PSoC® Creator™

Component Author Guide

Document # 001-42697 Rev. *T

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
http://www.cypress.com

Contents

1. Introduction

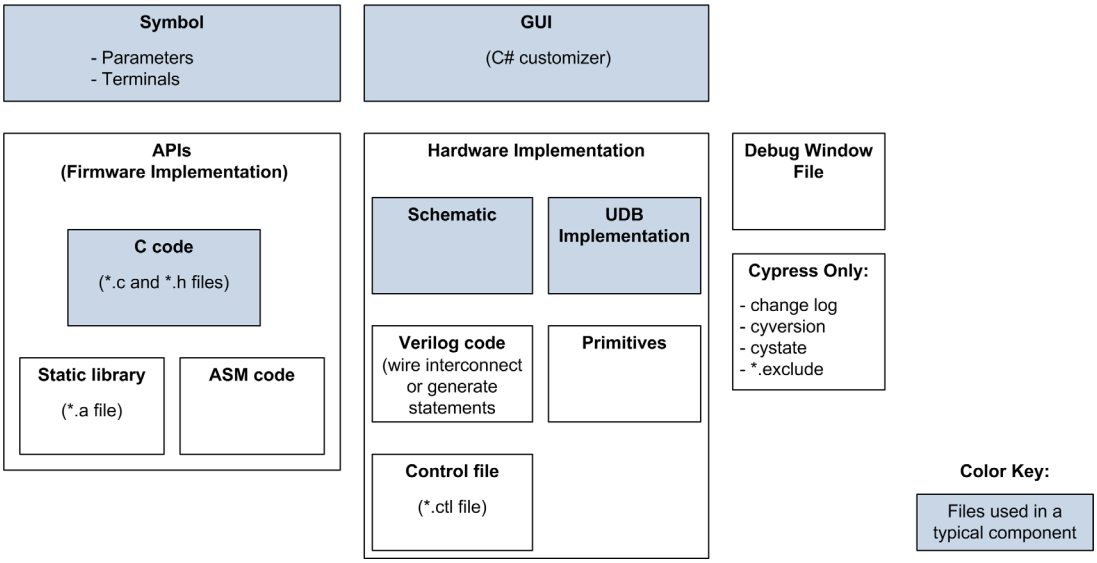
This guide provides instructions and information that will help you create Components for PSoC Creator. Some of this guide is intended for advanced users to create sophisticated Components that other users employ to interact with PSoC Creator. However, there are some basic principles in this guide that will also benefit novice users who may wish to create their own Components.

This chapter includes:

- What is a PSoC Creator Component?
- Component Interaction
- Component Creation Process Overview
- Cypress Component Requirements
- Component Parameter Overview
- References
- Conventions Used in the Guide
- Revision History

1.1 What is a PSoC Creator Component?

A PSoC Creator Component is a collection of files, such as a symbol, schematic, APIs, and documentation that defines functionality within the PSoC Device. Examples of Components include a timer, counter, and a mux. The following shows the various elements of a Component.

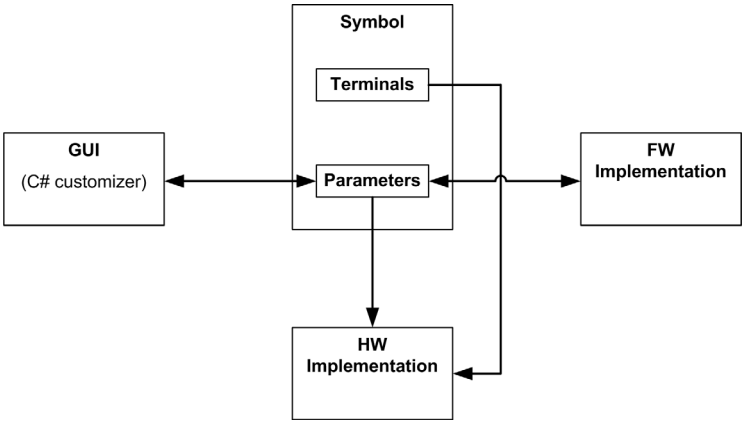


The most common files that make up a Component include the following:

- **Symbol –** A symbol contains the basic definition of a Component. It contains the top-level picture shown in the PSoC Creator Component Catalog, as well as the parameter definitions. There can be only one symbol in a Component.
- **Optional hardware implementation –** This is typically either a schematic or a Verilog file or a UDB editor file. Some Components have a *<project>.cyprimitive* file -- this is just a sentinel that says the backend tools inherently know about it and there is no explicit implementation.
  - **Schematic –** A schematic defines how a Component has been implemented visually. A schematic can be generic for any PSoC device, or it can be specific to a Family, Series and/or Device.
  - **Verilog –** Verilog can be used to define the functionality of a Component implemented in Verilog. There will only be one Verilog file in any given level of a Component. Verilog files found at different levels of the Component, such as at a Family, Series and/or Device, may not refer to each other.
- **Optional firmware –** This is typically made of a C header, source files, or static libraries. Assembly is also supported, but rarely used. The C header and source files are templates. They can't be compiled directly by C compiler. They get processed by PSoC Creator and turned into the generated source you see when building a design.
- **API –** Application Programming Interface. APIs define how to interact with a Component using C code. They can be generic for any PSoC device, or they can be specific to a Family, Series and/or Device.
- **Optional C# customizer (.cs and .resx files) –** There is help documentation available from the Help menu that details the APIs the tool exposes to Components. It's effectively a plug-in interface that lets Components customize certain default behaviors of the tool (like appearance, how it netlists to Verilog, generating APIs on-the-fly, etc.).
- **Control File –** The control file contains directives to the code generation module. For information on how to add a control file, see Add Control File on page 82. For more information about control files in general, refer to the Control File and Directives topics in the PSoC Creator Help.
- **Documentation –** The documentation of the Component is generally its datasheet.

1.2 Component Interaction

The following shows how the different pieces of a Component interact with one another.



1.3 Component Creation Process Overview

The process to create Components includes the following high-level steps. See references to various chapters in this guide for more information.

- Create the library project (Chapter 2)
- Create a Component/symbol (Chapter 2)
- Define symbol information (Chapter 3)
- Create the implementation (Chapter 4)
- Simulate the hardware (Chapter 5)
- Create API files (Chapter 6)
- Customize the Component (Chapter 8)
- Add tuning support (advanced) (Chapter 9)
- Add bootloader support (as needed) (Chapter 10)
- Add/create documentation and other files/documents (Chapter 7)
- Build and test the Component (Chapter 7)

Note These chapters provide a logical grouping of related information and they present one, but not the only, possible workflow process for creating Components. Refer also to Chapter 11 for various best practices to follow when creating Components.

1.4 Cypress Component Requirements

All Component development should occur from inside PSoC Creator itself; however, you may need to use additional tools, such as the Datapath Configuration Tool (under the PSoC Creator **Tools** menu). All code and schematic editing, symbol creation, interface definition, documentation authoring, etc., should occur within a given Component.

- All Components produced by Cypress must have a change log. Add this to the Component as a separate Component item just like the datasheet. This log file should be a simple text file. The datasheet must contain a “Change” section for each new version of the Component.
- Make sure that the Component version is added to the Component name. Also, do not include the version information in the display name (catalog placement), as the tool will add this for you.

1.4.1 File Names

Component file names (including any version number) must be compatible with C, UNIX, and Verilog, which are all are case sensitive.

1.4.2 Name Considerations

When creating a Component, its name will be the same name used for all elements of that Component, except schematic macros, customizer source files and API source files; therefore, it is important to choose the name appropriately.

1.4.3 File Name Length Limitations

Component names should be no longer than 40 characters, and resource file names should be no longer than 20 characters. Longer file names create problems with the length of path names when the Component is expanded in end-user designs.

1.4.4 Component Versioning

Cypress follows the following Component version policy, which is documented more fully by an internal specification. The following are recommendations for external customers developing their own Components.

PSoC Creator supports Component versioning and patching. A version of the Component (major and minor) is fully self-contained. Patching may be used rarely and only when it does not change any actual content (perhaps just a document change).

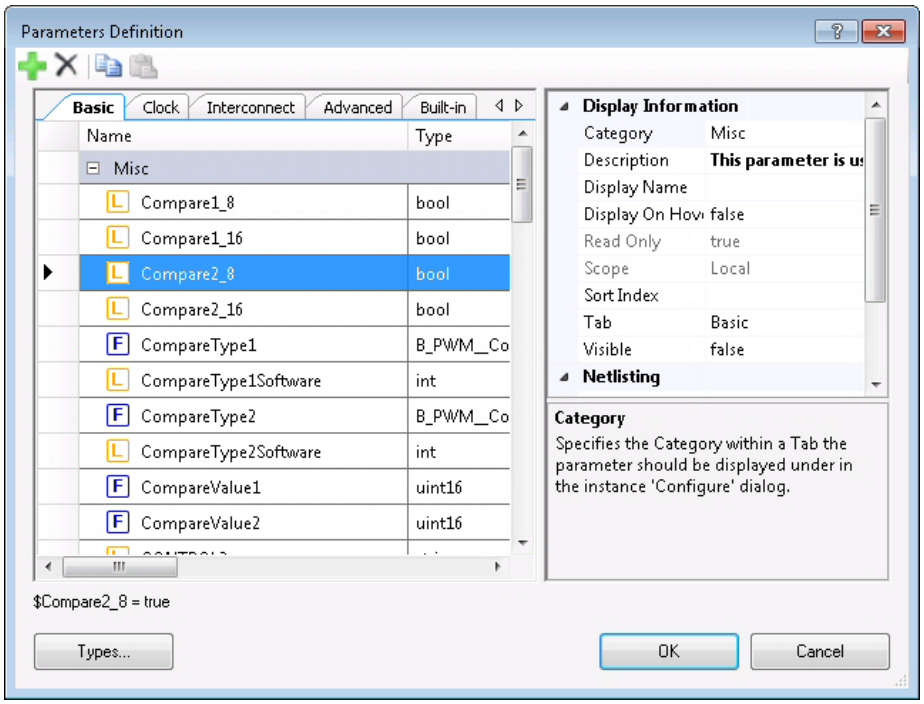
- **Major version:** major changes (e.g. API changes, Component symbol interface changes, functionality changes etc.) that may break the compatibility
- **Minor version:** No API changes. Bug fixes, additional features that do not break the compatibility
- **Patch:** No API changes. Documentation, bug fixes to the existing functionality of the Component that do not break the compatibility. Note that since patches are an aspect of the Component, when the user installs an updated Component, they automatically get these changes.

The version number will be carried as part of the Component name by appending "_v<major_num>_<minor_num>" to the Component name, where <major_num> and <minor_num> are integers that specify the major and minor versions respectively. For example, CyCounter8_v1_20 is version 1.20 of the CyCounter8 Component. Major and minor numbers are each integers on their own and do not make up a "real number." For example, v1_1 is not the same as v1_10 and v1_2 comes before v1_10.

The patch level will be an aspect of the Component (carried on the symbol) and hence is not reflected in the Component name. Incorporating the version number as part of the Component name will protect existing designs when there is a major change in the Component. Note that it is under the control of the Component author to name the major and minor versions.

1.5 Component Parameter Overview

Parameters in PSoC Creator are a set of one or more named expressions that define the behavior of a Component. You define parameters for a symbol using the Parameters Definition dialog.



A parameter consists of a **Name**, **Type**, **Value**, and an associated list of properties.

- The name can be almost whatever you want, as long as it is a legal identifier name. Also, the prefix "CY_" is reserved for built-in parameters (see Built-In Parameters on page 15). New parameter names cannot start with the string "CY_" or any case variation.
- The type is a pull-down menu listing all the data types available. For a description of the data types, refer to Data Types on page 165.
- The value is the default value for the instantiated Component. The default parameter value only defines the value used when a Component is first instantiated. The value is a snapshot at that time. If the default value for the Component is subsequently changed, that value does not get propagated down to any previously instantiated Component.
- The various properties for each parameter control how the parameter displays and behaves. See Define Symbol Parameters on page 31 for more information about these properties.

1.5.1 Formal versus Local Parameters

Symbols have two kinds of parameter definitions: formals and locals. Both can be displayed in the instance Configure dialog, but there are some key differences between them. Formals are normally visible and editable. They can be temporarily hidden and/or made read only. Locals are normally hidden, and are always read only. They can be made visible to show useful derived information (e.g., like a baud rate).

- Formal – These are how users configure a Component. They are inputs to the Component from the user. They serve the same purpose as arguments to a C function.
- Local – These are additional information required by the Component. They are calculated from formals. They serve the same purpose as stack variables in a C function.

In a symbol, formal parameters have constant default values and local parameters may have expressions with identifiers that refer to other parameters. The identifiers in the local parameter expressions come from the combined set of formal and local parameters. In effect, the parameter set is an evaluation context (for the local parameters).

Schematic documents have a parameter set. When open in the editor, the schematic has a parameter set that is a direct copy of its symbol parameter set. This copy is made at run time and is guaranteed to always stay in sync with the symbol. These parameters are not directly visible in PSoC Creator, except by editing the symbol. If a schematic does not have a symbol, then it does not have a parameter set.

An instance of a Component holds formal parameters from the Component's symbol. The instance creates a copy of its formal symbol parameters when it is dropped in a schematic. Users can change the values of the formal parameters to expressions. Formal parameters on an instance may refer to any parameters that exist on the schematic on which it was dropped. Formal parameters on an instance may not refer to any of its other parameters. Formal parameters are the means by which the user's configuration values get propagated through the design hierarchy by Component developers.

End users cannot change values of local parameters. The instance local parameters are evaluated in the context of the instance. That means they can refer to any other parameter on the instance, but may not refer to any parameters from the schematic on which it was dropped. See the Expression Evaluator appendix on page 165 for more information on evaluation contexts and their use.

Instance parameters match-by-name with symbol parameters; type does not matter. An in-memory instance has both an active parameter set and an orphan parameter. The active set are those parameters contained in the symbol's parameter set. The orphan set are those parameters that had been created for the instance, but the symbol changed, and those parameters are no longer valid parameters. For example, this allows end users to change their library search path, switch to an alternate implementation or version of a Component, and switch back to their original search path without losing any data.

Orphan parameters are transient. Orphan parameters will become active if the associated symbol is refreshed and now has formal parameters of those names. The elaborated schematic parameters are just a direct copy of the instance parameters from the parent instance.

1.5.2 Built-In Parameters

Built-In parameters are defined for each symbol by default. Built-in parameter definitions cannot be removed from the parameter definition editor. The names and the types of built-ins cannot be changed. Currently, PSoC Creator includes the following built-in parameters:

1.5.2.1 Formals:

Name	Description
INSTANCE_NAME	This is a special parameter. In the editor, it has the short name for the instance. This is what the user edits when they want to rename an instance. In the elaborated design, it provides access to the "full" or "hierarchical" instance name. This is the name that should be used (via `$INSTANCE_NAME`) in API files to ensure that there aren't any collisions). In other words, this name is nearly always the one Component authors should refer to, and it will have an appropriate value. **Note** The INSTANCE_NAME parameter returns a different value when used in the expression evaluator versus API and HDL generation. For the expression evaluator, it returns the leaf name of the instance; when used for API and HDL generation, it returns the path name of the instance.
CY_CONST_CONFIG (Config Data in Flash)	Controls whether the configuration structure is stored in flash (const, true) or SRAM (not const, false). Only visible for devices that support driver style APIs (for example, FMx).
CY_REMOVE (Disable)	This parameter is a flag to the elaborator, which tells it to remove the instance from the netlist. It works in conjunction with the live mux to allow Component authors to dynamically switch between two different implementations of a Component in a single schematic. The default value is false (that is, keep the instance around).
CY_SUPPRESS_API_GEN (Suppress API Generation)	This parameter can be used to indicate that a particular instance should not have an API generated for it (even though the Component has an API). The default value is false.

CY_COMMENT
(User Comments)
	Instance-specific comments. Component authors should include the comment in generated source (.c file) as follows:
	`/*`
	`` `$CY_COMMENT` ``
	`*/`

1.5.2.2 *Locals:*

Name	Description
CY_API_CALLBACK_HEADER_INCLUDE	This parameter is used during the build process to check for the existence of the *cyapicallbacks.h* file on disk in the project directory. □ If the file exists, this parameter will expand to `#include "cyapicallbacks.h"`. □ If the file does not exist, this parameter expands to an empty string.
CY_COMPONENT_NAME	This parameter contains the file system path to the Component's base directory (the directory where the symbol exists). It is only valid after elaboration. Before elaboration this parameter contains the value "__UNELABORATED__".
CY_CONTROL_FILE	This parameter is used to specify a control file for the Component instance. This is used internally to define fixed placement characteristics for specific Components. The default value (<:default:>) refers to the control file at the generic level (that is, <ComponentName>.ctl), if one is used. This value should not be changed. For information about how to create and use a control file in a design, refer to the “Control File” topic in the PSoC Creator Help. See also Add Control File on page 80.
CY_DATASHEET_FILE	This parameter is used to specify all documentation files for the Component instance. This includes the datasheet and all supporting documents for a Component. The default value (<:default:>) refers to the datasheet file (that is, <ComponentName>.pdf). If there are multiple files associated with the Component, a list of documents can also be provided using a semicolon separated string of filenames. For example: <ComponentName>.pdf;document1.pdf;document2.html **Note** The first document in the string ALWAYS must be the datasheet. The file system paths for all documents are relative to the Component’s base directory.
CY_FITTER_NAME	Hierarchical name in fitter format.
CY__INSTANCE_SHORT_NAME	If for some reason a Component author needs the short form of the instance name even in the elaborated design, it can be accessed via this parameter. End- users should NEVER see this. It isn't editable. Its value is automatically updated whenever the user modifies the INSTANCE_NAME parameter.
CY_PDL_DRIVER_NAME	Name of the PDL driver that this Component generates code (e.g., initialization structures) for.
CY_PDL_DRIVER_REQ_VERSION	Required version of the driver this Component is compatible with in the form Major.Minor.Patch. Major version number must be an exact match for the PDL driver. Minor and Patch are minimum versions within a major version.
CY_PDL_DRIVER_SUBGROUP	Optional subgroup of the driver name. Corresponds to Pack cSub.

Name	Description
CY_PDL_DRIVER_VARIANT	Optional special variant of the driver name. Corresponds to Pack cVariant
CY_VERSION	This parameter displays version and build information for PSoC Creator.
CY_MAJOR_VERSION (Component Major Version)	This parameter contains the major version number for the instance.
CY_MINOR_VERSION (Component Minor Version)	This parameter contains the minor version number for the instance.

1.5.3 Expression Functions

Anyone can access the following functions by using them in an expression.

You can make an expression that contains a function call. In order to use one of the functions in a text label, use the format `` `=IsError()` `` for example. If you want to set a parameter value to the result of one of these functions then just set the value to the function. Similarly, these functions can be used in a validator expression. See also Add Parameter Validators on page 35.

1.5.3.1 Device Information Functions

These functions may not have data in all situations. For example, when editing a schematic inside a Component in a library project there is no selected device available.

Function Name	Description
GetDeviceFamilyName	`GetDeviceFamilyName() : string` Get the family name of the selected device, such as PSoC 3, PSoC 4, PSoC 5, FM0p.
GetDeviceSeriesName	`GetDeviceSeriesName() : string` Gets the name of the series of the selected device. E.g., CY8C588, PSOC3/ PSoC 4/FM.
GetDeviceInternalName	`GetDeviceInternalName() : string` Gets the internal name of the selected device (e.g., PSoC3A, PSoC5LP, etc.).
GetDevicePartNumber	`GetDevicePartNumber() : string` Gets the part number of the selected device.
GetDevicePackageName	`GetDevicePackageName() : string` Gets the package name of the selected device.
GetFeatureCount	`GetFeatureCount(nameOfHardwareFeature) : int` Get the number of instances of a named hardware feature. Available hardware features names includes: PSoC 3/PSoC 5 devices: P3_ANAIF, P3_CAPSENSE, P3_CAN, P3_COMP, P3_DECIMATOR, P3_DFB, P3_DMA, P3_EMIF, P3_DSM, P3_I2C, P3_LCD, P3_LPF, P3_OPAMP, P3_PM, P3_SCCT, P3_TIMER, P3_USB, P3_VIDAC, P3_VREF. PSoC 4 devices: m0s8bless, m0s8can, m0s8peri, m0s8cpuss, m0s8cpussv2, m0s8cpussv3, m0s8csd, m0s8csdv2, m0s8ioss, m0s8iossv2, m0s8lcd, m0s8lpcomp, m0s8scb, m0s8srss, m0s8srssv2, s8srsslt, s8srsslta, m0s8tcpwm, s8pass4al, m0s8pass4a, m0s8pass4b, m0s8sar, m0s8smif, m0s8ssc, m0s8tss, m0s8udbif, m0s8udb, m0s8usbdss, m0s8usbpd, m0s8wco, m0s8crypto. Example: GetFeatureCount(“P3_DFB”) – returns the number of digital filter blocks.
GetFeatureCountDie	`GetFeatureCountDie(nameOfHardwareFeature) : int` Get the number of instances of a hardware feature, ignoring wounded features. Available hardware features is identical to GetFeatureCount().
GetFeatureParameter	`GetFeatureParameter(nameOfHardwareFeature, nameOfParameter) : int` Gets the integer value of a parameter of a named hardware feature. Available hardware feature names is identical to GetFeatureParameter(). For a list of available parameters, please contact Cypress.

1.5.3.1 Device Information Functions (continued)

These functions may not have data in all situations. For example, when editing a schematic inside a Component in a library project there is no selected device available.

Function Name	Description
GetFeatureParameterDie	`GetFeatureParameter(nameOfHardwareFeature, nameOfParameter) : int` Gets the integer value of a parameter of a named hardware feature, ignoring wounded features. Available hardware feature names and parameter names are identical to GetFeatureParameter().
GetFeatureVersion	`GetFeatureVersion(nameOfHardwareFeature) : int` Get the version of a named hardware feature. Available hardware feature names is identical to GetFeatureParameter().
GetJtagId	`GetJtagId() : int` Gets the JTAG id of the selected device.
GetResourceCount	`GetResourceCount(namedResource) : int` Get the number of instances of the named resource. Available resource names include: CSIDAC7, CSIDAC8, P4CSDCOMP, CAPSENSECSD, CAPSENSEGESTURE, and CAPSENSEADC.
GetSiliconRevision	`GetSiliconRevision() : int` Gets the revision of the selected device.
UsesDriverStyleApis	`UsesDriverStyleApis() : bool` Returns true if the selected device uses driver-style APIs.

1.5.3.2 Component Information Functions

These functions are used for Components.

Function Name	Description
GetComponentName	`GetComponentName() : string` Returns the name of the Component.
GetMajorVersion	`GetMajorVersion() : int` Returns the major version number of the Component.
GetMinorVersion	`GetMinorVersion() : int` Returns the minor version of the Component.
GetNameForEnum	`GetNameForEnum(enumTypeName, integralValue) : string` Gets enum identifier name the given integer value for the named enum type. Enums are created via the parameter dialog in the symbol editor.
GetStateForDisplay	`GetStateForDisplay() : string` Returns a string from Components's cystate file. The return value can be an empty string, "Prototype," "Obsolete," "Incompatible." An empty string indicates the Component is in the production state, there is no state file, there is a bad entry in the file, or has no entry for the target silicon.

1.5.3.3 *Misc. / Utility Functions*

These are miscellaneous/utility functions, used as described.

Function Name	Description

GetApiCallbackHeaderInclude	`GetApiCallbackHeaderInclude() : string`
	Returns the string #include “cyapicallbacks.h” if the customers design project has a cyapicallbacks.h header file. Returns the empty string if it does not exist. This is useful for Components implementing API callbacks and want to be backwards compatible with earlier version of PSoC Creator that do not generate this header by default.
	For example, place the following code in a Component firmware API template rather that directly including cyapicallbacks.h:
	`=GetApiCallbackHeaderInclude()`

GetMarketingNameWithVersion	`GetMarketingNameWithVersion() : string` Gets the application's full name and version (e.g., "PSoC Creator 3.0").
GetErrorText	`GetErrorText()` Returns the error message stored in a value of the error type.
InvalidFileNameErrorMsg	`InvalidFileNameErrorMsg(nameToCheck, ...) : string` Returns a string with an error message indicating why the give name(s) are invalid. See Also: IsValidFileName().
IsAssignableName	`IsAssignableName(nameToCheck) : bool` Returns true if the given name is legal for the INSTANCE_NAME parameter. This is typically used to validate user input when renaming a Component instance. Valid instance names: ■ Cannot be a C, C++, VHDL, or Verilog keyword. ■ Must start with a letter. ■ May contain letters, digits, and underscore (_). ■ May not contain 2 or more consecutive underscores (_).
IsError	`IsError(parameterToCheck) : bool` If the type of the argument is the error type, return true; else, return false.

1.5.3.3 Misc. / Utility Functions (continued)

These are miscellaneous/utility functions, used as described.

Function Name	Description
IsValidCCppIdentifierName	`IsValidCCppIndentiferName(nameToCheck)` Returns true if the given string is a legal C and C++ identifier name, otherwise it returns false.
IsValidCCppIdentifierWithError	`IsValidCCppIndentiferWithError(nameToCheck)` Returns true if the given string is a legal C and C++ identifier name. Returns a value of the error type with appropriate message if the identifier is not a legal C or C++ identifier.
IsValidFileName	`IsValidFileName(nameToCheck, ...)` Takes 1 or more arguments. Each argument is a ; delimited list of file names. Returns true if all file names are valid. Valid file names: ■ Are not the empty string. ■ Do not start with a whitespace character. ■ Do not end with a . or whitespace. ■ Are not reserved or restricted by the underlying operating system file system. Windows-based file systems typically forbid ASCII/ Unicode characters 1 through 31, as well as double quote ("), less-than (<), greater-than (>), pipe (|), backspace, null, and tab (\t), colon (:), backslash (\), and forward slash (/). ■ Are not reserved by the operating system. Windows systems reserve the names CON, PRN, AUX, NUL, COM1 – COM9, and LPT1 – LPT9. See Also: InvalidFileNameErrorMsg().
UnAssignableNameErrorMsg	`UnAssignableNameErrorMsg(instanceName)` Returns an string with an error message for the given potential value of the INSTANCE_NAME parameter. See Also: IsAssignableName().

1.5.3.4 Deprecated Functions

These are deprecated functions that are kept for backward compatibility purposes.

Function Name	Description
GetArchitecture	Not recommended for use. Replaced by GetDeviceFamilyName().
GetArchMemberName	Not recommended for use. Replaced by GetDeviceInternalName().
GetArchMemberName2	Not recommended for use. Replaced by GetDeviceInternalName().
GetDeviceFamily	Not recommended for use. Replaced by GetDeviceSeriesName().
GetDeviceFamily2	Not recommended for use. Replaced by GetDeviceSeriesName().
GetPartNumber	Not recommended for use. Replaced by GetDevicePartNumber().
GetPackageName	Not recommended for use. Replaced by GetDevicePackageName().
GetHierInstanceName	Not recommended for use. Use the $INSTANCE_NAME parameter.
GetShortInstanceName	Not recommended for use. Use the $CY_INSTANCE_SHORT_NAME parameter.
IsInSystemBuilder	Not recommended for use.

1.5.4 User-Defined Types

User-defined types (or enumeration types) are used to define parameters for symbols whose value comes from the enumeration. User-defined types can be “inherited.” For example, you can create a UDB implementation of the counter that is the Verilog implementation and a symbol. This symbol is placed in a top-level schematic with another symbol for the fixed function block. You could re-define all of the enumerated types and open up error possibilities or the top-level Component can use (inherit) the enumerated types from the lower level Component. See Define Symbol Parameters on page 31 and Add User-Defined Types on page 36.

1.6 References

This guide is one of a set of documents pertaining to PSoC Creator Component creation. Refer to the following as needed:

- PSoC Creator Help (Library Component Project and Basic Hierarchical Design topics, as well as the Symbol Editor topics)
- Cypress.com
  - KBA86338, Creating a Verilog-based Component
  - PSoC Creator Tutorial: Component Creation - Creating a Symbol
  - PSoC Creator Tutorial: Component Creation - Implementing with Verilog
  - Cypress Community Component Forum
- PSoC Creator Universal Digital Block (UDB) Editor Guide
- PSoC Creator Customization API Reference Guide
- PSoC Creator Tuner API Reference Guide
- PSoC Creator System Reference Guide
- Device-specific Technical Reference Manual (TRM)
- Warp™ Verilog Reference Guide

1.7 Conventions Used in this Guide

The following table lists the conventions used throughout this guide:

Convention	Usage
Courier New	Displays file locations and source code: `C:\ …cd\icc\, user entered text`
Italics	Displays file names and reference documentation: *sourcefile.hex*
[bracketed, bold]	Displays keyboard commands in procedures: [**Enter**] or [**Ctrl**] [**C**]
File > New Project	Represents menu paths: File > New Project > Clone
Bold	Displays commands, menu paths and selections, and icon names in procedures: Click the **Debugger** icon, and then click **Next**.

1.8 Revision History

Document Title: PSoC® Creator™ Component Author Guide Document # 001-42697		
Revision	**Date**	**Description of Change**
**	12/4/07	New document.
*A	1/30/08	Updates to all sections.
*B	9/23/08	Updates to all sections. Document name change to Component Author Guide.
*C	5/27/09	Product name change to PSoC Creator. Updates to all sections to remove TBDs and TODOs. Updated the customization section with current APIs.
*D	12/1/09	Updated versioning section. Added section for adding Schematic Macros. Added section for creating debug XML files.
*E	4/28/10	Changed copyright year to 2010.
*F	9/12/10	Added UDB Clock primitive. Added section for precompiled customizers. Updated Versioning section. Added Tuning chapter. Updated Datapath Configuration Tool; moved to Appendix B. Added Best Practices chapter.
*G	11/1/10	Added Bootloader chapter. Updated Verilog section in the Implementation chapter. Added section for adding/creating compatibility file.
*H	6/21/11	Added Verilog example code. Updated references to registers. Added information for Annotation Component. Added information about the exclude file. Updated Add Component Item dialog. Updated the API section to clarify what to include. Updated cystate file section. Updated the Simulation chapter. Added information about the cyversion file. Updated the Control register description. Updated customizer interfaces. Updated built-in parameters and expression functions. Added symbol property Doc.URL.
*I	9/13/12	Updated text for Control file and added reference to PSoC Creator Help. Updated #defines to incorporate PSoC 5LP. Fixed a few typos.
*J	12/4/12	Updated external Component. Fixed cy_ctrl_mode reference errors. Added Tools menu to Datapath Configuration Tool.

Document Title: PSoC® Creator™ Component Author Guide Document # 001-42697		
Revision	**Date**	**Description of Change**
*K	3/19/13	Updated the Expression Functions section. Added note about adding only one Tuner DLL and DMA Capability file. Updated the Verilog UDB Array section for PSoC 4.
*L	8/7/13	Added note for when an input terminal is not visible. Updated the description of GetArchMemberName(). Added section for UDB Editor, and updated the Implementation chapter. Added section for adding DMA Capability File.
*M	2/11/14	Added Visibility Expression field to Symbol properties. Added Expression Functions to support new devices.
*N	8/19/14	Added Library Support for Components. Updated UDB Editor Datapath and State Machine descriptions. Added instructions for adding static libraries. Updated Debug XML file section. Added a section for Dependencies in Finishing the Component chapter.
*O	5/7/15	Revised Chapters 1 – 3 and rearranged other chapters to improve document flow. Removed UDB Editor information to refer to the UDB Editor Guide (spec # 001-94131). Added “Advanced” to Customizing, Tuning, and Bootloader chapters. Moved the Datapath Configuration Tool appendix to a separate document (spec # 001-96549).
*P	8/25/15	Updated Local Built-In Parameters section. Add Macro Callbacks section. Updated Tuning Support to include SPI or UART. Fixed broken references to Datapath Configuration Tool. Updated New Project wizard. Updated Best Practices Clocking section.
*Q	1/11/16	Minor document edit.
*R	2/11/16	Minor document edit.
*S	8/30/16	Updated New Project screen capture. Added a note about editing the .cystate file. Updated Add Component Item screen captures and descriptions. Updated Symbol Parameters and Expression Functions. Added section for adding custom context menu items. Updated Debug XML File attribute. Fixed incorrect reference for information about control files.
*T	3/30/17	Updated copyright and logo. Updated Component requirements section. Updated Defining Symbol Information chapter to reflect GUI updates. Updated Macro Callbacks section. Minor edits throughout.

2. Creating Projects and Components

This chapter covers the basic steps to create a library project and add symbols using PSoC Creator.

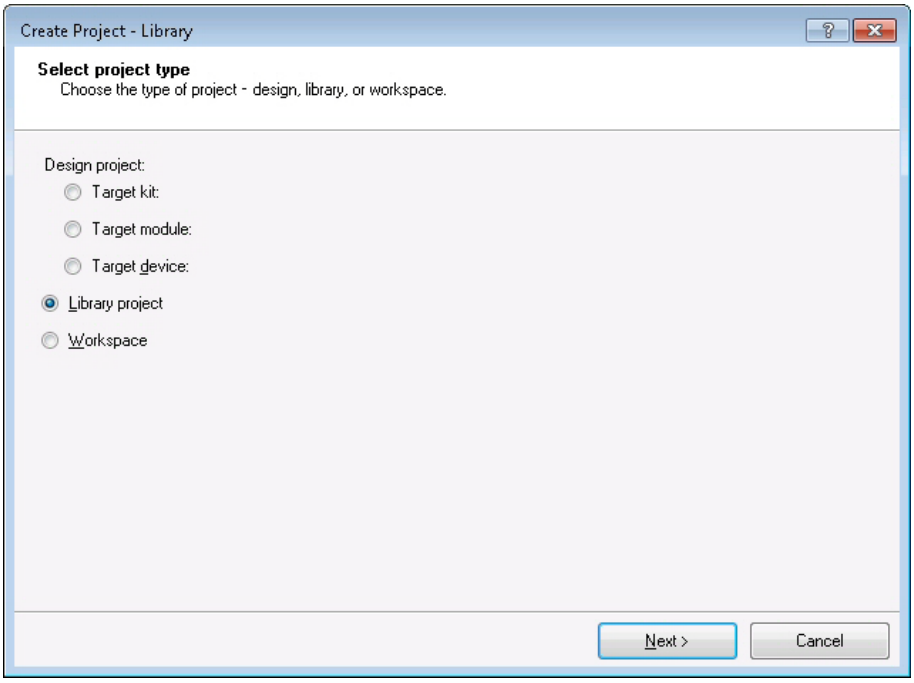
2.1 Create a Library Project

You can create one or more Components in either a design project or a library project. The main difference is that a design project is usually geared toward a specific device and a specific design goal, while a library project is just a collection of Components. Refer to the PSoC Creator Help for more information.

To use the Components, add the project in which they are contained as a dependency in PSoC Creator. See Add Dependency on page 107 for more information.

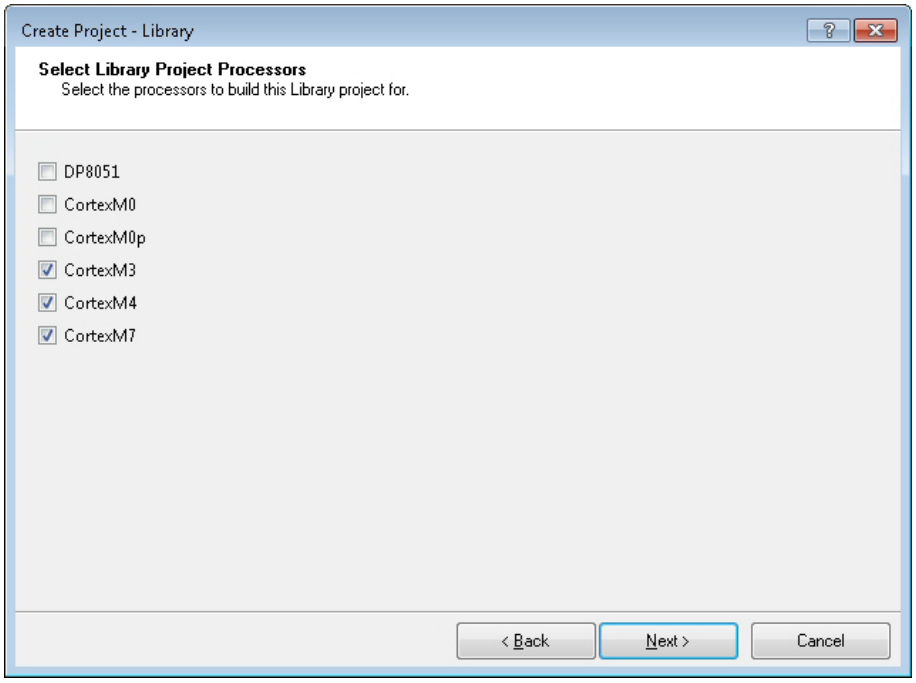
To create a library project:

1. Click **File > New > Project** to open the New Project wizard.



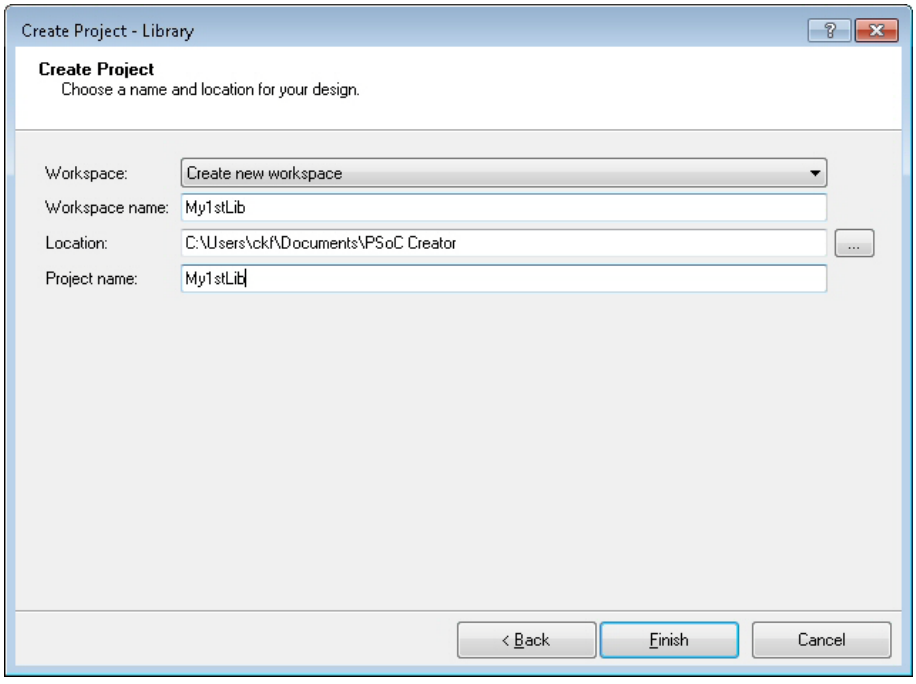
2. Select the **Library project** type and click **Next >**.

3. Select one or more processors for which the library project will be built.



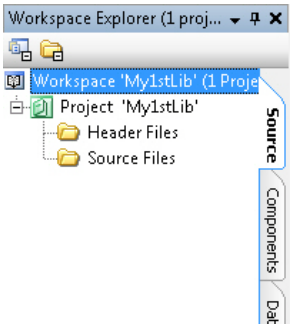
Note You cannot select the DP8051 processor and any of the Cortex processors, because they are not compatible. An error will display.

4. If a workspace is already open, select **Add to current workspace** or **Create new workspace**. Also, as desired, enter a **Workspace Name** and **Project Name**, and click the ellipsis (**...**) button to specify the **Location** to store the project.



5. Click **Finish**.

The project displays in the Workspace Explorer under the **Source** tab.



2.2 Add a Component Item (Symbol)

The first step in the process of creating a Component is to add one of several types of Component items to your project, such as a symbol, schematic, or Verilog file. When you add a Component item, you also create a Component indirectly. The process used to add any of these Component items is the same; however, each of the Component items requires a different set of steps to complete.

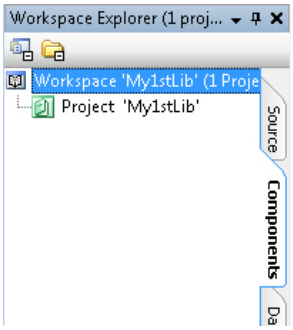
This section will focus on creating a symbol as the first Component item. There are two methods to create a symbol: an empty symbol and the symbol wizard.

Note You may also auto-generate a symbol from a schematic or Verilog, but this section will not cover that process. Refer instead to the PSoC Creator Help for instructions.

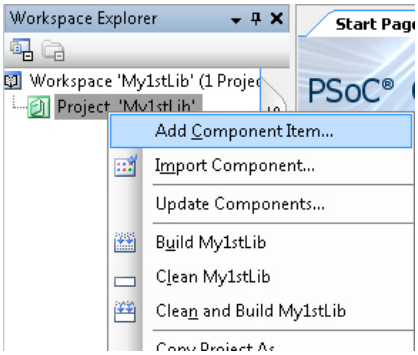
2.2.1 Create an Empty Symbol

This section describes the process to create an empty symbol and add shapes and terminals. (See Create a Symbol using the Wizard on page 29 for an alternate method of creating a symbol.)

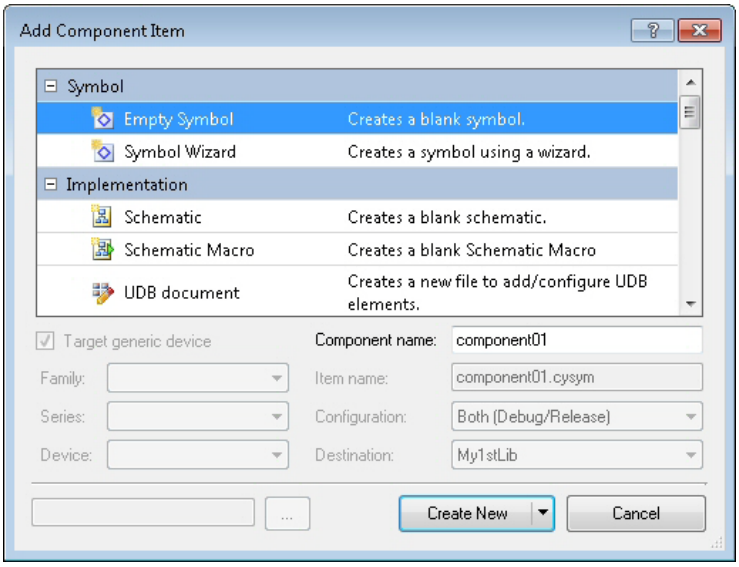
1. As needed, open the appropriate PSoC Creator Library project.
2. Click on the **Components** tab of the Workspace Explorer.



3. Right-click on the project and select **Add Component Item...**



The Add Component Item dialog displays.



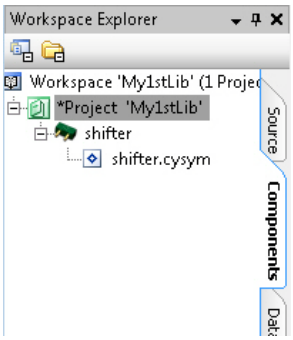
4. Select the Empty Symbol icon.
5. Enter the **Component name**, including version information.

   Note The symbol and all Component elements except schematic macros, customizer source files, and API source files will inherit this Component name. See Name Considerations on page 11 and Component Versioning on page 12.

   Also, there can be only one symbol file in the Component, and that symbol is always generic. Thus, the **Target** options are disabled.
6. Click **Create New** to allow PSoC Creator to create a new symbol file.

   Note You can also select **Add Existing** from the pull-down menu to select an existing symbol file to add to the Component.

The symbol displays in the Workspace Explorer tree, with a symbol file (.cysym) listed as the only node.



The Symbol Editor also opens the *<project_name>.cysym* file, and you can draw or import a picture that represents the Component.

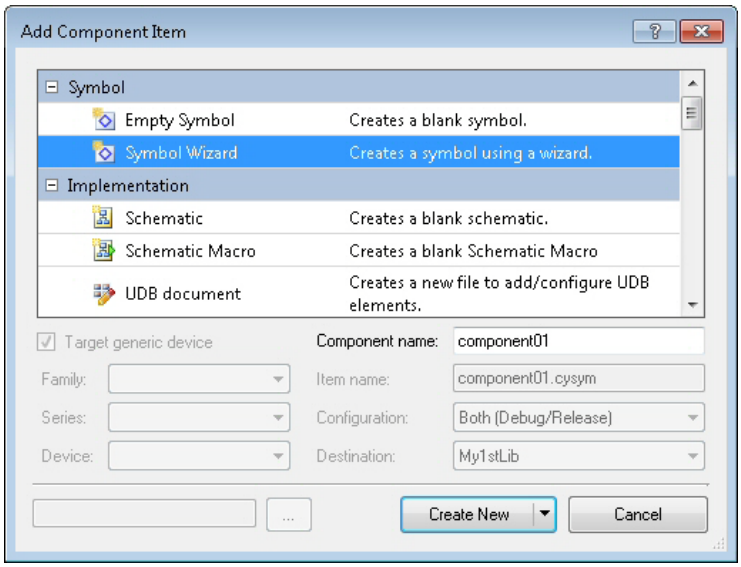
7. Draw basic shapes and terminals to define the symbol using the Symbol Editor. Refer to the PSoC Creator Help for more information.

   Note The plus sign (or cross-hairs) in the middle of the Symbol Editor canvas depicts the origin of your symbol drawing.

8. Click **File > Save All** to save changes to your project and symbol file.

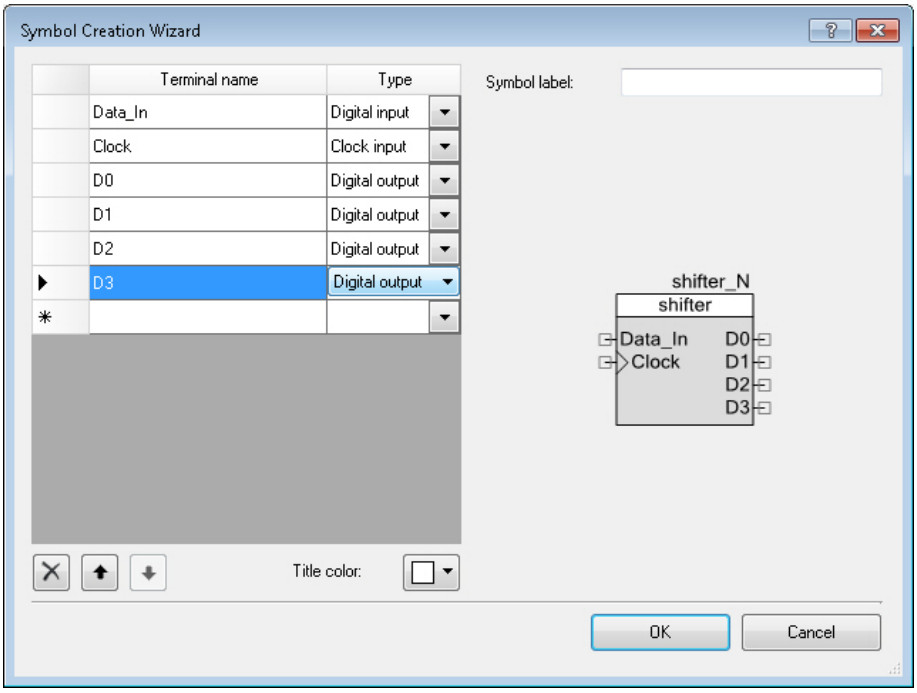
2.2.2 Create a Symbol using the Wizard

The Add Component Item dialog contains a Symbol Wizard icon in addition to the Empty Symbol icon. The benefit of using this wizard template is that PSoC Creator will create the basic symbol drawing and terminals for you.



1. Follow the instructions for creating a Component as described in Create an Empty Symbol on page 27.
2. Instead of choosing the Empty Symbol icon described in Step 4 on 28, choose the Symbol Wizard icon.

After clicking **Create New** on the Add Component Item dialog, the Symbol Creation Wizard displays.



3. Under **Add New Terminals**, enter the **Name**, and **Type** for the terminals you wish to place on the symbol.

   The **Symbol Preview** section will show a preview of how your symbol will appear on the Symbol Editor canvas.
4. Use the **Delete**, **Up**, and **Down** buttons to move and delete terminals, as needed.
5. Optionally, choose a **Title color** and specify a **Symbol label**.
6. Click **OK** to close the Symbol Creation Wizard.

   As with creating an empty symbol, the new Component displays in the Workspace Explorer tree, with a symbol file (.cysym) listed as the only node. However, your Symbol Editor will display the symbol created by the wizard, and it will be centered on the cross-hairs.
7. Make changes to your symbol drawing, as needed.
8. Click **File > Save All** to save changes to your project and symbol file.

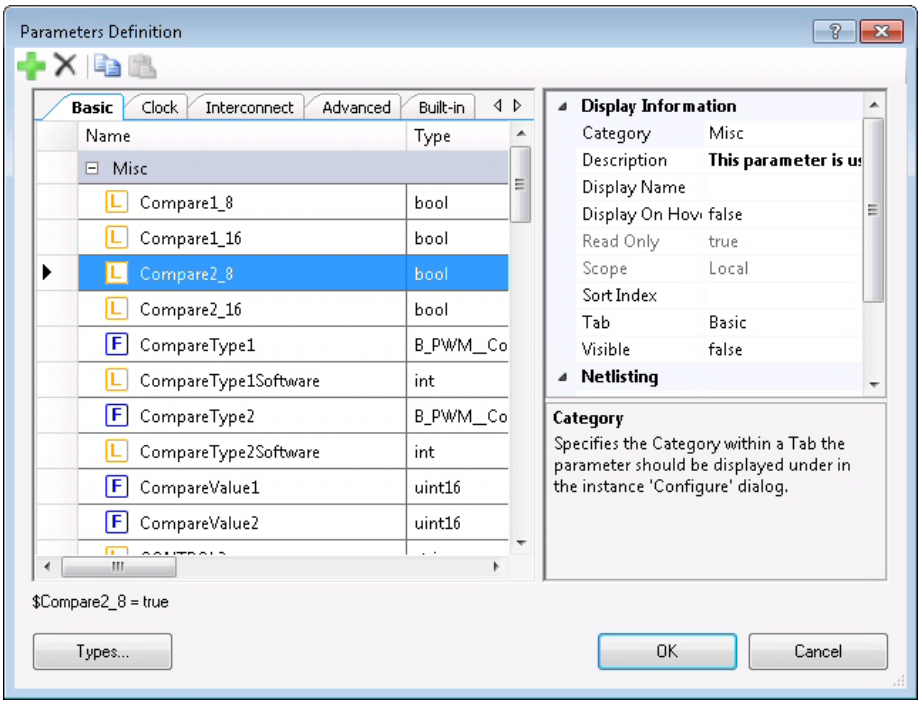
3. Defining Symbol Information

This chapter covers the process to define various symbol information, such as parameters, validators, and properties. For detailed information about symbol parameters, refer to Component Parameter Overview on page 13.

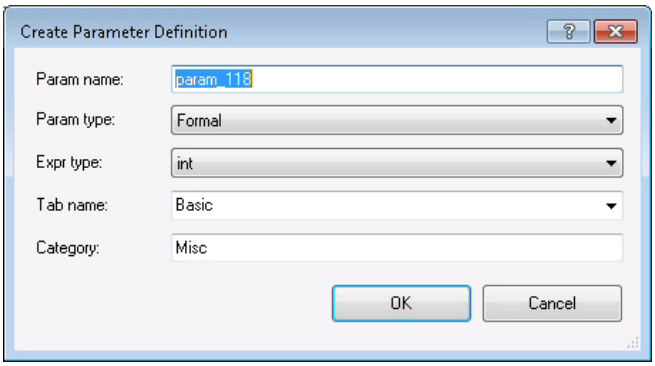
3.1 Define Symbol Parameters

To define parameters for a symbol:

1. Make the symbol file active by clicking the Symbol Editor canvas or the symbol file tab.
2. Right-click on the canvas and select **Symbol Parameters...** to open the Parameters Definition dialog.



3. To create one or more parameters for each symbol, click the **Add** button to open the Create Parameter Definition dialog.

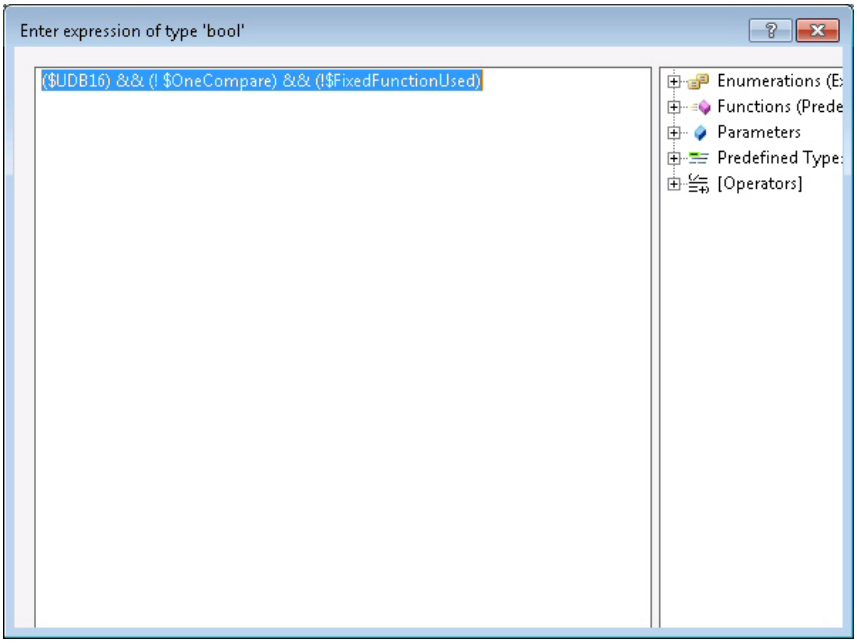


On this dialog, set the following basic parameter information.

- **Param name** – This is the name of the parameter.
- **Param type –** Parameters on a symbol can either be Formal or Local. Both can be displayed in the instance Configure dialog, but there are some key differences between them. See Formal versus Local Parameters on page 13 for more information.
  - Formal F – These are how users configure a Component. They are inputs to the Component from the user. They serve the same purpose as arguments to a C function. Formals are normally visible and editable. They can be temporarily hidden and/or made read only.
  - Local L – These are additional information required by the Component. They are calculated from formals. They serve the same purpose as stack variables in a C function. Locals are normally hidden, and are always read only. They can be made visible to show useful derived information (e.g., like a baud rate).
- **Expr type** – This is the expression type. Select the appropriate type from the pull-down menu.
- **Tab name** – This is the name of the tab where the parameter will be located. Select a name from the pull-down menu, if available, or type a name for a new tab.
- **Category** – This is the category name under which this symbol displays in the Parameter Editor dialog. This is a way to group parameters together.

When finished entering this information, click **OK** to close the Create Parameter Definition dialog.

4. Back on the Parameters Definition dialog, type a default **Value** for the parameter. If needed, click on the **Expression Editor** button f(x) to open the Expression Editor to type more complicated expressions.



Note The default parameter value only defines the value used when a Component is first instantiated. The value is a snapshot at that time. If the default value for the Component is subsequently changed, that value does not get propagated down to any previously instantiated Component.

5. On the right side of the Parameters Definition dialog, define the following properties:
   - **Category –** This is the same as on the Create Parameter Definition dialog from Step 3. You can change the category here, if needed.
   - **Description –** The description that displays to the end-user for this parameter in the instance Configure dialog.
   - **Display Name** – The name to display for the parameter in the instance Configure dialog and the instance tooltip. If not set, the parameter name will be used instead. This name should use title-style capitalization. Use it to display a user-friendly name (instead of a legal identifier name) to the user.
   - **Display On Hover** – An expression that specifies if the parameter value displays while hovering the mouse over the instance in the Schematic Editor.
   - **Read Only** – An expression that specifies whether or not the parameter can be changed by the end user.
     - This will always be true for Locals.
     - If a parameter is visible and there is nothing the user can do to make the value editable, a Local should be used instead of a Formal.
     - Only display (set Visible to true) a Formal if the user is expected to edit the value at some point.
   - **Scope** – Specifies the scope in which the parameter is accessible (Formal or Local).
   - **Sort Index** – Overrides the default alphabetical ordering of the parameters. This will also control the tab and category ordering. The tab/category containing the smallest sort index will

be presented first. Any parameter without a sort index specified will be sorted alphabetically within their category after all parameters that do have an index provided.

- **Tab** – Specifies the tab under which the parameter should be displayed on in the instance Configure dialog.
- **Visible** – An expression that specifies if the parameter should be visible in the instance Configure dialog.
- **Hardware** – If true, the parameter is included in the Verilog netlist generated during the netlisting phase of the build process.
- **Check Range –** When true, this will check that the parameter's value after each evaluation is in the range for the associated type. If out of range, PSoC Creator generates an expression evaluation error. Valid ranges are as follows:

Type	Valid Range
Enumerated Type	Any of the integer values that the enumerated type defines
Signed 16-bit Integer Type	-32768 .. x .. 32767
Unsigned 16-bit Integer Type	0 .. x .. 65535
Signed 8-bit Integer Type	-128 .. x .. 127
Unsigned 8-bit Integer Type	0 .. x .. 255
Boolean Type	true and false
All Other Types	All values are valid

- **Validators –** One or more validation expressions for the parameter. See Add Parameter Validators on page 35 for how to use this field; see Expression Evaluator appendix on page 165 for more information about CyExpressions.

6. When you are finished adding parameters, click **OK** to close the Parameters Definition dialog.

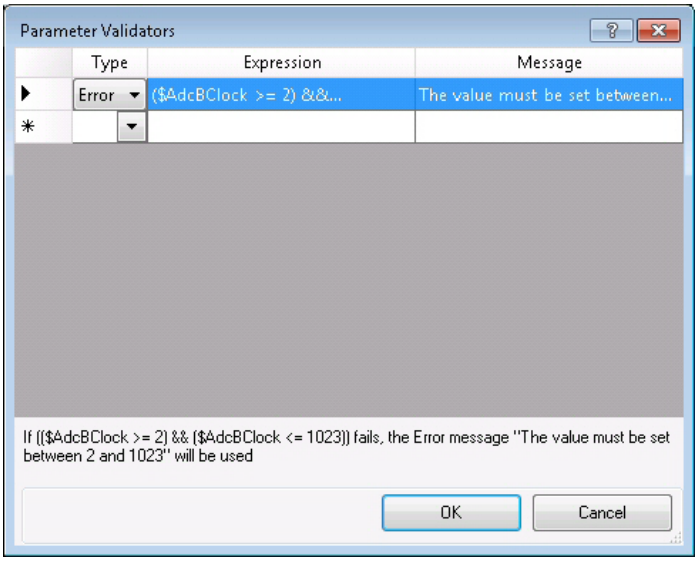
   Note You can copy one or more parameter definitions by using the Copy/Paste buttons from the toolbar. Also, multiple parameter properties can be changed at once by having multiple parameters selected when changing the property.

3.2 Add Parameter Validators

Parameter validation expressions are evaluated in the context of the instance. The validation expressions can refer to both formal and local parameters. They can be used to generate either errors, warnings, and/or infos. Only errors will stop a user from committing changes in the instance Configure dialog. When multiple validators fail for a single parameter, only the highest priority type messages will be displayed: Errors, then Warnings, then Information.

To add a parameter validator:

1. On the Parameters Definition dialog, click in the **Validators** field and then click the ellipsis button. The Parameter Validators dialog displays.



2. In the **Type** file, select Error, Warning, or Info as the expression type.
3. In the **Expression** field, type in the validation check expression. You can reference parameter names using the $ symbol. For example:

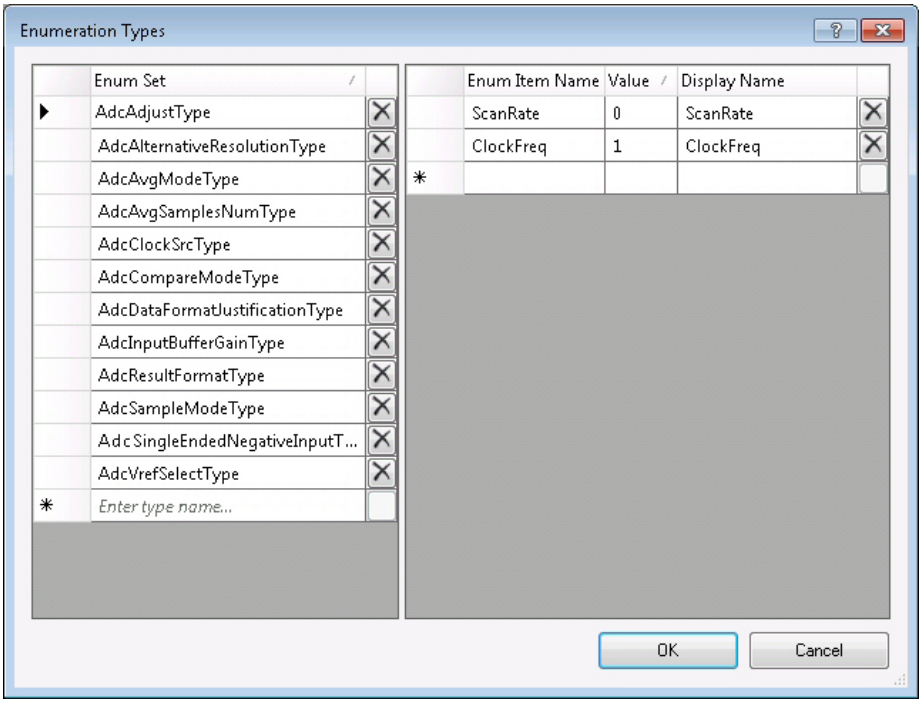
```
$param1 > 1 && $param < 16
($param1 == 8) || ($param1 == 16) || ($param1 == 24)
```

   See Expression Evaluator appendix on page 165 for more information.
4. In the **Message** field, type in the message to display if the validation check is not met.
5. To add another validator, click in any field for an empty row marked with >*, and type in the fields as appropriate.
6. To delete an expression, click the > symbol in the first column of the row to delete, and press the [**Delete**] key.
7. To close the dialog, click **OK**.

3.3 Add User-Defined Types

To create a user-defined type:

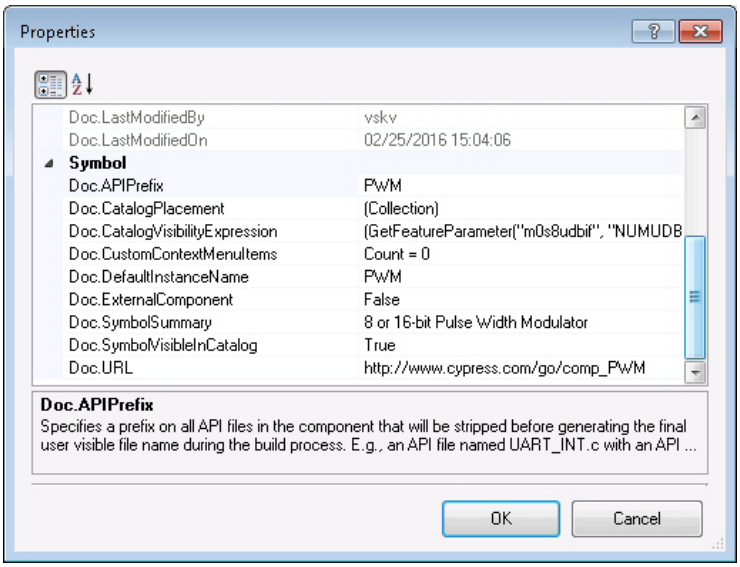
1. Click the **Types...** button at bottom of the Parameters Definition dialog to open the Enumeration Types dialog.



2. Type a name in the bottom row of the **Enum Set** table (where it says '*Enter type name…*').
3. In the **Enum Item Name** table (on the right), click in the empty row and type a name for the 1st name/value pair of the enumerated type; type a value under **Value** or accept the default.
4. Optionally, enter a string in **Display Name** that will display in the Component's Configure dialog pull down menu for that parameter.
5. Enter as many enum sets as needed and click **OK** to close the Enumeration Types dialog.

3.4 Specify Document Properties

Every symbol contains a set of properties that define different aspects of that symbol. To open the Properties dialog, right-click on the symbol canvas and select **Properties**.



You can specify the following document properties for each Component, as applicable:

- **Symbol Properties**
  - **Doc.APIPrefix –** Specifies a prefix that will be stripped on all API files in the Component before generating the final user-visible file name during code generation.

    For example, if you enter “Counter” as shown above for a counter Component, the generated header file for a Component with instance name “Counter_1” will be *Counter_1.h*. If you enter nothing in this property, the generated file would be *Counter_1_Counter.h*.

    Note The APIPrefix is only applied to API files that are part of a project on disk. It is not applied to API files generated by the API customizer.
  - **Doc.CatalogPlacement –** Defines how the Component will display in the Component Catalog. See Define Catalog Placement on page 38.
  - **Doc.CatalogVisibilityExpression –** Used to enter an expression to show the symbol in the Component Catalog. If this expression evaluates to 'false' and the "Show Hidden Components" option (Tools > Options > Design Entry > Component Catalog) is not enabled, the symbol (or schematic macro) will not be displayed in the Component Catalog.
  - **Doc.CustomContextMenuItems –** Used to add additional context menu items to open files, such as an API Reference. See Add Custom Context Menu on page 39.
  - **Doc.DefaultInstanceName –** Defines the default instance name for Components. If left blank, the instance name defaults to the Component name.
  - **Doc.ExternalComponent –** Specifies whether the symbol is an external Component: true or false. See Create External Component on page 38. **Note** The label reads as “Annotation” due to legacy issues.
  - **Doc.SymbolSummary –** Used to enter a brief description shown in the Component Catalog.
  - **Doc.SymbolVisibleInCatalog –** Specifies whether the Component displays in the Component Catalog: true or false.

- **Doc.URL –** Used to enter a complete web address or a file location on disk. If you enter a valid value in this field, then various menu items become activated to allow a user to navigate to the applicable web page, or open the specified file.

3.4.1 Create External Component

An external Component contains a symbol and parameters, but no implementation. It is used to document the design schematic, typically by representing off-chip devices and wiring. Cypress provides a number of external Components in a library, including resistors, capacitors, diodes, and so on. To specify a Component as an external Component:

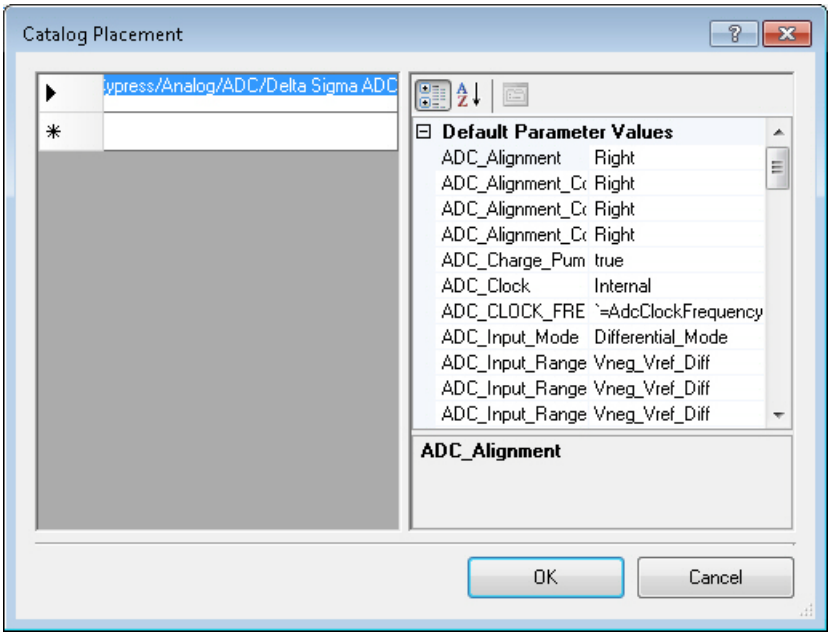
1. Set the **Doc.ExternalComponent** property in the Properties dialog to true.
   This is used to skip the Component during elaboration and to enable DRC errors when digital or analog wires/terminals are connected.
2. Then use an External terminal  from the Design Elements Palette (DEP) to define the symbol connections to off-chip resources.

Note A Component specified as an external Component cannot have non-external terminals; however, a regular Component can include external terminals.

An external Component will generate a DRC if it contains non-external content. The DRC will warn that the content will not be implemented. Likewise, the use of digital or analog terminals in an external Component will also generate a DRC. This DRC will warn that the terminal does not connect to anything functional.

3.4.2 Define Catalog Placement

You can define how symbols will be displayed in the Component Catalog under various tabs and trees using the Catalog Placement dialog.



If you do not define catalog placement information, the symbol will display by default under **Default > Components**.

1. Open the dialog by clicking the ellipsis button in the **Doc.CatalogPlacement** field in the Properties dialog.
2. On the left side of the dialog, enter placement definition in the first row of the table, using the following syntax:

   `t/x/x/x/x/d`

   `t` – The tab name. The tab order displayed in the Symbol Catalog is alphabetical and case insensitive.

   `x` – A node in the tree. You must have at least one node.

   `d` – The display name for the symbol (optional). If you do not specify the display name, the symbol name will be used instead; however, you must use the syntax: `t/x/`.

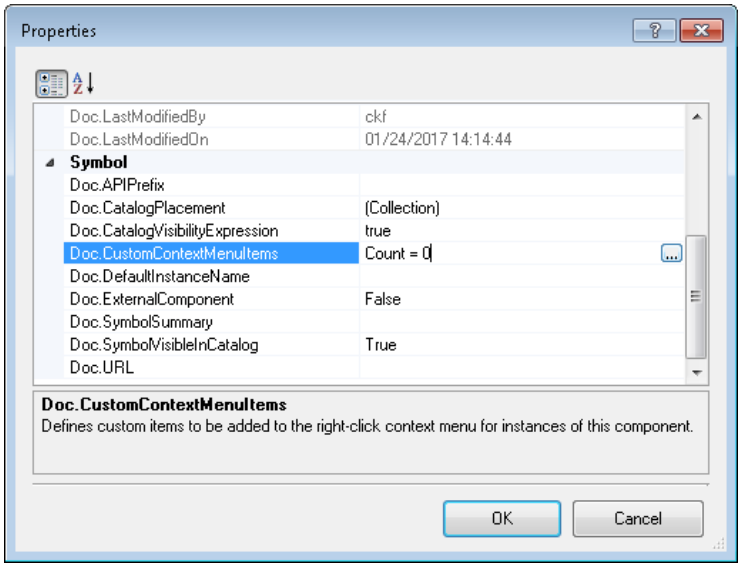
   For example, in the default Component Catalog for the Component named “PGA,” the tab name (`t`) is **Cypress**, the first node (`x`) is **Analog**, the second node (`x`) is **Amplifiers**, and the display name (`d`) is **PGA**: `Cypress/Analog/Amplifiers/PGA`.
3. On the right side of the dialog, under **Default Parameter Values**, enter default parameter values for this particular catalog placement definition, as needed.

   Note The default parameter value only defines the value used when a Component is first instantiated. The value is a snap shot at that time. If the default value for the Component is subsequently changed, that value does not get propagated down to any previously instantiated Component.
4. If desired, add more rows in the left side table to enter more than one catalog placement definition; set the same or different default parameter values for each additional row.
5. Click **OK** to close the dialog.

3.4.3 Add Custom Context Menu

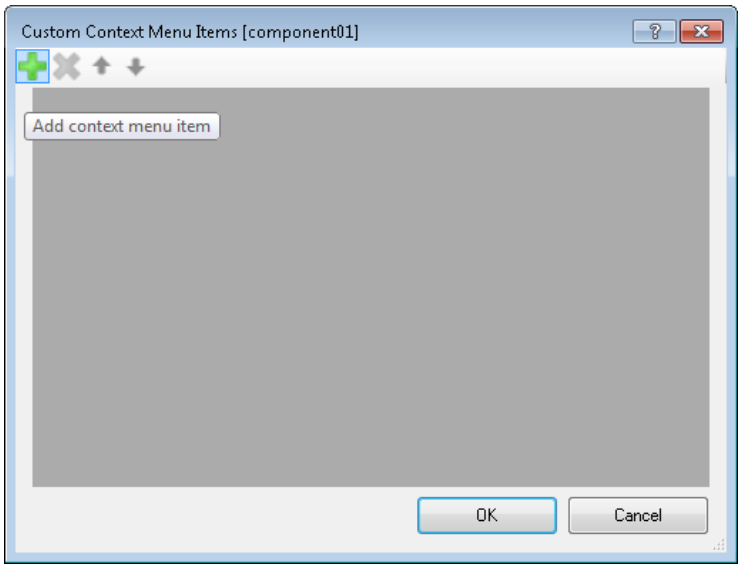
You can add custom context menus that will be visible when the user right-clicks on an instance of the Component. For example, you can add a context menu item to open an API Reference document, tuner application, and any other additional files that a user might need to open.

1. Right-click on the symbol canvas and select **Properties...** to open the Properties dialog.

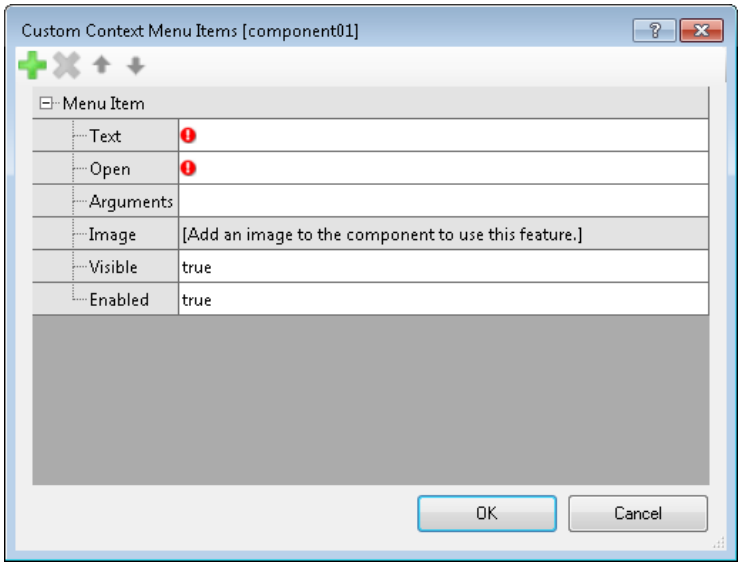


2. Click the ellipsis button in the **Doc.CustomContextMenuItems** field to open the Custom Context Menu Items dialog.

3. Click the “+” icon to add a menu item.



4. Complete the necessary fields.



The fields include:

- **Text –** This text will be shown for the context menu item.
- **Open –** Specifies document file path or URL to open when the user selects the context menu item.

  If the file to open exists in the same directory as the *<project>.cysym* file, choose the file from the drop-down list. Otherwise, manually type in file location's full/relative path or document URL (e.g., .\MyComp.chm, www.cypress.com, etc.). Relative paths are relative to the Component's folder.

  You can add links relative to the project directory by using the macro: `${CyPrjDirPath}`.

  You can add links relative to the PDL by using the macro: `${CyPdlPath}`.

  Example:

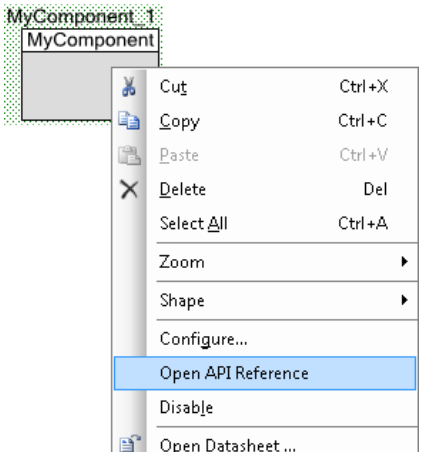
  ```
  ${CyPdlPath}\adc\adc_scan_multich_polling_sw\pdl_user.h
  ```

- **Arguments –** Optional. Specifies arguments to use when the 'Open' path is executed.

  Supports the following macro substitution: `${CyPdlPath}`, `${CyPrjDirPath}`.
- **Image –** Optional. Specifies the image file path to be shown on the left side of the context menu item.

  You must place an image file in the same directory as the *<project>.cysym* file to enable the drop-down list. Otherwise, you cannot specify an image file.
- **Visible –** Expression to determine whether this context menu item will be shown or not.
- **Enabled –** Expression to determine whether this context menu item will be enabled or not.

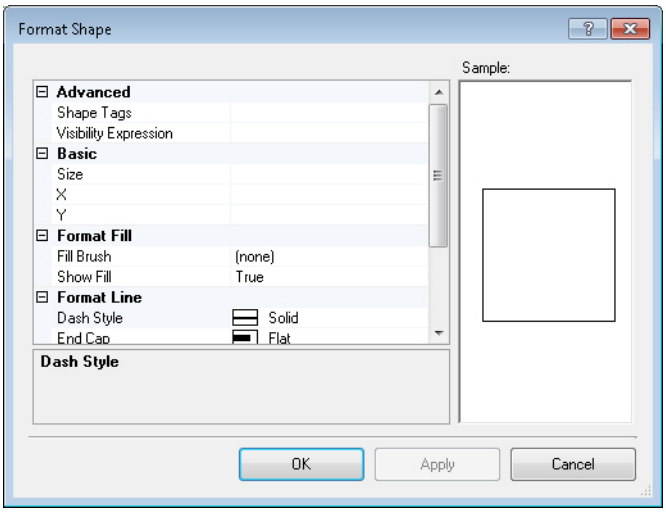
5. Click **OK** to close the Custom Context Menu Items dialog, and then click **OK** to close the Properties dialog.
6. Place the Component in a design project schematic and right-click on the Component instance. Note that the menu item is shown as specified.



3.5 Define Format Shape Properties

The Format Shape dialog is used to define properties for shapes. The types of properties available will vary depending on the selected shape(s). For example, text offers font, color, size, etc., while a line offers width, pen type, end cap, etc.

To open this dialog, select one or more shapes and click the **Format Shape** button.



3.5.1 Common Shape Properties

Most of the common shape properties in this dialog are self-explanatory, and they are similar to the shape formatting you will see in other word-processing or drawing programs. For most of these properties, you select a value from a pull-down menu.

3.5.2 Advanced Shape Properties

For some shapes, such as instance terminals, there are some advanced properties, including:

- **Default Expression –** Defines the default value for the shape expressed as `<size>'b<value>`, where `<size>` = number of bits, and `<value>` = the binary value.
- **Shape Tags –** Defines one or more tags to be associated with one or more shapes for use with the shape customization code; see Customizing Components (Advanced) on page 113.
- **Visibility Expression –** Defines an expression to evaluate whether or not the selected shape is visible. For example in the UART Component, this property is defined with the variable $FlowControl for the rts_n and cts_n terminals. If $FlowControl is set to true in the instance, then these terminals display in the schematic; if false, they are hidden.

  Note When setting the Visibility Expression of a terminal, the Default Expression must also be specified. It will be used when the Visibility Expression of that terminal evaluates to false.

4. Adding an Implementation

A Component implementation specifies the following:

- which devices are used, as well as which are not supported
- how the Component is built
- internal architecture and how it will be used

PSoC Creator allows you to create a Component using several design strategies for your specific needs. Depending on the design goal of your particular application, the design strategy may vary greatly between different implementation options. There are schematic implementations, schematic macros, hardware implementations using the universal digital block (UDB) Editor, hardware implementations using Verilog, and software implementations.

Schematic

The easiest and the most straight forward strategy to implement your Component is to use a schematic (see Implement with a Schematic on page 45). A schematic allows you to place existing Components on the design canvas much like any design. The design is then encapsulated into a Component with its own input and output terminals with a unique function that can be used in other designs. Use this method if you are using existing blocks to perform the new function.

This method does not allow you to work directly with the UDB elements and therefore limits the full potential available in UDBs, such as datapaths and advanced programmable logic device (PLD) features. If a separate UDB-based Component is available, then it can be used in a schematic-based Component like any other available in the Component Catalog.

Schematic Macro

Similar to the schematic implementation, it is possible to create a schematic macro that contains a design with pre-configured parameters and settings with several Components already linked together. The schematic macro then becomes available in the Component Catalog and allows you to use the pre-configured settings without worrying about the specifics of Component configurations. A schematic macro is usually created for Component implementations that have complex configurations. It is not normally used as a template for large designs with many Components on the schematic. See Create a Schematic Macro on page 46 for details on implementing a schematic macro.

UDB-Based Designs

For UDB-based designs, use the UDB Editor, or use Verilog and the Datapath Configuration Tool. Both methods allow access to UDB elements. However your preference may differ depending on the circumstances.

- The UDB Editor is used to graphically describe the digital functions in UDBs. It does not require you to know Verilog or have knowledge of the Datapath Configuration Tool. Therefore, it may be an easier option than the Verilog method. The UDB Editor takes care of many internal

configuration details simply by specifying the parameters of the UDB blocks on the design canvas. These graphical configurations are then used to generate Verilog code in real time and can be used to observe how the blocks translate into code. However, the trade-off in using the UDB editor is that some advanced features of the UDB are not supported with this method. Also for users who know Verilog, it may be faster for them to use the Verilog method when designing state-machines. Refer to the separate *UDB Editor Guide* available from the PSoC Creator Help menu.

- Implementing a UDB Component with Verilog is the most versatile but also the most complex and is not recommended for beginners. Verilog allows you to design state machines that are more refined than the UDB Editor. It may also be easier for users who are familiar with digital logic and Verilog to use this method. To gain access to the datapath, you will also need to use the Datapath Configuration Tool, which allows full functionality of the datapath. With this you will be able to use the full potential of UDBs to create efficient and complex logic. See Implement with Verilog on page 55.

Software

Implementations can also be just software. Implementing a Component with software means that no hardware is used or referenced in the design. For instance, this method is used in Components that act as an interface to link several codes together. See Implement with Software on page 68 for more details.

Exclude

If your Component is intended to support only certain PSoC families, series, or devices, then it is also possible to explicitly state that this Component is supported only for those specific parts. See Exclude a Component on page 69 for more information on excluding Components.

Implementation Priority

Note that for implementations using hardware such as schematic, UDB Editor or Verilog, PSoC Creator will use only one of these to make the Component and will ignore the rest. The priority of these implementations is as follows.

1. Verilog – Has the highest priority. Creator ignores UDB Editor or schematics in the Component workspace.
2. UDB Editor – Has higher priority than a schematic, but will be ignored if there is a Verilog file in the Component workspace.
3. Schematic – Has the lowest priority and will be ignored if either a Verilog or UDB Editor file is in the workspace.

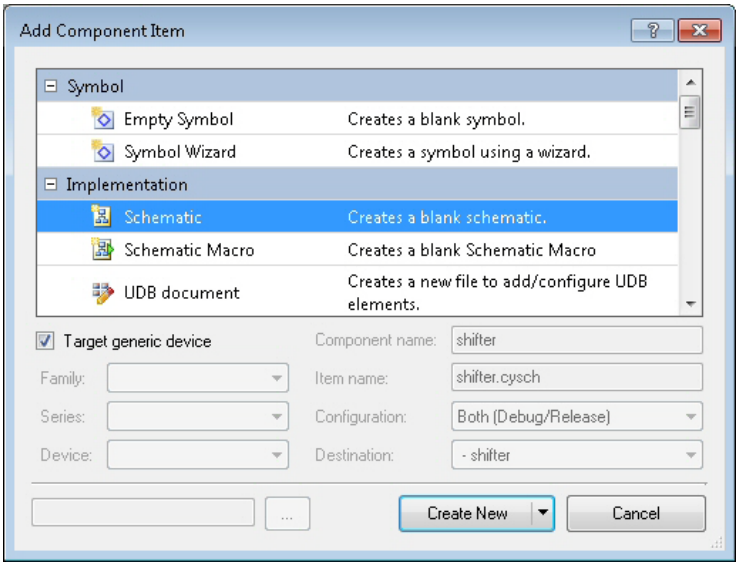
4.1 Implement with a Schematic

The schematic is a graphical diagram that represents the internal connectivity of the Component. When you finish creating the Component, instantiate it in a design, and build it (see Finishing the Component on page 81), the necessary files are generated to provide information for use with Warp and the fitter, etc.

4.1.1 Add a Schematic

To add a schematic file to your Component:

1. Right-click on the Component and select **Add Component Item**.
   The Add Component Item dialog displays.



2. Select the Schematic icon.
   Note The schematic will inherit the Component name.
3. De-select the **Target generic device** to specify the Family, Series, and/or Device using the drop down menus, or leave the check box selected to allow the Component to apply to all devices.
4. Click **Create New**.

In the Workspace Explorer tree, the schematic file (*<project>.cysch*) is listed in the appropriate directory, depending on the device option specified.

The Schematic Editor opens the *<project>.cysch* file, and you can draw a diagram that represents the Component connectivity at this time.

4.1.2 Complete the Schematic

Refer to the PSoC Creator Help section “Using Design Entry Tools > Schematic Editor” for details about using the Schematic Editor.

4.1.2.1 Design-Wide Resources (DWR) Settings

DWR settings are bound to DWR Component instances (clocks, interrupts, DMA, etc). Therefore the naming convention for these Components within your Component will assure that they always map correctly.

- They are associated with an auto generated internal ID.
- They are associated by hierarchical instance name.

If you create a Component with a DWR, you can rename it at will with no consequence. The internal ID is used to maintain the association.

If you delete a DWR from your Component, the system will fall back on using the hierarchical instance name. If you then add a new DWR Component, and give it a different name, the *<project>.cydwr* file will lose any settings the end user had associated with their DWR Component.

4.2 Create a Schematic Macro

A schematic macro is a mini-schematic that allows you to implement a Component with multiple elements, such as existing Components, pins, clocks, etc. A Component can have multiple macros. Macros can have instances (including the Component for which the macro is being defined), terminals, and wires.

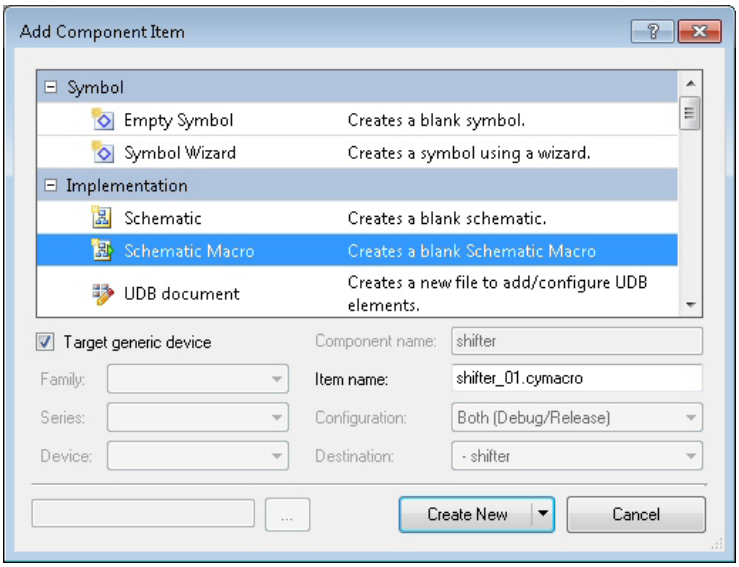
Schematic macros are typically created to simplify usage of the Components. Typical use cases of the Components in schematics will be prepared and made available as macros. The end user will use macros instead of using bare symbols.

4.2.1 Add a Schematic Macro Document

Add a schematic macro to a Component in the same manner as any other type of Component item. To add a schematic macro file to your Component:

1. Right-click on the Component and select **Add Component Item**.

   The Add Component Item dialog displays.



2. Select the Schematic Macro icon.
3. De-select the **Target generic device** to specify the Family, Series, and/or Device using the drop down menus, or leave the check box selected to allow the Component to apply to all devices.

   Schematic macros "belong" to a Component. Macros can be created at the Family, Series, Device, and generic levels. There can be multiple macros at the same level. This is different from other representations (schematic, Verilog) in the Component. The names of macro file names will be unique in a Component.
4. Click **Create New**.

In the Workspace Explorer, the schematic macro file (*<project>.cymacro*) is listed in the appropriate directory, depending on the device option specified.

The Schematic Macro Editor opens the *<project>.cymacro* file. You can add Components, define predefined settings such as a default clock frequency, and override default settings of each Component, etc. You can also use macros to predefine I/O pin configurations necessary for a communication interface or an analog Component, etc.

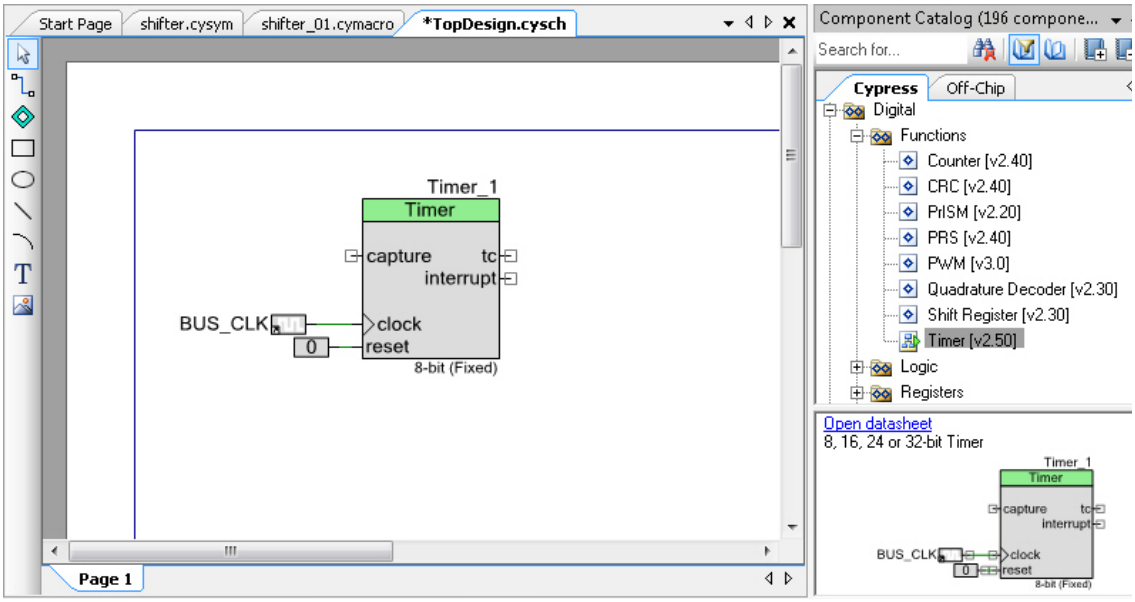
4.2.2 Define the Macro

The following shows an example for creating a Timer macro:

1. Place a Timer Component from the catalog.

   If desired, you could set the parameters of this Component to override the Timer's default values. Refer to the Component datasheet for various parameters available.
2. Place a clock Component from the library, and configure as necessary.
3. Connect the clock output to the "clock" input terminal of the Timer Component.
4. Place a "Logic Low" Component on the schematic and connect it to the "reset" input of the Timer.

5. Right click in the schematic macro somewhere and select **Properties** from the drop down.
   Change the Component catalog placement and summary to the desired settings.
6. Open the Timer symbol file, right click somewhere in the empty space and select **Properties** from the drop down. Make the setting to Hide the Component from the Component catalog.
   In a design project you will see the following:



7. Once you place this in your design, experiment with how you can delete the logic low or change the clock or timer settings, etc.

4.2.3 Versioning

Since macros belong to a Component (see above) they are implicitly versioned through the versioning of the Component. The macro name will not include the version number of the macro or Component.

4.2.4 Component Update Tool

The Component Update Tool is used to update instances of Components on schematics. When a macro is placed on a schematic the "macroness" of the placed elements disappears. The individual parts of the macro are now independent schematic elements. Since there are no "instances" of macros on a schematic, the Component Update Tool has nothing to update.

However, a schematic macro itself is defined as a schematic. That schematic may contain instances of other Components that can be updated via the Component Update Tool. The macro definition schematic is visible to the Component Update Tool and all appropriate updates are available.

4.2.5 Macro File Naming Conventions

The extension of the macro files will be ".cymacro". Default filenames will be <Component name>_ <[0-9][0-9]>.cymacro.

4.2.5.1 Macro and Symbol with Same Name

It is not recommended that Component authors create a macro with the same name and catalog placement as the symbol since this can lead to confusion for the end user.

4.2.6 Document Properties

Schematic macros have many of the same properties as a symbol. See Specify Document Properties on page 37 for more information.

4.2.6.1 Component Catalog Placement

The catalog placement property for the macro is similar to symbol catalog placement, except there are no parameters associated with the placement. The catalog placement editor allows the macro to be placed at multiple places in the Component catalog.

Schematic macros will be listed in the Component catalog similar to the way symbols are displayed in the catalog. If a macro display name is repeated in another Component, the macro is displayed with an index in parenthesis to denote that there are more than one instances of the macro with the same display name.

4.2.6.2 Summary Text

PSoC Creator allows symbols to have associated "summary text" that is displayed in the Component preview area when the symbol is selected in the Component Catalog. Macros will support this same feature. A summary text field will be available in the schematic macro editor and when the macro is selected in the Component catalog the text will be displayed in the preview area (handled the same way that the symbol is handled).

4.2.6.3 Hidden Property

Macros support the hidden property the same way the symbols support it. The property will be available for editing in the macro editor.

4.2.7 Macro Datasheets

Schematic macros will not have a separate datasheet. Instead, all macros defined for a given Component will be documented in a separate section of that Component's datasheet. When selected in the Component Catalog, the macro’s preview area will show a link to the Component's datasheet.

4.2.8 Post-Processing of the Macro

When a macro is dropped on a schematic, the names of instances, terminals, and wires have to be recomputed. At the end of post-processing, the instance, terminal, and wire names should not conflict with the existing names in the schematic.

For each instance, terminal, and wire in the macro, PSoC Creator must assign a non-conflicting name before adding it to the model. Each instance, terminal, and wire is added to the schematic making sure that the HDL name being used for the item is currently not used in the schematic already. The base names of the items will be suffixed with numbers and incremented until a unique HDL name is available.

4.2.9 Example

Assume there is a macro with one instance (named “i2c_master”), three terminals (named “sclk,” “sdata,” and “clk”) and three wires (wires are not named; their effective names are sclk, sdata, and clk, respectively). If the schematic is empty and a macro is dropped on the schematic, the names of the instances, terminals, and wires will be as follows:

- Instance: i2c_master
- Terminals: sclk, sdata, and clk
- Wires: sclk, sdata, and clk

If the macro is dropped again, the names would be as follows:

- Instance: i2c_master_1
- Terminals: sclk_1, sdata_1, and clk_1
- Wires: sclk_1, sdata_1, and clk_1

If the names are already used, then the suffix is incremented until a valid name is available.

4.3 Implement a UDB Component

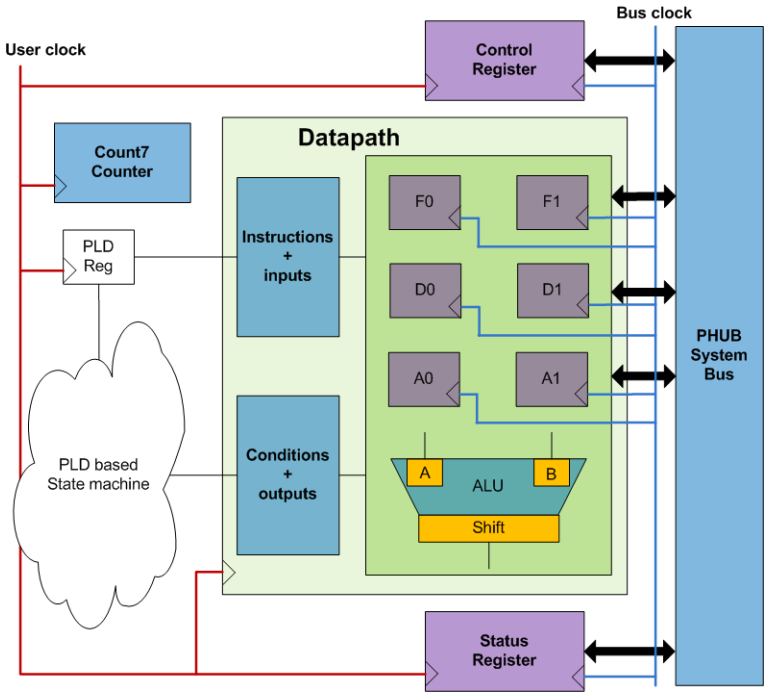
In order to fully utilize the resources available in UDBs, it is necessary to design a UDB Component using either the UDB Editor or with Verilog and the datapath configuration tool. Each method has its advantages and disadvantages, but in general the Verilog method is more advanced than the UDB Editor. In this section we describe the underlying architecture of the UDB and list out the available resources. Following this, designing with UDB Editor and with Verilog are presented.

4.3.1 Introduction to UDB Hardware

A UDB is a combination of uncommitted PLDs, a structured processor module (datapath), control and status registers, and a 7 bit counter (count7) that are connected through flexible routing. These elements can be used to form many types of logic and can be chained together to form large designs. A design may communicate with the CPU, with other hardware blocks in a PSoC device, or both. This flexibility allows the UDB to form logic that links other hardware in your design, or can be a stand-alone block that performs a new function.

4.3.1.1 UDB Overview

The following figure shows a high-level block diagram of a UDB. This figure highlights the main blocks in a UDB and how these are connected and controlled. Internal signals and the individual inputs and outputs are not shown. The blocks are color coded to differentiate between the types. Purple is a register, blue is a fixed block that performs a defined function, green is the datapath, and orange is the input and output of the arithmetic logic unit (ALU) and shifter. White is PLD logic. For detailed information on the UDB architecture, refer to the *TRM*.



UDBs are driven with a user clock and the bus clock. The bus clock reads from and writes to the registers in the datapath and the control/status registers. These data words travel through the PHUB system bus. The user clock drives the blocks in the UDB. Signals in a UDB can be routed to form a hardware output in a Component or can be used to drive the inputs of the structured blocks in a UDB.

- PLD – These are most often used to create logic to control the other structured resources available in a UDB. PLD based designs are composed of both combinational logic and sequential logic. The sequential logic is driven with the user clock. Although PLDs are the most flexible element in a UDB, it is also resource intensive. Therefore it is recommended to use the other structured blocks as much as possible when implementing large designs.
- Datapath – A datapath is an 8-bit wide processor that can be used to perform simple arithmetic and bitwise operations on data words. It can be chained to form 16-, 24-, and 32-bit wide processors. It can have up to 8 user defined instructions that are often driven using the PLD implemented state machine. Datapaths form the core of many UDB designs and should be used in preference over PLD designs when 8-bit words or larger are used. For more information on using the datapath, see Datapath Operation on page 52.
- Control Register – A control register is used in a UDB to communicate with the CPU. Using a control register, it is possible for the CPU to directly send commands to the UDB hardware. The reading and writing of the control register from the CPU is performed at the bus clock, unless it is in sync or pulse mode with the user defined clock.

- Status Register – A status register is used to notify the CPU on the status of the hardware signal states. The rate at which the status register is read by the CPU is controlled using the bus clock whereas the register is written by the UDB at the user clock. Status registers also have the ability to generate maskable interrupts. This is accomplished by using one pin for the interrupt output and the other 7 bits as the maskable triggers for the interrupt.
- Count7 Counter – UDBs contain a 7-bit counter that can be used instead of a counter implementation using PLDs or a datapath. This can save resources if the required counter bits are between 4 and 7 bits. The terminal count of the counter can then be used throughout your design.

4.3.1.2 Datapath Operation

A UDB-based PSoC device datapath is essentially a very small 8-bit wide processor with 8 states defined in a "dynamic configuration." Consecutive datapaths can be tied together to operate on wider datawidths using one of the following pre-defined modules. The following description provides a high-level description of the datapath and how it is used. For full details on the datapath, refer to the *TRM*.

Datapath Instructions

The datapath is broken into the following sections:

- ALU – An ALU is capable of the following operations on 8-bit data. When multiple datapaths are tied together to form 16, 24, and 32 bits, then the operations act on the full datawidth.
  - Pass-Through
  - Increment (INC)
  - Decrement (DEC)
  - Add (ADD)
  - Subtract (SUB)
  - XOR
  - AND
  - OR
- Shift – The output of the ALU is passed to the shift operator, which is capable of the following operations. When multiple datapaths are tied together to form 16, 24, and 32 bits, then the operations act on the full datawidth.
  - Pass-Through
  - Shift Left
  - Shift Right
  - Nibble Swap
- Mask – The output of the shift operator is passed to a mask operator, which is capable of masking off any of the 8 bits of the datapath.
- Registers – The datapath has the following registers available to the hardware and to the CPU with various configuration options defined in the static configuration registers.
  - Two Accumulator Registers: ACC0 and ACC1
  - Two Data Registers: DATA0 and DATA1
  - Two 4-byte deep FIFOs: FIFO0 and FIFO1 capable of multiple modes of operation
- Comparison Operators
  - Zero Detection: Z0 and Z1 which compare ACC0 and ACC1 to zero respectively and output the binary true/false to the interconnect logic for use by the hardware as necessary.

- FF Detection: FF0 and FF1 which compare ACC0 and ACC1 to 0xFF respectively and output the binary true/false to the interconnect logic for use by the hardware as necessary.
- Compare 0:

  Compare equal (ce0) – Compare (ACC0 & Cmask0) is equal to DATA0 and output the binary true/false to the interconnect logic for use by the hardware as necessary. (Cmask0 is configurable in the static configuration.)

  Compare Less Than (cl0) – Compare (ACC0 & Cmask0) is less than DATA0 and output the binary true/false to the interconnect logic for use by the hardware as necessary. (Cmask0 is configurable in the static configuration.)
- Compare 1:

  Compare equal (ce1) – Compare ((ACC0 or ACC1) & Cmask1) is equal to (DATA1 or ACC0) and output the binary true/false to the interconnect logic for use by the hardware as necessary. (Cmask1 is configurable in the static configuration)

  Compare Less Than (cl1) – Compare (ACC0 & Cmask0) is less than DATA0 and output the binary true/false to the interconnect logic for use by the hardware as necessary. (Cmask1 is configurable in the static configuration)
- Overflow Detection: Indicates the msb has overflowed by driving ov_msb output as a binary true/false to the interconnect logic for use by the hardware as necessary.

The datapath allows for many different configurations that are common in almost every Component that will be designed. Many functions within a datapath that can be implemented with Verilog fit into the PLDs. However, the PLDs will be used up very quickly, whereas the datapath is a fixed block. There will always be a trade-off between the number of datapaths and PLDs available. It is up to the designer to decide which of these is a more precious resource. Note that some functions, such as FIFOs, cannot be implemented in the PLDs.

Datapath Registers

Each datapath contains 6 registers - A0, A1, D0, D1, F0 and F1. These serve different functions with certain restrictions, and can be used in a variety of ways to form your design.

- Accumulator registers A0 and A1 are often used like RAM to hold temporary values entering and coming out of the ALU. These are the most versatile registers and are also the most accessed.
- Data registers D0 and D1 are not as versatile as the accumulator registers. They can be written by the datapath only from the FIFO and by consuming an extra d0_load or d1_load input signal. For this reason it is often used like ROM in the design.
- 4-word deep FIFOs F0 and F1 are often used as the input and output buffers for the datapath. These cannot directly source the ALU and the value in the FIFO must be loaded in to the accumulator register before it can be used by the ALU. See FIFO Modes on page 54 for more details on the FIFO configurations.

Datapath Inputs/Output

A datapath, regardless of data width can contain up to 6 input bits. Of the 6 input bits, up to 3 bits can be used to control the datapath instructions for that clock cycle. Therefore 8 unique datapath instructions can be used in the design. Each of these instructions can perform multiple operations in the same clock cycle, which can further optimize performance.

Similarly, a datapath can contain up to 6 outputs regardless of the data width. These outputs are used to send status signals from the datapath to either a status register or to other blocks in the design such as the state machine or the count7 counter. These status signals are generated from comparisons and internal logic in the datapath and do not include data bits directly from the registers or the ALU. It is possible however to access this information by using the shifter to serially shift out the bits in the ALU.

FIFO Modes

The 4-word deep FIFOs in datapaths can be configured to several modes using an auxiliary control register. This register is used by the CPU/DMA to dynamically control the interrupt, counter, and FIFO operations. Refer to the TRM for more information about the auxiliary control register.

FIFOs are set to either single buffer or normal mode.

- **Single buffer mode –** This mode configures the FIFO to be a 1-word deep buffer instead of the normal 4-word deep FIFO. Any value written to the FIFO immediately overwrites its content. This mode can be used if only a 1-register FIFO with its corresponding FIFO bus and block status signals are needed.
- **Normal mode –** Normal mode is the standard 4-word deep FIFO that can be used to fill up to four data words.

The data transfers to and from the FIFO are often controlled using the FIFO bus and block status signals. These are FIFO 0/1 block status (f0_block_stat, f1_block_stat), and FIFO 0/1 bus status (f0_bus_stat, f1_bus_stat) signals. The behaviors of these are dependent on the input/output mode and the auxiliary control register settings. The following table shows the possible configurations.

Note This is for illustrative purposes only. For more detail descriptions on the FIFO configuration, refer to the *TRM*.

Direction	Level Mode	Signal	Status	Description
Input	N/A	Block status	Empty	Asserted when there are no bytes left in the FIFO.
	NORMAL	Bus status	Not full	Asserted when there is room for at least 1 word in the FIFO.
	MID	Bus status	At least half empty	Asserted when there is room for at least 2 words in the FIFO.
Output	N/A	Block status	Full	Asserted when the FIFO is full.
	NORMAL	Bus status	Not empty	Asserted when there is at least 1 word available to be read from the FIFO.
	MID	Bus status	At least half full	Asserted when there are at least 2 words available to be read from the FIFO.

Level mode can be configured by setting the FIFO level mode of the auxiliary control register to either NORMAL or MID.

- NORMAL FIFO level - A NORMAL FIFO level allows the bus status signal to assert whenever there is at least 1 word that is ready to be read or written (depending on the FIFO direction).
- MID FIFO level - A MID FIFO level allows the bus status signal to assert whenever there are at least 2 words that are ready to be read or written (depending on the FIFO direction).

4.3.2 Implement with UDB Editor

The UDB Editor is a graphical tool used to construct UDB-based designs without the need of writing Verilog or using the more advanced datapath configuration tool. Refer to the *PSoC Creator UDB Editor Guide* for complete instructions on using this tool.

4.3.3 Implement with Verilog

You can describe the functionality of your Component using Verilog. PSoC Creator allows you to add a Verilog file to your Component, as well as write and edit the file. Should you choose to use a Verilog implementation, be aware that there are certain requirements, and that PSoC Creator only supports the synthesizable subset of the Verilog language; refer to the *Warp Verilog Reference Guide*. You also may need to use the Datapath Configuration Tool to edit datapath instance configurations.

Note Any primitive embedded in Verilog will **not** have an API automatically generated for it. The appropriate #define values will be created, but for example a cy_psoc3_control embedded in Verilog will not cause APIs to be generated that are normally produced when a Control Register symbol is placed on a schematic.

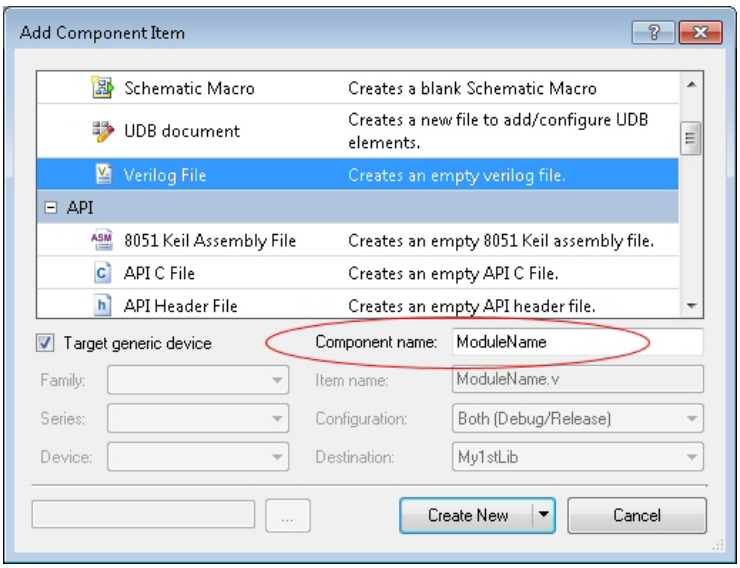
4.3.3.1 Verilog File Requirements

PSoC Creator allows for only one Verilog file per Component, and PSoC Creator only supports a synthesizable subset of the Verilog language.

The name of the Component created with a symbol in PSoC Creator must match exactly the name of the module defined in the Verilog file. PSoC Creator also creates the Verilog file with the same name as the Component. As a designer you are required to use this same name as your module name.

For example, a “ModuleName” Component created in PSoC Creator will generate a *ModuleName.v* file for you to develop the digital content. The top module in that file must be a “ModuleName” (case sensitive) for it to be built correctly in PSoC Creator.

In the following example, the text entered in the **Component name** text box (“ModuleName”) must match the module name used in the Verilog file [“module ModuleName(….)”].



You must think of the naming convention you want to follow when you create the symbol to start off the process as shown in the example above.

A file has been provided by Cypress that contains all of the constants expected in a Component Verilog file. These constants are available in the *cypress.v* file. This file is contained in the search path for all Verilog compilations in PSoC Creator. All Verilog files will use the *cypress.v* file and therefore must have this line before the module declaration:

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

4.3.3.2 *Add a Verilog File*

New File

If you do not have an existing Verilog file, you can use the **Generate Verilog** tool  , which will create a basic Verilog file with the appropriate file name, `'include "cypress.v"`, and basic module information based on how you have defined the symbol.

Existing File

If you have an existing Verilog file, you can add it as a Component item.

1. Right click on the Component and select **Add Component Item**.
   The Add Component Item dialog displays.
2. Select the Verilog icon.
   Note The Verilog file will inherit the Component name. See Verilog File Requirements on page 55.
3. If necessary, toggle the **Create New** button to **Add Existing**.
4. Click the ellipsis button [**...**], navigate to the location of the file, and click **Open**.
5. Click **Add Existing**.

The Verilog file is added to the Workspace Explorer with the appropriate file name.

4.3.3.3 *Complete the Verilog file*

Double-click on the Verilog file to open it, and complete the file as needed.

Note The Verilog implementation's module name must match its Component name; it is case sensitive.

For UDB-based PSoC devices, the Verilog implementation allows for Verilog2005 design and will compile into the PLDs available in the architecture. However. there are several other modules available which are optimized functions to prevent usage of the precious PLD resources. These modules and their use are listed in the next few sections.

4.3.4 UDB Elements

The PSoC Universal Digital Block (UDB) contains several individually programmable Components that can be accessed via Verilog when creating your design. The items contained in the UDB are the datapath, status/statusi register, control register, and count7 counter. When used in a way that requires a clock, the UDB Components can be driven by a special Verilog Component. This Component allows the customer to inform the fitter about the clock behavior to enforce for that Component in the UDB.

Note Many references in this guide and in the Component names specify “psoc3.” These references apply to all UDB-based PSoC devices, including PSoC 3 and PSoC 5LP.

4.3.4.1 *Clock/Enable Specification*

For those portions of a UDB that utilize a clock, it is possible to specify the type of clock required (synchronous vs asynchronous), as well as an enable signal for the clock. The fitter will use this information to inspect the clock, enable the type requested, and make any necessary changes to the routing/implementation of the clock within the UDB to ensure it meets the requested characteristics at the output clock.

cy_psoc3_udb_clock_enable_v1_0

This primitive allows for the specification of the clock characteristics for the UDB Components being driven by its output. The output from the element can only drive UDB elements. Any other Component will result in a DRC error in the fitter. The element has one parameter:

- sync_mode : A boolean parameter that designates whether the resulting clock should be synchronous (true) or asynchronous (false). The default is synchronous.

Instantiate the element as shown:

```
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 *Datapath(s)*

To Instantiate a datapath or multiple consecutive datapaths in your design, use one of the following module instantiations within your Verilog module.

cy_psoc3_dp

This is the base datapath element available. There are several other elements which build on this base element and they should be used before this element is chosen, because all I/Os available on this element are listed in the instantiation. However, there are several limitations because of the architecture of UDBs in UDB-based PSoC devices, such that many of these signals have a limited number of wires in your design to which they can connect. It would be wise to always use the cy_psoc3_dp8 module when a single datapath is necessary.

Each datapath has several parameters that can be passed as named parameters to the module itself. The Datapath Configuration Tool should be the method used to implement the parameters in the Verilog file for all datapaths. This tool will read a Verilog file, display all of the datapaths, and save the correct information back to the Verilog file for use by the compiler (Warp).

The parameters are:

- cy_dpconfig: the configuration for the datapath which includes the dynamic and static configurations. Default value is {128’h0,32’hFF00FFFF, 48’h0}
- d0_init: Initialization value for DATA0 Registers. Default value is 8’b0
- d1_init: Initialization value for DATA1 Registers. Default value is 8’b0
- a0_init: Initialization value for ACC0 Registers. Default value is 8’b0

- a1_init: Initialization value for ACC1 Registers. Default value is 8'b0

Instantiate this datapath as shown in the following example:

```
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

This is a single 8-bit wide datapath element.

This element has the same parameters for each datapath as the base datapath element. All parameters are appended with "_a" to indicate the LSB datapath (X=a for LSB datapath in the following list).

- cy_dpconfig_X,: the configuration for the datapath which includes the dynamic and static configurations. Default value is {128'h0,32'hFF00FFFF, 48'h0}
- d0_init_X: Initialization value for DATA0 Registers. Default value is 8'b0
- d1_init_X: Initialization value for DATA1 Registers. Default value is 8'b0
- a0_init_X: Initialization value for ACC0 Registers. Default value is 8'b0
- a1_init_X: Initialization value for ACC1 Registers. Default value is 8'b0

Instantiate this datapath as shown in the following example:

```
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           */ .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
);
```

cy_psoc3_dp16

This is two 8-bit wide datapath elements set up to be two consecutive datapaths for a 16-bit wide module. ALU, Shift, and Mask operations operate on the full 16-bit width of the data by having direct connections of the carry, shift-in, shift-out, and feedback signals between the individual datapaths.

This element has the same parameters for each datapath as the base datapath element. All parameters are appended with "_a" to indicate the LSB datapath and "_b" to indicate the MSB of the two datapaths (X=a for LSB datapath and X=b for MSB datapath in the following list).

- cy_dpconfig_X,: the configuration for the datapath which includes the dynamic and static configurations. Default value is {128'h0,32'hFF00FFFF, 48'h0}

- d0_init_X: Initialization value for DATA0 Registers. Default value is 8'b0
- d1_init_X: Initialization value for DATA1 Registers. Default value is 8'b0
- a0_init_X: Initialization value for ACC0 Registers. Default value is 8'b0
- a1_init_X: Initialization value for ACC1 Registers. Default value is 8'b0

Instantiate this datapath as shown in the following example:

```
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

This is three 8-bit wide datapath elements set up to be three consecutive datapaths for a 24-bit wide module. ALU, Shift, and Mask operations operate on the full 24-bit width of the data by having direct connections of the carry, shift-in, shift-out, and feedback signals between the individual datapaths.

This element has the same parameters for each datapath as the base datapath element. All parameters are appended with "_a" to indicate the LSB datapath, "_b" to indicate the middle datapath, and "_c" to indicate the MSB datapath of the three datapaths (X=a for LSB datapath, X=b for the middle datapath, and X=c for MSB datapath in the following list).

- cy_dpconfig_X,: the configuration for the datapath which includes the dynamic and static configurations. Default value is {128'h0,32'hFF00FFFF, 48'h0}
- d0_init_X: Initialization value for DATA0 Registers. Default value is 8'b0
- d1_init_X: Initialization value for DATA1 Registers. Default value is 8'b0
- a0_init_X: Initialization value for ACC0 Registers. Default value is 8'b0
- a1_init_X: Initialization value for ACC1 Registers. Default value is 8'b0

Instantiate this datapath as shown in the following example:

```
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

This element is four 8-bit wide datapath elements set up to be four consecutive datapaths for a 32-bit wide module. ALU, Shift, and Mask operations operate on the full 32-bit width of the data by having direct connections of the carry, shift-in, shift-out, and feedback signals between the individual datapaths.

This element has the same parameters for each datapath as the base datapath element. All parameters are appended with “_a” to indicate the LSB datapath, “_b” to indicate the lower middle datapath, “_c” to indicate the upper middle datapath, and “_d” to indicate the MSB datapath of the four datapaths (X=a for LSB datapath, X=b for the lower middle datapath, X=c for the upper middle datapath, and X=d for MSB datapath in the following list).

- cy_dpconfig_X,: the configuration for the datapath which includes the dynamic and static configurations. Default value is {128’h0,32’hFF00FFFF, 48’h0}
- d0_init_X: Initialization value for DATA0 Registers. Default value is 8’b0
- d1_init_X: Initialization value for DATA1 Registers. Default value is 8’b0
- a0_init_X: Initialization value for ACC0 Registers. Default value is 8’b0
- a1_init_X: Initialization value for ACC1 Registers. Default value is 8’b0

Instantiate this datapath as shown in the following example:

```
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 *Control Register*

A control register is writable by the CPU. Each of its 8 bits are available in the interconnect routing to control PLD operations or datapath functionality. Multiple control registers may be defined within a design but they will act independently.

To instantiate a control register in your design, use the following element instantiation within your Verilog code.

cy_psoc3_control

This 8-bit control register has the following available parameters, as follows:

- cy_force_order: A Boolean used by the compiler to improve ability of the router if order of the bits within the register is not required. The default value is False. Typically the order is important and this should be set to TRUE.
- cy_init_value: The initial value for the register to be loaded during chip configuration.
- cy_ctrl_mode_1, cy_ctrl_mode_0 (PSoC 3 ES3 Only): These two parameters are optional. Together they control which of the three modes of operation are to be used for each bit of the control register. Refer to the TRM for details about each of the modes:

cy_ctrl_mode_1	cy_ctrl_mode_0	Description
0	0	Direct mode (default)
0	1	Sync mode
1	0	Reserved
1	1	Pulse mode

Instantiate this control register as shown in the following examples:

Without Optional Mode Parameter:

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

With Optional Mode Parameter:

```
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 *Status Register*

A status register is readable by the CPU. Each of its 8 bits are available in the interconnect routing to provide status from the PLDs or datapaths. An adaptation of the 8-bit status register is also implemented, which allows for 7 bits of status and the 8th bit is used as an interrupt source by OR'ing together the masked output of the other 7 bits. This is shown in the cy_psoc3_statusi module. Multiple status registers may be defined within a design but they will act independently.

To instantiate a status register in your design, use one of the following module instantiations within your Verilog code.

cy_psoc3_status

This is an 8-bit status register. The element has the following available parameters. These should be passed as named parameters as shown in the example instantiation:

- cy_force_order: A Boolean used by the compiler to improve ability of the router if order of the bits within the register is not required. The default value is False. Typically the order is important and this should be set to TRUE.
- cy_md_select: A mode definition for each of the bits in the register. The bits represent transparent or sticky. The default value is transparent for each bit.

Instantiate this status register as shown in the following example:

```
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

This module is a 7-bit status register with an interrupt output based on those 7 status bits.

For the statusi register, there is a hardware enable and a software enable. Both enables must be enabled to generate an interrupt. The software enable is in the Aux Control register. Note that multiple bits in the Aux Control register can be for different Components, so an interrupt safe implementation should be done using Critical Region APIs. See the following for an example

```
#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);
```

The module has the following parameters which should be passed as named parameters as shown in the example instantiation:

- cy_force_order: A Boolean used by the compiler to improve ability of the router if order of the bits within the register is not required. The default value is False. Typically the order is important and this should be set to TRUE.
- cy_md_select: A mode definition for each of the bits in the register. The bits represent transparent or sticky. The default value is transparent for each bit.
- cy_int_mask: A mask register to select which bits are included in the generation of any of the 7 bits of the register. The default value is 0, which disables all 7 bits as interrupt generators.

Instantiate this status register as shown in the following example:

```
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

A simple 7-bit counter is available to avoid using up a whole 8-bit datapath or multiple PLDs to implement the same. This element uses other resources within the architecture to avoid using up the more precious PLD and datapaths. If your counter is between 3 and 7 bits then this module will save PLD or datapath resources. An example of where this counter is useful is as a bit counter for communication interfaces where you need a 4-bit counter which would tie up one whole datapath or one whole 4-macrocell PLD.

For the count7 counter, there is a hardware enable and a software enable. Both enables must be enabled to get the count7 counter to count. The software enable is in the Aux Control register. Note that multiple bits in the Aux Control register can be for different Components, so an interrupt safe implementation should be done using Critical Region APIs. See the following for an example:

```
#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

This element has the following available parameters which should be passed as named parameters as shown in the example instantiation:

- cy_period: A 7-bit period value. Default value is 7'b1111111.
- cy_route_ld: A Boolean to enable a routed signal as a load signal to the counter. If false terminal count reloads the counter with the period value passed as a parameter, if true either terminal count or the load signal will load the counter. Default is `FALSE.
- cy_route_en: A Boolean to enable a routed signal as an enable to the counter. If false then the counter is always enabled, if true then the counter is only enabled while the enable input is high. Default is `FALSE.

Instantiate this counter as shown in the following example:

```
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 Fixed Blocks

For all fixed blocks, a parameter of cy_registers is available to allow you to explicitly set the values for the given registers for the block. The values listed will be included in the configuration bitstream generated to program the part, which will be established before the main() function is called. The value for the parameter is a string which is expected to have a format of:

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

Where reg_name is the name of a register in the block as listed in the TRM, such as the CR1 register of the DSM. The value after the = is expected to be the hexadecimal value to be assigned to the register (the leading 0x is optional, but the value is always interpreted as hexadecimal). In the case where a register listed is one that is also configured by the fitter, the value listed in the parameter will take precedence.

4.3.6 Design-Wide Resources

Some Components are considered part of design-wide resources and are not included in Verilog files, such as:

- Interrupt Controller
- DMA Request

4.3.7 When to use Cypress Provided Primitives instead of Logic

- Register that needs to be accessed from the CPU
- Most ALU operations especially if it is wider than 4 bits
- Shifting operations especially if it is wider than 4 bits
- Masking operations especially if it is wider than 4 bits
- Counter operations especially if it is wider than 4 bits

4.3.8 Warp Features for Component Creation

PSoC Creator Components may use Verilog to define the digital hardware of the system. This Verilog supports the standards used in Verilog 2001.

4.3.8.1 Generate Statements

The ability to use generate statements is key to Component development in UDB-based PSoC devices because you cannot use the `ifdef statement to work with parameterized values in your Verilog file. For example, you may want to remove a datapath if the parameter set by the user requests an 8-bit wide data bus. This can be handled with a conditional generate statement. Warp supports Conditional and For-loop generate statements which can surround Verilog code.

Note It is important to remember scope when trying to define nets inside of a generate statement.

Note Generate statements add a layer to the naming convention passed to the API (in the generated *cyfitter.h* file) from the datapaths and control and status registers. If you use a generate statement with a “begin : GenerateSlice” statement then the Components are now pushed down one level in their naming. For example a datapath named “DatapathName” that is not inside of a generate statement will have the following variables defined in the *cyfitter.h* file.

```
#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
```

But if that same datapath is inside of the “GenerateSlice” generate statement the same variables will be defined in cyfitter.h as

```
#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 Generate Example

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

Conditional Generate Example

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

Parameter Usage

Parameters are allowed in the right hand side of a parameter assignment. This is critical for Component development particularly when assigning the datapath configuration information. For example, the SPI Component requires the ability to set the number of data-bits dynamically at build time. The MSB value bit-field of the datapath’s configuration requires a different value based on the data-width of the SPI transfer. A parameter called MSBVal can be defined which is assigned based on the existing parameter NumBits which defines the transfer size of the block. The method to do this is provided in the datapath configuration tool. For datapath configuration you should not be hand-editing the parameters passed to a datapath. But you may use this method in any other parameter definitions at your discretion.

localparam Usage and Named Parameters

Parameters are widely used in Components because of the configurable nature of them in PSoC Creator. It is important to protect your parameters so they are not accidentally overwritten with unexpected values. For this to work there are two features in the PSoC Creator version of Warp. These are the support of localparams and passing named parameters.

Defparam usage is very error prone so Warp has added the support of named parameter passing. This is used extensively as shown in the standard modules defined in previous sections. Named parameters are passed at the instance declaration time within the #(…) block as shown in the following example:

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

You can and should also protect your parameters with local parameters as much as possible. In the example above for SPI, MSBVal should be defined as a local parameter because it is set based on the parameter NumBits. It cannot be set by the user at a higher level. The code for this example would look like the following:

```
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 Implement with Software

Implementing with software is simply omitting the references for hardware. You would not use a schematic or Verilog; just software files. Everything else in the process would be the same as any other type of Component.

One example of a software only Component might be an interface that links the code of several Components.

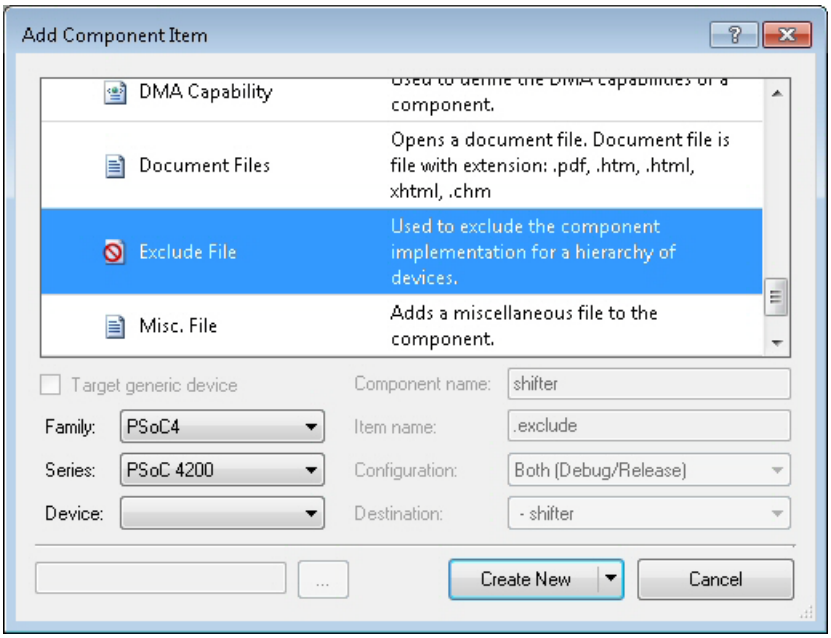
4.5 Exclude a Component

Along with creating implementations for a specific family, series, or device, you can exclude the Component as well. Excluding a Component means it will not be shown in the catalog or available for use for a specified family, series, or device.

Note A specific implementation will override the exclude file. For example, if you exclude an entire family, but create an implementation for a specific series or device, the Component will be available for that family or device.

To exclude a Component, add an exclude file to your Component as follows:

1. Right-click on the Component and select **Add Component Item**.
   The Add Component Item dialog displays.



2. Scroll down to the **Misc** section and select the Exclude File icon.
3. De-select the **Target generic device** to specify the Family, Series, and/or Device using the drop down menus, or leave the check box selected to allow the Component to apply to all devices.
4. Click **Create New**.

In the Workspace Explorer, the exclude is listed in the appropriate directory, depending on the **Target** option specified.

When you finish this Component and try to use it in a design, select the appropriate family, series, and/or device for which you added the exclude file, and verify that the Component is not available.

5. Simulating the Hardware

Cypress provides functional simulation models with PSoC Creator for Component developers to simulate and debug their designs. This chapter provides a how-to-guide to help you understand how CPU accesses (reads and writes) to various elements should be done.

5.1 Simulation Environment

Synthesis of Verilog source designs is made possible with Warp. Although Cypress does not at this time provide a Verilog simulator, some system files necessary for third party simulators to provide pre-synthesis simulations are available. Warp uses two source files to accomplish the synthesis of Verilog designs: *lpm.v* and *rtl.v*. These files are located in the following directory:

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

In this chapter, `$CYPRESS_DIR` is `$INSTALL_DIR/warp/`.

The *lpm.v* file is used to define certain values for the parameters used by the Library of Parameterized Modules (LPM) in the library. The *rtl.v* file is used to define various primitives used by the synthesis engine to map to the internal architecture of the targeted devices.

Also included in this directory is the cypress.v file, which links the design to the appropriate PSoC models. When using these models for synthesis, the following lines must be present in the user's source design:

```
`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
```

For Verilog simulators, use the `+incdir+` switch. For example:

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

or

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

The latter example is valid since the Verilog source files are also mirrored at `$CYPRESS_DIR/lib/sim/presynth/vlg` for convenience and compatibility with the Verilog pre-synthesis file structure.

Typically, the invocation of a Verilog simulator that utilizes these collections of modules would be:

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

You must use the appropriate `` `include `` clause to access the desired modules.

5.2 Model Location

The simulation models distributed with PSoC Creator (via Warp) are located in the Warp portion of the directory structure, located in:

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

5.3 Test Bench Development

In the simulation environment, a test bench must provide for several functions performed by the CPU in the physical device. To simulate that functionality, the test bench must contain blocks that drive the internal signals of the device models.

The device model contains elements that must be clocked, written, and/or read. Provisions were made in the device models to help with the manipulation of the elements by providing internal signals (CPU clock) and tasks (read/write) associated with the various elements. The following sections discuss this functionality and provide several examples.

5.3.1 Providing the CPU Clock

Although the models for the status register, control register, and datapath do not have a pin that is connected to the CPU clock, it is still necessary to supply such a clock to simulate what happens via the CPU in the silicon itself. In order to provide for CPU accesses to these Components, an internal `reg` has been included in each model. The models affected are the control register (cy_psoc3_control), both types of status register (cy_psoc3_status and cy_psoc3_statusi), and the datapath (cy_psoc3_dp). In each of these Components, the signal is named `cpu_clock`.

5.3.1.1 CPU Clock Example

In this example, there are several levels of hierarchy, but the bottom level (pattern_matcher) is an instantiation of a cy_psoc3_dp. The `cpu_clock` reg in that model is therefore manipulated via the test bench by this initial clause.

```
// 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
```

If a design contains more than one control register, status register, or datapath, the clock generator block should contain several assignment statements (such as the following examples).

```
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 Register Access Tasks

Access to the internal registers of the three mentioned Components by the CPU must also be modeled. To accomplish this, there have been several `task` functions added to each Component model. These tasks are detailed in the following list:

```
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;
```

These tasks are used to read and write the various registers in their respective Component models. Each call to any one of these tasks involves a single argument.

- For the write tasks, that argument is the 8-bit data that is to be written to the register.
- For the read tasks, the 8-bit register data will be retuned and assigned to that argument.

Typical calls to these tasks take the form:

```
// 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);
```

The following sections show usage examples:

5.3.2.1 FIFO Write

```
// 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 Read

```
// 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 Register Read

```
// 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 Register Write

```
// 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 Status Read

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

5.3.2.6 Control Write

```
// 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. Adding API Files

This chapter covers the process of adding application programming interface (API) files and code to a Component, as well as different aspects such as API generation and template expansion.

6.1 API Overview

API files define the Component interface, which provides the various functions and parameters for a Component. As a Component author, you create API file templates for the Component that become instance-specific for an end-user.

6.1.1 API generation

API generation is the process by which PSoC Creator creates instance-based code, including the generation of symbolic constants used to define hardware addresses. The hardware addresses are a result of placement (one of the steps in the fitter, which is part of the build process).

6.1.2 File Naming

All of the files in the API directory are templates expanded into instance_name-qualified files for compilation. For example, let the Counter API files be *Counter.c, Counter.h,* and *CounterINT.c*.

If there is a top-level Counter Component called foo, then files *foo_Counter.c, foo_Counter.h,* and *foo_CounterINT.c* are generated. If there is a lower-level Counter instance called bar_1_foo, then files *bar_1_foo_Counter.c, bar_1_foo_Counter.h,* and *bar_1_foo_CounterINT.c* are generated.

6.1.3 API Template Expansion

6.1.3.1 Parameters

When creating API template code, you can use any built-in, formal, and local parameter (including user-defined parameters) for template expansion using either of the following syntax:

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

Either syntax is valid, and both expand to provide the specified parameter value. For example:

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

For a Counter instance named foo_1, this code snippet would expand to:

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

6.1.3.2 User-Defined Types

The following constructs define the mappings for user defined types.

`'#DECLARE_ENUM type_name'` – declare the key names defined for the specified typename
`'#DECLARE_ENUM_ALL'` – declare the key names for all types used by the Component

Note A type_name might be defined locally (PARITY) or remotely (UART__PARITY).

Each of those directives results in a sequence of #define statements. For types local to the instance, the expanded name takes the form:

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

For borrowed types, the expanded name takes the form:

```
<path_to_instance>_<Componentname>__<keyname>
```

Note that the instance name is followed by a single underbar (as in the rest of the API), but the Component name, if used, is followed by a DOUBLE UNDERBAR (as it is in the rest of the system). This means that locally defined key values should not conflict with register names.

Example

```
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)
```

A design contains a schematic that contains one instance of Fox named bob. In the API file for bob, the following lines:

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

expand to:

```
#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
```

In the API file for bob, the following line:

```
'#declare_enum_all'
```

expands to:

```
#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 Conditional API Generation

In the specific case of control register, status register, statusi register, or interrupt, if the mapping of the design to the part determines that the Component is not being used (based on connectivity of the Component), the mapping process will remove the Component from the mapped design. To inform the API generation system that the code for the Component is no longer needed, an entry of:

```
#define `$INSTANCE_NAME`__REMOVED 1
```

will be generated in *cyfitter.h* to allow the Component API to be conditionally compiled into the design. In addition, any code that uses the APIs of the removed Component would need to be constructed to take advantage of the #define so the removed Component would not be accessed.

6.1.5 Verilog Hierarchy Subsitution

If you are implementing your design using Verilog, you may wish to access various fitter constants for control registers. PSoC Creator provides a `` `[...]` `` substitution mechanism for hierarchy path generation for you to access fitter constants. For example:

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

This code snippet expands to:

```
foo_1_uart_x_ctrl_ADDRESS
```

6.1.6 Macro Callbacks

Macro callbacks are specified by Component authors as extension points into Component APIs where users can (safely) inject new code. The general form should be:

```
#ifndef NAME_OF_MACROCALLBACK
    NAME_OF_MACROCALLBACK
#endif /* NAME_OF_MACROCALLBACK */
```

For more information, refer to the PSoC Creator Help.

To use the macro callback, it needs to be defined in a user-defined header file named c*yapiallbacks.h*. This file will be included in all generated source files that offer callbacks through the local built-in parameter named:

```
CY_API_CALLBACK_HEADER_INCLUDE
```

See Locals: on page 16 for more information.

Components should not directly #include *cyapicallbacks.h*. Instead they should do the following:

```
`=GetApiCallbackHeaderInclude()`
```

This ensures backward compatibility with older PSoC Creator projects (that don't have *cyapicallbacks.h)*, and ensures the firmware still builds if the user chooses to remove the *cyapicallbacks.h* header file.

6.1.6.1 Multiple Callbacks

It is perfectly acceptable to define multiple callbacks with different argument lists so that users can implement the one most appropriate to their needs. This requires the Component author to consider what variables to allow the user to "see" and/or "modify" (by reference).

6.1.6.2 User Code

A callback requires the user to complete the following:

- Define a macro to signal the presence of a callback (in *cyapiallbacks.h*).
- Write the function declaration (in *cyapiallbacks.h*).
- Write the function implementation (in any user file).

To complete the example, the *cyapiallbacks.h* file would include this code:

```
#define SimpleComp_1_START_CALLBACK
void SimpleComp_1_Start_Callback( void );
```

In any other user file, the user would write the SimpleComp_1_Start_Callback() function.

6.1.6.3 Inlining

The function declaration could be added to the Component source, which would save the user from having to write the prototype (one line per function). However, by reducing the Component code to just the call, the user is free to modify the declaration. This is most obviously beneficial with inlining.

Different compilers implement inlining with different syntaxes but they all (that we support) do it on the function declaration/implementation, and not the call. By extracting the declaration/implementation from the content code the user has full control over inlining, regardless of the compiler used.

6.1.7 Optional Merge Region

A merge region defines an area where end-users can write their own code that will be preserved during future updates of Components and source code. Merge regions are optional. The preferred method to preserve user code is to use Macro Callbacks.

You define the region using the following syntax:

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

Note The merge region identifier <region name> must be unique.

Anything an end-user places in this region will be preserved in subsequent updates of the file. If a subsequent version of the file does not contain the same named region, the entire region from the previous file will be copied to the end of the file and placed in comments.

6.1.8 API Cases

You may specify both an API and a schematic for the Component, or only one of them (only a Component of type “primitive” may have no schematic and no API). The following lists the possible cases of APIs that may – or may not – exist:

- A schematic is provided but no API exists

  In this case, the Component is nothing but a hierarchical combination of other Components that needs to be flattened when instantiated in a design.

  Note that when the user edits the design-entry document for the generic portion of the Component project, no family-specific primitives can be placed into the schematic. Likewise, when the

user edits the design-entry document for a family, only the primitives that are applicable for that particular family are available for placement into a schematic.

- An API exists but no schematic is provided

  In this case, the Component does not re-use existing Components and presents an API for use. This may occur for software-only Components that do not employ other Components.

- Both a schematic and an API are provided.

  This is a typical Component with an API and one or more primitives and/or other Components included in the design. When the schematic includes other instantiated Components that have their own APIs, the code generation process for a top-level design is somewhat complicated. For example, consider construction of a Count32 Component from two Count16 Components, with instance names u1 and u2 in the Count32 generic schematic. When the Count32 Component is instantiated in a top-level design, the code generation mechanism must be aware of the hierarchy when generating the APIs.

  In this case, the creator of the Count32 Component would need to use u1's API in the template code as `'$INSTANCE_NAME``[u1]`Init()`.

  Note that if the Count16 Component was composed of Count8 Components, for example, with identical substructure, then their instance names during code generation are hierarchical in nature as well.

- Neither a schematic nor an API is provided.

  This is an invalid configuration except for Components of type "primitive."

6.2 Add API Files to a Component

Use PSoC Creator to add as many API files as required for the Component:

1. Right click on the Component and select **Add Component Item**.

   The Add Component Item dialog displays.
2. Select the icon for the Component item you wish to add.
3. De-select the **Target generic device** to specify the Family, Series, and/or Device using the drop down menus, or leave the check box selected to allow the Component to apply to all devices.
4. Type in the **Item name**.
5. Click **Create New**.

   The Component item displays in the Workspace Explorer. Depending on the target options you specified, the Component item may be located in a subdirectory.
6. Repeat the process to add more API files.

6.3 Complete the .c file

The .c file contains the function and parameter definitions for your Component. It should include a reference to the basic Component header file you add to the Component, as shown in the following example:

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

6.4 Complete the .h file

The .h file generally contains the function prototypes for the Component. The header file should include references to the *cyfitter.h* and *cytypes.h* files, which are generated as part of the build. The *cyfitter.h* file defines all of the instance specific addresses calculated by the Code Generation step. The *cytypes.h* file provides macros and defines to allow code to be written tool chain and processor (PSoC 3/ PSoC 5) agnostic.

The following is an example of how to include those files:

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

When the header is generated, there are a few naming conventions used. For working registers (status, control, period, mask, etc.) the full instance name will be used (including any hierarchy) with a specified string appended (as shown in the following table):

Working Register	Name (Prepend <Instance name>_<GenerateSliceName> for each)
Status	_<StatusRegComponentName>__STATUS_REG
Status Auxiliary Control	_<StatusAuxCtrlRegComponentName>__STATUS_AUX_CTL_REG
Control	_<ControlRegComponentName>__CONTROL_REG
Control Auxiliary Control	_<ControlAuxCtrlRegComponentName>__CONTROL_AUX_CTL_REG
Mask	_<MaskRegComponentName>__MASK_REG
Period	_<PeriodRegComponentName>__PERIOD_REG
Accumulator 0	_<A0RegComponentName>__A0_REG
Accumulator 1	_<A1RegComponentName>__A1_REG
Data 0	_<D0RegComponentName>__D0_REG
Data 1	_<D1RegComponentName>__D1_REG
FIFO 0	_<F0RegComponentName>__F0_REG
FIFO 1	_<F1RegComponentName>__F1_REG
Datapath Auxiliary Control	_<DPAuxCtlRegComponentName>__DP_AUX_CTL_REG

7. Finishing the Component

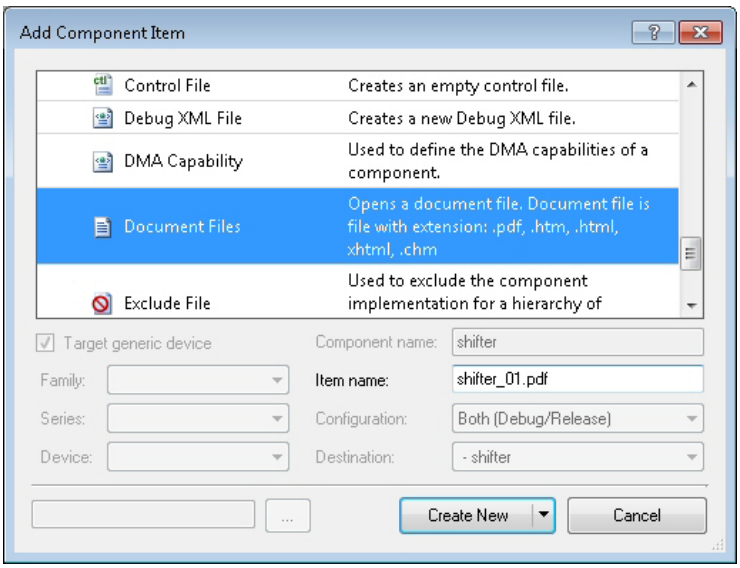
This chapter covers the various steps involved with finishing the Component, including:

- Add/Create Datasheet
- Add Control File
- Add/Create Debug XML File
- Add/Create DMA Capability File
- Add/Create .cystate XML File
- Add Static Library
- Add Dependency
- Build the project

7.1 Add/Create Datasheet

You can add documentation to your Component in any of PDF, HTML, and XML formats. However, a Component datasheet must be a PDF file. When a user opens the datasheet from the tool, it will always open the PDF file associated with the Component. You can create Component datasheets using MS Word, FrameMaker, or any other type of word processing program. Then, you can convert that file into PDF, and add it to the Component. The main requirement is that a PDF file must have the same name as the Component in order to be viewed from the Component Catalog.

1. Right-click on the Component and select **Add Component Item**.
   The Add Component Item dialog displays.



2. Select the **Document Files** icon under the **Misc** category from the templates, and type the name of the PDF file to add in **Item name**.

   Note This process applies to any number of miscellaneous files you wish to add to the Component. Click on a different template icon to add a different file type.
3. Click **Create New** to allow PSoC Creator to create a dummy file; select **Add Existing** from the pull-down to select an existing file to add to the Component.

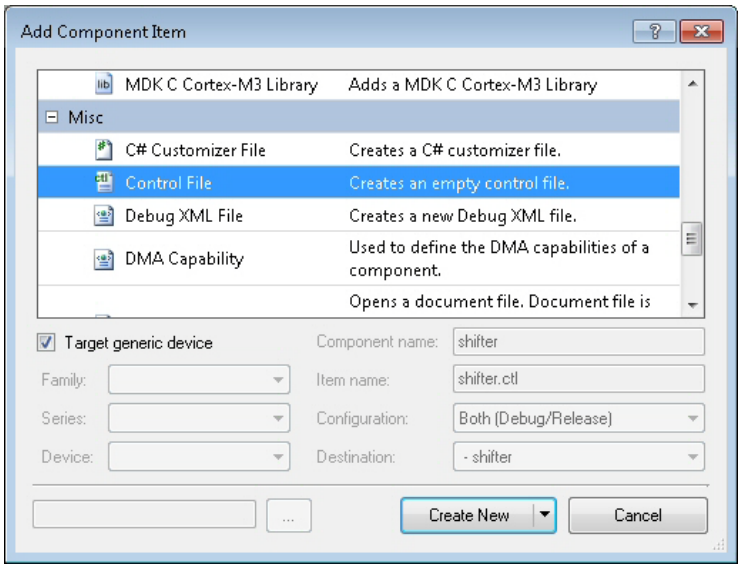
   The item displays in the Workspace Explorer and opens outside of PSoC Creator.
4. If applicable, copy any updates over the file included with the project.

7.2 Add Control File

A control file is used to define fixed placement characteristics for a Component instance. It allows a Component author to define the target datapath, PLD and status/control register resources. For more information about control files, refer to the PSoC Creator Help, under in the “Building a PSoC Creator Project” section. To add a control file:

1. Right-click on the Component and select **Add Component Item**.

   The Add Component Item dialog displays.



2. Select the **Control File** icon under the **Misc** category from the templates. The name will be the same as the Component.
3. Click **Create New** to allow PSoC Creator to create an empty file; select **Add Existing** from the pull-down to select an existing file to add to the Component.

   The item displays in the Workspace Explorer and opens as tabbed document in the work area.

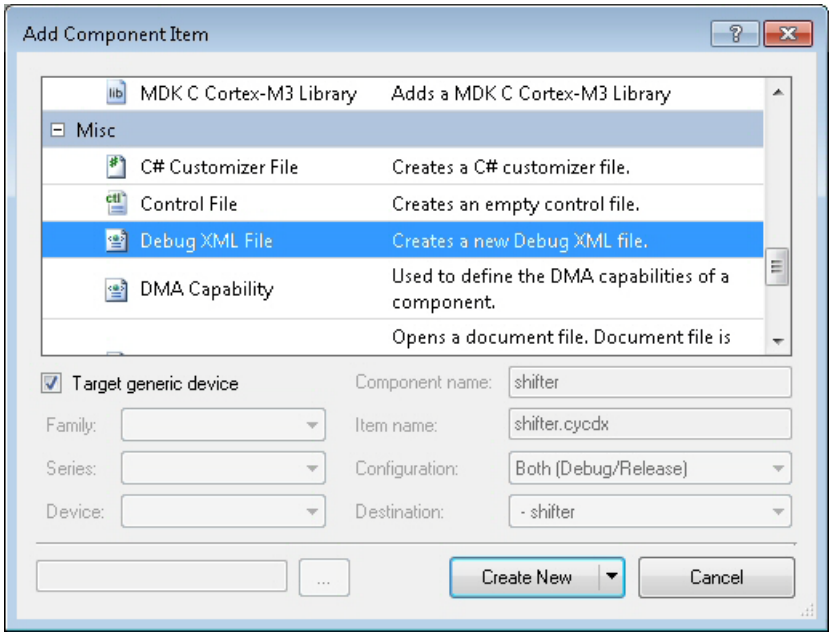
When the user enables the feature from the Component Configure dialog, the Component instance will be forced into the specified resources. Multiple instances are not allowed to use this feature and attempts to do so will generate an error.

7.3 Add/Create Debug XML File

PSoC Creator provides an optional mechanism for creating new debugger tool windows for any Component. To enable this feature on a specific Component, add an XML description of the Component. This information can consist of any blocks of memory used or individual registers that are important to the Component. This debug file is then used in the PSoC Creator Component Debug window. To learn how to use the Component Debug window, refer to the PSoC Creator Help.

1. Right-click on the Component and select **Add Component Item**.

   The Add Component Item dialog displays.
2. Select the **Debug XML File** icon under the **Misc** category from the templates; the name becomes the same as the Component, in this case *Component_A.cycdx*.



3. Click **Add Existing** to select an existing file to add to the Component; select **Create New** to allow PSoC Creator to create a dummy file.

The item displays in the Workspace Explorer and opens as a tabbed document in the PSoC Creator Code Editor.

7.3.1 XML Format

The following are the major elements and attributes that make up the debug XML files. While the items listed here should cover everything useful in creating a debug file, the full schema definition can be found in the following directory of your PSoC Creator installation:

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

Like other files in the Component, debug XML files are able to use template expansion (see Section 6.1.3) to evaluate parameters at build time. Additionally, addresses can be either the actual address, or a #define from *cydevice.h*, *cydevice_trm.h*, or *cyfitter.h*.

Note The names of items in the XML documents cannot contain spaces.

Block

The <block> field is used to wrap related registers/memories in a single block. A block can be a Component instance, a portion of the Component such as UDBs, sub-Components, or an abstract collection of similar registers or memories. A block must always have a unique name within a hierarchy. It can optionally provide a description and a block visibility expression.

Parameter	Type	Description
name	string	Defines the block name. This string will be appended to all sub items.
desc	string	Provides a description of the block.
visible	string	Specifies whether the contents of the block are displayed (string evaluates to bool). Applies to descendants.
hidden	string	Specifies whether the contents of the block are hidden (string evaluates to bool). Applies to the block itself.

Memory

The <memory> field represents a contiguous section of memory used by the Component. The memory section needs a unique name, the starting address of the memory, and the size of that memory. A <memory> must be placed within a <block>.

Parameter	Type	Description
name	string	Defines the name of the section of memory.
desc	string	Provides a description of the memory section.
BASE	string	The starting address of the memory section (string evaluates to uint).
SIZE	string	Number of bytes in the memory section (string evaluates to uint).
hidden	string	Specifies whether the contents of the memory section are hidden (string evaluates to bool).

Register

The <register> field is used to represent a register used internally by the Component. A <register> must be placed within a <block>.

Parameter	Type	Description
name	string	Defines the name of the register.
desc	string	Provides a description of the register.
address	string	The address of the register (string evaluates to uint).
size	string	Number of bits in the register (string evaluates to uint).
hidden	--	Specifies whether the contents of the block are displayed (string evaluate to bool)

Field

The <field> is used to define a portion of a register that has a specific meaning. It is a sub-item of <register> and should not be used outside of it. The definition allows meaning to be placed for the specific portions of the register when viewed from the Component debug window.

Parameter	Type	Description
name	string	Defines the name of the field.
desc	string	Provides a description of the field.
from	string	The high order bit for the field (string evaluate to uint).
to	string	The low order bit for the field (string evaluate to uint).
access	string	Defines the type of access available for the register such as read-only (“R”), write-only (“W”), or read and write (“RW”). Possible values are: R - Read only W - Write only RW - Read/Write RCLR - Read to clear RCLRW - Read to clear or write RWCLR - Read or write to clear RWOSET - Read or write once to set RWZCLR - Read or write zero to clear
hidden	string	Specifies whether the contents of the field section are hidden (string evaluates to bool).

Value

The <value> is used to define a specific value that has meaning to a register field. It is a sub-item of <field>. A value is an optional helper definition that shows up in the tool tip when a <register> with a <field> and a <value> is opened in the Component debug window. The definition allows quick viewing of what values of the specific field mean.

Parameter	Type	Description
name	string	Defines the name of the value.
desc	string	Provides a description of the value.
value	string	Specifies the value in binary string, such as “100”, “0101”, “00”.

7.3.2 Example XML File

The following is an example debug XML file. Note that all elements must be placed within a <block> and should follow the restrictions outlined above. The example contains a top level block, which is the Component instance in the project. It then shows either the UDB implementation or the Fixed Function implementation of the Component by using the `$FixedFunction` Component parameter.

Note Addresses shown in this example are not actual addresses; they are used here only as a demonstration.

```
<?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>
```

For additional examples, look at the Components shipped with PSoC Creator (e.g., I^2C). These are located in the following directory of your PSoC Creator installation:

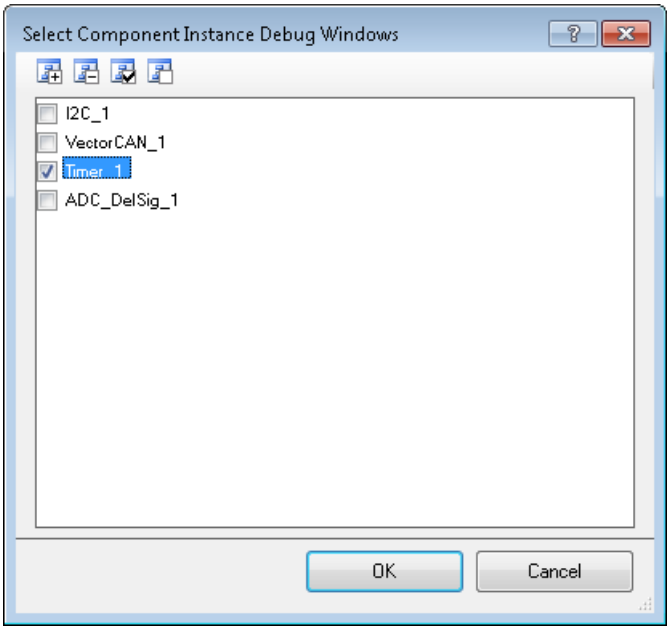
<install location>\PSoC Creator\<Version#>\PSoC Creator\psoc\content

7.3.3 Example Windows

The following sections show the different Debug windows that can be generated via the XML files. For each window shown, the relevant section of the XML file is also shown to indicate what is needed to get it. All examples here are from the Timer_v2_50 Component.

7.3.3.1 Select Component Instance Debug Window

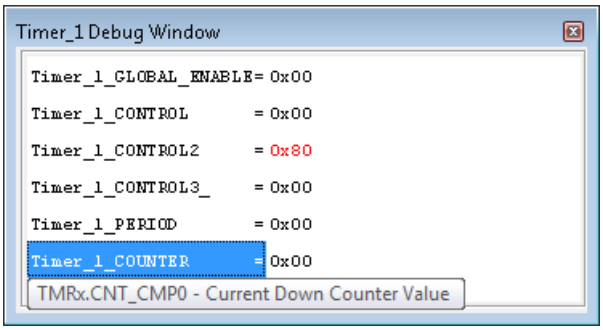
This window is where the user selects the debug window for the specific instance.




```
<?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>
```

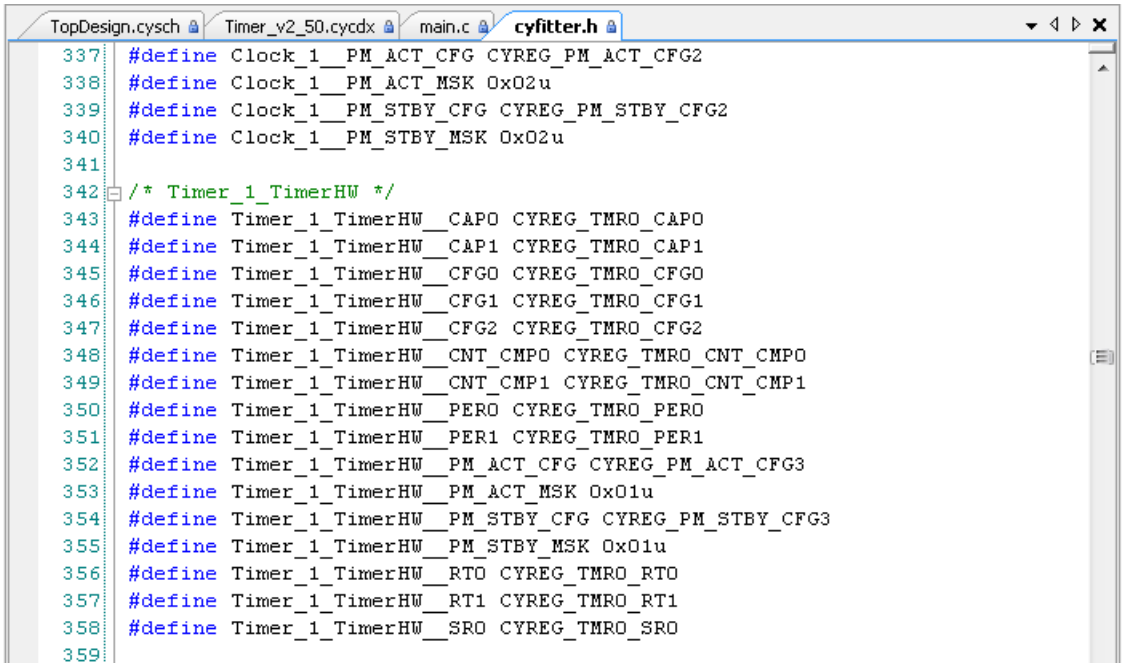
7.3.3.2 *Component Instance Debug Window*

This window shows the specific instance debug information.



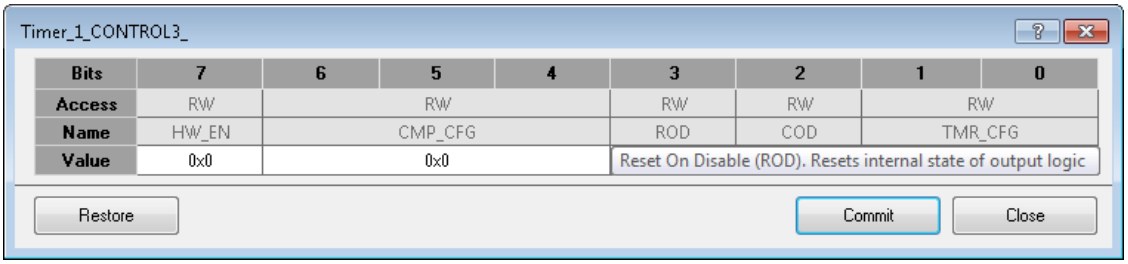

```
<?xml version="1.0" encoding="us-ascii"?>

<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>
```

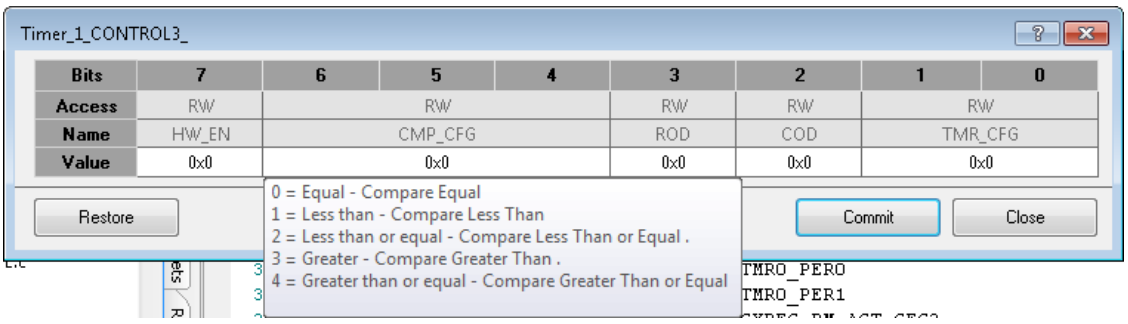
For the address field, the Component uses the Cypress template substitution mechanism to allow this to work for any Component name, as well as the ability to look up any value defined in *cydevice_trm.h* or *cyfitter.h*. In this case, it used a define from *cyfitter.h* which allows it to function independent of where the tool ended up placing the timer.



7.3.4 Registers Window

This window shows selected register details.





```
…
    <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>
…
```

7.4 Add/Create DMA Capability File

The DMA capability file allows your Component to use the PSoC Creator DMA wizard. The DMA Wizard simplifies configuring DMA for moving data quickly and reliably between sources and destinations.

7.4.1 Adding a DMA Capability File to a Component:

1. Right-click on the Component and select **Add Component Item**.

   The Add Component Item dialog displays.
2. Select the **DMA Capability** icon under the **Misc** category from the templates; the name becomes the same as the Component.
3. Click **Add Existing** to select an existing file to add to the Component; select **Create New** to allow PSoC Creator to create a dummy file.

   The item displays in the Workspace Explorer and opens as a tabbed document in the PSoC Creator Code Editor.

7.4.2 Editing Component Header File:

Generate human-readable source and destination addresses in the Component header file. This step is optional, but it enhances the user experience by providing logical names for memory locations. For example, FIFO 0 stores the result from a Component that calculates something. You could use the generic “<Component>_long_name_F0_REG” define provided in the *cyfitter.h* file, or you could rename it to something more useful, such as: “MyComponent_Result_PTR”. The user will see MyComponent_Result_PTR in the DMA wizard, and it will make using the tool easier.

If you have registers that are part of the hardware, you can find generic register definitions in the *cyfitter.h* file. These definitions will always be updated with the proper address regardless of where your Component pieces are placed by PSoC Creator. To see what you have available, build a test project with your Component in it, then open the generated *cyfitter.h* file. The following is an example:

```
/* 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
…
```

You can use the definition shown in red in your header file to redefine the abstract `X_UDB_OffsetMix_1_OffsetMixer_LSB__16BIT_F0_REG` into something more human friendly. For example:

Header File:

```
#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)
```

You now have a define `` `$INSTANCE_NAME_OUTPUT_PTR` `` to use anywhere; not only for the DMA Wizard, but for API and Component usage in general.

7.4.3 Completing the DMA Capability File:

The DMA capability file includes a large block of comments at the beginning to help understand the file.

Note Comments in the DMA Capability file are surrounded by '<!--' and '-->'. For example:

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

The beginning of the DMA Capability file looks like the following:

```
<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>
```

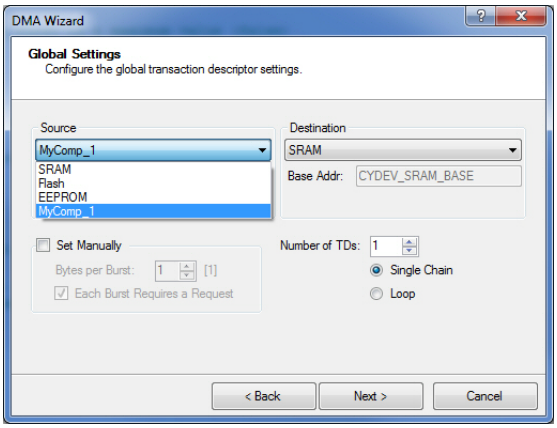
The following sections describe how each field impacts the DMA Wizard:

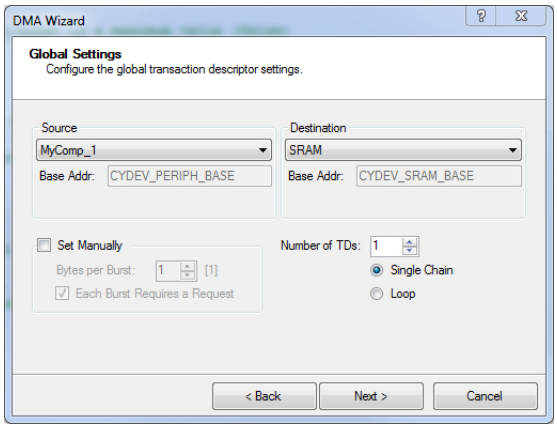
7.4.3.1 Category Name

You can have one or more categories.

- If you have one category, the DMA Wizard will only show your Component name when selecting source and destination.

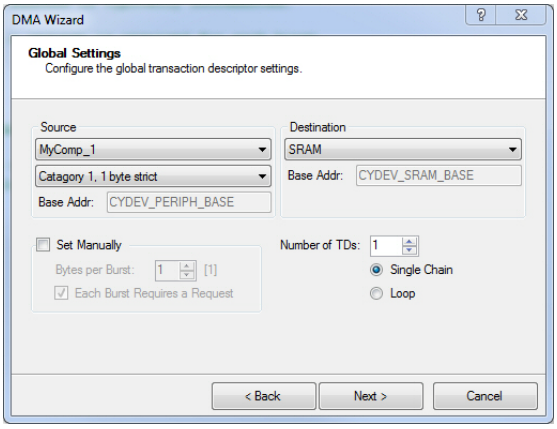
```
<DMACapability>
  <Category name="Catagory 1, 1 byte strict"
        enabled="true"
        bytes_in_burst="1"
        bytes_in_burst_is_strict="true"
        spoke_width="2"
        inc_addr="false"
        each_burst_req_request="true">
    <Location name="`$INSTANCE_NAME`_location_1_cat_1" enabled="true"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

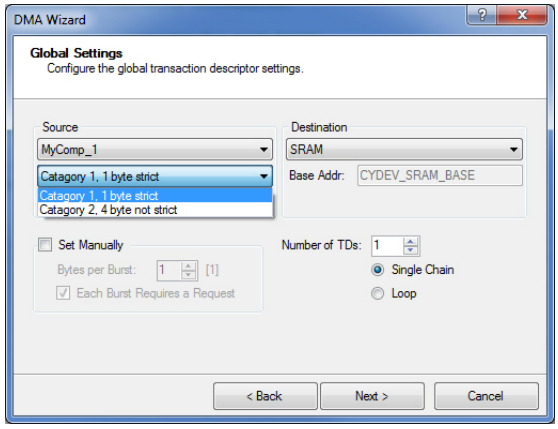




- If you have multiple categories, when your Component is selected in the source and destination, another drop-down menu will appear listing the available categories.

```
<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>
```

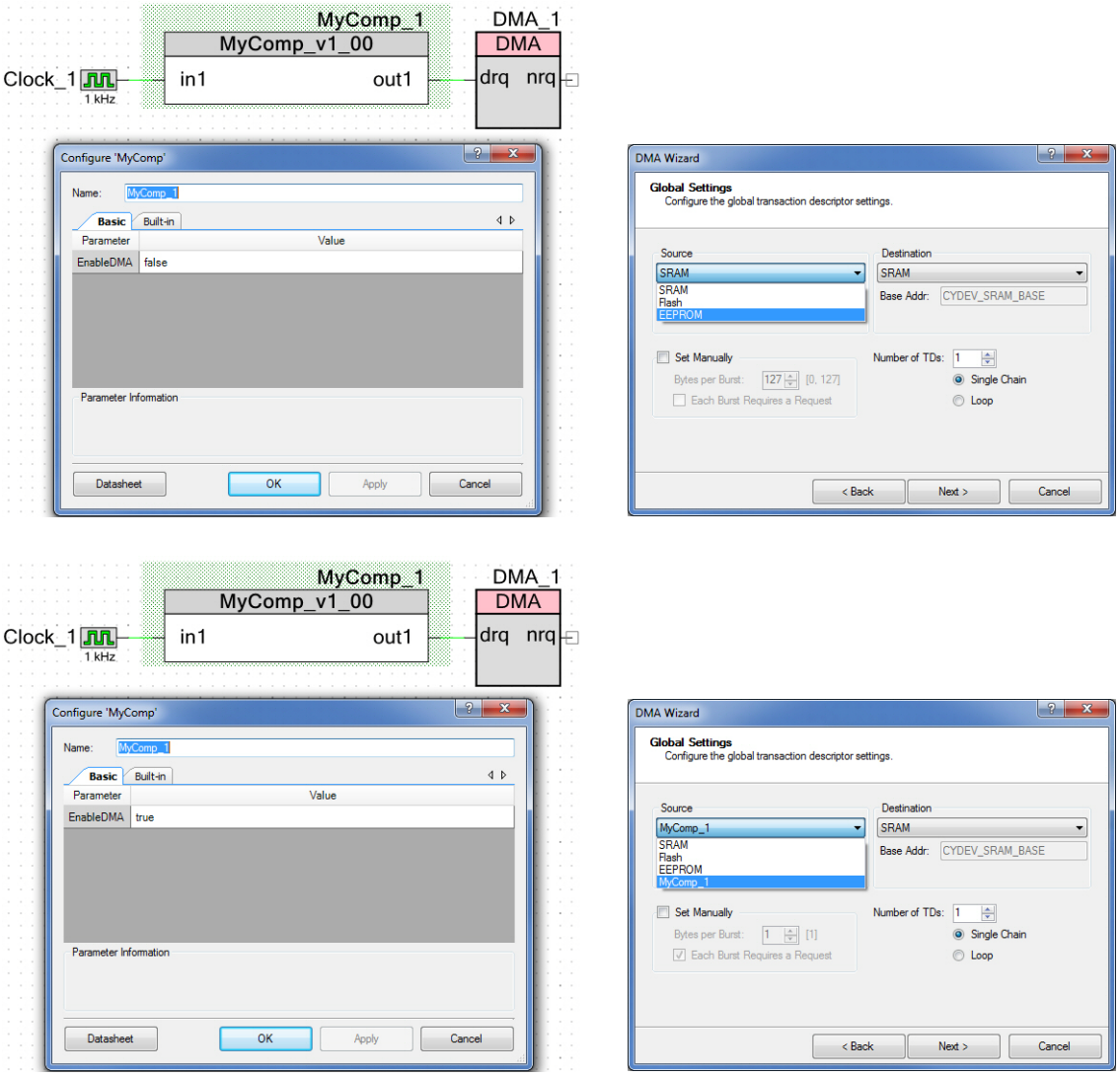




7.4.3.2 *Enabled*

The enable field can be set to “true” or “false”, or it can use a parameter passed from the customizer. This can be used to enable or disable categories depending on features enabled or disabled in the customizer. The syntax is such that you replace the content between the quotation marks with `=$YourParameterName` [include the back ticks (`)].

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



Note Any field in the DMA Capability file can be replaced with a parameter using the same syntax. You can also use Boolean expressions in these fields to produce more complex results. For example:

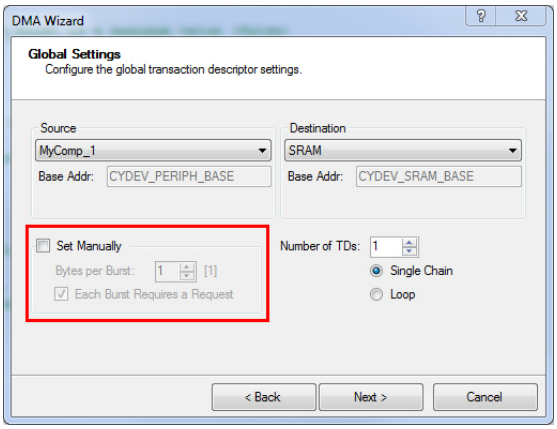
```
enabled="`=$EnableExtraDest && $EnableExtraLocations`"
```

7.4.3.3 *Bytes In Burst*

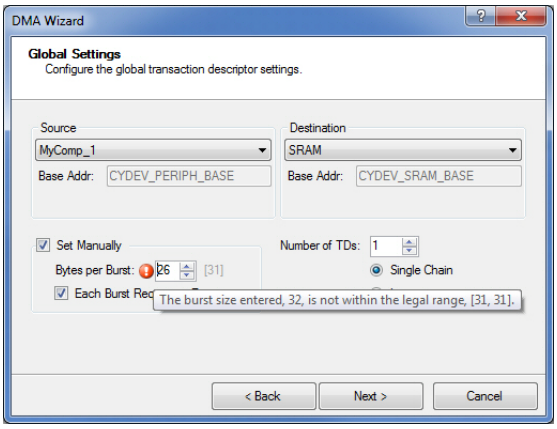
The bytes_in_burst parameter sets the initial value of the “Bytes per Burst” box in the DMA Wizard and can be from 0 to 127. This can either be a “Strict requirement” (the bytes per bust *must* be the value specified) or a maximum number of bytes per burst.

```
<DMACapability>
  <Category name="Catagory 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>
```



The user can override this setting by checking the **Set Manually** box. However, the tool will display an error if the value is over the maximum or go outside the “Strict” value:



This setting can also be a parameter from a customizer. Use a type of “uint8” with the customizer validator set to restrict the range from “0” to “127”.

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

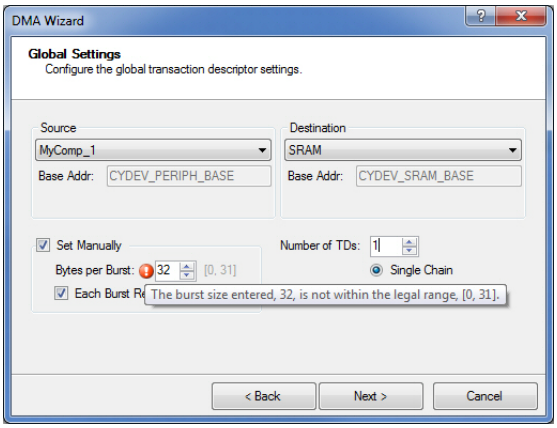
7.4.3.4 *Bytes in Burst is Strict*

When “bytes_in_burst_is_strict” is set to “true”, this field specifies if the value “bytes_in_burst” is a required value that should not deviate from the value specified. When set to “false” the value of “bytes_in_burst” is used to set to set an upper limit on the bytes per burst.

The user can always select **Set Manually** in the DMA wizard and override these settings, regardless of the value of the strict setting. However, there will be a warning.

```
<DMACapability>
  <Category name="Catagory 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>
```



7.4.3.5 *Spoke Width*

The “spoke_width” field is the spoke width (in the number of bytes) that the source/destination register resides on. The recommendation is to set to the spoke width associated with the source/destination registers. The spoke widths can be found in the *TRM* under the heading PHUB and DMAC. The following table is copied from the TRM:

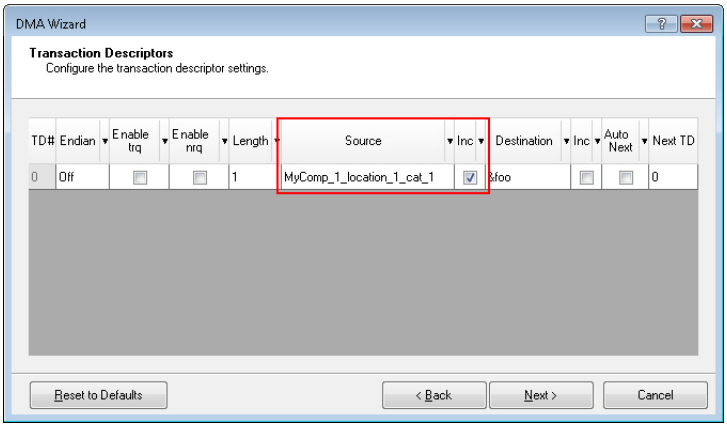
Spoke	Address Width (in bits)	Data Width (in bits)	Peripheral Names
0	14	32	SRAM
1	9	16	I/O interface, port interrupt control unit (PICU), external memory interface (EMIF)
2	19	32	PHUB local spoke, power management, clock, serial wire viewer (SWV), EEPROM
3	11	16	Delta-sigma ADC, analog interface
4	10	16	USB, CAN, fixed-function I^2C, 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="Catagory 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>
```

7.4.3.6 *Inc Addr*

The “inc_addr” field sets the initial check box state of the “increment source address” or “increment destination address” on the second page of the DMA Wizard (Transaction Descriptor or TD configuration).

```
<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`_location_1_cat_1" enabled="true"
    direction="source"/>
  </Category>
</DMACapability>
```

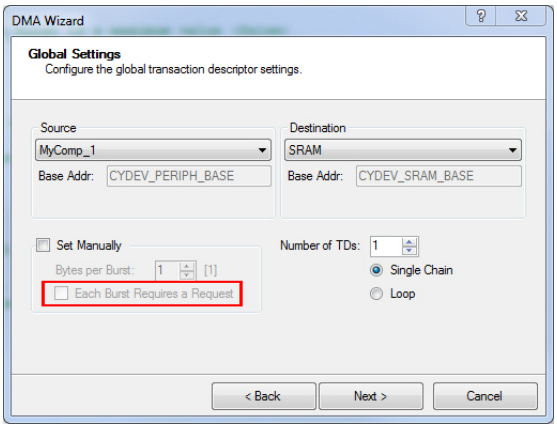


If the “direction” field of the location entry (see Location Name on page 98) is set to “destination” then when the “inc_addr” field is set to true, the **Destination Inc** check box will be checked instead of the **Source Inc** check box.

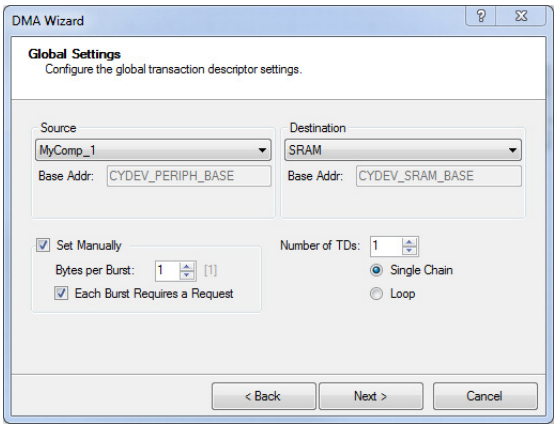
7.4.3.7 *Each Burst Requires A Request*

The “each_burst_requires_a_request” field specifies the initial value of the **Each Burst Requires a Request** check box in the DMA Wizard under the burst setting field.

```
<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="false">
    <Location name="`$INSTANCE_NAME`_location_1_cat_1" enabled="true"
    direction="source"/>
  </Category>
</DMACapability>
```



As with other fields, the user can check the **Set Manually** check box to override the initial setting, regardless of the value of the “each_burst_requires_a_request” field.



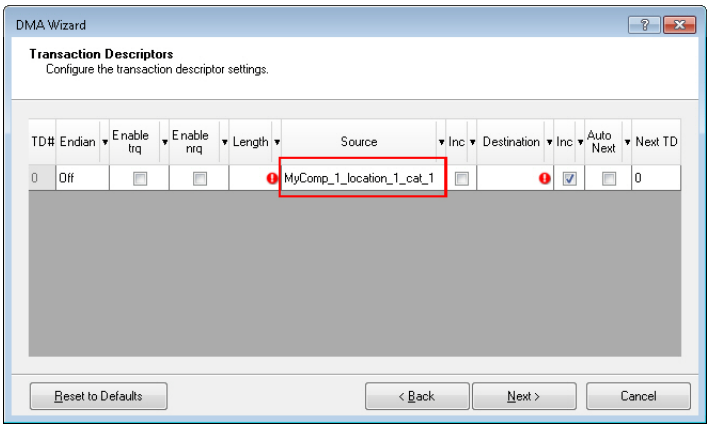
7.4.3.8 *Location Name*

The “Location” fields are the actual addresses that will be used in the TD configuration window. You may have one or more “Location” field in each category.

Note The syntax for the register name is slightly different than for a parameter.

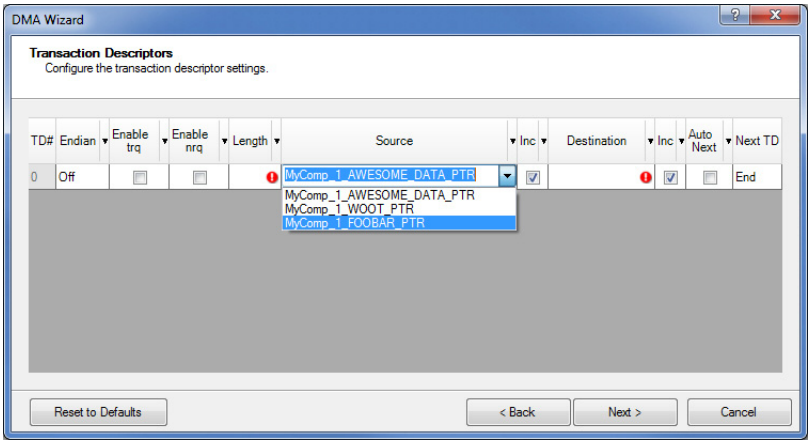
- If you only have one location in a category, then this location will automatically be populated in the appropriate source/destination location in the TD configuration window.

```
<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`_location_1_cat_1" enabled="true"
    direction="source"/>
  </Category>
</DMACapability>
```



- If you have multiple locations in the category, you will be given a choice from a drop down box containing all the listed locations.

```
<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="`$INSTANCE_NAME`_AWESOME_DATA_PTR" enabled="true"
    direction="source"/>
    <Location name="`$INSTANCE_NAME`_WOOT_PTR" enabled="true" direction="source"/>
    <Location name="`$INSTANCE_NAME`_FOOBAR_PTR" enabled="true" direction="source"/>
  </Category>
</DMACapability>
```



Enabled

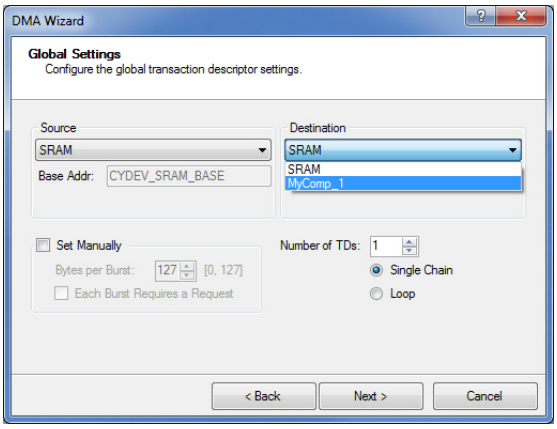
If the location is enabled by setting this field to “true”, then the entry will appear in the TD configuration window. If the location is set to “false” then the location will not show up in the list of options. This can be used to selectively enable / disable options based on parameters set in the customizer. The syntax is the same for the other fields:

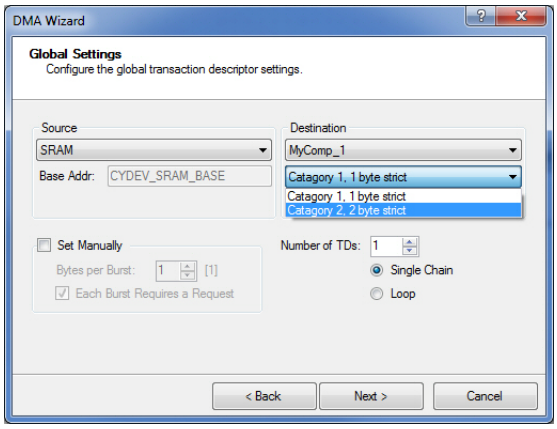
```
enabled="`=$EnableLocationAwesome`"
```

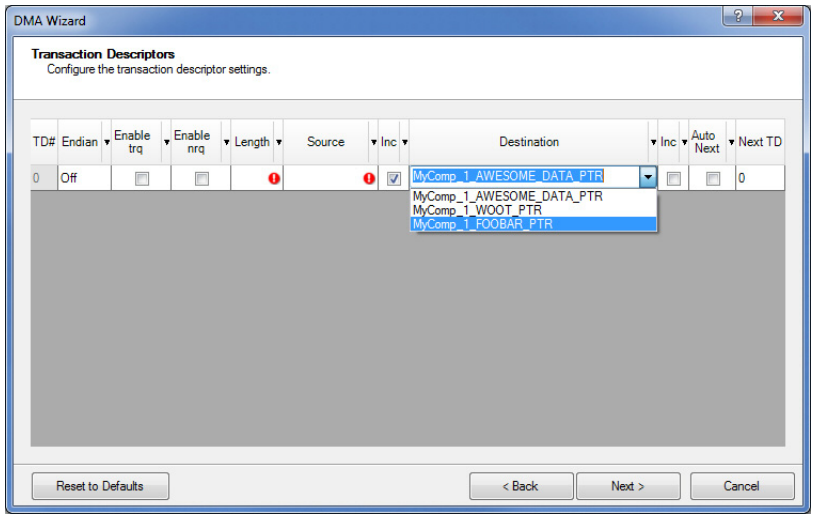
Direction

The “direction” field sets the location as “source”, “destination” or “both”. This setting impacts where the category will appear in the DMA wizard, as well as where the location and increment address will go.

```
<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>
```







7.4.4 Example DMA Capability File:

```
<!--
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="`=$Your Expression here`".
bytes_in_burst: Integer between 1 and 127. The number of bytes that can be sent/received in a single burst.
bytes_in_burst_is_strict: "true" or "false". Determines whether the bytes_in_burst is a maximum value (false)
or a specific value that must be used (true).
spoke_width: Integer between 1 and 4. The spoke width in bytes.
inc_addr: "true" or "false". Specifies whether or not the address is typically incremented.
each_busrt_req_request: "true" or "false". Specifies whether or not a request is required for each burst.
Location Attributes
===================
name: The name of the location to display to the user in the DMA Wizard.
enabled: [OPTIONAL] "true" or "false". If not provided it defaults to true. If false, this
location is not included in the DMA Wizard. Note: this value can be set to an expression
referencing parameters by using: enabled="`=$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>
```

7.5 Add/Create .cystate XML File

By adding a .cystate XML file, you can provide Component state information to your Components. This optional feature is used to generate DRC errors, warnings, and notes if a Component is not of production quality, or if the Component is used with incompatible versions of silicon. If you do not include a .cystate file, and/or if you do not use entries for a targeted device, then no DRCs will be generated.

Note Any changes you make to the .cystate file won't take effect until PSoC Creator has been restarted.

7.5.1 Adding the .cystate File to a Component

1. Right-click on the Component and select **Add Component Item**.

   The Add Component Item dialog displays.
2. Select the **Misc. File** icon under the **Misc** category from the templates.
3. For the name, enter the same name as the Component with a .cystate extension (for example, *cy_clock_v1_50.cystate*).
4. Click **Create New** to create an empty file; select **Add Existing** to select an existing file to add to the Component.

   The item displays in the Workspace Explorer and opens as a tabbed document in the PSoC Creator Code Editor.

7.5.2 States

In general, Components can be in one of three states: obsolete, production, or prototype.

- **Obsolete** defines a Component as not recommended for use.
- **Production** defines a Component as fully-tested and warranted for use in production designs.
- **Prototype** refers to everything else. There are two common uses: beta releases and example content.

7.5.3 State Messaging

The .cystate file supports the display of messages in the Notice List window with options for the type and the message.

7.5.3.1 Notice Type

The type of message can be one of the following:

- **Error** terminates a build
- **Warning** is a serious message, but does not stop the build
- **Note** is a relatively low-risk informational message.

7.5.3.2 Default Message

PSoC Creator has default messages for Prototype and Obsolete states. The Component can override these defaults by providing its own string.

- For a **Prototype** Component, unless overridden, the tool reports:

  The COMPONENT_NAME (INSTANCE_NAME) is a Prototype Component. It has not been tested to production quality and care should be taken if used in a shipping product.
- For an **Obsolete** Component, unless overridden, the tool reports:

  The COMPONENT_NAME (INSTANCE_NAME) is an Obsolete Component. It is no longer recommended for use and should be replaced with a production version of the Component (or an alternative implementation).

7.5.4 Best Practices

The following is the policy used for Cypress Components, and what Cypress recommends be used for all Components.

- **Production** Components should generate no messages from the tool.
- **Obsolete** Components should generate either warnings or errors.

  The intention is to aggressively remove obsolete Components them from user designs. When making a Component obsolete, use a warning in the first release. Then upgrade the message to an error in the next. This ensures you do not break older designs without warning.\

  In most cases it is recommended that the default message be overridden to direct the user to a better solution (that is, direct them to a specific new version or a new Component).
- **Prototype** Components should generate either notes or warnings.

  Unless there are known defects, a Prototype Component should issue a note with the default message. If a defect is found, then elevate the message to a warning and override the default message to explain the problem.

7.5.5 XML Format

The following is the schema for creating the *<project>.cystate* file.

```
<?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>
```

The following are the major elements of the .cystate file.

ComponentStateRules

ComponentStateRules is the root element that contains one or more rules for the Component. This element has a required attribute named “Version” and the only legal value is 1. It has one element, named Rule.

Rule

The Rule element defines a rule for the Component. Rules are listed in order of precedent. Once a rule is matched, the result is returned. Each Rule supports one set of the following elements:

- **Pattern –** This is the name of the pattern to match. The value for this element is specified by PatternType. It can be any of family, series, device, and/or revision. The pattern is not hierarchical, and it can be blank to search for everything.
- **State –** This is the name of the state. The value for this element is specified by StateType. It can be one of Production, Prototype, or Obsolete.
- **Severity –** This is the type of message given by the Component for this particular rule. The value for this element is specified by SeverityType. It can be one of:

- None: That is satisfactory or safe
- Note: Informational
- Warning: Attention is needed
- Error: Danger or Not valid

- **Message –** This is the actual message string. The “Message” element is optional and if it is not specified, then the tool displays the default message.

7.5.6 Example *<project>.cystate* File

The following example shows a *<project>.cystate* file that specifies the Component to be a prototype for a PSoC 5 device. If the device is PSoC 3 ES2, the Component will be obsolete, and the user will get an error to update to ES3. Also, if the device is PSoC 3 ES1, the Component will be obsolete with the default message.

```
<?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>
```

7.6 Add Static Library

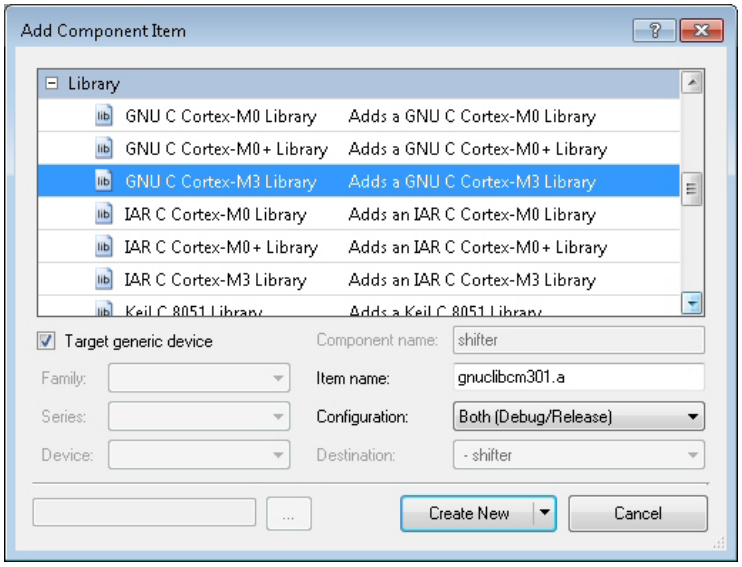
A Component may need to ship with pre-compiled libraries for the software stack. Different libraries may be added to an implementation of a Component for different compiler tool-chains and configurations (DEBUG/RELEASE).

A couple of reasons to include libraries might be:

- IP Protection – For communications Components, in particular, it is often desirable to "hide" the implementation of the software stack. Enabling static libraries in Components will allow developers to use the standard PSoC Creator Component development flow (i.e., not shipping a library as an extra piece) without having to provide source code for the whole software package. CapSense is a good candidate to take advantage of this feature.
- Build Performance – Components that require a lot of software can slow down the build performance of PSoC Creator, often re-building unchanged code many times. By shipping the non-volatile source code in a library, Component authors can avoid slowing down builds of projects that use their content.

To add a library to a Component:

1. Right-click on the Component in the Workspace Explorer and select **Add Component Item**.
   The Add Component Item dialog displays.



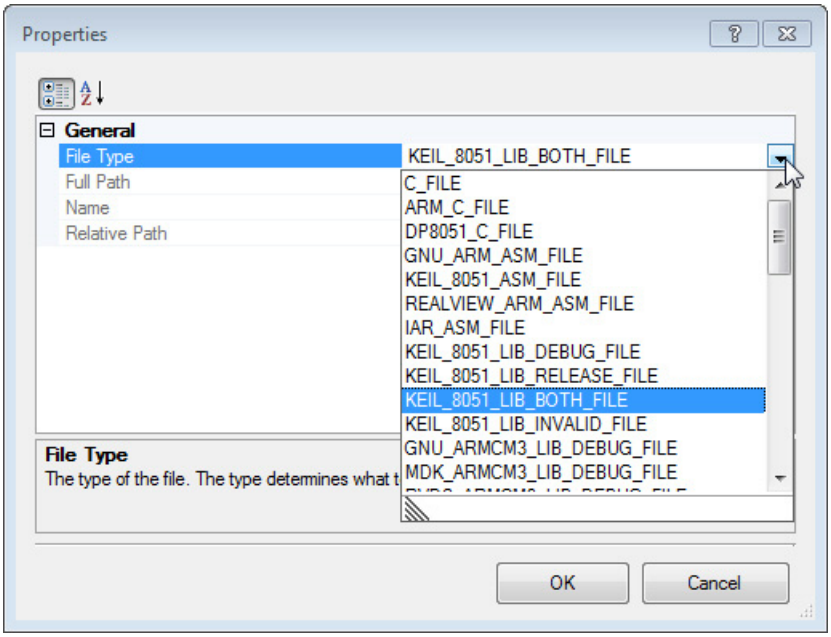
2. Select the appropriate library file type for the desired toolchain under the **Library** category from the templates. The **Item name** will default to something appropriate for the type selected; it can be changed as needed. See Best Practices on page 107.
3. When a Library item is selected, the **Configuration** field becomes active to select the desired configuration, as follows:
   - Debug
   - Release
   - Both
4. Click **Create New** to allow PSoC Creator to create the file, or select **Add Existing** from the pull-down to select an existing file.
   The item is added in the Workspace Explorer in the “Library” directory.

7.6.1 Best Practices

Component authors should ensure that no two Components with static libraries have code dependencies between those static libraries. That is, Comp1 should not have code in its API or Lib that uses symbols defined by the static library of Comp2.

Best practices for using static libraries in a Component is to create a Component that will have only the static library(ies) and add that Component to the schematic of Components using the static library. This will allow Component authors to update the top level Component without having to update the static library or vice versa. Additionally, Component authors should insure that all static libraries they add to a Component are added at the Family level. This insures that the Component author is adding CM-0 code to CM-0 based devices, CM-3 to CM-3, etc.

If during library creation, the Component author selects the wrong template (say GCC instead of MDK), the Component author can go to their Component, select the library with the wrong template and right-click, from there the author selects the properties item from the context menu. In the Properties window, use the **File Type** property to select the appropriate template for the library.



7.7 Add Dependency

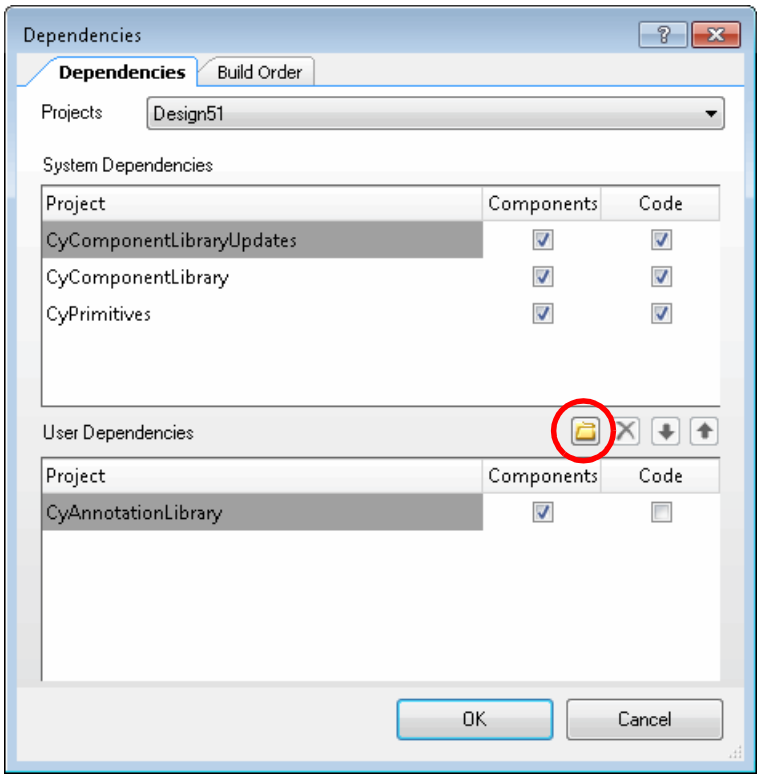
PSoC Creator relies on different System Dependencies that contain the Cypress-provided Components available in the Component Catalog. Once you have completed all the Components in your project, you can add it as a User Dependency or Default Dependency.

7.7.1 Add a User Dependency

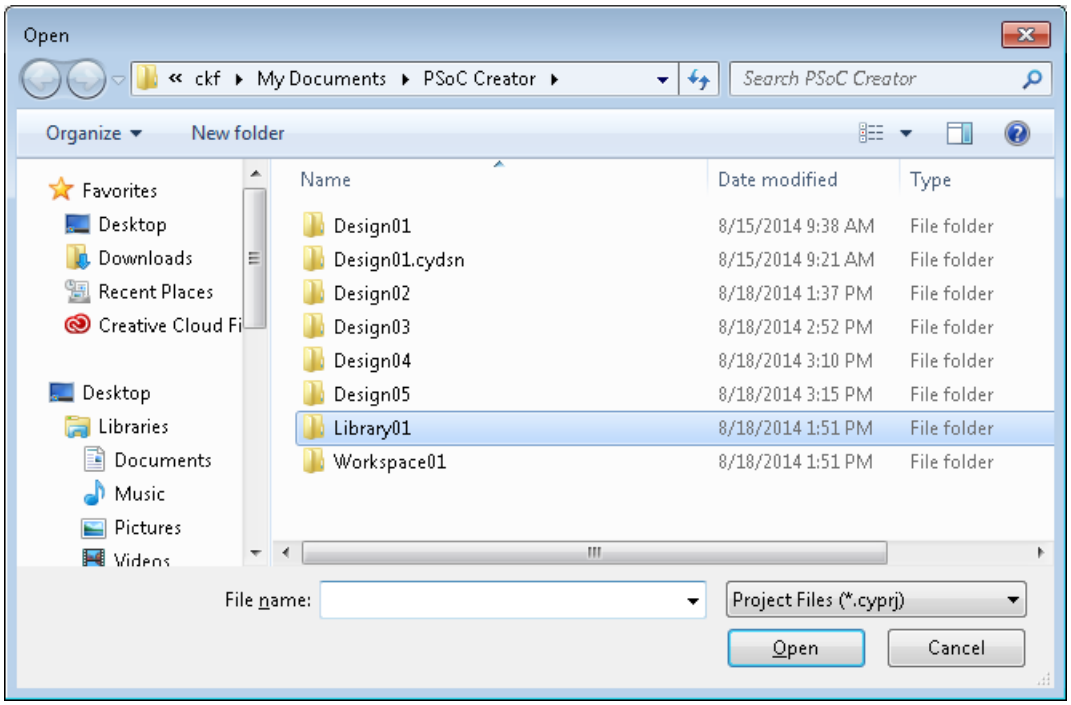
A User Dependency is used in a specific project.

1. Open or create a design project open in PSoC Creator.
2. Select **Project > Dependencies**.

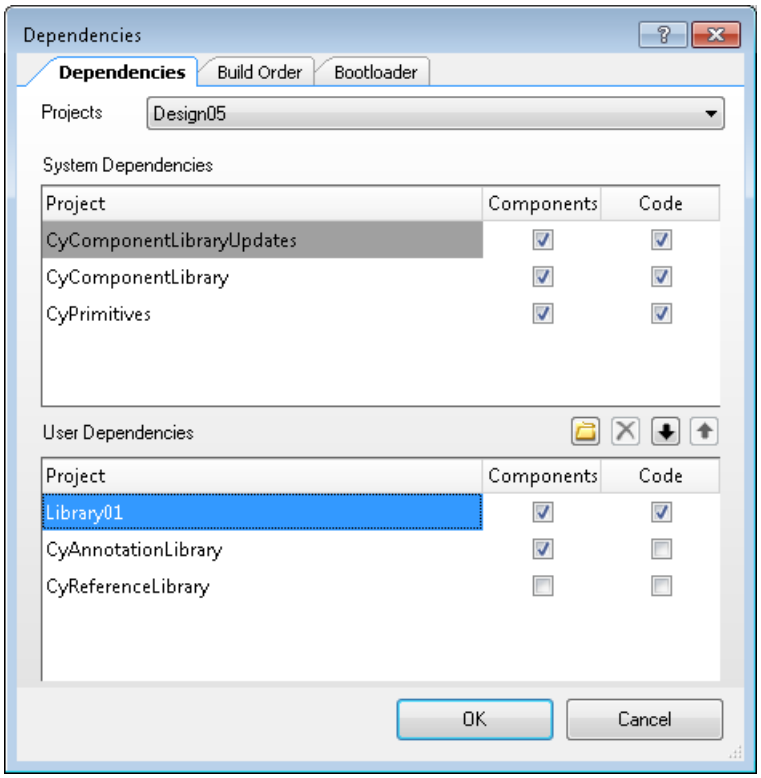
3. On the Dependencies dialog, under **User Dependencies**, click the **New Entry** button.



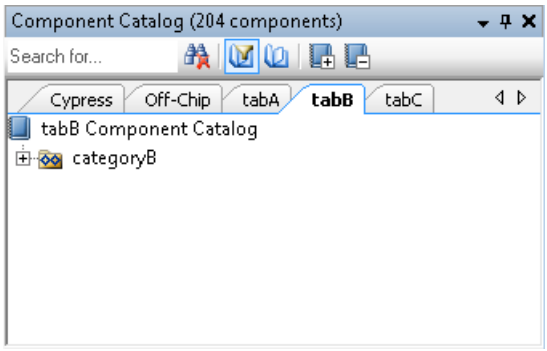
4. On the Open dialog, navigate to the location of the project, select it, and click **Open** to close the dialog.



5. On the Dependencies dialog, notice that the project you selected is listed under **User Dependencies**. Click **OK** to close the Dependencies dialog.



6. On the Component Catalog, notice that there may be one or more new tabs and categories of Components, depending on how the Components were configured.

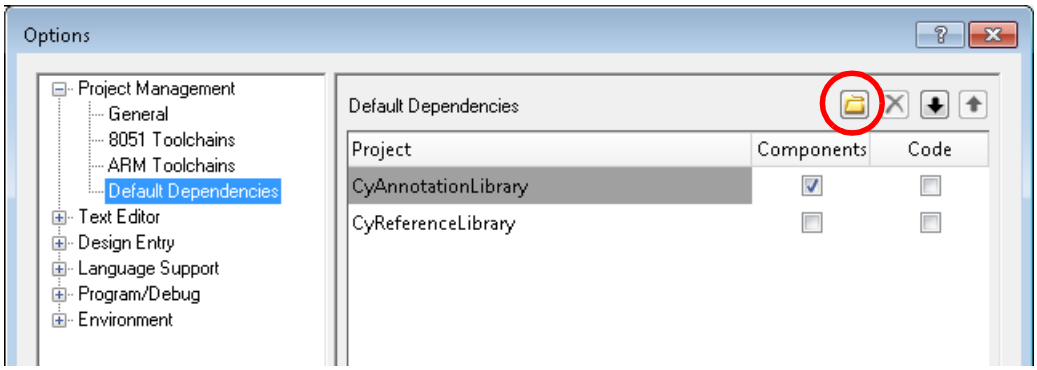


7.7.2 Add a Default Dependency

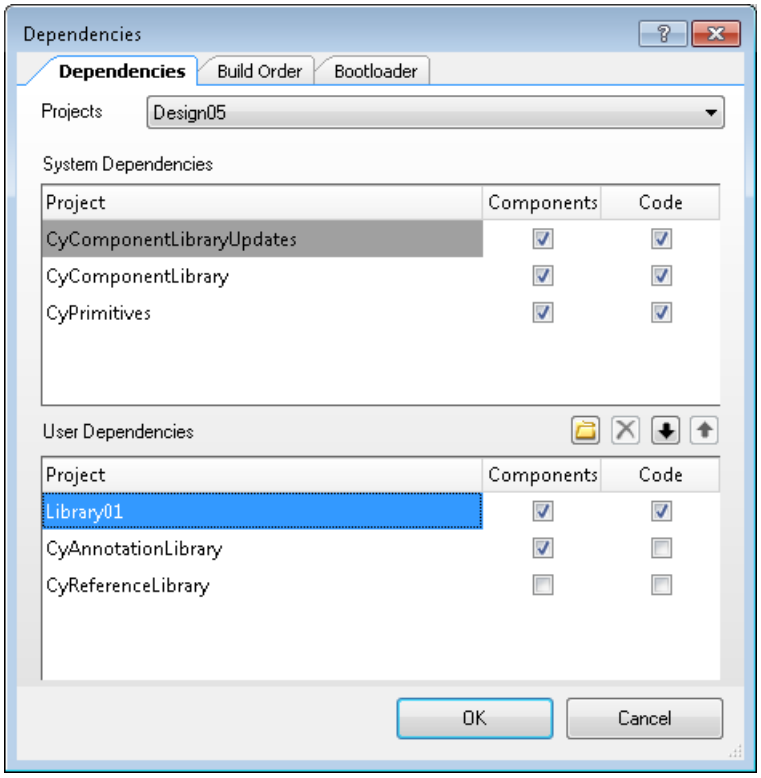
A Default Dependency is used in any PSoC Creator project.

1. With no project open in PSoC Creator, select **Tools > Options** to open the Options dialog.

2. On the dialog, expand the **Project Management** category and select **Default Dependencies**, then click the **New Entry** button.

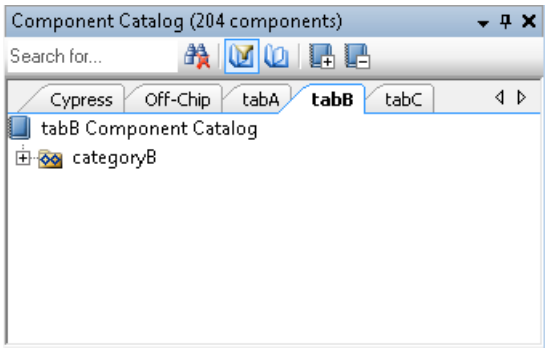


3. On the Open dialog, navigate to the location of the library project, select it, and click **Open** to close the dialog.
4. On the Options dialog, notice that the project you selected is listed under **Default Dependencies**. Click **OK** to close the Options dialog.

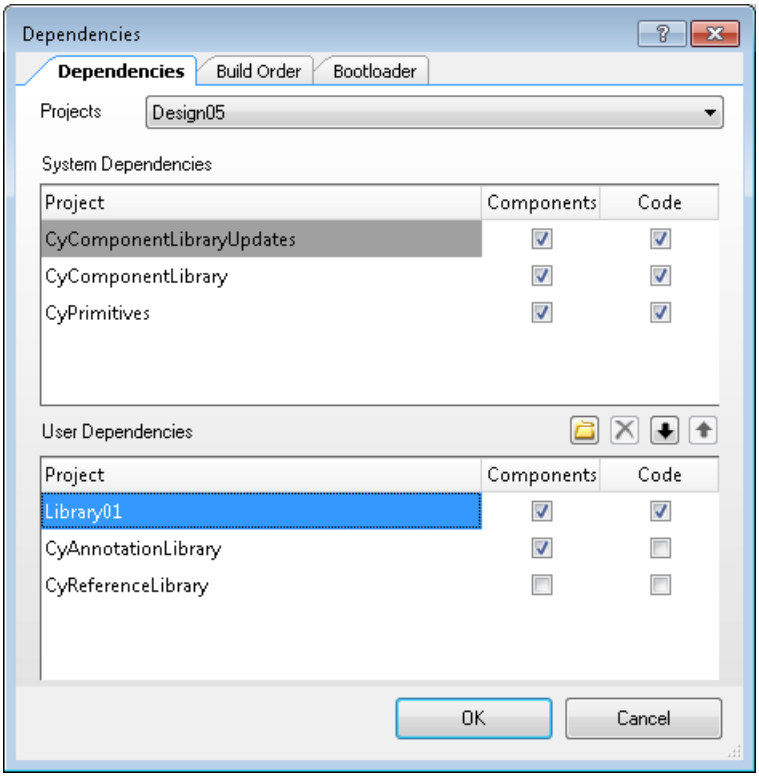


5. Open or create a design project.

6. On the Component Catalog, notice that there may be one or more new tabs and categories of Components, depending on how the Components were configured.



7. If you open the Dependencies dialog, notice that the project is already included under **User Dependencies**.



7.8 Build the project

When you have added and completed all the necessary Component items for the Component, you must instantiate the Component in a design in order to build and test it.

When a Component project is built, all portions of it are built, including the family-specific portions. After a build, the build directory will contain all aspects of the Component that the Component author has specified via the project manager.

8. Customizing Components (Advanced)

Customizing Components refers to the mechanism of allowing custom code (C#) to augment or replace the default behavior of an instantiated Component within PSoC Creator. The code is sometimes referred to as “customizers,” which may:

- customize the Configure dialog
- customize symbol shapes / display based on parameter values
- customize symbol terminal names, counts, or configuration based on parameters
- generate custom Verilog code
- generate custom C/assembly code
- interact with the clocking system (for clock and PWM Components)

This chapter provides examples for customizing a Component, provides instructions on customization source, and it lists and describes the interfaces that can be used to customize Components. The *icyinstancecustomizer.cs* C# source file provides the definitive parameters and return values for the methods provided in the customizer interfaces. The cydsextensions project contains the necessary source code for customization. This project is included with PSoC Creator, and the documentation is located in the *PSoC Creator Customization API Reference Guide* (*customizer_api.chm* file, located in the same directory as this *Component Author Guide*). The cydstoolkit project will contain code that complies with PSoC Creator Component versioning policies. Its primary purpose is to facilitate sharing of code between Components and PSoC Creator itself. Internal Cypress Component authors should place shared code here.

8.1 Customizers from Source

PSoC Creator accepts customizers developed in C#. The following sections describe different aspects of the C# source code.

8.1.1 Protecting Customizer Source

Source code customizers can depend on external assemblies. So if you want to avoid distributing your customizer as source code, you can have a source code stub which loads an external assembly, and just invokes methods in the external assembly.

8.1.2 Development flow

You will supply the C# source files, resource files, and assembly references for the given project. PSoC Creator will build the customization DLL out of the given code automatically at run time. This automatic build happens in the following cases:

1. A project is opened and it or any of its dependencies have customizer source files, but don't have a valid customizer DLL or the customizer DLL which was created earlier is outdated.
2. A project is built and it or any of its dependencies have customizer source files, but don't have a valid customizer DLL or the customizer DLL which was created earlier is outdated.

3. The Component author explicitly asks PSoC Creator to build a new customizer DLL.

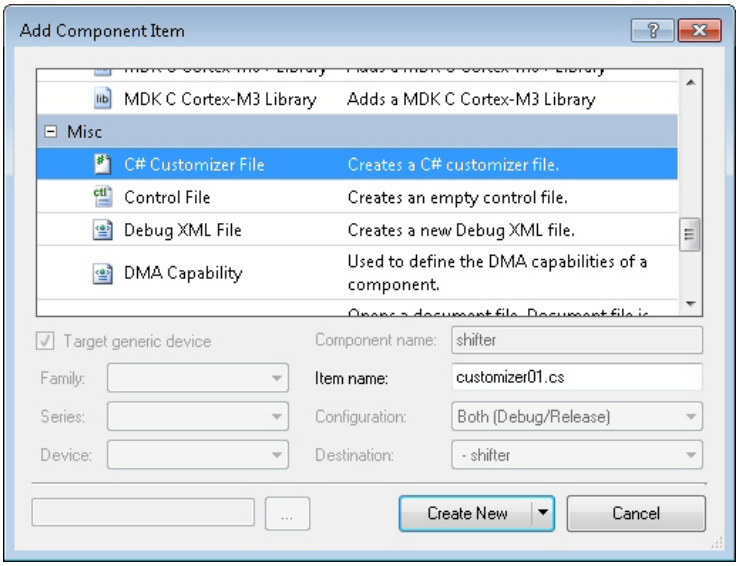
You will be able to specify whether to build the customizer DLL in "Debug" or "Release" mode, as well as specify command line options to the compiler.

If there are any errors/warnings during the above build procedure, the errors/warnings are displayed in the Notice List window. Since the DLL is built on the fly, there is no need to keep any of the built customizer DLLs in source control.

8.1.3 Add Source File(s)

To add source files to your Component:

1. Right-click on the Component and select **Add Component Item**.
   The Add Component Item dialog displays.



2. Select the C# icon under “Misc.”
3. Under **Target**, accept **Generic Device**.
4. For **Item Name**, type an appropriate name for the C# file.
5. Click **Create New**.

The Component item displays in the Workspace Explorer tree, in a sub-directory named “Custom.”

The Text Editor opens the .cs file, and you can edit the file at this time.

8.1.4 Create Sub-Directories in “Custom”

To add a sub-directory, right-click on the "Custom" directory and select **Add > New Folder**.

Note Any sub-directory created will be physical.

8.1.5 Add Resource Files

You can add .NET resource files (.resx) for the customizers. These files can be stored under the "Custom" directory or in sub-directories you created. Multiple resource files are allowed per Component.

To add the resource file:

1. Right click on the directory and choose **Add > New Item**.
2. On the New Item dialog, select the Resx file icon, type a name for the file, and click **OK**.

8.1.6 Name the Class / Customizer

There can only be one class which will implement the required customizer interfaces. This class must be named "CyCustomizer". The namespace under which this class resides must end with the name of the Component which it is customizing. For example, for a Component named "my_comp", the legal namespaces under which the "CyCustomizer" class could reside are:

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

See Usage Guidelines on page 117.

8.1.7 Specify Assembly References

You will be able to specify references to additional external DLLs for the customizers, from within PSoC Creator itself. The following assemblies are automatically included:

- "System.dll"
- "System.Core.dll"
- "System.Data.dll"
- "System.Windows.Forms.dll"
- "System.Drawing.dll"
- "System.Numerics.dll"
- "System.XML.dll"
- "cydsextensions.dll"
- "cydstoolkit.dll"

To specify .NET references and other user references, browse to the directory where the assembly resides and select it. In the case of .NET references you will have to browse to:

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

You can also specify relative paths in the assembly references dialog. You can add assembly references relative to the project top level directory (.cydsn). You will have to type in the relative path.

8.1.8 Customizer cache

The customizer cache is the directory where all the compiled customizer DLLs reside. The location of this directory depends on the executable which is compiling the DLLs. For example, if the customizer DLL is compiled from within PSoC Creator, the directory will be located at:

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

Similarly if the DLL is compiled during a build (within cydsfit), the directory will be located at:

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

8.2 Precompiled Component Customizers

A precompiled Component customizer is an advanced feature, and most Component authors will not need it. However, in the case of very large and static libraries with many Component customizers (such as libraries supplied with PSoC Creator), this feature can provide a performance improvement.

By design, Component customizers are rebuilt when necessary. Checking to see if the Component customizers are up to date requires inspecting all of the constituent files, and it can take some time to load those files from disk and inspect them.

To bypass loading and inspecting those files and to define the Component customizer assembly to use, it is possible to precompile the Component customizers for a library. This will build the Component customizer assembly, copy it into the library directory structure, and associate it with the library. The precompiled Component customizer is named _project_.dll and is visible in the Workspace Explorer at the top level of the project.

Note The existence of a precompiled Component customizer will bypass the source files. If the source files change, the Component customizer used will be silently out-of-date. In addition, a precompiled customizer is opened automatically, which will prevent the project from being renamed or the customizer from being updated. For these reasons, only use this feature if performance requires it, and only if you are sure that the Component customizers are unlikely to change.

To reduce confusion, PSoC Creator will refuse to build a new customizer assembly if a precompiled customizer assembly for a project already exists, since the precompiled customizer assembly will always be used.

To create a precompiled Component customizer assembly for a library, do the following.

1. Delete any existing precompiled Component customizer assembly from the project (see steps below).
2. Add the **Advanced Customizer** toolbar to PSoC Creator.
3. Click the **Build and Attach customizer** button.

To remove a precompiled Component customizer assembly from a library, do the following.

1. Remove the precompiled Component customizer assembly (named _project_.dll) from the project (but not from disk -- it is open, and the delete from disk will fail).
2. Exit from PSoC Creator, and then delete any precompiled Component customizer assembly in the project folder.

8.3 Usage Guidelines

8.3.1 Use Distinct Namespaces

The leafname of the namespace identifies the Component to which the customizer source applies. If resources (.resx) files are used, they too use that same name to identify the compiled resources.

When a Component is renamed or imported, all instances of the namespace within the customizer source are replaced.

To avoid unintended substitutions in the source code, and to allow shared source (where one Component can use common utility routines defined in another Component in the same library), all the Components in a library should have a common distinct prefix, for example:

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

8.3.2 Use Distinct External Dependencies

.NET resolves external assembly dependencies based on the assembly name, not the file location. So two different assemblies with the same name can get confused. To avoid problems with external dependencies, ensure that each externally referenced assembly has a distinct name. Experience shows that .NET does a better job distinguishing between strongly named assemblies.

8.3.3 Use Common Component To Share Code

If two customizers (e.g., A and B) want to share code, create a new Component called Common, which holds the code to be shared. Of course, if you import A or B, you also need to import Common.

8.4 Customization Examples

See the example project located in the following directory:

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

8.5 Interfaces

The customization support defines two sets of interfaces:

- Customization interfaces implemented by Components
- System interfaces used by customization interfaces

The interfaces are named with the version number concatenated to the end of the name. This allows for upgrades to the interface without breaking existing customizers.

For in depth documentation, refer to the *PSoC Creator Customization API Reference Guide* (*customizer_api.chm* file, located in the same directory as this *Component Author Guide*).

8.5.1 Clock Query in Customizers

PSoC Creator provides a robust mechanism for Component authors to provide and query for clock-related information. This is provided via a series of customization interfaces.

8.5.1.1 ICyTerminalQuery_v1

Has two methods called GetClockData. One takes two parameters (a terminal name, and an index) and returns the frequencies of the clock that drives the terminal. The second takes an "instance path", a terminal name, and an index and returns the clock frequency of buried clocks.

8.5.1.2 ICyClockDataProvider_v1

Components that generate or manipulate clocks can provide their frequency information to PSoC Creator and it will automatically be accessible to the GetClockData methods.

8.5.2 Clock API support

Certain Component APIs need access to the initial clock configuration in order to properly function (e.g., I2C). PSoC Creator will expose the initial configuration of the following via #defines in *cyfitter.h*.

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

9. Adding Tuning Support (Advanced)

Tuning support is a highly advanced feature that is used for a few limited Components, such as CapSense®, that require the ability for the user to tune the Component while it is in the design.

Tuning requires communication between the firmware Component and the host application. The firmware sends scan values to the GUI and the GUI sends updated tuning parameters down to the firmware. This is an iterative process in order to get the best possible results out of the device.

PSoC Creator supports this requirement by providing a flexible API that Component authors can use to implement tuning for their Component.

9.1 Tuning Framework

The tuning framework is a general purpose framework set up to support tuning by any Component. The framework primarily consists of two APIs. One API is implemented by Component authors and it is the launching point for a Component tuner. The second API is implemented by PSoC Creator and it provides a communication mechanism that a tuner uses to communicate with the PSoC device.

Communication to the device is done using an I^2C, EZ I^2C, SPI, or UART Component in the end user's design.

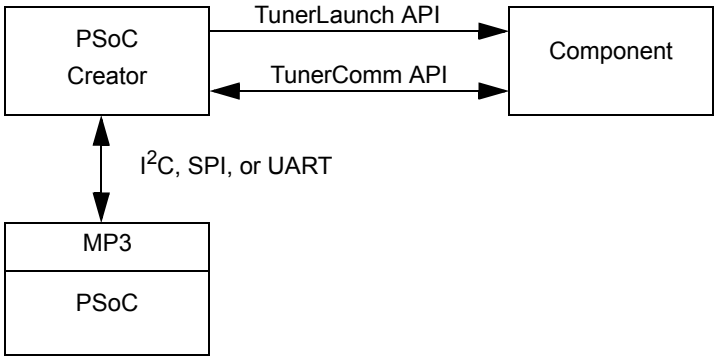
Tuning is typically done with a GUI that displays scan values on the various inputs and allows the user to set new tuning parameter values. The effects of the parameter changes are visible in real-time as they are applied.

The tuning application runs on a PC that is running PSoC Creator and it displays values as they come from a PSoC device. Parameter values are written from the PC down to the chip at the user's request.

It is the responsibility of the user to set up this two-way communication channel using a communication protocol that both the tuning framework and the tunable Component support. It is not possible for PSoC Creator to automatically set up a communication channel for the user's design because it is not safe to make assumptions about what communication is available or how the user is using the communication Components for additional purposes.

9.2 Architecture

The following diagram shows the PSoC Creator tuning framework. PSoC Creator launches tuning for a Component using the TunerLaunch API. The Component's tuner uses the TuningComm API to read and write to the device. PSoC Creator interacts with the PSoC device using I^2C, SPI, or UART and the MiniProg3.



9.3 Tuning APIs

PSoC Creator defines two APIs for use by Component authors to support tuning: LaunchTuner and TunerComm.

The following sections list and summarize the various tuning interfaces. For in depth documentation, refer to the *PSoC Creator Tuning API Reference Guide* (*tuner_api.chm* file, located in the same directory as this *Component Author Guide*).

9.3.1 LaunchTuner API

The LaunchTuner API is implemented by the Component and is called by PSoC Creator when the user selects the **Launch Tuner** option on a Component instance. Any Component that implements the LaunchTuner API is considered a tunable Component. PSoC Creator passes the tuner a communication object that implements the TunerComm API. The tuner uses this object to read and write from the device.

PSoC Creator launches the tuner as a non-modal window so that the user can do other things (such as debugging) while the tuner is running.

9.3.2 Communications API (ICyTunerCommAPI_v1)

The communications API defines a set the functions that allow the tuner GUI to communicate with the firmware Component without knowing any specifics about the communication mechanism being used. The application launching the tuner is responsible for implementing each of the functions for their desired communication protocol.

The TunerComm API is implemented by PSoC Creator and is used by a tuner to communicate with the PSoC device using a supported tuning communication channel.

The TunerComm API implements a data communication channel between the tuner GUI and the Component firmware on the chip. The contents of the data being communication are opaque to PSoC Creator. It is up to the Component tuner to make sure that the GUI and firmware agree on the contents of the data being passed.

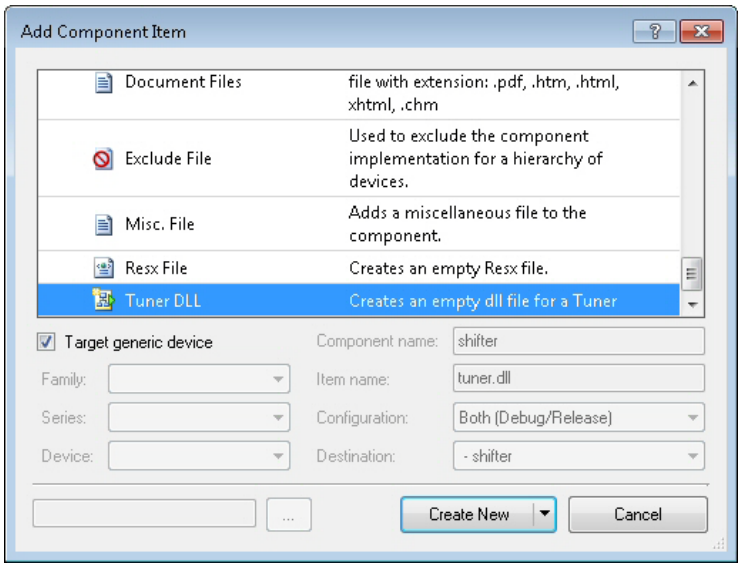
9.4 Passing Parameters

Parameters are passed between PSoC Creator and the tuner using the CyTunerParams class which is a simple wrapper class for name/value pairs.

9.5 Component Tuner DLL

The Component tuner code should be included with the Component as a tuner DLL. Add a “Tuner DLL” file to the Component using the Add Component Item dialog.

Note You cannot add more than one tuner DLL file to a Component.



PSoC Creator does not support source-based tuners. Therefore, the tuner must be compiled outside of the tool and supplied as a DLL.

9.6 Communication Setup

As described previously in this chapter, the communication setup between the tuner and the device is owned by the Component’s end user. A Component tuner has no knowledge of how this communication will be set up. To support this abstraction, the TunerComm API provides a set-up method. The Component tuner should call this method to initiate communication configuration.

Refer to the *PSoC Creator Tuning API Reference Guide* (*tuner_api.chm* file, located in the same directory as this *Component Author Guide*).

9.7 Launching the Tuner

PSoC Creator will provide a menu item that launches the tuner for a Component instance. This menu item is only available on the Component instance context menu since it only makes sense within the context of a particular tunable Component instance.

9.8 Firmware Traffic Cop

The user's design will contain a *main.c* file that implements the user's firmware design. In order to support tuning, the user's main function will need to contain "traffic cop" code that facilitates data transfer between the I^2C and tunable Component. The following pseudo code shows an example *main.c*.

```
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);
    }
  }
}
```

This main function relies on a way to synchronize the reads and writes between the chip and the host PC. This will likely be implemented by using a byte of data in the buffer for read and write enables.

9.9 Component Modifications

To support tuning the Component's firmware and APIs must be built to support tuning. It is likely that users will not want the overhead of tuning support in their final design so it is required that the Component configuration dialog give the user an option to put the Component into tuning mode. After the Component is in tuning mode, the user's design must be built and programmed into the device.

9.9.1 Communication Data

Data to/from the chip will be communicated via a shared c struct that contains fields for the scan values and the tuning parameter values. The exact structure of this struct is decided at build time by

the Component. The firmware that is built when in tuning mode will be aware of this structure so that it can process parameters correctly.

9.10 A simple tuner

The class that implements the tuner has full control over how it will operate. The following is example code for a tuner (the API documentation provides more details):

```
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());
    }
}
```

In this code, the tuner creates a custom GUI (MyForm) and then calls show dialog on this. Since this tuner is being run its own thread, it is safe for it to call the blocking call, ShowDialog(), without fear of hanging PSoC Creator.

PSoC Creator will call the LaunchTuner method and then wait for it to return. Once the method call returns, PSoC Creator will call GetParameters() to get the new parameter values which it will then store on the Component instance.

10. Adding Bootloader Support (Advanced)

This chapter describes how to provide bootloader support for a Component. For more information about the PSoC Creator bootloader system, refer to the *PSoC Creator System Reference Guide*.

To provide bootloader support in a Component, there are two general areas to modify:

- Firmware
- Customizer Bootloader Interface

10.1 Firmware

For a Component to support bootloading, it must implement five functions described in Section 10.1.2. These functions are used by the bootloader for setting up the communications interface and relaying packets back and forth with the host.

For more information about source code and implementing functions, see the Adding API Files chapter on page 75.

10.1.1 Guarding

Because there can be multiple bootloader supporting Components in the design at once, each with implementations of the necessary bootloader functions, all must be guarded for conditional preprocessor inclusion. This guard is:

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

    //Bootloader code

#endif
```

10.1.2 Functions

The following functions need to be implemented in the Component to support bootloading:

- CyBtldrCommStart
- CyBtldrCommStop
- CyBtldrCommReset
- CyBtldrCommWrite
- CyBtldrCommRead

The following sections provide function definitions.

10.1.2.1 void CyBtldrCommStart(void)

Description:	This function will start the selected communications Component. In many cases, this is just a call to the existing function `@INSTANCE_NAME`_Start()
Parameters:	None
Return Value:	None
Side Effects:	Starts up the communications Component and does any configuration necessary to allow data to be read and/or written by the PSoC.

10.1.2.2 void CyBtldrCommStop(void)

Description:	This function will stop the selected communications Component. In many cases, this is just a call to the existing function `@INSTANCE_NAME`_Stop()
Parameters:	None
Return Value:	None
Side Effects:	Stops up the communications Component and does any tear down necessary to disable the communications Component.

10.1.2.3 void CyBtldrCommReset(void)

Description:	Forces the selected communications Component to remove stale data. This is used when a command has been interrupted or is corrupt to clear the Component's state and begin again.
Parameters:	None
Return Value:	None
Side Effects:	Clears any cached data in the communications Component and sets the Component back to a state to read/write a fresh command.

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

Description:	Requests that the provided size number of bytes are written from the input data buffer to the host device. Once the write is done count is updated with the number of bytes written. The timeOut parameter is used to provide an upper bound on the time that the function is allowed to operate. If the write completes early it should return a success code as soon as possible. If the write was not successful before the allotted time has expired it should return an error.
Parameters:	uint8 *data – pointer to the buffer containing data to be written uint16 size – the number of bytes from the data buffer to write uint16 *count – pointer to where the comm. Component will write the count of the number of bytes actually written uint8 timeOut – amount of time (in units of 10 milliseconds) the comm. Component should wait before indicating communications timed out
Return Value:	CYRET_SUCCESS if one or more bytes were successfully written. CYRET_TIMEOUT if the host controller did not respond to the write in 10 milliseconds * timeOut milliseconds.
Side Effects:	None

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

Description:	Requests that the provided size number of bytes are read from the host device and stored in the provided data buffer. Once the write is done count is updated with the number of bytes written. The timeOut parameter is used to provide an upper bound on the time that the function is allowed to operate. If the read completes early it should return a success code as soon as possible. If the read was not successful before the allotted time has expired it should return an error.
Parameters:	uint8 *data – pointer to the buffer to store data from the host controller
	uint16 size – the number of bytes to read into the data buffer
	uint16 *count – pointer to where the comm. Component will write the count of the number of bytes actually read
	uint8 timeOut – amount of time (in units of 10 milliseconds) the comm. Component should wait before indicating communications timed out
Return Value:	CYRET_SUCCESS if one or more bytes were successfully read. CYRET_TIMEOUT if the host controller did not respond to the read in 10 milliseconds * timeOut milliseconds.
Side Effects:	None

10.1.3 Customizer Bootloader Interface

The bootloader requires that the communication Component is configured for both transfer in and out of the PSoC device. For Components that can be configured to meet this requirement, there is an interface that can be implemented by the customizer that will inform PSoC Creator of this support, as follows:

- ICyBootLoaderSupport

This interface is described in the the *PSoC Creator Customization API Reference Guide* (*customizer_api.chm* file, located in the same directory as this *Component Author Guide*), under “Common Interfaces.” It contains a single method that is used by the bootloader to determine whether the current configuration of the Component is bootloader compatible.

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

When the Component is bootloader ready, any instance placed in the design will be shown as an option for the bootloader IO Component in the Design-Wide Resources System Editor. The implementation of the single method within the customizer only needs to validate the current configuration of the Component to make sure the settings are compatable with bi-directional communication.

For more information about customizers, see the Customizing Components (Advanced) chapter on page 113.

11. Best Practices

This chapter covers general best practices to consider when creating Components, including:

- Clocking
- Interrupts
- DMA
- Low Power Support
- Component Encapulation
- Verilog

11.1 Clocking

When using the clocking resources internal to the PSoC device, there are several things to keep in mind. As with any digital system, clocking architectures must be completely understood and designed with the limitations and device characteristics firmly in mind.

Many of the digital resources available in the PSoC device can be clocked using the internal clocks of the device. However, this is not universally true. There are many other resources that can and will be contrived using clock sources other than those internally available in the chip. It is both of these cases that have to be considered when designing Components for use in the PSoC device.

11.1.1 UDB Architectural Clocking Considerations

Inside of each UDB, there may exist at least two clock domains. The first of these is the bus clock (BUS_CLK), used by the CPU in PSoC 3 and PSoC 5LP to access the registers internal to the UDB. The equivalent clock in PSoC 4 devices is the system clock (SYSCLK). The UDB registers include the status, control, and datapath internal registers.

The second class of clock domain is controlled by a “user” clock. This clock may have its origin entirely outside the PSoC internal clocking structure. This clock is the main clock for the implemented function. For PSoC 4 devices, the (internal) clock is driven via a HFCLK and its frequency must be below that of the SYSCLK to allow bus accesses by the CPU. The frequency restriction also applies to external clocks. PSoC 3 and 5LP devices do not have these restrictions.

To avoid metastable conditions in the UDB, synchronization flip-flops may be necessary whenever a signal crosses either clock domain boundary. There are a couple of ways that this can be accomplished with resources internal to the UDB. The first way is to use the PLD macrocells as synchronizers. The second way is to allocate the status register as a 4-bit synchronizer. The status register can be configured to allow its 8 internal flip-flops to become 4 dual flip-flop synchronizers. As expected, however, using the status register in this manner removes it from the pool of resources inside the UDB.

11.1.2 Component Clocking Considerations

When considering what clock should be used to clock a particular function in a design, it is sometimes tempting to allow both edges of a clock to be used. While this is sometimes acceptable in certain situations, it becomes very dangerous in others. If, for instance the clock source to be used is not a 50% duty cycle, the setup and hold timing budgets may be violated resulting in unexpected or unwanted results. To prevent these types of problems, all designs should use only *posedge* (Verilog instances) clocking and only connect clock pins of Components to the positive sense of the clock. If an event must happen on the negative edge of a clock, that can be accomplished by providing a 2X multiple of the needed clock to clock the circuitry appropriately.

11.1.3 UDB to Chip Resource Clocking Considerations

Signals that enter or leave the UDB array are not always intended for the I/O pins of the device. Contrarily, many of those signals connect to other resources on the PSoC such as fixed function blocks, DMA, or interrupts. Some of those blocks have built-in re-synchronizers, but others do not. Any time a signal crosses the boundary between the UDB array and one of these other elements, there is a possibility of problems. These signals have to be analyzed to ensure that they meet the required timing.

The timing analysis of signals crossing clock domain boundaries is necessary in all circumstances. Additionally, even for signals that are synchronous to the internal clocks, it may be necessary to validate their phase relationships to the region that they enter. A synchronous signal that is shifted by some amount may not satisfy the requirements for the timing in the destination circuitry.

11.1.4 UDB to Input/Output Clocking Considerations

The clocking structure available within the UDB that allows for global, BUS_CLK, and user clocks is not universally available to the rest of the PSoC architecture. Because of this limitation, signals sent to and/or received by the UDB must receive special consideration.

The GPIO registers have access only to BUS_CLK (PSoC 3/PSoC 5LP) or HFCLK (PSoC 4) and are not clock aligned with any external clock that a function may be using. Because of this limitation, any PSoC 3/PSoC 5LP output registers that are not clocked by BUS_CLK must use UDB resources before being sent to the output. This results in a very long clock to out path.

Any signal can be used by the UDB as an external clock signal. However, these external clocks do not come directly via the clock tree. They are routed through long paths before they can enter the clock tree. This makes the I/O timing problem more complex by creating long clock arrival times resulting in long set-up times.

11.1.5 Metastability in Flip-Flops

In any discussion of clocking for digital circuitry, it is necessary to understand the consequences of the clocking architecture. Paramount in that discussion should be a definition and explanation of metastable states of flip-flops.

Metastability can be defined as a period of time when the output of a flip-flop is unpredictable or is in an unstable (or metastable) state. Eventually, after some time, this state will resolve to a stable state of either a ‘1’ or a ‘0’. However, that resolution may not be quick enough for circuitry that is dependent upon the output to correctly evaluate the final result.

In combinatorial circuits, those outputs may cause glitches in the circuits that it drives. In sequential circuits, those glitches result in hazard conditions that could affect the storing of data in registers and

the decision processes for state machines. It is therefore imperative that metastable states be avoided.

There are several conditions that can result in metastable conditions, these conditions include:

- Signals that cross a clock domain boundary.
- Long combinatorial paths between flip-flops in a single clock domain.
- Clock skew between flip-flops in a single clock domain

Any one of these (or other) situations may cause a metastable condition. In general if the set-up time (`Tsu`) or hold time (`Th`) of the flip-flop is violated, a metastable state is possible if not probable.

11.1.6 Clock Domain Boundary Crossing

There are several storage elements internal to the PSoC 3/PSoC 5LP UDBs. Each one of these is accessible via the CPU BUS_CLK/SYSCLK. They are also accessible via some other clock source that is the primary clock for the circuit (ClkIn).

For PSoC 3/PSoC 5LP, this clock source can be selected from several places:

1. CPU BUS_CLK,
2. global clock (SrcClk), which can be any internal clock, such as IMO, ILO, and master clock, or
3. external clock (ExtClk).

For PSoC 4 devices, the clock selection is restricted to (2) clocks derived from High Frequency clock (HFLCLK) and (3) external clock (ExtClk).

If the ClkIn is the BUS_CLK/HFCLK, we can be assured that we have a single clock domain and worries about clock domain crossing can be eliminated. However, there may still remain situations where excessive skew is possible or long combinatorial paths exist.

11.1.7 Long Combinatorial Path Considerations

When long combinatorial paths are created in a design, it is possible that set-up time for a subsequent storage element is violated. To avoid such conditions, the total delays involved must be less than the cycle time of the clock. In other words, the following equation must be true:

$$Tco + Tcomb + Tsu < Tcycle \qquad \textbf{Equation 1}$$

`Tco` represents the clock to out time of the driving flip-flop. `Tcomb` is the combinatorial delay of the intervening circuitry. `Tsu` is the set-up time of the next flip-flop stage.

If these long paths are contained completely within a Component, they are more easily handled. Therefore, signals leaving a Component should be driven by a flip-flop to avoid problems. If this is not possible or desirable, a complete understanding of the timing from clock to output of the Component must be understood and communicated.

11.1.8 Synchronous Versus Asynchronous Clocks

Clocking architectures often contain multiple types of clocks. Some of these clocks may be synchronous to a master system clock and some of them may be asynchronous to that clock. For our purposes, we define the master system clock as that clock (or some derivative of it) that the CPU uses as its core clock. This is the clock that is referred to as BUS_CLK.

To best avoid metastability problems, signals that are interfaced to the CPU should be synchronized with the BUS_CLK/HFCLK signal. If they are already a derivative of that clock, then the task for a designer is to ensure that there are no long combinatorial path problems or problematic skews (discussed earlier).

If signals need to be interfaced to the MPU but are controlled by clocks that are asynchronous to BUS_CLK/HFCLK, they may need to be synchronized before being presented to that interface. There are some primitive Components that are available in PSoC Creator that can help with that task and are detailed in the following sections.

11.1.9 Utilizing cy_psoc3_udb_clock_enable Primitive

There is a means available in PSoC Creator accessible by instantiation of a Verilog primitive that can help with the complexities of clocking and handle some clock conditioning automatically. That primitive is the cy_psoc3_udb_clock_enable. The use of this primitive can aid in handling clocks that are synchronous as well as asynchronous.

The cy_psoc3_udb_clock_enable has inputs for an enable (enable) and clock (clock_in), a clock output (clock_out) that will drive the UDB Components, and a parameter to specify the intended synchronization behavior for the clock result (sync_mode).

The clock_in signal can be:

1. global clock (SrcClk, a clock source internal to PSoC such as IMO, ILO, master clock, HFCLK etc.)
2. local clock (output of a divider from a clock source SrcClk such as IMO, ILO, master clock, HFCLK etc.)
3. external clock (ExtClk, a clock routed from external source into the chip via a pin)

These can be either a global clock or a local clock and can be either synchronous or asynchronous to BUS_CLK/HFCLK. The enable signal can also be either synchronous or asynchronous to BUS_CLK/HFCLK. These two signals are then connected to the primitive and the user selects either synchronous or asynchronous mode.

Once these have been done, the fitter in PSoC Creator will determine the implementation necessary to obtain the requested clock behavior for the UDB elements and attach the appropriate signals to the mapped UDB results. The rule set used when mapping the clock/enable to the UDB is listed in the following table:

Mapping Clock/Enable to UDB

Inputs			UDB Translation			
clock_in	enable	sync_ mode	Enable Mode	Translated enable	clock_out	Description
Global clock, Sync to BUS_CLK/HFCLK	Sync to clock_in	Yes	Level	enable	clock_in	Already synchronous to clock_in so just output clock_in.
Global clock, Async to BUS_CLK/HFCLK	Sync to clock_in	Yes	Not allowed, error during synthesis			
Local clock, Sync to BUS_CLK/HFCLK	Sync to clock_in	Yes	Edge	clock_in & enable	SrcClk (clock_in)	clock_out is synchronous to the source clock.
Local clock, Async to BUS_CLK/HFCLK	Sync to clock_in	Yes	Edge	Sync(clock_in & enable)	BUS_CLK/ HFCLK	clock_in and enable are synced to clock_out. clock_out is synchronous to BUS_CLK/HFCLK.

Mapping Clock/Enable to UDB

Inputs			UDB Translation			
clock_in	enable	sync_ mode	Enable Mode	Translated enable	clock_out	Description
Global clock, Sync to BUS_CLK/HFCLK	Async	Yes	Level	Sync(enable)	clock_in	enable is synced to clock_out. clock_out is just clock_in.
Global clock, Async to BUS_CLK/HFCLK	Async	Yes	Not allowed, error during synthesis			
Local clock, Sync to BUS_CLK/HFCLK	Async	Yes	Edge	Sync(clock_in & enable)	SrcClk (clock_in)	enable and clock_in are synced to clock_out. clock_out is synchronous to source clock.
Local clock, Async to BUS_CLK/HFCLK	Async	Yes	Edge	Sync(clock_in & enable)	BUS_CLK/ HFCLK	enable and clock_in are synced to clock_out. clock_out is synchronous to BUS_CLK/HFCLK.
Global clock, Sync to BUS_CLK/HFCLK	Sync	No	Level	enable	clock_in	enable is already synced to clock_in. clock_out is just clock_in.
Global clock, Async to BUS_CLK/HFCLK	Sync	No	Level	enable	clock_in	enable is already synced to clock_in. clock_out is just clock_in.
Local clock, Sync to BUS_CLK/HFCLK	Sync	No	Level	enable	ExtClk (clock_in)	enable is already synced to clock_in. The Local clock is routed through the external clock routing path. Therefore clock_out is clock_in routed through ExtClk path.
Local clock, Async to BUS_CLK/HFCLK	Sync	No	Level	enable	ExtClk (clock_in)	enable is already synced to clock_in. The Local clock is routed through the external clock routing path. Therefore clock_out is clock_in routed through ExtClk path.
Global clock, Sync to BUS_CLK/HFCLK	Async	No	Level	Sync(enable)	clock_in	enable is synced to clock_in. clock_out is just clock_in.
Global clock, Async to BUS_CLK/HFCLK	Async	No	Level	Sync(enable)	clock_in	enable is synced to clock_in. clock_out is just clock_in.
Local clock, Sync to BUS_CLK/HFCLK	Async	No	Level	Sync(enable)	ExtClk (clock_in)	enable is synced to clock_in. The Local clock is routed through the external clock routing path. Therefore clock_out is clock_in routed through ExtClk path.
Local clock, Async to BUS_CLK/HFCLK	Async	No	Level	Sync(enable)	ExtClk (clock_in)	enable is synced to clock_in. The Local clock is routed through the external clock routing path. Therefore clock_out is clock_in routed through ExtClk path.

The Sync() function indicates that a double flip-flop has been used to synchronize the signal with clock_out. These double registers are taken from a status cell configured in sync mode. The Enable mode relates to the mode in which the internal logic chooses the clock signal in the Clock Select/ Enable control logic. This will be Edge type when the sync_mode is on and clock_in is a routed clock.

A typical instantiation of the cy_psoc3_udb_clock_enable primitive might look something like this:

```
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 Utilizing cy_psoc3_sync Component

Another useful tool is the cy_psoc3_sync primitive. This primitive is a double flip-flop used to synchronize signals to a clock. These double flip-flops are created as an option to the UDB status register. Even though the primitive is a single bit wide, it will consume an entire status register. It is possible, however, to instantiate four of these primitives and still consume a single register since there are eight flip-flops available.

A typical instantiation of the cy_psoc3_sync primitive might look something like this:

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

11.1.11 Routed, Global and External Clocks

There are three types of clocks available to the UDBs. These include:

- eight global clocks, output from user-selectable clock dividers
- BUS_CLK, the highest frequency clock in the system
- external clock, routed from an external signal and used as a clock input to support direct clocked functions (such as SPI Slave).

11.1.12 Negative Clock Edge Hidden Dangers

When designing with Verilog, we have determined that negedge statements should be avoided. This philosophy is also applicable to clocks that connect to instantiated Components or schematic Components.

Mixing positive edge triggering and negative edge triggering commonly restricts the length of time a signal has to transverse it path. If, for instance, a flip-flop is clocked with the positive edge and the resulting output is sent through some logic to another flip-flop that is clocked with a negative edge, the `Tcycle` is effectively cut in half. If there is a need to trigger on both edges of a clock, a double rate clock should be implemented and alternate positive edges should be used for triggering.

11.1.13 General Clocking Rules

- Do not create clocks in the Component. A clock signal should be an input to the Component and all clocked logic should be clocked on that single clock. For example, do not AND a clock signal to implement a gated clock.
- Do not take a global clock signal and create a combinatorial signal with it. Similarly do not send a global clock signal out of a Component. The timing of a global clock when it is used for anything other than as the clock input to a UDB element is not defined. To create a clock signal with a defined timing relationship, the output of a flip-flop clocked by another clock must be used. This allows the generation of a clock output at 1/2 or slower rate than the input clock.
- Drive all outputs from a Component from a clocked element (flip-flop) to minimize the clock to out time.
- Avoid asynchronous reset and preset. From a timing perspective they will cause a combinatorial timing path from the reset/preset signal to the registered output to exist.
- If a data signal will be asynchronous to a clock, then a synchronizer is required.
- Global clock signals should not be output from a Component.
- Avoid clocking on negative edges of clocks. Use a 2x clock instead.

11.2 Interrupts

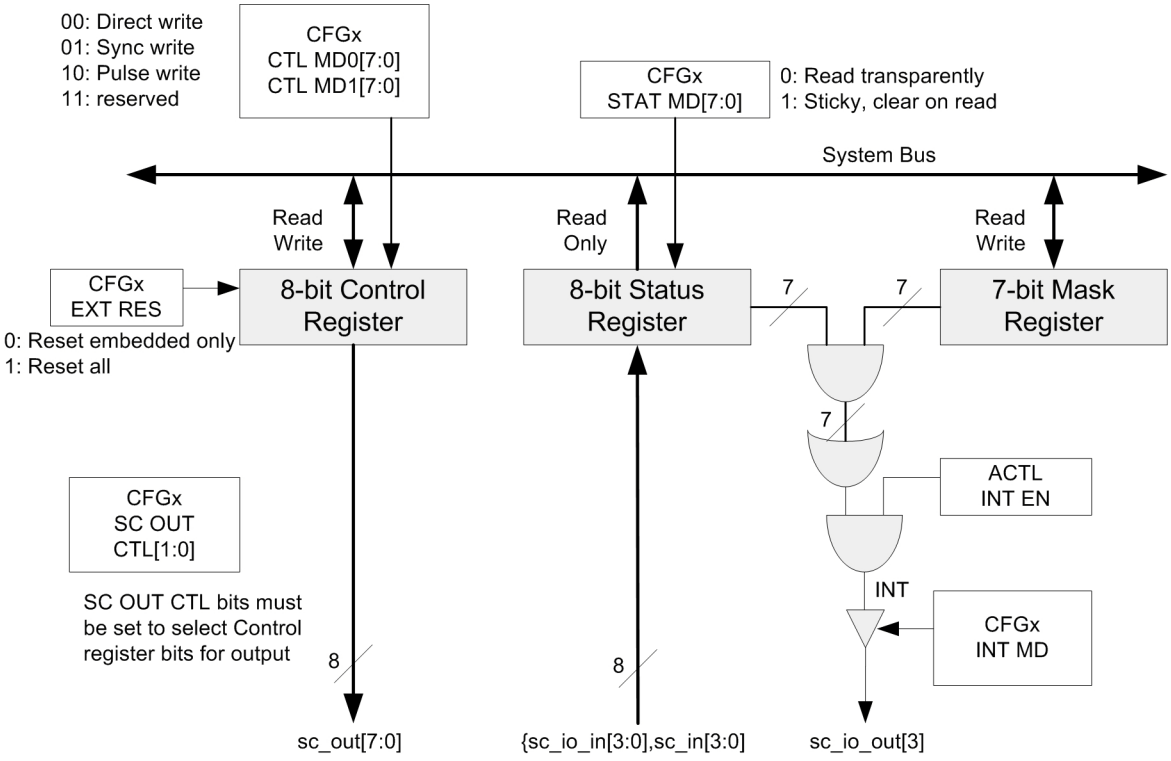
The main role of interrupts is to interact with CPU firmware or a DMA channel. Any data signal in the UDB array routing can be used to generate an interrupt or DMA request. The status register and FIFO status registers are considered the primary means of generating these interrupts. When small amounts of data and frequent interaction with the CPU is required, the status register is the logical choice for generating interrupts.

Whether an interrupt is buried in a Component or exposed as a terminal on the symbol should be based on whether the processing of the interrupt is specific to the application. If it is specific the Component should perform the necessary operations and then add merge banners for any application handling if appropriate.

The generation of interrupts or DMA request is specific to the design. For example an interrupt that is targeted for the CPU may be designed to be sticky (clear on read) but that same request for DMA would not be appropriate because the design of the transaction descriptors would have to incorporate a separate descriptor to clear the request in addition to the transaction descriptors to process the data.

11.2.1 Status Register

A high level view of the Status and Control module is shown below. The primary purpose of this block is to coordinate CPU firmware interaction with internal UDB operation.



The status register is read-only and it allows internal UDB state to be read out onto the system bus directly from internal routing. This allows firmware to monitor the state of UDB processing. Each bit of these registers has programmable connections to the routing matrix.

A status interrupt example would be a case where a PLD or datapath block generated a condition, such as a “compare true” condition, that is captured by the status register and then read (and cleared) by CPU firmware.

11.2.2 Internal Interrupt Generation and Mask Register

In most functions, interrupt generation is tied to the setting of status bits. As shown in the figure above, this feature is built into the status register logic as the masking (mask register) and OR reduction of status. Only the lower 7 bits of status input can be used with the built-in interrupt generation circuitry. By default the sc_io pin is in output mode and the interrupt may be driven to the routing matrix for connection to the interrupt controller. In this configuration, the MSB of the status register is read as the state of the interrupt bit.

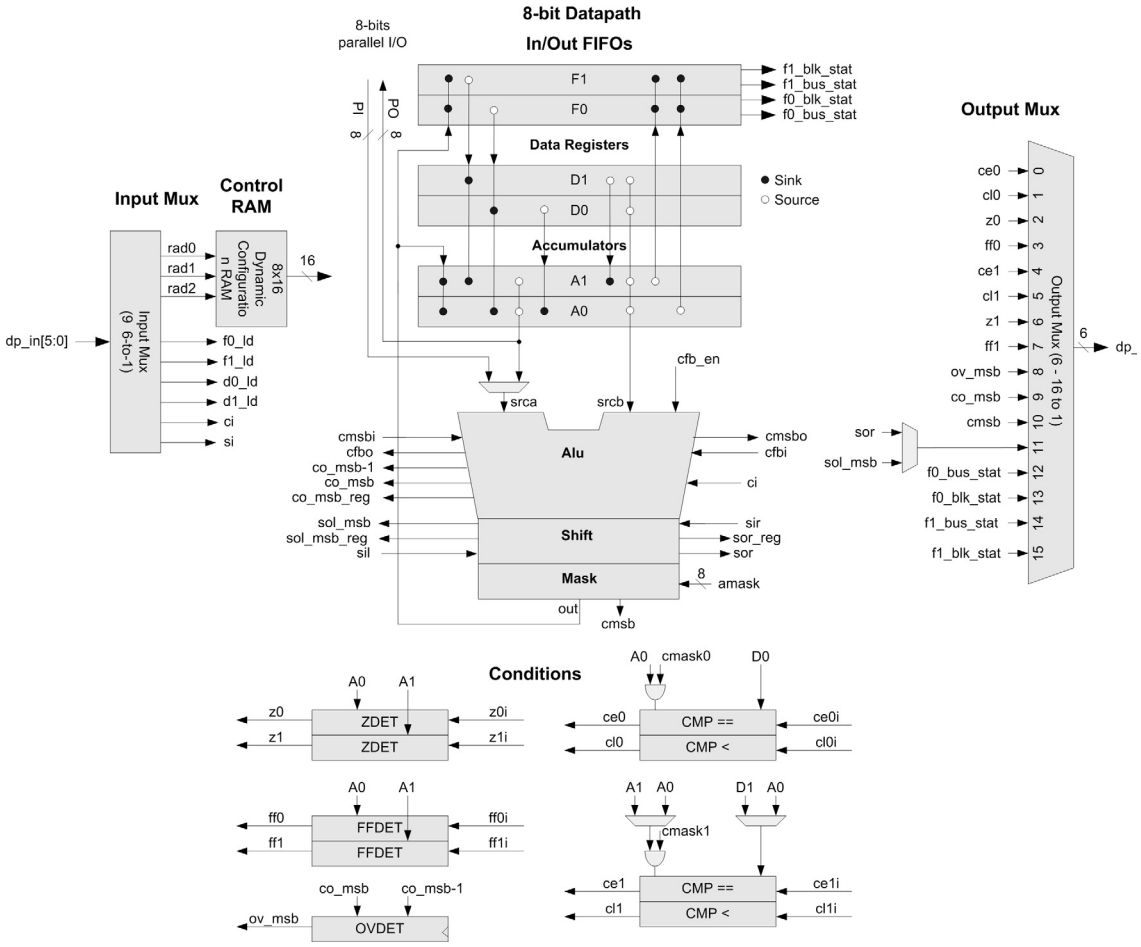
The status mode register (CFGx) provides mode selection for each bit of the status register. Transparent read is a mode in which a CPU read of the status register returns the state of the routing input signal. Sticky mode, which is a clear on read, is a mode which the input status is sampled and when the input goes high, the register bit is set and stays set regardless of the subsequent state of the input. The register bit is cleared on a subsequent read by the CPU. The selected clock for this block determines the sample rate. The rate should be greater than or equal to the rate at which the status input signals are being generated.

11.2.3 Retention Across Sleep Intervals

The mask register is retention and will retain state across sleep intervals. The status register is non-retention. It loses its state across sleep intervals and is reset to 0x00 on wakeup.

When large amounts of data are being streamed or rapid burst are being transmitted or received, DMA transactions are the most reasonable method of transport. In these cases, interrupts to control data flow should be performed using FIFO status registers. The following diagram shows a high-

level view of the datapath module. These FIFOs generate status that can be routed to interact with sequencers, interrupts, or DMA requests.



11.2.4 FIFO Status

There are four FIFO status signals (f1_blk_stat, f1_bus_stat, f0_blk_stat, f0_bus_stat), two for each FIFO, that can be independently configured for direction as an input buffer (system bus writes to the FIFO, datapath internally reads the FIFO), or an output buffer (datapath internally writes to the FIFO, and the system bus reads from the FIFO). The “bus” status is meaningful to the device system and should be routed to the DMA controller as a DMA request.

The “bus” status is primarily intended for the system bus control for DMA interaction (when the system should read or write bytes).

When implementing a buffer for transactions, a decision should be made with regard to how much RAM will be used to store the data. If the size of the buffer is less than or equal to 4 bytes, the buffer should be implemented with the FIFO hardware.

The buffering provided in the receive and transmit FIFOs allows the processing order of user application logic to be independent of the order of data transfer on the bus. The receive FIFO also absorbs the usually observed bursty nature of data on the interface. This FIFO could also decouple the operating frequency of the user application logic from the frequency of the specific bus. Consider both buffer overflow and underflow in the implementation.

11.2.5 Buffer Overflow

The receive FIFO should have enough space to accommodate all the pending (already scheduled or to be scheduled) data transfer to the particular port to avoid the potential overflows. Ideally, a receive FIFO design and status check mechanism should ensure that there are no data losses due to overruns.

To solve the overflow problem, designers must employ a look-ahead indication of the FIFO status. Thus, any port FIFO would indicate a satisfied status when the data path latency + status path latency + maximum burst transfer is less than being full. This implies that the high watermark for a FIFO must be set equal to data path latency + status path latency + maximum burst. Effectively, an additional mandatory space after satisfied indication has to be provided in the port FIFO to avoid buffer overflows.

11.2.6 Buffer Underflow

A receive port FIFO underflows when data falls below the low watermark and receives no data from the other end through the interface, and eventually goes empty even though the transmit FIFO has data to send for that port. This happens because the transmitter has exhausted the previously granted allotment before it gets the next update, for example status starving or hungry from the receiver. To prevent the underflow, the watermark of status indication must be set high enough so that the transmitter responds to FIFO space available indication from the receiver before the application logic drains the port data from the FIFO.

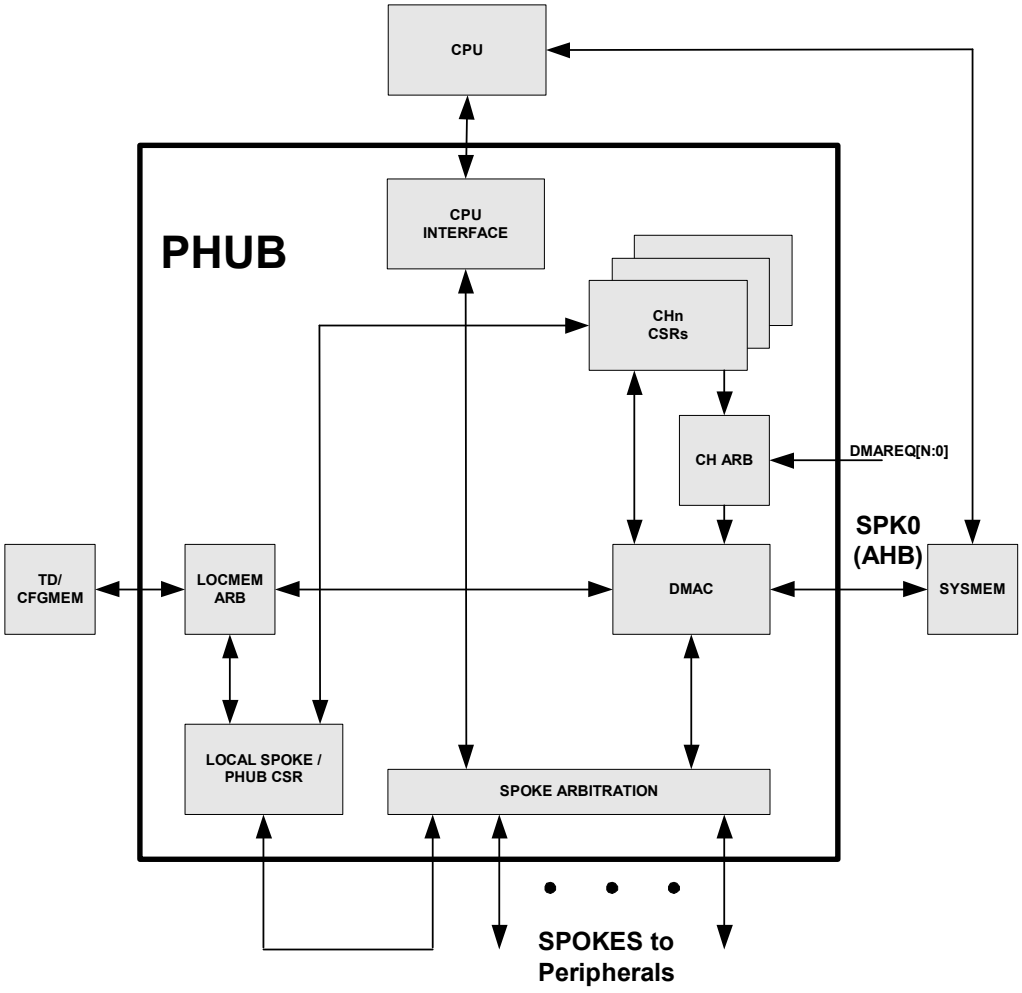
The time elapsed between the FIFO status indicating starving or hungry to get the data for that particular port is the total path latency, which is the sum of status update latency + status path latency + data scheduler latency and finally the data path latency. The first two numbers reflect the amount of time required in getting the burst information built up at the transmitter. The last two numbers define the amount of time required to get the data moved across the interface from transmit FIFO to the receive FIFO over the particular link.

The buffer underflow depends on the maximum read rate of the port FIFO by the application logic. To prevent underflow, software should program low watermark for each port FIFO, judiciously (that is, large enough).

11.3 DMA

The Peripheral HUB (PHUB) is a programmable and configurable central hub within a PSoC 3 and PSoC 5 device that ties the various on-chip system elements together utilizing standard Advanced Microcontroller Bus Architecture (AMBA) high-performance bus (AHB). The PHUB essentially utilizes a multi-layer AHB architecture allowing for simultaneous AMBA-lite style mastering. The PHUB contains a DMA controller (DMAC) that can be programmed to transfer data between system elements without burdening the CPU. PHUB contains logic that performs arbitration between DMAC and the CPU for access to PHUB's downstream spokes.

The diagram below illustrates the general connectivity between the CPU, PHUB, SYSMEM, TD/ CFGMEM and the downstream spokes.



The details with regard to the register map are contained in the *PSoC® 3, PSoC® 5 Architecture Technical Reference Manual* (TRM). The points of concern here are that the DMA controller offloads the CPU, it is a separate bus master, and that the DMAC arbitrates between multiple DMA channels. The DMA handler and associated APIs are outlined in DMA Component datasheet.

The main points of this section is to consider how to construct a Component to use DMA, methods of data transfer utilizing DMA, how to signal a transfer and what methods are best for transferring either large amounts of data or small packets that may only consist of single byte. To reduce the complexity

of configuration a DMA wizard is provided for the end user. As part of the Component development the capabilities of the Component with regard to DMA may be provided in an XML file.

11.3.1 Registers for Data Transfer

System bus connections are common to all UDBs and allow DMA access to registers and RAM in the UDBs for both normal operation and configuration.

Each datapath module has six 8-bit working registers. All registers are CPU and DMA readable and writable.

Each datapath contains two 4-byte FIFOs, which can be individually configured for direction as an input buffer or an output buffer. These FIFOs generate status that can be routed to interact with sequencers, interrupts, or DMA requests. For an input buffer, thesystem bus writes to the FIFO, and datapath internals read the FIFO. For an output buffer, datapath internals write to the FIFO, and the system bus reads from the FIFO.

For small transfers, the accumulator should be used especially when a single packet of data will be transmitted or received and where computation on the data is necessary before another packet is sent.

For large data transfers that require continuous streams of data, FIFOs are particularly useful. Along with the FIFO status logic, continuous data streams can be maintained without the loss of data.

Type	Name	Description
Accumulator	A0, A1	The accumulators may be a source for the ALU and destination of the ALU output. They also may be loaded from an associated data register or FIFO. The accumulators contain the current value of the function; for example, the count, CRC, or shift. These registers are non-retention; they lose their value in sleep and are reset to 0x00 on wakeup.
Data	D0, D1	The data registers contain the constant data for a given function, such as a PWM compare value, timer period, or CRC polynomial. These registers are retention. They retain their value across sleep intervals.
FIFOs	F0, F1	There are two 4-byte FIFOs to provide a source and destination for buffered data. The FIFOs can be configured both as input buffers, both as output buffers, or one input and one output buffer. Status signaling, which can be routed as a datapath output, is tied to the reading and writing of these registers. Examples of usage include buffered TX and RX data in SPI or UART and buffered PWM compare and buffered timer period data. FIFOs are non-retention. They lose their contents in sleep and the contents are unknown on wakeup. FIFO status logic is reset on wakeup.

11.3.2 Registers for Status

There are four FIFO status signals, two for each FIFO: fifo0_bus_stat, fifo0_blk_stat, fifo1_bus_stat and fifo1_blk_stat. The meaning of these signals depends on the direction of the given FIFO, which is determined by static configuration. The “bus” status is meaningful to the device system and is usually routed to the interrupt controller, DMA controller, or it could be polled through a status register. The “blk” status is meaningful to the internal UDB operation and is normally routed to the UDB Component blocks, such as a state machine built from PLD macrocells.

There are two status bits generated from each FIFO block that are available to be driven into the UDB routing through the datapath output multiplexer. The “bus” status is primarily intended for the system bus control for CPU/DMA interaction (when the system should read or write bytes). The “block” status is primarily intended for local control to provide FIFO state to the internal UDB state

machines. The meaning of the status depends on the configured direction (Fx_INSEL[1:0]) and the FIFO level bits.

FIFO level bits (Fx_LVL) are set in the auxiliary control register in working register space. Options are shown in the following table.

Fx_INSEL [1:0]	Fx_LVL	Signal	Status	Description
Input	0	fx_bus_stat	Not Full	This status is asserted when there is room for at least 1 byte in the FIFO. This status can be used to assert a system interrupt or DMA request to write more bytes into the FIFO.
Input	1	fx_bus_stat	At Least Half Empty	This status is asserted when there is room for at least 2 bytes in the FIFO.
Input	N/A	fx_blk_stat	Empty	This status is asserted when there are no bytes left in the FIFO. When not empty, the Datapath function may consume bytes. When empty the control logic may idle or generate underrun status.
Output	0	fx_bus_stat	Not Empty	This status is asserted when there is at least 1 byte available to be read from the FIFO. This status can be used to assert a system interrupt or DMA request to read these bytes out of the FIFO.
Output	1	fx_bus_stat	At least Half Full	This status is asserted when there is at least 2 bytes available to be read from the FIFO.
Output	N/A	fx_blk_stat	Full	This status is asserted when the FIFO is full. When not full, the Datapath function may write bytes to the FIFO. When full, the Datapath may idle, or generate overrun status.

11.3.3 Spoke width

The DMA controller transfers data on a spoke in sizes equal to the datawidth of the spoke. However, AHB rules require all data transfers be aligned to the address boundary equal to the size of the transfer. Which means ADR[1:0] of 32-bit transfers must equal 0b00, and ADR[0] of 16-bit transfers must equal 0. The address can take on any value for 8-bit transfers. This means that if the overall burst starts or ends on an address boundary that doesn't equal the datawidth of the spoke, then this creates a ragged start or end.

The following table defines the peripherals associated with a spoke and the width of the spoke.

PHUB Spokes	Peripherals	Spoke Datawidth
0	SRAM	32
1	IOs, PICU, EMIF	16
2	PHUB local configuration, Power manager, Clocks, Interrupt controller, SWV, EEPROM, Flash programming interface	32
3	Analog interface, Decimator	16
4	USB, CAN, I2C, Timers, Counters, PWMs	16
5	DFB	32
6	UDBs group 1	16
7	UDBs group 2	16

The source spoke and destination spoke can be different sizes. The burst engine will use the FIFO within the DMA controller as a funneling mechanism between the two spokes.

11.3.4 FIFO Dynamic Control Description

The configuration between internal and external access is dynamically switchable via datapath routing signals. The datapath input signals d0_load and d1_load are used for this control.

Note In the dynamic FIFO control mode, d0_load and d1_load are not available for their normal use in loading the D0/D1 registers from F0/F1.

In a given usage scenario, the dynamic control (dx_load) can be controlled with PLD logic or any other routed signal, including constants. For example, starting with external access (dx_load == 1), the CPU or DMA can write one or more bytes of data to the FIFO. Then toggling to internal access (dx_load == 0), the datapath can perform operations on the data. Then toggling back to external access, the CPU or DMA can read the result of the computation.

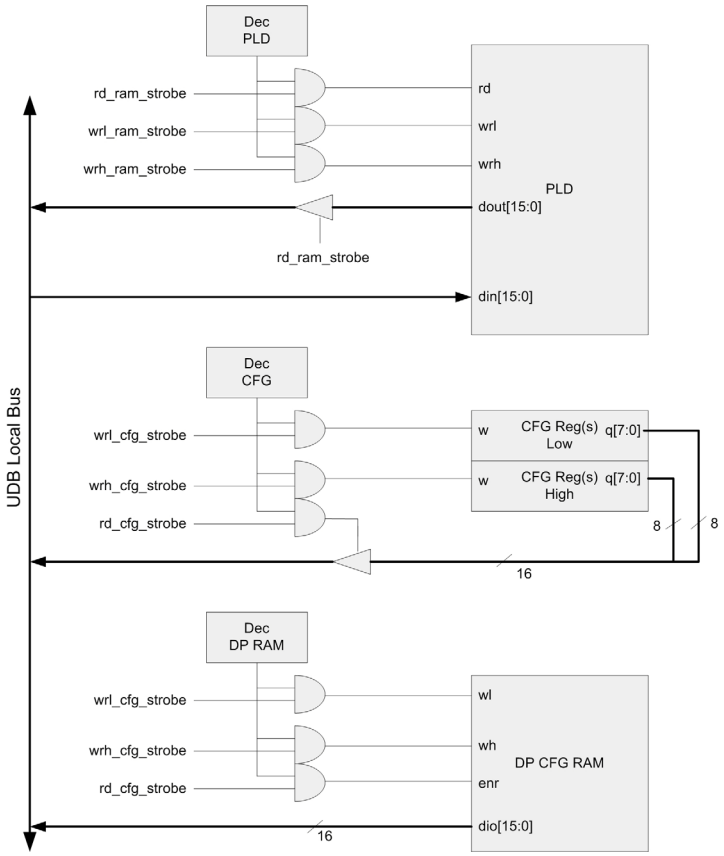
11.3.5 Datapath Condition/Data Generation

Conditions are generated from the registered accumulator values, ALU outputs, and FIFO status. These conditions can be driven to the UDB channel routing for use in other UDB blocks, for use as interrupts or DMA requests, or to globals and I/O pins. The 16 possible conditions are shown in the following table:

Name	Condition	Chain?	Description
ce0	Compare Equal	Y	A0 == D0
cl0	Compare Less Than	Y	A0 < D0
z0	Zero Detect	Y	A0 == 00h
ff0	Ones Detect	Y	A0 = FFh
ce1	Compare Equal	Y	A1 or A0 == D1 or A0 (dynamic selection)
cl1	Compare Less Than	Y	A1 or A0 < D1 or A0 (dynamic selection)
z1	Zero Detect	Y	A1 == 00h
ff1	Ones Detect	Y	A1 == FFh
ov_msb	Overflow	N	Carry(msb) ^ Carry(msb-1)
co_msb	Carry Out	Y	Carry out of MSB defined bit
cmsb	CRC MSB	Y	MSB of CRC/PRS function
so	Shift Out	Y	Selection of shift output
f0_blk_stat	FIFO0 block status	N	Definition depends on FIFO configuration
f1_blk_stat	FIFO1 block status	N	Definition depends on FIFO configuration
f0_bus_stat	FIFO0 bus status	N	Definition depends on FIFO configuration
f1_bus_stat	FIFO1 bus status	N	Definition depends on FIFO configuration

11.3.6 UDB Local Bus Configuration Interface

The following figure illustrates the structure of the interface of the configuration state to the UDB local bus interface.



There are three types of interfaces: the PLD, the configuration latches, and the DP configuration RAM. All configurations are writable as 16 bits to support DMA or as 16-bit processor operations. They are also separately writable as upper (odd addresses) and lower (even addresses) bytes. The PLD has unique read signals which implement RAM read and write timing. The CFG registers and DP CFG RAM share the same read and write control signals.

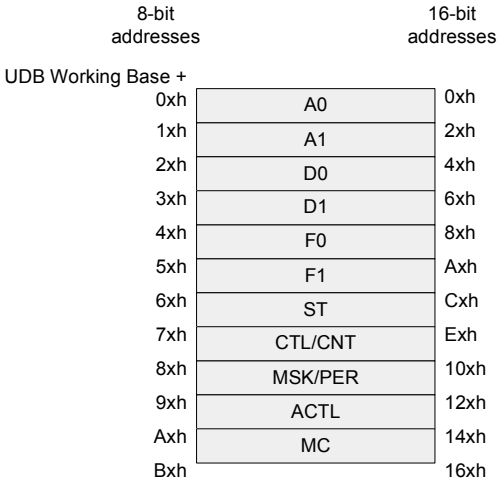
11.3.7 UDB Pair Addressing

Methods of data transfer using DMA depend on how the working and configuration registers are configured. There are three unique address spaces in the UDB pair.

- 8-bit Working Registers – A bus master that can only access 8-bits of data per bus cycle can use this address space to read or write any UDB working register. These are the registers that CPU firmware and DMA interacts with during the normal operation of the block.
- 16-bit Working Registers – A bus master with 16-bit capability can access 16-bits per bus cycle to facilitate the data transfer of functions that are inherently 16-bits or greater. Although this address space is mapped into a different area than the 8-bit mode, the same 8-bit UDB hardware registers are accessed, with two registers responding to the access.
- 8 or 16-bit Configuration Registers – These registers configure the UDB to perform a function. Once configured, are normally left in a static state during function operation. These registers maintain their state through sleep.

11.3.7.1 Working Register Address Space

Working registers are accessed during the normal operation of the block and include accumulators, data registers, FIFOs, status and control registers, mask register, and the auxiliary control register. The following figure shows the registe-r map for one UDB.



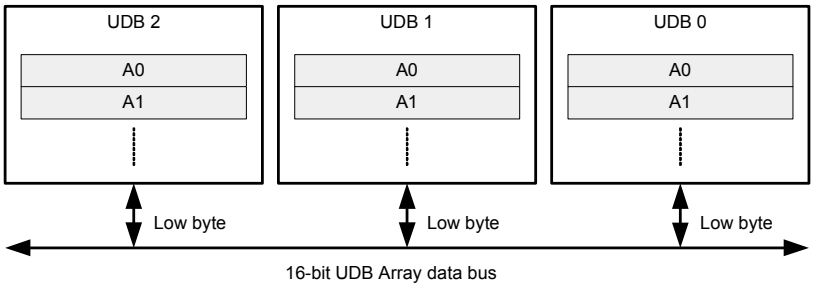
Note that UDBs can be accessed as 8- or 16-bit objects and each of these access methods has a different address space.

On the left, the 8-bit address scheme is shown, where the register number is in the upper nibble and the UDB number is in the lower nibble. With this scheme, the working registers for 16 UDBs can be accessed with an 8-bit address.

On the right is the 16-bit address, which is always even aligned. The UDB number is 5 bits instead of 4 due to the even address alignment. The upper 4 bits is still the register number. A total of 9 address bits are required to access 16 UDBs in 16-bit data access mode. Working registers are organized into banks of 16 UDBs.

11.3.7.2 8-Bit Working Register Access

In 8-bit register access mode, all UDB registers are accessed on byte-aligned addresses, as shown in the following figure. All data bytes written to the UDBs are aligned with the low byte of the 16-bit UDB bus.

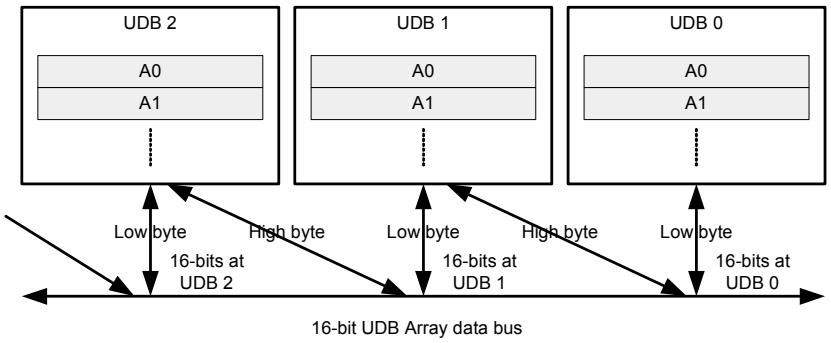


Only one byte at a time can be accessed in this mode and the PHUB will naturally align the valid odd (upper) or even (lower) byte back to the processor or DMA.

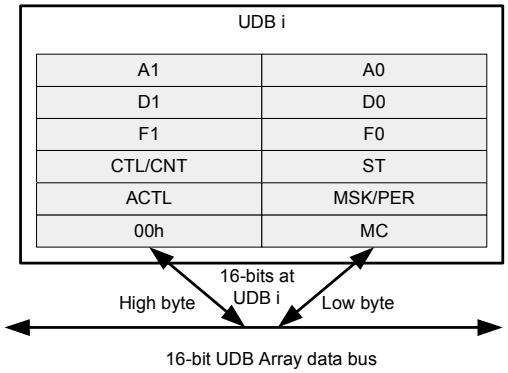
11.3.7.3 16-bit Working Register Address Space

The 16-bit address space is designed for efficient DMA access (16-bit datawidth). There are two modes of 16-bit register access: “default” mode and “concat” mode.

As shown in the following figure, the default mode accesses a given register in UDB ‘i’ in the lower byte and the same register in UDB ‘i+1’ in the upper byte. This makes 16-bit data handling efficient in neighboring UDBs (address order) that are configured as a 16-bit function.



The following figure shows the concat mode, where the registers of a single UDB are concatenated to form 16-bit registers. In this mode, the 16-bit UDB array data bus has access to pairs of registers in the UDB. For example, an access at A0, returns A0 in the low byte and A1 in the high byte.



11.3.7.4 16-bit Working Register Address Limitation

There is a limitation in the use of DMA with respect to the 16-bit working register address space. This address space is optimized for DMA and CPU access to a 16-bit UDB function. It is inefficient for use when the function is greater than 16-bits. This is because the addressing is overlapped as shown in the following table:

Address	Upper byte goes to	Lower byte goes to
0	UDB1	UDB0
2	UDB2	UDB1
4	UDB3	UDB2

When the DMA transfers 16 bits to address 0, the lower and upper bytes are written to UDB0 and UDB1, respectively. On the next 16-bit DMA transfer at address 2, you will overwrite the value in UDB1 with the lower byte of that transfer. To avoid having to provide redundant data organization in

memory buffers to support this addressing, it is recommended that 8-bit DMA transfers in 8-bit working space be used for functions over 16 bits.

11.3.8 DMA Bus Utilization

The DMA controller has a dual context in that they can be pipelined and thus can act in parallel. Generally, the spoke buses can achieve virtually 100% utilization for AHB bus cycles attributed to moving data.

The overhead of processing a channel is generally hidden in the background of data bursting. The arbitrating, fetching, or updating for a channel can occur in the background of the data bursting of another channel. Additionally, the data bursts of one channel can overlap with the data bursts of another channel, provided there is no spoke contention or contention for the source (SRC) or destination (DST) engines of the DMA controller.

11.3.9 DMA Channel Burst Time

Channel burst time is defined as the number of clocks it takes from the first request on the SRC spoke to the final ready on the DST spoke. An ideal burst involves an initial control cycle followed by a burst of data cycles (with subsequent control cycles pipelined in parallel). Thus an ideal burst on a spoke involves N+1 clock cycles to transfer N pieces of data, where N = burst length / peripheral width.

There are multiple variables that can affect this number:

- Existence of another channel context ahead in the DMA controller context pipe and the burst conditions that remain for that channel.
- The burst conditions of the channel:
  - Competition against the CPU for use of the spoke
  - SRC/DST peripheral readiness
  - SRC/DST peripheral widths
  - Length of the burst
  - Ragged starts and ends
  - Intra-spoke vs. inter-spoke DMA

Intra-spoke DMA requires the entire SRC burst to first be buffered in the DMA controller FIFO before the data is then written back out to the same spoke. In that case there are two of these N+1 length bursts that occur, and thus the general formula for an ideal intra-spoke burst is 2N+2.

Inter-spoke DMA allows the SRC and DST bursts to overlap. As data is being read from the SRC spoke and written into the DMA controller FIFO, the DST engine can write available FIFO data to the DST spoke. As a result of this overlapping, inter-spoke DMA is more efficient. The net result is that there are three overhead cycles to move a single piece of data from one spoke to the other. The initial control cycle on each spoke plus one “redundant” data cycle (it takes one data cycle on each spoke to move each piece of data). Thus the general formula for an ideal inter-spoke DMA burst is N+3 to move N pieces of data.

The following table shows the cycle times for some example ideal bursts:

Data Transactions (Spoke-Sized)	Intra-Spoke DMA Burst Phase (Clock Cycles)	Inter-Spoke DMA Burst Phase (Clock Cycles)
1	4	4
2	6	5
3	8	6
N	2N+2	N+3

11.3.10 Component DMA capabilities

Any Component that wants to provide its ability to work with the PSoC Creator DMA Wizard needs to have an XML file that includes its DMA capabilities. The XML format must include the ability to reflect instance specific information. This should be done with a static XML file that can have settings that are based on parameters set for the instance. See Add/Create DMA Capability File on page 91.

Note You cannot add more than one DMA Capability file to a Component.

11.4 Low Power Support

As a general rule, Components provide support for low power by providing APIs to retain non-retention registers and user parameters that would be lost during the exit from a low power mode. An API to save registers and parameters is called prior to entering a low power mode. Then an API is called after exiting the low power mode to restore registers and parameters. The specific registers to save are a function of the registers used in a design. The *TRM* specifies which registers are non-retention. Only the non-retention registers need to be saved when entering a low power mode.

11.4.1 Functional requirements

Provide a static data structure, based on the Component, to maintain the non-retention register values. The low power mode functions are implemented only when necessary. This gives consistent interfaces for all Components. Templates are defined for both the data structure and the functions required to initialize, save/restore, and sleep/wakeup. All of these functions are global. The save/restore functions may be used outside of the low power context.

11.4.2 Design Considerations

Define low power retention functions when necessary. These functions are placed in a separate file, *`$INSTANCE_NAME`_PM.c*. This allows the .o file with the static data structure to be removed at link time if the application does not use the low power functionality. In addition, the `$INSTANCE_NAME`_Enable() and `$INSTANCE_NAME`_Stop() functions enable/disable the alternate active register enables. This provides a mechanism to automatically enable and disable the alternate active template.

11.4.3 Firmware / Application Programming Interface Requirements

11.4.3.1 Data Structure Template

```
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 Save/Restore Methods

Save non-retention register values to the static data structure. Save only the specific Component register values.

```
`$INSTANCE_NAME`_SaveConfig()
{
/* Save non-retention register's values to backup data structure. */
}
```

Restore non-retention register values from the static data structure. Restore only the specific Component register values.

```
`$INSTANCE_NAME`_RestoreConfig()
{
/* Restore non-retention register values from backup data structure. */
}
```

Save the enable state of the Component. Use this state to determine whether to start the Component on wake-up. Stop the Component and save the configuration.

```
`$INSTANCE_NAME`_Sleep()
{
/* Save Component's enable state - enabled/disabled. */
    if(/* Component's block is enabled */)
    {
        backup.enableState = 1u;
    }
    else /* Component's block is disabled */
    {
       backup.enableState = 0u;
    }
    `$INSTANCE_NAME`_Stop();
    `$INSTANCE_NAME`_SaveConfig();
}
```

Restore the Component configuration and determine if the Component should be enabled.

```
`$INSTANCE_NAME`_Wakeup()
{
    `$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 *Additions to Enable and Stop Functions*

Enable the Component's alternate active register enable.

```
`$INSTANCE_NAME`_Enable()
{
    /* Enable block during Alternate Active */
}
```

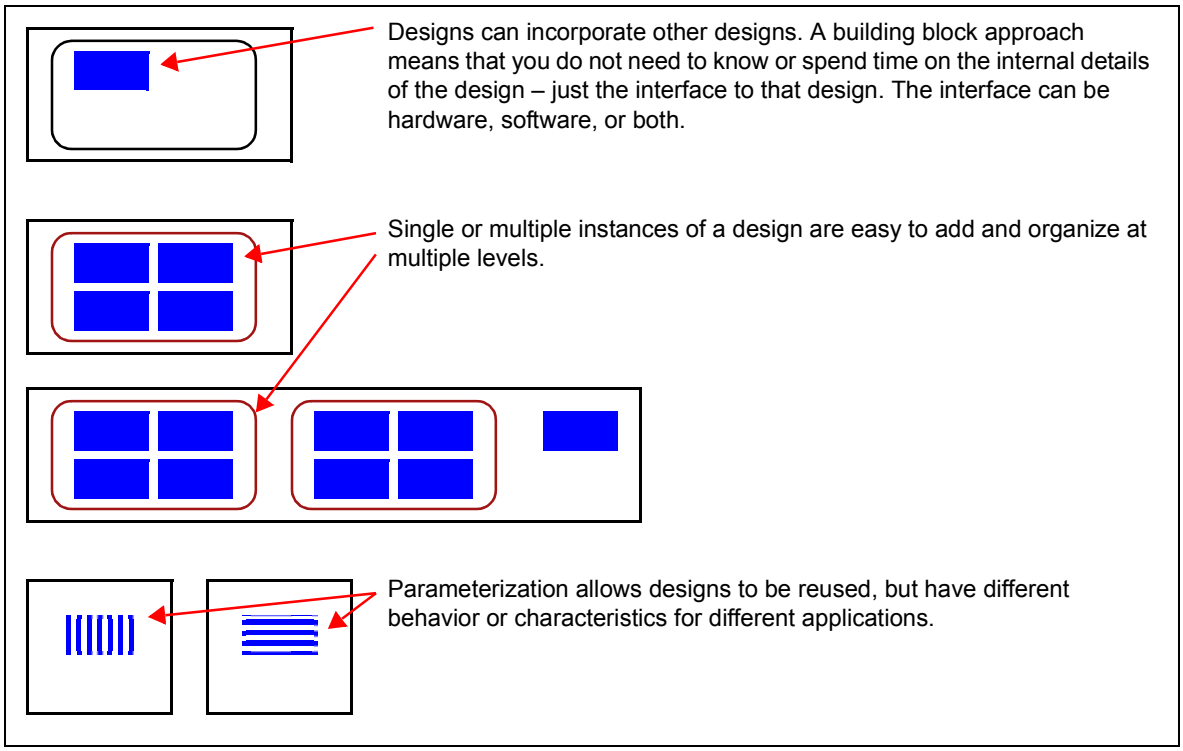
Disable the Component's alternate active register enable.

```
`$INSTANCE_NAME`_Stop()
{
    /* Disable block during Alternate Active */
}
```

11.5 Component Encapsulation

11.5.1 Hierarchical Design

When reusing a design in a hierarchical design system, you can take advantage of the following features:



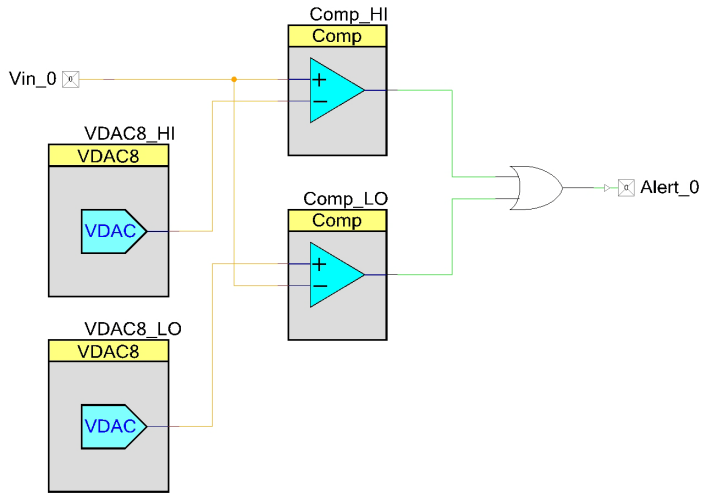
To make it easy to reuse a design with PSoC Creator, it should be encapsulated as a PSoC Creator Component. A design can be considered for encapsulation and reuse if it meets one or more of the following criteria:

- Implements a specific function. The general rule is that it should “do just one thing and do it well.”

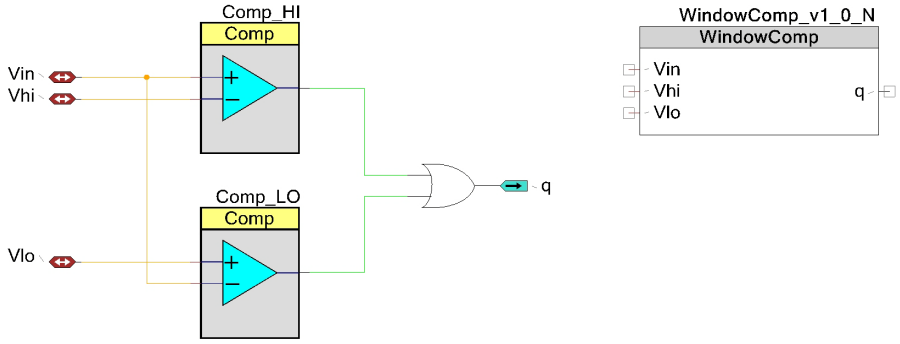
- Has a limited and relatively small set of inputs and outputs, in the form of either hardware terminals or API calls. The general rule is the fewer the better, but not so few that essential functionality is reduced.

The following pages provide a few examples:

In one of the simpler examples of when to encapsulate IP as a Component, consider what you might do if you are required to have a window comparator in your design. A window comparator activates when an input voltage is between two compare voltages. With PSoC Creator you would most likely design it as follows:

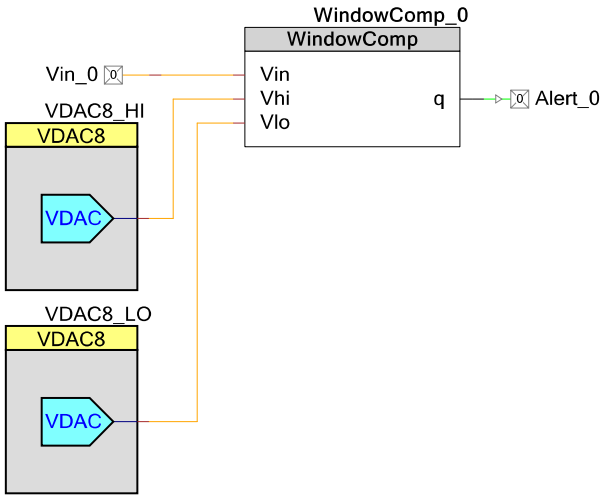


This design is a good candidate for encapsulation as a Component. It implements just one specific function: a window comparator. Plus, it has a limited and small set of inputs and outputs. It would also have a small API to start the comparators. So the basic, essential functionality of the design can be encapsulated into a Component, with a symbol, as follows:

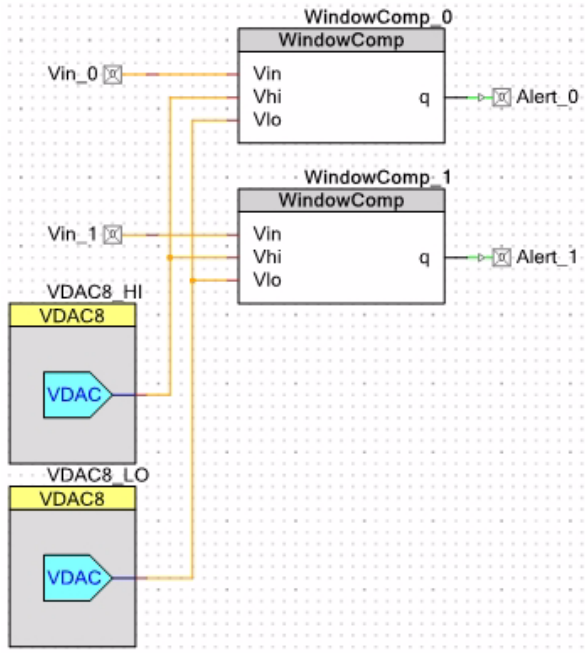


When encapsulating a design, an important decision is what to leave out of the Component. In the above example, the VDACs could have been brought into the Component. However, they really are not part of the essential design functionality, which is simply comparing a voltage to two reference voltages. The reference voltages could be provided by VDACs or by some other source. So in this case it is better practice to leave the VDACs out.

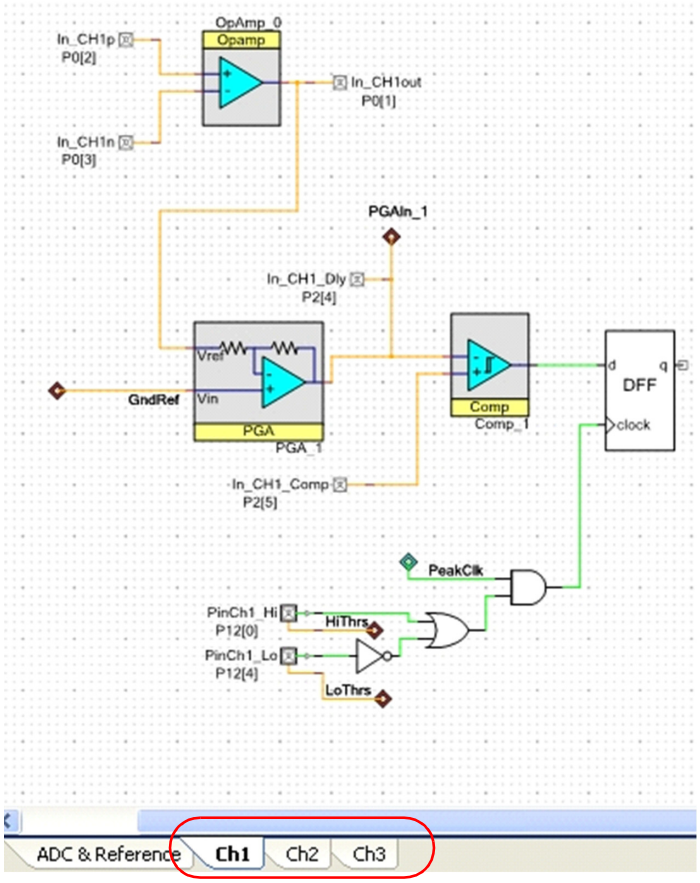
With encapsulation your top-level design becomes simpler:



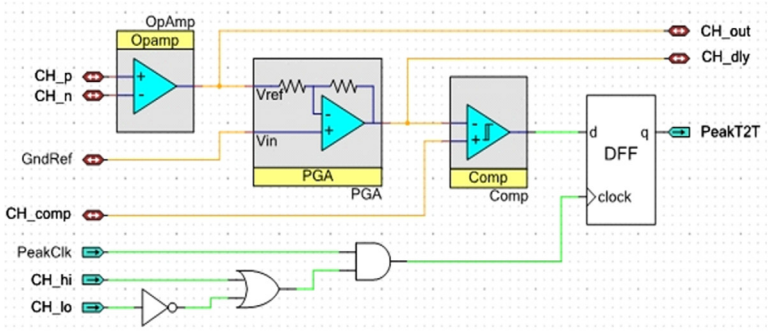
And in many cases it is easier to scale the design:

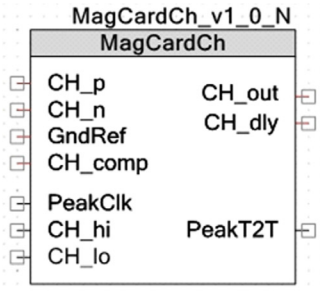


The following the mag card reader example is more complex. In this design, there is a central ADC and reference section along with multiple copies of circuitry for reading a single channel:



The basic design functionality in this case is converting the analog inputs from the channel into a form that is readable by a digital system. Again, because it is a well-defined and limited function it is a good candidate for encapsulation:





Again, you need to look at what to leave out of the Component. The original design uses two SIO pins with different thresholds. These could have been brought into the Component but it is better practice to keep pins at the top-level design. Also, the basic design requires only the digital signals CH_hi and CH_lo, and not necessarily their sources.

Also, since PSoC 3/5 opamps are closely associated with specific device pins, it may actually be better to keep the opamp out of the Component.

Finally, the design makes use of limited analog resources and routing, and multiple instances may not fit in some smaller PSoC 3/5 devices. This fact, along with possible voltage, temperature, or speed limitations should be communicated to the user. For example, what frequency should PeakClk be? It is known at the top level of the original design, but perhaps not known when the Component is reused.

This brings up an interesting issue: when NOT to encapsulate a design. If a design meets one or more of the criteria listed below, it may not be good practice to encapsulate it. If it is encapsulated then the relevant issues and limitations should be communicated such that the user becomes aware of the issues as soon as an attempt is made to reuse the Component.

- Incorporates critical resources in the PSoC 3/5, thereby making those resources unavailable in the higher-level design. Specific topics include:
  - usage of single-instance fixed-function blocks such as ADC_DelSig, CAN, USB, and I^2C
  - too much usage of more abundant resources such as UDBs, comparators, DACs, opamps, timers, DMA channels, interrupts, pins, clocks, etc.
  - too much usage of less obvious resources such as analog or DSI routing, flash, SRAM or CPU cycles
- Operates only under certain conditions, for example: CPU or bus_clk speed, or Vdd levels, or only with certain parts such as the PSoC 5 family.
- Multiple instances of the IP cannot be implemented.

11.5.2 Parameterization

In PSoC Creator, Components can be parameterized. That is, the behavior of an instance of a Component can be set at build time (typically by using a dialog box). Both hardware and software behavior can be set up by parameterization. Parameterization should be used in cases where:

- Different instances of the IP behave slightly differently but the overall functionality is unchanged. For example, an alert output on a fan controller may be set to be active high or active low.
- The differences in behavior are not expected to change at run-time.

If parameterization causes large changes to the functionality of different instances, then you should consider encapsulating the design in multiple Components. PSoC Creator allows multiple Components to be packaged in a single library project, so different versions of the Component can be kept together. For example, a single fan controller Component might have a parameter for number of fans to be controlled. Fan controllers with two different interfaces, like SPI and I2C, might be two separate Components in the “Fan Control” library project.

11.5.3 Component Design Considerations

11.5.3.1 Resources

Components should avoid embedding pin Components. Terminals should be used instead; the user can then connect the pins to the Component symbol at a higher level.

Clock Components may or may not be encapsulated in a Component. The advantage of not embedding a clock is that multiple Components can be connected to the same clock, thereby conserving clock resources. A Component can include a parameterization option to make a clock internal or external.

Interrupts and DMA channels may or may not be encapsulated in a Component. A Component may generate a “have data” signal that could be connected to either an IRQ or a DRQ, in which case the interrupt and DMA should not be embedded in the Component.

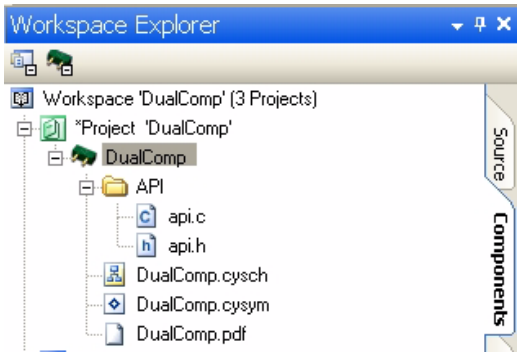
11.5.3.2 Power Management

Components should be designed to support power management issues, such as sleep, hibernate, and wakeup. APIs should include appropriate functions. Cypress-provided Components offer many examples of how to implement a power management API.

11.5.3.3 Component Development

PSoC Creator Components are basically just containers. They cannot be built nor can they be installed in a target device. They are intended to be linked into standard PSoC Creator projects which are then built and installed.

Components can contain several different file types: symbol (.cysym), schematic (.cysch), Verilog (.v), firmware source code (.c, .h, .a51, etc.), and documentation (.pdf, .txt). Components can also reference other Components, thus enabling hierarchical design techniques.



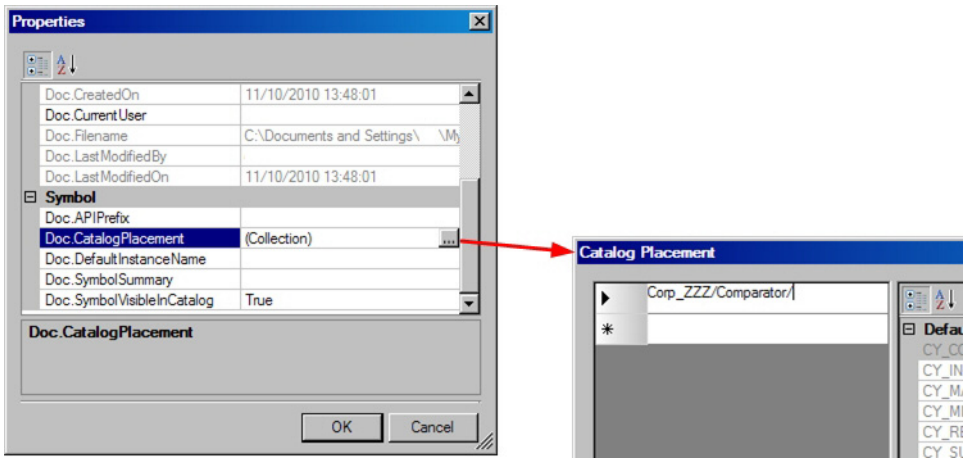
Components developed under this specification should have as a minimum a symbol file and a datasheet file. All other file types are optional depending on the function of the Component. For example, a hardware-only Component could have just a schematic or Verilog file, whereas a firmware-only Component could have just a .h and .c file for an API.

Reference Component Symbol

The symbol should always have an instance name `` `=$INSTANCE_NAME` `` (backward single quotes) annotation included. Other annotations should be added such that the user may more rapidly understand the function of the Component.

Component Catalog Placement

The symbol property Doc.CatalogPlacement controls where the user may find your Component in the PSoC Creator Component Catalog. A consistent tab and tree node naming scheme should be developed, especially if you create multiple Components. Note that using fewer tabs and more tree nodes will create a better fit in the Component Catalog on most users' screens.



For more information, see Define Catalog Placement on page 38.

Component datasheet

To properly document your Component, a datasheet should be included. The datasheet should include the design considerations mentioned above. For information about how to add a datasheet, see Add/Create Datasheet on page 81.

Component Versioning

The Component name can include version information, which is done by appending the following to the Component name:

"_v<major_num>_<minor_num>_<patch_level>"

Both <major_num> and <minor_num> are integers that specify the major and minor version numbers respectively. <patch_level> is a single, lower case, alpha character. Components should be versioned. For information about versioning, see Component Versioning on page 12.

11.5.3.4 Testing Components

Reusable designs are encapsulated as Components in a PSoC Creator library project. However neither a library project nor a Component by itself is very useful. Library projects cannot be built, programmed or tested. So one or more standard projects should be developed along with the Component, for the purpose of testing or demonstrating the functionality of the reusable design in that Component.

The reference Component should be included with each of its parameters set to as many different settings as possible. Note that in order to do this either multiple instances of the Component may need to be used or multiple test / demo projects may need to be created. All functions in the Component's API should be called at least once. All macros should be used at least once.

As much as possible, the test projects should support both PSoC 3 and PSoC 5, in various configurations. For example, standard and bootloadable, debug and release, different PSoC 5 compilers, and different compiler optimization settings.

11.6 Verilog

Many digital Components use Verilog to define the implementation of the Component. For more information, see Implement with Verilog on page 55.

This Verilog must be written in the synthesizable subset of Verilog. In addition to conforming to the Verilog subset, there are additional guidelines that should be followed when creating a Verilog-based Component. Many of these guidelines are practices accepted as best practices for any synthesis tool. Additional guidelines are specific recommendations related to the development of a PSoC Creator Component.

11.6.1 Warp: PSoC Creator Synthesis Tool

When processing any digital logic in a design, PSoC Creator will automatically run the Warp synthesis tool included as part of the PSoC Creator installation. This is a PSoC-specific version of the Warp synthesis tool that has been used in the past with Cypress programmable logic devices.

The specific synthesizable subset of Verilog supported by PSoC Creator is specified in the *Warp Verilog Reference Guide*. This synthesizable subset is similar to the subset implemented in other Verilog synthesis tools. Refer to the guide for a detailed description of the Verilog constructs supported.

The synthesizable portion of the Verilog design will be synthesized into PLD logic. The remaining UDB resources can be instantiated into a Verilog design, but they are never inferred by the synthesis process.

11.6.2 Synthesizable Coding Guidelines

11.6.2.1 Blocking versus Non-Blocking Assignments

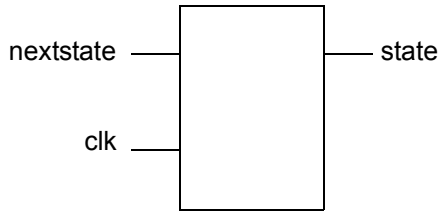
The Verilog language has two forms of assignment statements. The blocking assignment statement assigns the value immediately before progressing to the next statement. The non-blocking assignment statement assigns the value later when time progresses. These two types of assignments have a specific meaning from a simulation perspective. When these assignments are considered from a synthesis perspective and the resulting hardware, the results can differ. The best practices for a synthesizable design were developed so that the simulation results and the synthesis results match.

Implement sequential logic using non-blocking assignments

This rule causes the simulation results of clocked logic to match the hardware implementation. In the hardware implementation all registers clock in new state before that state is used to compute the next state.

```
always @(posedge clk)
begin
```

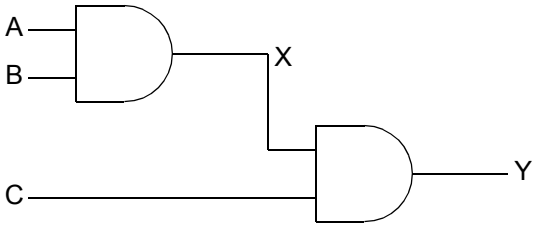
```
    state <= nextState;
end
```



Implement combinatorial logic in an “always” block with blocking assignments

This rule causes the simulation results to assign values to signals immediately as the always block is evaluated. This matches the hardware implementation when the delay through combinatorial logic is ignored. In this case results of one level of logic are used to compute the next level of logic immediately.

```
always @(A or B or C)
begin
    X = A & B;
    Y = X & C;
end
```



Do not mix blocking and non-blocking assignments in the same always block

This rule implies that combinatorial logic that requires more than a single assignment to implement should be described in a block separate from a sequential always block. This combinatorial logic can be implemented in a separate combinatorial block or in a continuous assignment statement.

Do not make assignments to the same variable in more than one always block

The synthesis engine in PSoC Creator will enforce this rule and generate an error if this rule is violated.

11.6.2.2 Case Statements

In Verilog there are three forms of a case statement: case, casex, and casez. The following rules should be followed when coding with these statements.

Fully define all case statements

This is best implemented by placing a default case into all case statements. That will automatically satisfy this rule. Including a default statement even when all cases are already covered will cause

this rule to continue to be satisfied when case-items are changed later in the development of the code. If a case-item is removed or the width of the case is changed, then the default statement is already present.

This rule is particularly important within a combinatorial always block. Including a default condition where the combinatorial result is assigned prevents the synthesis of latches in the design.

Use casez instead of casex

The casex and casez statements are similar and synthesize to the same result. The difference is in the results generated during simulation. The casex statement will match input values of “x” or “z” when a case-item specifies a don’t care bit, but with the casez statement only the “z” value will match don’t care bits in case-items. With the casex statement, simulations can miss uninitialized value design errors. With the casez statement this isn’t a concern since PSoC designs will not have “z” values internal to the synthesizable portion of the design.

Use the “?” for don’t care bits in casez statements

This rule has no impact on the synthesizable code. It is just for clarity to indicate that the intention is that that the specified “?” bits are don’t care. The alternate method where a “z” is used does not indicate the intent.

Do not use casez statements with overlapping conditions

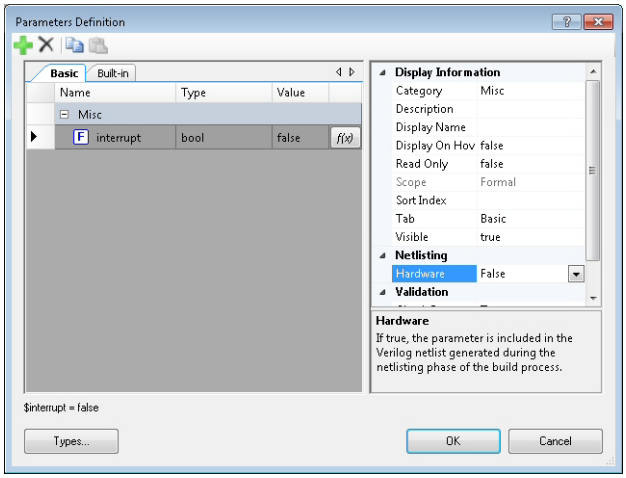
A casez statement with overlapping case-items will result in the synthesis of a priority encoder. The resulting logic is identical to the simulation results, but it can be difficult to determine the intended logic. The use of if-else if-else statements more clearly conveys the priority encoder intent.

11.6.2.3 Parameter Handling

Using parameters allows a Verilog instance to be synthesized based on the specific requirement of a Component instance.

Passing parameters to a Component

The parameters for a Component configured on the symbol can optionally be passed to the Verilog instance by setting “Hardware” in the Misc settings for the parameter to True.



When a parameter is passed to a Verilog module the parameter setting will be applied before synthesis is performed. The result is that the specific instance will be synthesized based on the

parameter settings. This method is appropriate for settings that can be made at build time, but can't be used for settings that will need to be determined at run time. Settings that can be changed at run time will need to be controlled by software through the datapath or control registers. Determining which parameters should be static and which need to be dynamic (run time changeable) will influence the complexity and resource requirements of the Component.

A Verilog based Component that has a Hardware parameter will get access to that parameter value with a parameter statement. This parameter statement will automatically be created in the Verilog template that can be created automatically from the symbol. The initial value will always be overridden with the value set for the specific instance of the Component.

```
parameter Interrupt = 0;
```

Generate statements

Often the value of a parameter will result in the need for significantly different Verilog code based on that value. This functionality can be implemented using a Verilog generate statement. For example if a Component can have either an 8-bit or 16-bit implementation, that will result in the need to instantiate either an 8-bit or 16-bit datapath Component. The only method to implement that functionality is a generate statement.

```
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
```

Passing parameters to module instances

A Verilog design will need to pass parameters to other modules that are instantiated in the design. Parameters are required by many of the standard hardware modules.

There are two ways that parameters can be passed in Verilog. The original method is the defparam statement. In the other method, from the Verilog-2001 specification, parameters can be passed using named parameters. The parameter names and values are included after a "#" in the module instance. It is recommended that parameters always be passed using named parameters.

```
cy_psoc3_statusi #(.cy_force_order(1),
    .cy_md_select(7'h07), .cy_int_mask(7'h07))
stsreg(
    .clock(clock),
    .status(status),
    .interrupt(interrupt)
);
```

Parameterized datapath instances

Datapath instances are configured based on a single complex configuration parameter. For multi-byte datapath configurations or for Components that use generate statements to support multiple possible datapath widths, it is often the case that the same configuration value will be used for

multiple configuration parameters. Duplicating the same information is prone to error and difficult to maintain, so a parameter value can be used so that the configuration data is only present in one place.

```
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 Latches

Latches should be avoided in a PSoC design. Some programmable logic devices have latches built into their architecture. That is not the case for PSoC devices. Each macrocell can be combinatorial or registered on a clock edge. If a latch is created in a PSoC design, the latch will be implemented using cross coupled logic. The use of latches limits the capability to do timing analysis due to the loop construct that is created. The combinatorial latch based implementation can also have timing issues since the length of the feedback paths is not controlled.

It is often the case that when a latch is present in a design, that a latch was not the intended functionality. A latch will be inferred during synthesis for any output from a combinatorial always block where there is at least one path through the block where the output is not assigned a value. This can be avoided by assigning to each output in every if-else clause and including a final else clause in an if-else chain. Alternatively this can be avoided by assigning a default value for each output at the top of the always block.

11.6.2.5 Reset and Set

The following rules should be followed for reset and set signals. The rules and description use the term reset, but this applies to both reset and set functionality.

Use synchronous reset if possible

An asynchronous reset acts like a combinatorial path from the control signal to the output of all registers that it impacts. By using an asynchronous reset, the timing from the reset signal propagated through the logic becomes the limitation on timing. When a synchronous reset is used, then the only additional timing analysis is simply the relationship of the reset signal with respect to the clock.

Use synchronous reset when the clock is free running

The only case where an asynchronous reset should be required is when the clock is not free running. If a register needs to be reset while the clock is not running, then an asynchronous reset signal will be required. If the clock is a free running clock, then the synchronous more of reset will have the same result without the added timing issues of an asynchronous signal.

Asynchronous reset and set can not be used for the same register

The hardware implementation for the PLD macrocells uses a single signal for asynchronous reset and set. A selection is made to use this signal as a reset, a set or not use the signal. The option of having both an asynchronous reset and set is not available in the hardware.

11.6.3 Optimization

11.6.3.1 Designing for Performance

The performance of a typical digital Component is determined by the longest combinatorial path between two registers. The time taken by that combinatorial path is determined by the mapping of the Verilog design to the available hardware.

Registering the dynamic configuration address

The longest path in a design that uses a datapath is often the path starting at a datapath accumulator or macrocell flip-flop that then goes through combinatorial logic (ALU, condition generation, PLD logic), and finally is used as the dynamic configuration address. To increase performance, this long path can often be split into two shorter paths. The natural place to insert a register in this path is often times before the dynamic configuration input. In many cases this input is driven by a macrocell output. Since all macrocell outputs have an optional register, making this output registered does not increase the resources used for the Component. Pipelining in this way does change the operation of the Component, so this type of implementation should be part of the initial architecture definition of the Component.

Registering the conditional outputs

Much like every macrocell in the PLD has an available flip-flop that can be optionally used, each of the conditions generated by the datapath is available as a combinatorial signal or as a registered value. These registers can be used without any resource usage impact to pipeline a design.

Registering outputs

Depending on the typical usage model for a Component, it can be beneficial to register the outputs of the Component. If there is more than one output and they are sent to pins, then registering these outputs will result in more predictable timing relationship between the output signals. If the outputs from the Component are used to feed another Component, then the performance of the system will be dependent on the output timing from this Component and the input timing of the destination Component. If the outputs are registered, the output portion of that path will be minimized.

Split PLD paths

Each PLD has 12 inputs and 8 product terms. If an output requires more inputs or product terms than can be implemented in a single PLD, the equation for the output will be split into multiple PLDs. This will add a PLD propagation delay and routing delay for every additional level of PLD path required. To improve performance any of these levels could be calculated in a pipelined fashion. This can be done without increasing resources since registers are available for every macrocell output. Pipelining will however change the timing relationship for the logic.

To determine the number of inputs that are needed to calculate a specific output all the inputs that are used in if statements, case statements and in the assignment statement for a particular output need to be counted.

11.6.3.2 Designing for Size

Typically the size of the programmable digital portion of a design is either limited by the PLD resources or by the datapath resources. The equivalent logic functionality of the datapath resources is much larger than the logic available in the PLD array. If a datapath implementation is possible, then the that implementation is likely the most resource efficient implementation method. When

building logic for the PLD, the specific architecture of the PSoC PLD must be taken into consideration.

Working with the 12 input PLD architecture

The PSoC PLD has 4 outputs and 12 inputs. When the design is synthesized, the number of inputs required for any one output is limited to 12. If the final output requires more than 12 inputs, then the function is broken into multiple logic pieces where each requires 12 inputs or less. After all the logic has been split into outputs requiring 12 or less inputs, this logic is packed into PLDs. Ideally each PLD will be used to produce 4 outputs. This will only be possible if the equations for 4 outputs can be found that when combined only need 12 inputs. If for example a single output requires all 12 inputs, then the only other outputs that can also be placed in that PLD will need to only require those same inputs.

The first step is to build the Component and observe the average statistics for the packed PLDs from the report file.

```
2629 ------------------------------------------------------------
2630 PLD Packing Summary
2631 ------------------------------------------------------------
2632 Packed 104 macrocells into 51 PLDs.
2633
2634         PLD Resource Type :     Average/LAB
2635     ======================================
2636                    Inputs :           11.31
2637                    Pterms :            2.02
2638               Macrocells :            2.04
```

If the average number of macrocells is significantly below 4.0 and the number of inputs is approaching 12.0, then the number of inputs required is the limiting factor in your design. To improve the packing, the number of inputs required will need to be reduced.

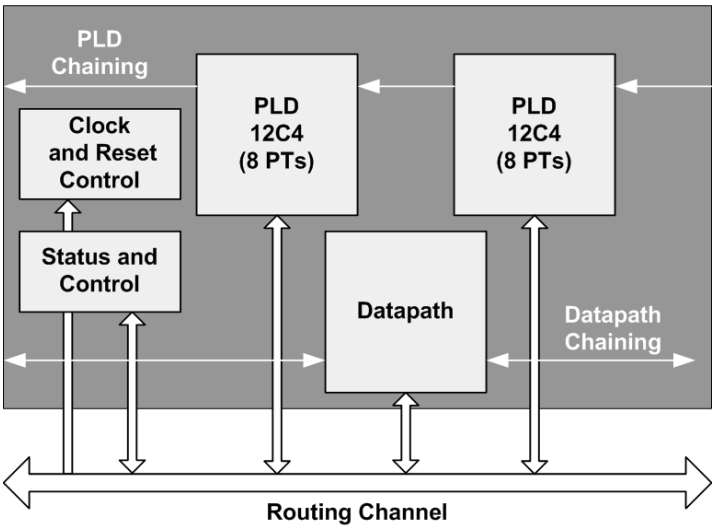
In some cases this can be done by restructuring the equations to consolidate a group of inputs to a single input. For example, if a multi-bit comparison is an input, then that can be replaced by a single result input. If this value is used to calculate multiple outputs (a multi-bit register), then forcing a specific partition of the logic can result in significant size reduction. To force a signal to be the output of a macrocell, use the cy_buf Component. The output of a cy_buf will always be a macrocell output.

```
wire cmp = (count == 10'h3FB);
cy_buf cmpBuf (.x(cmp), .y(cmpOut));
```

11.6.4 Resource choice

A Verilog Component implementation can use various resources that are available in the PSoC UDB architecture. Which resource to choose depends on the functionality required. All digital Components will use PLD logic. Some Components will also use datapath instances. All synthesized

logic will be placed in PLDs. Datapath implementation must be done by including a datapath instance in the Verilog design.



11.6.4.1 *Datapath*

The datapath resource has many times the equivalent logic gate capability as the PLD resource, so if applicable to the application the datapath resource should be the first choice.

Typical datapath applications

- Any operation where a FIFO is required. The two datapath FIFOs are the only FIFO path in the programmable digital system. For some designs the datapath can be used just to add the FIFO functionality to a PLD based hardware design.
- Most counting functions including increment, decrement or stepping by a constant. Registers are available to count, preload, compare and capture.
- Parallel to serial and serial to parallel conversions where the parallel connection is on the CPU side and the serial connection is on the hardware side.

Limitations of the datapath

- Parallel input into the datapath is limited. This restricts the ability to use the datapath where other hardware needs to provide a parallel value. Alternatives to parallel hardware loading may be possible. If enough cycles are available, then a value can be serially shifted in. If the CPU or DMA can get access to the value, then they can write the value to a FIFO or register in the datapath.
- Parallel output is possible, but the parallel output value is always the left input to the datapath ALU. The left input to the ALU can only be A0 or A1, so that limits the output to being A0 or A1, and it restricts the ALU operation that can be performed while using parallel output to also use that value as the left input of the ALU function.
- Only one ALU function can be performed at a time. If multiple operations are required, then a multi-cycle implementation using an multiple of the effective clock may be possible.
- There are 8 dynamic operations available. This is typically enough operations, but for some complex multi-cycle calculations (shift, add, inc) this can be a limitation.
- Only 2 registers are read/writable by the datapath. There are up to 6 register sources (A0, A1, D0, D1, F0, F1), but only 2 registers that can be written and then read back (A0, A1).

11.6.4.2 PLD Logic

The PLD logic resource is the most flexible digital hardware resource.

Typical PLD applications

- State machines
- Small counters (<= 4 bits) or counters where parallel input and output to other hardware are required.
- General purpose combinatorial logic

Limitations of the PLD

- The PLD does not have a direct path from or to the CPU. To get data from the CPU a control register is used. To send data to the CPU a status register is used.
- Maximum number of register bits equal to the number of UDBs * 8 (depends on the selected device). The control and status registers can be used to augment this number of bits, but neither of those resources provides a register that can be both written and read by the PLD.
- 12 input bits per PLD limits the efficiency and performance of wide functions. For example a wide mux function does not map well into the PLD.

A. Expression Evaluator

The PSoC Creator expression evaluator is used to evaluate expressions in PSoC Creator (except those in the debugger). This includes parameter values, expressions in strings, and code generation templates. The expression evaluation language is similar to the Perl 5 language. It borrows most of its operators, including their precedence, but adds a different type system, which is better suited to PSoC Creator application requirements.

A.1 Evaluation Contexts

There are two basic evaluation contexts: a document context, and an instance context. A document context consists of the formal and local parameters from the document, as well as the document properties. An instance context consists of the formal and local parameters of the instance, and the document properties of the referenced SYMBOL.

The evaluation of local and formal parameters was discussed in Formal versus Local Parameters on page 13. Annotations can have embedded expressions. Most annotations are evaluated in the document context. Annotations associated with instances are evaluated in the instance context.

A.2 Data Types

The expression evaluator includes the following first class types:

- bool Boolean (true/false)
- error The error type
- float Floating point, double precision
- int8 8-bit signed integer
- uint8 8-bit unsigned integer
- int16 16-bit signed integer
- uint16 16-bit unsigned integer
- int32 32-bit signed integer
- uint32 32-bit unsigned integer
- string Character string

A.2.1 Bool

Legal values include true and false.

A.2.2 Error

Values of the error type may be created automatically by the system, or by end users. This provides a standard means by which evaluation expressions can generate custom errors.

A.2.3 Float

64-bit double precision IEEE floating point. Written as [+-] [0-9] [.[0-9]*]? [eE [+-]? [0-9]+]

Legal: 1, 1., 1.0, -1e10, 1.1e-10

Illegal: .2, e5

A.2.4 Integers

Sized, signed, and unsigned integers. Integers may be expressed in any of the following three bases:

- hexadecimal – starts with 0x
- octal – starts with 0
- decimal – all other sequences of digits

Unsigned integer literals are written with a “u” suffix.

The following table shows the valid values for each specific type.

Type	Valid Values
INT8	-128 to 127
UINT8	0 to 255
INT16	-32,768 to 32,767
UINT16	0 to 65535
INT32	-2,147,483,648 to 2,147,483,647
UINT32	0 to 4294967295

A.2.5 String

Sequences of characters, enclosed in double quotes (“). Stored as .NET strings internally (UTF-16 encoded Unicode). Currently, only ASCII characters are recognized by the parser. \\ and \” are the only supported escape sequences in strings.

A.3 Data Type Conversion

Due to the Perl 5 inspired nature of the language, all data types are capable of being converted to every other type (with a single exception). The rules for data type conversion are very clear and predicable.

The one exception to the conversion rule is the error type. All data types may be converted to the error type, but the error type may not be converted to any other type.

A.3.1 Bool

Dest Type	Rules
Error	Becomes a generic error message.
Float	True becomes 1.0. False becomes 0.0.
Int	True becomes 1. False becomes 0.
String	True becomes “true”. False becomes “false”.

A.3.2 Error

Dest Type	Rules
Bool	Illegal
Float	Illegal
Int	Illegal
String	Illegal

A.3.3 Float

Dest Type	Rules
Bool	If the value of the float it 0.0, then false. Otherwise, true.
Error	Becomes a generic error message.
Int	Decimal portion is dropped and the resulting integer portion used. If the resulting value does not fit in a 32 bit integer, then the conversion yields 0.
String	Converts the float to a string representation. The precise format of the string is determined automatically.

A.3.4 Int

Dest Type	Rules
Bool	If the value of the int is 0, then false. Otherwise, true.
Error	Becomes a generic error message.
Float	Becomes the nearest floating point equivalent value. Generally this means appending a “.0”.
String	Converts the string to its decimal integer representation

A.3.5 String

Strings are a special case. There are 4 sub-types of strings. These include:

- bool-ish strings have the value “true” or “false”.
- float-ish strings have a floating point value as their first non whitespace characters. Trailing characters that are not a part of the float are ignored.
- int-ish strings have an integer value as their first non-whitespace characters. Trailing characters that are not a part of the float are ignored.
- other

A.3.5.1 Bool-ish string

Dest Type	Rules
Bool	If “true” then true, if “false” then false.
Error	Becomes an error message with the text of the string.
Float	If “true”, then 1.0. If “false” then 0.0
Int	If “true”, then 1. If “false” then 0.

A.3.5.2 Float-ish strings

Dest Type	Rules
Bool	Non-float looking text is dropped and the float part converts exactly like a real float.
Error	Becomes an error message with the text of the string.
Float	Non-float looking text is dropped and the float text is converted to a real float.
Int	Non-float looking text is dropped and the float part converts exactly like a real float.

A.3.5.3 Int-ish strings

Dest Type	Rules
Bool	Non-float looking text is dropped and the float part converts exactly like a real int.
Error	Becomes an error message with the text of the string.
Float	Non-float looking text is dropped and the float text is converted to a real int.
Int	Non-float looking text is dropped and the float part converts exactly like a real int.

A.3.5.4 Other strings

Dest Type	Rules
Bool	If the value is the empty string or “0”, then false. All other strings convert to true.
Error	Becomes an error message with the text of the string.
Float	Becomes 0.0.
Int	Becomes 0.

A.4 Operators

The expression evaluator supports the follow operators (in order of precedence from high to low).

- casts
- ! unary+ unary-
- * / %
- + - .
- < > <= >= lt gt le ge
- == != eq ne
- &&
- ||
- ?:

All operators force their arguments to a well defined type and yield a result in a well defined type.

A.4.1 Arithmetic Operators (+, -, *, /, %, unary +, unary -)

Arithmetic operators force their arguments to a number using the following rules in order:

- If either operand is an error, the result of the operation is the left-most error.
- If either operand is a float or is a float-ish string, then both values are forced to proper floats.
- Otherwise both operands are forced to integers.

If both operands are integers and at least one is unsigned then the operation is performed as an unsigned operation.

Arithmetic operators always yield an error value or a number of the same type as the operands after the operands have been converted using the rules defined above.

A.4.2 Numeric Compare Operators (==, !=, <, >, <=, >=)

Numeric comparison operators force their arguments to a number using the exact same rules as the arithmetic operators. Numeric compare operators always yield a bool value.

If both operands are integers and at least one is unsigned then the operation is performed as an unsigned operation.

A.4.3 String Compare Operators (eq, ne, lt, gt, le, ge)

String compare operators force their arguments to strings, unless either operand is an error. If either operand is an error, the left-most error is the result of the operation. String compare operators always yield an error value or a string.

A.4.4 String Concatenation Operator (.)

The string concatenation operator acts like the string compare operators for the purposes of converting argument values.

Note If the concatenation operator is directly preceded or followed by a digit, then it will be interpreted as a float (e.g., 1. = float; 1 . = string concatenation).

A.4.5 Ternary Operator (?:)

- If the first operand, the bool value, is an error type then the result is an error type.
- If the bool value is true, the value and type of the result is that of the expression between the ? and the :.
- If the bool value is false, then the value and type of the result is that of the expression that follows the :.

A.4.6 Casts

Casts (lowercase: cast) are of the form: cast(type, expr)

Casts evaluate the expression and convert the result of expr to the named type.

A.5 String interpolation

The expression evaluator is capable of interpolating evaluated expressions embedded in strings. The format is:

```
`= expr`
```

The `= marks the beginning of the expression. The = sign is not interpreted as part of the expression. Since the next ` found ends the expression it is not possible to nest `= blocks inside other `= blocks. Multiple expressions, provided they aren't nested, may be embedded in the same string and they will all be evaluated and interpolated. There is only one evaluation pass over the embedded expression. If the resutling string is a legal expression it will not be evaluated as an additional step.

The portion of the string from the starting ` to the ending ` is replaced with the result of the expression between the ` and `.

A.6 User-Defined Data Types (Enumerations)

From the perspective of the expression system user-defined types are simply integers. User-defined type values can appear in expressions but they are converted to the integer equivalent before being evaluated in the expression. See Add User-Defined Types on page 36 for information about how to add user-defined types to your symbol.