

Internal Components

Internal Components	1
Introduction	2
Schematics	3
PIN	5
Clock	7
Frequency Divider	10
Control Registers	11
Status Registers	12
Logic Gates	13
PWM	14
Digital Multiplexer / Demultiplexer	15
Analog –to-Digital (ADC) Converter	16
CapSense Fundamentals	17
CapSense Button	21
CapSense Slider	23
CapSense Tuning	25

Introduction

The PSoC family of chips have a rich set (>100) of internal components – virtual ICs – that enable the PSoC to do its magic. These components are a combination of configurable hard blocks (e.g. the analog to digital converter), soft blocks (synthesizable general purpose logic that is put into the Universal Digital Blocks [UDB]), and Firmware (C-programs that enable the microprocessor to control the registers and flow of data). Components hide the complexity of PSoC from the user by putting these three things together into schematic elements with a simple GUI (configuration wizard) to configure them.

To use a component simply pick it from the Component Catalog (on the right side of the screen) and drag it onto your schematic.

You may search for a specific component by typing in your search criteria into the search box at the top of the Component Catalog (see where I typed “pin”).

There are two tabs on the Component Catalog

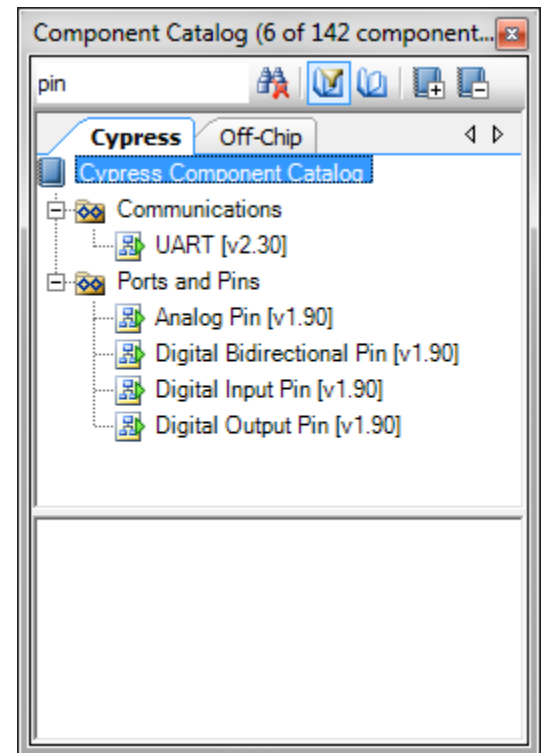
- “Cypress” – Internal Components
- “Off-Chip” – External Components

To configure a component on your schematic simply double click the component to bring up the configuration wizard.

When you place a component, Creator automatically gives it a name. The name is the “key” which you will use to access the component from the firmware. You should pick a name that makes sense in your design (seriously pick a name that makes sense). All of the firmware Application Programming Interface (API) functions will start with the name of the component (e.g. the clock component which you named “clock7” will have a function called “clock7_Start()” which you will use in your firmware to “start” the clock ticking).

Every component has a datasheet. The datasheet is a complete description of how the component works, how to access it etc. You can access the data sheet by:

- right clicking the component in the schematic and selecting datasheet
- clicking the datasheet tab from the “Workspace explorer”
- Using the document manager from the help tab
- Searching on Cypress.com

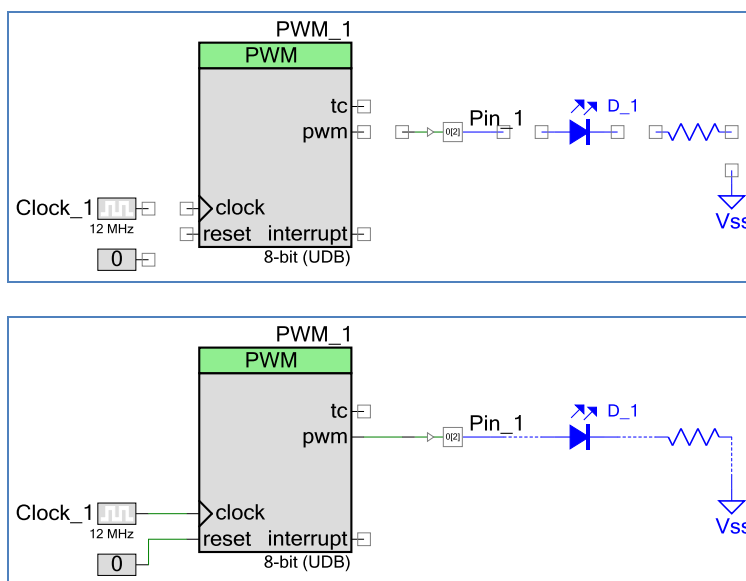


Schematics

Once components (both internal and external) are placed in the schematic, it is necessary to draw wires to connect them together. To do this use the wire tool on the left side of the schematic window or just press the key “w” when you are in the schematic, and then click along the path that you want the wire to follow.



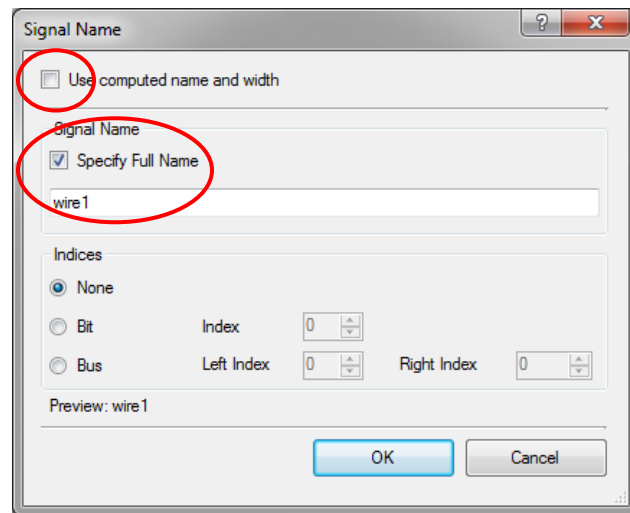
As a general rule, any square box terminal in your schematic should be connected to a wire – after wires are connected, the box disappears. The exception to this is that outputs from components do not have to be connected to anything if they are not used. The following shows an example of a schematic before and after components have been wired together. Note how the square boxes are not seen once things have been wired properly.



You can have more than one page of schematics in PSoC Creator – just right click on “Page 1” at the bottom of the schematic to add additional pages. You can also rename the pages so that they have more descriptive names than “Page 1”, “Page 2”, etc. This allows you to group elements together by function.

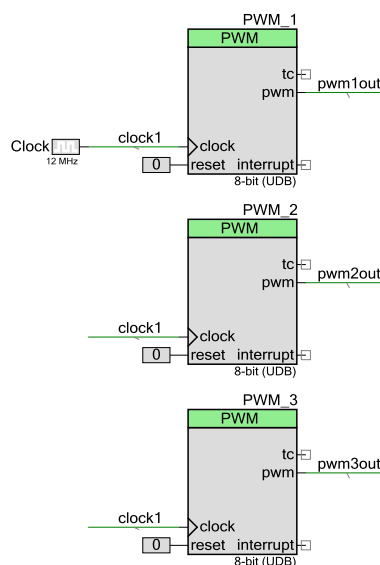
Starting with Creator 3.0, you can also choose “Disable Page” which will effectively comment out everything on that schematic page. This is very useful if you group components that are used for debugging on a single page.

You can connect wires between different schematic pages by naming them (double click on the wire and chose to specify the full name) and using the off-sheet connector symbol in your schematic. For example, the following will allow “wire1” to be connected together between different schematic pages.



Off-sheet connector symbol

Any two wires with the same name on the same page are automatically connected together even without off-sheet connectors. This is useful for things like clocks that go to many places. This can allow your schematic to be less cluttered. For example here are 3 PWMs with the same input clock connected by name:



PIN

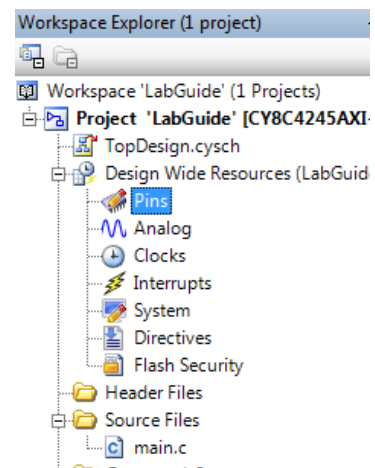
Description: The pin components are the interface between the internal PSoC components and the external components (outside world). PSoCs have four types of pins:

- Digital Output
- Digital Input
- Digital Bidirectional (which means it is switchable between Input/Output)
- Analog (can be either an input or an output)

Note that a pin can be more than one thing at once. For example it is possible to have a pin that is both a digital input pin and an analog pin at the same time.

Every schematic pin has a user selectable “name” which allows your PSoC firmware to address the pin. You should name the pin something that makes sense for your design – for instance “POT1” if you are connecting to a potentiometer (hint: really, name the pin something sensible).

On your Creator schematic, the pin is “virtual”, that is, it just has a name, but you don’t know what actual electrical PSoC pin the “schematic pin” is connected to. This is one of the coolest features of PSoC because it allows you to easily map and remap pins in your design to different electrical pins (say if the PCB needs to change for some reason). Other microcontrollers cannot in general perform this function. To connect the schematic pin to an electrical pin you need to open the Design Wide Resources (also called the DWR) by double clicking Pins in the Workspace Explorer. This will bring up the window shown here which will allow you to connect your pin name to a specific port on the chip using the provided drop-down menu. You should be aware that Creator will automatically assign the PIN if you don’t – which can sometimes make for a surprise.



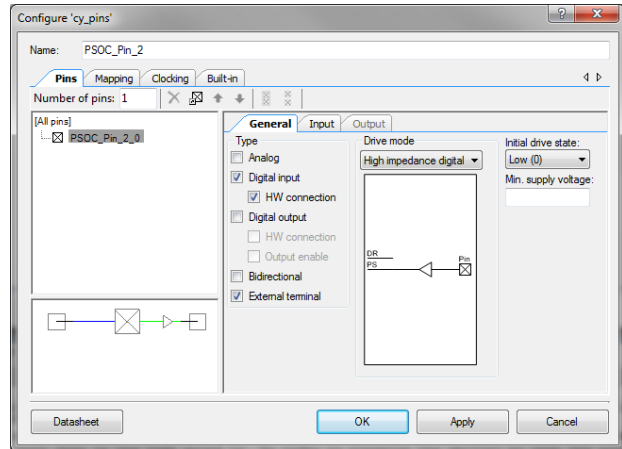
Note that we typically assign using the “Port” rather than “Pin”. These map to each other so you can do either one, but it is easier to find the port name written on a kit or in the documentation than it is to find the pin number. Once you set either the Port or Pin column, the other will fill in automatically.

Pins will also automatically Lock once you set them so that they will not change location unless you intentionally change them.

Name	Port	Pin	Lock
BLUE_LED			
Pin_1			
Pin_2			
PSOC_Pin			
PSOC_Pin_4			
PSOC_Pin_1			
PSOC_Pin_2			
PSOC_Pin_3			
RED_LED			

There are two important settings on the pin configuration:

- **HW Connection** (green wire) – If you have a firmware only pin (it doesn't connect to any schematic components) then you will need to disable this connection. Otherwise Creator expects it to connect to something on the schematic. Inputs and outputs each have a checkbox for this.
- **Show External Terminal** (blue wire) – this will enable a place on the pin for you to connect External Components.



Firmware APIs

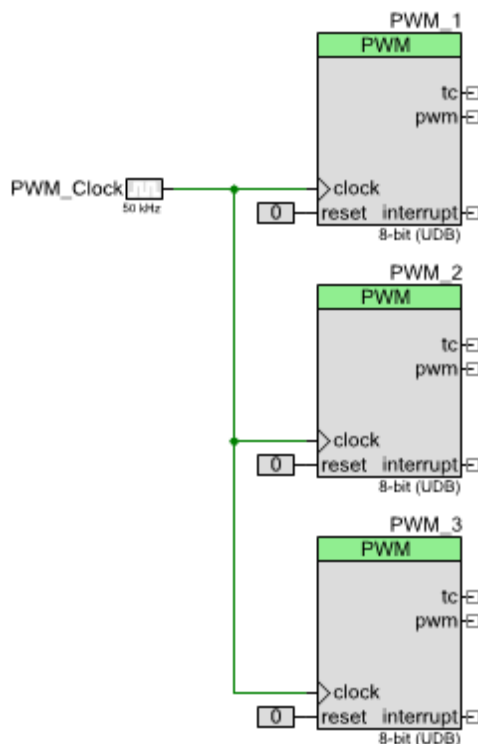
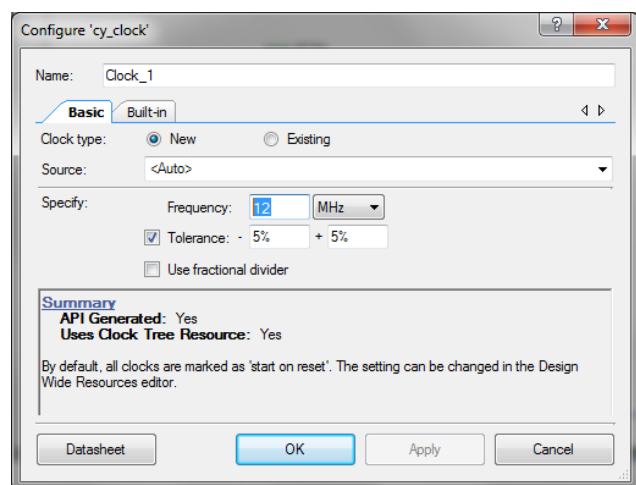
If you have the HW connection disabled (that is if you want to control an output pin or read an input pin from the firmware, then you will use the following API functions:

Output Pins: `<pin_name>_Write(<value>);`
 `<pin_name>` is the name that you give to the pin in the cusomizer
 `<value>` is the value that you want the pin to drive – “1” for high and “0” for low

Input Pins: `<return> = <pin_name>_Read();`
 `<pin_name>` is the name that you give to the pin in the cusomizer
 `<return>` is the value that is read from the pin – “1” for high and “0” for low

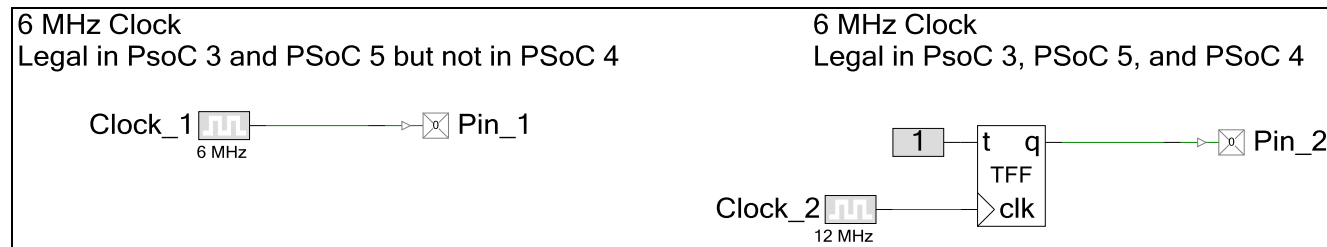
Clock

Description: Clocks are a fundamental component in PSoC projects. Almost all digital designs are driven by a clock and PSoC offers a lot of flexibility in how you set them up. The most important aspect of a clock is, obviously, its frequency. You can choose any frequency you like in the Clock component dialog and the tool will figure out the best signal that the device can provide.



Clock Resources: The clocks are generated from a specific block within the PSoC device and so there is a finite number of frequencies you can create. If you use up too many clocks the tool will tell you and you should look for ways to share them in the design. You can, for example, connect a clock to any number of PWMs, as shown above.

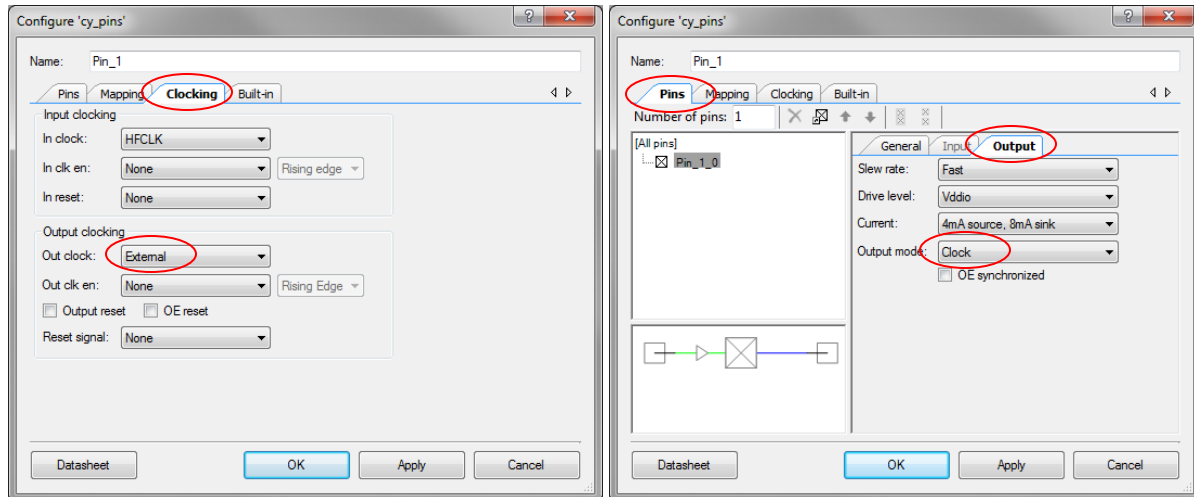
In PSoC4 you can only connect clocks to clock inputs on components (look at the above circuit). For instance this means you cannot connect a clock directly to a multiplexor. A technique that will enable you to connect a clock to a logic signal is to use a T-Flip Flop as a divider (**note** 12 MHz Input below is divided by 2 by the T-Flip Flop).



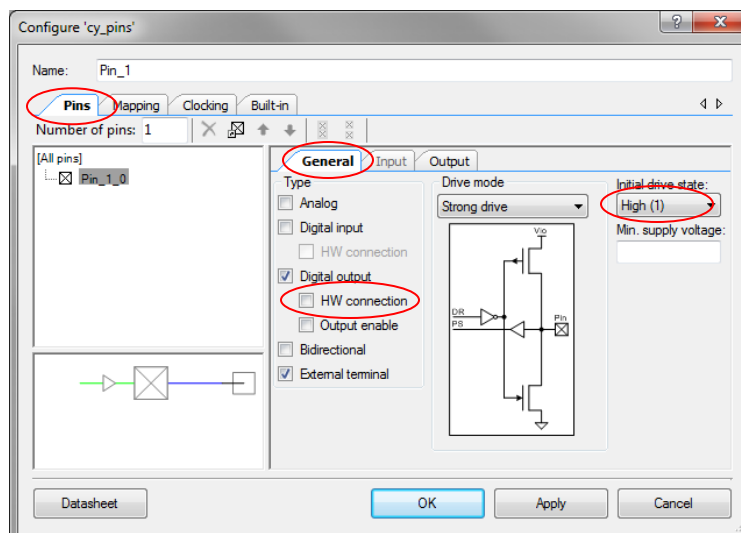
It is possible in PSoC 4 to connect a clock directly to an output pin using the following pin settings:

Set the “Out Clock” to “External” on the “Clocking” tab. This will allow you to connect any desired clock to the pin.

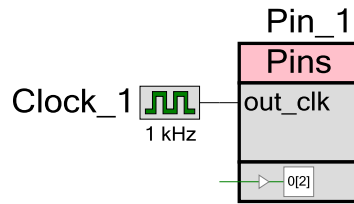
Set the “Output Mode” to “Clock” on the “Pins -> Output” tab. This will cause the pin to output the clock instead of the pin’s data value.



In this case, the data value for the pin is used as an enable. That is, the data value for the pin must be high for the pin to toggle. You can use this terminal as a gate to turn the clock on/off from the schematic if desired. If you want the clock to be free-running (or firmware controlled), you should turn off the “HW connection” and make sure the initial pin state is set to “High” on the “Pins -> General” tab.



As an example, a free-running 1 kHz clock routed to a pin on PSoC 4 would look like this:



The API that is used to change the clock frequency from the firmware is “Clock_SetDividerValue”. This function takes a 16-bit value so the clock divider cannot be set to a value larger than 65535. Therefore, on PSoC 4 if the HFCLK is set to the default value 24MHz the lowest frequency clock that you can get by setting the divider in firmware is 366.2Hz ($24,000,000 / 65535$). If you need a smaller value, you can do one of the following:

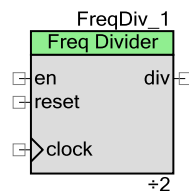
1. Set the HFCLK to a lower value. For example, an HFCLK Of 3MHz would allow you to create a clock as low as 45.8Hz using the set divider API. The minimum value for HFCLK on PSoC 4 is 3MHz so values lower than 45.8Hz are not possible using this method.
2. Use the “Frequency Divider” component (see the next section for a description).
3. Use a T Flip Flop (or a series of them) to divide the clock as shown earlier. Remember that each Flip Flop in series will divide the clock by 2.
4. Use a PWM to divide the clock. A 16-bit PWM at the maximum period will divide the PWM’s input clock by 65535 and an 8-bit PWM at the maximum period will divide the input clock by 255. For example, you could use a 10KHz clock as the input to an 8-bit PWM and then divide it down as necessary to achieve frequencies from 5KHz to 39.1Hz. You will typically want to set the compare value to $\frac{1}{2}$ the period to get a 50% duty cycle for the resulting output clock. The period and compare value can both be changed in firmware.

Frequency Divider

Description: This component is a convenient way to divide down the frequency of a clock.

This component takes a 32-bit divider value so the clock can be divided by any integer from 2 to 4294967295. With the maximum value, a 24MHz clock would be divided down such that it would take almost 3 minutes to complete one clock cycle!

The divider cannot be changed in firmware for this component. If you need to change the frequency in firmware you can change the input clock's frequency, use multiple frequency dividers that are multiplexed together, or use a PWM (see the PWM section).



The en (enable) and reset inputs can be used to control the output clock. When en is low, the output will stop switching (it may stop in either the high or low state). When reset is high, the output clock will stop and will be low. If you want the divider to always run, tie en to a logic high and tie reset to a logic low.

Notice that the symbol shows the divide value on the lower right corner (2 in this case).

The number of digital resources (UDB macrocells to be exact) used depends on the divide value chosen. Larger divide values will require more resources (the resources required can be found in the datasheet). Therefore, if you require very large divider values, it might be better to use a different method such as a PWM.

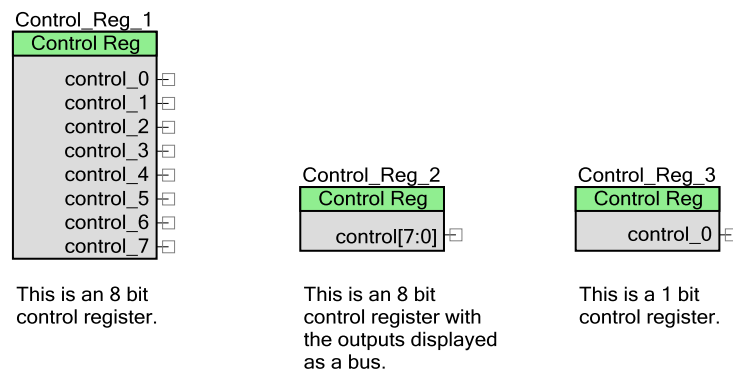
Control Registers

Description: Control registers are the means by which firmware can drive digital signals on the PSoC creator schematic. They can provide access for 1 to 8 signals.

The control register has an API that lets you write a number into the register that corresponds to a signal in the UDB array. Write 0x01 (hexadecimal 1) to set bit 0 high. Write 0x02 to set bit 1, 0x04 to set bit 2, 0x08 to set bit 3. Add the numbers up to set multiple bits, e.g. 0x0A will set bit 1 and 3, clear 0 and 2. Writing 0xFF will set all bits in an 8-bit register.

The outputs can be direct (set immediately), synchronized to a specified clock, or pulsed. See the component datasheet for details.

Schematic: Multi-bit registers can display the output terminals as individual wires or a bus (thick-lined wire).



Register Resources: The Control Register component uses one control cell in the UDB array. Note that each instance consumes a whole cell (8-bits), regardless of the size of the register chosen.

Firmware APIs

To set the value of a control register:

```
<reg_name>_Write(<value>);
```

<reg_name> is the name that you give to the register in the customizer

<value> is the value that you want to put in the register. For example:

0x00 will make all outputs of the register 0 (it will clear all bits in the register)

0x01 will make control_0 a 1 and will make all other outputs of the register 0

0xFF will set all 8 control outputs of a control register to 1

To read the value of a control register:

```
<return> = <reg_name>_Read();
```

<reg_name> is the name that you give to the register in the customizer

<return> is the value currently stored in the register

Status Registers

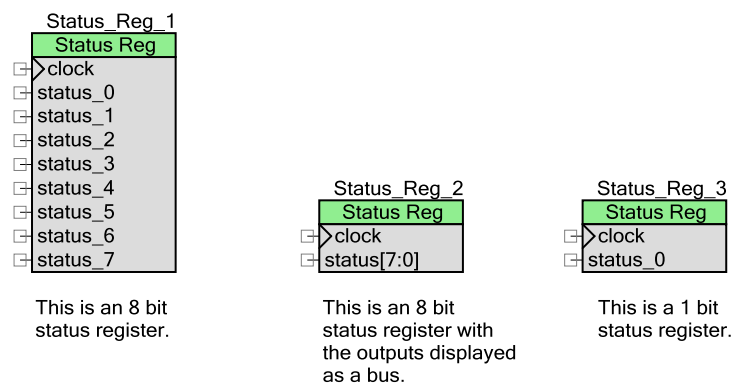
Description: Status registers are the means by which digital signals on the PSoC creator schematic can be monitored by firmware. They can provide access for 1 to 8 signals.

The status register has an API that lets you read the value of the connected signal(s). A value of 0x01 (hexadecimal 1) means bit 0 is high. A value of 0x02 means bit 1 is high, 0x04 means bit 2 is high, and 0x08 means bit 3 is high. A value of 0xFF means that all 8-bits are high.

The status register must be clocked so that the register is updated at the correct time so always attach a clock component to the input (the clock speed you should use depends upon how fast/often you want to read the signal).

The values in the status register can either be “Transparent” or “Sticky”. Transparent values will always reflect the state of the input (at the latest rising clock edge) while sticky values will remain set until they are cleared by the firmware. Therefore, sticky status register values are good to detect an edge triggered event.

Schematic: As with control registers, multi-bit status registers can display the terminals as individual wires or a bus (thick-lined wire).



Register Resources: The Status Register component uses one status cell in the UDB array. Note that each instance consumes a whole cell (8-bits), regardless of the size of the register.

Firmware APIs

To read the value of a status register:

```
<return> = <reg_name>_Read();
```

<reg_name> is the name that you give to the register in the cusomizer

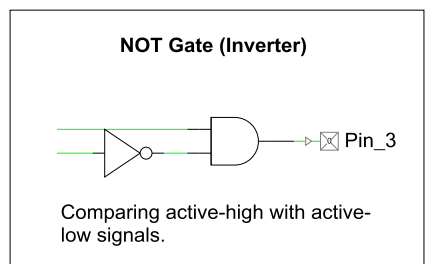
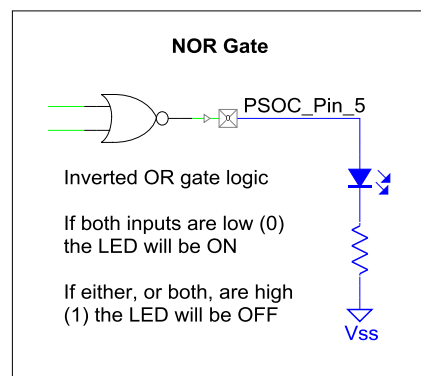
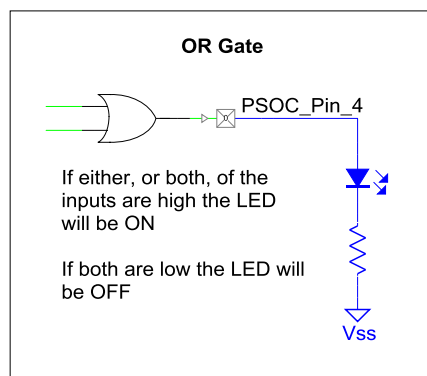
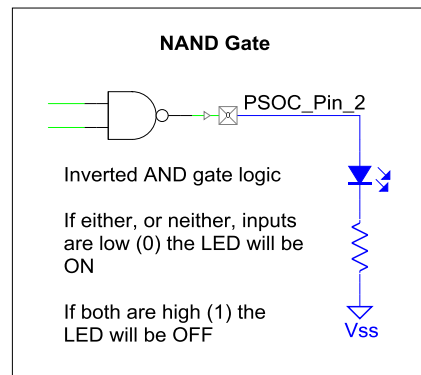
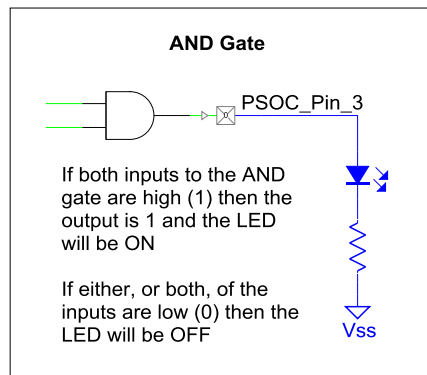
<return> is the value currently stored in the register

Logic Gates

Description: Logic gates allow you to manipulate digital signals. One of the most common uses is to do Boolean algebra. They are very simple components that use very few resources.

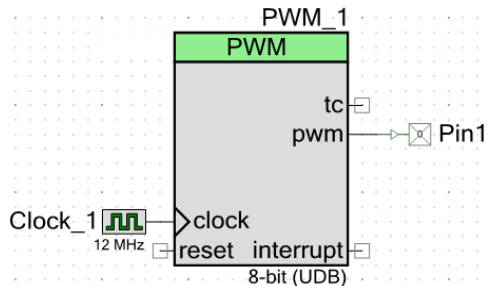
- AND / OR / XOR gates
- NAND / NOR XNOR gates
- Not gate (signal inverter)
- Logic High (1) and Logic Low (0)
- Multiplexer (Mux)

Schematic: Logic gate components use the industry standard images, not the typical rectangles. Note that, by default, there are always two inputs to logic gates (four for Muxes) but you can increase that number with the parameter editor dialog.



PWM

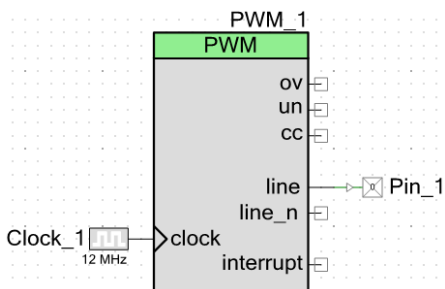
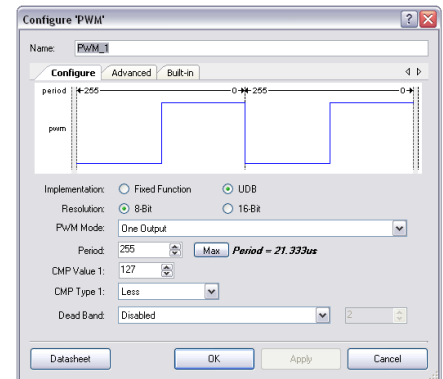
Description: The Pulse Width Modulator is a counter with a compare value. The component will count from a specified number to 0. It can run continuously or require a trigger to run. There can be one or more outputs from the same PWM component. The outputs are configurable based on what the



current count is with respect to a specified compare value. By changing the compare value relative to the period value, we can change the percentage of time the output is high compared to the period – this is called the duty cycle. For example a 50% duty cycle would mean the output is high half the time and low half the time. To accomplish this, the compare value should be $\frac{1}{2}$ the period value.

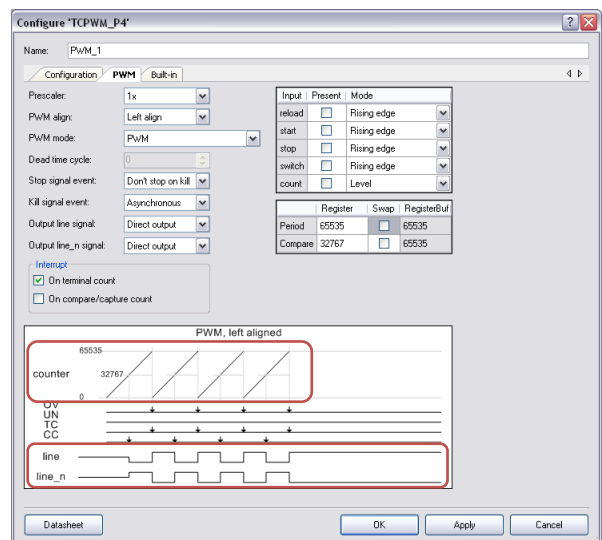
The speed is based off an input clock. A PWM is implemented using a counter fixed function block or using a datapath in a UDB.

In PSoC4, the PWM can be implemented using the TCPWM fixed function block. By using this block you save UDB resources which may be required for other tasks. In this case the block and configuration dialog look very different.



The output is line, or line_n. An output ending in “_n” means negative or opposite of what you’d normally expect. The configuration dialog has a helpful graph to show you the count over time, the compare value and the resulting output (line and/or line_n). Switching PWM align from left align to right align, the counter will count down instead of up.

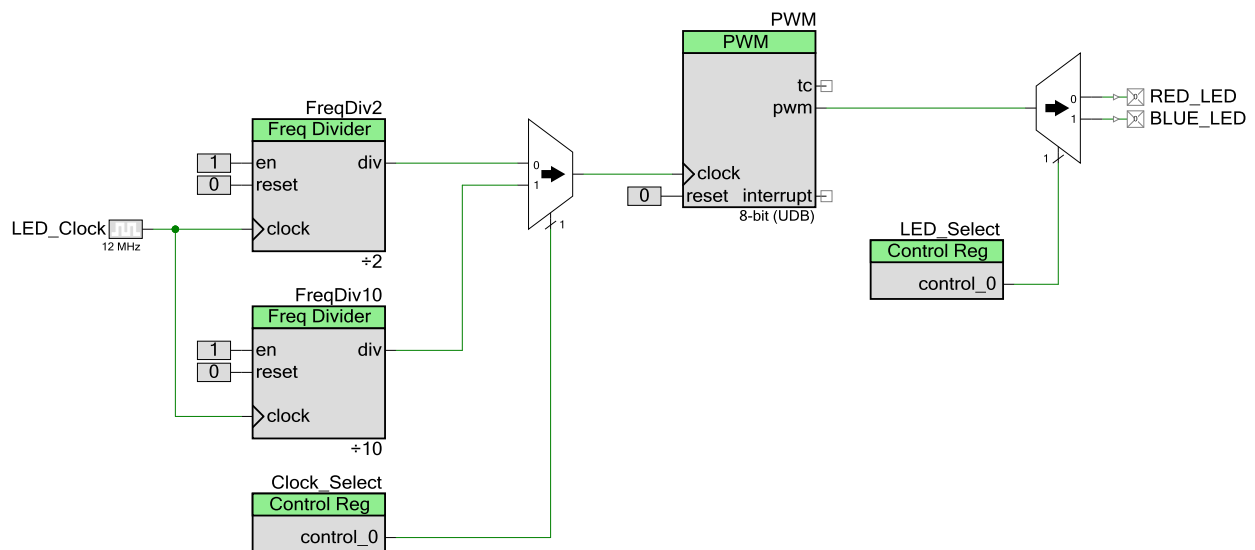
The APIs differ between the PSoC4 TCPWM and the UDB PWM. See the respective datasheet for details on how to set the period or compare value.



Digital Multiplexer / Demultiplexer

Description: The Mux and Demux allow you to change inputs (Mux) and outputs (Demux) at run-time. These components accept a number of signals on one side but only allow one to pass to the other. Another signal to the Mux is used to choose the active signal. Note that these components are directional – you cannot flip a Mux around to make it a Demux!

It is very common to “drive” a Mux from a control register, allowing you to choose the active signal from software. In the following example a Mux is used to choose the speed of the clock input to the PWM and a Demux is used to choose which pin (LED) is driven.



Mux Resources: The Multiplexer and Demultiplexer are implemented with logic equations and therefore are synthesized and mapped into PLD blocks within the UDB array. The component size and width determines the size of the logic equations and thus the number of PLDs used.

Analog-to-Digital (ADC) Converter

Description: An Analog-to-Digital Converter is a PSoC hardware component that can read an analog voltage from a pair of analog pins and convert it into a digital value which the PSoC CPU can read and interpret. The number of bits of the ADC will determine the resolution of the output. For the PSoC4 a 12-bit ADC the digital value will be

- $0 \rightarrow 2^{12} = 0 \rightarrow 4096$ for single ended
- $-2^{11} \rightarrow +2^{11} = -2048 \rightarrow +2048$ for differential

The input voltage range of the ADC is selectable. 0-5v or 0-1.024 or 0-3.3v or +-5v or +-3.3v or +- 1.024. The digital output value is known as “counts”. Each “count” represents a fraction of the total input range. For example: If the input range is +-5v (10v total) and the number of bits is 12 then each “count” count = $10v/(2^{12} \text{bit}) = 0.002441v/\text{count}$ which is also 409.668 counts per volt

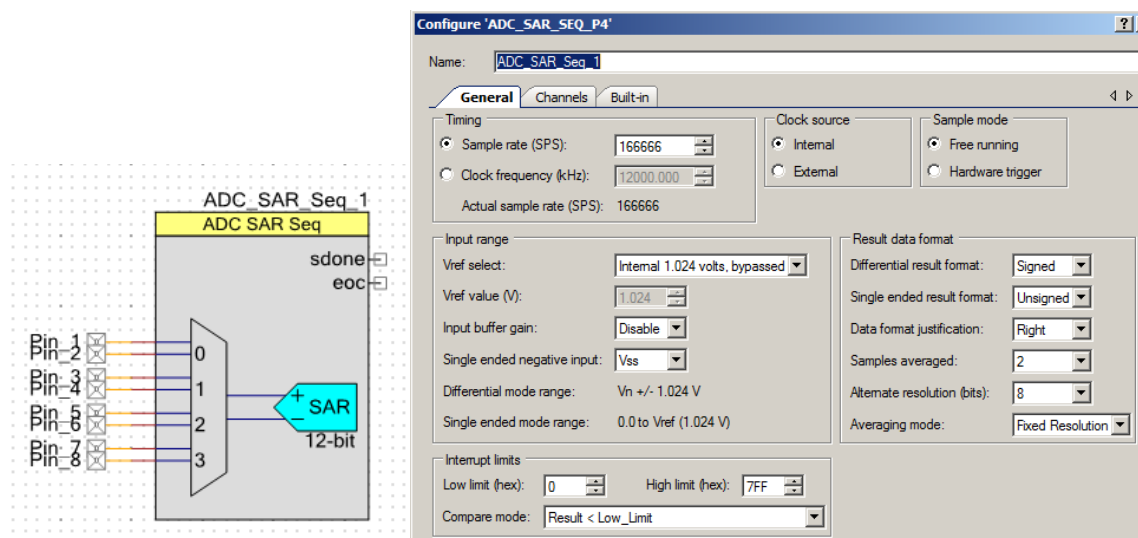
Example: input voltage = 2v then the counts = $409.668 * 2 = 819 \text{ counts}$

Creator can also give you a “single ended” measurement by automatically connecting one of the two analog pins to ground.

The Cypress PSoCs have two different kinds of ADCs

- Successive Approximation Register (SAR) – in PSoC3/4/5
- Delta-Sigma or Sigma-Delta (which way you call it is a religious war)

With PSoC4 there is an analog multiplexer that allows the user to read up to 8 inputs (or 4 differential inputs). Configure this on the “channels” input of the configuration wizard.



CapSense Fundamentals

Description: CapSense is Cypress’s capacitive-sensing solution that “just works” in noisy environments and in the presence of liquids. Most of the Cypress PSoC chips have an embedded (internal) CapSense block that is capable of simultaneously reading multiple buttons, sliders (see the CapSense Slider section), and other “widgets”. Cypress CapSense supports both self-capacitance and mutual-capacitance sensing and even dynamic self-cap and mutual-cap sensing to get the best of both methods. For example, mutual-cap sensing allows multi-touch detection and works better with large parasitic capacitance traces while self-cap sensing provides proximity detection and water tolerance.

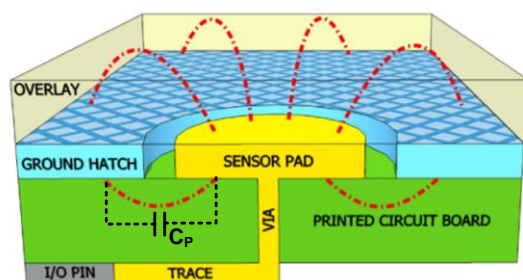
Feature	Self-Cap	Mutual-Cap	Dynamically Switching Self-Cap and Mutual-Cap
Number of GPIOs	One GPIO per button	One GPIO per Tx and one GPIO per Rx	One GPIO per Tx and one GPIO per Rx
Touchpad	Yes	Yes	Yes
Touch Gestures	Two fingers	Multi-Touch	Multi-Touch
Proximity Sensing	Yes	No	Yes
Proximity Gestures	Yes	No	Yes
Liquid Tolerance	Yes	No	Yes
Sensor Parasitic Capacitance	≤ 60 pF	≤ 200 pF	≤ 200 pF
Typical Applications	Mechanical button replacement Proximity Sensing Low-Cost Touchpad	Multi-Touch touchpad Touchscreen Liquid-Level sensing	Mechanical button replacement Proximity sensing Touchpad Touchscreen Liquid-Level sensing

Component Versions

On some devices there are multiple versions of the CapSense component. In general, you should always use the latest version (highest version number) since it will have the best performance and most features. The old versions are only kept for legacy purposes (i.e. for existing designs).

Self-Cap

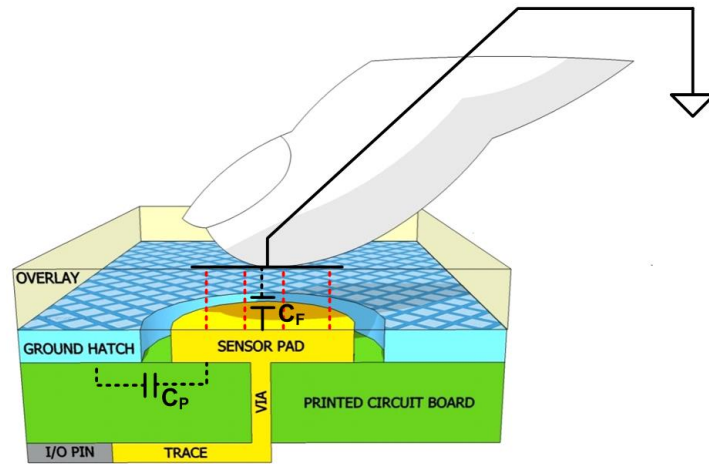
A typical self-capacitance based CapSense sensor consists of a conductive (usually copper or indium tin oxide) pad of proper shape and size, laid on the surface of a non-conductive material like PCB or glass. A non-conductive overlay serves as the touch surface for the button while PCB traces connect the sensor pads to PSoC GPIOs that are configured as CapSense sensor pins.



Typically, a ground hatch surrounds the sensor pad to isolate it from other sensors and traces. The intrinsic capacitance of the PCB trace or other connections to a capacitive sensor results in a sensor

parasitic capacitance (C_P). (Note that the red-colored electric field lines are only a representative of the electric-field distribution around the sensor, the actual electric field distribution is very complex).

When a finger is present on the overlay, the conductive nature and large mass of the human body forms a grounded, conductive plane parallel to the sensor pad. This adds a capacitance C_F i.e. finger capacitance, in parallel to the parasitic capacitance. Note that the parasitic capacitance C_P and finger capacitance C_F are parallel to each other because both represent the capacitance between the sensor pin and ground.

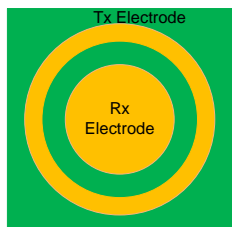


Mutual-Cap

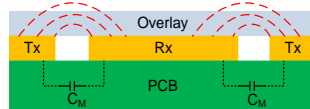
Mutual-capacitance sensing measures the capacitance between two electrodes, which are called transmit (Tx) and receive (Rx) electrodes.

In a mutual-capacitance measurement system, a digital voltage (signal switching between V_{DD} and GND) is applied to the TX pin and the amount of charge received on the RX pin is measured. The amount of charge received on the RX electrode is directly proportional to the mutual capacitance (C_M) between the two electrodes.

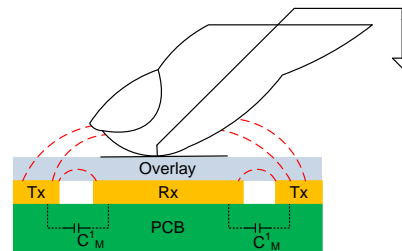
When a finger is placed between the TX and RX electrodes, some of the field lines terminate on the figure so the mutual-capacitance between the TX and RX electrodes decreases to C_M^1 . Because of the reduction in capacitance between the electrodes, the charge received on the RX electrode also decreases. The CapSense system measures the amount of charge received on the RX electrode to detect the touch/no touch condition.



Top View



Side View – No Touch



Side View - Touch

Raw Counts

The CapSense component uses dedicated circuitry internal to PSoC to convert the capacitance C_S into equivalent digital counts called raw counts. An increase in the raw counts indicates a finger touch (for Mutual-Cap the actual capacitance decreases for a touch but the PSoC handles this inversion so that the raw count that you see will always increase for a finger touch).

When a finger touches the sensor, the C_S increases from C_P to $C_P + C_F$, and the raw count increases. By comparing the change in raw count to a predetermined threshold, logic in the CapSense component decides whether the sensor is active (finger is present).

Signal to Noise Ratio (SNR)

The raw counts vary with respect to time, due to inherent noise in the CapSense systems. CapSense noise is the peak-to-peak variation in raw counts in the absence of a touch, and CapSense signal is the average shift in raw counts brought in by a finger touch on the sensor.

SNR is the ratio of CapSense signal to CapSense noise.

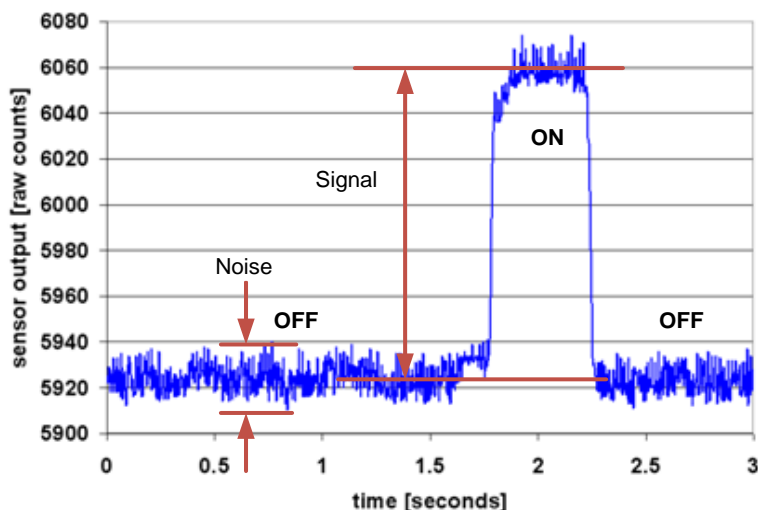
For a reliable touch detection, it is required to have a minimum SNR of 5:1, but a larger value is recommended if possible. This requirement comes from best practice threshold settings, which enable enough margin between signal and noise in order to provide reliable ON/OFF operation.

SmartSense

The CapSense algorithm is a combination of hardware and firmware blocks inside PSoC. Therefore, it has several hardware and firmware parameters required for proper operation. These parameters need to be tuned to optimum values for reliable touch detection and fast response.

SmartSense is a CapSense tuning method that automatically sets sensing parameters for optimal performance based on the user specified finger capacitance, and continuously compensates for system, manufacturing, and environmental changes.

SmartSense auto-tuning reduces design cycle time and provides stable performance across PCB variations, but requires additional RAM and CPU resources, as indicated in the CapSense Component datasheet, to allow runtime tuning of CapSense parameters.



On the other hand, manual tuning requires effort to tune optimum CapSense parameters, but allows strict control over characteristics of a capacitive sensing system, such as response time and power consumption. It also allows use of CapSense beyond the conventional button and slider applications such as proximity and liquid-level-sensing.

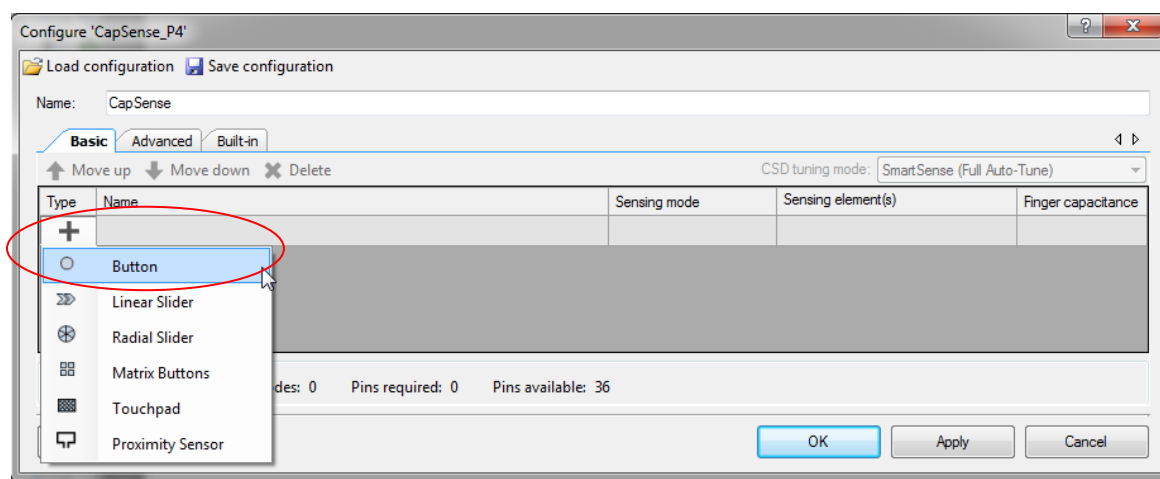
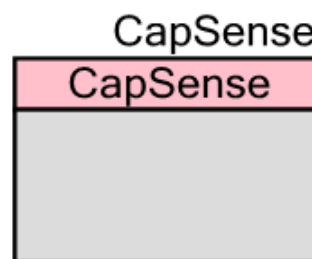
SmartSense is the recommended tuning method for all conventional CapSense applications. You should use SmartSense auto-tuning if your design meets the following requirements:

- The design is for conventional user-interface application like buttons, sliders, touchpad etc.
- The parasitic capacitance (C_p) of the sensors is within the SmartSense supported range as specified in component datasheet.
- The sensor-scan-time chosen by SmartSense allows you to meet the response time and power requirements of the end system.
- SmartSense auto-tuning meets the RAM/flash requirements of the design.

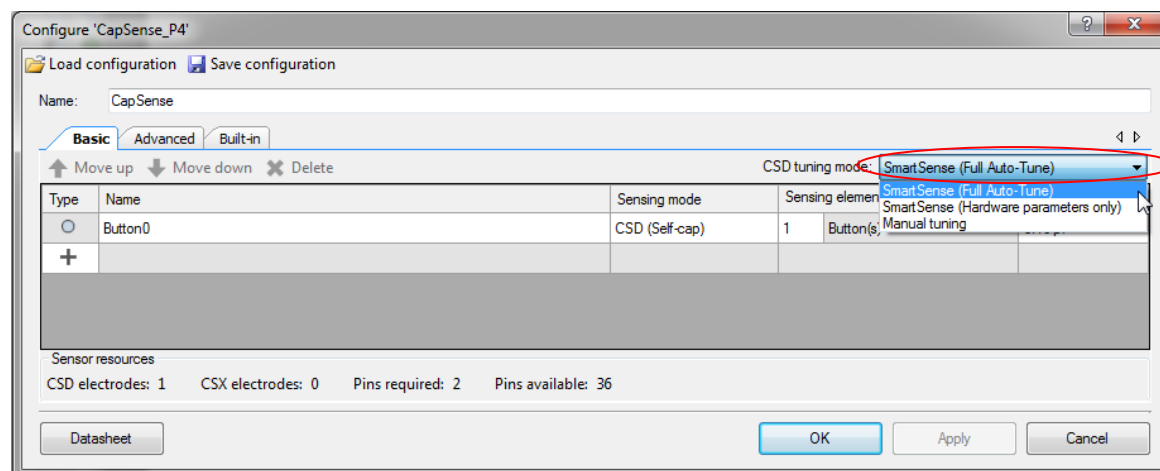
CapSense Button

Description: A CapSense button is simply a round copper PCB trace connected to a PSoC pin. In order to enable CapSense in your design you need to first add the CapSense component to your schematic.




Then configure the block. First, change the name to “CapSense”. Then add a Button Widget to your design by Pressing “+” in the basic view and selecting “Button”. Select “Sensing mode” as “CSD (Self-cap)” and “Sensing element(s)” as “1”. Because our board has no overlay on top of the buttons, the “Finger capacitance” should be increased from the default value to 0.5 pF or larger. This will prevent the possibility of false touches from being recognized due to coupling capacitance from your finger to CapSense traces on the board.



Next, choose the “CSD tuning mode”. The easiest tuning configuration is “SmartSense (Full Auto-Tune)” – a unique capability of Cypress that automatically adjusts the system to work with variations in layout and manufacturing.



You will need to assign the button to the appropriate pin. You will also need to assign Cmod to a pin. This is an external capacitor that is used by the CapSense block. On the CY8CKIT-042 Pioneer kit, Cmod is on P4[2]. Add a digital output pin called “led” to drive the LED that corresponds to the chosen button on the board. Make sure to turn off the HW connection since this pin will be controlled by the firmware.

	Name	Port	Pin	Lock
	\CapSense:Cmod\	P4 [2]	22	<input checked="" type="checkbox"/>
	\CapSense:Sns\ (Button0_Sns0)	P3 [4]	15	<input checked="" type="checkbox"/>
	led	P0 [0]	24	<input checked="" type="checkbox"/>

Then you need to write the CapSense firmware which has this basic flow:

1. Start the CapSense component
2. Setup the widget to scan
3. Start an initial scan
4. Check to see if the CapSense component is busy scanning... if it isn't then:
 - a. Process the CapSense data
 - b. Read the state of the button
 - c. Restart the CapSense scanning
 - d. Update the LED
 - e. Repeat from step 4

Note: CyGlobalIntEnable MUST be placed before CapSense_Start is called - if not the firmware will hang.

```
#include <project.h>
```

```
int main()
{
    uint8 button1=0;

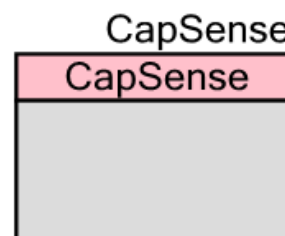
    /* Must enable interrupts before starting CapSense */
    CyGlobalIntEnable;

    CapSense_Start();
    CapSense_ScanAllWidgets(); /* Do initial scan */

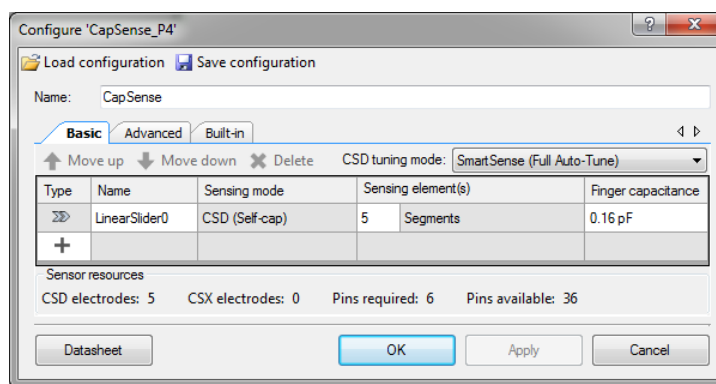
    for(;;)
    {
        if(!CapSense_IsBusy())
        {
            CapSense_ProcessAllWidgets();
            button1 = CapSense_IsWidgetActive(CapSense_BUTTON0_WDGT_ID);
            CapSense_ScanAllWidgets();
        }
        led_Write(button1); /* Turn on LED if the button is being pressed */
    }
}
```

CapSense Slider

Description: A CapSense slider is an array of closely spaced CapSense sensors. When you place your finger on the array of sensors, it will partially cover one sensor, completely cover another, then partially cover a third. To figure out the exact center of your finger the CapSense block will automatically calculate the “center of mass”, also known as the “centroid”. To use a capacitive touch slider, first add the CapSense block to your schematic and change the name to “CapSense”.



Next, configure your slider by pressing “+” in the “Basic” tab and selecting “Linear Slider”. The slider shown below will have 5 elements (PCB traces) – this is exactly what the CY8CKIT-042 PSoC 4 Pioneer has. By default the CapSense component will return a number between 0 and 99 (that is also known as the API resolution) but you can change it to match your application’s requirement. This setting is defined in the “Advanced” > “Widget Details” tab under “Maximum position”.



- When you barely touch the slider on one side it will return 0.
- When you are in the middle it will return 50.
- When you are barely touching the slider on the other side it will return 100.
- When you are not touching the slider at all, it will return a value of 0xFFFF.

Add a pin to the schematic for an LED and drive it from a PWM. Set the PWM period to 100. Remember to assign the LED and CapSense slider elements to pins.

Then you need to write the CapSense firmware which has this basic flow:

1. Start the PWM and CapSense components
2. Setup the widget to scan
3. Start an initial scan
4. Check to see if the CapSense component is busy scanning... if it isn't then:
 - a. Process the CapSense data
 - b. Read the centroid of the slider
 - c. Restart the CapSense scanning
 - d. Update the PWM compare value
 - e. Repeat from step 4

```

#include <project.h>

int main()
{
    uint16 position=0;

    /* Must enable interrupts before starting CapSense */
    CyGlobalIntEnable;

    PWM_Start();
    CapSense_Start();
    CapSense_ScanAllWidgets(); /* Do initial scan */

    for(;;)
    {
        if (!CapSense_IsBusy())
        {
            CapSense_ProcessAllWidgets();
            position = CapSense_GetCentroidPos(CapSense_LINEARSLIDER0_WDGT_ID);
            CapSense_ScanAllWidgets();
        }

        if (position != 0xFFFF) /* CapSense returns 0xFFFF for no touch */
        {
            PWM_WriteCompare(position);
        }
        else
        {
            PWM_WriteCompare(0);
        }
    }
}

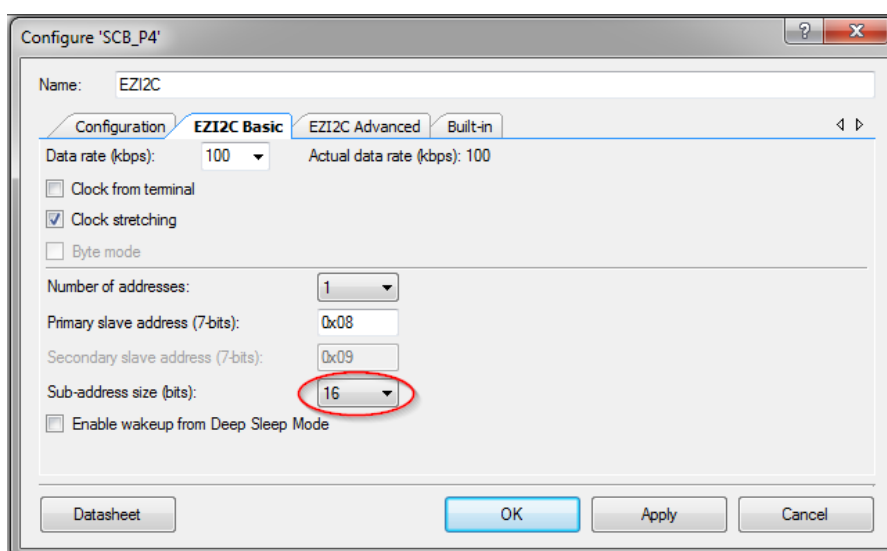
```


CapSense Tuning

If you use “SmartSense” then you may not have to tune the CapSense performance at all, but it is still good to know how to look at and plot the CapSense values relative to the touch thresholds. To do this, you use the “CapSense Tuner”.

To enable tuning, you must do the following:

1. Add an EZI2C component to your project. It should have a 16-bit sub-address size as shown below. The 16-bit sub-address size allows the tuner to use more than 256 register locations. Any Primary Slave Address value may be used but you will need to enter it into the tuner setup menu later on. Here we used a value of 0x08. Also make note of the data rate selected (100kbps in this case. Make sure you assign the correct I2C pins.

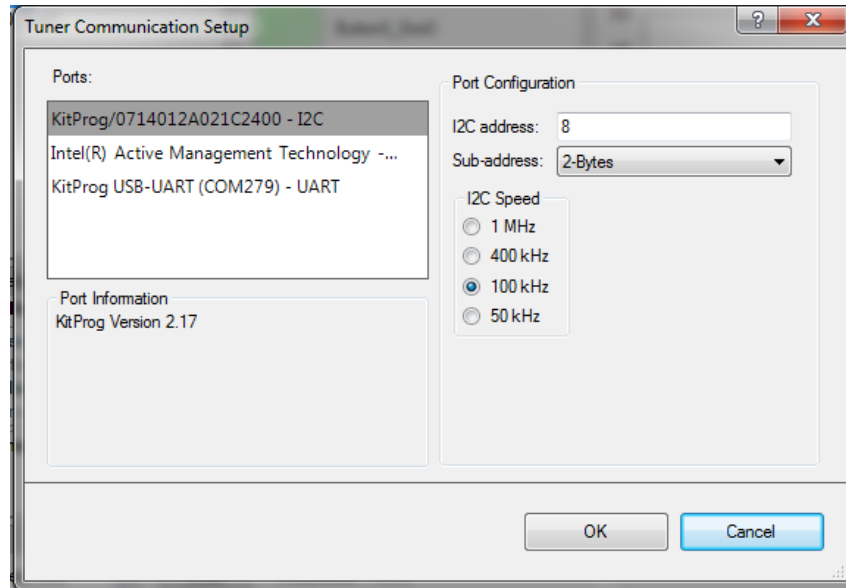


2. You now need to edit the code in order to “Start” the I2C block and point Buffer1 to the CapSense_dsRam variables. That is, you need to tell the I2C block where to find the CapSense values.

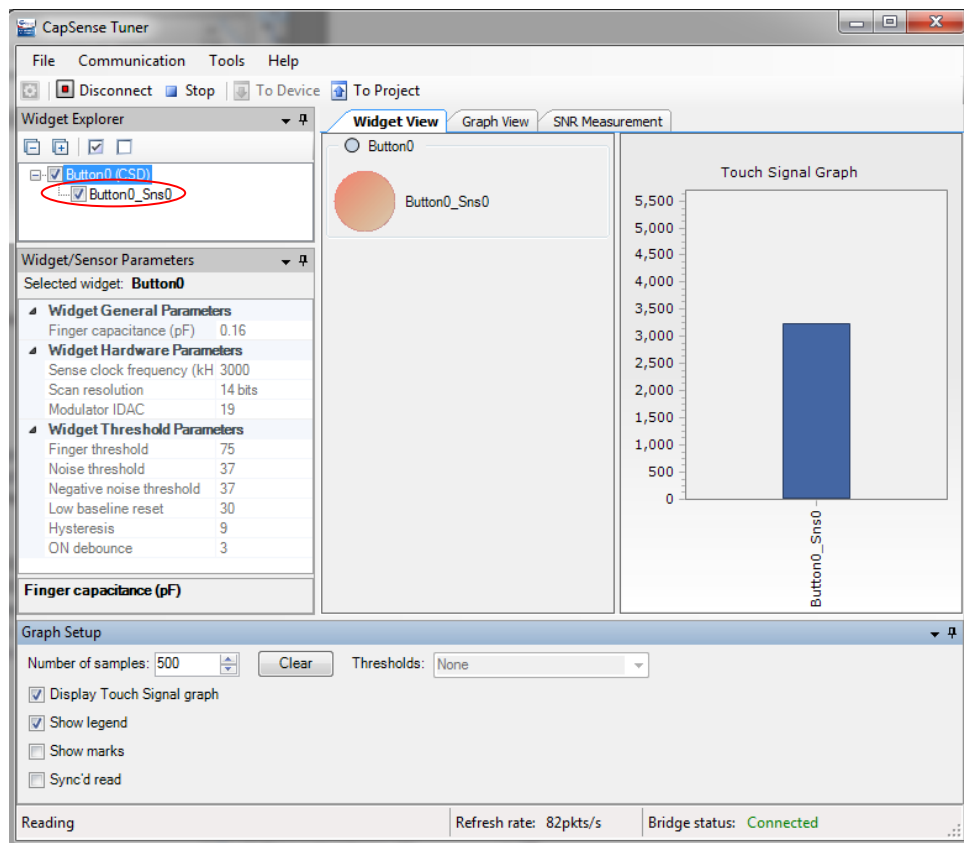
```
EZI2C_Start();  
  
/* Set up I2C data buffer to point to the CapSense data structure */  
EZI2C_EZI2CSetBuffer1(sizeof(CapSense_dsRam), sizeof(CapSense_dsRam),  
    (uint8 *) &CapSense_dsRam);
```

3. Next add the CapSense_RunTuner() API in order to synchronize the scan data between the device and tuner. This should be done after processing the scan and before starting the next scan (i.e. after CapSense_ProcessWidget but before CapSense_Scan).

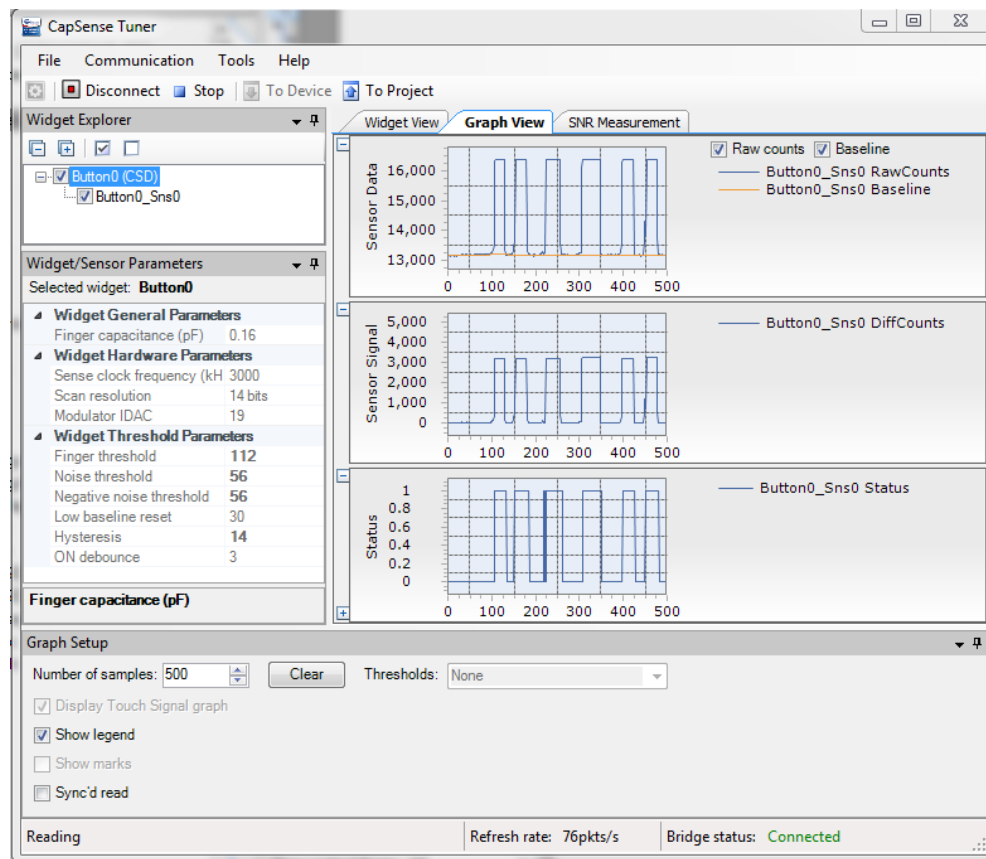
Once the changes shown above have been made, program the device normally. Then, right click on the CapSense component in Creator and select “Launch Tuner”. The tuner window will open. Click on “Tools” > “Tuner Communication Setup” and set it as shown here:



Once it is configured, press “OK”. Press the “Connect” button if it is not already connected. When it is connected, the “Connect” button will change to say “Disconnect”. Press the “Start” button and select the sensor check box in the “Widget Explorer”. Touch the button on the board and observe the signal as shown below. This figure is shown for a single button, but the tuner will work for any number of widgets.



Switch to the “Graph View” tab at the top to look at “Sensor Data”, “Sensor Signal” and “Sensor Status” in real time. For “Sensor Data” the RawCounts and Baseline can be viewed together or shown independently by checking and unchecking the appropriate boxes. Please also note in the “Widget/Sensor Parameters” window the “Widget Threshold Parameters” will update automatically as SmartSense continually calibrates the performance of the button.



Some of the Widget/Sensor parameters can be updated while others are greyed out. If you configured the button for manual tuning, you can update more of the parameters than you can in SmartSense mode. Once parameters are updated, you can download them to the device to see how they affect the performance, or you can upload them to the project to save their values in the CapSense component.

The SNR Measurement tab can be used to measure the signal to noise ratio for your CapSense button.

When finished, press “Stop” and close the tuner window.