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.

AN66627

PSoC® 3 and PSoC 5LP Intelligent Fan Controller

Author: Jason Konstas
Associated Project: Yes
Associated Part Family: All PSoC 3 and PSoC 5LP parts
Software Version: PSoC Creator™ 3.3 CP1 or higher
Related Application Notes: AN89346, AN78692

AN66627 demonstrates how to quickly and easily develop four-wire brushless DC fan control systems using PSoC® 3 or PSoC 5LP. The Fan Controller Component, available in PSoC Creator™, helps manage the fans in a variety of configurations. This application note also shows how to combine fan control and temperature sensing to create a complete thermal management solution using PSoC 3 and PSoC 5LP.

Contents

1 Introduction

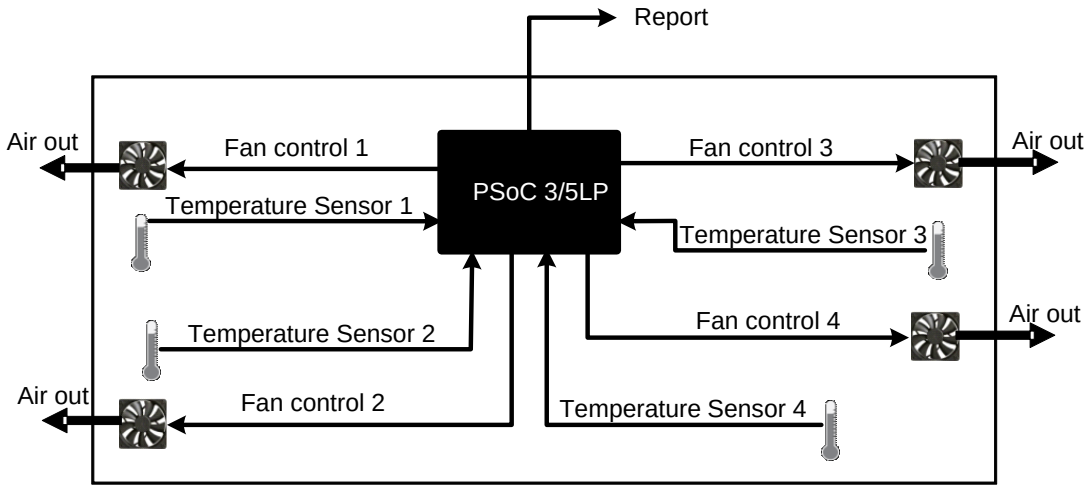
System cooling is a critical component of any high-power electronic system. As circuits become smaller, the demands on system designers to improve the efficiency of system thermal management are increasing. System cooling fans must run at the optimum speed to ensure that the system temperature is always below a defined limit. To achieve this, the electronic system needs a temperature measurement unit and a closed-loop fan speed controller.

PSoC® 3, PSoC 5LP have the resources required to implement a complete thermal management solution in a single chip (see Figure 1):

- The analog blocks, which include a 20-bit Delta-Sigma ADC, opamps, and IDACs, can be used to interface with analog temperature sensors such as diodes, thermistors, RTDs, and thermocouples.
- The pulse-width modulator (PWM) hardware blocks created using universal digital blocks (UDBs) can be used to drive as many as 16 independent fans. The UDBs can also be used for fan speed measurement using the tachometer signals from the fans.
- The Direct Memory Access (DMA) channels can be used along with the PWM and tachometer blocks to implement a completely hardware-based closed-loop fan control. Thus the CPU is freed for performing other application-related functions.

- The 32-bit ARM® Cortex™-M0 CPU (PSoC 5LP) or the 8-bit 8051 CPU (PSoC 3) can be used to implement firmware algorithms for measuring analog temperature sensors, and to implement the overall thermal management algorithms.
- The communication components can be used to report the fan status to an external host through the I^2C, SPI, or UART interface.

Figure 1. PSoC 3 and PSoC 5LP Thermal Management Application



PSoC Creator™, the integrated development environment (IDE) for PSoC, provides a Fan Controller Component to simplify the design process. The Component takes care of the closed-loop speed control. All you need to do is specify the required fan speed based on the system temperature. This application note shows how to use this Component to build a PSoC 3– and PSoC 5LP–based thermal management system, with the help of several example projects.

Other devices in the PSoC series—PSoC 1 and PSoC 4—can also be used for a fan controller solution. See the following application notes for these devices:

AN78692, PSoC 1 Intelligent Fan Controller

AN89346, PSoC 4 Intelligent Fan Controller

Select a PSoC device family based on the number of fans to be controlled and the number of temperature sensors to process. Table 1 compares the PSoC family products.

Table 1. Comparison of Different PSoC Families

Parameter	PSoC 1	PSoC 4	PSoC 3 and PSoC 5LP
Number of Fans	2	4	16
Sensor Types	RTD, Thermistor	RTD, Thermistor	Diode, RTD, Thermistor, Thermocouple
I^2CSensors	Yes	Yes	Yes

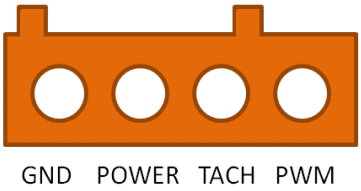
2 Four-Wire Fan Basics

Figure 2 shows a typical four-wire fan. Two of the wires are used to supply power to the fan. The other two wires are used for PWM and tachometer feedback for speed control and speed sensing, respectively.

Figure 2. Typical Four-Wire DC Fan

At the cabling level, wire color coding is not consistent across manufacturers, but the connector pin assignment is standardized. Figure 3 shows the connector pinout when viewed looking into the connector with the cable behind. Note that the connectors are keyed to prevent incorrect insertion into a fan controller board.

Figure 3. Four-Wire DC Fan Connector Pin Assignment



Fans come in standard sizes—with 40 mm, 80 mm, and 120 mm diameter being the most common. One of the specifications to be checked when selecting a fan for a cooling application is how much air the fan can move. This is specified either as cubic feet per minute (CFM) or cubic meters per minute (m^3/min). The size, shape, and pitch of the fan blades all contribute to the fan's capability to move air.

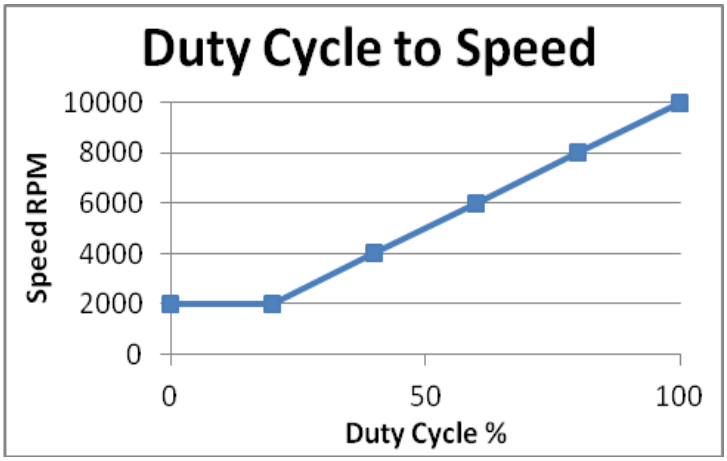
Smaller fans are typically used in space-constrained applications. They must run at a higher speed to move the same volume of air in a given period; therefore, they use more power and generate significantly more acoustic noise. This unavoidable tradeoff must be made to meet system requirements.

3 Fan Speed Control

The speed of a four-wire fan is controlled through the use of a PWM signal. Increasing the duty cycle of the PWM signal increases the fan speed, as Figure 4 shows. Fan manufacturers specify how the PWM duty cycle relates to nominal fan speed using either a table of data points or a graph.

It is important to note that at low duty cycle, not all fans behave the same way. Some fans stop rotating as the duty cycle approaches zero percent, whereas others rotate at a nominal specified minimum rpm. In both cases, the duty-cycle-to-rpm relationship can be nonlinear or not specified. A fan controller solution should include a mechanism to handle this fan behavior.

Figure 4. Example of a Duty-Cycle-to-Speed Chart



3-wire fan control: This application note covers only 4-wire fan control methods, and does not cover 3-wire fan control. In addition to the power, ground, and tachometer terminals present in 3-wire fans, 4-wire fans have an additional PWM control signal. 4-wire fans were introduced mainly to improve the acoustic noise reduction in the system, and are the widely used fans in the market today.

3-wire fan control involves controlling the power supply to the fan using a variable DC voltage or low frequency PWM signal. In the low frequency PWM control method, the tachometer signal is valid only when the fan is powered (PWM signal is at logic high). So, 3-wire fans need to be turned ON for the entire period of speed measurement in addition to the normal control operation. In 4-wire fans, the hall sensor and the internal controller are always powered, and the additional PWM signal separately controls the power to the fan coils. This results in the tachometer signal being valid under all conditions irrespective of the PWM control signal.

The FanController component in PSoC Creator supports only 4-wire fan control. But PSoC 3 and PSoC 5LP have all the hardware resources (PWM/Timer Blocks, additional digital glue logic) required to implement 3-wire fan control. Contact Cypress technical support in case you need assistance in developing a 3-wire fan control solution using PSoC.

4 Measuring Fan Speed

DC fans include Hall-effect sensors that sense the rotating magnetic fields generated by the fan's rotor as it spins. The output of the Hall-effect sensor is a pulse train (referred to as a tachometer signal) that has a frequency directly proportional to the fan speed. The number of pulses produced per revolution depends on the number of poles in the electromechanical construction of the fan.

For the most common four-pole brushless DC fan, the tachometer output from the Hall-effect sensor generates two high (T1 and T3) and low pulses (T2 and T4) per fan revolution, as Figure 5 shows.

If the fan stops rotating due to mechanical failure or other fault, the tachometer output signal remains static at either a logic-low or logic-high level.

There are two ways to measure the rotation speed of a fan:

1. Use a high-frequency clock to measure the period of the tachometer pulse.

2. Count the tachometer pulses over a fixed period of time.

The selection of a particular method depends on the frequency to be measured. For a low-frequency input signal, it is better to use the first method because it provides a precise result with the measurement time dependent on the period of the input signal. For a high-frequency input signal, the second method is preferred because the first method requires a very high frequency clock to achieve the same precision. In fan control applications, the tachometer signal frequency is around 600 Hz at 10,000 rpm for a four-pole fan. For such a frequency range, the first method can be used.

Since the tachometer line of a fan is an open drain / open collector output, it is necessary to pull the line high through a resistor, as Figure 6 shows. The input pins on PSoC 4 allow the use of built-in pull-up resistors so an external resistor is not necessary. To avoid the effect of noise on speed measurements, use a small value capacitor (on the order of a few nF) as a ground.

5 Cypress Development Kits for Thermal Management

Cypress provides a CY8CKIT-036 PSoC Thermal Management Expansion Board Kit (EBK) for evaluating thermal management applications using PSoC devices (see Figure 7). The EBK comes with two four-wire fans plus a variety of analog and digital temperature sensors to enable you to quickly prototype a complete thermal management system. For more information on the kit, go to www.cypress.com/go/CY8CKIT-036.

Figure 5. Tachometer Output of a Fan

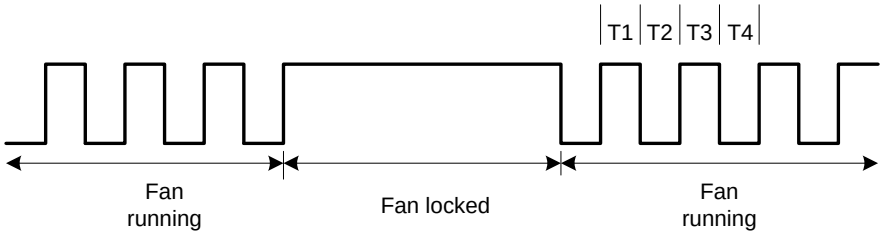


Figure 6. Tachometer Line Interface

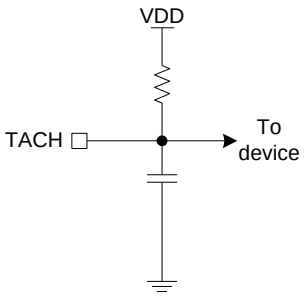
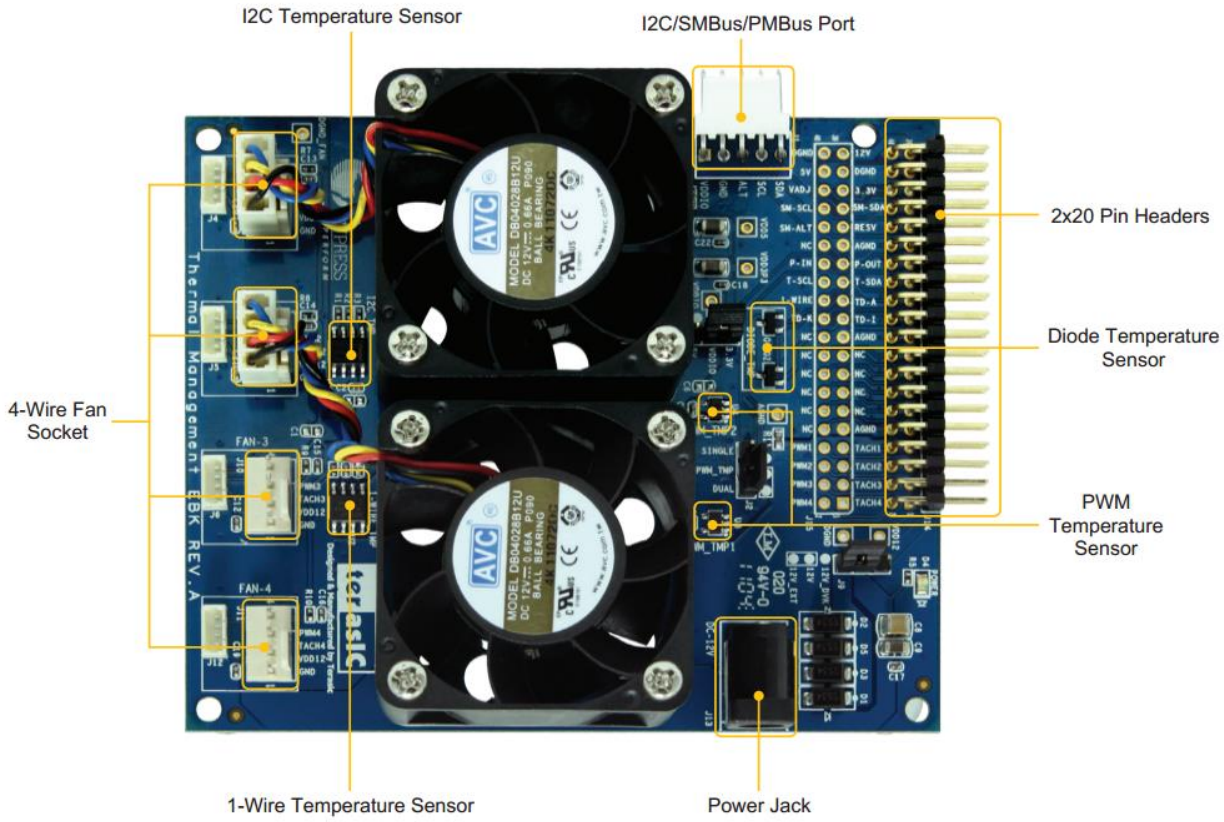


Figure 7. CY8CKIT-036 PSoC Thermal Management EBK



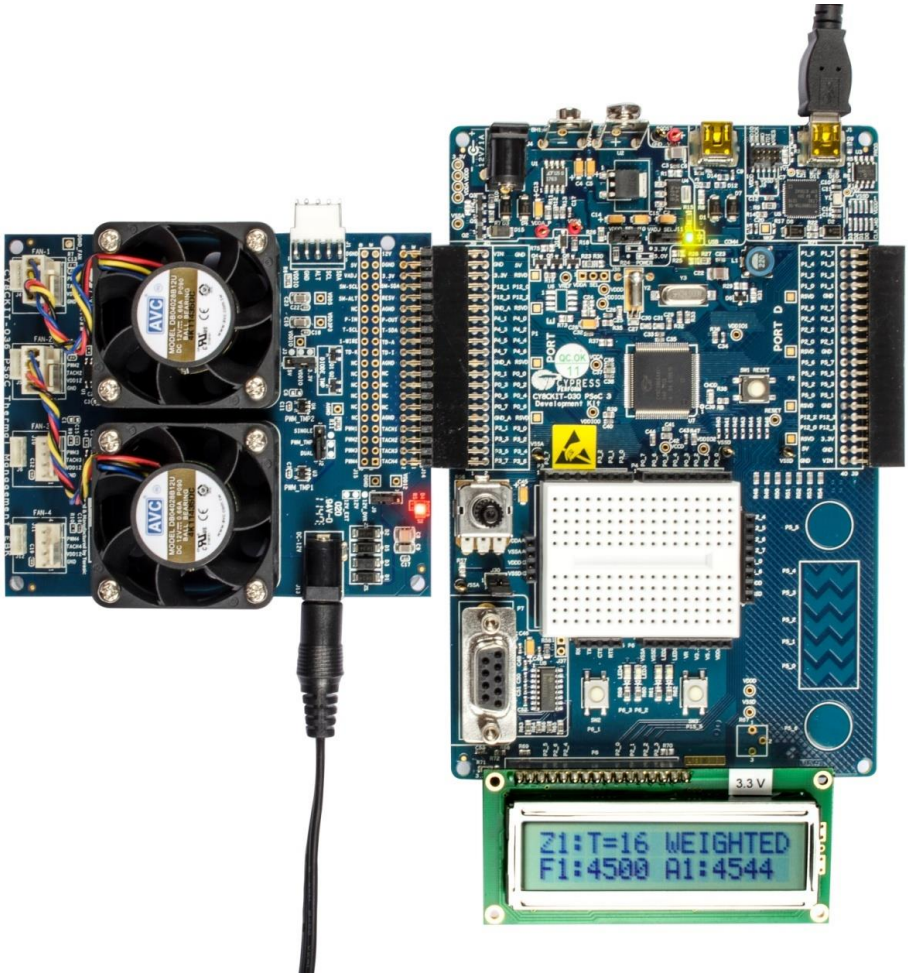
CY8CKIT-036 is designed to connect with the expansion ports of the different PSoC development kits so you can use one EBK to evaluate the entire portfolio of PSoC devices. Table 2 lists the hardware requirements for using CY8CKIT-036 with different PSoC devices.

Table 2. CY8CKIT-036 Hardware Requirement for PSoC 3 and PSoC 5LP

PSoC Device	Hardware Requirement
PSoC 3	CY8CKIT-001 PSoC DVK fitted with a PSoC CY8C38 Family Processor Module (MPN: CY8C3866AXI-040) Or CY8CKIT-030 PSoC 3 DVK
PSoC 5LP	CY8CKIT-001 PSoC DVK fitted with a PSoC CY8C58LP Family Processor Module (MPN: CY8C5868AXI-LP035) Or CY8CKIT-050 PSoC 5LP DVK

Figure 8 shows how the CY8CKIT-036 connects to CY8CKIT-030, a PSoC 3 DVK. The same connection is applicable for CY8CKIT-050 PSoC 5LP DVK. The EBK should be connected to Port E of the CY8CKIT-030 DVK. With the CY8CKIT-001 DVK, the EBK should be connected to Port A of the DVK.

Figure 8. CY8CKIT-036 EBK Connected to Port E of the CY8CKIT-036 EBK



The projects provided with this application note are designed to work with CY8CKIT-030 and CY8CKIT-050 DVKs by default. Changes should be made in the PSoC Creator project, as well as in the hardware connections, to evaluate the project in the CY8CKIT-001 DVK. Modifications are as follows:

- Connect the CY8CKIT-036 to Port A of the CY8CKIT-001 DVK. The pin assignments on the PSoC side for connecting CY8CKIT-036 EBK to Port A of the CY8CKIT-001 DVK and Port E of the CY8CKIT-030/CY8CKIT-050 DVK are the same. So, no pin assignment changes need to be made with respect to interfacing with CY8CKIT-036 EBK in the PSoC Creator project.
- The connections of the push buttons, LEDs, and potentiometer are hardwired to fixed PSoC pins in CY8CKIT-030/
  CY8CKIT-050 DVKs, whereas for CY8CKIT-001 DVK external wiring needs to be done for using push buttons, LEDs, and a potentiometer. The knowledge base article at www.cypress.com/?id=4&rID=51598 highlights the differences between the CY8CKIT-001 and CY8CKIT-030 kits. The application projects that use push button switches, LEDs, and a potentiometer need the pin assignments modified when porting the project to the CY8CKIT-001 DVK. No other changes are required in the project.
- Apply the following jumper settings in the CY8CKIT-036 EBK for evaluating the example projects:

- The CY8CKIT-036 EBK includes a 12-V DC high-current power supply that is capable of supplying the inrush current needed by the fans installed on that kit. Use that power supply connected to the power connector (J13), and set the power jumper (J9) on the CY8CKIT-036 EBK board to **12V_EXT** (the default setting), as Figure 9 shows.
- Power the PSoC interface circuitry on the CY8CKIT-036 (sensors and fan tachometer signals) to run at 3.3 V by setting jumper J3 for VDDIO at 3.3 V, as Figure 10 shows. This assumes that the PSoC chip in the DVK is also configured to operate at 3.3 V. If the PSoC chip is running at 5 V, set J3 for 5 V.

Figure 9. 12 V Power Supply Selection (Jumper J9)

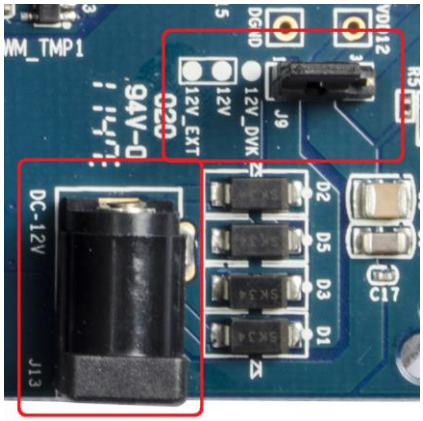
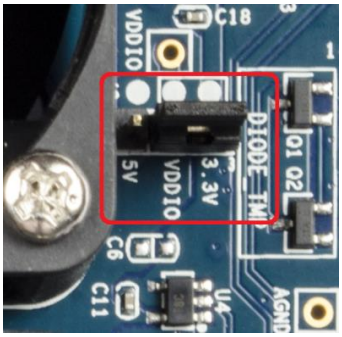


Figure 10. VDDIO Selection on KIT-036 Using Jumper J3



6 Example 1: Open-Loop Fan Control

Distributed with this application note is a compressed file that contains seven PSoC Creator example projects. Save the file to a convenient location on your hard drive and extract the contents to a local folder.

To get started with the example projects, double-click the *AN66627.cywrk* PSoC Creator workspace file.

1. In the **Workspace Explorer** tab to the left of the screen, expand project **1_OpenLoop** by clicking the small '+' icon to the left of it.
2. Double-click *TopDesign.cysch* to open the top-level design schematic for the hardware blocks inside PSoC.

Figure 11 shows the top-level design schematic of the project.

All of the example projects described in this application note are designed to run on either the CY8CKIT-030 PSoC 3 DVK or the CY8CKIT-050 PSoC 5LP DVK platform. The simplest way to get started is to attach the CY8CKIT-036 PSoC Thermal Management Expansion Board Kit (EBK) to Port E of the DVK, as explained in the Cypress Development Kits for Thermal Management section.

If you don't have the CY8CKIT-036 EBK, you can still wire up your own fans to the DVK using the signals available on the Port E expansion port connector of the DVK. The signal labeled "VIN" can be used to provide +12 VDC power to your fans.

A ground reference is on the adjacent pin of the Port E expansion port connector. The fan speed control PWM signals and tachometer signals can also connect to that connector. Follow the wiring schematic shown in Figure 11.

Double-click the fan controller component to open the component customizer, as shown in Figure 12.

For the first example project, the fan controller component is configured to use the manual control mode. In this mode, PSoC hardware blocks are used to implement the PWMs and the tachometer speed monitor (Tach), but user firmware is entirely responsible for fan speed control.

Figure 11. Top-Level Design Schematic for Example Project 1

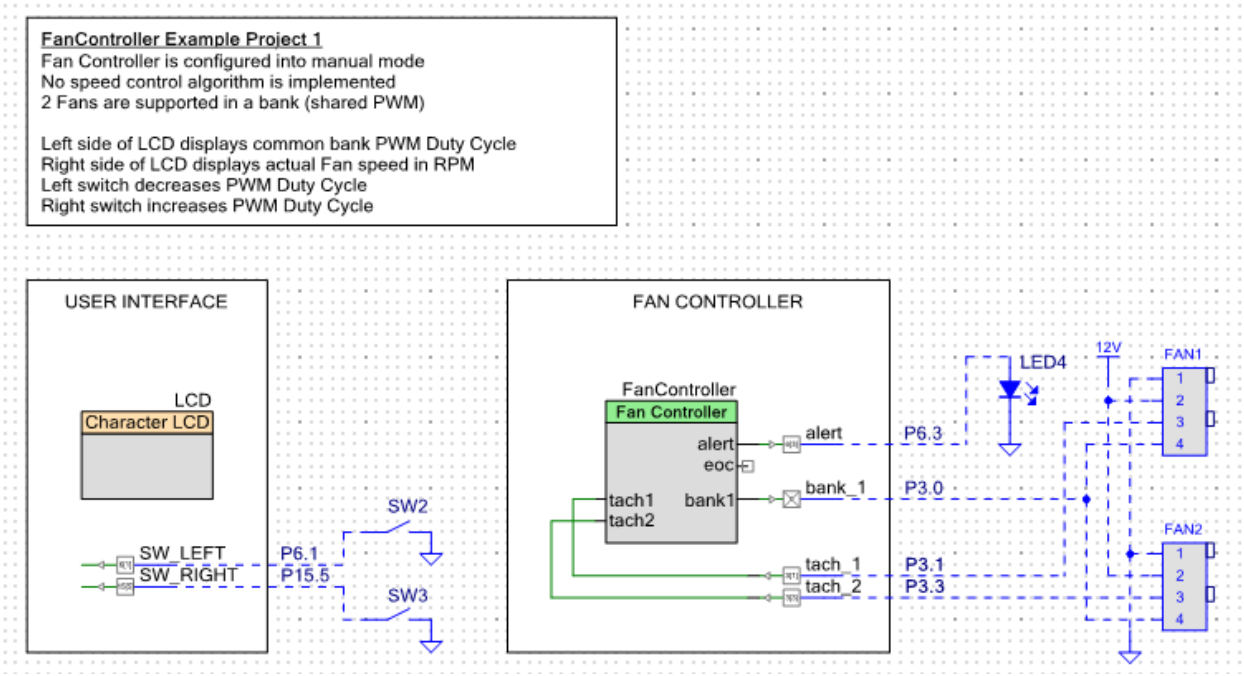


Figure 12. FanController Customizer: Basic Tab

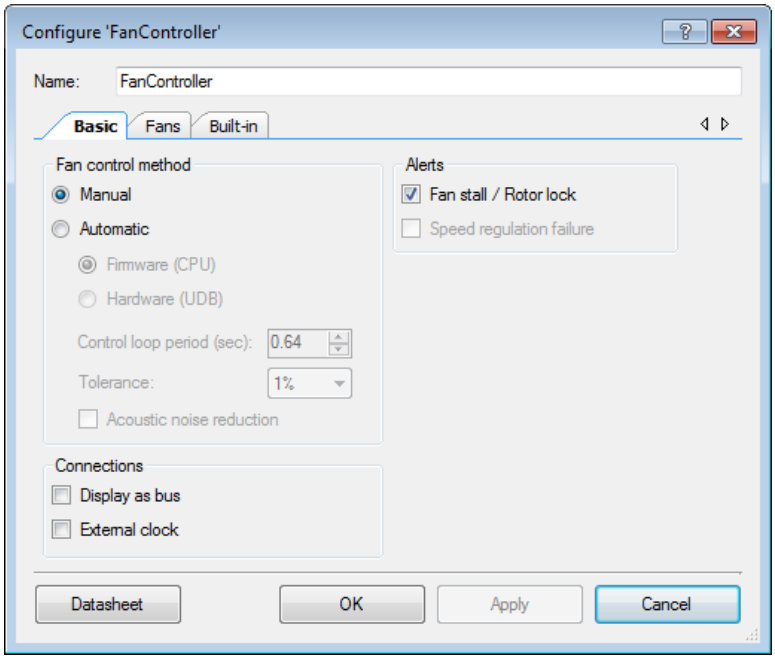
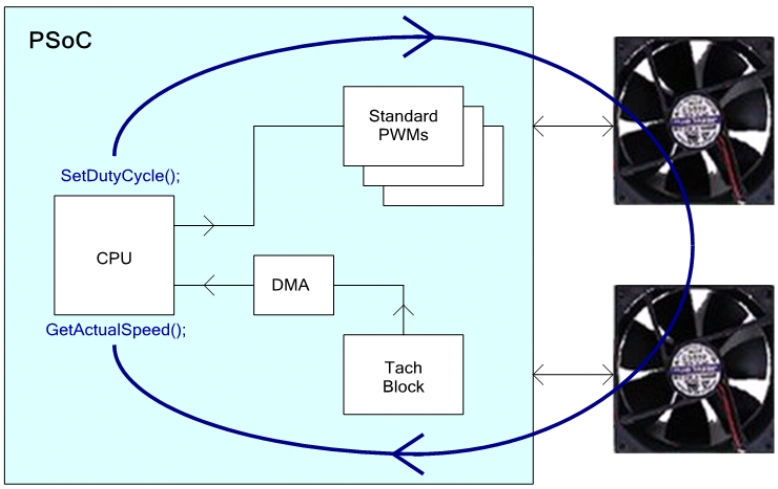


Figure 13 shows a simplified representation of the manual fan control mode. In this mode, firmware can set individual PWM duty cycles using the SetDutyCycle() API to change fan speeds. Actual speeds measured by the Tach block are transferred to SRAM using one channel of PSoC's DMA controller. Firmware can read individual fan speeds using the GetActualSpeed() API.

The Tach block can be configured to generate an alert signal if any of the fans stalls. This feature can be enabled by selecting the Fan Stall / Rotor Lock checkbox on the customizer **Basic** tab.

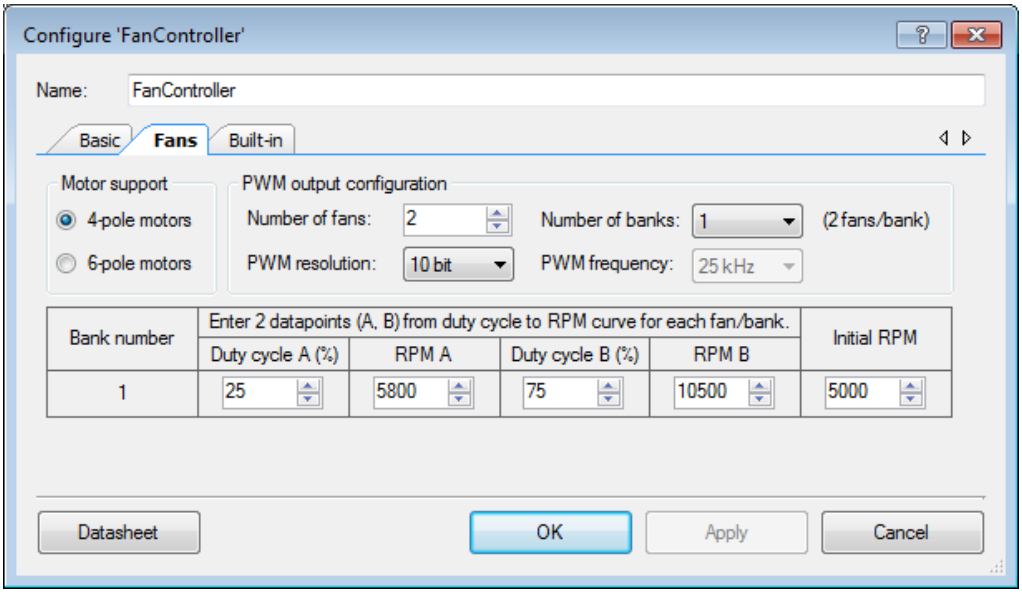
Figure 13. Firmware Fan Control Mode



Move to the **Fans** tab (see Figure 14) of the customizer to complete the configuration of the fan controller component. This tab enables you to configure all of the parameters related to the PWM fan drivers. You can enter the number of fans and the banking arrangement of the fans.

A fan bank is defined as multiple fans that are driven by the same PWM speed control signal. For the first example, we will drive two fans with the same PWM signal representing one bank of fans.

Figure 14. FanController Customizer: Fans Tab



The resolution of the PWM drivers can be set to 8 bit or 10 bit as required for your application. The 10-bit resolution gives finer speed control but uses more digital resources in the application. When you select 8-bit resolution, you can set the PWM frequency to either 25 kHz or 50 kHz. The industry standard for PWM drive signals is 25 kHz, but fans that support higher PWM control frequencies can be supported with this control.

The final section of this tab enables you to enter the electromechanical parameters of the fans you are working with. If you have this information from the fan manufacturer's datasheet, enter it here. The Duty A (%), rpm A, Duty B (%), and rpm B parameters represent two data points from the fan's duty-cycle-to-speed chart. If you are not able to get this information for the fans you are working with, stick with the default settings for now and you will be able to measure the actual duty-cycle-to-speed relationship for your fans in the first example project. After you have captured the data, you can return to the customizer and enter those parameters later. In this example project, the information provided by the fan manufacturer in the fan datasheet was entered in the component customizer.

Now that the component configuration is complete, program the project into the DVK by selecting **Debug** > **Program** from the pull-down menus. Note that all of the example projects will run on PSoC 3 or PSoC 5LP. The projects are configured for PSoC 3 by default. To switch to PSoC 5LP, simply change the target device using the Device Selector and rebuild the project to target the selected device.

If the project is running correctly, the text displayed on the debug LCD should display something like this:

```
50%    RPM1:     3250
Bank   RPM2:     3674
```

Note: The same PWM duty cycle is being driven to both fans on PSoC port pin P3.0. Be sure to connect this signal to the PWM speed control pin on both fans for this example. If the CY8CKIT-036 is connected to Port E of the CY8CKIT-030 or CY8CKIT-050 DVK, you can short pins 3.0 and 3.2 on the DVK prototype area to ensure the banked PWM signal drives both fans.

In the **Workspace Explorer**, double-click *main.c* to examine the firmware used in this example project. The code is shown in the Code 1 excerpt.

The purpose of this first example is twofold:

- To get a sense of the accuracy of the fans. It is common for two "identical" fans to spin at very different speeds, even when they are both driven with the same PWM duty cycle. This highlights that driving fans using conventional banks (a shared PWM drive signals across multiple fans) is far from optimal. When we explore the closed-loop hardware control mode in later examples, we will revisit this topic.
- To capture the duty-cycle-to-rpm relationship if the fan manufacturer's datasheet is not available.

Looking at the code excerpt from example 1, notice that no speed control algorithm has been implemented. Pressing the SW2 switch on the DVK decreases the duty cycle by 5 percent. Doing so should result in a reduction in speed of both fans. Pressing the SW3 switch on the DVK increases the duty cycle by 5 percent.

This example also introduces some of the basic APIs specific to the fan controller component. Refer to the comments in the code excerpt to understand the relevance of the API calls and the component datasheet for a full list of APIs, their parameters, return values, and operation.

Code 1. Example 1 *main.c* Firmware Listing

```
        * Duty cycles expressed in percent */

#define   MIN_DUTY    0
#define   MAX_DUTY    100
#define   INIT_DUTY   50
#define   DUTY_STEP   5

void main()
{
     uint16  dutyCycle = INIT_DUTY;

     /* Initialize the Fan Controller */
     FanController_Start();

     /* API uses Duty Cycles Expressed in
        Hundredths of a Percent */
     FanController_SetDutyCycle
     (1, dutyCycle*100);

     /* Initialize the LCD */
     LCD_Start();
     LCD_Position(0,0);
     LCD_PrintDecUint16(dutyCycle);
     LCD_PrintString("%");
     LCD_Position(1,0);
     LCD_PrintString("Bank");

     while(1)
     {
         /* Display Actual Speed Readings */
         LCD_Position(0,5);
         LCD_PrintString("RPM1: ");
         LCD_PrintDecUint16(
          FanController_GetActualSpeed(1));
         LCD_PrintString("     ");

         LCD_Position(1,5);
         LCD_PrintString("RPM2: ");
        LCD_PrintDecUint16(
          FanController_GetActualSpeed(2));
         LCD_PrintString("    ");
         CyDelay(250);

         /* Check for Button Press to Change
            Duty Cycle */
         if((!SW_LEFT_Read()) ||
            (!SW_RIGHT_Read()))
         {
             /* Left Switch = Decrease
                Duty Cycle */
             if(!SW_LEFT_Read())
             {
                if(dutyCycle > MIN_DUTY)
                {
                    dutyCycle -= DUTY_STEP;
                }
             }
```

```
            /* Right Switch = Increase
               Duty Cycle */
            else
            {
               if((dutyCycle += DUTY_STEP) >
                   MAX_DUTY)
               {
                 dutyCycle = MAX_DUTY;
               }
            }

            /* Adjust Duty Cycle of the Fan
               Bank */
            FanController_SetDutyCycle(1,
              dutyCycle*100);
            LCD_Position(0,0);
            LCD_PrintDecUint16(dutyCycle);
            LCD_PrintString("%  ");

            /* Switch Debounce *.
            CyDelay(250);
        }
    }
```

7 Example 2: Firmware Speed Control

Go to the Workspace Explorer and right-click **Project 2_FirmwareClosedLoop**. Select **Set As Active Project**.

Open *TopDesign.cysch* to start working on this project. The main change made to the top-level design schematic is the addition of a Status Register component, available in the standard **Cypress Component Catalog**, that the firmware can use to synchronize with hardware.

The purpose of this example is to use the same basic hardware configuration as the first example, but to add a user-defined firmware-based closed-loop speed control. From example project 2 onward, the fans will have individual PWM control. Add a wire to the DVK to connect the second fan's PWM control input to PSoC Port P3.2. This is shown in the top-level schematic of the project.

Open the component customizer and go to the **Fans** tab. The only required change is to remove fan banking by setting the number of banks to 0. Doing so will expose an additional row to enter the electromechanical parameters for the second fan. This is a good time to enter the fan parameters for the fans that you measured in example project 1. Now that you are setting up the component to drive two fans independently, it is possible to enter a different duty-cycle-to-rpm relationship for each fan. This capability allows the fan controller component to work with any combination of fans, giving system designers greater flexibility to mix and match different types of fans in their applications to meet system cooling needs.

Now that the component configuration is complete, program the project into the DVK by selecting **Debug > Program** from the pull-down menus.

If the project is running correctly, the text on the debug LCD should display something like this:

```
5000   5245   34.50%
F/W    4850   32.75%
```

The desired speed in rpm is displayed on the left. Pressing the left switch on the DVK decreases the desired speed by 500 rpm (the right switch increases by 500 rpm). A firmware algorithm responds by adjusting the duty cycle for both fans and works continuously at fine-tuning the duty cycle until the actual fan speeds approach the desired speed.

The center of the display shows the actual speed in rpms of both fans. If it displays zero, check the connection of the Tach signals from the fans. The right side of the display shows the duty cycle for each fan. In contrast to the first example, where the fans were driven with the same duty cycle and we observed the difference in actual speeds, this example shows the difference in duty cycles required to achieve the same desired speed on both fans.

The “F/W” text displayed at the bottom left of the display highlights that this project is using firmware speed control. This is done because the LCD display in the next example project (when hardware closed-loop control is introduced) is virtually identical, so this text helps identify which example project is currently running on PSoC.

In the Workspace Explorer, double-click *main.c* to examine the firmware used in this example project. The code is shown in the Code 2 excerpt.

Code 2. Example 2 *main.c* Firmware Listing

```
        /* PWM duty cycle controls - units are hundredths of a percent */
#define MIN_DUTY            50
#define MAX_DUTY          9950
#define DUTY_STEP_COARSE   100
#define DUTY_STEP_FINE       2

        /* Speed controls - units are RPM */
#define MIN_RPM           2500
#define MAX_RPM           9500
#define INIT_RPM          4500
#define RPM_STEP           500
#define RPM_DELTA_LARGE    500
#define RPM_TOLERANCE      100

void main()
{
      uint16  desiredSpeed = INIT_RPM;
      uint16  dutyCycle[2];
      uint8   fanNumber;
      char    displayString[6];

      FanController_Start();
      FanController_SetDesiredSpeed(1, desiredSpeed);
      FanController_SetDesiredSpeed(2, desiredSpeed);
      dutyCycle[0] = FanController_GetDutyCycle(1);
      dutyCycle[1] = FanController_GetDutyCycle(2);

      LCD_Start();
      LCD_Position(0,0);
      LCD_PrintDecUint16(desiredSpeed);
      LCD_Position(1,0);
      LCD_PrintString("F/W");

      while(1)
      {
            /* Synchronize firmware to end-of-cycle pulse from FanController */
            if (eocStatus_Read())
            {
                  for(fanNumber = 1; fanNumber <= 2; fanNumber++)
                  {
                        /* Display Fan Actual Speeds */
                        LCD_Position(fanNumber-1,5);
                        LCD_PrintDecUint16(FanController_GetActualSpeed(fanNumber));
                        LCD_PrintString("    ");
                        LCD_Position(fanNumber-1,9);

                        /* Fan Below Desired Speed */
                       if(FanController_GetActualSpeed(fanNumber) < desiredSpeed)
                       {
                            if((desiredSpeed - FanController_GetActualSpeed(fanNumber)) >
   RPM_DELTA_LARGE)
                            {
                                dutyCycle[fanNumber-1] += DUTY_STEP_COARSE;
                            }
                            else
                            {
                                dutyCycle[fanNumber-1] += DUTY_STEP_FINE;
                            }
                            if(dutyCycle[fanNumber-1] > MAX_DUTY)
                            {
                                dutyCycle[fanNumber-1] = MAX_DUTY;
                            }
```

```
                    }
                    /* Fan Above Desired Speed */
                    else if(FanController_GetActualSpeed(fanNumber) > desiredSpeed)
                    {
                        if((FanController_GetActualSpeed(fanNumber) - desiredSpeed) >
        RPM_DELTA_LARGE)
                        {
                            if(dutyCycle[fanNumber-1] > (MIN_DUTY+DUTY_STEP_COARSE))
                            {
                                dutyCycle[fanNumber-1] -= DUTY_STEP_COARSE;
                            }
                        }
                        else if((FanController_GetActualSpeed(fanNumber) - desiredSpeed) >
        RPM_TOLERANCE)
                        {
                            if(dutyCycle[fanNumber-1] > MIN_DUTY)
                            {
                                dutyCycle[fanNumber-1] -= DUTY_STEP_FINE;
                            }
                        }
                    }
                    FanController_SetDutyCycle(fanNumber, dutyCycle[fanNumber-1]);
                    /* Display Current Duty Cycle Settings (in 100ths of a percent) */
                    LCD_Position(fanNumber-1,10);
                    sprintf(displayString, "%5.2f", (((float)dutyCycle[fanNumber-1])/100));
                    LCD_PrintString(displayString);
                    LCD_PrintString("%    ");
                }
                CyDelay(250);

                /* Check for Button Press to Change Speed */
                if((!SW_LEFT_Read()) || (!SW_RIGHT_Read()))
                {
                    /* Decrease Speed */
                    if(!SW_LEFT_Read())
                    {
                        if(desiredSpeed > MIN_RPM)
                        {
                            desiredSpeed -= RPM_STEP;
                        }
                    }

                    /* Increase Speed */
                    else
                    {
                        desiredSpeed += RPM_STEP;
                        if(desiredSpeed > MAX_RPM)
                        {
                            desiredSpeed = MAX_RPM;
                        }
                    }
                    /* Display Updated Desired Speed */
                    LCD_Position(0,0);
                    LCD_PrintDecUint16(desiredSpeed);
                    FanController_SetDesiredSpeed(1, desiredSpeed);
                    FanController_SetDesiredSpeed(2, desiredSpeed);
                    dutyCycle[0] = FanController_GetDutyCycle(1);
                    dutyCycle[1] = FanController_GetDutyCycle(2);

                    /* Switch Debounce */
                    CyDelay(250);
                }
            }
        }
    }
```

Because the time taken to measure the actual speed from each fan is long (between 12 and 18 ms per fan at 5000 rpm, for example), it is important that the firmware speed control algorithm knows when new actual speeds are available before making the next decision about how to adjust the PWM duty cycles. The end-of-cycle (EOC) output terminal on the fan controller component serves this purpose. The EOC signal is generated by the Tach block inside the component and pulses high when a new actual speed sample is available from all fans connected to the component.

In this example project, the main firmware loop polls the status register component waiting for the EOC pulse from the fan controller component before running the duty cycle adjustment algorithm.

8 Example 3: Hardware Speed Control

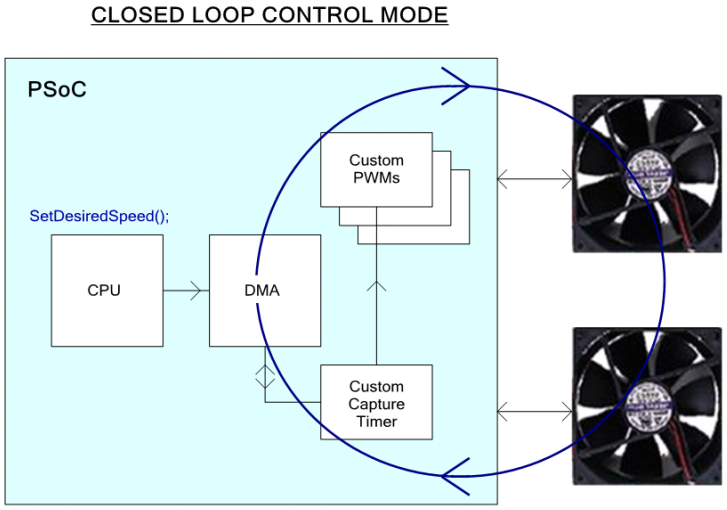
Go to the Workspace Explorer and right-click **Project 3_HardwareClosedLoop**. Select **Set As Active Project**.

Open *TopDesign.cysch* to start working on this project. The only change made to the top-level design schematic is the removal of the EOC Status Register component; firmware is not involved in the speed control in this project.

The purpose of this example is to replicate the functionality of the firmware-based fan control project. The custom hardware blocks inside the fan controller component perform the task of speed regulation, which frees the CPU to perform other tasks.

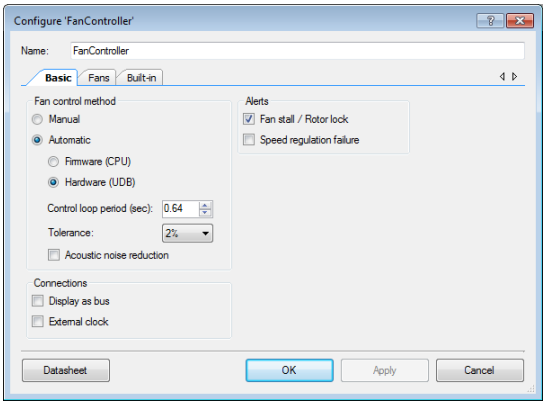
Figure 15 shows a simplified representation of the hardware fan control mode, in which firmware can set desired fan speeds using the SetDesiredSpeed() API. The UDBs inside the component will then work together to measure fan speeds and adjust duty cycles automatically. Actual fan speeds are still transferred to memory by the Tach block, which enables firmware to monitor fan speeds as a background activity if desired.

Figure 15. Hardware Fan Control Mode



Double-click the fan controller component to open the component customizer, as shown in Figure 16.

Figure 16. Configuring Closed-Loop Control in the FanController Component Customizer



Enable the hardware fan control mode by clicking the **Hardware (UDB)** radio button on the **Basic** tab under the **Automatic** option. Doing so exposes some new controls that can be used to optimize the behavior of the closed loop hardware system. You can use the **Firmware (CPU)** option to implement a firmware-based Proportional Integral Derivative (PID) control instead of a user-written firmware control. Refer to the fan controller component datasheet for details on this mode operation.

8.1 Control Loop Period

This parameter controls the dynamic response time of the hardware control loop. It controls how frequently the hardware will adjust the PWM duty cycles for each fan.

In situations in which there are only a few fans, a higher control loop period will ensure that the control loop can regulate the desired speed without oscillating or hunting around the desired speed target. In situations in which many fans are being controlled, a lower control loop period will ensure adequate response time to changes in fan speed. This parameter enables fine-tuning of the hardware control logic to match the selected fan's electromechanical characteristics.

The valid range for this parameter is 0 to 2.55 seconds. In a subsequent example project, we will add an I^2C interface to the fan controller component to enable real-time monitoring of the fan speeds and the duty cycle adjustments made by the hardware control loop. That setup enables designers to experiment with different control loop period settings until the desired performance is achieved.

8.2 Tolerance

This parameter sets the acceptable tolerance when specifying desired fan speed targets. The tolerance is specified as a percentage relative to the desired speed setting. This parameter also enables fine-tuning of the hardware control logic to match the selected fan's electromechanical characteristics.

The valid range for this parameter is 1 to 10 percent. The default setting is 1 percent. A tolerance parameter setting of 5 percent is recommended for an 8-bit PWM resolution, which you can select from the fan controller **Fans** tab.

8.3 Acoustic Noise Reduction

This parameter limits audible noise from fans by limiting the positive rate of change of speed. If enabled, and if the firmware requests an increase in the desired fan speed, the PWM duty cycle applied to the fan will increase gradually to the new setting rather than applying a sudden step change. This eliminates noisy fan whir from sudden speed increases. For this example project, leave the control in the default setting; you don't need to select it. We will enable this parameter in later projects to see what effect it has on the hardware control loop.

8.4 Alerts

For this example project, enable the Fan Stall / Rotor Lock alert. The alert pin is connected to LED4 on the DVK, so you have a visual indication if a fan stalls during operation.

Now that the component configuration is complete, program the project into the DVK by selecting **Debug** > **Program** from the pull-down menus.

If the project is running correctly, the text displayed on the debug LCD should display something like this:

```
4500   5245    34.50%
H/W    4850    32.75%
```

The desired speed in rpm is displayed on the left. Pressing the left switch on the DVK decreases the desired speed by 500 rpm (the right switch increases by 500 rpm). The hardware blocks in PSoC respond by adjusting the duty cycle for both fans and continuously work at fine-tuning the duty cycle until the actual fan speeds approach the desired speed.

In the Workspace Explorer, double-click *main.c* to examine the firmware used in this example project. The code is shown in the Code 3 excerpt.

This project is configured to behave similarly to example project 2 to demonstrate the power and convenience of hardware-driven closed-loop control. The main loop in the firmware project deals only with the user interface—the switches for requesting changes to desired speed, the LCD to display fan information, and the LED to display alert status.

For testing purposes, a stall condition can be created by either forcing the fan to stop rotating or by removing the fan from the connector on the DVK. For safety reasons and to observe behavior in the case of a persistent fault, remove either one of the fan connections from the DVK. When the stall is detected, the LCD will display the word *STALL* in place of the actual speed reading and the red LED4 on the DVK will blink.

When an alert is generated, the firmware calls the GetFanStallStatus() API, which deasserts the alert pin. Because the stall condition is persistent, the alert is generated again the next time the Tach hardware block measures the fan speed. The 500-ms delay in the firmware loop ensures that LED4 stays lighted long enough for you to see it.

Code 3. Example 3 *main.c* Firmware Listing

```
#define MIN_RPM   2500
#define MAX_RPM   9500
#define INIT_RPM  4500
#define RPM_STEP   500

void main()
{
    uint16  desiredSpeed = INIT_RPM;
    uint16  stallStatus;
    uint8   fanNumber;
    char    displayString[6];

    FanController_Start();
    FanController_SetDesiredSpeed(1, desiredSpeed);
    FanController_SetDesiredSpeed(2, desiredSpeed);
    LCD_Start();
    LCD_Position(0,0);
    LCD_PrintDecUint16(desiredSpeed);
    LCD_Position(1,0);
    LCD_PrintString("H/W");

    while(1)
    {
        /* Display Fan Actual Speeds and Duty Cycles */
        for(fanNumber=1; fanNumber<=2; fanNumber++)
        {
            LCD_Position(fanNumber-1,5);
            LCD_PrintDecUint16(FanController_GetActualSpeed(fanNumber));
            LCD_PrintString(" ");
            LCD_Position(fanNumber-1,10);
            sprintf(displayString, "%5.2f",
(((float)FanController_GetDutyCycle(fanNumber))/100));
```

```
            LCD_PrintString(displayString);
            LCD_PrintString("%    ");

            /* Check for Fan Stall (poll alert status) */
            if(FanController_GetAlertSource())
            {
                stallStatus = FanController_GetFanStallStatus();
                if(stallStatus & fanNumber)
                {
                    LCD_Position(fanNumber-1,5);
                    LCD_PrintString("STALL");
                }
                CyDelay(250);
            }
        }

        /* Check for Button Press to Change Speed */
        if((!SW_LEFT_Read()) || (!SW_RIGHT_Read()))
        {
            /* Decrease Speed */
            if(!SW_LEFT_Read())
            {
                if(desiredSpeed > MIN_RPM)
                {
                    desiredSpeed -= RPM_STEP;
                }
            }
           /* Increase Speed */
            else
            {
                desiredSpeed += RPM_STEP;
                if(desiredSpeed > MAX_RPM)
                {
                    desiredSpeed = MAX_RPM;
                 }
            }
            FanController_SetDesiredSpeed(1, desiredSpeed);
            FanController_SetDesiredSpeed(2, desiredSpeed);

            /* Display Updated Desired Speed */
            LCD_Position(0,0);
            LCD_PrintDecUint16(desiredSpeed);

            /* Switch Debounce */
            CyDelay(250);
        }
    }
   }
```

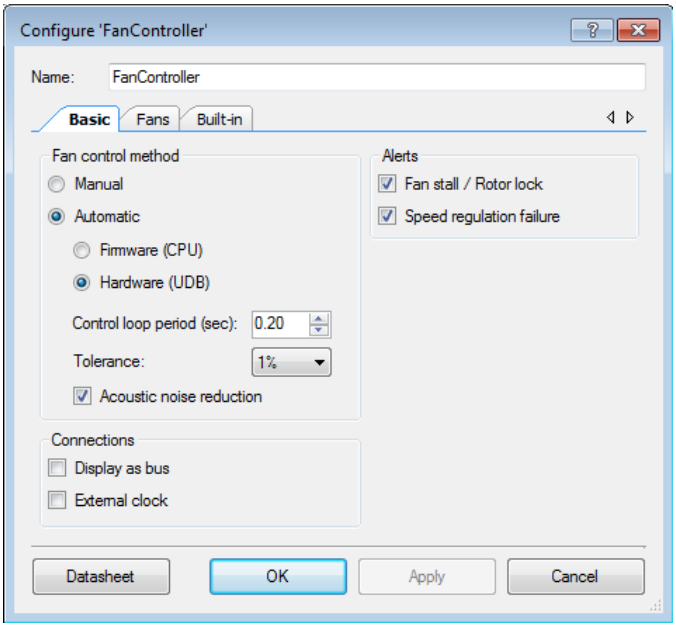
9 Example 4: Advanced Hardware Control

Go back to the Workspace Explorer and right-click **Project 4_AdvancedHardwareClosedLoop**. Select **Set As Active Project**.

Open *TopDesign.cysch* to start working on this project. The only change made to the top-level design schematic is the addition of an interrupt component, available in the standard **Cypress Component Catalog**.

Double-click the fan controller component to open the component customizer, as shown in Figure 17.

Figure 17. FanController Component Settings



The purpose of this project is to demonstrate the Acoustic Noise Reduction feature of the hardware speed control logic and how to use interrupts to respond to alerts instead of polling alert status in firmware. Select the **Acoustic Noise Reduction** checkbox to enable that feature.

9.1 Speed Regulation Failure Alerts

The closed-loop hardware controller generates speed regulation failure alerts in two cases:

- If the desired fan speed exceeds the current actual fan speed but the fan's duty cycle is already at 100 percent
- If the desired fan speed is below the current actual fan speed, but the fan's duty cycle is already at 0 percent

For this example project, select the **Speed Failure** checkbox in the **Alerts** section of the **Basic** customizer tab.

Now that the component configuration is complete, program the project into the DVK by selecting **Debug** > **Program** from the pull-down menus.

If the project is running correctly, the text displayed on the debug LCD should be the same as that observed in the previous hardware based control project.

If you left one or both of your fans disconnected from the DVK, you should see the word *STALL* on the LCD in place of the actual rpm. If so, this indicates that the CPU interrupts are working and being serviced properly. (This application note examines the interrupt service routine code later.) LED4 on the DVK is still connected to the alert pin. However, now that the servicing of the alert is handled by interrupts instead of polling, the response time is significantly faster. You may not be able to see the alert pin toggling. Connect an oscilloscope or logic analyzer to the alert pin on your DVK to see the operation and timing of that signal.

You can confirm that the speed regulation failure logic is working correctly in one of two ways:

- Set a desired speed beyond the maximum speed that your fan can achieve. To do that, simply change the MAX_RPM #define in *main.c* and use the right switch to increase the desired speed to that setting. When speed regulation is not possible, the LCD displays “SPEED” in place of the actual rpm.
- Set a desired speed below the minimum speed that your fan can achieve. To do that, simply change the MIN_RPM #define in *main.c* and use the left switch to decrease the desired speed to that setting. When speed regulation is not possible, the LCD will display “SPEED” in place of the actual rpm.

9.2 Firmware Details

In the Workspace Explorer, double-click *main.c* to examine the firmware used in this example project. The code is shown in the Code 4 excerpt.

Because this project requires interrupts to function, the CYGlobalIntEnable macro is called during initialization to enable interrupts to the CPU core. The AlertInt_Start() API is provided with the interrupt component and needs to be called to have an interrupt vector and priority assigned to that interrupt source.

The Code 5 excerpt shows the source code that needs to be manually entered into the API for the AlertInt component.

Near the top of the file, include the *FanController.h* file and reference the global variables defined in *main.c*. Then edit the AlertInt interrupt handler as shown.

The interrupt service routine for the alert signal calls the GetAlertSource() API to determine whether stall failures or speed regulation failures (or both) caused the alert. The GetFanStallStatus() or GetFanSpeedStatus() APIs return a bitmask with ‘1’s populated in each bit corresponding to the faulty fan number(s). These APIs also deassert the alert pin to prepare for future alerts.

A persistent stall failure or speed regulation failure will generate interrupts continuously at a rate dictated by the amount of time taken for the Tach block to measure the actual speeds of all fans in the system. In that situation, the designer may choose to disable alerts from a known-faulty fan to prevent the continuous assertion of interrupts to the CPU core. This is achieved using the SetAlertMask() API, which enables masking of alerts from the specified fan number. Masking alerts from a fan disables generating of both stall and speed regulation alerts from that fan until further notice.

Code 4. Example 4 *main.c* Firmware Listing

```
#define MIN_RPM    1000
#define MAX_RPM   20000
#define INIT_RPM   4500
#define RPM_STEP    500

uint16  stallStatus = 0;
uint16  speedStatus = 0;

void main()
{
    uint16 desiredSpeed = INIT_RPM;
    uint8  fanNumber;
    char   displayString[6];

    /* Globally Enable Interrupts to the CPU Core */
    CYGlobalIntEnable;

    FanController_Start();
    FanController_SetDesiredSpeed(1,
      desiredSpeed);
    FanController_SetDesiredSpeed(2,
      desiredSpeed);
    FanController_SetDutyCycle(1, 1000);
    FanController_SetDutyCycle(2, 1000);
    AlertInt_Start();

    LCD_Start();
    LCD_Position(0,0);
    LCD_PrintDecUint16(desiredSpeed);
    LCD_Position(1,0);
```

```
LCD_PrintString("H/W");

while(1)
{
    for(fanNumber=1; fanNumber<=2;
        fanNumber++)
    {
        /* Check for Fan Stall and Speed
          Failure (flags set in AlertInt
          ISR) */
        LCD_Position(fanNumber-1,5);
        if(stallStatus & fanNumber)
        {
            LCD_PrintString("STALL");
            stallStatus &= ~fanNumber;
            CyDelay(500);
        }

        else if(speedStatus & fanNumber)
        {
            LCD_PrintString("SPEED");
            speedStatus &= ~fanNumber;
            CyDelay(500);
        }
        else
        {
            /* Display Fan Actual Speeds
               When There are no Errors */
            LCD_PrintDecUint16(FanController_GetActualSpeed(fanNumber));
            LCD_PrintString(" ");
        }

        /* Always Display Fan Duty Cycles */
        LCD_Position(fanNumber-1,10);
        sprintf(displayString, "%5.2f",
          (((float)FanController_
             GetDutyCycle(fanNumber))/100));
        LCD_PrintString(displayString);
        LCD_PrintString("%    ");
    }

    /* Check for Button Press to Change
       Speed */
    if((!SW_LEFT_Read()) ||
       (!SW_RIGHT_Read()))
    {
        /* Decrease Speed */
        if(!SW_LEFT_Read())
        {
            if(desiredSpeed > MIN_RPM)
            {
                desiredSpeed -= RPM_STEP;
            }
        }

        /* Increase Speed */
        else
        {
            desiredSpeed += RPM_STEP;
            if(desiredSpeed > MAX_RPM)
            {
                desiredSpeed = MAX_RPM;
            }
        }
        FanController_SetDesiredSpeed(1,
          desiredSpeed);
        FanController_SetDesiredSpeed(2,
          desiredSpeed);
```

```
            /* Display Updated Desired Speed */
            LCD_Position(0,0);
            LCD_PrintDecUint16(desiredSpeed);

            /* Switch Debounce */
            CyDelay(250);
        }
    }
        }
```

Code 5. Example 4 *AlertInt.c* Firmware Listing

```
        /*****************************************
*  Place your includes, defines and code here
*****************************************/
/* `#START AlertInt_intc` */
#include "FanController.h"
extern uint16  stallStatus;
extern uint16  speedStatus;
/* `#END` */


CY_ISR(AlertInt_Interrupt)
{
     /*  Place your Interrupt code here. */
     /* `#START AlertInt_Interrupt` */

     uint8 alertStatus;
     if (alertStatus &
         FanController_STALL_ALERT)
     {
          StallStatus = FanController_
                        GetFanStallStatus();
     }

     /* If hardware UDB speed regulation failure
        alert, determine which fan(s) */
     if (alertStatus &
         FanController_SPEED_ALERT)
     {
          SpeedStatus = FanController_
                        GetFanSpeedStatus();
      }

    /* `#END` */
        }


     /* Determine alert source: stall or speed
        regulation failure (could be both) */
     alertStatus = FanController_
                   GetAlertSource();

     /* If stall alert, determine which
        fan(s) */
```

10 Example 5: Real-Time Monitoring

Go back to the Workspace Explorer and right-click **Project 5_I2CFanMonitoring**. Select **Set As Active Project**.

Open *TopDesign.cysch* to start working on this project. The changes made to the top-level design schematic is the removal of the LCD and the user interface switches, replaced by the addition of the EzI2C Slave component available in the standard **Cypress Component Catalog**.

The purpose of this project is to enable you to monitor the operation of the fan controller component in real time over I²C using a graphical charting GUI. This is useful for evaluating the dynamic performance of the closed-loop hardware controller and fine-tuning and optimizing the parameters set in the component customizer.

10.1 Firmware Details

In the Workspace Explorer, double-click *main.c* to examine the firmware used in this example project. The code is shown in the Code 6 excerpt.

This example highlights the benefit of using hardware closed-loop fan control. No code in the main firmware loop is responsible for controlling the fans. The firmware is responsible only for handling the I²C interface and translating I²C data to commands for the fan controller.

The example I²C data structure enables an I²C master to set desired speeds for each fan, and read actual speeds and current duty cycles.

A new API, OverrideHardwareControl(), is introduced in this example. This API enables designers to take back control from the closed-loop hardware and use firmware to control the fans. This could be useful in response to a fault condition or to implement a custom fan control algorithm temporarily. The same API can be used to pass fan control back to the hardware.

Program the project into the DVK by selecting **Debug > Program** from the pull-down menus. This project does not use the LCD, but it is not erased, so the LCD may still display data from Example 4.

Code 6. Example 5 *main.c* Firmware Listing

```
#define INIT_RPM          3000
#define NEW_COMMAND       0x0001
#define TOGGLE_OVERRIDE   0x8000

/* Define I2C Buffer Structure */
typedef struct
{
    /* R/W I2C variable - control word  */
    uint16 command;
    /* R/W I2C variables - desired speeds */
    uint16 desiredSpeed[2];
    /* R only I2C variables - actual speeds */
    uint16 actualSpeed[2];
    /* R only I2C variables - actual duty
       cycles*/
    uint16 dutyCycle[2];
} I2C_REGS;

void main()
{
    uint8 override = 0;

    /* Setup I2C Buffer */
    I2C_REGS i2cRegisters;
    I2C_Start();
    I2C_EnableInt();
    I2C_SetBuffer1(sizeof(i2cRegisters), 6 ,
                   (void *) &i2cRegisters);
    i2cRegisters.command = 0;
    i2cRegisters.desiredSpeed[0] = INIT_RPM;
    i2cRegisters.desiredSpeed[1] = INIT_RPM;

    /* Initialize Components */
```

```
CYGlobalIntEnable;
AlertInt_Start();
FanController_Start();
FanController_SetDesiredSpeed(1,
   i2cRegisters.desiredSpeed[0]);
FanController_SetDesiredSpeed(2,
   i2cRegisters.desiredSpeed[1]);

while(1)
{
    /* Synchronize firmware to hardware
       end-of-cycle pulse */
    if(eocStatus_Read())
    {
        /* Check if I2C Host Sent New
          Desired Speeds */
        if(i2cRegisters.command &
           NEW_COMMAND)
        {
            FanController_SetDesiredSpeed(1,
              i2cRegisters.desiredSpeed[0]);
            FanController_SetDesiredSpeed(2,
              i2cRegisters.desiredSpeed[1]);
            i2cRegisters.command &=
              ~NEW_COMMAND;
        }

        /* Check if I2C Host Sent Hardware
           Override Command */
        if(i2cRegisters.command &
           TOGGLE_OVERRIDE)
        {
            override ^= 1;
            FanController_Override
              HardwareControl(override);
            i2cRegisters.command &=
              ~TOGGLE_OVERRIDE;
        }

        /* Update I2C buffer with current
           Actual Speeds and Duty Cycles */
        i2cRegisters.actualSpeed[0] =
           FanController_GetActualSpeed(1);
        i2cRegisters.actualSpeed[1] =
           FanController_GetActualSpeed(2);
        i2cRegisters.dutyCycle[0]   =
           FanController_GetDutyCycle(1);
        i2cRegisters.dutyCycle[1]   =
           FanController_GetDutyCycle(2);
    }
}
```

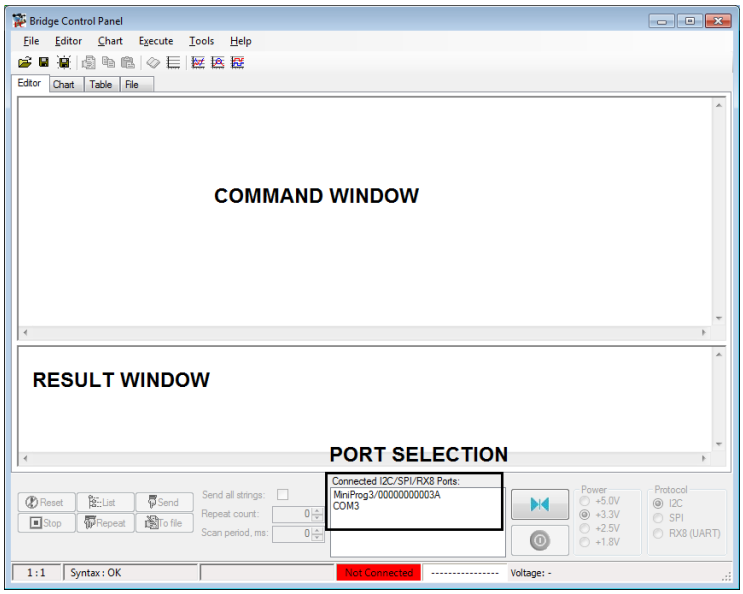
10.2 USB-to-I²C Bridge Control Panel

The MiniProg3 programmer is also capable of implementing a USB-to-I²C bridge adapter. We will take advantage of this feature to send and receive commands to PSoC over I²C.

Cypress Bridge Control Panel software is used to monitor fan speeds and to control fan speeds over the I²C interface. The Bridge Control Panel is a graphical user interface used to communicate with an I²C slave through an I²C-USB Bridge. If you are new to the Bridge Control Panel, see the Help documentation.

This software is installed automatically when the PSoC Programmer software is installed. You can open the software by going to **Start** > **All Programs** > **Cypress** > **Bridge Control Panel**. The GUI consists of three main controls: the port selection window, the command window, and the result window, as Figure 18 shows. When the MiniProg3 is connected to the PC, it is listed with its serial number in the connected ports window.

Figure 18. Bridge Control Panel

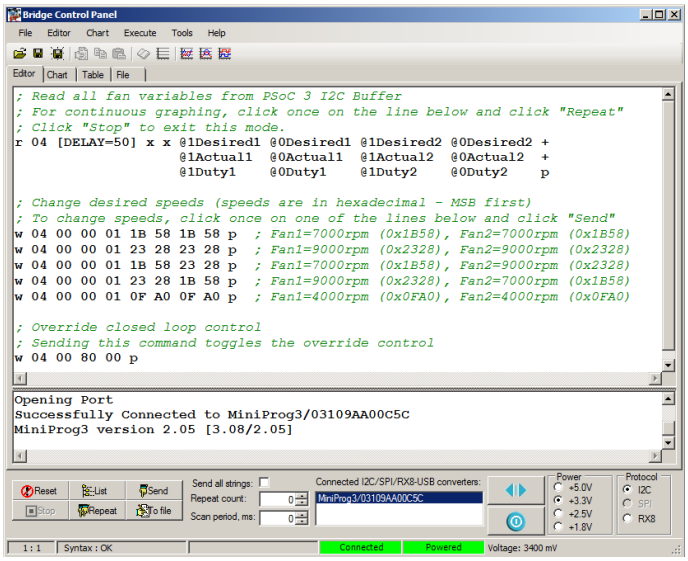


To send a command, you can either start typing in the command window or use the *.iic* file, which contains preset commands. Provided along with the example project are numerous files, located in the 5_I2CFanMonitoring folder, that are intended for use with the Bridge Control Panel software. The files *Fan_PSoC3.iic* and *Fan_PSoC5.iic* contain the commands for PSoC 3 and PSoC 5LP respectively. After launching the Bridge Control Panel software, use the **File** > **Open File** menu option to open the *.iic* files. After importing the *.iic* files, select **Chart** > **Variable Settings** from the pull-down menu and click **Load**. Open the *Fan.ini* file provided in the example project folder. This will be used to set the chart plotting options when using the software for fan speed monitoring.

Figure 19 shows how the Bridge Control Panel GUI should look when launched and properly configured.

Leave the DVK powered on, remove the MiniProg3 from the PSoC processor module, and connect the white 5-pin connector on the MiniProg3 to the 5-pin header on the CY8CKIT-036 PSoC Thermal Management EBK. If you don't have that kit, you will need to wire up the I²C signals as shown in the top-level schematic for this project. Ensure that you see the green Connected and Powered status boxes at the bottom of the Bridge Control Panel GUI. If the Powered status box does not light up immediately, check the "+3.3V" box.

Figure 19. Bridge Control Panel GUI



The **Editor** tab is exposed by default whenever the Bridge Control Panel software is opened. The editor provides a simple text scripting capability to send and receive I²C data. For full details on the scripting language, refer to the Help provided in the USB-to-I²C Bridge Control Panel software.

Code 7 shows the script provided for working with the example project.

Code 7. USB-to-I²C Bridge Control Panel Script

```
; Read all fan variables from PSoC 3 I2C Buffer
; For continuous graphing, click once on the
; line below and click "Repeat"
; Click "Stop" to exit this mode.
r 04 [DELAY=50] x x @1Desired1 @0Desired1 +                    @1Desired2 @0Desired2 +
                    @1Actual1  @0Actual1  +                    @1Actual2  @0Actual2  +
                    @1Duty1    @0Duty1    +  @1Duty2    @0Duty2    p

; Change desired speeds (speeds are in
; hexadecimal - MSB first)
; To change speeds, click once on one of the
; lines below and click "Send"

; Fan1=7000rpm (0x1B58), Fan2=7000rpm (0x1B58)
w 04 00 00 01 1B 58 1B 58 p
; Fan1=9000rpm (0x2328), Fan2=9000rpm (0x2328)
w 04 00 00 01 23 28 23 28 p
; Fan1=7000rpm (0x1B58), Fan2=9000rpm (0x2328)
w 04 00 00 01 1B 58 23 28 p
; Fan1=9000rpm (0x2328), Fan2=7000rpm (0x1B58)
w 04 00 00 01 23 28 1B 58 p
; Fan1=4000rpm (0x0FA0), Fan2=4000rpm (0x0FA0)
w 04 00 00 01 0F A0 0F A0 p

; Override closed loop control
; Sending this command toggles the override
; control
w 04 00 80 00 p
```

The format used in the script language is simple. The “r” at the beginning of a script line indicates an I²C read transfer. The next byte (in hexadecimal) is the I²C address in 7-bit format.

After the address is an optional delay parameter, specified in milliseconds. This is useful for setting a display update rate when continuous graphing is enabled.

The script line then contains a list of variable names set to match the I²C data structure defined in the example project. The “x x” indicates that the MiniProg3 should read the first 2 bytes from PSoC and discard them. Those first 2 bytes are used to send commands to PSoC, so you don’t need to read or view them here. We are interested in viewing the desired speeds, actual speeds, and duty cycles from the fan controller. Those are all 16-bit values transported over the native 8-bit I²C bus. The “@1” and “@0” prefixes to the variable names define the endian ordering of the bytes so that when we view the data in the chart GUI, the numbers will appear as 16-bit values.

Finally, the “p” at the end of each script line directs the MiniProg3 to generate an I²C stop condition that terminates the transfer.

Using this script file, we are able to accomplish several things:

- Read fan controller speeds and duty cycles in real time by continuously executing the first script line
- Set new desired speeds shown in the second section of the script file, as outlined in the comments
- Toggle between hardware closed-loop control mode and firmware control mode, as shown in the third section of the script file

The first exercise to gain familiarity with the Bridge Control Panel utility will be to display the actual speeds in real time and graph them over time. The power-on default speed for both fans was set at 3,000 rpm in *main.c*.

Single-click the script line beginning with “r” and then click the **Repeat** button at the bottom of the GUI. When you see the status text scrolling in the window at the bottom, click the **Chart** tab to see the results.

Figure 20 shows an example of a fan speeding up from stall with the parameter set to a low control loop period. The red line shows the target speed of 3,000 rpm. The green line shows the actual speed ramping up from zero, overshooting the target, and then oscillating a few times around the target speed before converging. The purple line shows the duty cycle controlled by the closed-loop controller hardware. The positive and negative duty cycle adjustments are evident.

Figure 21 shows an example of a fan speeding up from stall with the parameter set to a medium control loop period. The green line shows the actual speed ramping up from zero, overshooting once but then quickly settling back to the desired speed within tolerance.

Figure 22 shows an example of a fan speeding up from stall with the parameter set to a high control loop period. The green line shows the actual speed ramping up very slowly from zero, achieving the target speed and settling immediately.

Feel free to experiment with various combinations of the closed-loop hardware parameters. Don’t be alarmed if you initially find that the closed-loop controller cannot achieve regulation and the actual speed oscillates around the target desired speed. This is expected when few fans are connected and when a very low damping factor is set and/or a low tolerance setting is selected. Some tuning will be required to find the optimal set of control loop parameters.

Figure 20. Closed-Loop Fan Control: Speed-up from Stall (Low Control Loop Period)

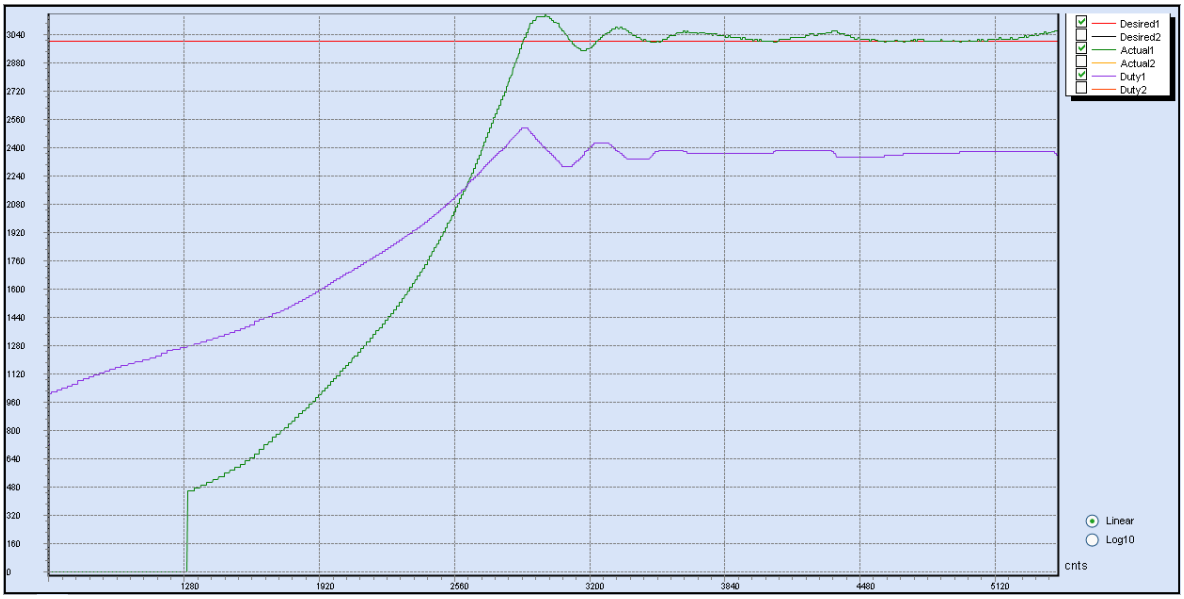


Figure 21. Closed-Loop Fan Control: Speed-up from Stall (Medium Control Loop Period)

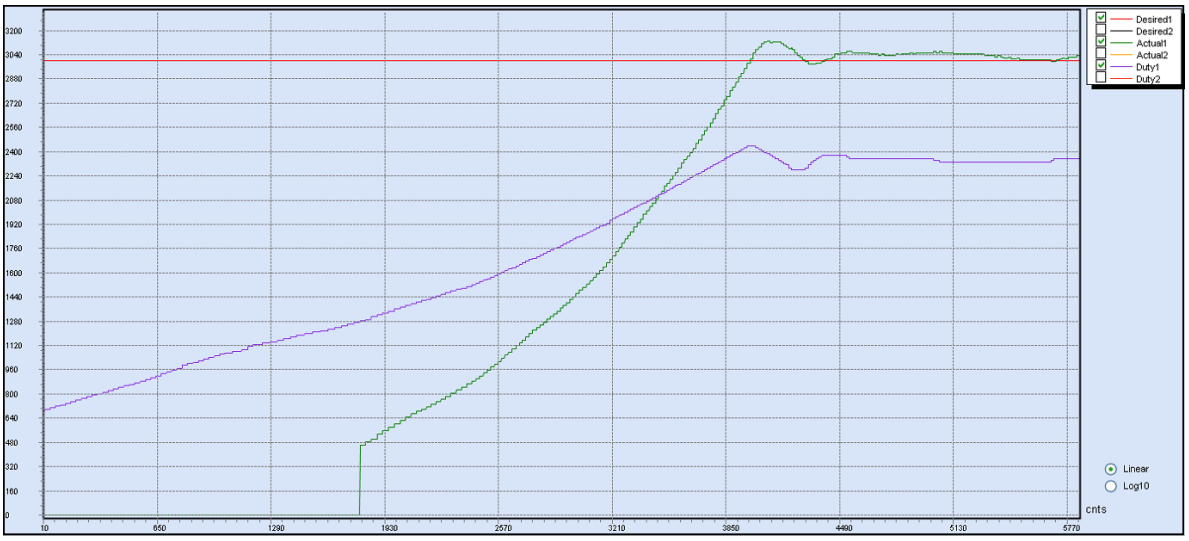


Figure 22. Closed-Loop Fan Control: Speed-up from Stall (High Control Loop Period)

The I²C write commands in the script enable you to independently change the desired speed for each fan. You can use these commands to see how the fan controller responds to speed changes and verify the dynamic behavior of the control loop. To send a command, click once on the script line and click the **Send** button or simply press Enter.

Remember that when acoustic noise reduction is enabled, the fan controller component limits the positive rate of change in fan speed. Changing the desired fan speed from 2,000 to 10,000 rpm results in a gradual speed change; this is far more pleasing to the ear than a sudden speed change that not only generates annoying fan whir noise but also causes sudden current surges in the fan's power supplies. Experiment with this feature by enabling or disabling it in the **Basic** tab of the component customizer.

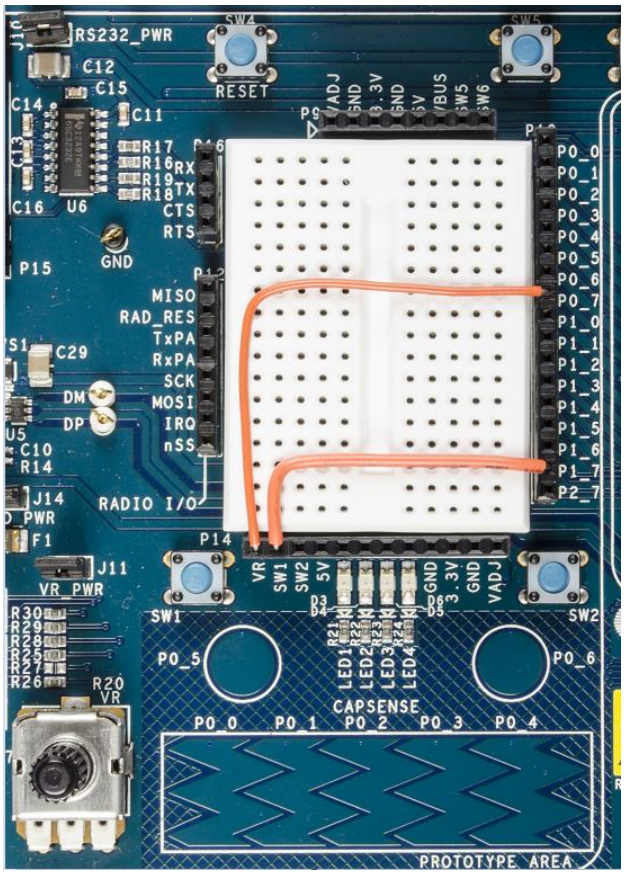
The I²C script file also enables you to turn off the closed-loop control mode by toggling the override control; you can see the difference in fan behavior when it is running in open-loop mode compared to closed-loop control mode. Disturbing the fan's airflow by either blocking the air intake or blowing air against the flow causes noticeable changes in fan speed in open-loop control mode. When you enable the closed-loop control mode, the fan controller will respond quickly and return the fans to their desired speed within tolerance.

11 Example 6: Thermal Management

Go back to the Workspace Explorer and right-click **Project 6_ThermalManagement_030_050**. Select **Set As Active Project**. This example project demonstrates how an entire thermal management application can be done using a single-chip PSoC 3 or PSoC 5LP device.

If you are using the CY8CKIT-001 DVK, select **Project 7_ThermalManagement_001**. The two projects are identical except for the pin assignment changes. To evaluate the thermal management project corresponding to the CY8CKIT-001 DVK, make external wire connections on the DVK for interfacing with the potentiometer and the switches. Connect the potentiometer output VR to pin P0[7], and the SW1 switch to pin P1[7], as shown in Figure 23. No such wiring connections are required on the CY8CKIT-030 or CY8CKIT-050 DVK, because the potentiometer and switches are hardwired to fixed port pins on those DVKs.

Figure 23. Wiring Connections on CY8CKIT-001 DVK



Fan speed control in the thermal management example project uses the thermal zone temperatures to which the fans belong. Each thermal zone has a set of fans and associated temperature sensors. The thermal zone temperature is calculated using the individual temperature sensors as inputs and applying an appropriate algorithm to calculate the zone temperature. A table in the firmware maps the zone temperature to the zone fan speed. This table is used to set the fan speed based on the zone temperature. The thermal management application calculates zone temperature measurements and sets the zone fan speed periodically.

11.1 Project Description

This project displays the zone temperature calculated by using the individual temperature sensor measurements. A character LCD display shows the thermal zone properties; you can change the menu on the LCD using a push button.

A digital temperature sensor (TMP175 I²C interface temperature sensor) and an analog temperature sensor (simulated using a potentiometer) are used as the individual temperature sensors. Two thermal zones are managed in this example project. Both are associated with the previously mentioned temperature sensors. However, the zone algorithms use different weighting factors on the sensor temperatures to calculate the zone temperature according to the following formulae:

$$Zone\ 1\ Temperature, Z1 = \frac{1}{10}\ T1\ +\ \frac{9}{10}\ T2$$

$$Zone\ 2\ Temperature, Z2 = \frac{99}{100}\ T1\ +\ \frac{1}{100}\ T2$$

Where T1 is the I²C temperature sensor temperature, T2 is the analog temperature sensor temperature (simulated using a potentiometer).

According to the formulae, the Zone 1 temperature is influenced heavily by the I²C temperature sensor (T1), whereas the Zone 2 temperature is influenced heavily by the analog temperature sensor (T2). From an end-application standpoint, the weighting factors can be considered as indicating the proximity of the temperature sensor to the thermal zone hotspot.

Once the zone temperatures are calculated, the speed of the fans associated with the thermal zone is set according to the table stored in the PSoC flash memory. The fan controller component ensures that the actual fan speeds are close to the desired fan speeds by modifying the duty cycle of the PWM signal to the fan. In this project, one fan is assigned for each thermal zone, for a total of two fans. Figure 24 and Figure 25 show the mapping between the zone temperature and the corresponding zone fan speed. When the zone temperature falls, a hysteretic control is applied to ensure that the fan speed does not alternate between two set points around the transition points of the thermal profile. When the zone temperature decreases, the fan speed is modified only when the temperature goes below the hysteresis-subtracted temperature value.

Figure 24. Zone 1 Thermal Profile

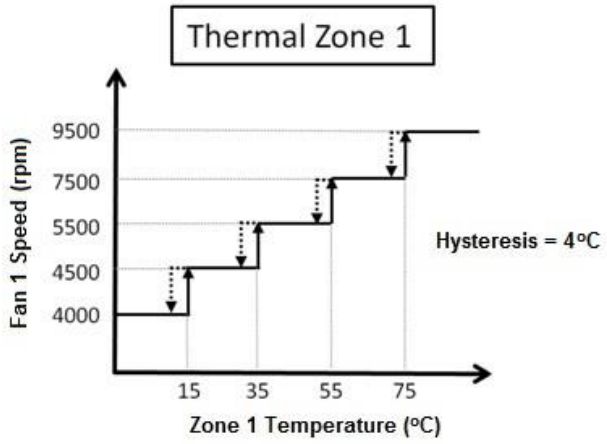
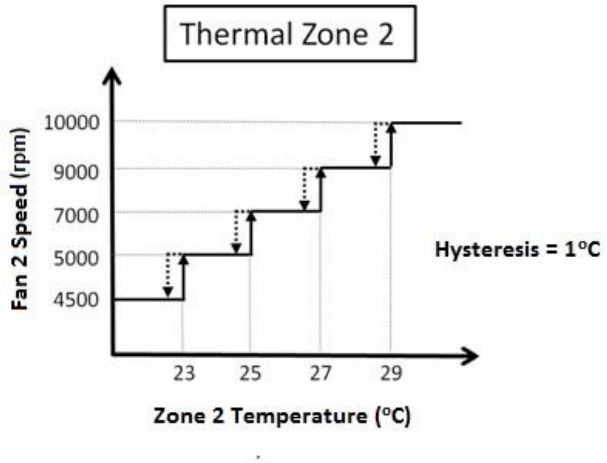


Figure 25. Zone 2 Thermal Profile

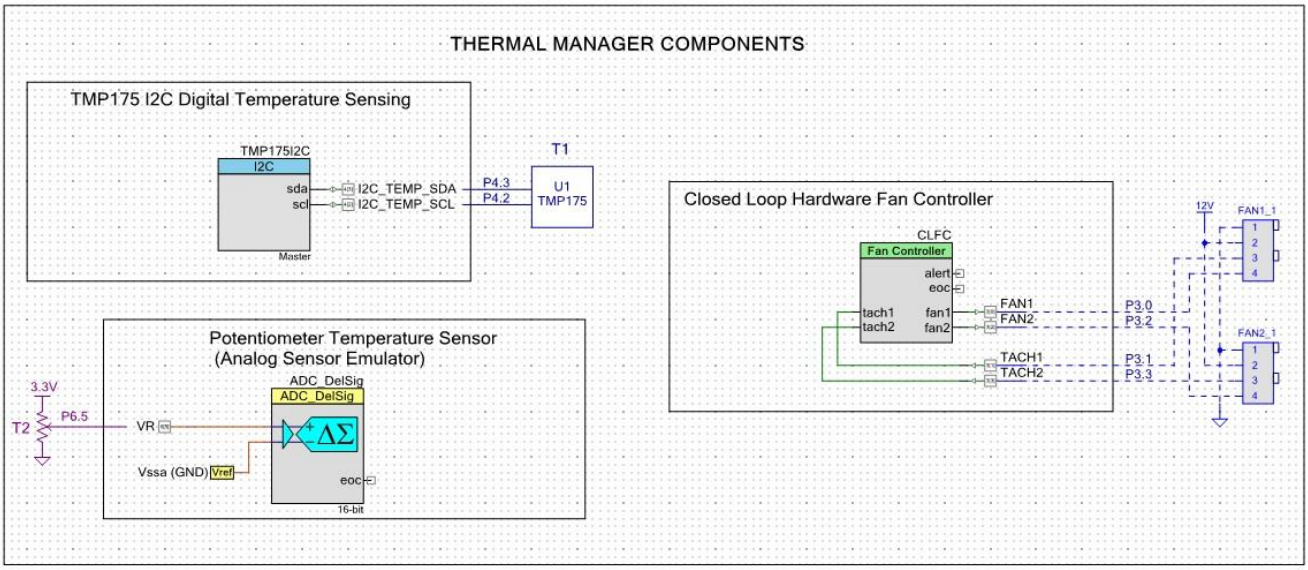


11.2 Project Schematic

Figure 26 provides a top-design schematic of the project, showing just the thermal manager components.

In the thermal management project, a potentiometer simulates an analog temperature sensor and the Delta-Sigma ADC in the PSoC chip measures the voltage. The ADC output is scaled appropriately in firmware to get the temperature value. A TMP175 I²C temperature sensor acts as the digital sensor, and an I²C master in the PSoC chip reads the temperature value. The composite zone temperature is calculated using the individual temperature-sensor values. The fan speed is determined, based on a zone-temperature-to-zone-fan-speed table stored in the PSoC flash memory. The desired fan speed provides input to the fan controller component in the PSoC device, which does a closed-loop control on the two fans to drive them at the desired speed.

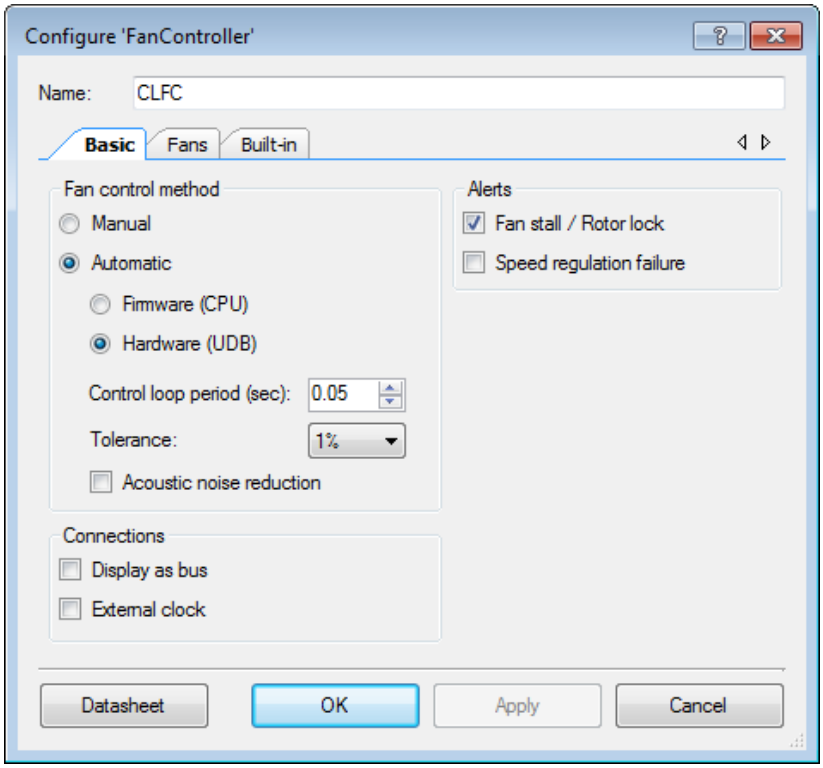
Figure 26. Top-Design Schematic of the Thermal Management Project



11.3 Component Configuration

Figure 27 and Figure 28 show how to configure the fan controller component in PSoC Creator.

Figure 27. Fan Controller Component “Basic” Tab



The fan control mode is set to Automatic, which means that the component itself ensures that the fans run at the desired speed. When you select the Hardware (UDB) control mode, hardware blocks inside the PSoC chip (universal digital blocks or DMA channels) implement a fan speed control loop. The other option is Firmware; in this case, a control loop implementing a proportional-integral-derivative (PID) control runs in the firmware to control fan speed. The control loop period is set to 50 ms, and the alert signal is enabled for detecting fan stall conditions. Refer to the fan controller component datasheet for further information on the component.

Figure 28 shows the configuration of the **Fans** tab of the fan controller component. This tab sets the characteristics of the fan being driven, including four-pole or six-pole motor support, the PWM resolution, and the frequency of the PWM signal. Another important setting is the duty-cycle-to-rpm relationship for each fan. Enter values based on the specifications given in the fan manufacturer datasheet. Enter two sets of values from the linear region of the duty-cycle-rpm curve in the component GUI. The fan controller component uses these values to determine the initial duty cycle of the PWM for a desired fan speed.

Figure 28. Fan Controller Component “Fans” Tab

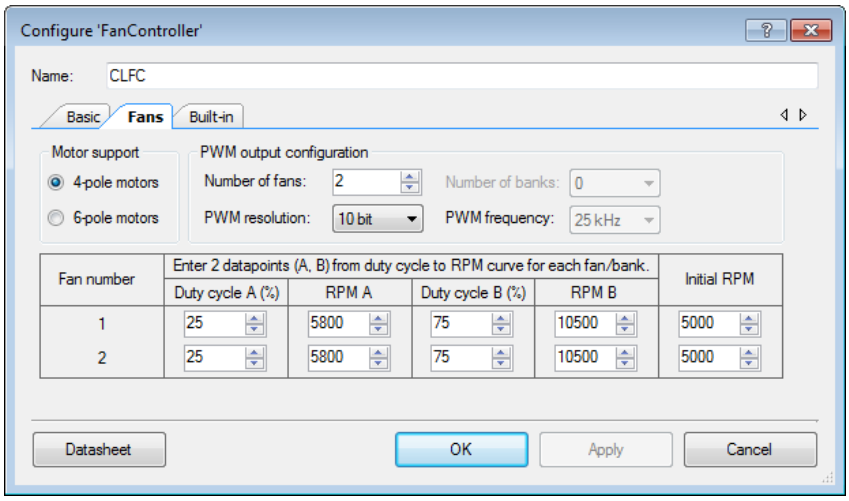


Figure 29 shows the configuration for the ADC, which measures the potentiometer voltage. The ±6.144 V range is chosen because the potentiometer voltage can vary from 0 to Vdda. To measure voltage near the supply rails, select the Bypass Buffer option for the buffer mode parameter.

Figure 29. ADC Configuration

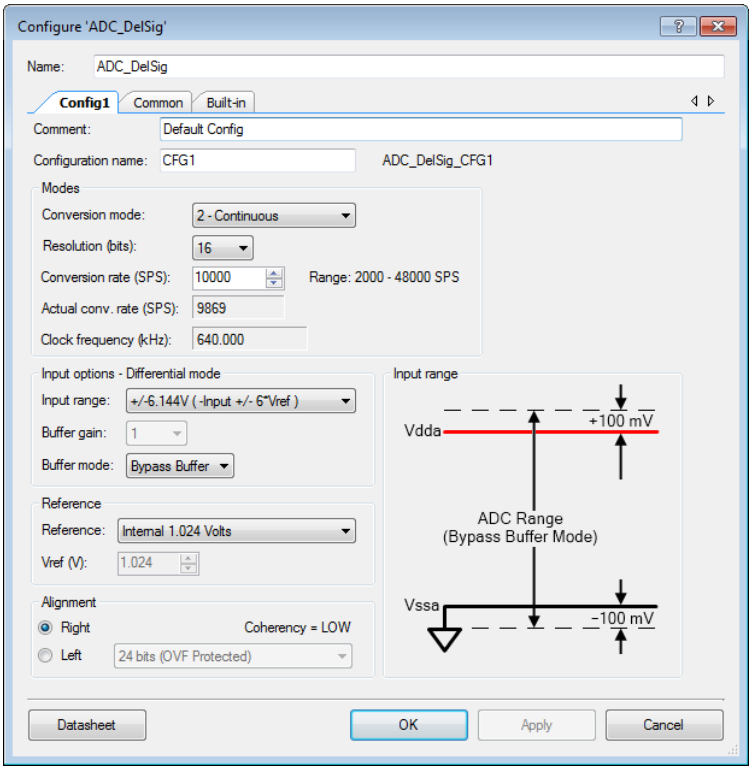
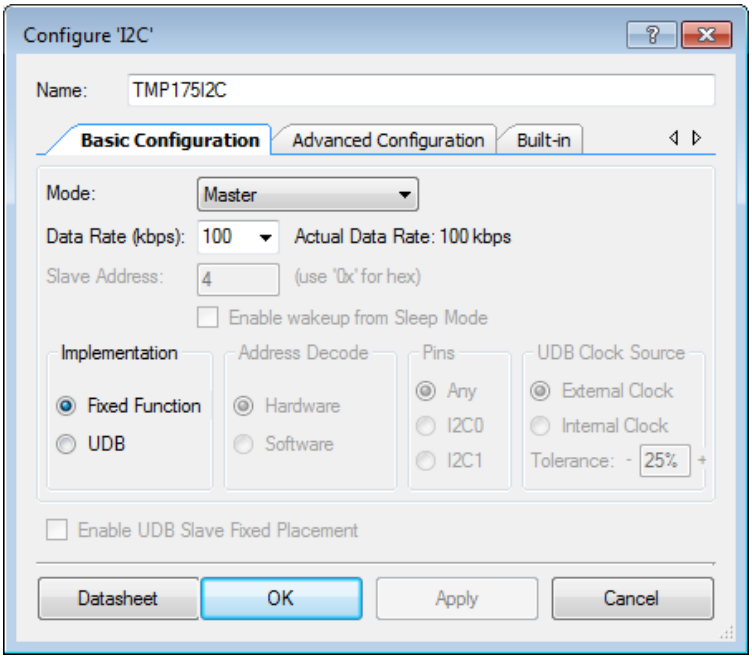


Figure 30 shows the configuration of the I²C master component, which measures the temperature of the TMP175 temperature sensor.

Figure 30. I²C Master Component Configuration



11.4 Firmware Structure

The thermal management firmware has been written in a modular format with different aspects of the functionality provided in separate source files and header files (see Table 3). This enables you to quickly understand the firmware structure and to modify the firmware easily to meet the application requirements.

Table 3. Source Files and Header Files in the Thermal Management Project

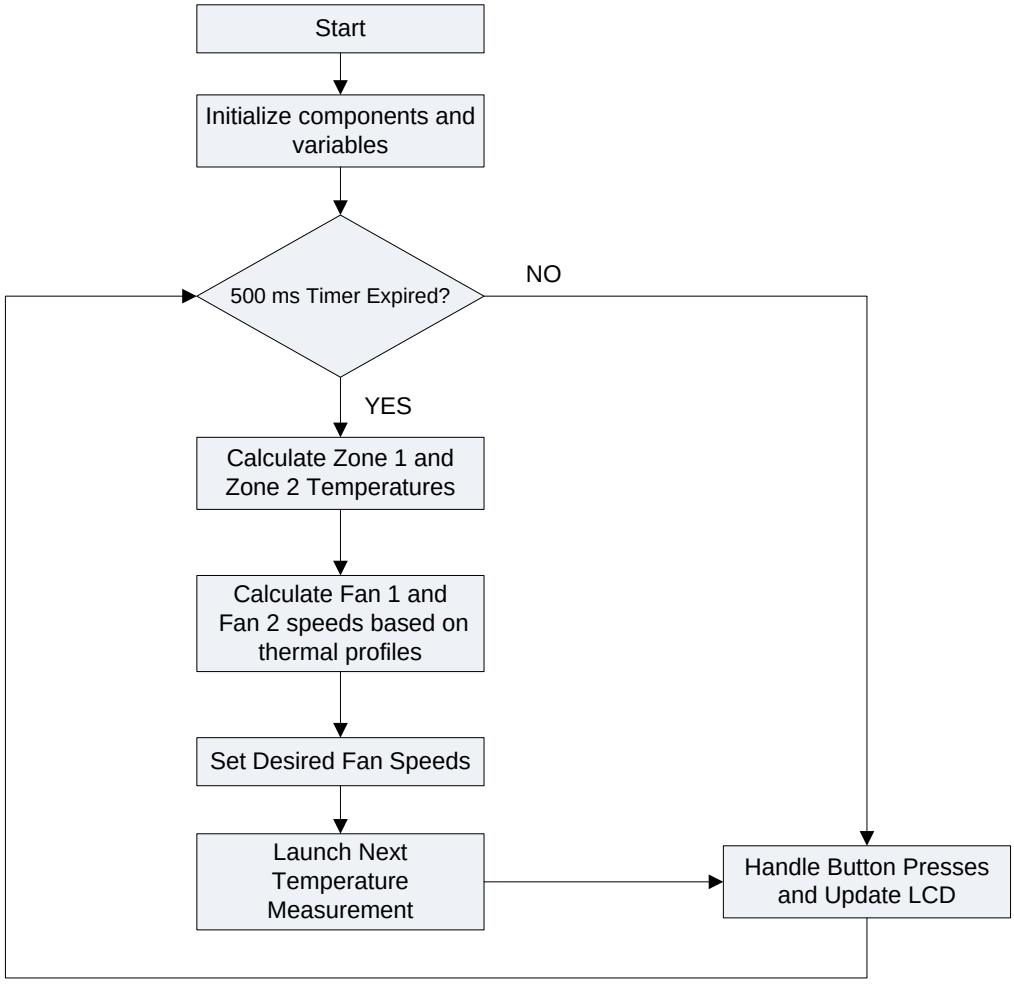
File Names	Purpose
ThermalManager.c, *ThermalManager.h*	These files contain the functions associated with thermal management. The *ThermalManager.c* source file contains definitions for thermal management functions such as getting individual sensor temperatures, using different zone temperature calculation algorithms, and setting the fan speed based on the zone temperature. The *ThermalManager.h* header file contains the macro definitions for setting the zone count, temperature sensor count, and so on. The header file also exposes the public function declarations that will be called from the `main()` function for implementing thermal management.
UserInterface.c, *UserInterface.h*	These files define functions associated with the project user interface. The *UserInterface.c* source file contains the function definitions for updating the LCD display based on the current zone state and handling button presses on the DVK. The *UserInterface.h* header file exposes the public function declarations that will be called from the `main()` function. The functions in these files are used for kit project demonstration purposes only and most likely will not be required in the end application.
main.c	This file defines the `main()` function that calls the thermal management and user interface functions continuously. The thermal management function is called periodically when a user-configurable timer period elapses.

11.5 Firmware Flowchart

Figure 31 shows the functional flow of the thermal management project firmware. Firmware flow is as follows:

- Initialize the thermal manager and user interface components on startup. The thermal manager components include the fan controller component and the components for interfacing with the temperature sensors (ADC and I^2C master). The user interface components include the character LCD and the interrupt service routine for handling button presses. The functions `ThermalManager_Start()` and `UserInterface_Start()` are called in *`main.c`* to initialize and start the components.
- A timer component, set to the required update period, services the thermal manager periodically. Modify the parameter `THERMAL_UPDATE_MS_RATE` in *`main.c`* to set the required update period in milliseconds.
- Whenever the timer expires, the function `ServiceThermalManager()` is called to calculate the zone temperature and to set the fan speed based on the zone temperature.
- The user interface, which consists of the character LCD and the push button, is serviced in the background in the main code by calling the function `ServiceUserInterface()`.

Figure 31. Thermal Management Firmware Functional Flow



11.6 Configuring the Zone Properties

All of the parameters that define the zone-temperature algorithm and the zone-temperature-to-fan-speed algorithm are defined at the top of the *ThermalManager.c* file. To modify these settings, refer to *ThermalManager.h* for the relevant keywords.

Configure the properties of the thermal zone by editing the zoneConfigEntry structure definition in the *ThermalManager.c* file, as shown in Figure 32. The comments to the right of the structure element in this figure are the structure element names, as declared in *ThermalManager.h*.

Figure 32. Configuring the Thermal Zone Properties in the *ThermalManager.c* file

ThermalManager.c

```
/* Zone Characteristics */
ZoneConfigEntry zoneConfig[ZONE_COUNT] =
    {
        /* Zone1 */
        {1,                                 /* fanCount           */
        0x0001,                             /* fanSelect          */
        FAN_SPEED_DEFAULT,                  /* fanCurrentSetSpeed */
        TEMP_SPEED_FUNC_LOOKUP_TABLE,       /* fanTempSpeedFunc   */
        4,                                  /* tempHysteresis     */
        2,                                  /* tempSensorCount    */
        0x0003,                             /* tempSensorSelect   */
        TEMP_COMPOSITE_WEIGHTED_AVG,        /* tempCompositeFunc  */
        {1,9,0,0,0,0,0,0,0,0}               /* tempSensorWeight   */
        },

        /* Zone2 */
        {1,                                 /* fanCount           */
        0x0002,                             /* fanSelect          */
        FAN_SPEED_DEFAULT,                  /* fanCurrentSetSpeed */
        TEMP_SPEED_FUNC_LOOKUP_TABLE,       /* fanTempSpeedFunc   */
        1,                                  /* tempHysteresis     */
        2,                                  /* tempSensorCount    */
        0x0003,                             /* tempSensorSelect   */
        TEMP_COMPOSITE_WEIGHTED_AVG,        /* tempCompositeFunc  */
        {99,1,0,0,0,0,0,0,0,0}              /* tempSensorWeight   */
        }
    };
```

Configure the following thermal zone properties using the structure variable `zoneConfig`, shown in Figure 32.

- The `fanCount` structure element specifies the number of fans in a thermal zone—in the current project, one. The maximum value of the number of fans in a thermal zone is determined by the definition `FAN_MAX` in *ThermalManager.h*.
- The `fanSelect` structure element is used as a bit mask value to select the fans assigned to a particular zone. Each bit corresponds to one fan. In the example project, Fan 1 is assigned to Zone 1 and Fan 2 is assigned to Zone 2.
- The `fanCurrentSetSpeed` structure element specifies the desired speed of the fans assigned to a particular zone. This variable will be continuously updated based on the current zone temperature and is initialized to a default value defined by `FAN_SPEED_DEFAULT` in the *ThermalManager.h* file.

- The `fanTempSpeedFunc` structure element specifies how the fan speed is computed from the zone temperature. Table 4 shows the available options.

Table 4. Options for Mapping the Zone Temperature to the Zone Fan Speed

Parameters for **fanTempSpeedFunc**	Meaning
`TEMP_SPEED_FUNC_LOOKUP_TABLE`	A table stored in flash maps the zone temperature to a corresponding fan speed. The structure `thermalLookup` defined in *`ThermalManager.c`* is used to store the values.
`TEMP_SPEED_FUNC_LINEAR`	Use this option to specify a linear relationship between the temperature and fan speed. Add your own code in the `CalcFanSpeed()` function of the *`ThermalManager.c`* file to define a linear relationship.
`TEMP_COMPOSITE_CUSTOM`	Use this option to specify a custom algorithm to map the zone temperature and fan speed. Add your own code in the `CalcFanSpeed()` function in *`ThermalManager.c`* file to define a custom algorithm.

The example project selects `TEMP_SPEED_FUNC_LOOKUP_TABLE`, which uses a table as follows:

- The `tempHysteresis` structure element specifies the hysteresis value in degrees Celsius. This value is applied when the temperature falls (see Figure 24 and Figure 25). The hysteresis setting ensures that the fan speed is not oscillating between two speeds when the temperature is near the transition points.
- The `tempSensorCount` structure element specifies the number of temperature sensors associated with a particular thermal zone. Its value should be less than the maximum count value as determined by the `TEMP_SENSOR_MAX` definition in *`ThermalManager.h`*.
- The `tempSensorSelect` structure element is used as a bit mask value to select the temperature sensors associated with a particular zone. Each bit corresponds to one temperature sensor.
- The `tempCompositeFunc` structure element determines the algorithm for calculating the zone temperature based on the individual sensor temperatures. Table 5 shows the available options.
- The `tempSensorWeight` structure element specifies the weighting factors assigned to the individual temperature sensors for calculating the zone temperature. For the example project, the weights are assigned based on the following formulae:

$$Zone\ 1\ Temperature, Z1 = \frac{1}{10}\ T1\ +\ \frac{9}{10}\ T2$$

$$Zone\ 2\ Temperature, Z2 = \frac{99}{100}\ T1\ +\ \frac{1}{100}\ T2$$

Table 5. Options for Zone Temperature Calculation Algorithm

Parameters for **tempCompositeFunc**	Meaning
`TEMP_COMPOSITE_STRAIGHT_AVG`	The average of the individual sensor temperatures associated with the zone gives the zone temperature.
`TEMP_COMPOSITE_WEIGHTED_AVG`	The weighted average of the individual sensor temperatures associated with the zone gives the zone temperature. The weighting factors are specified by the structure element tempSensorWeight. The example project uses this option.
`TEMP_COMPOSITE_MAX_TEMP`	The maximum value of the individual sensor temperatures associated with the zone gives the zone temperature.
`TEMP_COMPOSITE_CUSTOM`	The user can define a custom algorithm as applicable in the GetZoneTemp() function. The example project has a placeholder for the user code to implement the custom algorithm as applicable.

In the example project, the zone temperature is represented as an integer variable. The temperature calculations in the example project use integer arithmetic to reduce the execution time.

The example project uses the look-up table method mentioned in Table 4 to map the zone temperature to the zone fan speed. Figure 33 shows the look-up table used in this example project. The table has been created based on the thermal profiles for the two zones, as shown in Figure 24 and Figure 25. Modify the look-up table as your application requires.

Figure 33. Zone Temperature: Fan Speed Look-up Table in the *ThermalManager.c* file

ThermalManager.c

```
49
50  /* Zone Temperature - Fan speed Look up table in flash */
51  CYCODE const ThermalLookupTableEntry thermalLookup[ZONE_COUNT][MAX_LOOKUP_TABLE_ENTRIES] =
52      {
53          /* Zone1 Temp/Speed Lookup */
54          {
55          {0, 4000},
56          {15,4500},
57          {35,5500},
58          {55,7500},
59          {75,9500},
60          },
61
62          /* Zone2 Temp/Speed Lookup */
63          {
64          {0, 4500},
65          {23,5000},
66          {25,7000},
67          {27,9000},
68          {29,10000}
69          }
70      };
71
```

11.7 Thermal Management Functions

Table 6 lists and describes several functions that APIs use for thermal management. Refer to the source code in *ThermalManager.c* for implementation details on these functions.

Table 6. Thermal Management Functions in *ThermalManager.c*

Function	Description
`ThermalManager_Start()`	This function starts the fan controller component and the components associated with the temperature sensors, such as the I²C master and the ADC. It also calls the `LaunchTempMeasure()` function to get the initial sensor temperatures and the zone temperatures. This function should be called once in the startup code in main to initialize and start the components.
`ServiceThermalManager()`	This function sets the desired fan speed based on the current zone temperature. It also calls the `LaunchTempMeasure()` function to update the zone temperature. This function should be called periodically from the main code to do thermal management.
`GetZoneTemp()`	This function returns the temperature of a zone from the individual sensor temperatures associated with the zone. Use the appropriate zone temperature algorithm (depending on the zone property configuration) according to Table 5.
`CalcFanSpeed()`	This function calculates the desired fan speed for a zone based on the zone temperature. Use the appropriate fan speed calculation algorithm (depending on the zone property configuration) according to Table 4.
`SetFanSpeed()`	This function updates the desired speed of the fans associated with a zone. Use this function after calling the `CalcFanSpeed()` function.
`LaunchTempMeasure()`	This function calculates the individual sensor temperatures. The example project calls only the I²C temperature sensor and potentiometer temperature sensor functions. Add function calls for additional temperature sensors as required in your application.
`GetSensorTemp()`	This function returns the individual sensor temperature.
`GetFanCurrentSetSpeed()`	This function returns the current desired fan speed for a zone.

Function	Description
`CalcWeightedAverageZoneTemp()`	This function returns the zone temperature calculated based on the weighted average of the sensor temperatures.
`CalcAverageZoneTemp()`	This function returns the zone temperature calculated based on the average of the sensor temperatures.
`CalcMaxTempZoneTemp()`	This function returns the zone temperature as the maximum temperature among the sensor temperatures.
`TemperatureToSpeedLookupTable()`	This function returns the desired fan speed for a zone based on the look-up table stored in the memory. Hysteresis is applied for calculating the fan speed when the zone temperature is decreasing.
`GetI2CTemp()`	This function returns the current temperature of the TMP175 temperature sensor.
`GetPotentiometerTemp()`	This function returns the current temperature of the potentiometer that is used to simulate an analog temperature sensor.

11.8 Testing the Project

In this project, the fan speed is controlled based on the temperature of the corresponding thermal zone. To test the closed-loop speed control of Fan 1, vary the potentiometer on the DVK. This simulates the condition of changing the corresponding zone temperature in the project. The speed of Fan 1 is adjusted according to the temperature-versus-speed graph shown in Figure 24. To test the closed-loop speed control of Fan 2, touch the I^2C temperature sensor (U1) on the CY8CKIT-036 EBK. This results in variations in the measured I^2C-based sensor temperature. The speed of Fan 2 is adjusted according to the graph shown in Figure 25.

On reset, the LCD displays the characteristics of Zone 1 after printing a Welcome message. These characteristics include the zone temperature (Z1), the temperature calculation algorithm, the desired fan speed (F1), and the actual fan speed (A1). The fan speeds will match the corresponding zone thermal profiles. Note that when the temperature decreases, the hysteresis factor shown in the thermal zone profiles will determine the reduction in fan speed at the transition points. When the fan speed is zero, the message “Fault” will be printed in place of the actual speed.

Figure 34, Figure 35, and Figure 36 show the format of the LCD display for different menu settings.

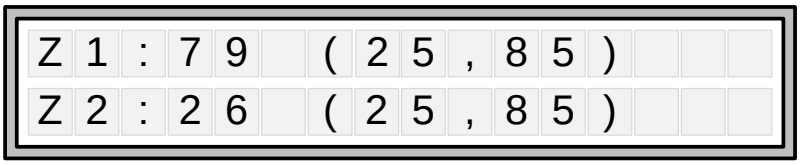
Figure 34. Zone 1 Summary

Press the push button (SW1 on CY8CKIT-001 or SW2 on CY8CKIT-030) once to view the characteristics of Zone 2.

Figure 35. Zone 2 Summary

Press the push button again to view the individual sensor temperatures for each zone and the weighted composite zone temperature. On each line of the LCD display, the individual sensor temperatures are displayed after the composite zone temperature and are enclosed within parentheses, as Figure 36 shows. The first sensor temperature within parentheses is the TMP175 I^2C temperature sensor temperature followed by the temperature of the simulated analog temperature sensor (potentiometer).

Figure 36. Temperature Sensor Summary



Pressing the push button again will repeat the previously mentioned display sequence.

12 Summary

Working through all of the example projects will lead to a good understanding of the various operating modes of the fan controller component and the associated APIs. The component enables designers to quickly and easily begin working on thermal management applications with minimal firmware development required.

The closed-loop hardware control mode provides many benefits, including reduced firmware development time and optimal fan speeds, reducing acoustic noise and power consumption. The custom logic available within the PSoC architecture provides immense value in these applications, enabling designers to build complex control systems in hardware, freeing up the CPU to handle other tasks, such as protocol handling or other critical system management tasks.

The unique ability of the PSoC architecture to combine custom digital logic, analog signal chain processing, and an MCU in a single device enables system designers to integrate many external fixed-function ASSPs. This powerful integration capability not only reduces BOM cost, but also results in PCB board layouts that are less congested and more reliable.

Document History

Document Title: AN66627 - PSoC® 3 and PSoC 5LP Intelligent Fan Controller

Document Number: 001-66627

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3159398	JK	02/10/2011	New Application Note.
*A	3265851	JK	06/30/2011	Updated Document Title to read as “PSoC® 3 and PSoC 5 – Intelligent Fan Controller – AN66627”. Updated attached example project (to use Fan Controller Component v1.10).
*B	3550058	JK	03/13/2012	Updated Software Version as “PSoC Creator™ v2.0” in page 1. Removed “Going Forward”. Added “PSoC Thermal Management EBK”. Updated attached example project (for use with PSoC Creator v2.0). Updated to new template.
*C	3576608	JK	04/10/2012	Updated attached example project (to use the new Fan Controller v1.21 component to address a defect found in the SetDesiredSpeed() API). Updated to new template.
*D	3676155	JK	07/13/2012	Updated Software Version as “PSoC Creator™ v2.1” in page 1. Updated Getting Started with the Fan Controller Component: Updated figures “FanController Customizer – Basic Tab”, “FanController Customizer – Fans Tab”. Updated Example 3 – Hardware Speed Control: Updated figure “Configuring Closed Loop Control in the FanController Component Customizer”. Updated Example 4 – Advanced Hardware Speed Control: Updated figure “FanController Component Customizer Settings for Example Project 4”. Updated attached example project (to use the production Fan Controller component v2.10).
*E	3819555	JK	11/22/2012	Updated Software Version as “PSoC Creator™ v2.1 SP1” in page 1. Updated attached example project (to use Fan Controller v2.20 released through PSoC Creator v2.1 SP1; also changed example project pin assignments to work with the CY8CKIT-036 PSoC Thermal Management Expansion Board Kit (EBK) connected to either the CY8CKIT-030 PSoC 3 Development Kit (DVK) or CY8CKIT-050 PSoC 5LP DVK).
*F	4209587	VVSK	12/3/2013	Updated Software Version as “PSoC Creator™ v3.0 CP7 or Higher”. Removed “Fan Cables and Connectors”. Added “Cypress Development Kits for Thermal Management”. Removed “PSoC Thermal Management EBK”. Added “Example 6: Thermal Management”.
*G	4274640	VVSK	02/11/2014	No technical updates. Completing Sunset Review.
*H	4660967	VVSK	02/13/2015	Updated Software Version as “PSoC Creator™ v3.0 SP2 or higher” in page 1. Updated Fan Speed Control: Updated Figure 4; and also updated description. Updated Cypress Development Kits for Thermal Management: Updated Table 2 (Updated details in “Hardware Requirement” column). Updated attached example project. Completing Sunset Review.

*I	5080454	VVSK	01/11/2016	Updated Software Version as “PSoC Creator™ 3.3 CP1 or higher” in page 1. Added Related Application Notes in page 1. Updated attached example project (to PSoC Creator 3.3 CP1). Completing Sunset Review.
*J	5705324	AESATMP9	04/21/2017	Updated logo and copyright.
*K	6433775	SNVN	01/08/2019	Updated to new template. Completing Sunset Review.

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 MCU

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