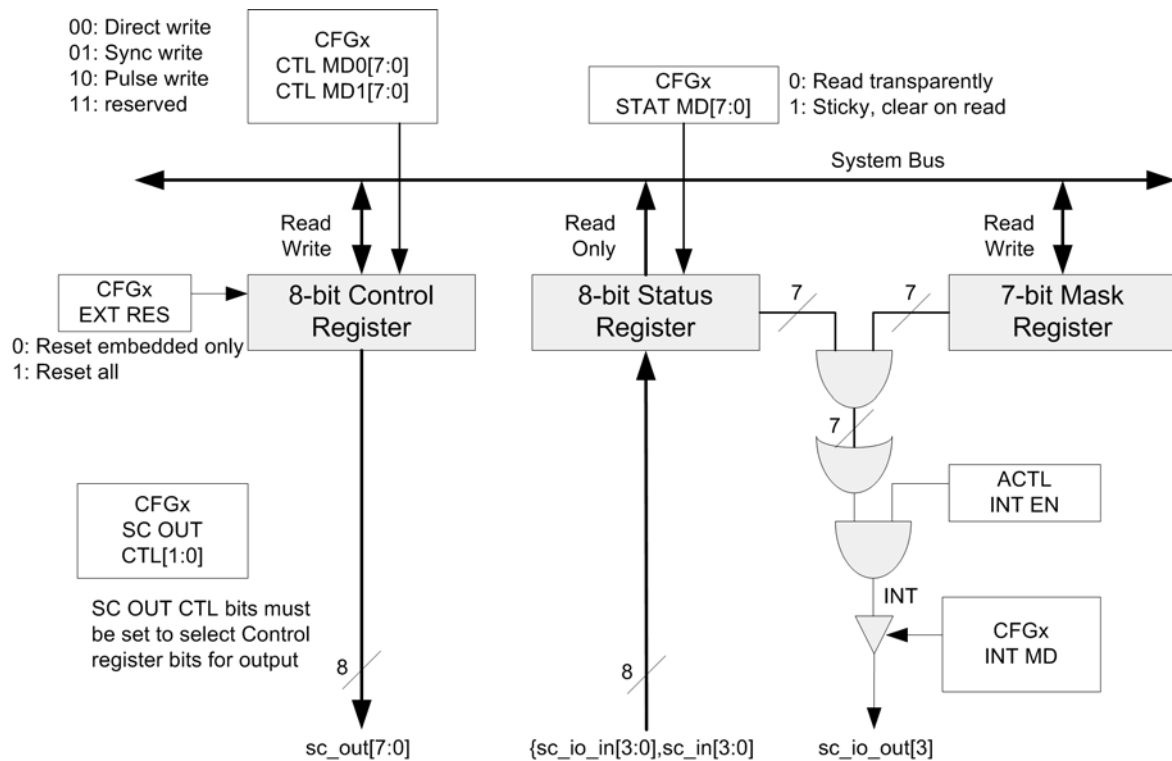
中断的主要作用是与 CPU 固件或 DMA 通道进行交互。可以使用连接的 UDB 阵列中的任何数据信号来生成中断或 DMA 请求。状态寄存器和 FIFO 状态寄存器是生成这些中断的主要工具。当需要少量数据和要求与 CPU 频繁交互时，状态寄存器是生成中断的合理选择。

中断在组件中被隐藏还是显示为符号上的终端，取决于中断处理是否特定于应用。如果中断处理特定于应用，那么组件将执行必要的操作，然后为适当处理的应用添加合并标志。

中断或 DMA 请求的生成特定于设计。例如，面向 CPU 的中断被设计为粘滞（读取后将清除），但同样的 DMA 请求并不适合，这是因为传输描述符的设计除了处理数据的传输描述符外，还必须包含单独的描述符，用于清除请求。

11.2.1 状态寄存器

下面显示的是状态和控制模块的高级视图。该模块的主要用途是协调 CPU 固件与内部 UDB 操作之间的交互。



状态寄存器是只读的，它同时允许将内部 UDB 状态从内部路由直接读出到系统总线中。这样一来，固件便能够监控 UDB 处理的状态。这些寄存器的每一位均有到路由矩阵的可编程连接。

一个状态中断示例是在 PLD 或数据路径模块生成了某个条件后发生的情况，例如由状态寄存器捕获，随后由 CPU 固件读取（并清除）的“比较结果为真”的条件。

11.2.2 内部中断生成和屏蔽寄存器

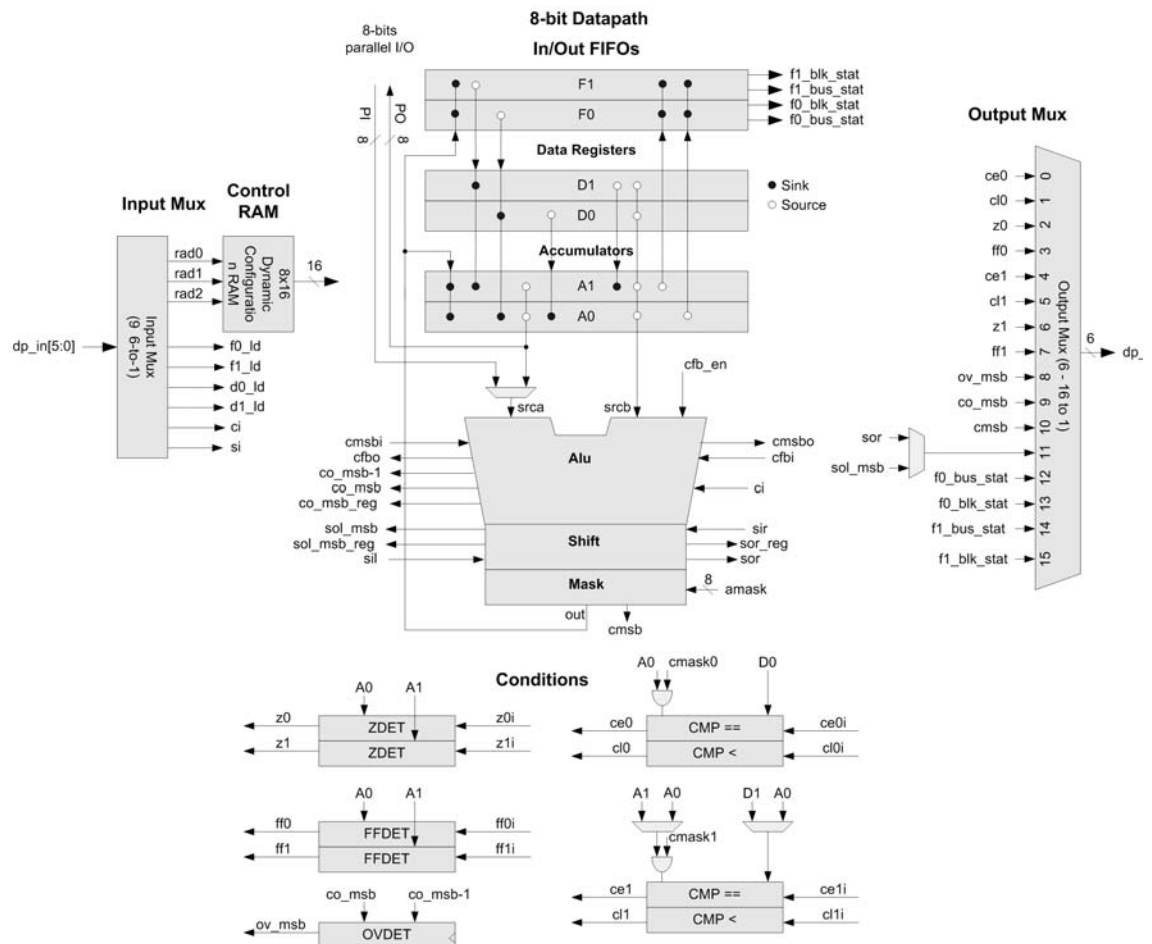
几乎在所有功能中，中断生成以状态位的设置为条件。如上图所示，该功能作为屏蔽（屏蔽寄存器）和 OR 减少状态被内置到状态寄存器逻辑内。只有状态输入的低 7 位可以与内置中断生成电路一起使用。默认情况下，sc_io 引脚为输出模式，并且可以向路由矩阵驱动中断，以实现连接到中断控制器。在该配置中，状态寄存器的最高有效位作为中断位的状态被读取。

状态模式寄存器（CFGx）提供了每个状态寄存器位的模式选择。透明读取指的是 CPU 读取状态寄存器时将返回路由输入信号的状态的模式。粘滞（读取后将清除）是一种模式。在该模式下，对输入状态进行采样并在输入为高电平时，无论输入的后续状态如何，都将寄存器位置位，并保持其置位状态。在后续读取时，通过 CPU 清除寄存器位。该模块的选定时钟用于确定采样率。该采样率不能小于生成状态输入信号的速率。

11.2.3 睡眠间隔保持

屏蔽寄存器是保留寄存器并在睡眠间隔间保持它的状态。状态寄存器是非保留寄存器。在睡眠间隔内它会释放状态，并在唤醒时复位为 0x00。

当串流大量数据或发送或接收快速突发时，DMA 传输是最佳选择。在这种情况下，需要使用 FIFO 状态寄存器执行控制数据流的中断。下图显示的是数据模块的高级视图。这些 FIFO 能够生成状态信号，可以对这些信号进行布线，使之与序列发生器、中断或 DMA 请求相交互。



11.2.4 FIFO 状态信号

共存在 4 个 FIFO 状态信号（f1_blk_stat、f1_bus_stat、f0_blk_stat、f0_bus_stat），每个 FIFO 有两个信号，可将这些 FIFO 单独配置为输入缓冲区（系统总线写入到 FIFO，数据路径内部读取 FIFO）或输出缓冲区（数据路径内部写入到 FIFO，系统总线读取 FIFO）。对于器件系统，“bus”（总线）状态信号具有重要的意义，并且应该将该信号布线给 DMA 控制器，作为 DMA 请求信号使用。

（当系统需要读取或写入字节，）“总线”状态信号主要用于控制系统总线，从而与 DMA 相交互。

当实现各个数据交换的缓冲区时，需要决定用于存储数据的 RAM 空间大小。如果缓冲区大小未超过 4 个字节，那么应该使用 FIFO 硬件实现缓冲区。

通过接收和传输 FIFO 中所提供的缓冲区，用户应用逻辑的处理顺序可独立于总线上的数据传输顺序。接收 FIFO 也能处理在接口上通常观察到的数据突发特性。该 FIFO 还能将用户应用逻辑的操作频率和特定总线的频率相互分离开。进行该操作时，请考虑缓冲区的上溢和下溢条件。

11.2.5 缓冲区上溢

接收 **FIFO** 要有足够的空间用于接收所有被传输到特定端口的挂起（已经预定或安排）数据，以避免发生潜在的上溢情况。在理想的情况下，接收 **FIFO** 设计和状态检查机制需要确保不会发生因上溢现象造成数据损失的情况。

要想解决上溢问题，设计人员必须采用 **FIFO** 状态的前瞻指示。因此，如果数据路径延迟、状态路径延迟和最大突发传输的时间的总和小于 **FIFO** 已满的时间，那么任意端口 **FIFO** 都会表示满足状态。这便意味着必须将 **FIFO** 的高水印设置为数据路径延迟、状态路径延迟以及最大突发传输时间的总和。实际上，当指出满足条件后必须在端口 **FIFO** 中多提供一个额外的空间，以避免缓冲区的溢出现象。

11.2.6 缓冲区下溢

接收端口 **FIFO** 的下溢现象是指当数据下降到低于低水印时，即使发送 **FIFO** 端口仍给该端口传送数据但也不能通过接口接收来自其他端的任何数据，最终变为空白的。这种现象发生的原因是发送器在得到下一次更新前已经用完了先前的分配额，因此接收器会缺少数据。为了避免发生下溢现象，必须将状态指示的水印设置为足够高的水平，以便在应用逻辑释放来自 **FIFO** 端口数据前，发送器能够响应接收器指定的可用 **FIFO** 空间。

自从 **FIFO** 状态信号表示特定端口缺乏数据的已用时间被计算为路径的总延迟，即等于状态更新的延迟、状态路径的延迟、数据预计的延迟和数据路径的延迟之和。前两个数值表示为获取发送器建立的数据突发传输的信息所占用的时间。后两个数值指的是获取通过接口的特定连接从发送 **FIFO** 传给接收的数据所需要的时间。

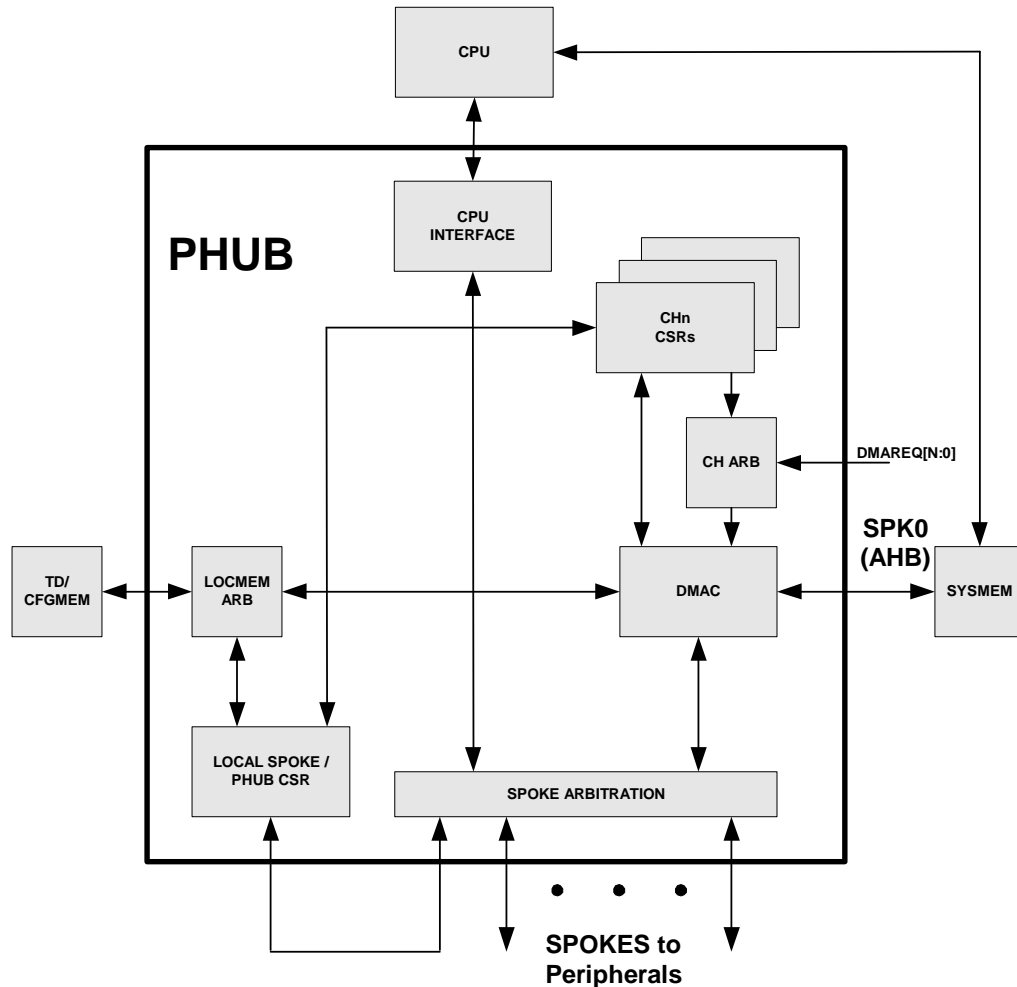
缓冲区的下溢现象取决于由应用逻辑读取 **FIFO** 端口的最大读取速率。要避免发生下溢情况，软件需要将每个端口 **FIFO** 的水印设为低水平（即足够大的水平）。

11.3 DMA

外设 HUB（PHUB）是 PSoC 3 和 PSoC 5 中的可编程和可配置的中央集线器；通过使用标准的先进微控制器总线结构（AMBA）的高性能总线（AHB），它会将偏上系统的不同组件连接起来。

PHUB 主要采用多层 AHB 结构，以便能够同时使用 AMBA-lite 型主器件。PHUB 包含一个 DMA 控制器（DMAC），可以编程该 DMAC 以便在系统中各个组件间进行数据传输而不需要 CPU 参加处理。PHUB 包含用以执行 DMAC 和 CPU 间仲裁的逻辑，以访问 PHUB 的下行多层并行访问路径。

下图描述了 CPU、PHUB、SYSMEM、TD/CFGMEM 和下行多层并行访问路径之间的通用连接。



有关寄存器映射的详细信息，请参考 *PSoC® 3、PSoC® 5 架构数据参考手册*（TRM）。在这里需要关注下面两点内容：第一，DMA 控制器可卸载 CPU，它是一个独立的总线主设备；第二，DMAC 对多个 DMA 通道进行仲裁。DMA 组件数据手册已经对 DMA 处理器和相关的 API 进行了简单介绍。

本节主要讨论的问题为：如何构建某个组件以使用 DMA；使用 DMA 传输数据的方法；如何通知传输的结束，以及传输大量数据或仅包含一个字节的小数据包的最佳方法。为了简化配置工作，已经为最终用户提供了一个 DMA 向导。作为开发组件工作的一部分，有关 DMA 的组将功能可能会在 XML 文件中提供。

11.3.1 用于数据传输的寄存器

系统总线通常可以连接到所有 UDB；通过这些连接，DMA 可以访问 UDB 中的寄存器和 RAM，以进行正常的操作和配置。

每个数据通路模块都包含六个 8 位的工作寄存器。CPU 和 DMA 可以对所有寄存器进行读和写操作。

每个数据路径都包含两个 4 字节的 FIFO，可将这些 FIFO 配置为输入缓冲区或输出缓冲区。这些 FIFO 能够生成状态信号，可以对这些信号进行布线，使之与序列发生器、中断或 DMA 请求相交互。将它们配置为输入缓冲区时，系统总线会写到 FIFO，并且数据路径将内部读取 FIFO。将它们配置为输出缓冲区时，数据路径将内部写入到 FIFO，并且系统总线会读取 FIFO。

对于小数据传输，当准备传送或接收单个数据包，并且在传送另一个数据包前需要进行计算数据时，需要使用累加器。

对于需要连续数据流的大数据传输，FIFO 特别有用。可以保留 FIFO 状态逻辑以及连续数据流而不会丢失数据。

类型	名称	说明
累加器	A0、A1	累加器可以作为 ALU 的时钟源或 ALU 输出的目标使用。可以从相应的数据寄存器或 FIFO 加载这些累加器。累加器保存了函数的当前值，如计数值、CRC 或移位。这些寄存器是非保留寄存器；在睡眠模式下这些寄存器会丢失它们中的数值，并在唤醒时被复位为 0x00。
数据	D0、D1	数据寄存器保存了某个给定函数的常量数据，如 PWM 比较值、定时器周期或 CRC 多项式。这些寄存器是保留寄存器。它们的值在各个睡眠间隔中被保留。
FIFO	F0、F1	两个 4 字节 FIFO 为缓冲的数据提供了时钟源和目标。可以将这两个 FIFO 全都配置为输入缓冲器、输出缓冲器，或将其中一个配置为输入缓冲器，将另一个配置为输出缓冲器。可将状态信号布线为数据路径的输出，通过该信号可以对这些寄存器进行读和写操作。所使用的示例包括 SPI 或 UART 中缓冲的 TX 和 RX 数据以及缓冲 PWM 比较和缓冲定时器周期数据。这些 FIFO 是非保留 FIFO。在睡眠模式下会丢失它们的数据，并在唤醒时处于未知状态。FIFO 状态逻辑在唤醒时被复位。

11.3.2 状态寄存器

有四个 FIFO 状态信号，每个 FIFO 使用其中两个：fifo0_bus_stat、fifo0_blk_stat、fifo1_bus_stat 和 fifo1_blk_stat。这些信号的意义取决于已经赋予 FIFO 的方向，该方向由静态配置决定。对于器件系统，bus（总线）状态信号具有重要的意义；通常将该信号布线到中断控制器、DMA 控制器，另外可以通过状态寄存器轮询它。blk 状态信号对内部的 UDB 操作非常有意，通常将该信号布线到 UDB 组件模块（如由各个 PLD 宏单元构成的状态机）。

每个 FIFO 模块所生成的两个状态位都可以用于通过使用数据路径输出复用器对 UDB 进行布线。当系统需要读取或写入字节时，bus 状态信号主要用于控制系统总线，实现与 CPU/DMA 相交互。block（模块）状态主要用于局部控制目的，为内部 UDB 状态机提供 FIFO 状态。状态位的意义取决于配置方向位（Fx_INSEL[1:0]）和 FIFO 电平位。

FIFO 电平位（Fx_LVL）在工作寄存器空间中的辅助控制寄存器内进行设置。下表显示的是它们的各个选项。

Fx_INSEL [1:0]	Fx_LVL	信号	状态	说明
输入	0	fx_bus_stat	未滿	FIFO 中剩余大于 1 字节的空间时，会确认该状态。可以使用它来确认将多个字节写入 FIFO 的系统中断或 DMA 请求。
输入	1	fx_bus_stat	至少有一半为空	FIFO 中至少有 2 字节的空间时，将确认该状态。
输入	N/A	fx_blk_stat	空	FIFO 中没有任何数据字节时，将确认该状态。FIFO 非空时，数据路径函数可以使用这些字节。FIFO 为空白时，控制逻辑可以处于闲置状态或生成欠载状态。
输出	0	fx_bus_stat	非空白	FIFO 中至少有一个可以读取的字节时，将确认该状态。可通过这种情况确认从 FIFO 中读取多个字节的系统中断或 DMA 请求。
输出	1	fx_bus_stat	至少一半为满	FIFO 中至少有两个可以读取的字节时，将确认该状态。
输出	N/A	fx_blk_stat	满	在 FIFO 已满时，将确认该状态。FIFO 未滿时，数据路径函数可以将字节写入 FIFO 内。FIFO 已满时，数据路径可以处于闲置状态或生成一个欠载状态。

11.3.3 多层并行访问路径的数据带宽

DMA 控制器在某个多层并行访问路径上传输的数据大小等于该多层并行访问路径的数据带宽。但是 AHB 规则要求与地址边界对齐的所有数据传输大小要与传输要求相等。因此要求 32 位传输的 ADR[1:0] 必须等于 0b00，16 位传输的 ADR[0] 必须为 0。该地址可以为 8 位传输采取任意值。换句话说，如果在地址边界上开始或结束的整个突发传输不等于多层并行访问路径的数据带宽，那么会创建一个不均匀的开始或结束。

下表提供了与多层并行访问路径相关的外设以及多层并行访问路径的宽度。

PHUB 多层并行访问路径	模拟	多层并行访问路径的数据带宽
0	SRAM	32
1	IO、PICU、EMIF	16
2	PHUB 局部配置、电源管理程序、时钟、中断控制器、SWV、EEPROM、闪存编程接口	32
3	模拟接口、抽取滤波器	16
4	USB、CAN、I2C、定时器、计数器、PWM	16
5	DFB	32
6	UDB 组 1	16
7	UDB 组 2	16

源多层并行访问路径和目标多层并行访问路径的宽度可能不一样。突发引擎将 DMA 控制器内的 FIFO 作为两个多层并行访问路径之间的漏斗机制使用。

11.3.4 FIFO 动态控制说明

通过使用数据路径布线信号，可以在内部和外部访问间进行动态切换。数据路径输入信号 `d0_load` 和 `d1_load` 用于控制该操作。

注意：在动态 FIFO 控制模式下，一般不能使用 `d0_load` 和 `d1_load` 从 F0/F1 加载 D0/D1 寄存器的值。

在特定的使用场合下，动态控制信号（`dx_load`）可以由 PLD 逻辑或任意其他的布线信号（包括常量）控制。例如，CPU 可以从外部访问开始（`dx_load == 1`），将一个或更多的数据字节写入到 FIFO 内。然后切换到内部访问（`dx_load == 0`），数据路径可以对数据进行各种操作。最后返回外部访问，CPU 或 DMA 可以读取计算结果。

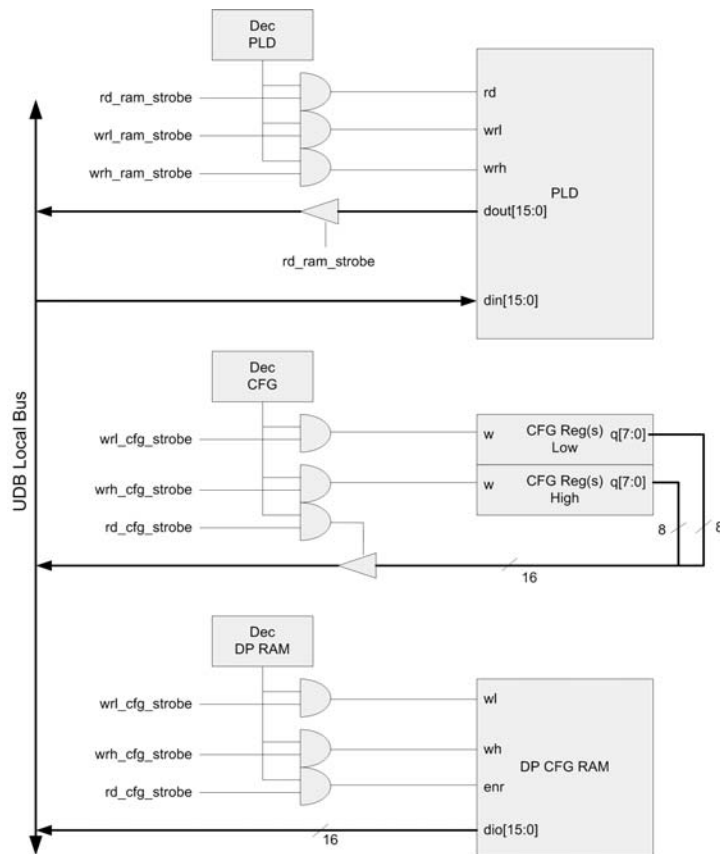
11.3.5 数据路径条件 / 数据生成

条件是由寄存累加器值、ALU 输出和 FIFO 状态信号生成的。可以将这些条件驱动到数字布线，以用于其他 UDB 模块并作为各中断或 DMA 请求使用；也可以将它们驱动到 I/O 引脚上。下表显示的是 16 个可能发生的条件：

名称	条件	是否链接	说明
ce0	“等于”比较	是	$A0 == D0$
cl0	“小于”比较	是	$A0 < D0$
z0	零检测	是	$A0 == 00h$
ff0	“1”值检测	是	$A0 = FFh$
ce1	“等于”比较	是	$A1$ 或 $A0 == D1$ 或 $A0$ （动态选择）
cl1	“小于”比较	是	$A1$ 或 $A0 < D1$ 或 $A0$ （动态选择）
z1	零检测	是	$A1 == 00h$
ff1	“1”值检测	是	$A1 == FFh$
ov_msb	溢出	否	$Carry(msb) \wedge Carry(msb-1)$
co_msb	进位	是	对 MSB 定义位进行进位
cmsb	CRC MSB	是	CRC/PRS 功能的最高有效位
so	移出	是	移位输出选择
f0_blk_stat	FIFO0 模块状态	否	定义取决于 FIFO 的配置情况
f1_blk_stat	FIFO1 模块状态	否	定义取决于 FIFO 的配置情况
f0_bus_stat	FIFO0 总线状态	否	定义取决于 FIFO 的配置情况
f1_bus_stat	FIFO1 总线状态	否	定义取决于 FIFO 的配置情况

11.3.6 UDB 局部总线配置接口

下图显示了用于给 UDB 局部总线接口配置状态的接口架构。



共有 3 种接口类型，包括：PLD、配置锁存器以及数据路径配置 RAM。可以将所有配置编写为 16 位（以支持 DMA）或 16 位处理器操作。同样也可以将它们单独编写为高位（奇地址）字节和低位（偶地址）字节。PLD 具有能够实现 RAM 的读和写时序操作的独特读取信号。CFG 寄存器和 DP CFG RAM 共同使用了相同的读和写控制信号。

11.3.7 UDB 对寻址

使用 DMA 的数据传输方法取决于对工作寄存器和配置寄存器的配置情况。在 UDB 对中有 3 个独特的地址空间。

- **8 位工作寄存器：**它是一个总线主设备。在每个总线周期中它只能访问 8 位数据。通过该地址空间，可以对任意 UDB 的工作寄存器进行读 / 写操作。在模块进行正常操作的过程中，这些寄存器会与 CPU 固件和 DMA 相交互。
- **16 位工作寄存器：**它是一个具有 16 位功能的 16 位总线主设备。在每个总线周期中，它能够访问 16 位数据，这样会便于传输 16 位或更大的数据。与 8 位模式相比，虽然该地址空间被映射到不同的区域，但仍能访问相同的 8 位 UDB 硬件寄存器，有两个寄存器响应该访问操作。
- **8 位或 16 位的配置寄存器：**这些寄存器将配置 UDB，使之执行某项功能。配置这些寄存器后，它们在实现功能时通常处于静态。睡眠时，这些寄存器仍能保持它们的状态。

11.3.7.1 工作寄存器地址空间

模块进行正常操作时会访问工作寄存器，这些寄存器包括累加器、数据寄存器、FIFO、状态和控制寄存器、屏蔽寄存器以及辅助控制寄存器。下图显示的是一个 UDB 的寄存器映射情况。

8-bit addresses		16-bit addresses
UDB Working Base +		
0xh	A0	0xh
1xh	A1	2xh
2xh	D0	4xh
3xh	D1	6xh
4xh	F0	8xh
5xh	F1	Axh
6xh	ST	Cxh
7xh	CTL/CNT	Exh
8xh	MSK/PER	10xh
9xh	ACTL	12xh
Axh	MC	14xh
Bxh		16xh

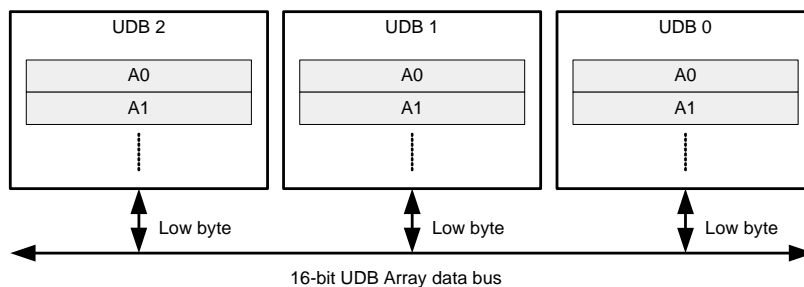
请注意，可以将 UDB 作为 8 位或 16 位对象进行访问，其中每次访问方法会使用一个不同的地址空间。

左侧显示的是 8 位地址方案，其中寄存器编号范围为高位半字节，UDB 编号范围则为低位半字节。通过该方案可以使用 8 位地址访问 16 个 UDB 的工作存储器。

右侧显示的是 16 位地址，该地址始终为偶对齐的。由于偶对齐的特性，UDB 编号范围为 5 位，并非为 4 位。高 4 位仍是寄存器编号。在 16 位数据访问模式下，为了访问 16 个 UDB，总共需要使用 9 个位地址。工作寄存器被组织成 16 个 UDB 组。

11.3.7.2 8 位工作寄存器访问

在 8 位寄存器访问模式下，所有 UDB 寄存器都按字节对齐的地址进行访问，如下图所示。被写入到 UDB 中的所有数据字节都与 16 位 UDB 总线的低字节相对齐。

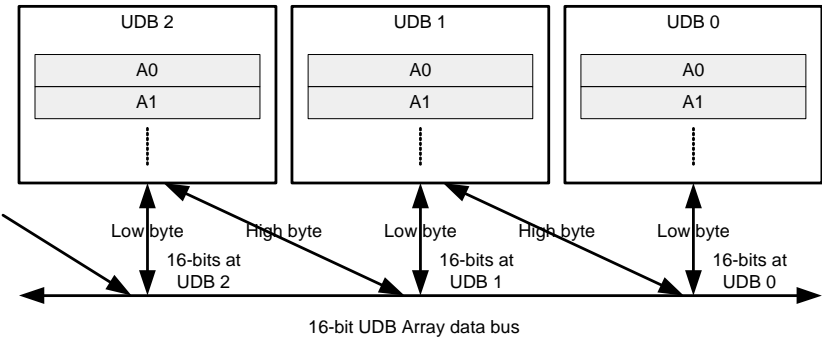


在该模式下，每次只能访问一个字节，PHUB 会自动使有效的奇（高）字节或偶（低）字节与处理器或 DMA 相对齐。

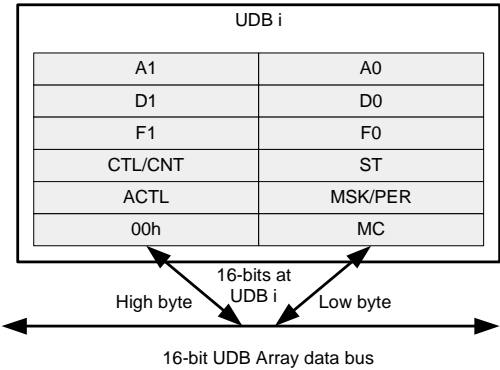
11.3.7.3 16 位工作寄存器地址空间

16 位地址空间是针对有效地访问 DMA 而设计的 1（6 位数据带宽）。16 位寄存器访问可以按以下两种模式进行：“default”（默认）模式和“concat”模式。

如下图所示，默认模式将访问 UDB ‘i’ 中低字节的特定寄存器以及 UDB ‘i+1’ 中高字节的相同寄存器。这样便能够在被配置为 16 位功能的相邻 UDB 中有效地处理 16 位数据。



下图显示的是“concat”模式，其中单个 UDB 的寄存器联合构成 16 位寄存器。在该模式下，16 位 UDB 阵列数据总线将访问 UDB 中的各对寄存器。例如，如果访问寄存器 A0，则会返回低字节的 A0 和高字节的 A1。



11.3.7.4 16 位工作寄存器地址的限制

使用与 16 位工作寄存器地址空间相关的 DMA 收到一定的限制。通过优化该地址空间，DMA 和 CPU 可以访问 16 位功能的 UDB。对于功能超过 16 位的 UDB，该地址空间的使用效率较低。这是因为地址会被重叠，如下表所示：

地址	高字节被写入到	低字节被写入到
0	UDB1	UDB0
2	UDB2	UDB1
4	UDB3	UDB2

当 DMA 将 16 位数据传送到地址 0 时，低字节和高字节会分别被写入到 UDB0 和 UDB1。下一次 DMA 将 16 位传送到地址 2 时，会使用所传输的低字节覆盖掉 UDB1 中的值。要避免这种情况，必须提供存储器缓冲区中的冗余数据组织来支持该地址，推荐将 8 位工作寄存器空间中的 8 位 DMA 传输应用于功能超过 16 位的 UDB。

11.3.8 DMA 总线的使用

DMA 控制器具有双重的使用背景，它能够进行流水线操作，因此可并行活动。通常情况下，由于进行数据传输，在各个 AHB 的总线周期内多层并行访问路径总线可以达到几乎 100% 的使用率。

处理通道的开销一般被隐藏在数据突发背景中。可以在发生某个通道的数据突发时对另一通道进行仲裁、提取数据并更新等操作。此外，如果没有任何多层并行访问路径的争用、源的争用（SRC）或 DMA 控制器的目标（DST）引擎，那么某一通道的数据突发可以与其他通道的数据突发相重叠。

11.3.9 DMA 通过到突发时间

通道突发时间被定义为从 SRC 多层并行访问路径上的第一个请求到 DST 多层并行访问路径上就绪的最终请求所需要的时钟周期数量。理想的突发时间包括一个初始控制周期和各个跟随的数据突发周期（含流水线式的并行后续控制周期）。因此，多层并行访问路径上理想的突发时间是用于传输 N 个数据块的 $N + 1$ 个时钟周期，其中 N 是突发长度 / 外设宽度。

下面各个变量都能影响 N 的值：

- 在 DMA 控制器背景管道出现前已经存在另一个通道背景，以及预留给该通道的突发条件。
- 通道的中下述突发条件：
 - 与 CPU 争用多层并行访问路径
 - SRC/DST 外设就绪
 - SRC/DST 外设的宽度
 - 突发的长度
 - 不齐整的开始和结束
 - 内部 spoke 和互联 spoke DMA

内部 spoke DMA 要求先将整个 SRC 突发数据缓存在 DMA 控制器的 FIFO 中，然后将这些数据写入到相同的 spoke 内。这样会发生两个 $N+1$ 长度的数据突发，因此理想的内部 spoke 突发时间长度的通用公式为 $2N+2$ 。

内部 spoke DMA 允许 SRC 和 DST 的数据突发重叠。当读取 SRC spoke 的数据并将数据写入到 DMA 控制器 FIFO 内时，DST 引擎可以将 FIFO 的可用数据写入到 DST spoke 内。该重叠情况会使内部 spoke DMA 更加有效。最终可以使用 3 个开销周期来将一个数据块从一个 spoke 移动到另一个。（由于传输每一个 spoke 上的每一个数据块都需要占用一个数据周期），将每个 spoke 上的初始控制周期加上“冗余”的数据周期。因此，理想内部 spoke DMA 突发时间的通用公式为 $N+3$ ，这样才能传输 N 个数据块。

下表显示的是一些理想突发示例的周期：

数据传输操作 (按 Spoke 大小)	内部 Spoke DMA 传输相位 (时钟周期)	交互 Spoke DMA 传输相位 (时钟周期)
1	4	4
2	6	5
3	8	6
N	2N+2	N+3

11.3.10 组件 DMA 的功能

想使用 PSoC Creator DMA 向导的任意组件，要求具有包含其 DMA 功能的 XML 文件。XML 文件格式必须包含反应实例特定信息的功能。应该使用静态 XML 文件来实现。该文件包含基于实例参数的设置。请参见[位于第 115 页上的添加 / 创建 DMA 功能文件](#)。

注意：每个组件只能添加一个 DMA 功能文件。

11.4 低功耗支持

作为一般规则，通过提供用于保存退出低功耗模式时会被丢失的非保留寄存器值及用户参数的 API，组件可提供低功耗支持。进入低功耗模式前，会调用用于保存寄存器值和参数的 API。退出低功耗模式后，将调用一个恢复寄存器值和参数的 API。保存的特定寄存器是设计中使用的寄存器功能。技术参考手册 (TRM) 指定了非保留寄存器。进入低功耗模式时，只用保存非保留寄存器。

11.4.1 功能的要求

根据具体的组件，提供一个静态的数据结构来保持非保留的寄存器值。只有需要时，才支持低功耗模式功能。这样可使所有组件的接口统一起来。数据结构和功能需要用于初始化、保存 / 恢复和睡眠 / 唤醒工作的模板已经进行了定义。所有这些功能都是全局的。保存 / 恢复功能可能使用于低功耗以外的情况。

11.4.2 设计注意事项

需要时，请定义低功耗保留的函数。可在单独文件 ``${INSTANCE_NAME}`_PM.c` 中找到这些函数。如果应用不使用低功耗的功能，则这样允许在连续时间内移除带有静态数据结构的 `‘.o’` 文件。此外，``${INSTANCE_NAME}`_Enable()` 和 ``${INSTANCE_NAME}`_Stop()` 函数会使能 / 禁用备用有效寄存器使能功能。这样便提供了一个机制用于自动使能和禁用备用的有效模板。

11.4.3 固件 / 应用编程接口要求

11.4.3.1 数据结构模板

```
typedef struct _`${INSTANCE_NAME}`_BACKUP_STRUCT
{
    /* Save component's block enable state */
    uint8 enableState;

    /* Save component's non-retention registers */
}
```

```
} `$_INSTANCE_NAME`_BACKUP_STRUCT;
```

11.4.3.2 保存/恢复方法

将非保留寄存器的值保存为静态数据结构。只用保存特定组件的寄存器值。

```
`$_INSTANCE_NAME`_SaveConfig()
{
/* Save non-retention register's values to backup data structure. */
}
```

从静态数据结构恢复非保留寄存器的值。只用恢复特定组件的寄存器值。

```
`$_INSTANCE_NAME`_RestoreConfig()
{
/* Restore non-retention register values from backup data structure. */
}
```

保存组件的使能状态。使用该状态来确定是否在唤醒时启动组件。停止该组件并保存配置。

```
`$_INSTANCE_NAME`_Sleep()
{
/* Save component's enable state - enabled/disabled. */
if(/* Component's block is enabled */)
{
    backup.enableState = 1u;
}
else /* Component's block is disabled */
{
    backup.enableState = 0u;
}
`$_INSTANCE_NAME`_Stop();
`$_INSTANCE_NAME`_SaveConfig();
}
```

恢复组件配置并确定是否应该使能组件。

```
`$_INSTANCE_NAME`_Wakeup()
{
    `$_INSTANCE_NAME`_RestoreConfig();
/* Restore component's block enable state */
if(0u != backup.enableState)
{
    /* Component's block was enabled */
    `$_INSTANCE_NAME`_Enable();
} /* Do nothing if component's block was disabled */
}
```

11.4.3.3 添加使能和停止功能

使能组件的备用激活寄存器使能。

```
`$INSTANCE_NAME`_Enable()
{
    /* Enable block during Alternate Active */
}
```

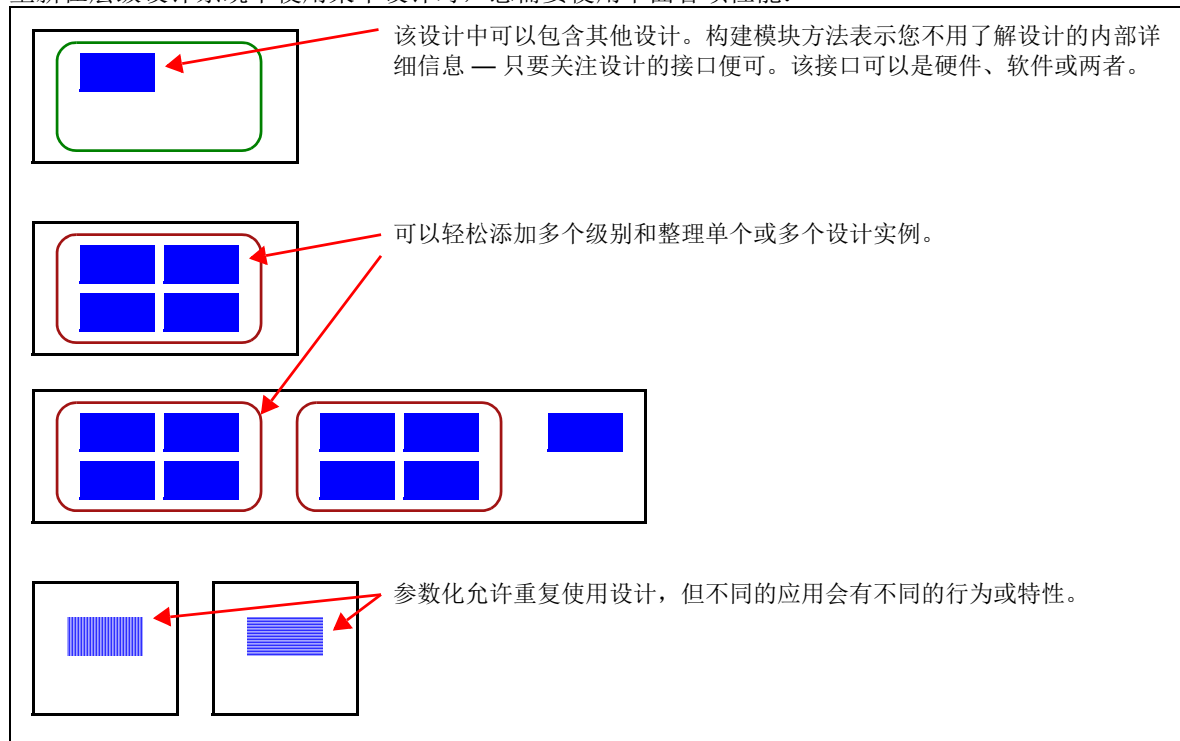
禁用组件的备用激活寄存器使能。

```
`$INSTANCE_NAME`_Stop()
{
    /* Disable block during Alternate Active */
}
```

11.5 组件封装

11.5.1 层级设计

重新在层级设计系统中使用某个设计时，您需要使用下面各项性能：

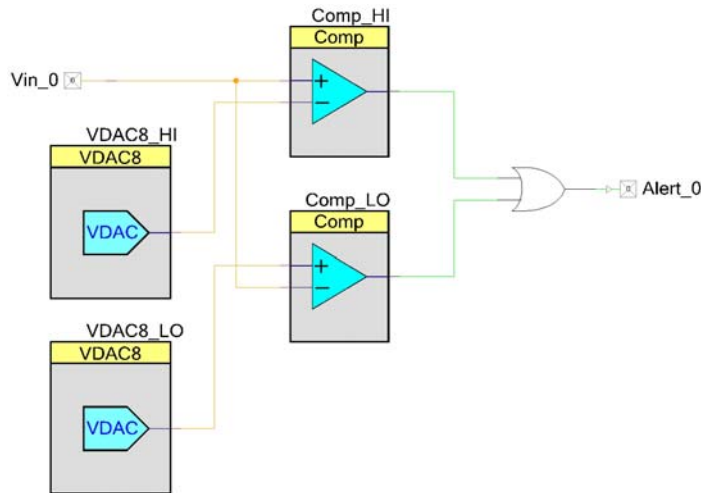


为了能重新在 PSoC Creator 中轻松使用设计，应该将该设计包装为 PSoC Creator 组件。如果某一设计符合下面一项或多项标准，那么可以封装和重新使用它：

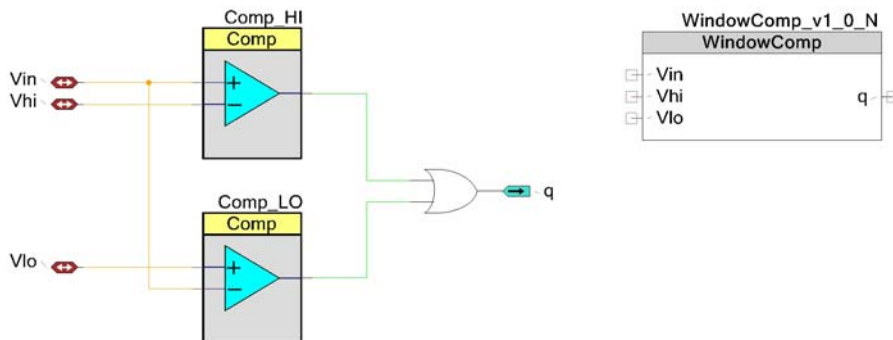
- 实现某个特定功能。通常的规则是“专而精”。
- 具有以硬件终端或 API 调用为形式的有限和相对小的输入和输出组。通用规则越少越好，但不能过少，否则会降低基本功能。

下面各页介绍了一些示例：

在说明何时封装一个作为组件 IP 的简单示例中，如果您需要在设计中提供一个窗口比较器，请考虑您可能要进行的操作。当输入电压处于两个比较电压值之间的范围内时，窗口比较器将被激活。对于 PSoC Creator，大部分设计如下：

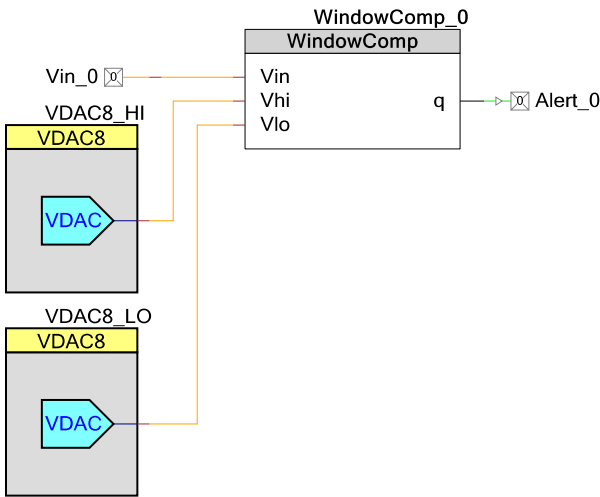


这种设计非常有利于将其封装为一个组件。它只实现一个特定功能，即用作窗口比较器。此外，它具有一个有限且较小的输入和输出集。它还具有较小的 API，用以启动比较器。因此，可以使用一个符号将该设计的基本必要功能封装为一个组件，如下所示：

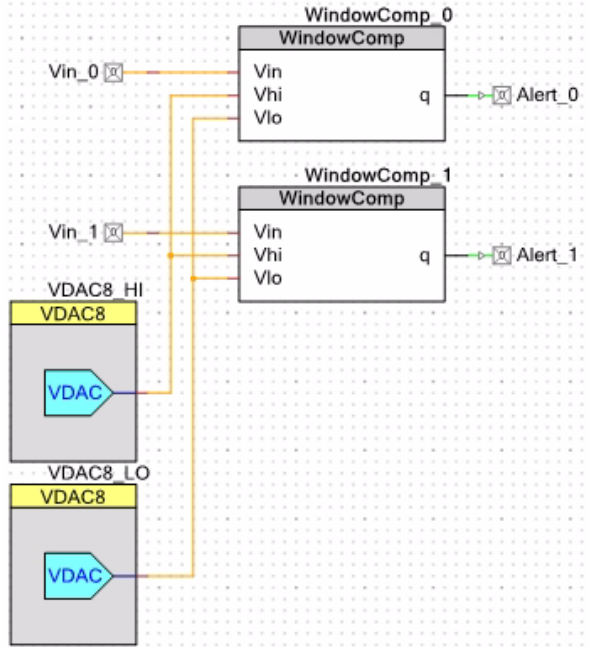


包装某个设计时，必须考虑移除组件中的哪个成分。在上述示例中，可以将各个 VDAC 封装在组件中。但是，它们并不是该设计的重要组成部分，它们只是对一个电压值与两个参考电压值进行比较。参考电源可以由 VDAC 或某些其他源提供。因此，在这种情况下，最好移除 VDAC。

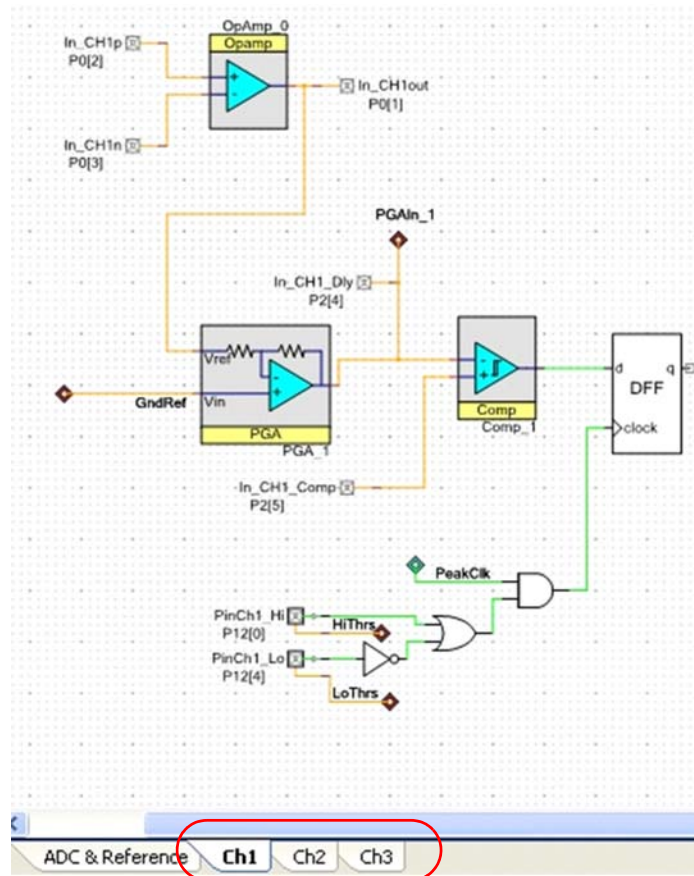
通过包装形式，您的顶层设计显得更加简单：



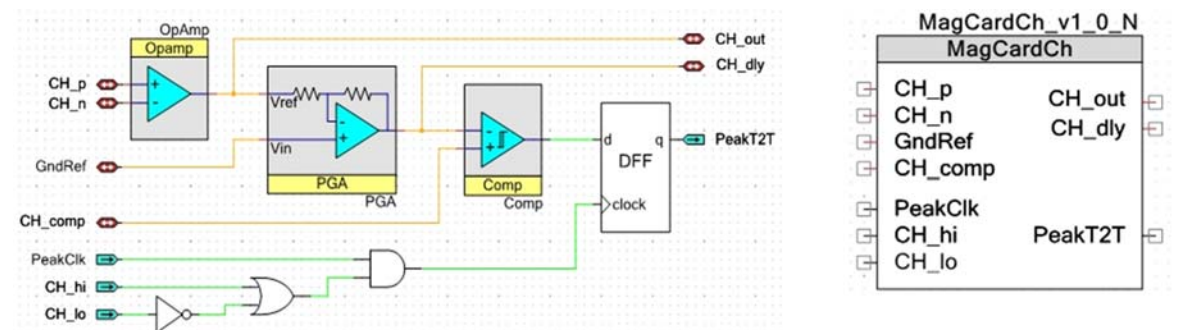
在多种情况下，也能更加轻松地扩展该设计：



下面介绍的磁卡读卡器是一个更加复杂的示例。在该设计中，存在一个中央 ADC、参考部分以及多个电路副本，用于读取单个通道：



此时，该设计的基本功能就是将各个模拟输入从该通道转换为数字系统能读取的形式。此外，由于这是一个已定义且有限的功能，因此应该使用它进行包装：



您还需要考虑移除组件中的哪个成分。原始设计使用两个 SIO 引脚和它的两个不同阈值。可以将这些引脚封装在组件中，并且最好在顶层设计中。其次，基本设计只需要数字信号 CH_h 和 CH_lo，而不需要它们的源。

此外，由于 PSoC 3/5 运算放大器与特定器件的引脚密切相关，因此使该运算放大器放置在组件范围外会更好。

最后，该设计利用了有限的模拟资源和路由，另外所使用的多个实例可能不适合某些较小的 PSoC 3/5 设备。用户应该了解上述情况以及可能使用的电压、温度或速度的限制。例如，PeakClk 的频率应该为多大？在原始设计的顶层可以知道该频率的值，但是重新使用该组件时，可能已经忘记了该值。

这种情况可引起一个很有趣的问题：什么时候不要包装某个设计。如果某个设计满足下面一个或多个条件，那么包装设计可能得不到一个好的实践。如果设计被封装，那么用户应该了解相关问题和限制，以便在尝试再次使用组件时，他们考虑到这些问题。

- 将关键资源添加到 PSoC 3/5 中，这样，在更高级别的设计中，不能使用这些资源。特定主题包括：
 - 使用单实例固定功能模块，如 ADC_DelSig、CAN、USB 和 I²C
 - 频繁使用的大量资源，如 UDB、比较器、DAC、运算放大器、定时器、DMA 通道、终端、引脚、时钟等
 - 频繁使用的不太明显的资源，如模拟或 DSI 路由、闪存、SRAM 或 CPU 周期
- 仅在特定条件下进行操作，例如，CPU 或 bus_clk 速度，或 Vdd 电平或仅与特定器件（如 PSoC 5 系列）配合工作。
- 不能执行多个 IP 实例。

11.5.2 参数化

在 PSoC Creator 中，可以对组件进行参数化处理。因此可以在构建时设置组件的实例行为（通常通过使用对话框）。通过参数化处理，可以设置硬件和软件行为。应该在下面场合中进行参数化处理：

- 不同 IP 实例的行为略有不同，但总体功能不变。例如，可以将风扇控制器上的警报输出设置为高电平有效或低电平有效。
- 预期行为差异不会在运行时发生变化。

如果参数化使不同实例的功能发生大变化，那么您应该使用多个组件包装设计。PSoC Creator 允许单一库项目包含多个组件，因此可以保持组件的多种版本。例如，单一风扇控制器组件可能拥有一个参数用来控制多个风扇。具有两个不同接口（如 SPI 和 I²C）的风扇控制器可能是“风扇控制”库项目中的两个单独的组件。

11.5.3 组件设计中的注意事项

11.5.3.1 使用资源

组件中不应该嵌入引脚组件，而应该使用终端；这样用户可以将引脚连接到更高级别的组件符号。

一个组件中可以包含时钟组件，也可以不包含。不嵌入时钟的优点是多个组件可以连接到相同的时钟，因此能够节省时钟资源。一个组件可以包含一个参数选项，用于选择内部或外部时钟。

一个组件也可以选择包含中断和 DMA 通道，也可以不包含。一个组件可能生成一个“含数据”信号，该信号可以连接到 IRQ 或 DRQ；在这种情况下，不应该将中断和 DMA 通道嵌入到组件内。

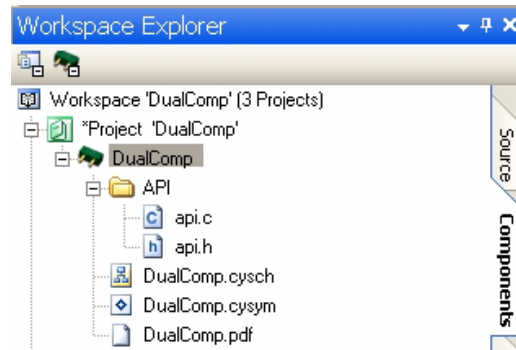
11.5.3.2 电源管理

需要设计组件支持电源管理操作，比如：睡眠、休眠和唤醒。API 应包含相应的函数。赛普拉斯的组件提供了多个示例以解释如何实现电源管理 API。

11.5.3.3 组件开发

PSoC Creator 组件基本上都是一个一个的容器。用户既不能编译它们，又不能在目标器件中安装它们。主要是通过它们连接到标准的 PSoC Creator 项目，然后对这些项目进行编译和安装。

组件可以包含几种文件类型，如：符号（.cysym）、原理图（.cysch）、Verilog（.v）、固件源代码（.c、.h、.a51、等等）以及文档（.pdf、.txt）。各个组件也可以参照其他组件，以此实现层次设计技术。



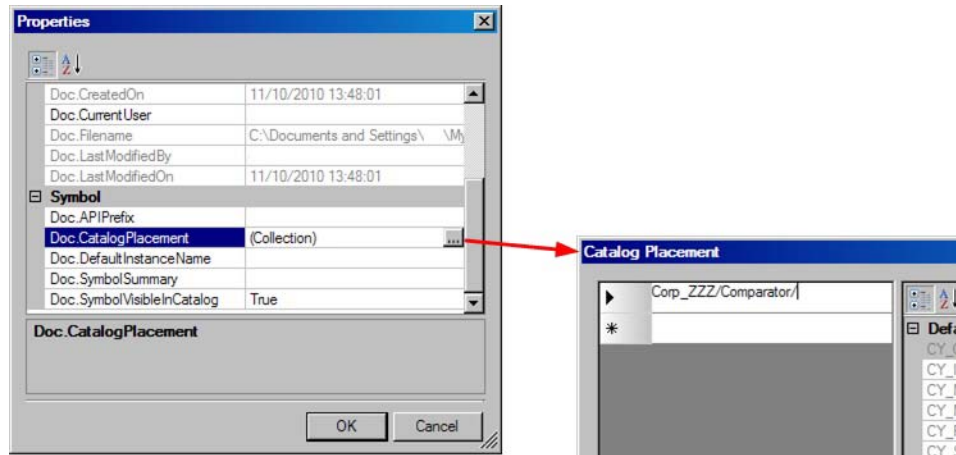
按照该规范开发的组件至少要拥有一个符号文件和一个数据手册文件。根据组件的功能来选择其他文件类型。例如，单硬件组件可能带有原理图文件或 Verilog 文件，而单固件组件可能带有一个 .h 和 .c 文件，用于保存 API。

参考组件符号

符号应始终包含示例名称 `=\$INSTANCE_NAME`（向后的单引号）注释在内。另外，还应该添加其他注释内容，这样用户能够更快了解组件的功能。

组件目录放置

符号属性 `Doc.CatalogPlacement` 控制着所需组件在 PSoC Creator 组件目录中所在的位置。尤其是创建多个组件时，应使用一致的选项卡和树形节点命名方案。推荐使用更少的选项卡和更多的树形节点，以创建一个能适应大多数用户的屏幕组件目录。



更多有关信息，请参见 [位于第 29 页上的定义目录位置](#)。

组件数据手册

为了正确记录您的组件，应该包含一个数据手册。该数据手册要注明上述设计注意事项。更多有关添加一个数据手册的信息，请参见[位于第 105 页上的添加 / 创建数据手册](#)。

组件版本

组件名称可以包括版本信息；通过将下面的信息添加到组件名称中来实现：

“_v<major_num>_<minor_num>_<patch_level>”

<major_num> 和 <minor_num> 都是整数，分别表示主要版本号和次要版本号。<patch_level> 是单一的小写字母字符。组件应该被版本化。有关版本化的信息，请参考[位于第 14 页上的组件版本](#)

11.5.3.4 测试组件

在 PSoC Creator 库项目中，将可重复使用的设计作为一个组件使用。然而，库项目或组件本身并不太有用。不能对库项目进行编译、编程或测试。因此，需要一同开发一个或多个标准项目和组件，这样可以测试或演示组件中可重复使用设计的功能。

应该在设置不同的情况下对参考组件的参数进行测试，并且测试的设置情况越多越好。请注意，为实现该操作，要使用多种组件的实例，或要创建多个测试 / 演示项目。组件的所有 API 函数至少要调用一次，另外全部宏也应该至少要使用一次。

测试项目需要支持尽可能多的 PSoC 3 和 PSoC 5 的配置。例如：标准与可引导加载、调试与发布、不同的 PSoC 5 编译器以及不同编译器优化设置。

11.6 Verilog

很多数字组件都采用了 Verilog 来定义组件的实现。更多有关信息，请参见 [位于第 61 页上的使用 Verilog 实现硬件](#)。

必须将该 Verilog 代码分成可合成的 Verilog 子集。除了符合 Verilog 子集外，当创建基于 Verilog 的组件时还要遵循其它指南。很多指南是所有合成工具的最佳实践。当开发 PSoC Creator 组件时，推荐使用特定的其它指南。

11.6.1 Warp: PSoC Creator 合成工具

当处理设计中全部数字逻辑时，PSoC Creator 会自动运行安装 PSoC Creator 时所附加的 Warp 合成工具。这是 Warp 合成工具的 PSoC 特定版本，赛普拉斯可编程逻辑器件已经使用了该工具。

PSoC Creator 支持的可合成特定 Verilog 子集在 *Warp Verilog 参考指南*中进行了介绍。该可合成子集与其他 Verilog 合成工具中实现的子集相同。有关受支持的 Verilog 结构的详细说明，请参考指南。

Verilog 设计的可合成部分会被合成为 PLD 逻辑。剩下的 UDB 资源可用于 Verilog 设计，但无法通过合成过程生成它们。

11.6.2 可合成的编码指南

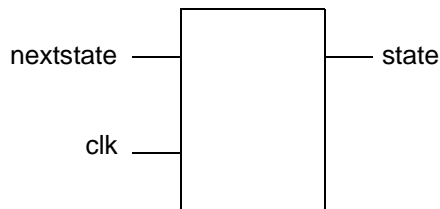
11.6.2.1 阻塞与无阻塞赋值

Verilog 语言的赋值语句具有两种形式。在进行下一个语句前，阻塞赋值语句要及时分配值。无阻塞赋值语句则会经过延迟时间后才会分配值。从仿真的角度来看，这两种赋值类型拥有具体的意义。但从合成方面和得到的硬件的角度来看，这些分配会产生不同的结果。对可合成设计的最佳实践进行开发，以便使仿真结果与合成结果相匹配。

使用无阻塞赋值实现序列逻辑

如果使用该规则，那么时钟逻辑的仿真结果与硬件可实现匹配。在硬件实现中，需要使所有结果进入新的状态，然后才使用该状态计算下一个状态。

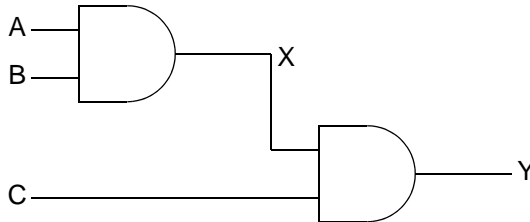
```
always @(posedge clk)
begin
    state <= nextState;
end
```



使用阻塞赋值实现 “always” 模块中的组合逻辑

该规则会使：计算 “always” 模块时，仿真结果会立即给信号分配数值。当忽略组合逻辑的延迟时，仿真结果会与硬件实现匹配。在这种情况下，会立即使用一个逻辑级别的结果进行计算下一个逻辑级别。

```
always @(A or B or C)
begin
    X = A & B;
    Y = X & C;
end
```



不能在同一个 “always” 模块中混合使用阻塞和无阻塞赋值

该规则指明：应在与相续 “always” 模块独立的模块中编写需要多个赋值来实现的组合逻辑。在单独的组合模块或连续的赋值语句中可以实现该组合逻辑。

在 “always” 模块中，请勿为相同变量创建赋值。

PSoC Creator 中的合成引擎会执行该规则，如果违背了该规则，那么它会生成一个错误。

11.6.2.2 Case 语句

在 Verilog 中，case 语句有三种形式：case、casex 以及 casez。编码这些语句时应遵循下面规则。

完全定义所有 case 语句

将 default case 放在所有 case 语句中是最好的方法。这样可自动满足该规则。如果放置好所有 case 后仍然添加了 default 语句，那么在后面的代码开发过程中即使更改了 case 项，也可满足该规则。这是因为即便是删除了 case 项或更改 case 的宽度，则仍存在 default 语句。

在组合的 “always” 模块中，该规则特别重要。包含默认条件（带有分配的组合结果）会阻止设计中锁存器的合成。

使用 **casez** 替换 **casex**

casex 和 **casez** 语句是相同的，合成结果也一样。它们的差别只是仿真过程中所生成的结果。当 **case** 项指定了一个 “无需关注” 位时，**casex** 语句会与 “x” 或 “z” 的输入值相匹配；但对于 **casez** 语句，则只有 “z” 值会与 **case** 项中指定的 “无需关注” 位相匹配。对于 **casex** 语句，仿真过程会遗漏设计中未初始化值的错误。对于 **casez** 语句，这不是什么问题，因为 PSoC 设计的可合成部分不会使用 “z” 值。

在 **casez** 语句中 “无需关注” 位用 “?” 来表示

该规则对合成代码没有任何影响。这样只是为了清楚地表明所指定的 “?” 是 “无需关注” 位。而使用 “z” 值的另一种方法并没有这样的意思。

请勿在使用重叠条件时使用 **casez** 语句

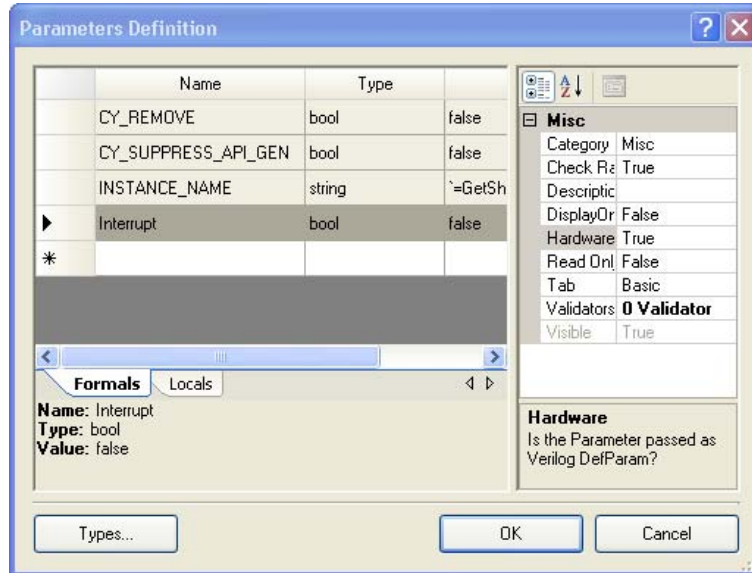
同时使用重叠 **case** 项和 **casez** 语句将会引起优先级编码器的合成。结果逻辑与仿真结果相同，但很难确定预期逻辑。通过使用 **if-else if-else** 语句，可以明确传达优先级编码器含义。

11.6.2.3 参数处理

根据组件实例的具体要求，通过参数可以合成 Verilog 实例。

将参数传递给组件

通过将 Misc 域中的 “Hardware” 设置为 True，可将符号上配置的组件参数传递给 Verilog 实例。



如果一个参数被传递给 Verilog 模块，那么执行合成操作会用到参数设置。结果是根据参数设置将要合成的特定实例。这种方法适用于编译时进行的设置，但不能用于运行时所确定的设置。可在运行过程中更改的设置需要通过数据路径或控制寄存器进行软件控制。确定参数是静态的还是动态的（运行时间可更改）会影响组件的复杂性和资源需求。

拥有一个硬件参数的基于 Verilog 的组件会通过参数语句访问该参数。会自动在 Verilog 模板中创建该参数语句，使用符号可以自动创建该模板。初始值始终由设置给组件的特定实例的值覆盖掉。

```
parameter Interrupt = 0;
```

生成语句

通常，参数值会引起基于该值的 Verilog 代码不同。通过使用一个 Verilog 生成的语句可以实现该功能。例如，如果组件使用了 8 位或 16 位实现，那么需要实例化一个 8 位或 16 位数据路径组件。生成语句是实现该功能的唯一方法。

```
parameter [7:0] Resolution = WIDTH_8_BIT;

generate
  if (Resolution == 8) begin : dp8
    // Code for 8-bit
  end
  else begin : dp16
    // Code for 16-bit
  end
endgenerate
```

将参数传递到模块实例中

Verilog 设计需要将参数传递给设计中其他进行实例化的模块。多个标准的硬件模块需要采用这些参数。

有两种方法可将参数传递给 Verilog。最初的方法是使用 `defparam` 语句。另一种方法则是在 Verilog-2001 规范内，可以使用命名的参数传送参数。模块实例中的 “#” 后面表示的是参数名和参数值。建议使用命名参数传送该参数。

```
cy_psoc3_statusi #(.cy_force_order(1),  
    .cy_md_select(7'h07), .cy_int_mask(7'h07))  
stsreg(  
    .clock(clock),  
    .status(status),  
    .interrupt(interrupt)  
);
```

参数化的数据路径实例

根据单个复杂配置参数配置数据路径实例。对于多字节数据路径配置或对于使用生成语句支持多个数据路径宽度的组件，多个配置参数会使用相同的配置值。重复相同的信息容易发生错误，并且难以维护，因此可以使用某个参数值，以便将数据配置在一个位置。

```
parameter config0 = {  
    `CS_ALU_OP_PASS,  
    // Remainder of the value not shown here  
};  
  
cy_psoc3_dp8 #(.cy_dpconfig_a(config0)) dp0 (  
    // Remainder of the instance not shown here  
);
```

11.6.2.4 锁存器

PSoC 设计中不应该使用锁存器。一些可编程逻辑器件的架构拥有内置锁存器。但 PSoC 器件并没有锁存器。每个宏单元可以在一个时钟边沿上进行组合或寄存。如果在 PSoC 设计中创建了一个锁存器，那么会采用交叉耦合逻辑来实现锁存器。采用锁存器需要创建循环结构，因此会限制时序分析功能。基于组合锁存器的实现可能发生时序问题，因为反馈路径的长度不受控制。

当设计中有锁存器时通常发生的情况：该锁存器并不是预期的功能。在对长久可组合模块的输出进行合成过程中会生成一个锁存操作。长久模块至少存在一个传输过该模块的路径，并且未对该路径的输出赋值。if-else 链中包含最终的 else 子句，并且为每个 if-else 子句的所有输出分配值，从而可以避免上述问题。另外，通过将一个默认值分配给 always 模块顶部上的所有输出也可以避免该问题。

11.6.2.5 复位和设置

复位和设置信号时需要遵循下面规则。这些规则和说明都采用了复位术语，实际上该规则适用于复位和设置功能。

使用同步复位（若可以）

一个异步复位与从控制信号到所有受其影响的寄存器输出的组合路径的功能相类似。通过使用异步复位，复位信号传输到逻辑的时序会成为时序的限制。使用一个同步复位时，附加时序分析会表示复位信号与相应时钟的关系。

时钟自由运行时会使用同步复位

只在时钟不是自由运行的情况下才会使用异步复位。如果时钟静止时需要复位一个寄存器，那么需要采用一个异步复位信号。如果是一个自由运行时钟，则同步复位会产生同样的结果，而不会发生异步信号的时序问题。

不能将异步复位和设置使用于同一个寄存器

PLD 宏单元的硬件实现会采用单一的信号进行异步复位和设置。可以选择使用该信号进行复位、设置，还是不使用该信号。硬件中没有提供异步复位和设置的选项。

11.6.3 优化

11.6.3.1 性能优化

一个典型数字组件的性能是通过两个寄存器间可组合最长路径来确定的。该可组合路径所需时间是将 Verilog 设计映射到有效硬件的时间。

寄存动态配置地址

在使用了某个数据路径的设计中，最长的路径通常开始于数据路径的累加器或宏单元的触发器，经过可组合逻辑（ALU、条件生成或 PLD 逻辑），最终结束于动态配置地址输入。为了提高性能，该路径一般被分为两个更短的路径。一般将寄存器插入到该路径中输入动态配置的前方。在大多数情况下，该输入是由一个宏单元输出驱动的。由于所有宏单元输出均有可选寄存器，所以寄存该输出不会增加组件使用资源。按照这种方式进行传输会改变组件的操作，因此组件的初始架构定义应该包含这种实现。

寄存有条件输出

PLD 中每一个宏单元都有一个可选使用的触发器。与此相同，数据路径生成的每个条件可作为一个组合信号或一个寄存值使用。可以使用这些寄存器进行流水线设计，却不会影响使用资源。

寄存输出

根据典型的组件使用模型，有利于寄存组件的输出。如果存在多个输出，并且将它们传送给引脚，那么寄存这些输出会使输出信号间的时序关系的预测更准确。如果将组件的输出提供给其他组件，那么系统的性能则取决于该组件的输出时序和目标组件的输入时序。如果输出被寄存，那么该路径的输出部分将被最小化。

分离 PLD 路径

每个 PLD 都拥有 12 个输入和 8 个乘积项。如果一个输出需要在单一的 PLD 中可以实现更多的输入或乘积项，则输出的公式会被分为多个 PLD。这样，所需 PLD 路径的每一个额外级别都会添加一个 PLD 传输延迟和路由延迟。为了提高性能，所有级别会按流水线的方式进行计算。不需要增加使用资源，因为每个宏单元输出都有可用的寄存器。但流水线操作会更改逻辑的时序关系。

为了确定用于计算具体输出的输入数量，需要计算 if 语句、case 语句和赋值语句中所使用的全部输入。

11.6.3.2 大小优化

通常，所设计的可编程数字部分的大小受 PLD 资源或数据路径资源的限制。数据路径资源中的等效逻辑功能比 PLD 阵列中存在的逻辑功能大得多。如果一个数据路径的实现是合理的，那么该实现可能是资源最有效的实现方法。当编译 PLD 的逻辑时，必须考虑到 PSoC PLD 的特定架构。

使用具有 12 个输入的 PLD 架构

PSoC PLD 拥有 4 个输出和 12 个输入。合成设计时，每个输出所需要的输入量不可大于 12。如果最终的输出需要多于 12 个输入，则该功能会被分成多个逻辑部分，其中每个部分需要 12 个或更少的输入。所有逻辑被分为需要 12 个或更少输入的输出后，该逻辑会封装在 PLD 中。理想的情况是使用一个 PLD 生成 4 个输出。有 4 个输出的公式在组合时仅需要 12 个输入，这样的实现可行。例如，一个输出需要使用所有 12 个输入，那么只有其他输出需要使用相同输入时，才可放置在 PLD 中。

第一步是构建组件，然后通过记录文件中观察包装 PLD 的平均统计信息。

```

2629 -----
2630 PLD Packing Summary
2631 -----
2632 Packed 104 macrocells into 51 PLDs.
2633
2634     PLD Resource Type :      Average/L&B
2635     =====
2636           Inputs :           11.31
2637           Pterms :           2.02
2638           Macrocells :       2.04

```

如果宏单元的平均数量显著低于 4.0，并且输入数量接近 12.0，那么您设计中所需的输入数量便受限。为了改进封装性能，需要减少所需的输入数量。

在某些情况下，可以通过重组公式来将一组输入合成为一个输入，从而减少输入数量。例如，如果多比特比较是一个输入，则可以使用一个结果输入代替它。如果使用该值来计算多个输出（多比特寄存器），那么强制逻辑的特定部分可以明显降低设计大小。要想将一个信号强制作为宏单元的输出，请使用 `cy_buf` 组件。`cy_buf` 的输出始终是宏单元输出。

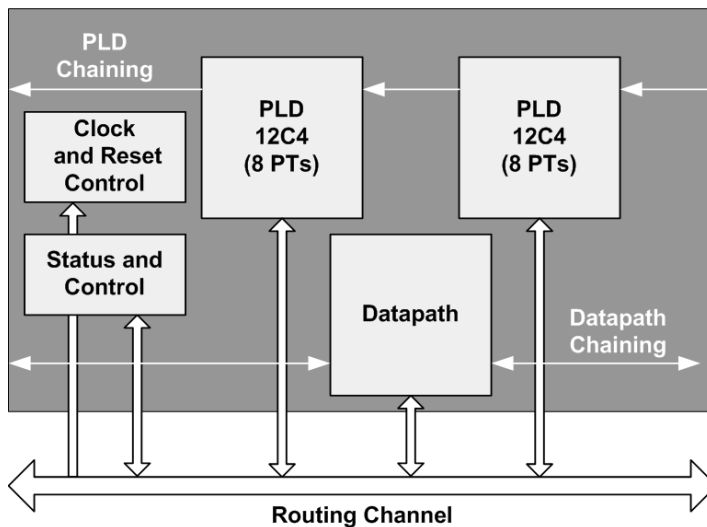
```

wire cmp = (count == 10'h3FB);
cy_buf cmpBuf (.x(cmp), .y(cmpOut));

```

11.6.4 资源选择

一个 Verilog 组件实现可以使用在 PSoC UDB 架构中有效的多种资源。根据所需功能选择资源。所有数字组件将使用 PLD 逻辑。一些组件还使用了数据路径实例。将所有合成的逻辑放在 PLD 中。必须将数据路径实例包含在 Verilog 设计内，以完成数据路径的实现。



11.6.4.1 数据路径

与 PLD 资源相比，数据路径资源的等效逻辑门能力多几倍，因此，如果数据路径资源适用于应用程序，则应该优先选择它。

典型数据路径应用

- 是指需要使用 FIFO 的操作。两数据路径的 FIFO 是可编程数字系统中的 FIFO 路径。对于某些设计，可以使用数据路径将 FIFO 功能添加到基于 PLD 的硬件设计内。
- 主要计数功能包括递增、递减或恒定增加。寄存器可用于进行计算、预加载、比较和捕获。
- 并行至串行和串行至并行的转换，其中并行连接指的是 CPU，并串行连接指的是硬件。

数据路径的限制

- 数据路径的并行输入受限。这样便限制了对硬件需要提供并行值的数据路径的使用。可以加载替换并行硬件的组件。如果有足够的实现周期，那么可以连续移入某个值。如果 CPU 或 DMA 可以访问该值，那么它们可以将该值写入到数据路径中的 FIFO 或寄存器中。
- 并行输出是可能的，但并行输出值始终是数据路径 ALU 的左侧输入。ALU 的左侧输入仅为 A0 或 A1，因此限制输出为 A0 或 A1，并限制可使用并行输出执行的 ALU 操作，以便可以将该值作为 ALU 功能的左侧输入。
- 每次只能执行一个 ALU 功能。如果需要执行多个操作，需要使用多个有效时钟来实现多个周期。
- 共有 8 个动态操作。一般这些操作已经足够了，但是对于一些较复杂的多周期计算（移位、加法和递增等），这些操作还不够。
- 数据路径仅能对 2 个寄存器进行读 / 写操作。虽然有 6 个寄存器源（A0、A1、D0、D1、F0 和 F1），但是只有两个寄存器是可读 / 写的（A0 和 A1）。

11.6.4.2 PLD 逻辑

PLD 逻辑资源是一个最灵活的数字硬件资源。

典型的 PLD 应用

- 状态机
- 小型计数器（ ≤ 4 位）或输入和输出并行的计数器。
- 通用的组合逻辑

PLD 的限制

- PLD 和 CPU 之间没有直接相连路径。您可以使用一个控制寄存器来获取 CPU 的数据。另外，还可以使用一个状态寄存器将数据发送给 CPU。
- 寄存器的最大位数等于“UDB 数量 * 8”（根据所选定的器件）。通过使用控制寄存器和状态寄存器可以增位的数量，但这些资源不会提供 PLD 可以写入和读取的寄存器。
- 每个 PLD 拥有 12 输入位，这样会限制大功能的效果和性能。例如，一个广复用功能并不能被良好地映射到 PLD 内。

A. 表达式评估工具



PSoC Creator 表达式评估工具用于对 PSoC Creator 中的各个表达式（调试器中的表达式除外）进行评估。评估内容包括以字符串格式表达的参数值和代码生成模板。表达式评估语言类似于 Perl 5 语言。它借用了 Perl 5 语言的大部分运算符，包括其优先级，但添加了一种不同类型的系统，该系统更符合 PSoC Creator 应用的要求。

A.1 评估上下文

有两个基本的评估上下文：文档上下文和实例上下文。文档上下文包括文档的形式参数和局部参数，以及文档属性。实例上下文则包括实例的形式参数和局部参数，以及参考 SYMBOL（符号）的文档属性。

[第 22 页上的正式参数与局部参数](#) 详细介绍了局部参数和形式参数的评估。注释可以是带嵌入式的表达式。大部分注释在文档上下文中被评估。与实例相关的注释会在实例上下文中进行评估。

A.2 数据类型

表达式评估工具包含以下几种基本类型：

- bool Boolean（真 / 假）
- error 错误类型
- float 浮点，双精度
- int8 带符号的 8 位整数
- uint8 无符号的 8 位整数
- int16 带符号的 16 位整数
- uint16 无符号的 16 位整数
- int32 带符号的 32 位整数
- uint32 无符号的 32 位整数
- string 字符串

A.2.1 Bool

有效值包括真和假。

A.2.2 Error

错误类型的数值可能是由系统自动创建或由最终用户创建的。这样提供了一个标准的方法，评估表达式可使用该方法来生成自定义错误。

A.2.3 Float

IEEE 标准的 64 位双精度浮点数。具体表示为 `[+-] [0-9] [.[0-9]*]? [eE [+-]? [0-9]+]`

有效值: 1、1.、1.0、-1e10、1.1e-10

无效值: .2、e5

A.2.4 Int

不同字长、带符号和不带符号。可以使用下面三种基本形式中的任意一种来表达整数:

- 十六进制 — 以 0x 开始
- 八进制 — 以 0 开始
- 十进制 — 是指所有其它数字序列

无符号整数常量通常后缀字符 “u”。

下表显示了各种类型的有效值。

类型	有效值范围
INT8	-128 ~ 127
UINT8	0 ~ 255
INT16	-32,768 ~ 32,767
UINT16	0 ~ 65535
INT32	-2,147,483,648 ~ 2,147,483,647
UINT32	0 ~ 4294967295

A.2.5 String

字符串序列，两边使用双引号（“”）。在内部以 “.NET” 字符串的形式储存这些字符串（使用 UTF-16 Unicode 编码）。目前，只有 ASCII 字符可被解析器识别。\\ 和 \” 是字符串中唯一受支持的转义序列。

A.3 数据类型转换

由于 Perl 5 语言的特性，所有数据类型都能转换为其它类型（除一个例外）。数据类型的转换规则非常清晰，并且可预测。

只有错误类型是一个例外。所有数据类型都可能会转换为错误类型，但错误类型却不能转换为其它类型。

A.3.1 Bool

目标类型	规则
错误	变为一条通用的错误消息。
浮点	如果 Bool 值为真，那么转为浮点数时，该值为 1.0。如果 Bool 值为假，那么转为浮点数时，该值为 0.0。
整数	如果 Bool 值为真，那么转为整数时，该值为 1。如果 Bool 值为假，那么转为整数时，该值为 0。
字符串	如果 Bool 值为真，那么转为字符串时，该值为真。如果 Bool 值为假，那么转为字符串时，该值为假。

A.3.2 错误

目标类型	规则
Bool	非法
浮点	非法
整数	非法
字符串	非法

A.3.3 浮点

目标类型	规则
Bool	如果浮点数值为 0.0，那么在转换为 Bool 时，该值为假。否则，该值为真。
错误	变为一条通用的错误消息。
整数	十进制部分被丢掉，结果中的整数部分被使用。如果转换结果值大于 32 位整数，那么转换结果将变为 0。
字符串	将浮点数转换为一个字符串。自动确定字符串的正确格式。

A.3.4 整数

目标类型	规则
Bool	如果整数值为 0，那么转为 Bool 时，该值为假。否则，该值为真。
错误	变为一条通用的错误消息。
浮点	变为最接近浮点的等效值。通常，它表示添加一个 “.0”。
字符串	将字符串转换为它的十进制整数

A.3.5 字符串

字符串是一种特殊情况。字符串一共有以下四种子类型：

- **bool-ish** 字符串的值为“真”或“假”。
- **float-ish** 字符串具有一个浮点值；这些字符串的第一个字符都不是空白的。不属于浮点的尾随字符将被忽略。
- **int-ish** 字符串具有一个整数值；这些字符串的第一个字符都不是空白的。不属于浮点的尾随字符将被忽略。
- 其他子类型的字符串

A.3.5.1 *bool-ish* 字符串

目标类型	规则
Bool	如果 Bool-ish 字符串的值为真，那么转为 Bool 型时，该值为真；如果 Bool-ish 字符串的值为假，那么转为 Bool 型时，该值为假。
错误	变成一段字符串文本的错误信息。
浮点	如果 Bool-ish 字符串的值为真，那么转为浮点数时，该值为 1.0。如果 Bool-ish 字符串的值为假，那么转为浮点数时，该值为 0.0。
整数	如果 Bool-ish 字符串的值为真，那么转为整数时，该值为 1；如果该值为假，那么转为整数时，该值为 0。

A.3.5.2 *Float-ish* 字符串

目标类型	规则
Bool	非浮点的文本被去掉，浮点部分则作为一个真正的浮点进行转换。
错误	变成一段字符串文本的错误信息。
浮点	非浮点的文本被去掉，浮点文本则被转换为真正的浮点。
整数	非浮点的文本被去掉，浮点部分则作为一个真正的浮点进行转换。

A.3.5.3 *Int-ish* 字符串

目标类型	转换规则
Bool	非浮点的文本被去掉，浮点部分则作为一个真正的整数进行转换。
错误	变成一段字符串文本的错误信息。
浮点	非浮点的文本被丢掉，浮点文本则被转换成一个真正的整数。
整数	非浮点的文本被去掉，浮点部分则作为一个真正的整数进行转换。

A.3.5.4 其他子类型的字符串

目标类型	转换规则
Bool	如果其他字符串的值是空字符串或 '0'，那么转为 Bool 时，该值为假。所有其他字符串的值转换为真。
错误	变成一段字符串文本的错误信息。
浮点	转换为 0.0。
整数	转换为 0。

A.4 运算符

表达式评估工具支持下面各运算符（按优先级从高到低的顺序排列）。

- casts
- ! unary+ unary-
- * / %
- + - .
- < > <= >= lt gt le ge
- == != eq ne
- &&
- ||
- ?:

所有运算符均强制将其参数转换为一个定义明确的类型，并且得到的所有结果的类型也应是明确的。

A.4.1 算术运算符（+、-、*、/、%、unary +（一元加）、unary -（一元减））

算术运算符遵循下列规则强制将其参数转换为某个数值：

- 如果操作数是一个错误类型的值，那么运算的结果是最左边的错误值。
- 如果操作数是一个浮点类型或 **float-ish** 字符串，那么各个值均强制被转换为适当的浮点数。
- 否则，操作数将被强制转换为整数。

如果各操作数均为整数，并且至少其中有一个为无符号的整数类型，那么运算将作为无符号运算执行。

使用上述定义的规则将操作数进行转换后，算术运算符始终会生成与操作数类型相同的数值或错误值。

A.4.2 数值比较运算符（==、!=、<、>、<=、>=）

通过使用与算术运算符完全相同的规则，数值比较运算符会强制将其参数转换为某个数值。数值比较运算符始终会生成一个 **Bool** 值。

如果各操作数均为整数，并且至少其中有一个为无符号的整数类型，那么运算将作为无符号运算执行。

A.4.3 字符串比较运算符（eq、ne、lt、gt、le、ge）

除非操作数为错误类型，否则字符串比较运算符会强制将其参数转换为字符串。如果操作数为错误类型，则运算的结果是最左边的错误值。字符串比较运算符始终会生成一个错误值或字符串。

A.4.4 字符串串联运算符（.）

字符串串联运算符同字符串比较运算符一样，都是为了转换其参数。

注意：如果串联运算符前面或后面紧挨着一个数字，那么它被视为一个浮点类型（例如，`1. =` 浮点；`1. =` 字符串串联）。

A.4.5 三元运算符（?:）

- 如果第一个操作数应该是 **Bool** 类型的值，但它却是一个错误的类型，那么其结果会是错误类型的值。
- 如果该 **Bool** 类型的值为真，那么其结果值和类型将表达为 “?” 或 “:”。
- 如果该 **Bool** 类型的值为假，那么其结果值和类型后面将是 “:”。

A.4.6 Cast 运算符

Cast（小写为 **cast**）的转换格式为 **cast**（类型，表达式）

Cast 运算符评估了表达式，并将 **expr** 的结果转换为已命名的类型。

A.5 字符串插值

表达式工具能够计算和评估字符串中所嵌入的表达式。其格式为：

`` = expr ``

``` 表示表达式开始。不会将 `=` 视为表达式的一部分。因为下一个 ``` 表示表达式的结束，所以不能将 ``` 嵌套在其他 ``` 中。只要多个表达式没有嵌套，那么它们都可以被嵌入到同一个字符串中，会对这些表达式进行评估和计算。只对该嵌入式的表达式进行一次评估。如果结果字符串是一个合法的表达式，那么不会将它评估为额外的步骤。

从开始处的 ``` 到结束处的 ``` 之间所有的字符串被 ``` 和 ``` 间的结果表达式所代替。

## A.6 用户自定义的数据类型（枚举）

对于表达式系统，用户自定义的数据类型是整数的。用户定义类型的数值可以在表达式中显示出，但在表达式被评估前，这些值将被转换为等效的整数。请参考第 27 页上的[添加用户定义类型](#)，了解如何将用户定义的数据类型添加到您的符号内。

## B. 数据路径配置工具



数据路径配置工具用于修改 PSoC 组件的 Verilog 实现中数据路径实例配置。该附录提供了有助于使用工具进行修改 Verilog 文件的指导和信息。

有关使用 Verilog 实现组件的信息，请参考第 61 页上的使用 Verilog 实现硬件。

### B.1 通用功能

数据路径配置工具用于：

- 读取现有的 Verilog 文件
- 修改现有的数据路径配置
- 创建新配置
- 删除现有的配置

您可以将这些更改另存为一个 Verilog 文件，而不会影响其他所有 Verilog 的执行。该工具仅适用于现有的 Verilog 文件；不能使用它创建新的 Verilog 执行。

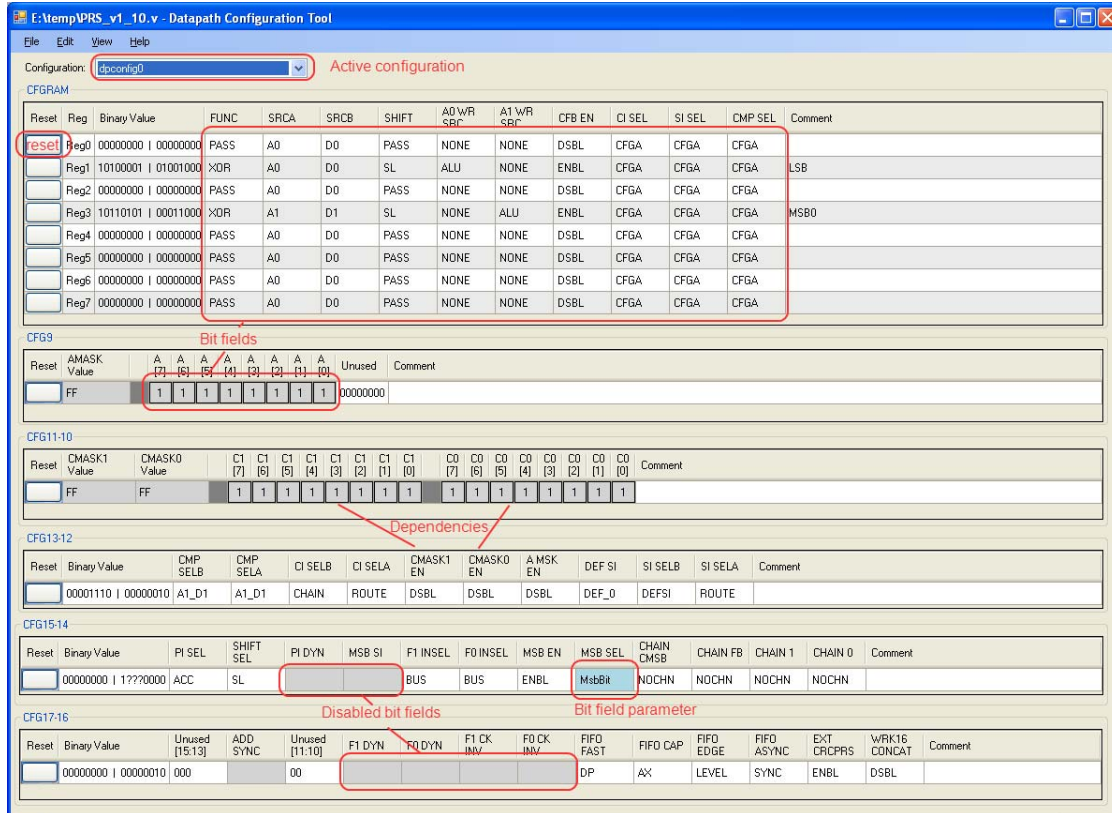
打开 Verilog 文件后，它将被解析为现有的数据路径配置。这些配置可以被定义为参数、局部参数或在数据路径中直接被定义为已命名的参数。工具会在应用顶部的 **Configuration**（配置）下拉列表中显示所有检测到的配置。

您可以修改任意位字段和注释内容，在各种配置内或在一个单独的配置中复制和粘贴数据，并执行整个数据路径的复制、粘贴、创建或删除操作。

某些操作仅适用于数据路径（例如，删除一个数据路径或修改寄存器的初始值）。有效的数据路径是指包含了所选配置的数据路径。如果配置被定义为一个参数，那么不存在任何有效数据路径，并且所有数据路径选项均被禁用。

## B.2 框架

### B.2.1 界面



它主要包括以下各组成部分：

- **配置下拉列表** — 它包含了在有效的 Verilog 文件中检测到的所有配置。当前选择的配置被称为“有效配置”或“所选配置”。
- **配置寄存器的数据网** — 它包括了复位寄存器按键（用于将整个寄存器恢复为其默认配置）、寄存器二进制值字段、位字段配置控制以及注释字段。二进制数值字段是只读的。位字段控制通常是带有预定义值的下拉列表。CFG9、CFG10、CFG11 寄存器都是掩码寄存器，通过点击单独位按键可以切换这些寄存器的值。
- **位字段参数** — 每个位字段均包含一个参数值，而不是一个常量。当使用一个参数来定义位字段时，该参数名称会作为字段值，并以蓝色背景显示。
- **未使用的位字段** — 它们是只读字段，并且其数值始终为 0。
- **被禁用的位字段** — 指的是未使用于有效配置的位。用户可以使用这些位的上下文菜单来使能它们。在特定版本的芯片中，可能要禁用某些位。被禁用的位字段以灰色背景显示，并且不存在任何值。它们的初始值始终为 0。

## B.2.2 菜单

### B.2.2.1 File（文件）菜单

菜单项	快捷方式	说明
Open（打开）	[Ctrl] + [o]	将显示一个对话框，通过该对话框可以打开现有的文件（已经过滤为 .v Verilog 文件）
Close（关闭）		关闭有效文件
Recent Files （最近的文件）		访问之前打开的文件
Save（保存）	[Ctrl] + [s]	保存有效文档
Save As（另存为）		将显示一个对话框，用于将有效文件保存为另一个文件
Exit（退出）		退出 PSoC 数据通路配置工具

### B.2.2.2 Edit（编辑）菜单

菜单项	快捷方式	说明
Copy Datapath （复制数据路径配置）	[Ctrl] + [c]	复制所选的数据路径配置
Paste Datapath （粘贴数据路径配置）	[Ctrl] + [v]	将完整的数据路径配置从剪贴板粘贴到有效的数据路径内
Paste Dynamic （粘贴动态配置）		将完整的数据路径配置的动态配置信息从剪贴板粘贴到有效的配置内（全部 8 个寄存器都要粘贴动态配置）
Reset Datapath （复位数据路径配置）		将当前的数据路径配置复位为默认的字字段值
New Datapath （新的数据路径实例）		添加一个新的数据路径实例
Delete Datapath （删除数据路径实例）		删除数据路径实例

### B.2.2.3 View（视图）菜单

菜单项	快捷方式	说明
Initial Register Values （初始寄存器值）		显示一个对话框，通过它为有效数据路径配置初始寄存器值

### B.2.2.4 Tools（工具）菜单

菜单项	快捷方式	说明
Options > Splash screen on Startup （各选项 > 启动时的启动屏幕）		启动数据路径配置工具时，选择用于使能或禁用启动屏幕的选框

### B.2.2.5 Help（帮助）菜单

菜单项	快捷方式	说明
Documentation（文档）	[F1]	打开该文档。
About（编译信息）		打开 About 对话框，获取有关编译组件的信息

## B.3 常见任务

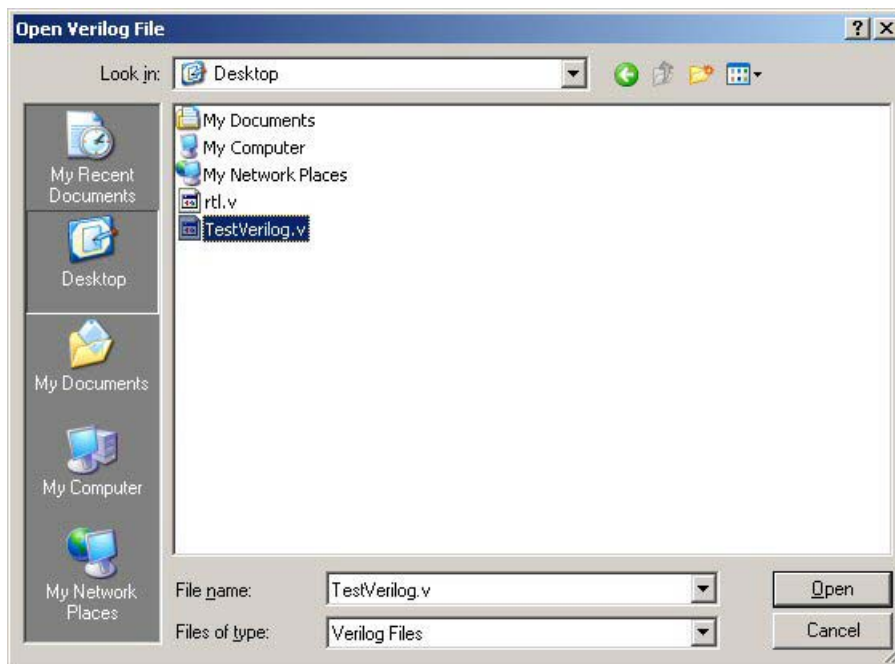
### B.3.1 启动数据路径配置工具

要想启动该工具，请从 **Start**（启动）菜单中依次选择 **Cypress > PSoC Creator 1.0 > Component Development Kit > Datapath Configuration Tool**。

启动该应用时，它不包含任何数据路径配置的信息。要使用该工具开始进行操作，您必须打开现有的 Verilog 文件。

### B.3.2 打开 Verilog 文件

1. 要想打开 Verilog 文件，请从 **File** 菜单选择 **Open...**（打开）项。  
这时会显示 **Open Verilog**（打开 Verilog 文件）对话框。



**注意：**您还可以使用 **File** 菜单上的 **Recent Files**（最近打开的文件）或将 Verilog 文件直接拖放到应用的主要部分，以此打开最近文件，无需使用该对话框。

2. 导航并选择需要更新的文件，然后点击 **Open**。  
该对话框被关闭，然后工具会从 Verilog 文件中收集所有相关信息并更新窗口的剩余部分。

### B.3.3 保存文件

要想保存有效文件，请从 **File** 菜单下选择 **Save**（保存）项。

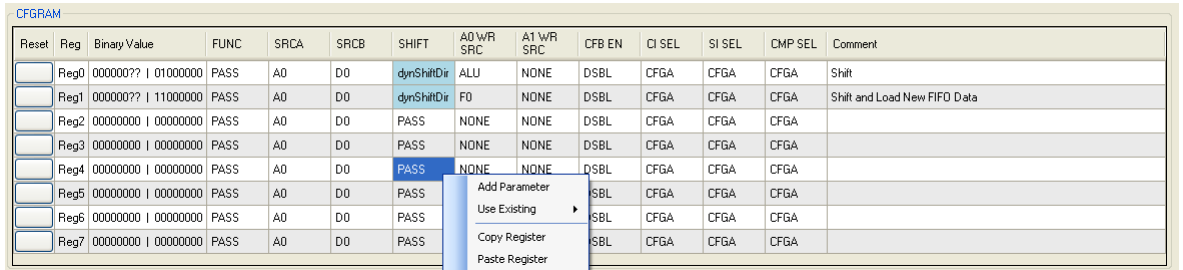
要想将文件另存为其它名称和 / 或保存在其它位置，请从 **File** 菜单中选择 **Save As**（另存为）项。

## B.4 使用位字段参数

除了一个固定的预定义常量字段集外，位字段还支持一个参数值。

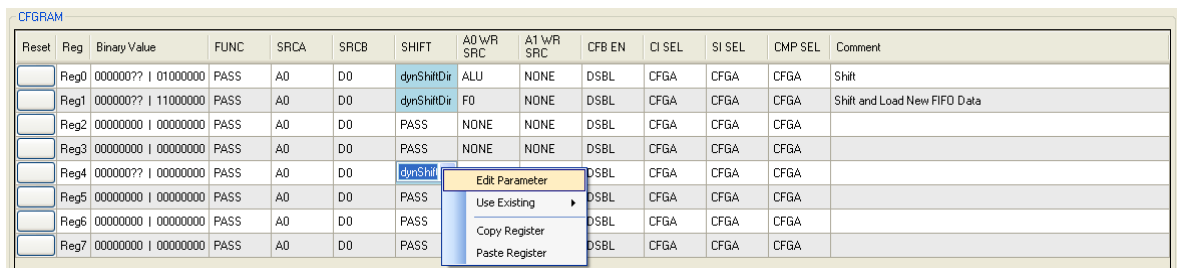
### B.4.1 将参数添加到枚举位字段内

枚举位字段包含 **Add Parameter**（添加参数）上下文菜单项。



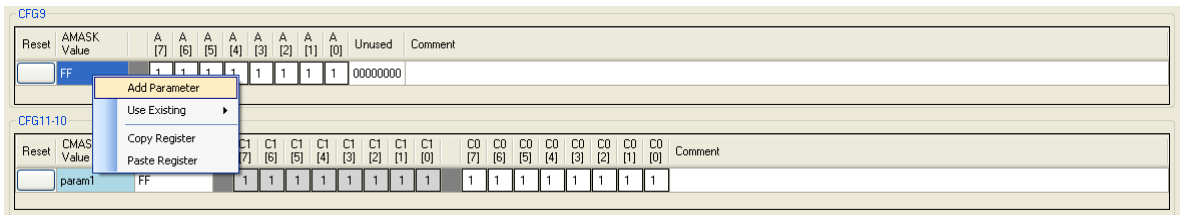
使用该指令，用户可以为位字段提供任意参数名称作为一个附加选择。添加参数后，位字段的下拉菜单会包括所有预定义常量以及所有新添加的参数值选项。如果选择了参数，它将以蓝色显示，表示它不是预定义值。

要更改参数名称，请右键点击位字段，然后从上下文菜单内选择 **Edit Parameter**（编辑参数）。



### B.4.2 将参数添加到掩码位字段内

要想将一个参数添加到掩码位字段内，请从 **Value** 列中的上下文菜单中选择 **Add Parameter**（添加参数）项，然后输入参数名称。添加参数后，单独位会变成只读的。



要想移除某个参数，请从 **Value** 列中的上下文菜单中选择 **Remove Parameter**（移除参数）项。

### B.4.3 位字段的依赖属性

数据路径配置中包含了一些对其他位产生影响的位。根据它们的值，其他位字段可能变成未使用字段。要想使这些字段对用户可见，根据其它位字段的值，某些位字段会被禁用。

CFG11-10																			
Reset	CMASK1 Value	CMASK0 Value	C1 [7]	C1 [6]	C1 [5]	C1 [4]	C1 [3]	C1 [2]	C1 [1]	C1 [0]	C0 [7]	C0 [6]	C0 [5]	C0 [4]	C0 [3]	C0 [2]	C0 [1]	C0 [0]	Comment
	FF	FF	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

CFG13-12												
Reset	Binary Value	CMP SELB	CMP SELA	CI SELB	CI SELA	CMASK1 EN	CMASK0 EN	A MSK EN	DEF SI	SI SELB	SI SELA	Comment
	00000000   10100011	A1_D1	A1_D1	ARITH	ARITH	ENBL	DSBL	ENBL	DEF_0	DEFSI	CHAIN	

CFG15-14														
Reset	Binary Value	PI SEL	SHIFT SEL	PI DYN	MSB SI	F1 INSEL	F0 INSEL	MSB EN	MSB SEL	CHAIN CMSB	CHAIN FB	CHAIN 1	CHAIN 0	Comment
	07000100   17770000	ACC	ShiftDir	DS	REG	A0	BUS	ENBL	dpMSb...	NOCHN	NOCHN	NOCHN	NOCHN	

在上面的截图中，CMASK0、SI SELB 和 SI SELA 位均被禁用。其原因是 CMASK0 EN 位被设为 DSBL，MSB SI 位被设为 REG。

## B.5 使用各种配置

有两种方法可用于配置数据路径实例的元素。

第一种方法是将数据路径的配置直接放入到数据路径实例中。对于多字节的数据路径实例，将一个单独的数值集使用于整体实例的每一个字节。当在 Verilog 文件中只使用一次特殊配置时，经常使用这个方法。

第二种方法是将一个数据路径的配置分配给某个参数。然后，该参数被用于数据路径实例。当多个实例使用了同一个配置时，该方法显得非常方便，它可以减少重复并能够简化组件的维护。该方法由计数器库组件使用。这两种方法可以使用于同一个 Verilog 文件中。

它们受 PSoC 数据路径配置工具的支持。所有配置均被添加到 **Configuration** 下拉列表中。

### B.5.1 配置命名

在数据路径实例中，标签就是数据路径实例的名称。在 dp8、dp16、dp24 和 dp32 数据路径实例中，根据配置实例的字节，配置标签的后缀可以是“\_a”、“\_b”、“\_c”或“\_d”。另外，在下拉列表中配置名称的尾部添加了数据路径大小（8、16、24 或 32）。在参数实例中，配置标签就是参数名称。

Configuration:	
	dpMISO_a (8)
CFGGRAM	dpMISO_a (8)
	dpMOSI_a (8)
	dpMISO_a (16)
	dpMISO_b (16)
	dpMOSI_a (16)
	dpMOSI_b (16)

Reset	Reg
	Reg0

## B.5.2 编辑配置

要想使配置生效，请在 **Configuration** 下拉列表中选择需要编辑的配置。通过选择单元下拉列表中其中一个预定义值或添加一个参数，可修改寄存器的位字段。对有效配置进行的所有更改均被自动保存在应用中的目标储存器内，因此用户能够在不同配置间进行切换，而无需担心丢失数据。直到完成主要菜单中的“保存”或“另存为”操作为止，数据才会被保存在 Verilog 文件中。

## B.5.3 复制、粘贴配置

要想将有效的配置数据复制到剪贴板上，请选择 **Edit** 菜单中的 **Copy**（复制）指令。

要想粘贴剪贴板上的配置数据，请选择 **Edit** 菜单中的 **Paste**（粘贴）指令。该指令会使用剪贴板上的值替代有效配置中的所有位字段和注释。

如果仅需要粘贴动态寄存器数据，那么请在 **Edit** 菜单中选择 **Paste Dynamic**（粘贴动态）指令。

**注意：**如果您只需要粘贴动态寄存器数据，您也应该使用 **Copy** 指令，实现复制。

要想复制一个单独的寄存器，请从该寄存器的任意位字段的上下文菜单中选择 **Copy Register**（复制寄存器）指令。

要想粘贴一个单独的寄存器，请从该寄存器的任意位字段的上下文菜单中选择 **Paste Register**（粘贴寄存器）指令。

## B.5.4 复位配置

要想将当前的数据路径配置复位为默认位的字段值，请在 **Edit** 菜单上选择 **Reset Datapath**（复位数据路径）指令。

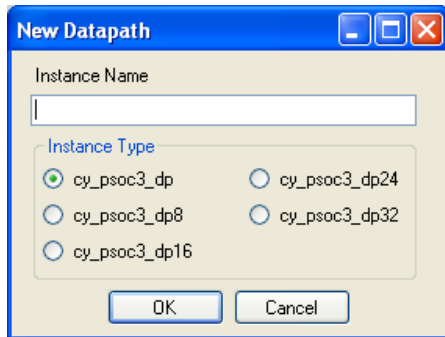
要想将数据路径配置的特殊寄存器复位为它的默认值，那么请在主表格的那一行中按下数据网中 **Reset**（复位）列的按键。

除 CFG9-11 的掩码字段外，所有位字段的默认值均为 **0**。CFG9-11 掩码字段的默认值为 **1**。动态寄存器的默认注释值为 **Idle**（空闲），其它寄存器的默认值都是一个空字符串。

## B.6 使用数据路径实例

### B.6.1 创建新的数据路径实例

要想创建一个新的数据路径，请从 **Edit** 菜单中选择 **New Datapath**（新数据路径）指令。这时候会出现 **New Datapath** 对话框。



将新数据路径的名称键入到 **Instance Name**（实例名称）文本框中，选择数据路径类型，然后点击 **OK**。

新的数据路径将插入到当前 Verilog 文件尾部 **endmodule** 的前面。

使用所有位字段默认值对新数据路径配置进行初始化，所有输入信号将被连接至 1'b0，而所有输出信号则处于未连接状态。

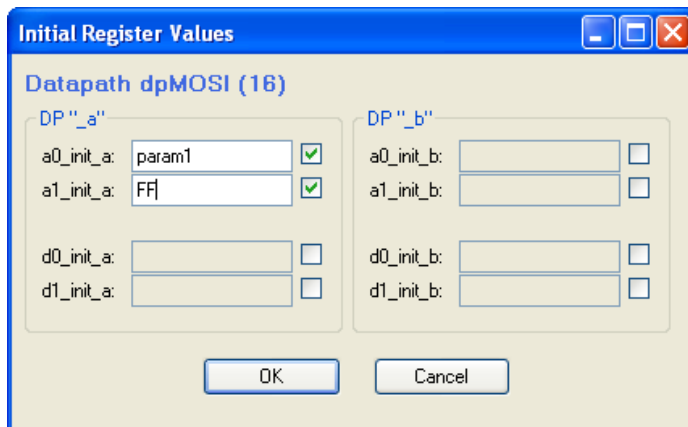
### B.6.2 删除数据路径实例

要想删除有效的数据路径，请从 **Edit** 菜单下选择 **Delete Datapath**（删除数据路径）指令。

如果没有任何有效的数据路径，那么该选项将在菜单中被禁用。如果数据路径存在多个配置（dp16、dp24 或 dp32），那么将会删除所有配置。

### B.6.3 设置初始寄存器值

要想设置有效数据路径的初始寄存器值，请在 **View**（视图）菜单上选择 **Initial Register Values**（初始寄存器值）指令。这时将出现 **Initial Register Values** 对话框。



检查将被定义的条目附近的各个框，然后将一个 8 位的十六进制值数值（8'h?）或一个参数值输入到编辑字段中。然后点击 **OK**，以提交这些更改。初始值将作为已命名的参数传递到某个数据路径中，因此所有未定义的值都无法被传递到数据路径内，并会在内部设置为其默认值。

如果没有任何有效的数据路径，那么该选项会在菜单中被禁用。