Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

AN49079

PSoC® 1 CapSense Plus™: Dynamically Configuring CapSense®

Author: Vibheesh Bharathan

Associated Project: Yes

Associated Part Family: CY8C21x34, CY8C24x94

Software Version: PSoC Designer™ 5.4

Related Application Notes: AN2104, AN2397

Dynamic reconfiguration is one of the powerful features available in PSoC 1. This application note explains how to create a PSoC 1 CapSense Plus application with dynamic reconfiguration.

Contents

Introduction 1
What Is a Configuration? 1
How Dynamic Reconfiguration Works 2
  Base Configuration Concept 2
  GPIO Drive Modes Across Configurations 2
  Global Resources Settings 2
  Interconnections and Configuration 3
Creating a New Configuration in PSoC Designer 3
  Handling Interrupts 3
  Using APIs for Reconfiguration 3
  Changing the Configuration in Real Time 4
  Optimizing for Speed or Size 4
CapSense Plus Project 4
  Variable Clocks and CapSense (CSD) 5
  Deciding on the Configuration 6
  Code Walkthrough 7
  Schematic Diagram of the Circuit for Testing 7
  Testing the Example Project 8
Summary 9
Worldwide Sales and Design Support 11

Introduction

CapSense is a widely accepted technology that is used to replace conventional mechanical switches. Creating an application by adding user modules like PWM or ADC to a CapSense device reduces the BOM cost and adds value to the design. This application note explains how to use dynamic reconfiguration to create PSoC 1 CapSense Plus applications and provides tips on using the reconfiguration efficiently.

What Is a Configuration?

A configuration is set of code that enables a particular block or set of blocks, interconnections, GPIO settings, and global settings to make the device work as specific hardware. If a set of digital and analog blocks is configured to work as a CapSense block, that same set of blocks can be reused to implement other functions like ADC, counter, and so on at two different instances in time. This process of reusing digital and analog resources is called “dynamic reconfiguration.”

Why implement such an idea to design an application with PSoC 1? If the PSoC 1 device used for the design does not have the required digital and analog blocks to implement all functionalities, you can use dynamic reconfiguration to reuse the blocks and time share PSoC 1 hardware efficiently. This gives a cost advantage to the solution, as PSoC 1 is priced according to the number of blocks in the device. Thus, the reuse of hardware saves money.

How Dynamic Reconfiguration Works

Consider a digital block, which has a set of seven registers—input, output, function, control, and three data registers—that control the block's functionality. The input and output registers select the source of input to the block and route the output of the block to an I/O or other blocks. The function register decides the intended functionality of the block (whether the block should work as a counter, a timer, or something else). The control register enables and disables the block, maintains status flags, and so on. The overall functionality of a particular block is completely dependent on these register values. (Refer to the "Register Details" chapter in the PSoC Technical Reference Manual).

When a block is configured to work as a particular function (user module), the above-mentioned registers are written with specific values. Changing these values during run time allows you to change the functionality of the device dynamically, and hence it is called "dynamic reconfiguration."

In short, all the resources inside the PSoC 1 digital and analog blocks; all global resources like CPU speed, clock dividers, and so on; GPIO pin type; type and drive mode; and digital and analog interconnects are controlled by RAM-based registers whose values can be changed during run time to create a different functionality. This makes PSoC 1 more flexible and dynamically reconfigurable.

Base Configuration Concept

Each PSoC 1 project has one base configuration and optional loadable configurations. The base configuration is automatically loaded during power up. You can load or unload any other optional configuration, as required. It is preferable to keep all the user modules that have to be active throughout the program's run time in the base configuration. Loading the base configuration is time consuming and requires more code, since it must configure all the register values based on the settings in the Device Editor. Do not unload the base configuration, as it takes the blocks and port pins to the reset state. The example project provided in this application note has a base configuration and three optional configurations for implementing the ADC, EZI2C, and CapSense peripherals.

GPIO Drive Modes Across Configurations

The GPIO drive mode settings in the base configuration remain unchanged over all optional loadable configurations until a change is made explicitly for each configuration in the GPIO setting window. If an application needs a port pin to be configured as strong drive mode in the first configuration and as resistive pull-up in the second configuration, you can do so by setting it in the port setting window individually for each configuration. The value of the port data resistor remains the same across all configurations. If the value of the port data register (PRTxDR) has to be changed during the configuration change, the application program should change the value of PRTxDR.

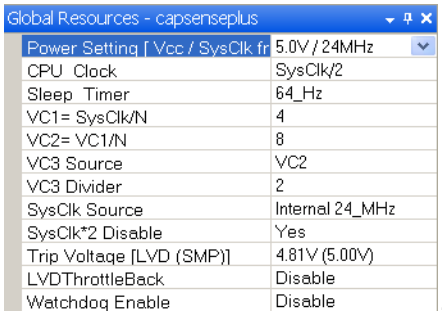
Remember the following points when configuring GPIOs in dynamic reconfiguration:

- All GPIO settings in the base configuration are applied globally to the optional loadable configurations.
- Changes made to the GPIO settings inside an optional loadable configuration remain local to that configuration.
- The value of the port data register does not change when changing configurations. Any change required in the data registers after switching configurations has to be done in firmware by writing to the PRTxDR register.

Global Resources Settings

The global resources behave the same as GPIO. All global resources settings in the base configuration remain unchanged throughout the optional configurations unless they are explicitly changed inside those configurations. Keeping the global parameters the same in the configurations helps to keep parameters like power setting, CPU clock, and SMP, which hardly need to change, the same. Figure 1 shows a screen shot of the **Global Resources** window.

Figure 1. Global Resources Window



Interconnections and Configuration

When a new configuration is created, the connections between the port pins and global bus nets remain unchanged, whereas the connections established between the global bus and the row interconnects are no longer available. This helps minimize code change when switching between the configurations. So, if you want a particular connection between a row output net and a global out net, make this connection in the new configuration.

Similarly, for the analog, the column clock, column mux input, and column clock selections remain unchanged unless explicitly changed over configurations.

Figure 2 shows the base configuration, which establishes the connection between a digital block to row interconnects (A), from row interconnects to the global bus net (B), and from the global bus net to the port pin (C).

Figure 2. Interconnect View of Base Configuration

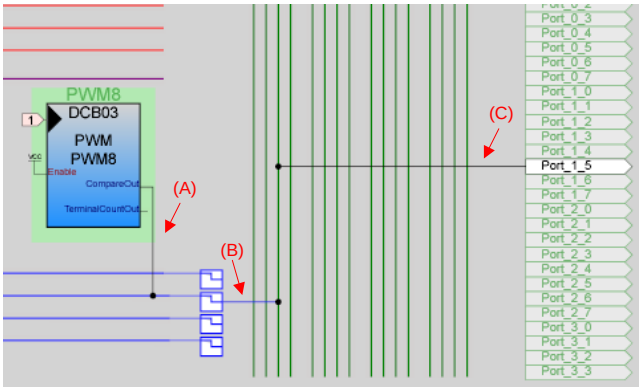
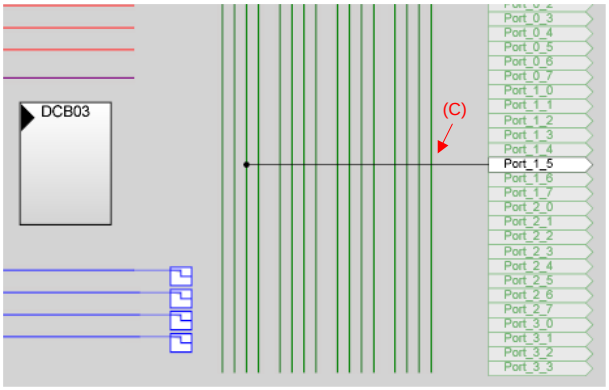


Figure 3 shows the interconnect view of a newly created optional loadable configuration in which only the connection between the global bus net to port pin (C) remains unchanged, and other connections are removed. This is the default connection scheme provided when a new configuration is created. You can modify it, if required.

Figure 3. Interconnect View of Optional Configuration



Creating a New Configuration in PSoC Designer

Choose **Interconnect** > **Add loadable configuration to…** in PSoC Designer™ 5.4 and **Config** > **Loadable Configuration** in PSoC Designer 4.4 to create, rename, or delete a loadable configuration. Selecting the create option opens the blank Device Editor (interconnect view) page, where no user modules are placed in any of the blocks. The number of configurations that can be made is limited only by the program memory and the size of the configuration.

Handling Interrupts

When one hardware block is shared between configurations, the interrupt branches to the same vector address per the vector table regardless of the configuration. An intermediate function, Dispatch_INTERRUPT_ <Interrupt number>, is called from the interrupt vector location, in which the dispatch routine determines the current active configuration and the interrupt branches to the corresponding user-defined ISRs for execution. All these dispatch interrupt functions are located in the *psocdynamicint.asm* file.

The effect of dynamic reconfiguration on the interrupts is an increase in latency, as a dispatch function is executed to decide on the active configuration. Do not change the *boot.asm* (*boot.tpl*) file to redirect the ISR calls to a user-defined function because every shared interrupt should go through the interrupt dispatch function. Doing so will produce conflicts in the interrupt servicing, rendering the system unstable.

Using APIs for Reconfiguration

After a new configuration is created in PSoC Designer, APIs are generated to load and unload the configurations. The function LoadConfig<Config Name> configures the required analog or digital block to work as the selected user module and establishes the interconnection of the blocks (between the blocks or input and output routings) as defined in that configuration. This function writes specific values to the registers that configure the analog/digital blocks, interconnects, and so on as defined in the Device Editor.

The function UnloadConfig<Config Name> resets the registers that are configured for a particular configuration. Note that unloading a configuration does not return the device to a previously loaded configuration. It only resets the resources. To load a previously running configuration, call the relevant API function.

Also, when a configuration is loaded or unloaded using the previously mentioned functions, only the registers that correspond to the resources that are changed between the configurations are modified. The code generated to load or unload the configurations and to change the register values is located in the *psocdynamic.asm* and *psocconfigtbl.asm* files.

Changing the Configuration in Real Time

Unload the current configuration before loading another one if both configurations use common hardware. This avoids conflicts in register configuration. Reconfiguring the set of hardware blocks without unloading the present configuration creates conflicts. However, never unload the base configuration, as that will take the device to an unstable state.

Consider the following situation. A project requires a PWM on DBB00 to run continuously. DBB01 should be time shared to work as a Timer and a Counter at different instances. To implement this, place the PWM in DBB00 in the base configuration. Since the base configuration is loaded by default and is never unloaded, DBB00 always operates as a PWM. Create two loadable configurations and configure DBB01 as a Timer in one configuration and as a Counter in another configuration. Ensure that the Timer or Counter does not use the same row interconnect resource as the PWM in the base configuration. In the application program, when DBB01 has to be configured as a Timer, load the Timer configuration. When DBB01 has to function as a Counter, unload the Timer configuration and then load the Counter configuration. To revert to the functionality of the Timer, unload the Counter configuration and load the Timer configuration.

Time Requirements for Changing the Configuration

The time required to change from one configuration to another depends on the number of registers that need to be updated in the new configuration. For example, to configure a digital block from a Timer to a PWM, six registers are updated. A CPU clock of 24 MHz takes about 1.5 µs. Similarly, to change a PGA to a Comparator, only two registers need be updated, which takes about 0.5 µs. Apart from these register writes, which configure the digital or analog block, there are register writes that update the interconnects, GPIO drive modes, and so on.

The most accurate method of determining the time required is to calculate the number of CPU cycles taken by the LoadConfig function (by adding the CPU cycles taken by each assembly instruction on the API, or measuring the execution time of the API by toggling a port pin) and then calculate the total time from the CPU speed.

Optimizing for Speed or Size

There are two options to create code for the loadable configurations, as listed in Table 1. Choose **Project** > **Settings** and select the **Chip Editor** tab. The first option, “Loop (Size Efficient),” generates code that uses less flash memory, but takes a longer time to load. In this method, all the values to be written to the registers are put in a table. The LoadConfig API loops through the table and updates the registers.

The second option, “Direct Write (Speed Efficient),” uses a larger amount of flash, but loads faster. In this method, all the registers are written directly using the mov reg[..] instructions.

Table 1. Code Optimization Options

Option Name	Effect in the Program
Loop (Size Efficient)	Consumes less code memory, but takes more time to load the configuration
Direct Write (Speed Efficient)	Loads the configuration faster, but consumes more code size

CapSense Plus Project

This example project controls the speed of a motor, based on the analog input from a feedback system. The ON/OFF control (user interface) of the motor is implemented with two CapSense switches.

This system requires one CapSense Sigma Delta (CSD) User Module for user interface switches, one 8-bit PWM to control the speed of the motor, an ADC to sample the analog input, and an I^2C slave interface (EzI2Cs) for tuning the CapSense buttons. Table 2 describes the resource requirements for the application.

Table 2. Resource Requirements

User Module Needed for Application	PSoC 1 Blocks Required for Each User Module
8-Bit PWM	1 digital block
8-Bit ADC	1 digital and 2 analog blocks
CapSense – CSD	3 digital and 3 analog blocks
EzI2C	I^2C dedicated block

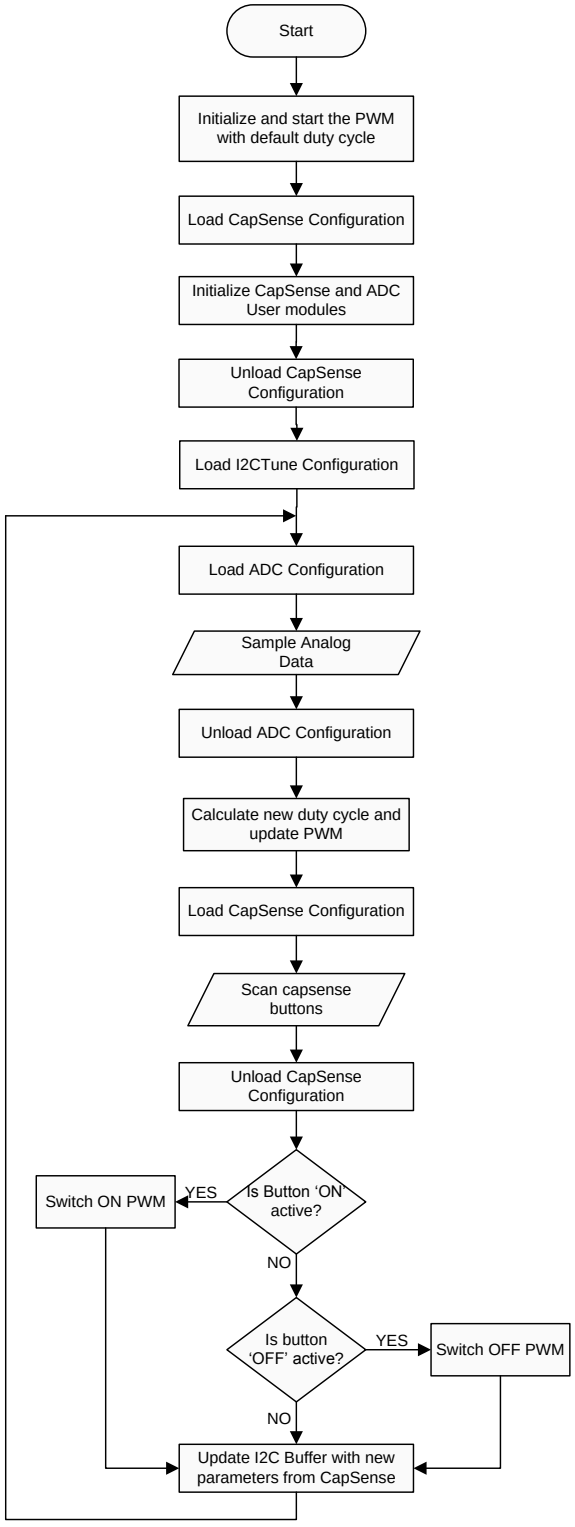
The example project requires five digital blocks and five analog blocks. The device used for the example project includes only four analog and four digital blocks. Dynamic reconfiguration enables the complete functionality of the example project to be implemented with four analog and four digital blocks.

Variable Clocks and CapSense (CSD)

When the CSD User Module is active, regardless of the selections made in the **Global Parameters** window, it alters the values of the VC1, VC2, and VC3 variable clocks based on the scanning speed and resolution. The value of the variable clocks for each setting of the user module is located in the user module datasheet in the “Resolution” subsection under “Parameters and Resources.” This change of variable clock values affects other configurations that are active along with the CapSense configurations if they use a different variable clock divider setting than the CSD. The workaround for this problem is to use the same VC1, VC2, and VC3 dividers set by CSD in the other configurations.

Figure 4 shows the flow chart of the application, based on which the configurations are created.

Figure 4. Application Flow Chart



Deciding on the Configuration

The CSD and the ADC do not operate continuously. Therefore, these user modules should be placed in two optional loadable configurations. On the other hand, the PWM, which controls the motor, has to be continuously active and hence should be placed in the base configuration. The EzI2C is placed in another optional loadable configuration, as described in Table 3.

Figure 5 shows a complete view of how user modules are placed in each configuration without any resource conflict between configurations.

The PWM User Module in the base configuration is active throughout the program. As mentioned in Interconnections and Configuration, when a configuration (ADC, CapSense, or I2CTune) is loaded with the base configuration, the connection between the row interconnects and global bus is no longer available. Therefore, you should make this connection manually in all the configurations that are loaded with the base configuration so that the user modules will work properly in the base configuration.

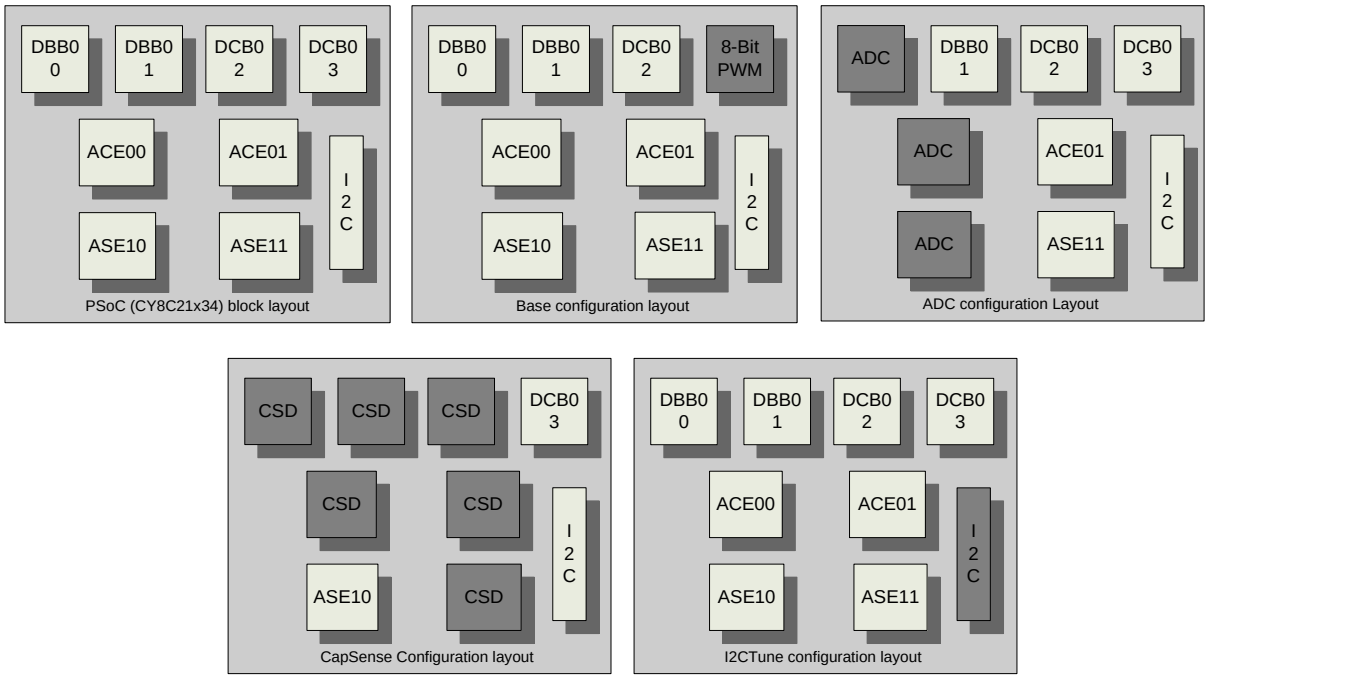
Table 3. Configurations

Required User Modules	Required PSoC 1 Blocks	Configuration
8-Bit PWM	1 digital block	CapSense Plus (base configuration)
8-Bit ADC	1 digital block and 2 analog blocks	ADC (optional configuration 1)
CapSense – CSD	3 digital blocks and 3 analog blocks	CapSense (optional configuration 2)
EzI2C	I^2C dedicated block	I2CTune (optional configuration 3)

Note The EzI2C User Module ideally should be included in the base configuration, as it should work continuously like the PWM. This is because when a new configuration is loaded, a set of global registers like the global digital interconnect, IDAC, decimator, and I2C are written with a reset value based on a comparison with the base configuration. Therefore, the I2C stops working when a configuration is changed. As a workaround, the I2C is placed in a loadable configuration (I2CTune) and loaded once at the beginning of the program. This configuration is never unloaded. In this way, the loading and unloading of other configurations does not affect the I2C operation.

This same concept applies to the decimator and IDAC, so take care when using them in your applications.

Figure 5. User Module Placement in Each Configuration



Code Walkthrough

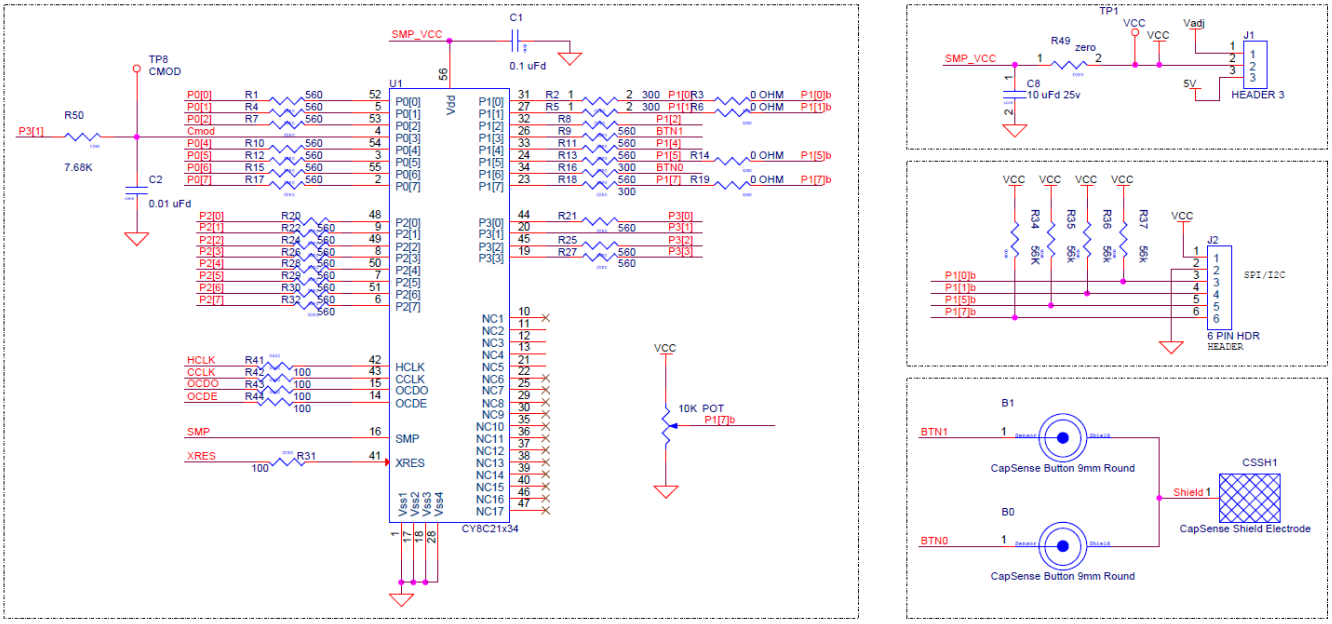
The following sequence is based on the program of the example project attached to this application note.

1. The software starts the PWM with a default duty cycle value, as the base configuration is loaded by default.
2. The I2CTune configuration is loaded over the base configuration. The I2C buffers are initialized and the I2C interface used for tuning CapSense is started. This configuration is never unloaded.
3. Now that the base configuration and the I2C configuration are active, the ADC configuration is loaded to sample the analog input. The ADC is started, and a conversion is initiated. When the conversion is complete, the ADC configuration is unloaded.
4. Based on the ADC result, a new PWM duty cycle is calculated, and the PWM is updated.
5. The CapSense configuration is loaded. The CSD User Module is started, the buttons are scanned using the ScanAllSensors function, and the baselines are updated. Remember that the baseline and finger thresholds should not be initialized at this stage because they are initialized in the beginning. The finger threshold, baseline, and raw counts are firmware implementations that persist throughout the program and across all configurations. After the sensors are scanned, the CapSense configuration is unloaded.
6. The last part of the software is the ON/OFF control of the PWM and updating the I2C buffers with the latest data from the CapSense User Module.

Schematic Diagram of the Circuit for Testing

The schematic diagram of the circuit is shown in Figure 6.

Figure 6. Schematic Diagram of CY3280-21x34 UCC Kit



All the components are already present and connected in the CY3280-21x34 UCC Kit. The only external component that needs to be connected is the system 5K potentiometer to provide 0 V to 4 V of variable voltage input to the ADC. This potentiometer should be connected to the P1.7 port pin using the J2 header, as described in Table 4.

Testing the Example Project

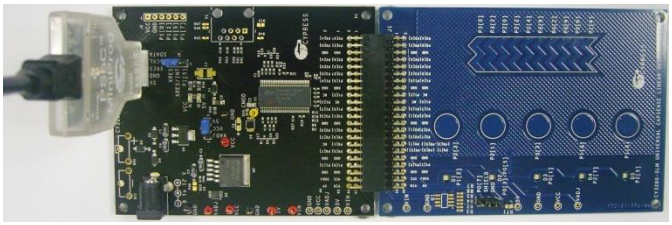
The example CapSense project can be tested on Cypress's CY3280-21x34 UCC Kit or CY3214-PSoCEvalUSB Kit (with modification to the CapSense button pin numbering). The procedure to test the project with the CY3280-21x34 UCC Kit and monitor the performance is as follows.

Table 4. Pin Configurations

Functional Input/Output	PSoC 1 Pin Number
PWM output	P1.5
ADC analog input	P1.7
CapSense buttons	P1.3 and P1.6 (Button 1 and 2 in CY3280-21x34 UCC Kit)
EzI2C – interface	SCL-P1.1, SDA-P1.0 (ISSP Header)

1. Connect the CY3280-SLM Kit to the CY3280-21x34 UCC Kit, as shown in Figure 7.

Figure 7. Connect CY3280-SLM to CY3280-21x34 UCC

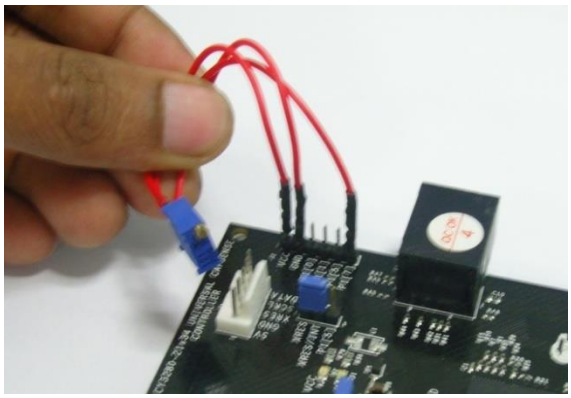


2. Configure the jumper settings:
   - Jumper on J2 on CY3280-SLM should be between GND and SHIELD.
   - Jumper on J1 on CY3280-21x34 UCC should be between 5 V and VCC.
   - Jumper on J4 on CY3280-21x34 UCC should be between XRES and XRES/XRES/INT.
3. Download the code to the CY3280-21x34 UCC Kit using MiniProg1 (shown in Figure 8), ICE-Cube, or MiniProg3.

Figure 8. Download Code (MiniProg1)

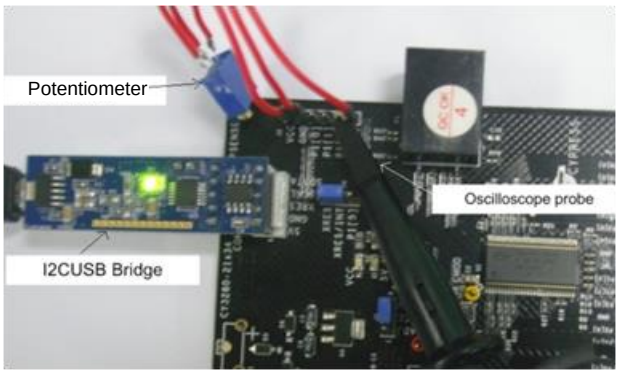
4. Connect the analog input (feedback), which can swing between 0 V and 4 V (for testing purposes, a simple potentiometer can be used), to the P1.7 port pin (use the header in the CY3280-21x34 UCC Kit), as shown in Figure 9.

Figure 9. Connect Analog Input



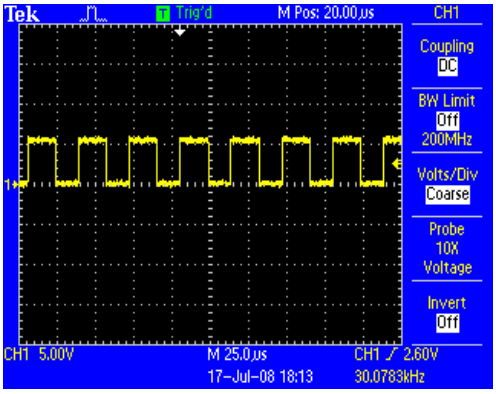
5. Connect the PWM output pin (P1.5) to the motor driving circuit or oscilloscope (use the header in the CY3280-21x34 UCC Kit).
6. Power the board from MiniProg1, MiniProg3, CY3240-I2USB Bridge, or adapter, as Figure 10 shows.

Figure 10. Connect PWM Output Pin and Power the Board



7. Monitor the application. The duty cycle of the PWM (30 kHz) varies from 32 percent to 67 percent as the ADC sampled data varies from 0 count to 62 counts. Using CapSense switch-1 and switch-2 stops and starts the PWM (Figure 11).

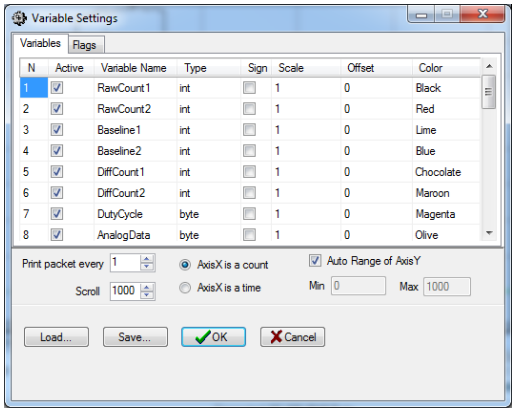
Figure 11. PWM Output on Oscilloscope



8. To monitor the CapSense parameters and tune the CapSense switches, connect the CY3240-I2USB Bridge and open the Bridge Control Panel software in the computer (refer to AN2397 for more information on how to use the CY3240-I2USB Bridge)

9. Open the **Variable Settings** window from **Chart** > **Data Setting** in the **File** menu (Figure 12). Click the **Load** button and open the *USBIIC_Cmd_CapSensePLUS.ini* file attached to this application note. Then click the **OK** button.

Figure 12. CY3240-I2USB Bridge Variable Settings

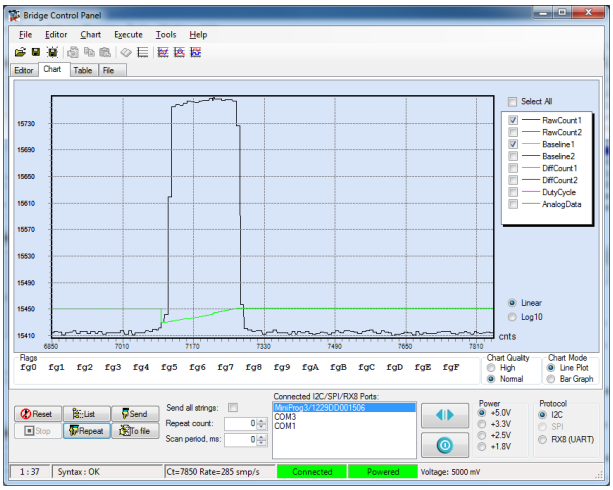


10. Open the file *I2CReadCommands.iic* attached to this application note by clicking **File** in the menu bar and then **Open file**.

11. Press **Enter** and click the **repeat** button in the USB-to-I²C bridge. Then move to the **Chart** window.

12. Monitor the raw count, baseline, and difference count data for the CapSense switches; the duty cycle of the PWM; and the analog sampled data, as shown in Figure 13.

Figure 13. CapSense Data on Bridge Control Panel



Summary

This application note explained how to build a CapSense Plus application when not enough blocks are available in the device. You can build additional useful and powerful applications using the same dynamic reconfiguration concept by adding analog and digital user modules like Amplifiers, Comparators, Filters, DACs, ADCs, Counters, Timers, PWMs, PRS, SPI, and UART. The ability to time share the hardware makes PSoC 1 unique among conventional microcontrollers and programmable devices.

About the Author

Name: Vibheesh Bharathan.

Title: Systems Engineer Staff

Document History

Document Title: PSoC® 1 CapSense Plus™: Dynamically Configuring CapSense® - AN49079

Document Number: 001-49079

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2617502	BVI	12/08/2008	New Application Note
*A	3124084	BVI	12/30/2010	Added figures and replaced the obsolete CY3213 kit with CY3280-21x34 UCC and CY3280-SLM kits in the example project implementation section. Revised the test setup details.
*B	4274568	DCHE	02/07/2014	Removed reference of AN2352 and added reference of AN2397 Application Note instead in all instances across the document. Added hyperlinks to CY3280-21x34 UCC kit. Updated in new template. Completing Sunset Review.
*C	4581722	DCHE	11/27/2014	Added references to Bridge Control Panel. Added Figure 6. Updated Figure 7, Figure 12 and Figure 13.
*D	5709419	AESATP12	04/26/2017	Updated logo and copyright.
*E	5779774	DCHE	06/20/2017	Updated project to PSoC Designer 5.4. Renamed the Application Note to PSoC® CapSense Plus™: Dynamically Configuring CapSense®.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709